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:

```
(base) tirthshah@Tirths-MacBook-Pro PDEU-Sem-5 % python3 -u "/Users/tirthshah/Doc

Caesar Cypher Encryption/Decryption

Enter the operation you want to perform: Encryption(1)/Decryption(0): 1

Enter the message you want to encrypt: Tirth

Enter the encryption key you want to use: 14

Encrypted message:  Hwfhv
```

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): ")
```

```python
# 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):
```

```python
            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:

```
(base) tirthshah@Tirths-MacBook-Pro PDEU-Sem-5 % python3 -u "/Users/tirthshah/Docume

Caesar Cypher Encryption/Decryption

Enter the operation you want to perform: Encryption(1)/Decryption(0): 1

Enter the message you want to encrypt: HELLO

Enter the encryption key you want to use: 3

Enter the keyword you want to use: KEY

Encrypted message:  BYFFI
```

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 multilayered 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:

```
\PlayFair Cypher Encryption/Decryption


Enter the message you want to encrypt: hello

Enter the encryption key you want to use: occur

Matrix:

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


Encrypted Message:  KBKYHR

Matrix:

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


Decrypted Message:  HELLO
```

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()
```

```python
# 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:

```
Improved PlayFair-Vigenère Cypher Encryption/Decryption

Enter the message you want to encrypt: hello
Enter the encryption key: occur
Enter the key A value: 2
Enter the key B value: 4

Matrix:

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


Encrypted Message:  NYGOTY

Matrix:

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


Decrypted Message:  HELLO
```

Practical-3

Q1. Hill Cypher and Improved Hill Cypher

A1. Basic Hill Cipher

The Hill Cipher, invented by Lester S. Hill in 1929, is a polygraphic substitution cipher based on linear algebra. It encrypts blocks of text, rather than individual letters, making it more resistant to frequency analysis. The cipher uses a square key matrix to transform a block of plaintext into ciphertext.

Encryption:

1. **Key Matrix**: A square matrix of size `n x n` is chosen as the key. This matrix must be invertible modulo 26 (since the cipher typically works with the 26 letters of the English alphabet).
2. **Message Matrix**: The plaintext is divided into blocks of size `n`. If the length of the plaintext is not a multiple of `n`, it is padded with a filler character (commonly 'X').
3. **Matrix Multiplication**: Each block of plaintext is converted into a vector and multiplied by the key matrix. The resulting vector is then reduced modulo 26 to ensure it maps back to a valid character.
4. **Ciphertext**: The resultant vectors are converted back to letters to form the ciphertext.

Decryption:

1. **Inverse Key Matrix**: To decrypt the message, the inverse of the key matrix modulo 26 is required. This is only possible if the determinant of the key matrix is non-zero and has a multiplicative inverse modulo 26.
2. **Matrix Multiplication**: The ciphertext blocks are multiplied by the inverse key matrix, and the resulting vectors are reduced modulo 26 to retrieve the original plaintext.

Improved Hill Cipher

The Improved Hill Cipher enhances the basic version by introducing additional steps to make the cipher more secure.

Enhancements:

1. **Matrix Rotation**: The key matrix is first rotated to increase complexity. This rotation involves transposing the matrix and then reversing the elements in each row.
2. **Column Shifting**: After rotating the matrix, the columns of the matrix are shifted to the right. The amount of shifting is determined by the sum of the ASCII values of the key characters modulo the matrix size. This adds another layer of permutation, making the cipher harder to break.
3. **Encryption and Decryption**: Similar to the basic version, but with the added steps of rotating and shifting the key matrix before encrypting or decrypting the message. The inverse key matrix is also computed after applying these transformations.

These improvements increase the cipher's resistance to attacks by complicating the relationship between the plaintext and ciphertext.

