Python Team Project Report

[Lisette, Divya, Olivia, Tavish]

[LC2]: Team [10]

[ENGR133]

[Professor Dustker]

[10/14/2024]

1. Project Motivation

There are several image processing methods that are used in biomedical engineering that have been advancing the technological abilities of this discipline for years. According to an article titled *Image Processing Techniques in Biomedical Engineering*, some of these methods include scanning the structures of the internal organs and examining their behavior, 2D or 3D ultrasound imaging, along with multi-slice computed tomography, and endoscopic imaging. These are not the only imaging modalities, however. There are other methods such as the use of radiography, thermography, nuclear medicine, CT, and much more. The vast availability of these advances can help outline possibilities regarding non-invasive medical operations, as well as the detection and diagnosis of cancer. This industry also offers the unique opportunity for specialty focuses, such as heart imaging. Modern heart imaging is one of the most challenging topics in modern healthcare, and comes along with expansive development and growth. Current methods in this specialty include the use of noninvasive ultrasounds, cineangiography, electrocardiograms (ECG), electroencephalograms (EEG), and more. This field, now referred to as biomedical image processing, combines biomedical engineering and computer science, and is intended to improve the accuracy of the information that is obtained from the images. According to an article titled *Advances in biomedical signal and image processing - A systematic review*, this field utilizes techniques such as segmentation, enhancement, detection of the region of interest, the thresholding technique, the pre-filtering method, and morphological operations. These techniques can be used to further analyze and improve the care provided through biomedical engineering, which is essential for the proper treatment in healthcare. Imaging processing methods are an essential part of biomedical engineering, and provide interesting, new, and challenging projects that can help the progression of this field.

2. Project Overview and Methods

Our project involves using the steganography technique to hide and extract messages in images, by implementing the Least Significant Bit (LSB) technique. Steganography is the practice of hiding an image, message, or file within something that isn't a secret. The LSB method specifically hides data in the least significant bits of pixel values in digital images, making small changes that the naked human eye would be unable to detect.

Background on Steganography and LSB
Algorithms Implemented
The project is comprised of three main ciphers: XOR, Caesar, and Vigenère, that are used for the purpose of encrypting and decrypting the message before and after the encoding of the images. The user is first required to choose a cipher and then

input a plaintext message along with a key in the process. The message is then encrypted using a specific cipher as follows: O

- XOR Cipher: The message is first converted to binary, and + :: for each pair of symbols, the algorithm takes the pointer of the first symbol and the key. The result obtained after the execution of this XOR operation must be the ASCIl value of the character that is to be encrypted.
- Caesar Cipher: A substitution cipher that shifts each letter of the message by a specified number of positions in the alphabet.
- Vigenère Cipher: A common polyalphabetic cipher that perturbs characters in a message because of the presence of corresponding digits in the key. The key is recycled if it runs out of characters to be attached to.

The encrypted message is then changed into the form of a binary string, and the designed start and end sequences are added to make it a complete message. The embed_msg function is the one which takes the input binary message and hides it in the least significant bit of the pixels of an image with a given bit offset (encode_msgfunction). If the length of the message is more, that is, if it is more than the number of bits available in the image, an error will be popped.
This project also incorporates a comparison function that inspects the pixel values of two images to determine the amount of difference in pixel values. This technique can be used to verify the fact that the encoded message does not produce any visible effect on the image.

Methods and Evidence

The choice of LSB is a wise one for this project, because it is one + :: of the most popularly used techniques in image steganography because of its easy implementation and efficiency. It is a reliable way of concealing a message and the image's visual quality is also not affected. Among the mix of XOR, Caesar, and Vigenère ciphers, each one was chosen considering their standpoints of strengths coming from different encryption aspects and not only from simple and complex methods. Besides encryption, each of the ciphers adds disparate levels of security. The fact of having multiple ciphers to choose from comes with additional advantages as diverse solutions opened up in both encoding and decoding schemes.

3. Discussion of Algorithm Design

This project implements a multi-faceted algorithm for encrypting and hiding text in an image using three ciphers: XOR, Caesar, and Vigenère. It also provides functionality

for decoding the hidden message from the image and decrypting it back into plaintext. In this project, we have decided to use a modular approach to ensure clarity, maintainability, and flexibility. Each function within the code is designed to perform a specific task—whether it is encryption, binary conversion, image encoding, or comparison—making the code both easier to read and more adaptable. By keeping the functions self-contained and reusable, adjustments or expansions can be made to individual components without impacting the overall system.

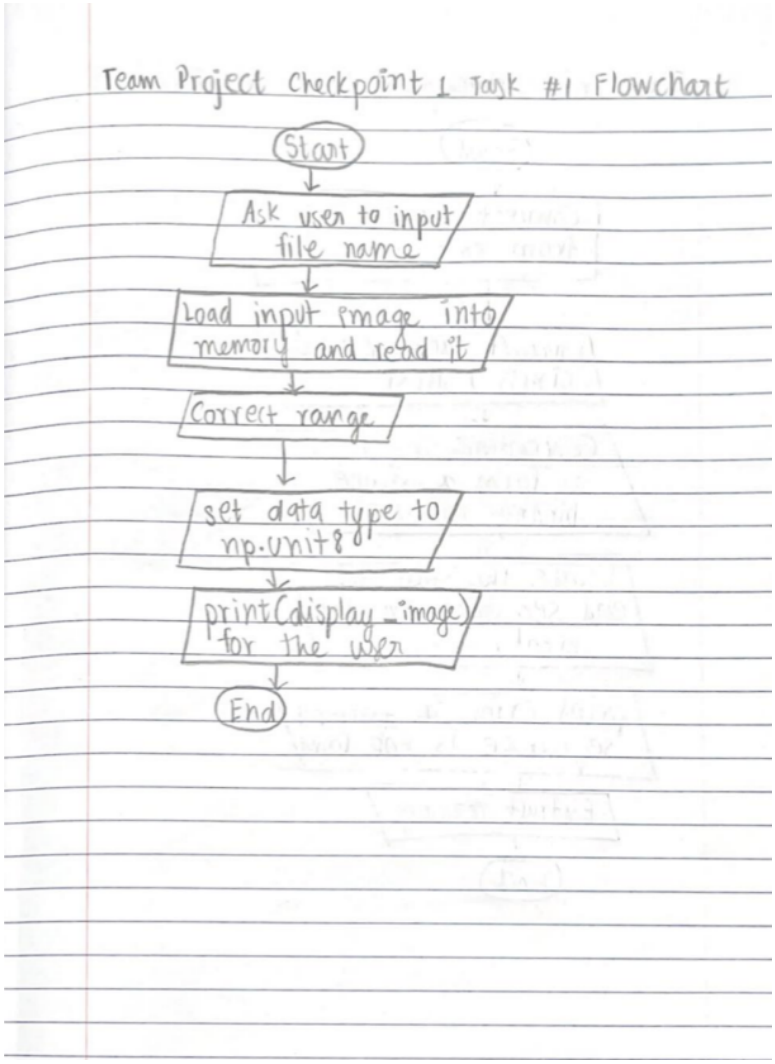
Our decision to modularize the code comes with numerous benefits. First, it enhances clarity, as smaller functions are easier to understand in isolation. For instance, if changes are needed for a particular encryption algorithm, such as the Vigenère cipher, those changes can be made without needing to modify or even fully grasp the image encoding logic. This structure also supports testability, allowing each function to be debugged independently. If an issue arises in one part of the program, we can focus on that segment alone without being distracted by unrelated functionality like encryption. Reusability is another key advantage of this modular design. Functions like xor() and encode_msg() can be easily used in order programs or applied in different contexts without needing significant changes. Furthermore, the modular design ensures that the algorithm is extensible, enabling easy additions of new ciphers or encoding techniques. For example, when the XOR and Caesar ciphers are introduced, only the relevant cipher function needs to be implemented, while the rest of the workflow (binary conversion, encoding, etc.) remains intact. Overall, the modular structure ensures that the algorithm is easy to maintain and extend while making the code easier to understand, test, and reuse across different tasks. Below is an explanation of the core components of each of the code.

CP1

The purpose of this task is to use the concept of steganography, specifically using the Least Significant Bit (LSB) technique to extract hidden messages from images. By breaking down the process into modular functions, we can better understand the syntax involved in loading images, extracting LSBs, converting binary data to text, and handling potential errors. This task also reinforced our skills in image processing, binary data manipulation, and error handling in Python.

