**Information Security Lab**



Information Security Lab,

Department of Computer Science,

School of Technology,

Pandit Deendayal Energy University

By:

Tirth Shah,

22BCP230

Under the guidance of:

Dr. Rutvij H. Jhaveri, Department of Computer Science, School of Technology, Pandit Deendayal Energy University

**Certificate**



This is to certify that **Tirth Shah,** student of **G6-Div3 CSE’26** with

enrolment number **22BCP230** has satisfactorily completed his work

in **Information Security Lab** under the guidance of **Dr. Rutvij H. Jhaveri.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_**

Lab Instructor Head of the department

Practical-1

Q1. Caesar Cypher and Improved Caesar Cypher

**Caesar Cypher: An Overview**

The Caesar Cypher, named after Julius Caesar who used it in his private correspondence, is a type of substitution Cypher where each letter in the plaintext is shifted a certain number of places down or up the alphabet. It is one of the simplest and most widely known encryption techniques.

**Key Characteristics:**

1. **Shift Value (Key):** The number of positions each letter in the plaintext is shifted. For example, with a shift of 3, A becomes D, B becomes E, and so on.
2. **Alphabet Wrap-Around:** The alphabet is treated as circular, so after Z comes A again. This means a shift of 1 on Z would result in A.
3. **Case Sensitivity:** Traditionally, the Cypher is case-sensitive, meaning 'A' and 'a' are considered distinct and are encrypted separately.

**Encryption Process:**

1. **Input:** A plaintext message and a shift value (key).
2. **Shift:** Each letter in the plaintext is shifted by the specified key. Non-alphabetic characters remain unchanged.
3. **Output:** The resulting Cyphertext.

**Decryption Process:**

1. **Input:** A Cyphertext message and the same shift value (key) used for encryption.
2. **Shift Back:** Each letter in the Cyphertext is shifted backward by the specified key to retrieve the original plaintext.
3. **Output:** The original plaintext message.

**Example:**

* **Plaintext:** HELLO
* **Key:** 3
* **Encryption:**
  + H (shift by 3) -> K
  + E (shift by 3) -> H
  + L (shift by 3) -> O
  + L (shift by 3) -> O
  + O (shift by 3) -> R
  + **Cyphertext:** KHOOR
* **Decryption:**
  + K (shift back by 3) -> H
  + H (shift back by 3) -> E
  + O (shift back by 3) -> L
  + O (shift back by 3) -> L
  + R (shift back by 3) -> O
  + **Plaintext:** HELLO

**Applications:**

* Historically used in military and government communication.
* Educational purposes to demonstrate basic encryption techniques.
* Simple puzzles and games for recreational cryptography.

**Limitations of the Caesar Cypher**

1. **Susceptibility to Brute-Force Attacks:**
   * With only 25 possible shifts, it is easy for an attacker to try all possible keys and decrypt the message.
2. **Frequency Analysis Vulnerability:**
   * The Cypher does not alter the frequency of letters, allowing attackers to use frequency analysis to break the encryption based on the known frequency of letters in the language.
3. **Lack of Complexity:**
   * The simplicity of the Cypher means it provides very little security and can be easily broken with minimal computational effort.
4. **Fixed Shift Key:**
   * The use of a single shift key for the entire message makes it easy to deCypher once the key is known.
5. **Not Suitable for Modern Communications:**
   * Given its weaknesses, the Caesar Cypher cannot protect sensitive information against modern cryptographic analysis and attacks.
6. **No Integrity or Authentication:**
   * The Cypher provides no mechanisms to ensure the integrity of the message or authenticate the sender, making it vulnerable to tampering and impersonation.

**Code:**

print("\nCaesar Cypher Encryption/Decryption\n")

choice = input("Enter the operation you want to perform: Encryption(1)/Decryption(0): ")

# Populating the alphabet table before hand without loop to avoid any overhead

alphabetTable = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9, 'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R': 17, 'S': 18, 'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}

reverseAlphabetTable = {v: k for k, v in alphabetTable.items()}

if choice == "1":

input\_message = input("\nEnter the message you want to encrypt: ")

key = int(input("\nEnter the encryption key you want to use: "))

encrypted\_message = ""

for c in input\_message:

if (65 <= ord(c) <= 90):

encrypted\_message += reverseAlphabetTable[(alphabetTable[c] + key) % 26]

elif (97 <= ord(c) <= 122):

temp = ord(c) - 32

encrypted\_message += reverseAlphabetTable[(alphabetTable[chr(temp)] + key) % 26].lower()

else:

encrypted\_message += c

print("\nEncrypted message: ", encrypted\_message, end="\n\n")

elif choice == "0":

input\_message = input("\nEnter the message you want to decrypt: ")

key = int(input("\nEnter the decryption key you want to use: "))

decrypted\_message = ""

for c in input\_message:

if (65 <= ord(c) <= 90):

decrypted\_message += reverseAlphabetTable[(alphabetTable[c] - key) % 26]

elif (97 <= ord(c) <= 122):

temp = ord(c) - 32

decrypted\_message += reverseAlphabetTable[(alphabetTable[chr(temp)] - key) % 26].lower()

else:

