|  |  |
| --- | --- |
|  |  |
| Lab Report |  |
|  |  |
|  | 04 Sep 2024 |
|  | Information SecurityM. Hamza Gohar |

Lab #1

# OBJECTIVE:

To encode and decode a group of digits using **Ceaser Cipher**

# DESCRIPTION:

The **Caesar cipher** is a simple encryption technique used in cryptography. It works by shifting each letter in the plaintext by a fixed number of positions in the alphabet.

Formulas:

For encryption

C= E (k,p) = (p+k) Mod 26 For decryption

P= D (k,C) = (C-k) Mod 26

# Code:

text=str(input("Enter the string that you want to convert")) key=int(input("Enter the key"))

EncodingDecoding=input("Select e for encoding and d for decoding") if (EncodingDecoding =="e"):

def encoding(text,key):

result=""

for i in range(len(text)):

char=text[i]

if char.isupper():

# for capital letters

result+=chr((ord(char) + key -65)%26+65) # ord() returns the unicode of a character

else:

#for small letters result+=chr((ord(char)+key -97)%26+97)

return result

print(encoding(text,key))

#

elif (EncodingDecoding =="d"):

def decoding(text,key):

result=""

for i in range(len(text)):

char=text[i]

if char.isupper():

# for capital letters

result+=chr((ord(char) - key -65)%26+65) # ord() returns the unicode of a character

else:

# for small letters

result+=chr((ord(char)- key -97)%26+97)

return result

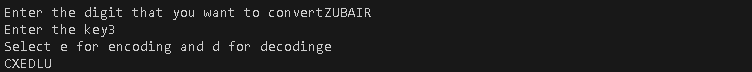
print(decoding(text,key))

else:

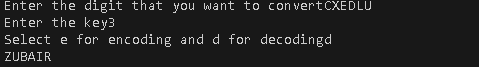
print("Plz select correct coding pattern")

# Output:

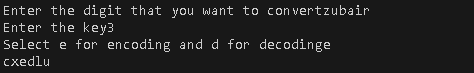
For capital letters encoding



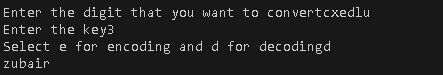
For capital letters decoding



For small letters encoding



For small letters decoding



Lab #2

*Objective:*

To convert plain text to cipher text using **Playfair Cipher**

Description:

* The **key word** is converted to a 5x5 matrix with no repeating words and I and J occupy the same space in the matrix
* There are no repeating words in that matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| M | O | N | A | R |
| C | H | Y | B | D |
| E | F | G | I/J | K |
| L | P | Q | S | T |
| U | V | W | X | Z |

* But here since we cant represent a 5x5 matrix using simple python we will store the keyword and remaining alphabets in a list (array)
* We will mimic the behavior of 5x5 matrix in array with the help of following formulas

**Row=index//5** // rounds down the numbers after decimal point in python

## **Column=index%5**

* In the code we will make all the keywords uppercase and replace I with J to keep consistency
* Now for **Plain Text** we need to create pairs for the whole sentence if there are any spaces those spaces will be excluded and the pairs will be made of 2 words
* If a letter repeats itself side by side we will add X to it

e.g. HELLO in this case LL => LX,LO

* If there are odd number of letters then we will add X to the letter remaining at the end

e.g. LOL => LO,LX

* Now locate the **pairs** we created from plain text in our array from upper formulas
* If they are in the same rows we will have to move right by using following formulas

## **Row1 \*5 + (column1+1)%5 Row2 \*5 + (column2+1)%5**

* If they are in the same column then we will move down by using following formulas

## **((Row1 +1)%5)\*5 + column1 ((Row1 +2)%5) \*5+ column2**

* For any other possibility **row1 \* 5 + col2 row12 \* 5 + col1**

# ***PROGRAM***

# Create the key square from the keyword def create\_key\_square(keyword):

keyword = keyword.upper().replace("J", "I") # J is usually replaced by I key\_square = []

used = []

# Add letters from keyword to the key square for char in keyword:

if char not in used and char.isalpha(): # Avoid duplicates and non-alphabet chars key\_square.append(char)

used.append(char)