Code:

```
import math # For math functions like sqrt, ceil

def copyMatrix(A):
    return [row[:] for row in A] # Copy the matrix row wise by copying the row
whole row using [:] slicing

def detRec(A, total=0): # Recursive function to calculate the determinant of a
matrix

    dimension = list(range(len(A))) # Get the dimension of the matrix that is
dimension

    if len(A) == 1 and len(A[0]) == 1:
        return A[0][0]  # If the matrix is of size 1x1 then return the only element

    if len(A) == 2 and len(A[0]) == 2:
        val = A[0][0] * A[1][1] - A[1][0] * A[0][1]
        return val # If the matrix is of size 2x2 then return the determinant of
the matrix

    for fc in dimension: # Iterate over all column where fc is the column in focus
```

```
        ASubForFocCol = copyMatrix(A) # Copy the matrix to a new matrix
ASubForFocCol
        ASubForFocCol = ASubForFocCol[1:] # Remove the first row of the matrix
        height = len(ASubForFocCol) # Row of the new matrix

        for i in range(height): # Iterate over all the rows of the new matrix
            a = ASubForFocCol[i][0:fc] # Get the elements of the row before the
column in focus
            b = ASubForFocCol[i][fc+1:] # Get the elements of the row after the
column in focus
            ASubForFocCol[i] = ASubForFocCol[i][0:fc] + ASubForFocCol[i][fc+1:] #
Remove the column in focus from the row

        sign = (-1) ** (fc) # Calculate the sign of the element in focus
        sub_det = detRec(ASubForFocCol) # Calculate the determinant of the new
matrix
        total += sign * A[0][fc] * sub_det # Add the product of the element in
focus and the determinant of the new matrix to the total

    return total

def getAdjointMatrix(mat):

    n = len(mat) # Get the dimension of the matrix
    adjointMat = []

    for i in range(0, n): # Iterate over all the rows of the matrix
        row = [] # Create a new row
        for j in range(0, n): # Iterate over all the columns of the matrix
            subMat = [] # Create a new matrix
            for k in range(0, n): # Iterate over all the rows of the matrix
                if k == i:  # If the row is the same as the row in focus then
                    continue
                temp = [] # Create a new row
                for l in range(0, n): # Iterate over all the columns of the matrix
                    if l == j: # If the column is the same as the column in focus
then
                        continue
                    temp.append(mat[k][l]) # Add the element to the row
                subMat.append(temp) # Add the row to the matrix
            row.append(detRec(subMat)) # Add the determinant of the matrix to the
row
        adjointMat.append(row) # Add the row to the matrix

    for i in range(0, n):
        for j in range(0, n):
            adjointMat[i][j] = ((-1) ** (i + j)) * adjointMat[i][j] # Calculate the
cofactor of the element

    # Transpose the matrix
```

```
    for i in range(0, n):
        for j in range(i, n):
            temp = adjointMat[i][j]
            adjointMat[i][j] = adjointMat[j][i]
            adjointMat[j][i] = temp

    return adjointMat

def getModularInverse(n):

    for i in range(26):
        if (n * i) % 26 == 1:
            return i
    return -1

def printMatrix(mat):

    n = len(mat)

    for i in range(0, n):
        for j in range(0, n):
            print(mat[i][j], end = " ")
        print()

def matMult(keyMat, messageMat):

    result = [] # Create a new matrix to store the result

    n = len(keyMat) # Get the dimension of the matrix

    temp0 = 0

    for i in range(0, n):
        temp = []
        temp0 = 0
        for j in range(0, n):
            temp0 += keyMat[i][j] * messageMat[j][0]    # We didn't use a third
loop because it is a column matrix so the column value will be 0
        temp.append(temp0 % 26) # Add the result to the row
        result.append(temp) # Add the row to the matrix

    return result

class HillCypher: # Class for Hill Cypher

    n = 0 # Dimension of the matrix

    def getKeyMatrix(self, key):

        lenOfKey = len(key)
```

```
        keyMat = [] # Create a new matrix to store the key

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

        keyList = list(key) # Convert the key to a list
        keyList = [ord(i) - ord('A') for i in keyList] # Convert the key to a list
of integers

        self.n = math.sqrt(lenOfKey) # Get the dimension of the matrix
        n = self.n # Get the dimension of the matrix

        nextSq = math.ceil(n) ** 2 # Get the next square number

        paddingValue = ord('X') - ord('A') # Get the padding value

        if len(keyList) < nextSq:
            keyList += [paddingValue] * (nextSq - len(keyList)) # Add the padding
value to the key

        n = int(math.sqrt(len(keyList))) # Get the dimension of the matrix

        for i in range(0, n):
            row = [] # Create a new row
            for j in range(0, n):
                row.append(keyList[i * n + j]) # Add the element to the row
            keyMat.append(row) # Add the row to the matrix

        return keyMat

    def getInverseKeyMatrix(self, key):

        keyMat = self.getKeyMatrix(key) # Get the key matrix
        det = detRec(keyMat) # Get the determinant of the matrix

        if det == 0:
            return None

        adjointMat = getAdjointMatrix(keyMat) # Get the adjoint matrix

        detInv = getModularInverse(det) # Get the modular inverse of the
determinant

        if detInv == -1:
            return None

        n = len(adjointMat)

        for i in range(0, n):
            for j in range(0, n):
```

```
                adjointMat[i][j] = (((adjointMat[i][j]) % 26) * detInv) % 26 #
Calculate the inverse of the matrix

        return adjointMat

    def getMessageMatrixList(self, key, message):

        lenOfMessage = len(message) # Get the length of the message
        n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2)) # Get the dimension
of the matrix

        if lenOfMessage % n != 0: # If the length of the message is not divisible
by the dimension of the matrix
            message = message.upper().replace(" ", "")
            message += "X" * (n - (lenOfMessage % n)) # Add the padding value

        messageMatList = [] # Create a new list to store the message

        messageList = list(message) # Convert the message to a list
        messageList = [ord(i) - ord('A') for i in messageList] # Convert the
message to a list of integers

        for i in range(0, len(messageList), n): # Iterate over the message with
skip of n indices
            mat = [] # Create a new matrix
            for j in range(0, n): # Iterate over the dimension of the matrix
                listInt = [messageList[i + j]] # Create a new list
                mat.append(listInt) # Add the list to the matrix
            messageMatList.append(mat) # Add the matrix to the list

        return messageMatList

    def encrypt(self, key, input_message):

        keyMat = self.getKeyMatrix(key) # Get the key matrix

        messageMatList = self.getMessageMatrixList(key, input_message) # Get the
message matrix list

        encryptedMessage = "" # Create a new string to store the encrypted message
        encryptedMessageList = [] # Create a new list to store the encrypted
message

        for messageMat in messageMatList: # Iterate over the message matrix list

            encryptedMat = matMult(keyMat, messageMat) # Multiply the key matrix
with the message matrix

            n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2)) # Get the sqrt
of next square number which will be the length after padding
```

```
            encryptedMessageList.append(encryptedMat) # Add the encrypted matrix to
the list

            for i in range(0, n):
                encryptedMessage += chr(encryptedMat[i][0] + ord('A')) # Add the
encrypted message to the string


        return encryptedMessageList, encryptedMessage

    def decrypt(self, key, input_message):

        keyInv = self.getInverseKeyMatrix(key) # Get the inverse key matrix

        if keyInv == None:
            return None

        messageMatList = self.getMessageMatrixList(key, input_message) # Get the
message matrix list

        decryptedMessage = ""

        for messageMat in messageMatList:

            decryptedMat = matMult(keyInv, messageMat) # Multiply the inverse key
matrix with the message matrix

            n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2)) # Get the sqrt
of next square number which will be the length after padding

            for i in range(0, n):
                decryptedMessage += chr(decryptedMat[i][0] + ord('A')) # Add the
decrypted message to the string

        # Remove the trailing 'X' characters

        while decryptedMessage[-1] == 'X':
            decryptedMessage = decryptedMessage[:-1]

        return decryptedMessage

class ImprovedHillCypher:

    def getKeyMatrix(self, key):

        lenOfKey = len(key)

        keyMat = []

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