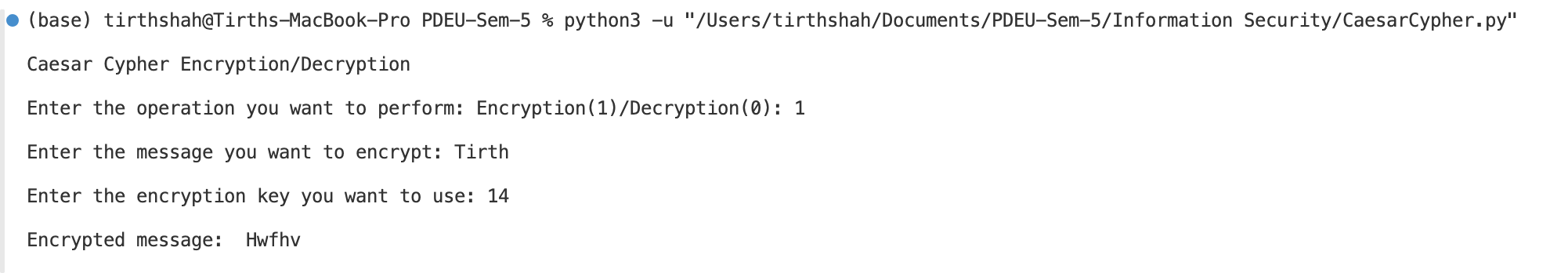
decrypted\_message += c

print("\nDecrypted message: ", decrypted\_message, end="\n\n")

else:

print("\nInvalid choice! Please enter 1 for encryption or 0 for decryption.")

**Output:**



**Improved Caesar Cypher: An Overview**

The Improved Caesar Cypher enhances the traditional Caesar Cypher by incorporating additional security measures, making it more robust against attacks. This version uses a keyword to create a variable shift pattern, combined with a simple hash function to determine the shift value, thereby increasing the complexity and security of the encryption process.

**Key Improvements:**

1. **Keyword-Length Adjustment:**
   * The keyword is adjusted to match the length of the input message, ensuring a consistent shift pattern throughout the entire message.
2. **Hash Function for Shift Value:**
   * A simple hash function based on the keyword generates a shift value, adding an extra layer of security and variability to the encryption process.

**Example:**

* **Plaintext:** HELLO
* **Keyword:** KEY
* **Key:** 3

**Step-by-Step Encryption Process:**

1. **Adjust the Keyword Length:**
   * The keyword "KEY" needs to be adjusted to match the length of the plaintext "HELLO".
   * Adjusted Keyword: "KEYKE"
   * This is done by repeating the keyword until it matches the length of the plaintext.
2. **Calculate the Shift Value:**
   * The shift value is calculated using a simple hash function based on the adjusted keyword and the provided key.
   * The hash value is the sum of the ASCII values of the characters in the keyword "KEYKE":
     + K: 75
     + E: 69
     + Y: 89
     + K: 75
     + E: 69
     + Hash Value = 75 + 69 + 89 + 75 + 69 = 377
   * The key is adjusted: Key = Key \* 17
     + Key = 3 \* 17 = 51
   * The shift value is calculated as: Hash Value % Key
     + Shift Value = 377 % 51 = 20
3. **Encrypt Each Character:**
   * Now, each character of the plaintext "HELLO" is shifted by the calculated shift value (20).
   * **H (shift by 20) -> B**
     + 'H' is at index 7 in the alphabet.
     + New index = (7 + 20) % 26 = 1
     + The character at index 1 is 'B'.
   * **E (shift by 20) -> Y**
     + 'E' is at index 4 in the alphabet.
     + New index = (4 + 20) % 26 = 24
     + The character at index 24 is 'Y'.
   * **L (shift by 20) -> F**
     + 'L' is at index 11 in the alphabet.
     + New index = (11 + 20) % 26 = 5
     + The character at index 5 is 'F'.
   * **L (shift by 20) -> F**
     + 'L' is at index 11 in the alphabet.
     + New index = (11 + 20) % 26 = 5
     + The character at index 5 is 'F'.
   * **O (shift by 20) -> I**
     + 'O' is at index 14 in the alphabet.
     + New index = (14 + 20) % 26 = 8
     + The character at index 8 is 'I'.
4. **Cyphertext:**
   * After shifting each character, the resulting Cyphertext is "BYFFI".