# Add the rest of the letters (A-Z except J)

for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ": # 'J' is omitted if char not in used:

key\_square.append(char) return key\_square

# Prepare the text for encryption/decryption by making pairs of letters def prepare\_text(plain\_text):

plain\_text = plain\_text.upper().replace("J", "I") # Replace J with I pairs = []

i = 0

# Form letter pairs, inserting 'X' if needed while i < len(plain\_text):

a = plain\_text[i]

b = plain\_text[i + 1] if i + 1 < len(plain\_text) else 'X'

if a == b: # If both letters are the same, add an 'X' between them pairs.append(a + 'X')

i += 1

else:

pairs.append(a + b) i += 2

return pairs

# Find position of a letter in the key square def find\_position(char, key\_square):

index = key\_square.index(char) # Find the index of the character in the key square row = index // 5

col = index % 5 return row, col

# Encrypt or decrypt a pair of letters based on Playfair rules def process\_pair(pair, key\_square, encrypt=True):

row1, col1 = find\_position(pair[0], key\_square) row2, col2 = find\_position(pair[1], key\_square)

# Rule 1: Same row if row1 == row2:

if encrypt:

return key\_square[row1 \* 5 + (col1 + 1) % 5] + key\_square[row2 \* 5 + (col2 + 1)

% 5]

else:

return key\_square[row1 \* 5 + (col1 - 1) % 5] + key\_square[row2 \* 5 + (col2 - 1) %

5]

# Rule 2: Same column elif col1 == col2:

if encrypt:

return key\_square[((row1 + 1) % 5) \* 5 + col1] + key\_square[((row2 + 1) % 5) \* 5

+ col2]

else:

return key\_square[((row1 - 1) % 5) \* 5 + col1] + key\_square[((row2 - 1) % 5) \* 5

+ col2]

# Rule 3: Rectangle rule else:

return key\_square[row1 \* 5 + col2] + key\_square[row2 \* 5 + col1]

# Main encryption function

def playfair\_encrypt(plain\_text, keyword): key\_square = create\_key\_square(keyword) pairs = prepare\_text(plain\_text)

cipher\_text = '' for pair in pairs:

cipher\_text += process\_pair(pair, key\_square, encrypt=True) return cipher\_text

# Main decryption function

def playfair\_decrypt(cipher\_text, keyword): key\_square = create\_key\_square(keyword) pairs = prepare\_text(cipher\_text)

plain\_text = '' for pair in pairs:

plain\_text += process\_pair(pair, key\_square, encrypt=False) return plain\_text

# Get user input

keyword = input("Enter the keyword: ") plain\_text = input("Enter the plaintext: ")

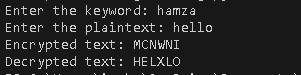
# Encrypt and display the result

cipher\_text = playfair\_encrypt(plain\_text, keyword) print("Encrypted text:", cipher\_text)

# Decrypt and display the result

decrypted\_text = playfair\_decrypt(cipher\_text, keyword) print("Decrypted text:", decrypted\_text)

# ***OUTPUT***



Lab#03

# Objective :

Encrypt and decrypt using hill cipher and vignere cipher

# Description:

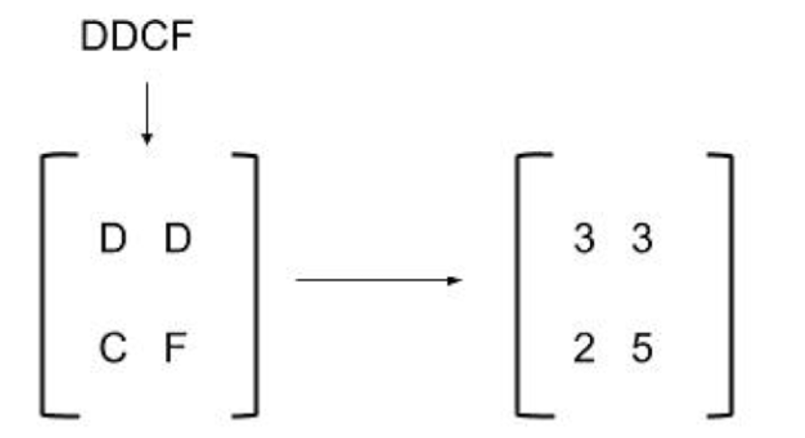
# Hill Cipher

Encrypting using the Hill cipher depends on the following operations −

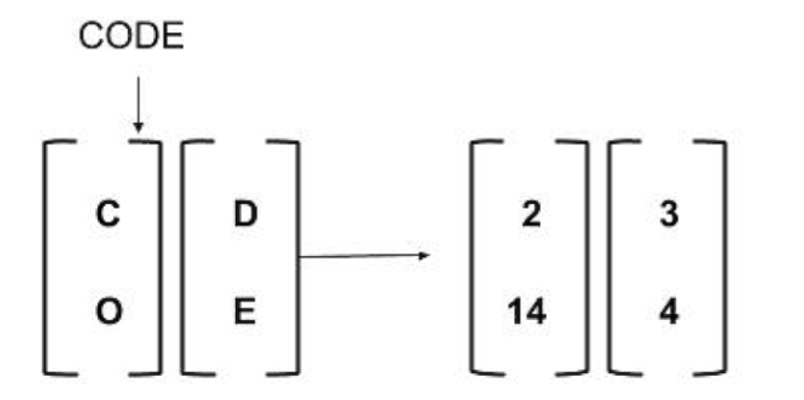
**E(K, P) = (K\*P) mod 26**

Here K is our key matrix, and P is the vectorized plaintext

Our key and plaintext are converted into numbers according to their index in the alphabet. For example