CP1 Task 1

Flowchart:



Steps:

- Call `main` function: Control transfers to the `main` function which is responsible for prompting the user and calling the `display_image` function.
- `main` function:
    - Prompt user for image name: Prompts the user to enter the image name including its extension.
    - Call `display_image` function: Passes the user-entered image name to the `display_image` function.
- `display_image` function:

    - Load the image: Reads the image data using `plt.imread` function based on the provided image name.
    - Print image data type: Prints the data type of the loaded image.
        - Check if image is float32: If the data type is `float32`, it converts the image to `uint8` data type using scaling and casting.
    - Display the image:
        - Check number of dimensions: If the image has less than 3 dimensions (grayscale), it displays the image using `imshow` function with a grayscale colormap.
        - For RGB images: If the image has 3 or more dimensions (RGB), it displays the image using the `imshow`function.
    - End of `display_image` function: Control returns back to the `main` function after displaying the image.
- End of `main` function: Control returns back to the starting point where the program terminates.

Explanation of Code Functions:

`main` function:

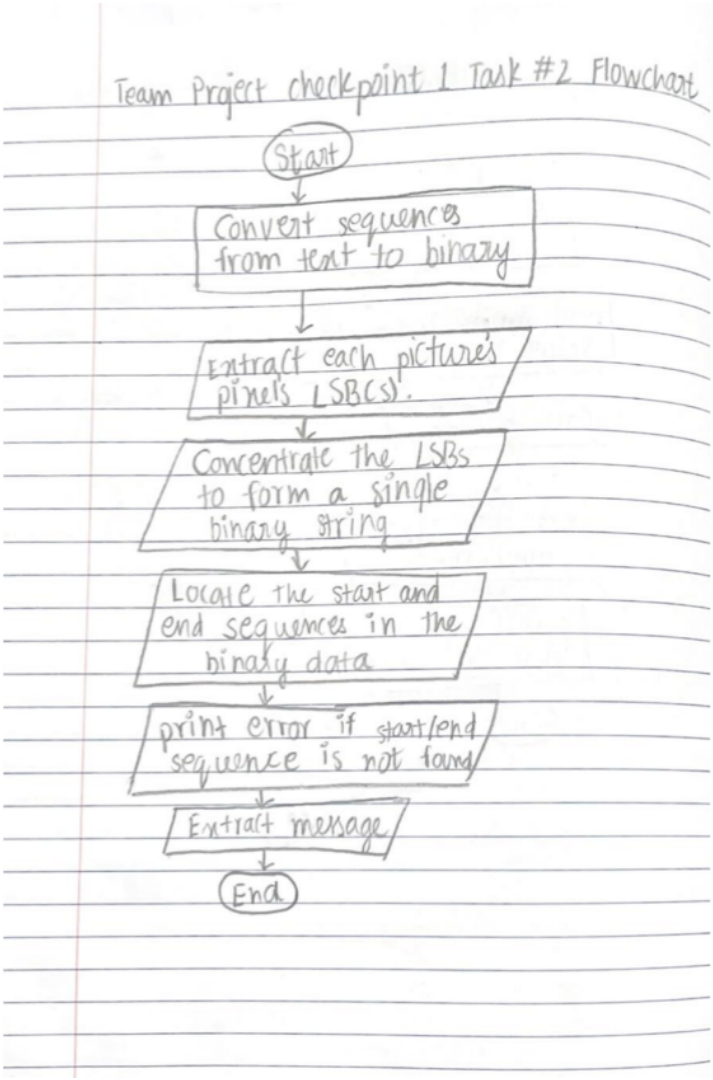
- This function serves as the entry point for the program execution.
- It prompts the user to enter the image name and calls the `display_image` function to process and display the image.
- By separating the user interaction from the image processing logic, the `main` function becomes more focused and easier to understand.

`display_image` function:

- This function encapsulates the logic for loading, processing, and displaying an image.
- It takes the image name (including extension) as input.
- It loads the image from the file system using `plt.imread`.
- It checks the data type of the loaded image and converts it to `uint8` if necessary. This ensures that the image data is in a consistent format for further processing.
- It displays the image using `plt.imshow`, considering the number of dimensions (grayscale or RGB images).
- By separating this functionality into a dedicated function, the code becomes more reusable and easier to maintain. If changes are needed to how the image is displayed, they can be made within this function without affecting other parts of the program.

CP1 Task 2

Flowchart:



Steps:

- Convert sequences from text to binary: Convert the start and end sequences from text to binary format using `text_to_binary`.
- Extract each picture's pixels (LSBs): Extract the least significant bit (LSB) from each pixel in the image using `extract_lsb`.
- Concentrate the LSBs to form a single binary string: Combine the extracted LSBs into a single binary string.
- Locate the start and end sequences in the binary data: Find the positions of the start and end sequences within the binary string.

- Print error if start/end sequence is not found: If either the start or end sequence is not found, print an error message.
- Extract message: If both start and end sequences are found, extract the hidden message between them.

Explanation of Code Functions:

1. `text_to_binary(text)`:

- Converts the given text string into a binary string by converting each character to its ASCII code and then to its binary representation.

2. `extract_lsb(image_name)`:

- Loads the image from the specified file path.
- Converts the image to `uint8` format if necessary.
- Iterates through each pixel in the image.
- Extracts the LSB from each pixel's color channels (red, green, blue).
- Concatenates the extracted LSBs into a single binary string.

3. `extract_message(binary_data, start_seq, end_seq)`:

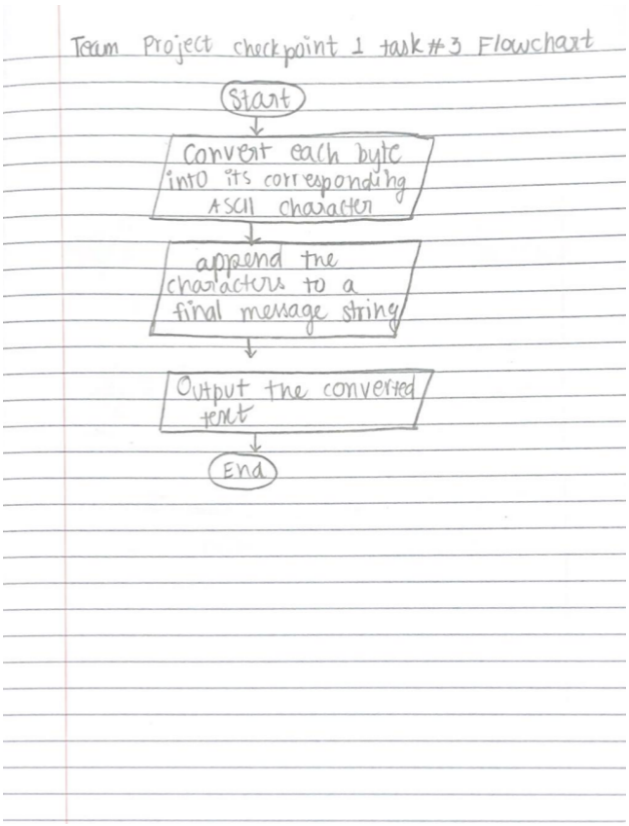
- Finds the index of the start sequence within the binary data.
- Finds the index of the end sequence within the binary data, starting from the index after the start sequence.
- If either the start or end sequence is not found, returns `False`.
- If both sequences are found, extracts the substring between the start and end sequences and returns it as the hidden message.

4. `main()`:

- Prompts the user to enter the image path, start sequence, and end sequence.
- Converts the start and end sequences to binary using `text_to_binary`.
- Extracts the LSBs from the image using `extract_lsb`.
- Extracts the hidden message using `extract_message`.
- Prints the extracted message or an error message if the sequences are not found.

CP1 Task 3

Flowchart:



Steps:

- Convert each byte into its corresponding ASCII character: Iterate through the binary string in 8-bit chunks, convert each chunk to an integer, and then to its corresponding ASCII character.
- Append the characters to a final message string: Concatenate the converted characters to form the final message string.
- Output the converted text: Print the final message string.
- End

Explanation of Code Functions:

1. `binary_to_text(binary)`:

- Iterates through the binary string in 8-bit chunks.
- Converts each chunk to an integer using base 2.
- Converts the integer to its corresponding ASCII character using `chr`.

- Concatenates the characters to form the final message string.