**Summary of Encryption:**

* **Plaintext:** HELLO
* **Adjusted Keyword:** KEYKE
* **Shift Value:** 20
* **Cyphertext:** BYFFI

**Step-by-Step Decryption Process:**

1. **Use the Same Adjusted Keyword and Shift Value:**
   * Adjusted Keyword: "KEYKE"
   * Shift Value: 20
2. **Decrypt Each Character:**
   * Now, each character of the Cyphertext "BYFFI" is shifted back by the calculated shift value (20).
   * **B (shift back by 20) -> H**
     + 'B' is at index 1 in the alphabet.
     + New index = (1 - 20 + 26) % 26 = 7
     + The character at index 7 is 'H'.
   * **Y (shift back by 20) -> E**
     + 'Y' is at index 24 in the alphabet.
     + New index = (24 - 20 + 26) % 26 = 4
     + The character at index 4 is 'E'.
   * **F (shift back by 20) -> L**
     + 'F' is at index 5 in the alphabet.
     + New index = (5 - 20 + 26) % 26 = 11
     + The character at index 11 is 'L'.
   * **F (shift back by 20) -> L**
     + 'F' is at index 5 in the alphabet.
     + New index = (5 - 20 + 26) % 26 = 11
     + The character at index 11 is 'L'.
   * **I (shift back by 20) -> O**
     + 'I' is at index 8 in the alphabet.
     + New index = (8 - 20 + 26) % 26 = 14
     + The character at index 14 is 'O'.
3. **Plaintext:**
   * After shifting each character back, the resulting plaintext is "HELLO".

**Summary of Decryption:**

* **Cyphertext:** BYFFI
* **Adjusted Keyword:** KEYKE
* **Shift Value:** 20
* **Plaintext:** HELLO

**Code:**

print("\nCaesar Cypher Encryption/Decryption\n")

choice = input("Enter the operation you want to perform: Encryption(1)/Decryption(0): ")

# Populating the alphabet table beforehand without loop to avoid any overhead

alphabetTable = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9, 'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R': 17, 'S': 18, 'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}

reverseAlphabetTable = {v: k for k, v in alphabetTable.items()}

def adjustLength(keyword, input\_message):

diff = len(input\_message) - len(keyword)

newKeyword = ""

if diff < 0:

for i in range(len(input\_message)):

newKeyword += keyword[i]

elif diff == 0:

newKeyword = keyword

else:

for i in range(len(input\_message)):

newKeyword += keyword[i % len(keyword)]

return newKeyword

def simpleHash(keyword, key):

hashValue = 0

for i in range(len(keyword)):

hashValue += ord(keyword[i])

key = key \* 17

return hashValue % key

def imporvedCaesarEncrypt(input\_message, key, keyword, alphabetTable, reverseAlphabetTable):

sameLengthKeyword = adjustLength(keyword, input\_message)

shiftValue = simpleHash(sameLengthKeyword, key)

encrypted\_message = ""

for c in input\_message:

if (65 <= ord(c) <= 90):

encrypted\_message += reverseAlphabetTable[(alphabetTable[c] + shiftValue) % 26]

elif (97 <= ord(c) <= 122):

temp = ord(c) - 32

encrypted\_message += reverseAlphabetTable[(alphabetTable[chr(temp)] + shiftValue) % 26].lower()

else:

encrypted\_message += c

return encrypted\_message

def imporvedCaesarDecrypt(input\_message, key, keyword, alphabetTable, reverseAlphabetTable):

sameLengthKeyword = adjustLength(keyword, input\_message)

shiftValue = simpleHash(sameLengthKeyword, key)

decrypted\_message = ""

for c in input\_message:

if (65 <= ord(c) <= 90):

decrypted\_message += reverseAlphabetTable[(alphabetTable[c] - shiftValue) % 26]

elif (97 <= ord(c) <= 122):

temp = ord(c) - 32

decrypted\_message += reverseAlphabetTable[(alphabetTable[chr(temp)] - shiftValue) % 26].lower()

else:

decrypted\_message += c

return decrypted\_message

if choice == "1":

input\_message = input("\nEnter the message you want to encrypt: ")

key = int(input("\nEnter the encryption key you want to use: "))

keyword = input("\nEnter the keyword you want to use: ")

encrypted\_message = imporvedCaesarEncrypt(input\_message, key, keyword, alphabetTable, reverseAlphabetTable)

print("\nEncrypted message: ", encrypted\_message, end="\n\n")

elif choice == "0":

input\_message = input("\nEnter the message you want to decrypt: ")

key = int(input("\nEnter the decryption key you want to use: "))

keyword = input("\nEnter the keyword you want to use: ")