And vectors as



# Code:

# Encryption

# Function to convert a letter to a number (A=0, B=1, ..., Z=25)

def char\_to\_num(c):

    return ord(c) - ord('A')

# Function to convert a number to a letter (0=A, 1=B, ..., 25=Z)

def num\_to\_char(n):

    return chr(n + ord('A'))

def hill\_cipher\_encrypt(plaintext, key\_matrix):

    # Padding the plaintext to be a multiple of 3 with 'X'

    while len(plaintext) % 3 != 0:

        plaintext += 'X'

    ciphertext = ""

    # Process the plaintext in blocks of 3 characters

    for i in range(0, len(plaintext), 3):

        block = plaintext[i:i + 3]  # Get 3 characters at a time

        # Convert characters to numbers

        num\_block = [char\_to\_num(c) for c in block]

        # Encrypt using matrix multiplication with key matrix

        encrypted\_block = [

            (key\_matrix[0][0] \* num\_block[0] + key\_matrix[0][1] \* num\_block[1] + key\_matrix[0][2] \* num\_block[2]) % 26,

            (key\_matrix[1][0] \* num\_block[0] + key\_matrix[1][1] \* num\_block[1] + key\_matrix[1][2] \* num\_block[2]) % 26,

            (key\_matrix[2][0] \* num\_block[0] + key\_matrix[2][1] \* num\_block[1] + key\_matrix[2][2] \* num\_block[2]) % 26

        ]

        # Convert numbers back to characters and add to ciphertext

        ciphertext += ''.join([num\_to\_char(num) for num in encrypted\_block])

    return ciphertext

key\_matrix = [

     [6, 24, 1],

    [13, 16, 10],

    [20, 17, 15]

]

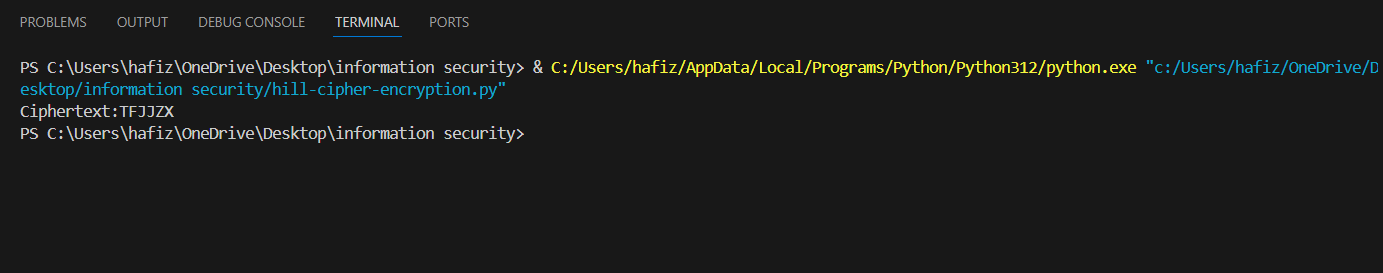
# Example usage

plaintext = "HELLO"

ciphertext = hill\_cipher\_encrypt(plaintext, key\_matrix)

print(f"Ciphertext:{ciphertext}")

# Output



# Decryption

import numpy as np

# Function to convert a letter to a number (A=0, B=1, ..., Z=25)

def char\_to\_num(c):

    return ord(c) - ord('A')

# Function to convert a number to a letter (0=A, 1=B, ..., 25=Z)

def num\_to\_char(n):

    return chr(n + ord('A'))

def mod\_inverse(a,mod):

    a=a%mod

    for x in range(1,mod):

        if (a\*x)%26==1:

            return x

def mod\_matrix\_inverse(key\_matrix, mod):

    matrix=np.array(key\_matrix)

    # Compute the determinant of the matrix

    det = int(round(np.linalg.det(matrix)))

    det\_inv = mod\_inverse(det, mod)  # Get the modular inverse of the determinant

    # Compute the adjoint of the matrix

    adj = (np.round(np.linalg.inv(matrix) \* det)).astype(int)  # Adjugate matrix

    # Multiply the adjoint by the modular inverse of the determinant

    matrix\_inv = (det\_inv \* adj) % mod

    return matrix\_inv.tolist()

def hill\_cipher\_decrypt(ciphertext, key\_matrix):

    plaintext = ''

    inverse\_key\_matrix=mod\_matrix\_inverse(key\_matrix,26)