2. `main()`:

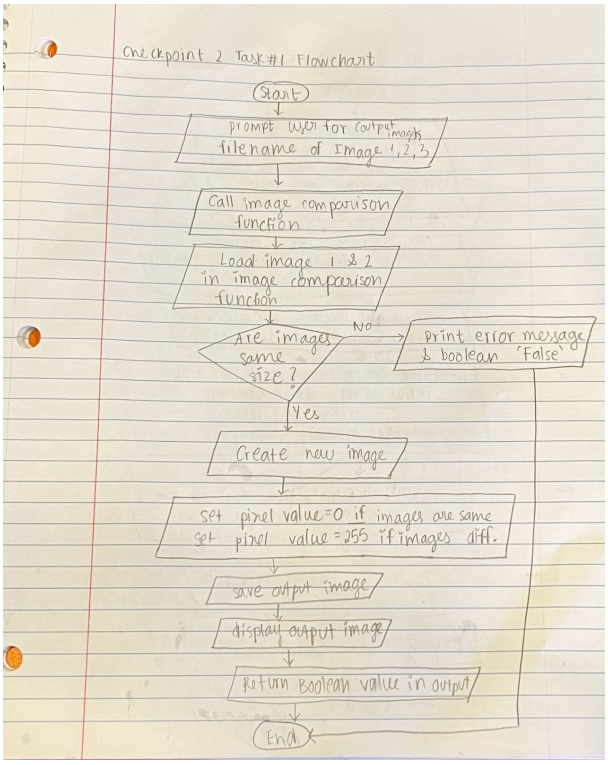
- Prompts the user to enter the binary message.
- Calls the `binary_to_text` function to convert the binary message to text.
- Prints the extracted message.

CP2

The purpose of this task is to help us understand the concepts of image comparison and encryption. By developing functions to compare images pixel by pixel and implementing various encryption techniques, we gained a deeper understanding of how to manipulate and protect data. Additionally, we also practiced converting encrypted messages into binary strings and encoding them within images. This task reinforced our skills in image processing, encryption algorithms, and binary data manipulation.

CP2 Task 1

Flowchart:



Steps:

1. Import libraries: Imports `matplotlib.pyplot` for image processing and visualization.

2. `compare_images(img1, img2, img3_name)`:
    - Purpose: Compares two images pixel by pixel and creates a new image highlighting the differences.
    - Parameters:
        1. `img1`: Path to the first image.
        2. `img2`: Path to the second image.
        3. `img3_name`: Path for the output image.
    - Returns: `True` if images are identical, `False` otherwise.
    - Steps:
        1. Loads the input images using `plt.imread`.
        2. Converts images to `uint8` format if necessary.
        3. Checks if images have the same dimensions and mode (RGBA vs grayscale).
        4. Iterates through each pixel of both images, comparing corresponding pixel values.
        5. Creates a new image (`img3`) to store the comparison results.
        6. Assigns 0 to pixels that are identical and 255 to pixels that differ.
        7. Displays and saves the output image (`img3`).
        8. Returns `True` if no differences were found, `False` otherwise.

3. `main()`:

- Purpose: Handles user input, calls the `compare_images` function, and prints the comparison result.
- Steps:
    1. Prompts the user to enter the paths for the first, second, and output images.
    2. Calls the `compare_images` function with the provided paths.
    3. Prints a message indicating if the images are the same or different based on the function's return value.

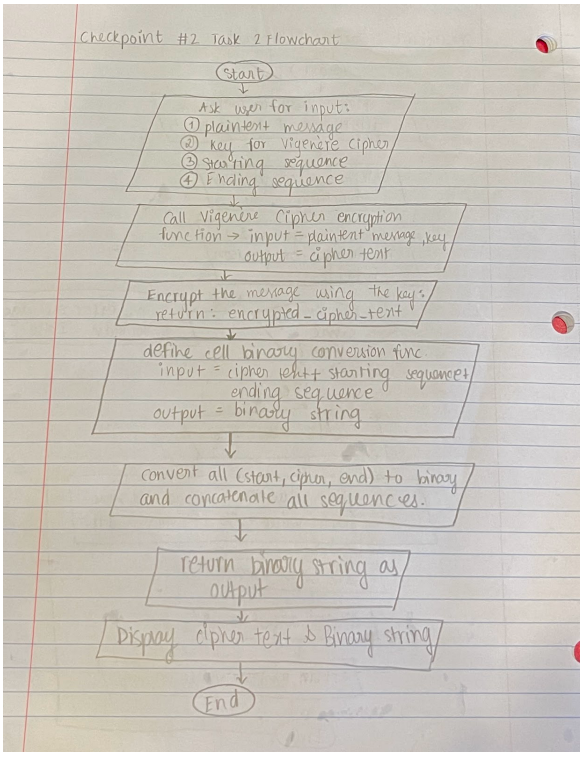
Design Rationale for Modularization:

- Separation of Concerns: The `compare_images` function encapsulates the image comparison logic, improving code clarity and maintainability.
- Reusability: The `compare_images` function can potentially be reused in other projects requiring image comparisons.

- Testability: The modular structure allows for easier testing of the `compare_images` function.

CP2 Task 2

Flowchart:



Purpose:

This code combines encryption using the Vigenère cipher with text-to-binary conversion. It takes a plaintext message, a key, and start/end sequences as input, encrypts the message using Vigenère cipher, and then converts the encrypted message to binary format with added start/end sequences.

Steps of Execution:

- Import functions: Imports the necessary functions from the `tp1_team_2_10` module.
- Prompt user for input: Prompts the user to enter the plaintext message, key, and start/end sequences.
- Encrypt message: Encrypts the message using the `cipher` function, applying the Vigenère cipher algorithm.

- Convert to binary: Converts the encrypted message, along with the start and end sequences, to binary format using the `text_to_binary` function.
- Print output: Prints the encrypted message in both text and binary formats.

Explanation of Functions:

1. `cipher(message, key)`:

- Encrypts the given plaintext message using the Vigenère cipher with the provided key.
- Iterates through each character in the message.
- Determines the character's case (uppercase, lowercase, or number).
- Applies the Vigenère cipher algorithm using the corresponding character from the key.
- Appends the encrypted character to the output string.
- Returns the encrypted message.

2. `text_to_binary(start, text, end)`:

- Converts the given text, start sequence, and end sequence into binary format.
- Concatenates the binary representations of the start sequence, text, and end sequence.
- Formats the binary string into 8-bit chunks with spaces between them.
- Returns the formatted binary string.

3. `main()`:

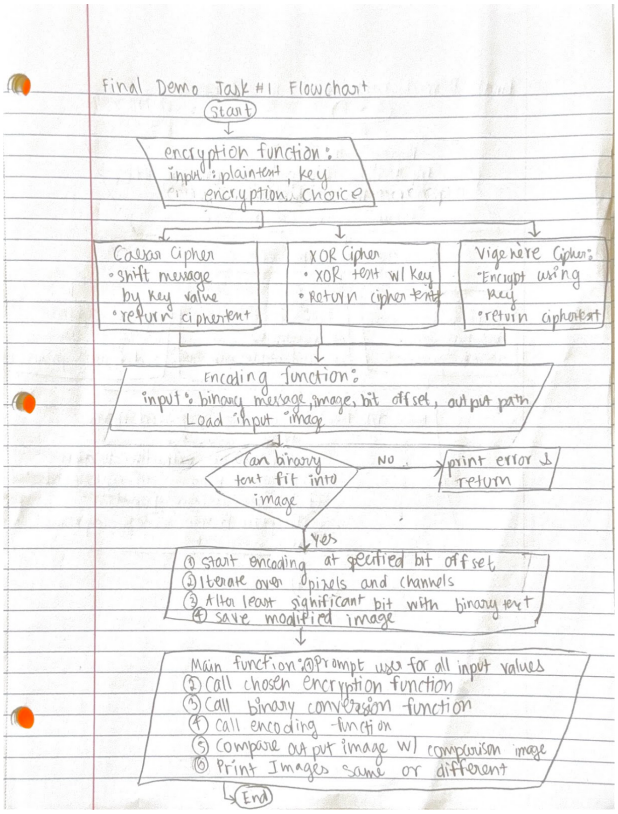
- Handles user input, calls the `cipher` and `text_to_binary` functions, and prints the output.

Design Rationale for Modularization:

- Separation of Concerns: The functions are well-defined, each handling a specific task.
- Reusability: The `cipher` and `text_to_binary` functions can be potentially reused in other projects.
- Maintainability: Changes can be made within individual functions without affecting the entire code.
- Testability: Each function can be tested independently.

Final Demo Task 1

Flowchart:



Purpose of Code:
This code combines text encryption techniques using various ciphers (XOR, Caesar, Vigenère), text-to-binary conversion, and image steganography. It allows the user to encrypt a message, convert it to binary (with start/end sequences), and then embed the binary message in an image using the Least Significant Bit (LSB) technique. It can compare the original and encoded images to visually identify differences and produce an output image outlining the identified differences.

Steps of Execution:

1. Import libraries: Imports matplotlib.pyplot, PIL.Image, and numpy for image processing and manipulation.
2. main function:
    - Prompts the user for cipher type, message, key, start/end sequences, bit offset, image paths (input and output), and image for comparison.
    - Based on the chosen cipher (xor, caesar, or vigenère), encrypts the message using the corresponding function.
    - Converts the encrypted message to binary format with added start/end sequences using text_to_binary.
    - Formats the binary message with spaces for readability.