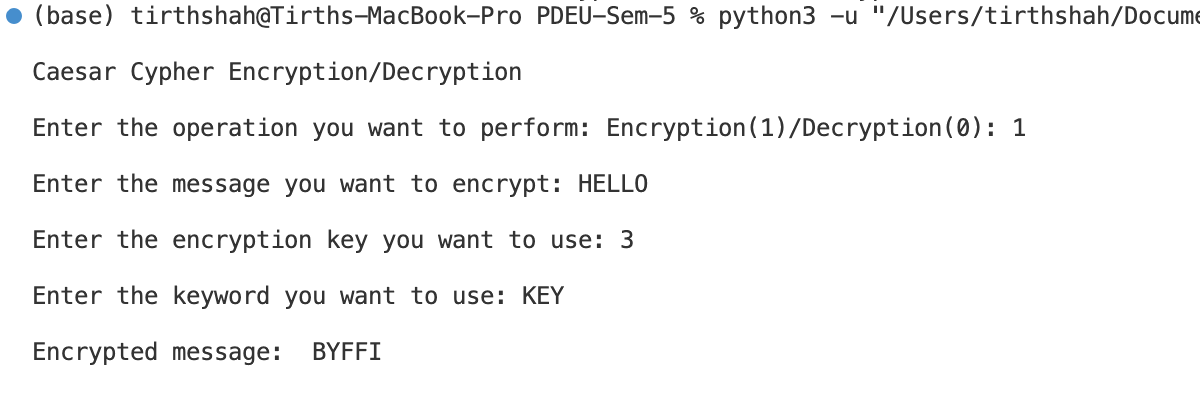
decrypted\_message = imporvedCaesarDecrypt(input\_message, key, keyword, alphabetTable, reverseAlphabetTable)

print("\nDecrypted message: ", decrypted\_message, end="\n\n")

else:

print("\nInvalid choice! Please enter 1 for encryption or 0 for decryption.")

**Output:**

****

**Conclusion:**

The traditional Caesar Cypher uses a fixed shift for encryption, making it simple but easily breakable. The Improved Caesar Cypher adds complexity by using a keyword-based hash to determine a variable shift, enhancing security. This added complexity makes it more resistant to basic attacks, though both Cyphers remain vulnerable due to the only 25 possible shifts for both of them.

Practical-2

**Q1. Write a program to implement normal playfair cipher and improvised playfair cipher**

**A1.**

**Normal Playfair Cipher**

The Playfair Cipher is a manual symmetric encryption technique and was the first literal digraph substitution cipher. The scheme was invented in 1854 by Charles Wheatstone but bears the name of Lord Playfair for promoting its use.

**Steps for Normal Playfair Cipher:**

1. **Key Matrix Generation**: Create a 5x5 matrix using a keyword. Remove duplicate letters from the keyword and fill the matrix with remaining letters of the alphabet. Traditionally, 'I' and 'J' are treated as the same letter.
2. **Prepare Plaintext**: Modify the plaintext to ensure it can be encrypted in pairs. If a pair of identical letters appear, insert an 'X' between them. If the plaintext has an odd number of characters, append an 'X' at the end.
3. **Encryption Rules**:
   * **Same Row**: Replace each letter with the letter immediately to its right (wrap around to the beginning if needed).
   * **Same Column**: Replace each letter with the letter immediately below it (wrap around to the top if needed).
   * **Rectangle**: Replace each letter with the letter in the same row but in the column of the other letter of the pair.

## Improved Playfair-Vigenère-Affine Cipher with Shuffling

### Introduction

In this lab, we implement an encryption and decryption system that combines the Playfair, Vigenère, and Affine ciphers, followed by a simple character shuffling step. This multi-layered approach enhances the security of the encryption process. Below, we detail the steps and functions involved in this encryption and decryption scheme.

### Playfair Cipher

The Playfair cipher is a manual symmetric encryption technique. It encrypts pairs of letters (digraphs), making it more secure than simple substitution ciphers.

#### Steps for Playfair Encryption:

1. **Matrix Generation**: A 5x5 matrix is generated using a key, skipping one letter (usually 'J').
2. **Input Modification**: The input message is modified to ensure there are no repeating characters in a pair, and 'X' is added if necessary.
3. **Pairwise Encryption**: Each pair of letters is encrypted based on their positions in the matrix.

#### Functions:

* getMat(key): Generates the Playfair matrix.
* modifyInput(input\_message): Modifies the input message for Playfair encryption.
* playFairEncrypt(input\_message, key): Encrypts the message using the Playfair cipher.
* playFairDecrypt(cypher, key): Decrypts the message using the Playfair cipher.

