Python Team Project Report

Fabricio Giusti Oliveira Monteiro, Timothy Jiang, Raunak Dani, Rohan Shah

LC5: Team 13

ENGR 13300

Prof. Harrison-Smith

October 16, 2024

1. Project Motivation

Image processing and cipher techniques are used in various engineering disciplines. One such discipline is biomedical engineering, where these methods are used to perform tasks such as image quality enhancement and data security.

Due to the urgent nature of the medical field, crucial images such as MRI scans, x-rays and ultrasounds often excessive noise due to factors such as low light or fast acquisition times. Techniques like Gaussian filtering and median filtering help reduce noise while preserving important features (Gonzalez & Woods, 2018). For contrast enhancements, methods such as histogram equalization are used to highlight specific structures, improving diagnostic accuracy.

Cypher techniques play a large role in data security for the medical field. The Advanced Encryption Standard (AES) is widely used to protect medical images from unauthorized access (Srinivasan & Rajan, 2020). This helps medical agencies comply with privacy regulations such as HIPAA, which mandates the protection of patient information.

2. Project Overview and Methods

Steganography is a technique that has been used for a long time - specifically in the use case of hiding messages within another medium, in the case of our entire project, this other medium was through pictures. This technique of hiding a secret code/message within an image so that it still looks the same visually, but contains lots of data is a method using the least significant bit or LSB. The LSB method, as the name suggests, changes the least significant bit of the pixel values in the image, so that a secret code can be incorporated within the image. This method works, because the least significant bit (the one that contributes the least to the image to the value of a byte and the color of the pixel) alters the image minimally, and not enough for one to visually see. While there

are other methods of encrypting text into images, the LSB method is one of the most classic and easiest methods for steganography.

Now, within our entire project, we used several different algorithms using the LSB method to encode a message into an image. These different methods included the XOR, Vigenère, and Caesar ciphers. Starting with the XOR cipher, this cipher works by comparing the bits between the plaintext and the key. If the individual bits vary, then this cipher outputs 1, otherwise if the individual bits are the same, this cipher outputs a 0. Next, the Vigenère cipher works by shifting each letter in the plain text by the numerical value of the of each letter in the key. The key repeats itself to match the length of the plaintext, and allows each character within the plaintext to be shifted a different amount compared to the surrounding letters in the plaintext, as the shift is dependent on the key itself. Finally, the Caesar cipher works by shifting all letters in the plaintext by the same amount. While a Vigenère cipher shifts every single character a different amount, the Caesar cipher shifts all characters the same amount.

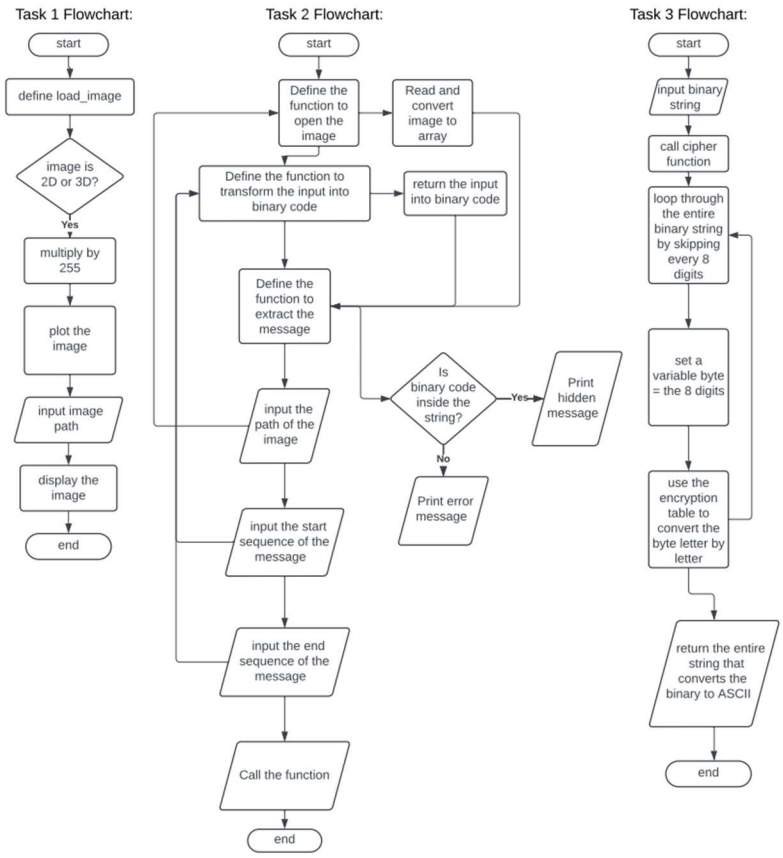
For encoding these ciphers, our group used several resources. Of these resources, the most helpful were stackoverflow and docs.python.org. We were able to extract a lot of information from these resources, but we also used other resources such as GeeksforGeeks. Due to our groups lack of knowledge specifically in python, let alone ciphers, these websites helped us familiarize ourselves with how ciphers work, and how we could learn to code our xor, vigenere, and caesar ciphers.

3. Discussion of Algorithm Design

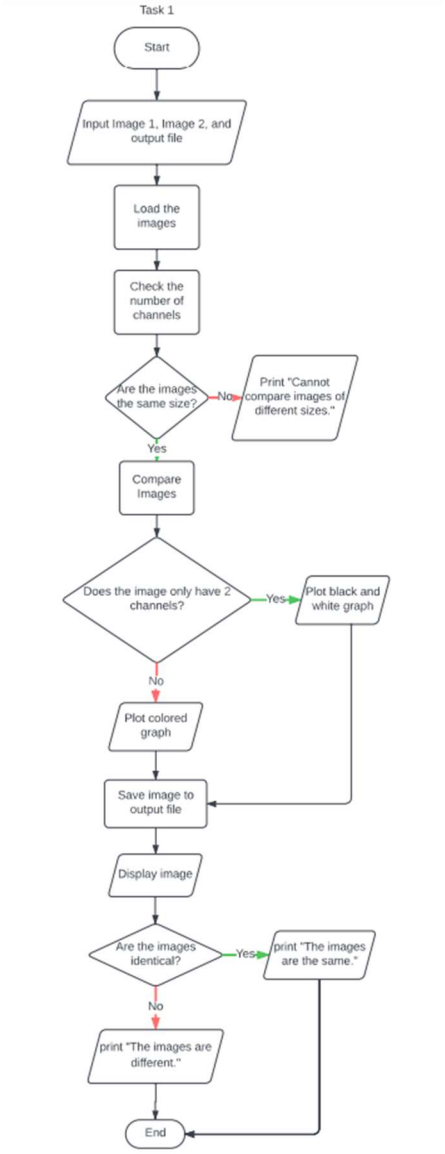
The design rationale for modularizing the code in the steganography program is to ensure flexibility, reusability, and ease of adaptation for future users. By breaking the program into distinct modules, such as encryption, message encoding, and image handling, each function can

be independently modified or replaced without affecting the overall system. This allows users to easily integrate new encryption techniques or image formats as needed. Future users can encode their own messages by simply selecting an image, inputting a message, and using the existing algorithms to encrypt and embed the message. The modular structure streamlines the process, enabling users to customize the encoding or decoding process while maintaining clarity and simplicity in the code. The algorithm begins by accepting an image file and a message from the user. The message is then encrypted using a chosen cipher, such as Caesar, XOR, or Vigenere. After encryption, the program encodes the message by converting both the encrypted text and the pixel data into binary format. It then modifies the least significant bits (LSB) of selected pixels in the image to embed the binary representation of the encrypted message. Once the encoding is complete, the program saves and displays the modified image with the hidden message. To retrieve the message, the decoding process extracts the binary data from the image's pixel values, converts it back into text, and decrypts it to reveal the original message.