- Calls encode_msg to embed the binary message into the input image, specifying the offset and output path.
- If successful, prints a confirmation message and calls compare_images to compare the original and encoded images.
- Based on the compare_images output, prints a message indicating if the images are the same or different.

1. Encryption Functions (xor, caesar, vigenère):

- These functions implement the chosen encryption algorithm (XOR, Caesar, or Vigenère) based on user input.
- Each function takes the message and key as input and returns the encrypted message.

2. text_to_binary(start, text, end):

- Converts the text message, start sequence, and end sequence into binary format.
- Concatenates the binary representations and returns the combined binary string.

3. encode_msg(binary_msg, input_path, encoded_path, offset):

- Takes the binary message, input image path, output path, and bit offset as input.
- Loads the image using PIL.Image.
- Checks if the image is grayscale or RGB to determine the maximum number of embeddable bits.
- Verifies if the message length exceeds the available bits in the image.
- Iterates through the image pixels, starting at the specified offset.
- For each pixel, clears the LSB and replaces it with the current bit from the binary message.
- Converts the modified pixel value back to an unsigned integer and updates the image data.
- Saves the modified image with the encoded message as the specified output path.
- Returns True on successful encoding, False otherwise (if the message is too long).

4. compare_images(img1, img2, img3_name):

- Takes the paths of two images (original and encoded) and the output path for the difference image as input.
- Loads both images using matplotlib.pyplot.
- Compares the dimensions and modes (grayscale vs RGB) of the images.
- If dimensions or modes differ, prints an error message and returns False.
- Iterates through each pixel of both images, comparing corresponding pixel values.
- For each pixel difference, creates a new pixel with a maximum value (white) in the difference image.
- For each pixel with the same value, creates a new pixel with a minimum value (black) in the difference image.
- Saves the generated difference image using plt.savefig.
- Returns True if the images are identical, False otherwise.

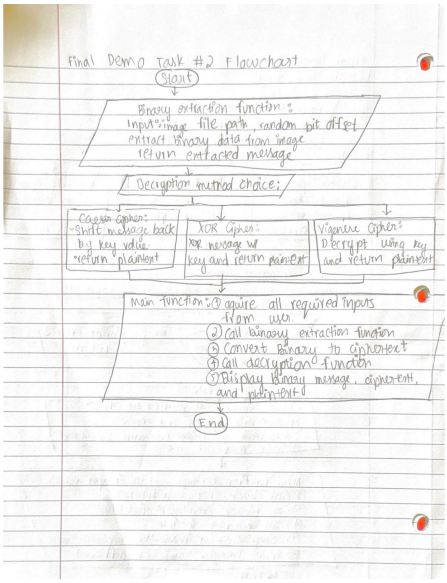
Modularization:

- Separation of Concerns: Each function handles a specific task, improving code clarity and maintainability.

- Reusability: The encryption functions, text_to_binary, and potentially compare_images can be reused in other projects.
- Testability: Modular structure allows for easier testing of individual functions.

Final Demo Task 2

Flowchart:



Purpose:

This code, just like task 1, combines text encryption techniques, text-to-binary conversion, image steganography (embedding binary data into an image), and decryption. It allows users to encrypt a message, embed it in an image, extract the embedded message, and then decrypt it using the corresponding cipher.

Steps of Execution:

1. **Import libraries:** Imports necessary libraries for image processing, binary conversion, and text manipulation.
2. **main function:**
    - Prompts the user for cipher type, key, start/end sequences, and image path.
    - Converts start/end sequences to binary.
    - Extracts the binary message from the image using extract_lsb.
    - Extracts the hidden message within the binary data using extract_message.
    - If a hidden message is found, converts it to text using binary_to_text.

- Decrypts the text message using the appropriate decryption function based on the chosen cipher.
- Prints the extracted message, converted text, and decrypted message.

Explanation of Functions:

1. Encryption Functions (xor, caesar, vigenere):

- These functions implement the chosen encryption algorithm (XOR, Caesar, or Vigenère).
- Each function takes the message and key as input and returns the encrypted message.

2. Decryption Functions (decrypted_xor, decrypted_caesar, decrypted_vigenere):

- These functions implement the corresponding decryption algorithms for each cipher.
- They take the encrypted message and key as input and return the decrypted message.

3. text_to_binary(text):

- Converts the given text string into a binary string.

4. extract_lsb(image_name):

- Extracts the least significant bit (LSB) from each pixel in the image, forming a binary string.

5. extract_message(binary_data, start_seq, end_seq):

- Extracts the hidden message within the binary data based on the specified start and end sequences.

6. binary_to_text(binary):

- Converts a binary string into text by interpreting 8-bit chunks as ASCII characters.

Rationale for Modularization:

- **Clear Separation of Concerns:** Each module focuses on a distinct aspect of the application, improving code organization and readability.

- **Improved Reusability:** Functions within modules can be reused in other projects or parts of the application, promoting efficiency.
- **Enhanced Maintainability:** Changes or bug fixes can be made within specific modules, reducing the risk of unintended side effects.
- **Easier Testing:** Individual modules can be tested independently, simplifying the testing process.
- **Scalability:** The modular structure makes it easier to add or remove features without affecting the entire codebase.

4. References

a. *Biomedical image processing*. (1981). PubMed. https://pubmed.ncbi.nlm.nih.gov/7023828/

Colantonio, S., Moroni, D., & Salvetti, O. (n.d.). *Image processing in biomedical applications*. https://si.isti.cnr.it/documents/tutorial-biomedical.pdf

b. GeeksforGeeks. (2024, July 18). *Vigenère cipher | Cryptography*. GeeksforGeeks. Retrieved October 18, 2024, from https://www.geeksforgeeks.org/vigenere-cipher/

c. Hassanpour, R. (2011). *Image processing techniques in biomedical engineering*. In S. Suh, V. Gurupur, & M. Tanik (Eds.), Biomedical engineering. Springer. https://doi.org/10.1007/978-1-4614-0116-2_14

d. Okta. (2024, August 30). *Steganography: The art of hiding in plain sight*. Okta. Retrieved October 18, 2024, from https://www.okta.com/identity-101/steganography/

e. Rajeswari, J. (2017, May 2). *Advances in biomedical signal and image processing – A systematic review*. Informatics in Medicine Unlocked. https://doi.org/10.1016/j.imu.2017.04.003

5. Appendix

1. User manual

Checkpoint 1

This file extracts a binary message encoded into a photo. The user should prepare an image file containing the encryption, and the start and end sequences of the encryption before running the program.

Install matplotlib and numpy before use.

How to use:
1. Enter the path of the image file containing the encryption
2. Enter the start and end sequences of the encryption

The program would provide the following output:
1. The extracted message, in binary
2. The binary message converted into text

Sample (Fig. 1. beau_and_bear_col_LC2_10_c.png):

Enter the path of the image you want to load: *beau_and_bear_col_LC2_10_c.png*
Enter the start sequence: *341*
Enter the end sequence: *&g@*
Below is the img_array output of beau_and_bear_col_LC2_10_c.png:
Extracted Message:
00110101001101110010000001111001011011100111001001101010011110000010000001001100011010100111001101111001011100010110101001110111001000000010000100100001000100001
Converted Text: 57 ynrjx Ljsyqjw !!!

Checkpoint 2 Task 1

This file compares two photos and outputs a photo displaying the differences. The user should prepare two photos to compare before running the program.

Install matplotlib before use.

How to use:

1. Enter the paths of the two image files to be compared
2. Enter the path of the output image

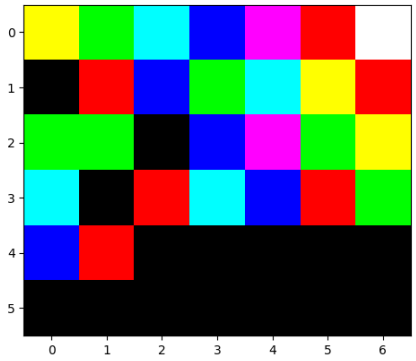
The program would provide the following output:

1. An image displaying the differences of the two photos
2. Whether the images are different or not

Sample (Fig. 2. ref_col_e.png, Fig. 3. ref_col.png):

Enter the path of your first image: *ref_col_e.png*
Enter the path of your second image: *ref_col.png*
Enter the path for the output image: *diff.png*
The images are different.

Output Image (Fig. 4. diff.png):



Checkpoint 2 Task 2 and 3:

This file encrypts the plaintext provided by the user using Vigenère cipher, converts the encrypted message to binary, and encodes the binary to an image. The user should prepare the plaintext and key, start and end sequence, and the photo they want the encryption to be encoded to before running the program.

Install Pillow and numpy before use.