### Vigenère Cipher

The Vigenère cipher is a method of encrypting alphabetic text by using a simple form of polyalphabetic substitution.

#### Steps for Vigenère Encryption:

1. **Key Extension**: The key is extended to match the length of the message.
2. **Character-wise Encryption**: Each character of the message is encrypted using the corresponding character of the key.

#### Functions:

* vignereEncrypt(input\_message, key): Encrypts the message using the Vigenère cipher.
* vignereDecrypt(cypher, key): Decrypts the message using the Vigenère cipher.

### Affine Cipher

The Affine cipher is a type of monoalphabetic substitution cipher, where each letter in an alphabet is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter.

#### Steps for Affine Encryption:

1. **Parameters Selection**: Choose two keys, a and b, such that a is coprime with 26.
2. **Mathematical Transformation**: Apply the Affine transformation (a \* x + b) % 26 for encryption.

#### Functions:

* affineEncrypt(plaintext, a, b): Encrypts the message using the Affine cipher.
* affineDecrypt(ciphertext, a, b): Decrypts the message using the Affine cipher.
* mod\_inverse(a, m): Finds the modular inverse of a under modulo m.
* nextCoPrime(a): Finds the next coprime of a.

### Shuffling

A simple character shuffling step to further obfuscate the encrypted message.

#### Steps for Shuffling:

1. **Character Swap**: Swap every two characters in the string.

#### Function:

* shuffleTwo(cipher): Swaps every two characters in the string.

### Encryption Process

1. **Playfair Encryption**: Encrypt the input message using the Playfair cipher.
2. **Vigenère Encryption**: Encrypt the Playfair encrypted message using the Vigenère cipher.
3. **Affine Encryption**: Encrypt the Vigenère encrypted message using the Affine cipher.
4. **Shuffling**: Shuffle the characters of the Affine encrypted message.

### Decryption Process

1. **Unshuffling**: Reverse the shuffling step.
2. **Affine Decryption**: Decrypt the shuffled message using the Affine cipher.
3. **Vigenère Decryption**: Decrypt the Affine decrypted message using the Vigenère cipher.
4. **Playfair Decryption**: Decrypt the Vigenère decrypted message using the Playfair cipher and remove padding characters.

**Normal Cipher Code:**

import string

def getMat(key):

key = key.upper().replace("J", "I")

usedAlphas = set()

matList = []

# Add key characters to the matrix

for k in key:

if k not in usedAlphas and k in string.ascii\_uppercase:

usedAlphas.add(k)

matList.append(k)

# Add remaining characters to the matrix

for a in string.ascii\_uppercase:

if a not in usedAlphas and a != "J":

usedAlphas.add(a)

matList.append(a)

# Generate the 5x5 matrix

mat = [matList[i:i + 5] for i in range(0, 25, 5)]

return mat

def modifyInput(input\_message):

input\_message = input\_message.upper().replace(" ", "").replace("J", "I")

formatted\_message = ""

i = 0

while i < len(input\_message):

formatted\_message += input\_message[i]

if i + 1 < len(input\_message):

if input\_message[i] == input\_message[i + 1]:

formatted\_message += 'X'

i += 1

else:

formatted\_message += input\_message[i + 1]

i += 2

else:

formatted\_message += 'X'

i += 1

return formatted\_message

def findPosition(char, mat):

for i, row in enumerate(mat):

if char in row:

return i, row.index(char)

return None

def displayMat(mat):

print("\nMatrix: \n")

for row in mat:

print(" ".join(row))

print()

def playFairEncrypt(input\_message, key):

mat = getMat(key)

displayMat(mat)

modified\_input = modifyInput(input\_message)

encrypted = ""

i = 0

while i < len(modified\_input):

a = modified\_input[i]

b = modified\_input[i + 1]

row1, col1 = findPosition(a, mat)

row2, col2 = findPosition(b, mat)

if row1 == row2:

encrypted += mat[row1][(col1 + 1) % 5]

encrypted += mat[row2][(col2 + 1) % 5]

elif col1 == col2:

encrypted += mat[(row1 + 1) % 5][col1]

encrypted += mat[(row2 + 1) % 5][col2]

else:

encrypted += mat[row1][col2]

encrypted += mat[row2][col1]

i += 2

return encrypted

def playFairDecrypt(cypher, key):

mat = getMat(key)

displayMat(mat)

plain = ""

i = 0

while i < len(cypher):

a = cypher[i]

b = cypher[i + 1]

row1, col1 = findPosition(a, mat)

row2, col2 = findPosition(b, mat)

if row1 == row2:

plain += mat[row1][(col1 - 1) % 5]

plain += mat[row2][(col2 - 1) % 5]

elif col1 == col2:

plain += mat[(row1 - 1) % 5][col1]

plain += mat[(row2 - 1) % 5][col2]

else:

plain += mat[row1][col2]

plain += mat[row2][col1]

i += 2

return plain

print("\PlayFair Cypher Encryption/Decryption\n")

input\_message = input("\nEnter the message you want to encrypt: ")

key = input("\nEnter the encryption key you want to use: ")