# Process the plaintext in blocks of 3 characters

    for i in range(0, len(ciphertext), 3):

        block = ciphertext[i:i + 3]  # Get 3 characters at a time

        # Convert characters to numbers

        num\_block = [char\_to\_num(c) for c in block]

# decrypt using matrix multiplication with  inverse key matrix

        encrypted\_block = [

            (inverse\_key\_matrix[0][0] \* num\_block[0] + inverse\_key\_matrix[0][1] \* num\_block[1] + inverse\_key\_matrix[0][2] \* num\_block[2]) % 26,

            (inverse\_key\_matrix[1][0] \* num\_block[0] + inverse\_key\_matrix[1][1] \* num\_block[1] + inverse\_key\_matrix[1][2] \* num\_block[2]) % 26,

            (inverse\_key\_matrix[2][0] \* num\_block[0] + inverse\_key\_matrix[2][1] \* num\_block[1] + inverse\_key\_matrix[2][2] \* num\_block[2]) % 26

        ]

        # Convert numbers back to characters and add to plaintext

        plaintext += ''.join([num\_to\_char(num) for num in encrypted\_block])

    return plaintext

key\_matrix = [

     [6, 24, 1],

    [13, 16, 10],

    [20, 17, 15]

]

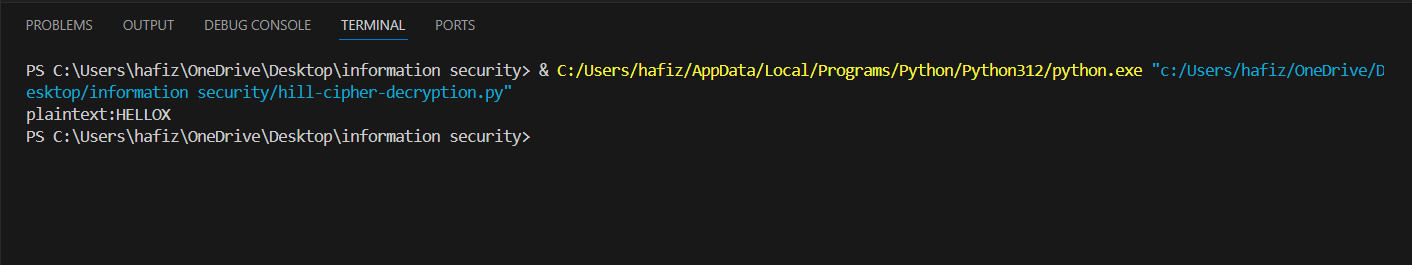
# Example usage

ciphertext = "TFJJZX"

plaintext = hill\_cipher\_decrypt(ciphertext, key\_matrix)

print(f"plaintext:{plaintext}")

# Output



# VIGENÈRE CIPHER

In this scheme, the 26 Caesar ciphers with shifts of 0 through 25 are used. Which cipher is to be applied at a specific point in text is determined by the key. Each cipher is denoted by a key letter, which is the ciphertext letter that substitutes for the plain text letter a. The encryption and decryption formulas for this cipher are

Ci = (pi + ki mod m) mod 26

pi = (Ci- ki mod m) mod 26

# Code

# Encryption

# Function to convert a letter to a number (A=0, B=1, ..., Z=25)

def char\_to\_num(c):

    return ord(c) - ord('A')

# Function to convert a number to a letter (0=A, 1=B, ..., 25=Z)

def num\_to\_char(n):

    return chr(n + ord('A'))

# Function to repeat the key to match the length of the plaintext

def repeat\_key(plaintext, key):

    repeated\_key = ""

    key\_length = len(key)

    for i in range(len(plaintext)):

        repeated\_key += key[i % key\_length]  # Repeat the key characters

    return repeated\_key

def vignere\_cipher\_encrypt(plaintext, key):

    plaintext = plaintext.upper()  # Ensure all letters are uppercase

    key = key.upper()  # Ensure the key is uppercase

    repeated\_key = repeat\_key(plaintext, key)  # Repeat the key

    ciphertext = ""