How to use:

1. Enter the plaintext to be encrypted
2. Enter the key used for the Vigenère encryption
3. Enter the start and end sequences
4. Enter the path of the original image file
5. Enter the path of the output image

The program would provide the following output:

1. The encrypted message
2. The binary output message
3. The encoded image

Sample (Fig. 5. bear_col.png):

Enter the plaintext you want to encrypt: *Team 10 is impressive!!!*
Enter the key for Vigenere cipher: *rvlEDTXQmn*
Enter the start sequence: *117*
Enter the end sequence: *711*
Enter the path of the image: *bear_col.png*
Enter the path for the encoded image: *encrypted.png*
Encrypted Message using Vigenere Cipher: Kzlq 03 uf dxtuxpiuiv!!!
Binary output message: 00110001 00110001 00110111 01001011 01111010 01101100 01110001 00100000 00110000 00110011 00100000 01110101 01100110 00100000 01100100 01111000 01110100 01110101 01111000 01110000 01101001 01110101 01101001 01110110 00100001 00100001 00100001 00110111 00110001 00110001
Message successfully encoded and saved to: encrypted.png

Output Image (Fig. 6. encrypted.png):

Demo Task 1:

This file encrypts the plaintext provided by the user using either XOR, Vigenère, or Caesar cipher, converts the encrypted message to binary, and encodes the binary to an image after a particular bit offset. The output image is then compared to another image provided by the user. The user should prepare the plaintext and key, start and end sequences, the photo they want the encryption to be encoded to before running the program, and the photo to compare the output image to.

Install matplotlib, Pillow, and numpy before use.

How to use:

1. Enter the cipher to be used for encryption
2. Enter the plaintext to be encrypted
3. Enter the key used for the encryption (if caesar encryption is chosen, the key must be an integer)
1. Enter the start and end sequences
2. Enter the bit offset before the encoded message
3. Enter the path of the original image file
4. Enter the path of the output image
5. Enter the path of the image to be compared to

The program would provide the following output:

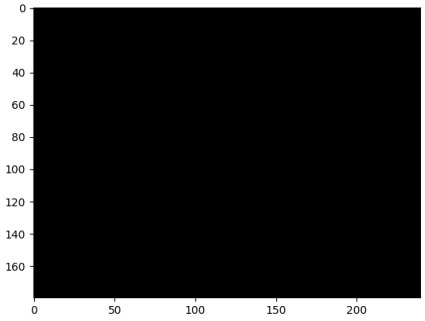
1. The encrypted message
2. The binary output message
3. The encoded image
4. The image displaying the difference between the compared images
5. The result of the comparison between the image provided and the output image

Sample (Fig. 7. beau_and_bear_col.png, Fig. 8. beau_and_bear_col_LC2_10_c.png) :

Enter the cipher you want to use for encryption: *caesar*
Enter the plaintext you want to encrypt: *57 times Gentler !!!*
Enter the key for the cipher: *31*
Enter the start sequence: *341*
Enter the end sequence: *&g@*
Enter the bit offset before you want to start encoding: *59*
Enter the path of the input image: *beau_and_bear_col.png*
Enter the path for your encoded image: *encrypted_c.png*

Enter the path of the image you want to compare: *beau_and_bear_col_LC2_10_c.png*
Encrypted Message using Caesar Cipher: 57 ynrjx Ljsyqjw !!!
Binary output message: 00110011 00110100 00110001 00110101 00110111 00100000 01111001 01101110 01110010 01101010 01111000 00100000 01001100 01101010 01110011 01111001 01110001 01101010 01110111 00100000 00100001 00100001 00100001 00100110 01100111 01000000
Message successfully encoded and saved to: encrypted_c.png.
The images are the same.

Output Image (Fig. 9. encrypted_c.png, Fig.10. diff.png) :



Demo Task 2:
This file extracts a binary message encoded into a photo, and decrypts the message to text using XOR, Vigenère, or Caesar cipher. The user should prepare an image file containing the encryption, and the start and end sequences, and the key of the encryption before running the program.

Install matplotlib and numpy before use.

How to use:

1. Enter the cipher to be used for decryption
2. Enter the key of the encryption (if caesar is chosen, the key must be an integer)
3. Enter the start and end sequences of the encryption
4. Enter the path of the image file containing the encryption

The program would provide the following output:

1. The extracted encrypted message, in binary
2. The binary encrypted message converted into text
3. The decrypted message

Sample (Fig. 11. beau_and_bear_col_LC2_10_c.png):

Enter the cipher you want to use for encryption: caesar
Enter the key for the cipher: 31
Enter the start sequence: 341
Enter the end sequence: &g@
Enter the path of the input image: REF_0_beau_and_bear\beau_and_bear_gry_LC2_10_c.png
Below is the img_array output of REF_0_beau_and_bear\beau_and_bear_gry_LC2_10_c.png:
Extracted Message:
00110101001101110010000001111001011011100111001001101010011110000010000001001100011010100111001101111001011100010110101001110111001000000010000100100001000100001
Converted Binary Text: 57 ynrjx Ljsyq

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

Lisette collaborated and worked on every function and task, answered questions, debugged the code, and offered help to every member. Lisette also compiled the overall structure (import statements) of each task file

Checkpoint 1 Task 1:
Display Images function: Lisette
Main Function: Tavish

Checkpoint 1 Task 2:
Text to Binary function: Lisette
Extract LSB function: Olivia
Extract message function: Divya
Main function: Tavish

Checkpoint 1 Task 3:
Binary to text function: Divya
Main function: Olivia

Checkpoint 2 Task 1:
Compare Images function: Lisette (50-70), Divya (35-49), Olivia (71-84)
Main function: Tavish

Checkpoint 2 Task 2:
Cipher function: Lisette, Tavish, and Divya
Text to Binary function: Olivia
Main: Tavish

Checkpoint 2 Task 3:
Encode message function: Lisette and Tavish
Vigenere function: Lisette, Divya, Tavish
Text to Binary function: Olivia
Main: Tavish

Final Demo Task 1:
XOR function: Lisette
Caesar function: Lisette
Vigenere function: Divya and Lisette
Encode message function: Lisette and Tavish
Text to Binary Function: Olivia
Compare images function: Lisette, Olivia, Divya
Main function: Tavish

Final Demo Task 2:
Decrypted XOR: Lisette
Decrypted Caesar: Lisette
Decrypted Vigenere: Divya and Lisette
Text to Binary function: Olivia
Extract LSB: Olivia
Extract message function: Divya
Binary to Text: Lisette
Main function: Tavish

Divya:
Created flowcharts for each task so that we could have an easier and more streamlined process while coding. For each part of the code that I wrote, I created comments every 2-3 lines that would help my team members understand the code when going over it. This helped create clarity. Additionally, I compiled together all of the data required for our final report to summarize our progress, challenges, and solutions.

Tavish:
I actively debugged code for all tasks and encouraged team discussions through probing questions, promoting deeper analysis and understanding of all parts of the code, including the parts they may not have worked on. By facilitating knowledge sharing and open dialogue, I helped build a cohesive team where everyone felt empowered to contribute and learn. This strengthened our problem-solving abilities and improved overall team performance. Additionally, I added comments to all tasks I worked on to enhance understanding for those who didn't write the code.

Lisette: I actively worked on and engaged with all aspects of the code, debugging components to ensure optimal performance. I was available to answer questions and provide assistance to team members, fostering collaboration. I compiled the final files, reviewing each one carefully while adding detailed comments to clarify key functions and enhancing existing documentation. This thorough approach improved the quality of our final product and promoted a culture of continuous learning within the team.

Olivia: I organized team meetings in order to facilitate open communication and collaboration. I added comments to the functions I worked on to help further the understanding of our code. I actively participated in discussions, and asked questions about the code, which encouraged deeper thinking and ensured that everyone was aligned on project goals. This collaborative approach helped strengthen our teamwork and improve the overall quality of our work.

Collaboration methods:

3. Ensuring that everyone attended all of the meetings, and actively participated.
4. Asking questions regarding code you may not have worked on to assure that all teammates understand all parts of the code.
5. Being understanding of each other's pre-decided time commitments.
6. Being flexible with meetings and everyone's availability.
7. Following our decided times and location for the team meetings
8. Delegating tasks in an evenly distributed manner to be more time efficient.
9. Met several times throughout a standard week to better manage the advanced workload.

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

Before initiating each checkpoint, our team came together to engage in thorough discussions. We carefully reviewed the directions, components, and deliverables for each task, ensuring that everyone had a clear understanding of the requirements. This collaborative approach allowed us to brainstorm various strategies and methods for tackling different areas of the checkpoint, fostering a creative environment where all ideas were valued.

Once we had a solid grasp of the tasks ahead, we delegated responsibilities based on each member's strengths and interests. While working on our individual assignments, we

prioritized maintaining open lines of communication. Regular check-ins helped us share updates on our progress, and we encouraged team members to ask for assistance or offer help as needed. This collaborative spirit not only kept us aligned but also enhanced our problem-solving capabilities.