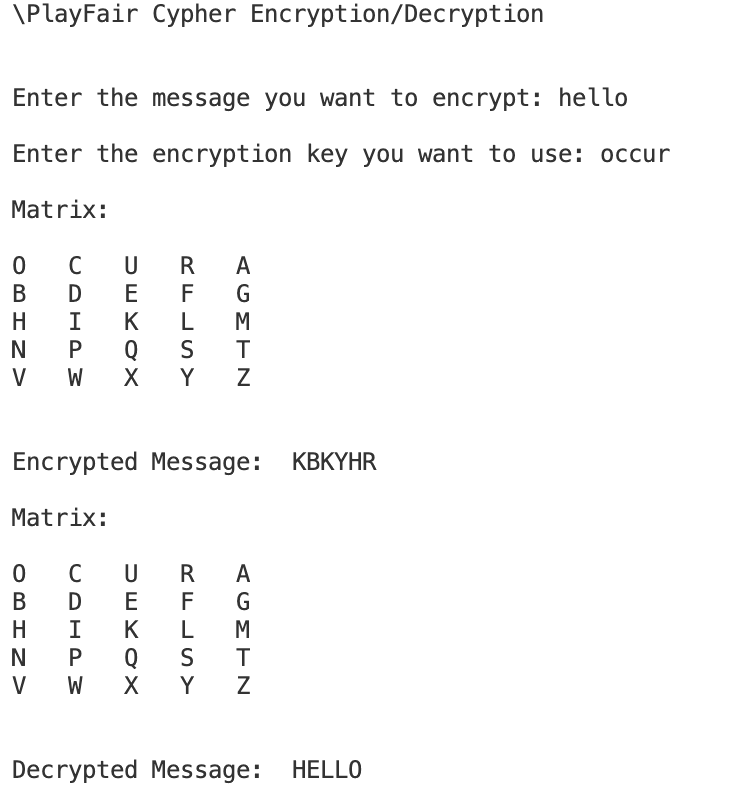
encrypted\_message = playFairEncrypt(input\_message, key)

print("\nEncrypted Message: ", encrypted\_message)

decrypted\_message = playFairDecrypt(encrypted\_message, key).replace("X", "")

print("\nDecrypted Message: ", decrypted\_message, "\n")

**Output:**

****

**Improvised Playfair Cipher Code:**

import math

def autoKeyGeneration(key, input\_message): # Generates key of the same length as the input message

key = list(key) # Convert key to list

if len(input\_message) == len(key): # If the key is the same length as the input message, return the key as is

return "".join(key)

elif len(input\_message) < len(key): # If the key is longer than the input message

return "".join(key[:len(input\_message)]) # Return the key truncated to the length of the input message

else: # If the key is shorter than the input message

for i in range(len(input\_message) - len(key)): # Append the key to itself until it is the same length as the input message

key.append(key[i % len(key)])

return "".join(key)

def getMat(key): # Generate the Playfair matrix

usedAlphas = set() # Set to keep track of used alphabets

matList = [] # List to store the matrix

skipped = False # Flag to check if a character has been skipped

skippedChar = 'X' # Skipped Character

replaced = False # Flag to check if a character has been replaced

replacedChar = 'X' # Replaced Character

mat = [] # Matrix

key = key.upper() # Convert key to uppercase

for k in key: # Iterate over the key

if k not in usedAlphas: # If the alphabet has not been used

usedAlphas.add(k) # Add the alphabet to the used alphabets set

matList.append(k) # Add the alphabet to the matrix list

alphabets = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

for a in alphabets: # Iterate over the alphabets

if a not in usedAlphas: # If the alphabet has not been used

if len(matList) >= 12 and not skipped: # If the matrix list has 12 or more elements and a character has not been skipped

skipped = True

skippedChar = a

continue

if not replaced and len(matList) >= 12: # If the matrix list has 12 or more elements and a character has not been replaced

replaced = True

replacedChar = a

usedAlphas.add(a)

matList.append(a)

for i in range(5): # Generate the matrix

l = [] # List to store the row

for j in range(5):

index = 5 \* i + j

l.append(matList[index]) # Append the element to the row

mat.append(l) # Append the row to the matrix

return mat, skippedChar, replacedChar

def modifyInput(input\_message): # Modify the input message to fit the Playfair cipher