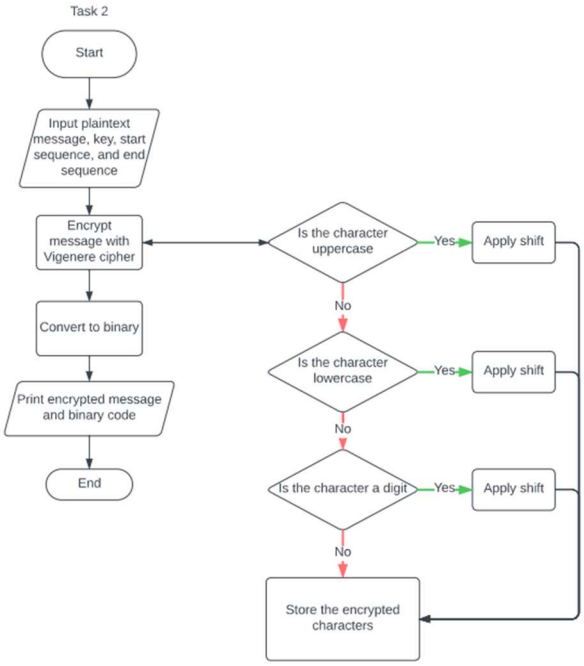
Checkpoint 1 Flowchart:



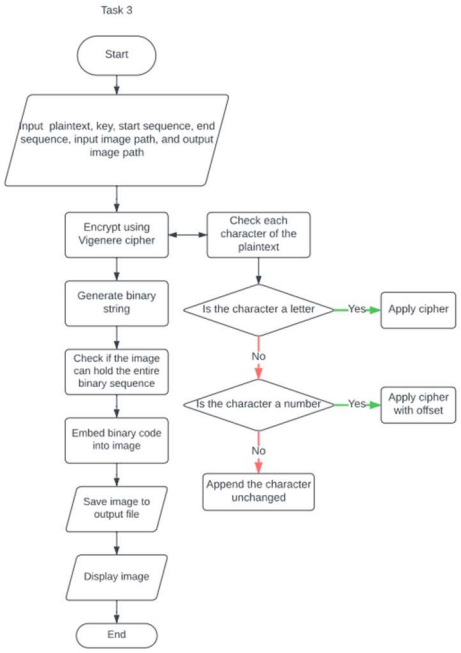
Checkpoint 2 Task 1 Flowchart:



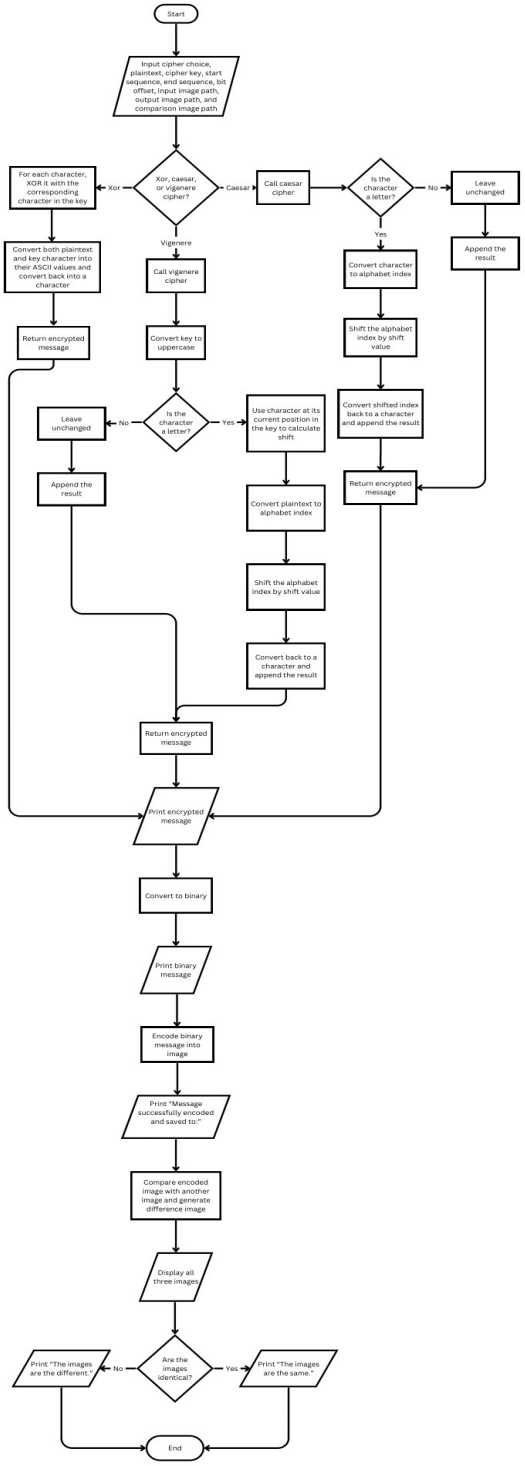
Checkpoint 2 Task 2 Flowchart:



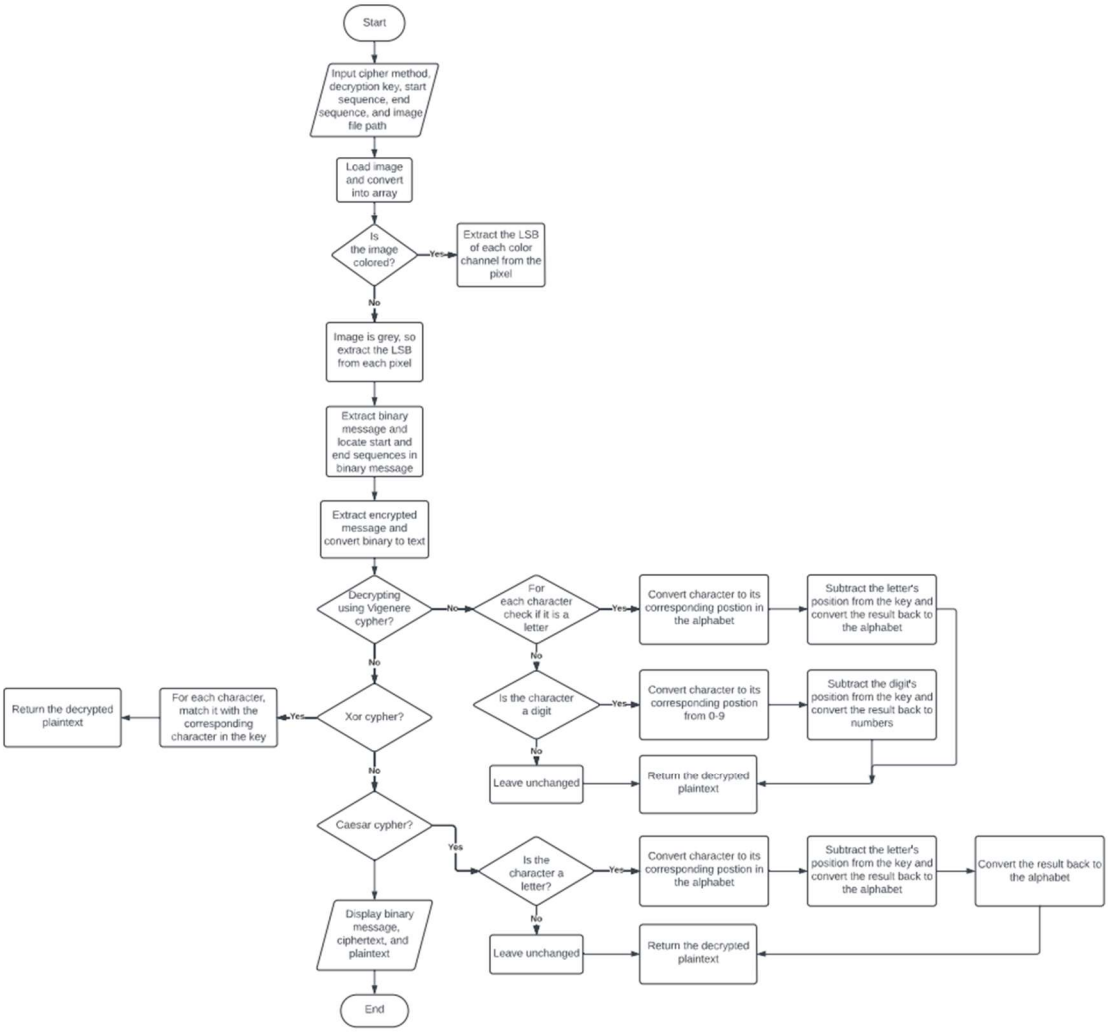
Checkpoint 2 Task 3 Flowchart:



Demo Task 1 Flowchart:



Demo Task 2 Flowchart:



4. References

GeeksforGeeks. *GeeksforGeeks*, https://www.geeksforgeeks.org/.

Gonzalez, Rafael C., and Richard E. Woods. Digital Image Processing. Pearson, 2018.

Litjens, Geert, et al. "A Survey on Deep Learning in Medical Image Analysis." Medical Image Analysis, vol. 42, 2017, pp. 60-88.

Python Software Foundation. *Python Documentation*, Python Software Foundation, https://docs.python.org/.

Srinivasan, S., and R. Rajan. "Securing Medical Images Using AES Encryption." Journal of Medical Systems, vol. 44, no. 2, 2020, Article 25.

Stack Overflow. *Stack Overflow*, Stack Exchange Inc., https://stackoverflow.com/

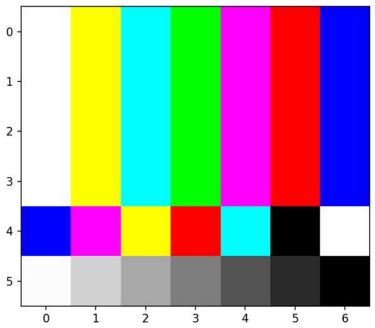
5. Appendix

1. User manual

The operation of the programs is simple. Firstly, the user must have all the required images and files in the same folder/directory. Not doing so will return an error message when running the code. When ready, all the user needs to do is run the code, and depending on the situation, input the required information when prompted. Users should be aware of capitalization in their inputs, as incorrect capitalization might return an error or false results. Below are some sample input and outputs for reference.

Checkpoint 1 Task 1 Sample Input/Output:

```
Enter the path of the image you want to load: ref_col.png
```



Checkpoint 1 Task 2 Sample Input/Output:

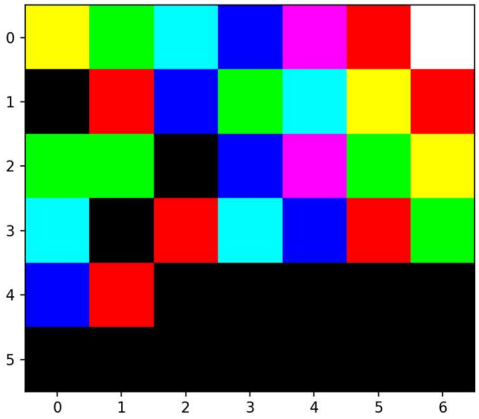
```
Enter the path of the image you want to load: ref_col_p.png
Enter the start sequence: 007
Enter the end sequence: 700
Extracted Message:  0100110001010101010000110100101101011001
```

Checkpoint 1 Task 3 Sample Input/Output:

```
Enter the binary message: 01001000011001010110110001101100011011111
Converted text: Hello
```

Checkpoint 2 Task 1 Sample Input/Output:

```
Enter the path of your first image: ref_col.png
Enter the path of your second image: ref_col_e.png
Enter the path for the output image: diff.png
```

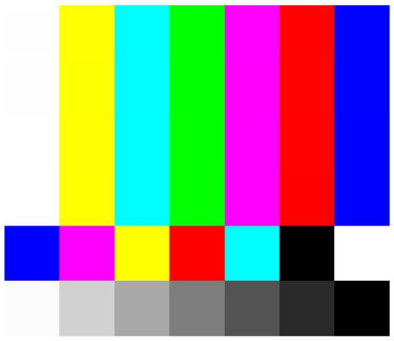


Checkpoint 2 Task 2 Sample Input/Output:

```
Enter the plaintext message: hi
Enter the key: secret
Enter the start sequence: 0
Enter the end sequence: 1
Encrypted Message using Vigenere Cipher: zm
Binary output message: 00110000 01111010 01101101 00110001
```

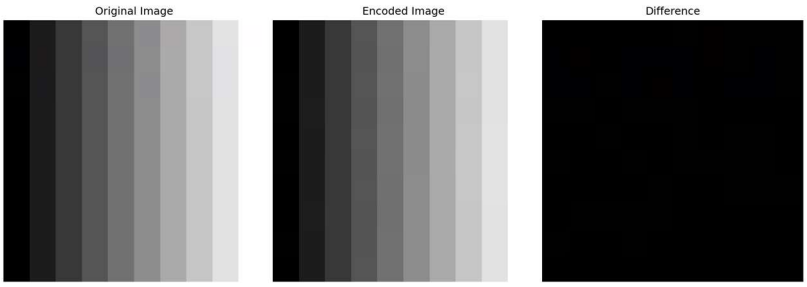
Checkpoint 2 Task 3 Sample Input/Output:

```
Enter the plaintext you want to encrypt: hello
Enter the key for Vigenere ciphertxt: Beau
Enter the start sequence: 007
Enter the end sequence: 700
Enter the path of the image: ref_col.png
Enter the path for the encoded image: ref_col_v.png
Encrypted Message using Vigenere Cipher:  iilfp
Binary output message:  00110000 00110000 00110111 011010
10111 00110000 00110000
Message successfully encoded and saved to:  ref_col_v.png
```