To effectively manage our time and ensure we met our deadlines, we scheduled meetings multiple days a week well in advance. This proactive approach enabled us to allocate sufficient time for completing all required tasks to the best of our ability. Before each meeting, we collectively set specific goals for what we aimed to achieve. This focus on accountability helped keep everyone motivated and aligned with our overall objectives.

If we found ourselves falling short of our goals during a meeting, we took immediate action by organizing additional sessions to address any outstanding tasks or challenges. This flexibility allowed us to adapt our schedule as needed, ensuring that we remained on track and continued making progress toward our project milestones.

Overall, our commitment to communication, planning, and teamwork played a crucial role in our success as a group, enabling us to navigate challenges effectively and produce high-quality results.

11. Code

Checkpoint 1

```
# display string to binary format
def text_to_binary(text):
    s=''
    for i in text:
        s=s+format(ord(i), '08b')
    return s

# taking LSBs from the image
def extract_lsb(image_name):
    img = plt.imread(image_name)
    imagetype = img.dtype
    if imagetype == 'float32':
        img = (img * 255).astype('uint8')
    binary_data = ""

    # taking LSBs from each pixel
    for row in img:
        for pixel in row:
            if len(img.shape)<3:
                binary_data = binary_data + bin(pixel)[-1]
            else:
                r=pixel[0]
                g=pixel[1]
                b=pixel[2]
                binary_data = binary_data + bin(r)[-1]
```

```
                binary_data = binary_data + bin(g)[-1]
                binary_data = binary_data + bin(b)[-1]

    return binary_data

# extracting the message between the start and end sequences
def extract_message(binary_data, start_seq, end_seq):
    start_index = binary_data.find(start_seq)
    end_index = binary_data.find(end_seq, start_index + len(start_seq))

    if start_index == -1 or end_index == -1:
        return False

    else:
        return binary_data[start_index + len(start_seq):end_index]

# convert binary to ASCII text
def binary_to_text(binary):
    message = ''
    for i in range(0, len(binary), 8):
        byte = binary[i:i+8]
        message = message + chr(int(byte, 2))
    return message

def main():
    # enter path of image and start and end sequences
    image_name = input('Enter the path of the image you want to load:
')
    start_sequence_text = input('Enter the start sequence: ')
    end_sequence_text = input('Enter the end sequence: ')

    # convert the start and end sequences to binary
    start_sequence = text_to_binary(start_sequence_text)
    end_sequence = text_to_binary(end_sequence_text)

    # get binary data from image
    binary_data = extract_lsb(image_name)

    # get hidden message in binary
    hidden_message_bits = extract_message(binary_data, start_sequence,
end_sequence)

    print(f'Below is the img array output of {image_name}:')

    if hidden_message_bits != False:
        # convert hidden message from binary to text
        hidden_message=binary_to_text(hidden_message_bits)
        # output hidden message
        print(f'Extracted Message: {hidden_message_bits}')
        print(f'Converted Text: {hidden_message}')
    else:
```

```
        print('Start or end sequence not found in the image.')

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

Checkpoint 2 Task 1

```
import matplotlib.pyplot as plt

def compare_images(img1,img2,img3_name):
    output=True

    img1 = plt.imread(img1)
    imagetype = img1.dtype
    if imagetype == 'float32':
        img1 = (img1 * 255).astype('uint8')

    img2 = plt.imread(img2)
    imagetype = img2.dtype
    if imagetype == 'float32':
        img2 = (img2 * 255).astype('uint8')

    if len(img1.shape) == len(img2.shape):
        if img1.shape[0] == img2.shape[0] and img1.shape[1] ==
img2.shape[1]:
            img3=[]
            for row in range(img1.shape[0]):
                img3.append([])
                for pixel in range(img1.shape[1]):
                    if len(img1.shape)<3:
                        if img1[row][pixel] == img2[row][pixel]:
                            img3[row].append(0)
                        else:
                            img3[row].append(255)
                            output=False

                    else:
                        img3[row].append([])
                        for color in range(3):
                            if img1[row][pixel][color] ==
img2[row][pixel][color]:
                                img3[row][pixel].append(0)
                            else:
                                img3[row][pixel].append(255)
                                output=False
            plt.figure()
            if len(img1.shape) < 3:
                plt.imshow(img3, cmap='gray')
            else:
                plt.imshow(img3)
            plt.savefig(img3_name)
```

```
            plt.show()
        else:
            print('Cannot compare images of different sizes')
            output=False
    else:
        print('Cannot compare images in different modes (RGBA and L).')
        output=False
    return output

def main():
    img1=input('Enter the path of your first image: ')
    img2=input('Enter the path of your second image: ')
    img3=input('Enter the path for the output image: ')
    output=compare_images(img1,img2,img3)
    if output == True:
        print('The images are the same.')
    else:
        print('The images are different.')


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

Checkpoint 2 Task 2 and 3

```
from PIL import Image
import numpy as np

def encode_msg(binary_msg, input_path, output_path):
    image = np.array(Image.open(input_path))

    # Check if the image is grayscale (2D) or RGB (3D)
    if len(image.shape) == 2:  # Grayscale image
        max_bits = image.shape[0] * image.shape[1]
    else:  # RGB image
        max_bits = image.shape[0] * image.shape[1] * 3

    if max_bits < len(binary_msg):
        print('Given message is too long to be encoded in the image.')
        return

    index = 0
    if len(image.shape) == 2:  # Grayscale image
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                if index < len(binary_msg):
                    pixel = image[i, j]
                    new_pixel = (int(pixel) & ~1) |
int(binary_msg[index])
                    image[i, j] = np.uint8(new_pixel)
```

```
                    index += 1
                else:
                    break
    elif len(image.shape) == 3:  # RGB image
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                for color in range(3):  # RGB channels
                    if index < len(binary_msg):
                        # Get the pixel value (as uint8)
                        pixel = image[i, j, color]
                        # Replace the LSB with the message bit
                        new_pixel = (int(pixel) & ~1) |
int(binary_msg[index])
                        # Update the pixel value in the image array
                        image[i, j, color] = np.uint8(new_pixel)
                        index += 1
                    else:
                        break

    # Save the encoded image
    encoded_image = Image.fromarray(image)
    encoded_image.save(output_path)
    return True

def vigenere(message,key):
    encrypted=''
    a=0

    for i in range(len(message)):
        if a>=len(key):
            a=0

        if message[i].isalpha():
            if message[i].islower():
                shift='a'
            elif message[i].isupper():
                shift='A'
            new_char=chr((ord(message[i].upper())+ord(key[a].upper()))%
26+ord(shift))
        elif message[i].isnumeric():
            new_char = chr((ord(message[i])-
ord('0')+(ord(key[a].lower())-ord('a')))%10+ord('0'))
        else:
            new_char=message[i]

        encrypted+=new_char
        a+=1

    return encrypted

def text_to_binary(start,text,end):
    #Initializing empty strings for the binary
```

```
    text_b=''
    start_b=''
    end_b=''
    #Converting each character to binary (8 bits), also appending it to
each variable
    for i in text:
        text_b += format(ord(i), '08b')
    for i in start:
        start_b += format(ord(i), '08b')
    for i in end:
        end_b += format(ord(i), '08b')
    #Linking the start, text, and end strings
    s=start_b+text_b+end_b
    #Returns binary string
    return s


def main():
    #Define message, key, start, end, and image paths as user inputs
    message = input('Enter the plaintext you want to encrypt: ')
    key = input('Enter the key for Vigenere cipher: ')
    start = input('Enter the start sequence: ')
    end = input('Enter the end sequence: ')
    input_path = input('Enter the path of the image: ')
    output_path = input('Enter the path for the encoded image: ')
    #Define the result of our vigenere function as ciphered msg
    ciphered_msg=vigenere(message,key)
    #Call text to binary function and labeled result as binary msg
    binary_msg=text_to_binary(start,ciphered_msg,end)
    #Initializing formatted binary as an empty string
    formatted_binary = ''
    #Using a for loop to add the formatted binary message to
formatted_binary
    for i in range(0, len(binary_msg), 8):
        formatted_binary += binary_msg[i:i+8] + ' '
    encoded_msg=encode_msg(binary_msg,input_path,output_path)
    #Printing the outputs
    print(f'Encrypted Message using Vigenere Cipher: {ciphered_msg}')
    print(f'Binary output message: {formatted_binary}')
    #Print statement confirming message was encoded and listing where
if the encoded_msg ran successfully
    if encoded_msg==True:
        print(f'Message successfully encoded and saved
to:  {output_path}')


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

Demo Task 1