    # Encrypt each character in the plaintext

    for i in range(len(plaintext)):

        if plaintext[i].isalpha():  # Ignore non-alphabet characters

            # Shift the plaintext character by the key character

            shift = (char\_to\_num(plaintext[i]) +char\_to\_num(repeated\_key[i])) % 26

            ciphertext += num\_to\_char(shift)

        else:

            ciphertext += plaintext[i]  # Keep non-alphabet characters unchanged

    return ciphertext

# Example usage

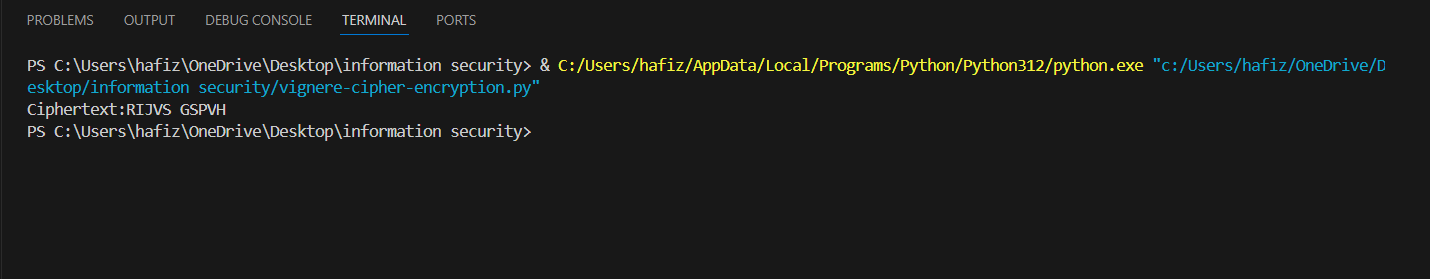
plaintext = "HELLO WORLD"

key = "KEY"

ciphertext = vignere\_cipher\_encrypt(plaintext, key)

print(f"Ciphertext:{ciphertext}")

# Output



# Decryption:

# Function to convert a letter to a number (A=0, B=1, ..., Z=25)

def char\_to\_num(c):

    return ord(c) - ord('A')

# Function to convert a number to a letter (0=A, 1=B, ..., 25=Z)

def num\_to\_char(n):

    return chr(n + ord('A'))

# Function to repeat the key to tongue the length of the ciphertext

def repeat\_key(ciphertext, key):

    repeated\_key = ""

    key\_length = len(key)

    for i in range(len(ciphertext)):

        repeated\_key += key[i % key\_length]  # Repeat the key characters

    return repeated\_key

def vignere\_cipher\_decrypt(ciphertext, key):

    ciphertext = ciphertext.upper()  # Ensure all letters are uppercase

    key = key.upper()  # Ensure the key is uppercase

    repeated\_key = repeat\_key(ciphertext, key)  # Repeat the key

    plaintext = ""

    # decrypt each character in the ciphertext

    for i in range(len(ciphertext)):

        if ciphertext[i].isalpha():  # Ignore non-alphabet characters

            # Shift the ciphertext character by the key character

            shift = (char\_to\_num(ciphertext[i]) -char\_to\_num(repeated\_key[i])) % 26

            plaintext += num\_to\_char(shift)

        else:

            plaintext += ciphertext[i]  # Keep non-alphabet characters unchanged

    return plaintext

# Example usage

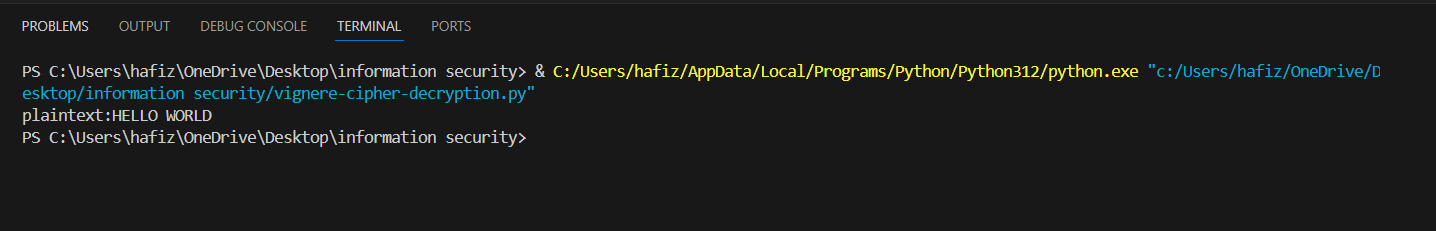
ciphertext = "RIJVS GSPVH"

key = "KEY"

plaintext= vignere\_cipher\_decrypt(ciphertext, key)

print(f"plaintext:{plaintext}")

# Output



Lab#04

# Objective:

* To encrypt and decrypt using rotor machine
* Use steganography to encode a message into the image and decode it

# Description(Rotor Machine):

* A **rotor** is basically a scrambled sequence of alphabets that is used to encode a text in a cyclic manner
* The letter to be encoded is mapped to the letter with the same index in the rotor . Rotor is then rotated one alphabet from its previous position and the resulting alphabet is the encoding .It may be kept in mind that as we encrypt letters in the plaintext the rotor moves(rotate) one unit backwards from its previous position for each encryption
* When two rotors are used the concept of a **notch** arises. A notch is a letter associated with a rotor. When the rotation count of the first rotor has not yet reached the index number of the notch the second encryption using the second rotor happens without rotation of the second rotor. However when the notch is reached the second rotor is triggered and starts rotating . Same is the case with three or four rotors . The notch for each rotor determines when the next rotor would start rotating.