Demo Task 1 Sample Input/Output:

```
Enter the cipher you want to use for encryption: xor
Enter the plaintext you want to encrypt: Sunny
Enter the key for the cipher: 3RR0R
Enter the start sequence: 12
Enter the end sequence: 21
Enter the bit offset before you want to start encoding: 13
Enter the path of the input image: ref_gry.png
Enter the path for your encoded image: gry_x.png
Enter the path of the image you want to compare: ref_gry_x.png
Encrypted Message using Xor Cipher: `'<^+
Binary output message: 00110001001100100110000000100111001111000101111000101011001100100011100
Message successfully encoded and saved to: gry_x.png
The images are different.
```



Demo Task 2 Sample Input/Output:

```
Enter the cipher you want to use for encryption: xor
Enter the key for the cipher: 3RR0R
Enter the start sequence: 12
Enter the end sequence: 21
Enter the path of the input image: ref_gry_x.png
Extracted Binary Message:  01100000000100111001111000101111000101011
Converted Binary Text: `'<^+
Converted text: Sunny
```

2. **Project management plan** *(*Team members' contribution*)*

Our team used WhatsApp as our primary mode of communication. Additionally, when files of code needed to be shared remotely, did that via emailed ZIP file. We feel that this was effective as we observed reasonably fast response times from all members of the group. The group's responsiveness to one another online helped us be efficient and was essential to successful completion of the project.

Fabricio's role in our team was troubleshooting and optimization. His ability to identify errors and resolve them made him an essential member. He did this by systematically seeking out the root cause of the issues, and then used processes such as trial and error to fix them. In addition, he helped redesign the code to make it more efficient where possible.

Timothy's role in our team was to organize logic and search for methods of completing the task. In charge of flowchart creation, his strengths in mapping out the task at hand and determining a plan to complete it helped our group maximize efficiency when writing the code. Within the code, he worked on adding comments to each line for maximal clarity.

Raunak's role in our team was the bulk drafting the code. He would produce a workable draft of each assignment, and then collaborate with the team members to improve it. His ability to produce solid initial drafts and facilitate productive teamwork was crucial to our success.

Rohan's role in our team was organization and project management. He would download and create all the necessary files and organize them for submission. He would also plan team meetings as necessary. Within the code, he worked on testing the sample cases, commenting and troubleshooting errors to prepare for the demonstration.

In the future, we can take more time out of class to gain a deeper understanding of the material, so we are better prepared. We can do this by watching and studying the provided class

materials ahead of time, as well as consuming supplementary information from tools such as Stack Overflow.

3. **Discussion of design process** *(Approach to the design process)*

In Task 1, we defined the main requirements: implementing multiple encryption methods (Vigenère, XOR, and Caesar ciphers), embedding encrypted binary messages into images at a specific bit offset, and ensuring the accuracy of encryption. After analyzing various encryption methods, we synthesized this knowledge to create encryption functions for each cipher and developed an encoding function to embed messages in images. We generated multiple solutions for embedding data, evaluated them through testing, and ensured that the system could handle flexible user inputs, such as encryption key, message start and end sequences, and bit offset. We also incorporated image comparison to verify whether the modified image differed from the original.

In Task 2, we focused on decrypting and extracting messages from images. We began by updating our binary extraction function to handle messages starting at random bit offsets. Then, we implemented decryption functions for each cipher and ensured the correct decryption of ciphertext back into plaintext. Our main function collected necessary inputs, extracted the hidden message, decrypted it, and compared the extracted ciphertext with the original image to verify accuracy. We evaluated the success of each step by checking if the right binary message was extracted, if the binary correctly converted to ciphertext, and if the decryption returned the correct plaintext. Throughout both tasks, we considered risks such as message extraction failure. This iterative process allowed us to refine each step, ensuring the project met all the design and functional requirements.

4. **Code**

Checkpoint 1 Task 1 Code:

```
import matplotlib.pyplot as plt

import numpy as np



#turning image into array

def load_image(file_name):

   image = plt.imread(file_name)

   #image = Image.open(file_name)

   image_array = np.array(image)



   if image_array.dtype == 2 or image_array.dtype == 3:

      image_array = (image_array * 255).astype(np.uint8)

   return image_array

#outputting image

def display_image(image_array):

   plt.imshow(image_array, cmap='gray' if image_array.ndim == 2 else None)

   plt.axis()

   plt.show()

#user input the reference file
```

```
def main():
    file_name = input("Enter the path of the image you want to load: ")
    image_array = load_image(file_name)
    display_image(image_array)


if __name__ == "__main__":
    main()
```

Checkpoint 1 Task 2 Code:

```
import numpy as np
import matplotlib.pyplot as plt


def image(file_name):
    #Reusing the same code from Task 1 to read the image and convert it to an array
    image = plt.imread(file_name)
    image_array = np.array(image)
    if image_array.dtype != np.uint8:
        image_array = (image_array * 255).astype(np.uint8)
    #New code that indexes the 3D array to extract the last bit of every bite
```

```
    total_binary = []
    #Accounting for both 2D and 3D arrays depending on if the image is colored or greyscale
    if len(image_array.shape) == 3:
        #Index 3 total times to get the 8th bit of colored image
        for i in range(image_array.shape[0]):
            for j in range(image_array.shape[1]):
                for k in range(image_array.shape[2]):
                    total_binary.append(image_array[i, j, k] & 1)
    elif len(image_array.shape) == 2:
        #Index 2 total times to get the 8th bit of greyscale image
        for i in range(image_array.shape[0]):
            for j in range(image_array.shape[1]):
                total_binary.append(image_array[i, j] & 1)


    return total_binary


#Converting the user input to binary
def convert_input(input_text):
```

```
    #Found on stack overflow https://stackoverflow.com/questions/18815820/how-to-convert-string-to-binary
    input_bi = ''.join(format(ord(i), '08b') for i in input_text)
    return input_bi


def extract(file_name, start, end):
    image_binary = image(file_name)
    #Converts each element of binary_data to a string and then concentrates them into a single binary string
    binary_string = ''.join(map(str, image_binary))


    #find the start binary code inside the string
    start_index = binary_string.find(start)
    #find the end binary code inside the string
    end_index = binary_string.find(end, start_index + len(start))
    #if the start and end binary code are inside the string
    if start_index != -1 and end_index != -1:
        start_index += len(start)
        hidden_message = binary_string[start_index:end_index]
```

```
            print("Extracted Message: ", hidden_message)
        #if it cannot find the start or the end point inside the string
        else:
            print("Start or end sequence not found in image.")


    def main():
        #Gets the user inputs to use in the code
        image_path = input("Enter the path of the image you want to load: ")
        start = input("Enter the start sequence: ")
        end = input("Enter the end sequence: ")
        #Converting start and end sequence to binary
        start_bi = convert_input(start)
        end_bi = convert_input(end)
        extract(image_path, start_bi, end_bi)
    #Ending code
    if __name__ == "__main__":
        main()
```