```
rt matplotlib.pyplot as plt
from PIL import Image
import numpy as np


def xor(message,key):
    # convert the message to binary
    message_b=''
    for i in message:
        # convert each character in the message to its ASCII code, and
then to binary
        message_b+=format(ord(i), '08b')

    # convert the key to binary
    key_b=''
    for j in key:
        # convert each character in the key to its ASCII code, and then
to binary
        key_b+=format(ord(j), '08b')

    # encrypt each message binary using XOR
    a=0 # index of key
    encrypted_msg_b='' # stores the encrypted message
    for i in range(len(message_b)):
        if a>=len(key_b): # resets key index to 0 if message index
exceeds key length
            a=0
        encrypted_msg_b += str(int(message_b[i]) ^ int(key_b[a])) # XOR
between the message bit and the key bit and convert it to a string
        a+=1

    # convert the binary to a text string
    encrypted_msg = ''
    for i in range(0, len(encrypted_msg_b), 8):
        byte = encrypted_msg_b[i:i+8] # 8 bits at a time
        encrypted_msg += chr(int(byte, 2)) # convert each byte to a
character
    return encrypted_msg


def caesar(message, key):
    # initialize values
    low_alph = 'abcdefghijklmnopqrstuvwxyz'
    up_alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    encrypted=''
    key=int(key) # convert key to integer

    # encrypt each message letter using caesar
    for letter in message:
        if letter in low_alph:
```

```
            new_letter=low_alph[(low_alph.index(letter)+key)%26] # find
shifted letter
        elif letter in up_alph:
            new_letter=up_alph[(up_alph.index(letter)+key)%26] # find
shifted letter
        else: # use the same character if it is not alphabetic
            new_letter=letter
        encrypted+=new_letter
    return encrypted


def vigenere(message, key):
    # Initialize an empty string to store the encrypted message
    encrypted = ''
    # Initialize a variable to cycle through the key and count the
index
    a = 0

    # Iterate through each character in the message
    for i in range(len(message)):
        # Reset the key counter if it reaches the end of the key
        if a >= len(key):
            a = 0

        # Check if the current character is alphabetic
        if message[i].isalpha():
            # Determine the shift based on whether the character is
lowercase or uppercase
            if message[i].islower():
                shift = 'a'
            elif message[i].isupper():
                shift = 'A'
            # Encrypt the alphabetic character using the Vigenere
cipher formula
            new_char = chr((ord(message[i].upper()) +
ord(key[a].upper())) % 26 + ord(shift)) #ord(shift) maps the new char
to the correct ascii value
        # Check if the current character is numeric
        elif message[i].isnumeric():
            # Encrypt the numeric character using a modified Vigenere
cipher for numbers
            new_char = chr((ord(message[i]) - ord('0') +
(ord(key[a].lower()) - ord('a'))) % 10 + ord('0')) #ord('0') maps the
new char to the correct ascii value
        # If the character is neither alphabetic nor numeric, leave it
unchanged
        else:
            new_char = message[i]

        # Add the encrypted (or unchanged) char to the result
        encrypted += new_char
        # Move to the next character in the key
```

```
        a += 1

    # Return the fully encrypted message
    return encrypted

def encode_msg(binary_msg,input_path,encoded_path,offset):
    image = np.array(Image.open(input_path)) # Import message
    # Check if the image is grayscale or RGB, find the maximum number
of bits
    if len(image.shape) == 2:  # Grayscale image
        max_bits = image.shape[0] * image.shape[1]
    else:  # RGB image
        max_bits = image.shape[0] * image.shape[1] * 3

    # Check if the total number of bits exceeds the maximum number of
bits
    if max_bits < len(binary_msg)+offset:
        print('Given message is too long to be encoded in the image.')
        return False

    index = 0 # count the index of the message bits

    if len(image.shape) == 2:  # Grayscale image
        row=offset//image.shape[1] # row after offset
        column=offset%image.shape[1] # column after offset
        for i in range(row,image.shape[0]):
            for j in range(column,image.shape[1]):

                if index < len(binary_msg): # check if all characters
of the message have been encoded
                    pixel = image[i, j] # set to current pixel

                    new_pixel = (int(pixel) & ~1) |
int(binary_msg[index])
                    # clear the least significant bit (LSB) of the
pixel using '& ~1' and then replace it with the current bit from
binary_msg at the specified index using '|', This embeds 1 bit of the
binary message into the LSB of the pixel.

                    image[i, j] = np.uint8(new_pixel) # save as uint8
integer to pixel
                    index += 1 # move to the next character
                else:
                    break
            column=0 # reset column after first row to create
indentation in the encrypted text

    elif len(image.shape) == 3:  # RGB image
            pixel_offset = offset // 3 # RGB channel after offset
            row = pixel_offset // image.shape[1] # row after offset
```

```
            column = pixel_offset % image.shape[1] # column after
offset
            bit = offset % 3

            for i in range(row, image.shape[0]):
                for j in range(column, image.shape[1]):
                    for color in range(bit, 3):  # each RGB channel

                        if index < len(binary_msg): # check if all
characters of the message has been encoded
                            pixel = image[i, j, color] # set to current
channel

                            new_pixel = (int(pixel) & ~1) |
int(binary_msg[index])
                            # clear the least significant bit (LSB) of
the pixel using '& ~1' and then replace it with the current bit from
binary_msg at the specified index using '|', This embeds 1 bit of the
binary message into the LSB of the pixel.

                            image[i, j, color] = np.uint8(new_pixel) #
save as uint8 integer to channel
                            index += 1
                        else:
                            break
                    bit = 0  # Reset color channel after first pixel
                column = 0  # Reset column after first row

    # Save the encoded image
    encoded_image = Image.fromarray(image)
    encoded_image.save(encoded_path)
    return True

def text_to_binary(start,text,end):
    #Initializing variables as an empty string
    text_b=''
    start_b=''
    end_b=''
    #Converting to ASCII code then binary
    for i in text:
        text_b += format(ord(i), '08b')
    for i in start:
        start_b += format(ord(i), '08b')
    for i in end:
        end_b += format(ord(i), '08b')
    #Combining the start, text, and end binary values to a string
    s=start_b+text_b+end_b

    return s
```

```
def compare_images(img1,img2,img3_name):
    # Initialize a boolean variable to track if the images are
identical
    output = True
    # Read the first image using matplotlib
    img1 = plt.imread(img1)
    # Get the data type of the image
    imagetype = img1.dtype
    # If the image is in float32 (decimal)format, convert it to uint8
(0-255 range)
    if imagetype == 'float32':
        img1 = (img1 * 255).astype('uint8')

    # Read the second image using matplotlib
    img2 = plt.imread(img2)
    # Get the data type of the image
    imagetype = img2.dtype
    # If the image is in float32 format, convert it to uint8 (0-255
range)
    if imagetype == 'float32':
        img2 = (img2 * 255).astype('uint8')

    # Checks if both images are in same mode (grayscale/rgb)
    if len(img1.shape) == len(img2.shape):
        # Check if both images have the same width and height
        if img1.shape[0] == img2.shape[0] and img1.shape[1] ==
img2.shape[1]:
            # Initialize an empty list to store the difference image
            img3 = []
            # Iterate through each row of the images
            for row in range(img1.shape[0]):
                # Add a new row to img3
                img3.append([])
                # Iterate through each pixel in the row
                for pixel in range(img1.shape[1]):
                    # Check if the image is grayscale (2D)
                    if len(img1.shape) < 3:
                        # Compare the pixel values of both images
                        if img1[row][pixel] == img2[row][pixel]:
                            # If pixels are the same, add black (0) to
img3
                            img3[row].append(0)
                        else:
                            # If pixels are different, add white (255)
to img3
                            img3[row].append(255)
                            # Set output to False as the images are
different
                            output = False
                    # If the image is color (3D)
                    else:
                        # Add a new pixel to img3
```

```
                        img3[row].append([])
                        # Iterate through each color channel (R, G, B)
                        for color in range(3):
                            # Compare the color values of both images
                            if img1[row][pixel][color] ==
img2[row][pixel][color]:
                                # If color values are the same, add
black (0) to img3
                                img3[row][pixel].append(0)
                            else:
                                # If color values are different, add
white (255) to img3
                                img3[row][pixel].append(255)
                                # Set output to False as the images are
different
                                output = False
            # Create a new figure for plotting
            plt.figure()
            # Check if the image is grayscale
            if len(img1.shape) < 3:
                # Display the difference image in grayscale
                plt.imshow(img3, cmap='gray')
            else:
                # Display the difference image in color
                plt.imshow(img3)
            # Save the difference image to a file
            plt.savefig(img3_name)
            # Display the plot
            plt.show()
        else:
            # Print an error message if the images have different sizes
            print('Cannot compare images of different sizes')
            # Set output to False as the images cannot be compared
            output = False
    else:
        # Print an error message if the images have different modes
(e.g., RGBA vs. L)
        print('Cannot compare images in different modes (RGBA and L).')
        # Set output to False as the images cannot be compared
        output = False
    # Return the result of the comparison (True if identical, False
otherwise)
    return output


def main():
    #Define needed variables
    cipher = input('Enter the cipher you want to use for encryption: ')
    message  = 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_path = input('Enter the path of the input image: ')
    encoded_path = input('Enter the path for your encoded image: ')
    image_compare = input('Enter the path of the image you want to
compare: ')