# Code for encryption

class Rotor:

    def \_\_init\_\_(self, wiring, notch):

        self.wiring = wiring  # Scrambled alphabet (like 'EKMFLGDQVZNTOWYHXUSPAIBRCJ')

        self.notch = notch  # Notch position where the rotor will trigger the next rotor to rotate

        self.position = 0  # Initial rotor position (0 to 25)

    def rotate(self):

        # Rotates the rotor by one position

        self.position = (self.position + 1) % 26

    def encrypt\_letter(self, letter):

        # Encrypt the letter by adjusting it with the rotor's current position

        index = (ord(letter) - ord('A') + self.position) % 26

        encrypted\_letter = self.wiring[index]

        return encrypted\_letter

class SimpleRotorMachine:

    def \_\_init\_\_(self, rotors):

        self.rotors = rotors  # List of Rotor objects

    def rotate\_rotors(self):

        # Rotate the first rotor, and others if necessary (if the first reaches its notch)

        self.rotors[0].rotate()

        if self.rotors[0].position == ord(self.rotors[0].notch) - ord('A'):

            self.rotors[1].rotate()

    def encrypt\_message(self, message):

        encrypted\_message = ""

        for letter in message:

            if letter.isalpha():

                self.rotate\_rotors()

                letter = letter.upper()

                # Pass the letter through each rotor

                for rotor in self.rotors:

                    letter = rotor.encrypt\_letter(letter)

                encrypted\_message += letter

            else:

                encrypted\_message += letter  # Skip spaces or punctuation

        return encrypted\_message

# Example usage:

# Create rotors with wiring (scrambled alphabets) and notch positions

rotor1 = Rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ', 'Q')  # Rotor I with notch at Q

rotor2 = Rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE', 'E')  # Rotor II with notch at E

# Create the rotor machine with 2 rotors

simple\_rotor\_machine = SimpleRotorMachine([rotor1, rotor2])

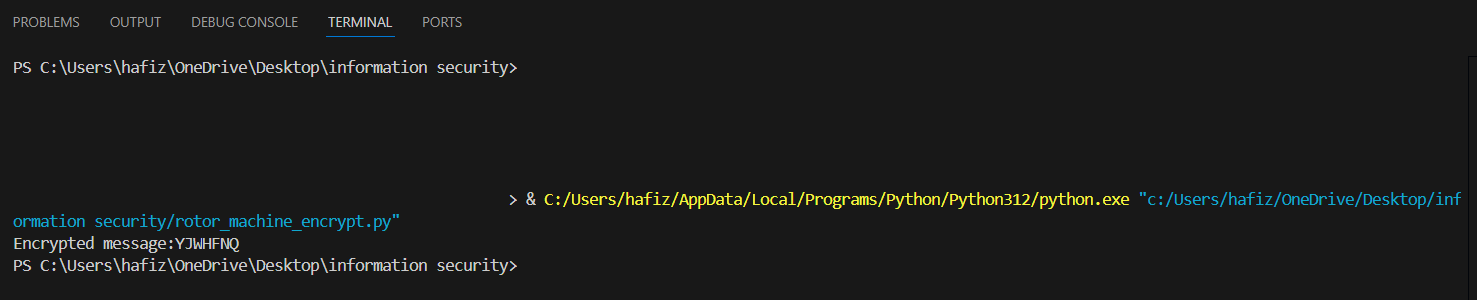
# Encrypt a message

message = "HUZAIFA"

encrypted\_message = simple\_rotor\_machine.encrypt\_message(message)

print(f"Encrypted message:{encrypted\_message}")

# Output



# Code for decryption

class Rotor:

    def \_\_init\_\_(self, wiring, notch):

        self.wiring = wiring  # Scrambled alphabet (like 'EKMFLGDQVZNTOWYHXUSPAIBRCJ')

        self.notch = notch  # Notch position where the rotor will trigger the next rotor to rotate

        self.position = 0  # Initial rotor position (0 to 25)

    def rotate(self):

        # Rotates the rotor by one position

        self.position = (self.position + 1) % 26

    def decrypt\_letter(self, letter):