Checkpoint 1 Task 3 Code:

```
def cipher(full_string):
   #uses a conversion table similar to the ashbash cipher used in Py4 individual task 2
   convert_table = {
      '01000001': 'A', '01000010': 'B', '01000011': 'C', '01000100': 'D', '01000101': 'E', '01000110': 'F',
'01000111': 'G', '01001000': 'H',
      '01001001': 'I', '01001010': 'J', '01001011': 'K', '01001100': 'L', '01001101': 'M', '01001110': 'N',
'01001111': 'O', '01010000': 'P',
      '01010001': 'Q', '01010010': 'R', '01010011': 'S', '01010100': 'T', '01010101': 'U', '01010110': 'V',
'01010111': 'W', '01011000': 'X',
      '01011001': 'Y', '01011010': 'Z', '01100001': 'a', '01100010': 'b', '01100011': 'c', '01100100': 'd',
'01100101': 'e', '01100110': 'f',
      '01100111': 'g', '01101000': 'h', '01101001': 'i', '01101010': 'j', '01101011': 'k', '01101100': 'l',
'01101101': 'm', '01101110': 'n',
      '01101111': 'o', '01110000': 'p', '01110001': 'q', '01110010': 'r', '01110011': 's', '01110100': 't',
'01110101': 'u', '01110110': 'v',
      '01110111': 'w', '01111000': 'x', '01111001': 'y', '01111010': 'z', '00100000': ' ', '00101110': '.',
      '00110000': '0', '00110001': '1', '00110010': '2', '00110011': '3', '00110100': '4', '00110101': '5',
      '00110110': '6', '00110111': '7', '00111000':'8', '00111001':'9', '00100001':'!'
   }
```

```
    #creates an empty string that stores the new word letter by letter
    encrypted = ''
    #iterates through the inputted binary, splits it into segments of 8, then uses the conversion table to evaluate binary to words, letter by letter
    for i in range(0, len(full_string), 8):
        byte = full_string[i:i+8]
        encrypted += convert_table.get(byte)
    return encrypted


def main():
    #asks for the binary input
    binary_message = input("Enter the binary message: ")
    #calls the cipher function to convert the binary input into a word
    print(f"Converted text: {cipher(binary_message)}")


if __name__ == "__main__":
    main()
```

Checkpoint 1 Task 4 Code:

```
#Importing the required libraries

import numpy as np

import matplotlib.pyplot as plt



def image(file_name):

   #Reusing the same code from Task 1 to read the image and convert it to an array

   image = plt.imread(file_name)

   image_array = np.array(image)

   if image_array.dtype != np.uint8:

      image_array = (image_array * 255).astype(np.uint8)

   #New code that indexes the 3D array to extract the last bit of every bite

   total_binary = []

   #Accounting for both 2D and 3D arrays depending on if the image is colored or greyscale

   if len(image_array.shape) == 3:

      #Index 3 total times to get the 8th bit of colored image

      for i in range(image_array.shape[0]):

         for j in range(image_array.shape[1]):

            for k in range(image_array.shape[2]):
```

```
                total_binary.append(image_array[i, j, k] & 1)
    elif len(image_array.shape) == 2:
        #Index 2 total times to get the 8th bit of greyscale image
        for i in range(image_array.shape[0]):
            for j in range(image_array.shape[1]):
                total_binary.append(image_array[i, j] & 1)

    return total_binary

#Converting the user input to binary
def convert_input(input_text):
    #Found on stack overflow https://stackoverflow.com/questions/18815820/how-to-convert-string-to-binary
    input_bi = ''.join(format(ord(i), '08b') for i in input_text)
    return input_bi
   #input_int = int(input_text)
    #input_bi = bin(input_int)[2:]
    #return input_bi
```

```
def extract(file_name, start, end):
    image_binary = image(file_name)
    #Converts each element of binary_data to a string and then concentrates them into a single binary string
    binary_string = ''.join(map(str, image_binary))

    #find the start binary code inside the string
    start_index = binary_string.find(start)
    #find the end binary code inside the string
    end_index = binary_string.find(end, start_index + len(start))
    #if the start and end binary code are inside the string
    print(f"Below is the img_array output of {file_name}:")
    if start_index != -1 and end_index != -1:
        start_index += len(start)
        hidden_message = binary_string[start_index:end_index]
        print("Extracted Message:", hidden_message)
    #if it cannot find the start or the end point inside the string
```

```
    else:
        return "Start or end sequence not found in image."
    return hidden_message


def cipher(full_string):
    #uses a conversion table similar to the ashbash cipher used in Py4 individual task 2
    convert_table = {
        '01000001': 'A', '01000010': 'B', '01000011': 'C', '01000100': 'D', '01000101': 'E', '01000110': 'F',
'01000111': 'G', '01001000': 'H',
        '01001001': 'I', '01001010': 'J', '01001011': 'K', '01001100': 'L', '01001101': 'M', '01001110': 'N',
'01001111': 'O', '01010000': 'P',
        '01010001': 'Q', '01010010': 'R', '01010011': 'S', '01010100': 'T', '01010101': 'U', '01010110': 'V',
'01010111': 'W', '01011000': 'X',
        '01011001': 'Y', '01011010': 'Z', '01100001': 'a', '01100010': 'b', '01100011': 'c', '01100100': 'd',
'01100101': 'e', '01100110': 'f',
        '01100111': 'g', '01101000': 'h', '01101001': 'i', '01101010': 'j', '01101011': 'k', '01101100': 'l',
'01101101': 'm', '01101110': 'n',
        '01101111': 'o', '01110000': 'p', '01110001': 'q', '01110010': 'r', '01110011': 's', '01110100': 't',
'01110101': 'u', '01110110': 'v',
        '01110111': 'w', '01111000': 'x', '01111001': 'y', '01111010': 'z', '00100000': ' ', '00101110': '.',
```

```
        '00110000': '0', '00110001': '1', '00110010': '2', '00110011': '3', '00110100': '4', '00110101': '5',
        '00110110': '6', '00110111': '7', '00111000':'8', '00111001':'9', '00100001':'!'
    }
    #creates an empty string that stores the new word letter by letter
    encrypted = ''
    #iterates through the inputted binary, splits it into segments of 8, then uses the conversion table to evaluate binary to words, letter by letter
    for i in range(0, len(full_string), 8):
        byte = full_string[i:i+8]
        encrypted += convert_table.get(byte)
    return encrypted


def main():
    #Gets the user inputs to use in the code
    image_path = input("Enter the path of the image you want to load: ")
    start = input("Enter the start sequence: ")
    end = input("Enter the end sequence: ")
    #Converting start and end sequence to binary
```

```
    start_bi = convert_input(start)
    end_bi = convert_input(end)
    #extract(image_path, start_bi, end_bi)
    result = extract(image_path, start_bi, end_bi)
    if result != "Start or end sequence not found in image.":
        print(f"Converted text: {cipher(result)}")
    else:
        print("Start or end sequence not found in the image.")
#Ending code
if __name__ == "__main__":
    main()
```