    #Determining which cipher the user wants to use out of xor, Caesar,
and Vigenere or printing error message
    if cipher == 'xor':
        encrypted_msg = xor(message,key)
        print(f"Encrypted Message using XOR Cipher: {encrypted_msg}")
    elif cipher == 'caesar':
        encrypted_msg = caesar(message,key)
        print(f"Encrypted Message using Caesar Cipher:
{encrypted_msg}")
    elif cipher == 'vigenere':
        encrypted_msg = vigenere(message,key)
        print(f"Encrypted Message using Vigenere Cipher:
{encrypted_msg}")
    else:
        print('Invalid cipher Input')
        return

    #Defining the output of text to binary fucnction as binary msg
    binary_msg=text_to_binary(start,encrypted_msg,end)

    #Formating binary to have the needed spaces between characters
using a loop and indexing
    formatted_binary = ''
    for i in range(0, len(binary_msg), 8):
        formatted_binary += binary_msg[i:i+8] + ' '
    print(f'Binary output message: {formatted_binary}')

    encoded_msg =
encode_msg(binary_msg,input_path,encoded_path,offset)

    #Printing the cofirmation message if encode msg ran successfully
    if encoded_msg==True:
        print(f'Message successfully encoded and saved to:
{encoded_path}.')
        compare=compare_images(image_compare,encoded_path,'diff.png')
        #Printing whether the images are the same or differnt based on
the output of compare function
        if compare == True:
            print('The images are the same.')
        else:
            print('The images are different.')
```

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

Demo Task 2

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


# display string to binary format

def decrypted_xor(message,key):
    # convert the message to binary
    message_b=''
    for i in message:
        # convert each character in the message to its ASCII code, and
then to binary
        message_b+=format(ord(i), '08b')

    # convert the key to binary
    key_b=''
    for j in key:
        # convert each character in the key to its ASCII code, and then
to binary
        key_b+=format(ord(j), '08b')

    # decrypt each message binary using XOR
    a=0 # index of key
    decrypted_msg_b=''
    for i in range(len(message_b)):
        if a>=len(key_b): # resets key index to 0 if message index
exceeds key length
            a=0
        decrypted_msg_b += str(int(message_b[i]) ^ int(key_b[a])) # XOR
between the message bit and the key bit
        a+=1

    # convert the binary to a text string
    decrypted_msg = ''
    for i in range(0, len(decrypted_msg_b), 8):
        byte = decrypted_msg_b[i:i+8] # 8 bits at a time
        decrypted_msg += chr(int(byte, 2)) # convert each byte to a
character
    return decrypted_msg


def decrypted_caesar(message, key):
    # initialize values
```

```
    low_alph = 'abcdefghijklmnopqrstuvwxyz'
    up_alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    decrypted=''
    key=int(key) # convert key to integer

    # decrypt each message letter using caesar
    for letter in message:
        if letter in low_alph:
            new_letter=low_alph[(low_alph.index(letter)-key)%26] # find
shifted letter
        elif letter in up_alph:
            new_letter=up_alph[(up_alph.index(letter)-key)%26] # find
shifted letter
        else: # use the same character if it is not alphabetic
            new_letter=letter
        decrypted+=new_letter
    return decrypted


def decrypted_vigenere(message, key):
    # Initialize an empty string to store the decrypted message
    decrypted = ''
    # Initialize a counter to keep track of the current position in the
key
    a = 0

    # Loop through each character in the message
    for i in range(len(message)):
        # Reset the key position counter if we've reached the end of
the key
        if a >= len(key):
            a = 0

        # Check if the current character is a letter
        if message[i].isalpha():
            # Determine the shift based on whether the letter is
lowercase or uppercase
            if message[i].islower():
                shift = 'a'
            elif message[i].isupper():
                shift = 'A'

            # Decrypt the letter using the Vigenère cipher formula
            # 1. Convert both message and key characters to uppercase
            # 2. Subtract the ASCII values and apply modulo 26
            # 3. Add the result to the appropriate shift ('a' or 'A')
            new_char = chr((ord(message[i].upper()) -
ord(key[a].upper())) % 26 + ord(shift))

        # Check if the current character is a number
        elif message[i].isnumeric():
```

```
            # Decrypt the number using a similar formula, but with
modulo 10
            # This allows for number shifting in the range 0-9
            new_char = chr((ord(message[i]) - ord('0') -
(ord(key[a].lower()) - ord('a'))) % 10 + ord('0'))

        # If the character is neither a letter nor a number, keep it
unchanged
        else:
            new_char = message[i]

        # Add the decrypted character to the result string
        decrypted += new_char
        # Move to the next character in the key
        a += 1

    # Return the fully decrypted message
    return decrypted


def text_to_binary(text):
    #Initializing variable as an empty string
    s=''
    ##Converting to ASCII code then binary
    for i in text:
        s=s+format(ord(i), '08b')
    return s

# taking LSBs from the image
def extract_lsb(image_name):
    img = plt.imread(image_name) #read image
    imagetype = img.dtype
    if imagetype == 'float32':
        img = (img * 255).astype('uint8') # convert float values from
image data types to unit8 integers
    binary_data = ""

    # taking LSBs from each pixel
    for row in img:
        for pixel in row:
            if len(img.shape)<3: #grayscale
                binary_data = binary_data + bin(pixel)[-1] #convert the
pixel integers to binary
            else:
                r=pixel[0]
                g=pixel[1]
                b=pixel[2]
                binary_data = binary_data + bin(r)[-1]
                binary_data = binary_data + bin(g)[-1]
                binary_data = binary_data + bin(b)[-1]

    return binary_data
```

```
# extracting the message between the start and end sequences
def extract message(binary data, start seq, end seq):
    #Finding and defining the start and end sequences in the
binary data
    start index = binary data.find(start seq)
    end index = binary data.find(end seq, start index + len(start seq))

    #Returning 'False' if the start or end is not found
    if start index == -1 or end index == -1:
        return False
    
    else:
        #Returning the values between but not including the start and
end squence
        return binary data[start index + len(start seq):end index]



# convert binary to ASCII text
def binary to text(binary):
    # Initialize an empty string to store the final message
    message = ''
    
    # Loop through the binary string, processing 8 bits at a time
    for i in range(0, len(binary), 8):
        # Extract a chunk of 8 bits (1 #read iamge byte) from the
binary string
        byte = binary[i:i+8]
        
        # Convert the 8-bit binary chunk to its decimal (integer)
equivalent that can be found in the ASCII table
        # Then, convert the integer to its corresponding string
character
        # Add the character to the final message string
        message = message + chr(int(byte, 2))
    
    # Return the complete message converted from binary to text
    return message

def main():
    # enter path of image,start and end sequences, cipher, and key
    cipher = input('Enter the cipher you want to use for encryption: ')
    key = input('Enter the key for the cipher: ')
    start sequence text = input('Enter the start sequence: ')
    end sequence text = input('Enter the end sequence: ')
    input file = input('Enter the path of the input image: ')
    
    # convert the start and end sequences to binary
    start sequence = text to binary(start sequence text)
    end sequence = text to binary(end sequence text)

    # get binary data from image
```

```
    binary_data = extract_lsb(input_file)

    # get hidden message in binary
    hidden_message_bits = extract_message(binary_data, start_sequence,
end_sequence)

    if hidden_message_bits != False:
        print(f'Below is the img array output of {input_file}:')
        # convert hidden message from binary to text
        hidden_message=binary_to_text(hidden_message_bits)
        # output hidden message
        print(f'Extracted Message: {hidden_message_bits}')
        print(f'Converted Binary Text: {hidden_message}')
        #Determining which cipher the user wants to use out of xor,
Caesar, and Vigenere or printing error message
        if cipher == 'xor':
            decrypted_msg=decrypted_xor(hidden_message, key)
        elif cipher == 'vigenere':
            decrypted_msg= decrypted_vigenere(hidden_message, key)
        elif cipher == 'caesar':
            decrypted_msg=decrypted_caesar(hidden_message, key)
        else:
            print('Invalid Cipher Type')
        print('Converted text:',decrypted_msg)
    else:
        print('Start or end sequence not found in the image.')


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

12.