```
        keyList = list(key)
        keyList = [ord(i) - ord('A') for i in keyList]

        self.n = math.sqrt(lenOfKey)
        n = self.n

        nextSq = math.ceil(n) ** 2

        paddingValue = ord('X') - ord('A')

        if len(keyList) < nextSq:
            keyList += [paddingValue] * (nextSq - len(keyList)) # Add the padding
value to the key

        n = int(math.sqrt(len(keyList))) # Get the dimension of the matrix

        # Make the matrix from list

        for i in range(0, n):
            row = []
            for j in range(0, n):
                row.append(keyList[i * n + j])
            keyMat.append(row)

        return keyMat

    def getInverseKeyMatrix(self, key):

        keyMat = self.getKeyMatrix(key) # Get the key matrix
        keyMat = self.rotateMatrix(keyMat) # Rotate the matrix
        keyMat = self.shiftColsRight(keyMat, key) # Shift the columns to the right
        det = detRec(keyMat) # Get the determinant of the matrix

        if det == 0:
            return None

        adjointMat = getAdjointMatrix(keyMat) # Get the adjoint matrix

        detInv = getModularInverse(det) # Get the modular inverse of the
determinant

        if detInv == -1:
            return None

        n = len(adjointMat) # Get the dimension of the matrix

        for i in range(0, n):
            for j in range(0, n):
                adjointMat[i][j] = (((adjointMat[i][j]) % 26) * detInv) % 26 #
Calculate the inverse of the matrix
```