Checkpoint 2 Task 1 Code:

```
import numpy as np
import matplotlib.pyplot as plt


def compare_images(image1_filename, image2_filename, output_filename):
    # Load the images
    image_1 = plt.imread(image1_filename)
```

```
    image_2 = plt.imread(image2_filename)



    # Ensure both images have the same number of channels
    if image_1.shape[-1] == 4:
        image_1 = image_1[:, :, :3]
    if image_2.shape[-1] == 4:
        image_2 = image_2[:, :, :3]



    # Check if the images are the same size
    if image_1.shape != image_2.shape:
        print("Cannot compare images of different sizes.")
        return False



    # Compare the images
    diff = np.abs(image_1 - image_2)
    if image_1.ndim ==2:
        cmap="grey"
    else:
```

```
        cmap=None
    plot_this = np.where(diff > 0, 255, 0).astype(np.uint8)


    plt.imsave(output_filename, plot_this)
    if cmap:
        plt.imshow(plot_this, cmap=cmap)
    else:
        plt.imshow(plot_this,)
    plt.show()


    # Determine if there are any differences
    identical = not np.any(diff)
    return identical


def main():
    # Prompt the user for file names
    image_1_file = input("Enter the path of your first image: ")
    image_2_file = input("Enter the path of your second image: ")
```

```
    output_file = input("Enter the path for the output image: ")


    # Compare the images
    identical = compare_images(image_1_file, image_2_file, output_file)


    if identical:
        print("The images are the same.")
    else:
        print("The images are different.")


if __name__ == "__main__":
    main()
```

Checkpoint 2 Task 2 Code:

```
def vigenere_encrypt(text, key):
    # List to store the encrypted characters
    converted = []


    # Define the alphabet and digit sets
```

```
    alphabet_upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    alphabet_lower = "abcdefghijklmnopqrstuvwxyz"

    digits = "0123456789"



    # Get the length of the key for repeating it

    key_length = len(key)



    # Convert the key into a list that is shifted, for example A=0, B=1, ...

    key_shifts = [alphabet_upper.index(char.upper()) for char in key]



    # Encrypt each character in the text

    for i in range(len(text)):

        char = text[i]

        shift = key_shifts[i % key_length]



        # Check if the character is an uppercase letter

        if char in alphabet_upper:

            index = alphabet_upper.index(char)
```

```
        new_index = (index + shift) % 26
        encrypted_char = alphabet_upper[new_index]

    # Check if the character is a lowercase letter
    elif char in alphabet_lower:
        index = alphabet_lower.index(char)
        new_index = (index + shift) % 26
        encrypted_char = alphabet_lower[new_index]

    # Check if the character is a digit
    elif char in digits:
        index = digits.index(char)
        new_index = (index + shift % 10) % 10
        encrypted_char = digits[new_index]

    # Leave non-alphabetic and non-numeric characters unchanged
    else:
        encrypted_char = char
```

```
        converted.append(encrypted_char)


    # Combines the list to a string, which can be returned at the end of the function
    return ''.join(converted)


def string_to_binary(message, start_seq, end_seq):
    # Convert a string to its binary representation using ASCII values
    def to_binary(s):
        return ' '.join(format(ord(char), '08b') for char in s)


    # Get binary forms of the sequences and the message
    start_bin = to_binary(start_seq)
    message_bin = to_binary(message)
    end_bin = to_binary(end_seq)


    # Combine them with spaces in between
    return start_bin + ' ' + message_bin + ' ' + end_bin
```

```
def main():
    # Get user inputs
    text = input("Enter the plaintext message: ")
    key = input("Enter the key: ")
    start = input("Enter the start sequence: ")
    end = input("Enter the end sequence: ")

    # Encrypt the message and convert it to binary
    encrypted_message = vigenere_encrypt(text, key)
    binary_message = string_to_binary(encrypted_message, start, end)

    # Print the results
    print("Encrypted Message using Vigenere Cipher:", encrypted_message)
    print("Binary output message:", binary_message)

if __name__ == "__main__":
    main()
```

Checkpoint 1 Task 3 Code:

```
#Importing the required modules
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np


#Defining the encrypt function
def encrypt(text, passphrase):
    cipher_txt = [] #Empty array
    key_len = len(passphrase) #Setting the key length
    index = 0 #Initialising index
    num = "0123456789"


    for character in text:
        # Check if the character is a letter
        if character.isalpha():
            offset = ord(passphrase[index % key_len].upper()) - ord('A')
            #Check if the charcter is uppercase
```

```
        if character.isupper():
            new_char = chr((ord(character) - ord('A') + offset) % 26 + ord('A'))
        #Otherwise, handle lowercase
        else:
            new_char = chr((ord(character) - ord('a') + offset) % 26 + ord('a'))
        #Append the encrypted character
        cipher_txt.append(new_char)
        #Move to the next character
        index += 1
    #Check if the chasracter is a number
    elif character in num:
        offset = (int(character) + ord(passphrase[index % key_len].upper()) - ord('A')) % 10
        cipher_txt.append(str(offset))
        index += 1
    #Ignores non characters and non numbers
    else:
        cipher_txt.append(character)
        index += 1
```

```
    return ''.join(cipher_txt)


# Convert each character in the start, end, and content to an 8-bit binary string
def binary_string(start, content, end):
    def string_to_binary(s):
        return ' '.join(format(ord(ch), '08b') for ch in s)
    start_bin = string_to_binary(start)
    content_bin = string_to_binary(content)
    end_bin = string_to_binary(end)
    return f"{start_bin} {content_bin} {end_bin}"


def encode_image(binary_data, inputimg, encoded_image_path):
    # Split the binary data into a sequence of individual bits
    binary_sequence = list(''.join(binary_data.split()))
    # Open the input image
    img = Image.open(inputimg)
    # Convert the image to a numpy array
    img_array = np.array(img)
```

```
# Check if the image can hold the entire binary sequence
if len(binary_sequence) > img_array.size:
    return


bin_index = 0
#Check if the image is RGB
if len(img_array.shape) == 3:
    # Iterate through each pixel in the image
    for i in range(img_array.shape[0]):
        for j in range(img_array.shape[1]):
            for k in range(img_array.shape[2]):
                if bin_index < len(binary_sequence):
                    # Modify the least significant bit of the pixel channel using 0xFE
                    img_array[i, j, k] &= 0XFE
                    img_array[i, j, k] |= int(binary_sequence[bin_index])
                    bin_index += 1
                    cmap = None
```

```
            else:
                #Exit if all bits are encoded
                break
        if bin_index >= len(binary_sequence):
            break
    if bin_index >= len(binary_sequence):
        break
#Check if the image is greyscale
elif len(img_array.shape) == 2:
    for i in range(img_array.shape[0]):
        for j in range(img_array.shape[1]):
            if bin_index < len(binary_sequence):
                img_array[i, j] &= 0XFE
                img_array[i, j] |= int(binary_sequence[bin_index])
                bin_index += 1
                cmap = "gray"
            else:
                break
```

```
        if bin_index >= len(binary_sequence):
            break


    #Convert from an array back into an image
    encoded_img = Image.fromarray(img_array)
    #Save encoded image to the path that is inputted
    encoded_img.save(encoded_image_path)
    print("Message successfully encoded and saved to: ", encoded_image_path)


    #Showing the output
    if cmap:
        plt.imshow(img_array, cmap=cmap)
    else:
        plt.imshow(img_array,)
    plt.axis('off')
    plt.show()


#Getting user inputs and printing required outputs
```

```
def main():
    text_input = input("Enter the plaintext you want to encrypt: ")
    cipher_key = input("Enter the key for Vigenere ciphertxt: ")
    start = input("Enter the start sequence: ")
    end = input("Enter the end sequence: ")
    input_img = input("Enter the path of the image: ")
    output_img = input("Enter the path for the encoded image: ")
    encrypted_text = encrypt(text_input, cipher_key)
    print("Encrypted Message using Vigenere Cipher: ", encrypted_text)
    binary_msg = binary_string(start, encrypted_text, end)
    print("Binary output message: ", binary_msg)
    encode_image(binary_msg, input_img, output_img)


if __name__ == "__main__":
    main()
```