        # decrypt the letter by adjusting it with the rotor's current position

        index = (self.wiring.index(letter)-self.position) % 26

        decrypted\_letter = chr(index+ord('A'))

        return decrypted\_letter

class SimpleRotorMachine:

    def \_\_init\_\_(self, rotors):

        self.rotors = rotors  # List of Rotor objects

    def rotate\_rotors(self):

        # Rotate the first  rotor, and others if necessary (if the first reaches its notch)

        self.rotors[1].rotate()

        if self.rotors[1].position == ord(self.rotors[1].notch) - ord('A'):

            self.rotors[0].rotate()

    def decrypt\_message(self, message):

        encrypted\_message = ""

        for letter in message:

            if letter.isalpha():

                self.rotate\_rotors()

                letter = letter.upper()

                # Pass the letter through each rotor

                for rotor in self.rotors:

                    letter = rotor.decrypt\_letter(letter)

                encrypted\_message += letter

            else:

                encrypted\_message += letter  # Skip spaces or punctuation

        return encrypted\_message

# Example usage:

# Create rotors with wiring (scrambled alphabets) and notch positions

rotor1 = Rotor('EKMFLGDQVZNTOWYHXUSPAIBRCJ', 'Q')  # Rotor I with notch at Q

rotor2 = Rotor('AJDKSIRUXBLHWTMCQGZNPYFVOE', 'E')  # Rotor II with notch at E

# Create the rotor machine with 2 rotors

simple\_rotor\_machine = SimpleRotorMachine([rotor2, rotor1])

# Encrypt a message

message = "YJWHFNQ"

decrypted\_message = simple\_rotor\_machine.decrypt\_message(message)

print(f"Encrypted message:{decrypted\_message}")

# Output

A black screen with yellow and green lines

Description automatically generated

# Stenography:

* **Steganography** is the practice of hiding secret information within another non-secret medium in such a way that the presence of the information is concealed.
* The most common form of steganography is hiding information in digital media, such as images, audio, or video files. The secret message is embedded in such a way that it doesn’t alter the appearance or quality of the media to the human eye or ear.
* One popular method of steganography is to hide information in images by manipulating the **Least Significant Bit (LSB)** of the pixel values in an image.
* Digital images consist of pixels, each represented by a set of bits (e.g., 8 bits for grayscale or 24 bits for RGB images). The idea behind LSB steganography is that changing the last (least significant) bit of a pixel’s color value only introduces a tiny, nearly imperceptible change in the color.

Effectively, the least significant bit of pixel color is changed in such a way that taken together they spell out the message in binary form

This is done by comparing the LSB of the pixel color with the current bit of the message if they are the same the bit is left unchanged however if they are different the LSB is changed to match the current bit

# Secret key

A secret key is a binary sequence that is appended at the end of the message. This key is only known by the receiver and the sender. When the receiver reaches the secret key in the process of decryption he knows that now the message has ended and beyond that point the pixel values do not represent message

# Implementation

In the following code the blue color value of pixels is changed in sequence until the whole message is embedded into the image . The secret key of 11111110 is used

# Code for embedding the message in the image

from PIL import Image

def encrypt\_message(input\_image\_path,ouput\_image\_path,message):

    image=Image.open(input\_image\_path)

    encoded\_image=image

    height,width=image.size

    binary\_message=''.join(format(ord(char),'08b') for char in message) +'11111110'

    pixels=list(encoded\_image.getdata())

    pixel\_index=0

    for value in binary\_message:

        pixel=list(pixels[pixel\_index])

        pixel[2]=pixel[2]& ~1|int(value)

        encoded\_image.putpixel((pixel\_index%height,pixel\_index//height),tuple(pixel))

        pixel\_index+=1

    encoded\_image.save(ouput\_image\_path)

#example

input\_image\_address="input\_image.png"

output\_image\_address="output\_image.png"

message="huzaifa"

encrypt\_message(input\_image\_address,output\_image\_address,message)

A tree with the sun shining through it

Description automatically generated

The above image is the input image



The above image is the output image

# Code for getting the message

from PIL import Image

def decrypt\_message(input\_image\_path):

    image=Image.open(input\_image\_path)

    height,width=image.size

    pixels=list(image.getdata())

    binary\_message=""

    for pixel in pixels:

        binary\_message+=str(pixel[2]&1)

    bytes\_message=[binary\_message[i:i+8]for i in range(0,len(binary\_message),8)]

    decrypted\_message=""

    for byte in bytes\_message:

        if byte=='11111110':

            break

        decrypted\_message +=chr(int(byte,2))

    return decrypted\_message

#example

#input\_image\_address="input\_image.png"

output\_image\_address="output\_image.png"

message=decrypt\_message(output\_image\_address)

print(message)

# Output

A black screen with yellow and green text

Description automatically generated

Lab#05

# Objective:

To encrypt and decrypt using fiestal cipher

# Description

## Encryption

In the fiestal cipher the byte to be encoded is split into two halves. These halves go through a specified number of rounds(4 in our example) and then are combined back together in the end which is the encrypted form. In each round the current right half is added with the key for that round and is XOR with the current left half .This new value is set as the new right half and the new left half is set to the current right half.