```
        return adjointMat

    def rotateMatrix(self, matr):

        n = len(matr[0])

        # Transpose the matrix

        for i in range(0, n):
            for j in range(i, n):
                temp = matr[i][j]
                matr[i][j] = matr[j][i]
                matr[j][i] = temp

        # Reverse each row with two poninters

        for i in range(0, n):
            for j in range(0, n // 2):
                temp = matr[i][j]
                matr[i][j] = matr[i][n - j - 1]
                matr[i][n - j - 1] = temp

        return matr

    def shiftColsRight(self, matr, key):

        n = len(matr[0])

        sumOfKey = 0

        for i in key: # Get the sum of the ASCII values of the characters in the
key
            sumOfKey += ord(i)

        shift = sumOfKey % n # Get the shift value

        shifted_matrix = [[0] * n for _ in range(n)] # Initialize the shifted
matrix

        for i in range(0, n):
            for j in range(0, n):
                shifted_matrix[i][(j + shift) % n] = matr[i][j] # Shift the columns
to the right

        return shifted_matrix

    def getMessageMatrixList(self, key, message):

        lenOfMessage = len(message)
```

```
        n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2)) # Next square
number

        if lenOfMessage % n != 0: # Padding the message if the length is not
divisible by the dimension of the matrix
            message = message.upper().replace(" ", "")
            message += "X" * (n - (lenOfMessage % n))

        messageMatList = [] # Create a new list to store the message

        messageList = list(message) # Convert the message to a list
        messageList = [ord(i) - ord('A') for i in messageList] # Convert the
message to a list of integers in Z26

        for i in range(0, len(messageList), n):
            mat = [] # Create a new matrix
            for j in range(0, n):
                listInt = [messageList[i + j]] # Create a new list
                mat.append(listInt) # Add the list to the matrix
            messageMatList.append(mat) # Add the matrix to the list


        return messageMatList

    def encrypt(self, key, input_message):

        keyMat = self.getKeyMatrix(key)
        keyMat = self.rotateMatrix(keyMat)
        keyMat = self.shiftColsRight(keyMat, key)

        messageMatList = self.getMessageMatrixList(key, input_message)

        encryptedMessage = ""
        encryptedMessageList = []

        for messageMat in messageMatList:

            encryptedMat = matMult(keyMat, messageMat)

            n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2))

            encryptedMessageList.append(encryptedMat)

            for i in range(0, n):
                encryptedMessage += chr(encryptedMat[i][0] + ord('A'))


        return encryptedMessageList, encryptedMessage

    def decrypt(self, key, input_message):
```

```
        keyInv = self.getInverseKeyMatrix(key) # Get the inverse key matrix

        if keyInv == None:
            return None

        messageMatList = self.getMessageMatrixList(key, input_message) # Get the
message matrix list

        decryptedMessage = "" # Create a new string to store the decrypted message

        for messageMat in messageMatList:

            decryptedMat = matMult(keyInv, messageMat) # Multiply the inverse key
matrix with the message matrix

            n = int(math.sqrt(math.ceil(math.sqrt(len(key))) ** 2)) # Get the sqrt
of next square number which will be the length after padding

            for i in range(0, n):
                decryptedMessage += chr(decryptedMat[i][0] + ord('A')) # Add the
decrypted message to the string

        # Remove the trailing 'X' characters

        while decryptedMessage[-1] == 'X':
            decryptedMessage = decryptedMessage[:-1]

        return decryptedMessage


# Driver code


hill = HillCypher()

print("\nHill Cypher Encryption: ", end="")

enc = hill.encrypt("HILL", "HELLO")[1]

print(enc)

print("\nHill Cypher Decryption: ", end="")

dec = hill.decrypt("HILL", enc)

print(dec)

improvedHill = ImprovedHillCypher()

print("\nImproved Hill Cypher Encryption: ", end="")
```

```
enc = improvedHill.encrypt("HILL", "HELLO")[1]

print(enc)

print("\nImproved Hill Cypher Decryption: ", end="")

dec = improvedHill.decrypt("HILL", enc)

print(dec)
```

Output:

```
Hill Cypher Encryption: DRJIWR

Hill Cypher Decryption: HELLO

Improved Hill Cypher Encryption: PWQBNB

Improved Hill Cypher Decryption: HELLO
```