input\_message = input\_message.upper().replace(" ", "") # Convert the input message to uppercase and remove spaces

formatted\_message = "" # Formatted message

i = 0 # Counter

while i < len(input\_message):

formatted\_message += input\_message[i] # Append the character to the formatted message

if i + 1 < len(input\_message): # If there is another character in the input message

if input\_message[i] == input\_message[i + 1]: # If the current character is the same as the next character

formatted\_message += 'X' # Append 'X' to the formatted message

i += 1 # Increment the counter

else:

formatted\_message += input\_message[i + 1] # Append the next character to the formatted message

i += 2 # Increment the counter by 2

else:

formatted\_message += 'X' # Append 'X' to the formatted message

i += 1 # Increment the counter

# Example: "HELLOO" -> "HELXLOOX"

return formatted\_message

def findPosition(a, mat): # Find the position of a character in the Playfair matrix

for i, row in enumerate(mat):

if a in row:

return i, row.index(a)

return None

def displayMat(mat): # Display the Playfair matrix

print("\nMatrix: \n")

for row in mat:

print(" ".join(row))

print()

def playFairEncrypt(input\_message, key):

mat, skippedChar, replacedChar = getMat(key) # Generate the Playfair matrix

displayMat(mat) # Display the Playfair matrix

modified\_input = modifyInput(input\_message.replace(skippedChar, replacedChar)) # Modify the input message

encrypted = ""

i = 0

while i < len(modified\_input):

a = modified\_input[i] # Get the first character

b = modified\_input[i + 1] # Get the second character

row1, col1 = findPosition(a, mat) # Find the position of the first character in the matrix

row2, col2 = findPosition(b, mat) # Find the position of the second character in the matrix

if row1 == row2: # If the characters are in the same row

encrypted += mat[row1][(col1 + 1) % 5] # Append the right character to the encrypted message

encrypted += mat[row2][(col2 + 1) % 5] # Append the right character to the encrypted message

elif col1 == col2: # If the characters are in the same column

encrypted += mat[(row1 + 1) % 5][col1] # Append the character below to the encrypted message

encrypted += mat[(row2 + 1) % 5][col2] # Append the character below to the encrypted message

else: # If the characters are in different rows and columns

encrypted += mat[row1][col2] # Append the character at the intersection to the encrypted message

encrypted += mat[row2][col1] # Append the character at the intersection to the encrypted message

i += 2

return encrypted

def playFairDecrypt(cypher, key):

mat, skippedChar, replacedChar = getMat(key)

displayMat(mat)

plain = ""

i = 0

while i < len(cypher):

a = cypher[i]

b = cypher[i + 1]

row1, col1 = findPosition(a, mat)

row2, col2 = findPosition(b, mat)

if row1 == row2:

plain += mat[row1][(col1 - 1) % 5] # Append the left character to the decrypted message

plain += mat[row2][(col2 - 1) % 5] # Append the left character to the decrypted message

elif col1 == col2:

plain += mat[(row1 - 1) % 5][col1] # Append the character above to the decrypted message

plain += mat[(row2 - 1) % 5][col2] # Append the character above to the decrypted message

else:

plain += mat[row1][col2]

plain += mat[row2][col1]

i += 2

return plain.replace(replacedChar, skippedChar) # Replace the skipped character with the original character

def vignereEncrypt(input\_message, key):

encrypted = ""

i = 0

input\_message = input\_message.replace(" ", "")

key = key.replace(" ", "")

input\_message = input\_message.upper()

while i < len(input\_message):

a = input\_message[i] # Get the character

b = key[i % len(key)] # Get the key character

encrypted += chr((ord(a) + ord(b) - 2 \* ord('A')) % 26 + ord('A')) # Encrypt the character

i += 1

return encrypted

def vignereDecrypt(cypher, key):

decrypted = ""

i = 0

cypher = cypher.replace(" ", "")

key = key.replace(" ", "")

cypher = cypher.upper()

while i < len(cypher):

a = cypher[i]

b = key[i % len(key)]

decrypted += chr((ord(a) - ord(b) + 26) % 26 + ord('A'))

i += 1

return decrypted

def mod\_inverse(a, m):

# Function to find the modular inverse of a under modulo 26

for x in range(1, 26):

if (a \* x) % 26 == 1:

return x

raise ValueError("No modular inverse found for a = {} and 26 = {}".format(a, 26))

def nextCoPrime(a):