Demo Task 1 Code:

```
from PIL import Image
import numpy as np
```

```
import matplotlib.pyplot as plt


def input_to_bi(input): #Transform the string into binary code
    return ''.join(format(ord(c), '08b') for c in input)


def bi_to_text(bi):
    return ''.join(chr(int(bi[i:i+8], 2)) for i in range(0, len(bi), 8))
    # Convert a binary string (sequence of '0' and '1') into a text string
    # The binary string is split into chunks of 8 bits (1 byte), and each byte is converted into a character using chr()


def xor_cipher(plain_txt, key):
    return ''.join(chr(ord(c) ^ ord(key[i % len(key)])) for i, c in enumerate(plain_txt))


    # Perform XOR encryption/decryption on the given plaintext using the provided key
    # Each character in the plaintext is XORed with the corresponding character from the key
    # The key is repeated cyclically to match the length of the plaintext
```

```
def caesar_cipher(plain_txt, key):
    shift = int(key) % 26 #converts value to a 0-26 scale for the caesar cipher
    result = "" #initiliazes empty string
    for char in plain_txt:
        if char.isalpha(): #if the character is an alphabet
            ascii_offset = ord('A') if char.isupper() else ord('a') #checks if the character is upper or lower case, and then sets the offset based on that
            result += chr((ord(char) - ascii_offset + shift) % 26 + ascii_offset)
            #Shifts the character by the given amount (wrapping within the alphabet as needed), and converts it back to a character, adding it to the final encrypted string (result).
        else:
            result += char #just adds the result to the code without shifting it as it isnt an alphabet
    return result



def vigenere_cipher(plain_txt, key):
    key = key.upper() #changes the key to all uppercase letters
    result = "" #initializes an empty string
    key_index = 0 #starts index at 0
```

```
    for char in plain_txt: #loops through every character in the text for encryption
        if char.isalpha(): #if the character is an alphabet
            ascii_offset = ord('A') if char.isupper() else ord('a') #the offset is either 65 (A) or 97(a),
depending on if the letter is upper/lower case.
                #This sets the value of the character = 0, so that we go in a range of 0-26
            shift = ord(key[key_index % len(key)]) - ord('A')
            #Cycles the key over and over again to match the charactes. From here, it subtracts the letter
matching the key index with A, to find the total shift
            result += chr((ord(char) - ascii_offset + shift) % 26 + ascii_offset)
            #this then uses the total shift found above along with the character given to then output the
new encrypted message into a string.
            key_index += 1
        else:
            result += char #adds the character if it is not an alphabet by not shifting it
    return result


def encode_image(input_image, output_image, bi_message, offset):
    # Convert the image to RGB and then convert into an array
    img = Image.open(input_image).convert("RGB")
```

```
img_array = np.array(img)

# Calculate the total number of bits and see if the message length exceeds the total number of bits

total_bits = img_array.size * 8

if (offset + len(bi_message)) > total_bits:

    raise ValueError("Given message is too long to be encoded in the image.")

# Convert binary into a list of characters

binary_sequence = list(bi_message)

bin_index = 0

bit_count = 0



# Iterate over each pixel in the image array

for i in np.ndindex(img_array.shape[:-1]):

    # Once the bit count reaches the bit offset, start encoding the message.

    if bit_count >= offset and bin_index < len(binary_sequence):

        # Iterate over the RGB channels of each pixel.

        for channel in range(img_array.shape[2]):

            # For each channel, modify the LSB with the current binary message bit.

            if bin_index < len(binary_sequence):
```

```
            img_array[i][channel] &= 0XFE
            img_array[i][channel] |= int(binary_sequence[bin_index])
            bin_index += 1  # Move to the next bit in the binary message.
        bit_count += 3 # Increment the bit count by 3
        if bin_index >= len(binary_sequence):
            break
    # Convert the array back into an image and save to output file
    encoded_img = Image.fromarray(img_array)
    encoded_img.save(output_image)


def compare_images(image1_path, image2_path, output_path):
    #Converting both images to RGB
    image1 = Image.open(image1_path).convert("RGB")
    image2 = Image.open(image2_path).convert("RGB")
    # Check if images are the same size
    if image1.size != image2.size:
        raise ValueError("Images must have the same size")
    # Turning the images into arrays
```

```
img1_array = np.array(image1)

img2_array = np.array(image2)

# Calculate the absolute difference between the two image arrays

diff_array = np.abs(img1_array.astype(np.float32) - img2_array.astype(np.float32))

#Convert difference array into 8bit and create an image from the difference

diff_image = Image.fromarray(diff_array.astype(np.uint8))

# Save the image to the output file

diff_image.save(output_path)

# Create three subplots

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

# Display original image

ax1.imshow(image1)

ax1.set_title("Original Image")

ax1.axis('off')

# Display encoded image

ax2.imshow(image2)

ax2.set_title("Encoded Image")

ax2.axis('off')
```

```
    # Display the difference between the two images
    ax3.imshow(diff_image)
    ax3.set_title("Difference")
    ax3.axis('off')


    plt.tight_layout()
    plt.show()


    return np.array_equal(img1_array, img2_array)


def main():
    # Obtain inputs from the user
    cipher_choice = input("Enter the cipher you want to use for encryption: ")
    plain_txt = input("Enter the plaintext you want to encrypt: ")
    key = input("Enter the key for the cipher: ")
    start = input("Enter the start sequence: ")
    end = input("Enter the end sequence: ")
    offset = int(input("Enter the bit offset before you want to start encoding: "))
```

```
input_image = input("Enter the path of the input image: ")

output_image = input("Enter the path for your encoded image: ")

compare_image = input("Enter the path of the image you want to compare: ")



# Determine which cipher is being used

if cipher_choice == "xor":

    encrypted_message = xor_cipher(plain_txt, key)

    name_cipher = "Xor"

elif cipher_choice == "caesar":

    encrypted_message = caesar_cipher(plain_txt, int(key))

    name_cipher = "Caesar"

elif cipher_choice == "vigenere":

    encrypted_message = vigenere_cipher(plain_txt, key)

    name_cipher = "Vigenere"



# Print the encrypted message as a string

print(f"Encrypted Message using {name_cipher} Cipher: {encrypted_message}")

# Convert the message to a binary string and print as groups of 8 bits
```

```
    start_bi = input_to_bi(start)
    end_bi = input_to_bi(end)
    encrypted_message = input_to_bi(encrypted_message)
    bi_message = start_bi + encrypted_message + end_bi
    print(f"Binary output message: {bi_message}")


    # Encode the encrypted message into the input image and save to the output file
    encode_image(input_image, output_image, bi_message, offset)
    print(f"Message successfully encoded and saved to: {output_image}")
    # Compare the output image with another image and save the differences to an output file
    images_identical = compare_images(output_image, compare_image, "diff_image.png")
    if images_identical:
        print("The images are the same.")
    else:
        print("The images are different.")


if __name__ == "__main__":
    main()
```