## Decryption

In the fiestal cipher decryption the cipher text goes through round but the roles of left and right halves are reversed. The order of keys is also reversed to get the correct answer

## Details

To get the left half we right shift the original byte by 8. To get the right half we AND the byte with the number FFh(in hexadecimal). To combine both halves at the end we simply left shift the left half and OR it with the right half

# Code for encryption and decryption

import random

# Round function: A simple function that operates on the right half and the key

# In this case, we add the right half to the key and take modulo 256 to keep it in 8-bit range

def round\_function(right\_half, key):

    return (right\_half + key) % 256

# Feistel Cipher Encryption

def feistel\_encrypt(plaintext, keys, num\_rounds=4):

    L = plaintext >> 8  # Left 8 bits

    R = plaintext & 0xFF  # Right 8 bits

    # Print the initial values of L and R for better understanding

    print(f"Initial Left Half (L): {bin(L)}")

    print(f"Initial Right Half (R): {bin(R)}")

    # Step 2: Perform the encryption for a specified number of rounds

    for i in range(num\_rounds):

        print(f"\n--- Round {i + 1} ---")

        # Save the current value of R to a temporary variable (temp) before swapping

        temp = R

          # Step 3: Apply the round function to the right half (R) and the round key (keys[i])

        # Then XOR the result with the left half (L), and store the result in the new R

        R= L^ round\_function(temp, keys[i])

        # Step 4: Swap the halves: set L to the old value of R (stored in temp)

        L= temp

        # Print the new values of L and R after this round

        print(f"Round {i + 1} Left Half (L): {bin(L)}")

        print(f"Round {i + 1} Right Half (R): {bin(R)}")

    # Step 5: Combine the two 8-bit halves (L and R) back into a single 16-bit ciphertext

    # L is shifted 8 bits to the left, and R is added to form the final 16-bit ciphertext

    ciphertext = (L << 8) | R

    # Return the resulting ciphertext

    return ciphertext

# Example usage of Feistel Cipher encryption

def feistel\_decrypt(ciphertext, keys, num\_rounds=4):

    L = ciphertext >> 8  # Left 8 bits

    R = ciphertext & 0xFF  # Right 8 bits

    # Print the initial values of L and R for better understanding

    print(f"Initial Left Half (L): {bin(L)}")

    print(f"Initial Right Half (R): {bin(R)}")

    # Step 2: Perform the decryption

    for i in range(num\_rounds):

        print(f"\n--- Round {i + 1} ---")

        # Save the current value of R to a temporary variable (temp) before swapping

        temp = L

          # Step 3: Apply the round function to the right half (R) and the round key (keys[i])

        # Then XOR the result with the left half (L), and store the result in the new R

        L= R^ round\_function(temp, keys[i])

        # Step 4: Swap the halves: set L to the old value of R (stored in temp)

        R= temp

        # Print the new values of L and R after this round

        print(f"Round {i + 1} Left Half (L): {bin(L)}")

        print(f"Round {i + 1} Right Half (R): {bin(R)}")

    # Step 5: Combine the two 8-bit halves (L and R) back into a single 16-bit plaintext

    # L is shifted 8 bits to the left, and R is added to form the final 16-bit plaintext

    plaintext = (L << 8) | R

    # Return the resulting ciphertext

    return plaintext

# A 16-bit example plaintext in binary

plaintext = 0b1101011101001010  # 16-bit binary input

# Generate random 8-bit round keys (for 4 rounds)

keys = [random.randint(0, 255) for \_ in range(4)] # Example keys

print(f"Plaintext: {bin(plaintext)}")  # Print the original plaintext

print("Encryption")

# Encrypt the plaintext using Feistel Cipher

ciphertext = feistel\_encrypt(plaintext, keys)

#reverse the keys for decryption

keys.reverse()

#decrypt the ciphertext

print("\nDecryption")

decoded\_plaintext=feistel\_decrypt(ciphertext,keys)

# Print the resulting ciphertext alongside with plaintext in binary

print(f"\nCiphertext: {bin(ciphertext)}")

print(f"decoded plaintext: {bin(decoded\_plaintext)}")

Output