# Function to find the next co-prime of a

for i in range(a + 1, 26):

if math.gcd(a, i) == 1:

return i

return 1

def affineEncrypt(plaintext, a, b):

a = nextCoPrime(a)

ciphertext = ''

for char in plaintext:

x = ord(char.upper()) - ord('A') # Convert the character to a number

encrypted\_char = (a \* x + b) % 26 # Encrypt the character

ciphertext += chr(encrypted\_char + ord('A')) # Convert the number to a character

return ciphertext

def affineDecrypt(ciphertext, a, b):

plaintext = ''

a = nextCoPrime(a)

a\_inv = mod\_inverse(a, 26) # Find the modular inverse of a

for char in ciphertext:

y = ord(char.upper()) - ord('A') # Convert the character to a number

decrypted\_char = (a\_inv \* (y - b)) % 26 # Decrypt the character

plaintext += chr(decrypted\_char + ord('A')) # Convert the number to a character

return plaintext

def shuffleTwo(cipher):

cipher = list(cipher) # Convert the cipher to a list

for i in range(0, len(cipher), 2):

if i + 1 < len(cipher):

cipher[i], cipher[i + 1] = cipher[i + 1], cipher[i] # Swap the characters

return "".join(cipher)

# Example: "EHLLO" -> "HELLO"

print("\nImproved PlayFair-Vigenère Cypher Encryption/Decryption\n")

input\_message = input("Enter the message you want to encrypt: ").upper()

key = input("Enter the encryption key: ").upper()

keyA = int(input("Enter the key A value: "))

keyB = int(input("Enter the key B value: "))

# Step 1: Playfair

pf\_encrypted = playFairEncrypt(input\_message, key)

# Step 2: Vigenère

vig\_encrypted = vignereEncrypt(pf\_encrypted, key)

# Step 3: Affine

affine\_encrypted = affineEncrypt(vig\_encrypted, keyA, keyB)

# Step 4: Shuffle

shuffled = shuffleTwo(affine\_encrypted)

print("\nEncrypted Message: ", shuffled)

# input\_message = input("Enter the message you want to decrypt: ").upper()

# key = input("Enter the decryption key: ").upper()

# Step 4: Shuffle

shuffled = shuffleTwo(shuffled)

# Step 3: Affine

affine\_decrypted = affineDecrypt(shuffled, keyA, keyB)

# Step 2: Vigenère

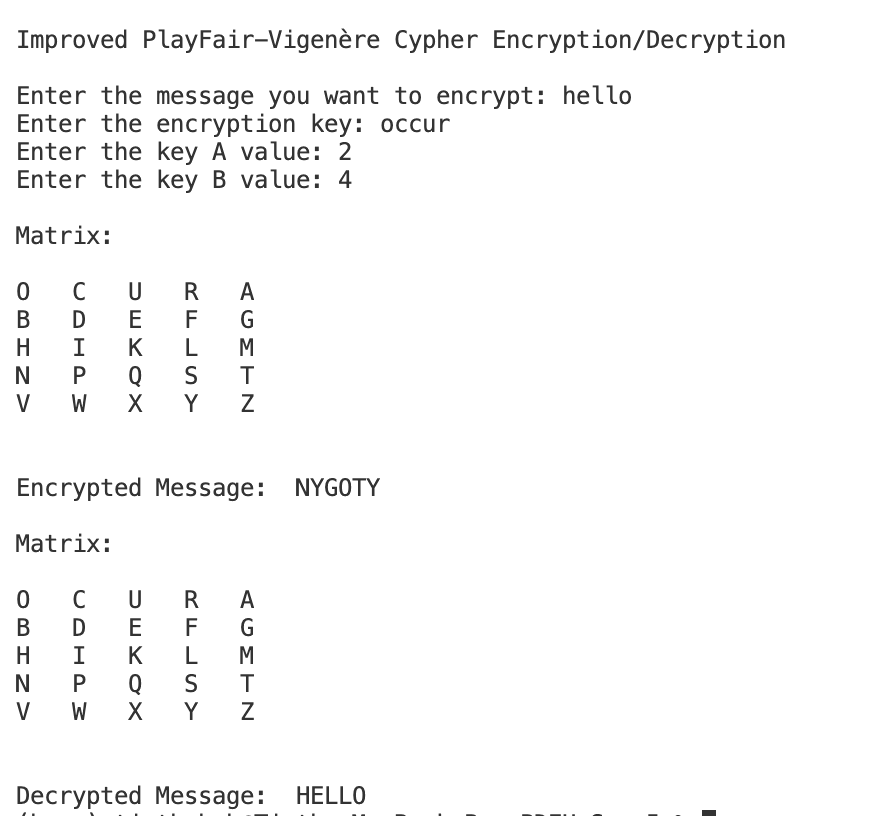
vig\_decrypted = vignereDecrypt(affine\_decrypted, key)

# Step 1: Playfair

final\_decrypted = playFairDecrypt(vig\_decrypted, key).replace('X', '')

print("\nDecrypted Message: ", final\_decrypted)

**Output:**

****