Demo Task 2 Code:

```
import numpy as np
import matplotlib.pyplot as plt


def image(file_name):
    #Reusing the same code from Checkpoint 1 Task 1 to read the image and convert it to an array
    image = plt.imread(file_name)
    image_array = np.array(image)
    if image_array.dtype != np.uint8:
        image_array = (image_array * 255).astype(np.uint8)
    #New code that indexes the 3D array to extract the last bit of every bite
    total_binary = []
    #Accounting for both 2D and 3D arrays depending on if the image is colored or greyscale
    if len(image_array.shape) == 3:
        #Index 3 total times to get the 8th bit of colored image
        for i in range(image_array.shape[0]):
            for j in range(image_array.shape[1]):
                for k in range(image_array.shape[2]):
```

```
                total_binary.append(image_array[i, j, k] & 1)
    elif len(image_array.shape) == 2:
        #Index 2 total times to get the 8th bit of greyscale image
        for i in range(image_array.shape[0]):
            for j in range(image_array.shape[1]):
                total_binary.append(image_array[i, j] & 1)
    return total_binary


def convert_input(input_text):
    #Found on stack overflow https://stackoverflow.com/questions/18815820/how-to-convert-string-to-binary
    input_bi = ''.join(format(ord(i), '08b') for i in input_text)
    return input_bi


def extract(image_bi, start, end):
    #Converts each element of binary_data to a string and then concentrates them into a single binary string
    binary_string = ''.join(map(str, image_bi))
```

```
    #find the start binary code inside the string
    start_index = binary_string.find(start)
    #find the end binary code inside the string
    end_index = binary_string.find(end, start_index + len(start))
    #if the start and end binary code are inside the string
    if start_index != -1 and end_index != -1:
        start_index += len(start)
        hidden_message = binary_string[start_index:end_index]
        print("Extracted Binary Message: ", hidden_message)
    #if it cannot find the start or the end point inside the string
    else:
        print("Start or end sequence not found in image.")
        return False
    return hidden_message


# Function to convert a binary string to text
def binary_to_text(binary):
    # Convert every 8 bits (1 byte) of the binary string to a character
```

```
    return ''.join(chr(int(binary[i:i+8], 2)) for i in range(0, len(binary), 8))


# Function to decrypt a Caesar cipher given a shift value
def caesar_decrypt(ciphertext, shift):
    shift = int(shift) % 26  # Ensure the shift is within the range of the alphabet (0-25)
    plaintext = ""  # Initialize an empty string for the decrypted text
    # Loop through each character in the ciphertext
    for char in ciphertext:
        if char.isalpha():  # Only decrypt alphabetic characters
            # Determine if the character is uppercase or lowercase
            ascii_offset = ord('A') if char.isupper() else ord('a')
            # Shift the character back by the given shift value
            plaintext += chr((ord(char) - ascii_offset - shift) % 26 + ascii_offset)
        else:
            # Non-alphabetic characters remain unchanged
            plaintext += char
    return plaintext  # Return the decrypted text
```

```
# Function to decrypt using XOR cipher with a repeating key
def xor_decrypt(ciphertext, key):
    # Loop through the ciphertext and XOR each character with the corresponding character in the key
    return ''.join(chr(ord(c) ^ ord(key[i % len(key)])) for i, c in enumerate(ciphertext))


# Function to decrypt a Vigenère cipher given a key
def vigenere_decrypt(ciphertext, key):
    key = key.upper()  # Ensure the key is uppercase
    key_length = len(key)  # Get the length of the key
    key_int = [ord(i) for i in key]  # Convert key characters to integers (ASCII values)
    plaintext = []  # Initialize an empty list for the decrypted text


    # Loop through each character in the ciphertext
    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():  # Only decrypt alphabetic characters
            # Determine if the character is uppercase or lowercase
            offset = 65 if ciphertext[i].isupper() else 97
            c = ord(ciphertext[i].upper()) - 65  # Convert character to a number (A=0, B=1, etc.)
```

```
            k = key_int[i % key_length] - 65  # Get corresponding key character and convert to a number
            # Decrypt the character and append it to the plaintext list
            decrypted_char = chr((c - k) % 26 + offset)
            plaintext.append(decrypted_char.lower() if ciphertext[i].islower() else decrypted_char)
        elif ciphertext[i].isdigit():  # Decrypt numeric characters (optional enhancement)
            c = int(ciphertext[i])
            k = key_int[i % key_length] - 65  # Use modulo 10 for numbers
            decrypted_char = str((c - (k % 10)) % 10)
            plaintext.append(decrypted_char)
        else:
            # Non-alphabetic and non-numeric characters remain unchanged
            plaintext.append(ciphertext[i])
    return ''.join(plaintext)  # Return the decrypted text as a string


# Main function to coordinate the extraction and decryption process
def main():
    # Prompt user for decryption method, key, start and end sequences, and image path
    cipher = input("Enter the cipher you want to use for encryption: ")
```

```
key = input("Enter the key for the cipher: ")

start_sequence = input("Enter the start sequence: ")

end_sequence = input("Enter the end sequence: ")

input_image = input("Enter the path of the input image: ")



# Extract the binary message from the image

image_bi = image(input_image)

start_bi = convert_input(start_sequence)

end_bi =convert_input(end_sequence)



# Convert the extracted binary message to text (ciphertext)

message_bi = extract(image_bi,start_bi,end_bi)

if message_bi == False:

   return

else:

   ciphertext = binary_to_text(message_bi)

# Determine which decryption method to use based on user input

   if cipher == 'caesar':
```

```
            plaintext = caesar_decrypt(ciphertext, key)
        elif cipher == 'xor':
            plaintext = xor_decrypt(ciphertext, key)
        elif cipher == 'vigenere':
            plaintext = vigenere_decrypt(ciphertext, key)


    # Print the results: binary message, ciphertext, and decrypted plaintext
        print(f"Converted Binary Text: {ciphertext}")
        print(f"Converted text: {plaintext}")


# Run the main function when the script is executed
if __name__ == "__main__":
    main()
```