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): ")
```

```
# 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:

```
(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 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:

```
\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()
```

```
# 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
```

Practical-4

Q1. ImplementVigenère Cipher and Improvised Vigenère Cipher

A1. Vigenère Cipher Overview

The **Vigenère Cipher** is a method of encrypting alphabetic text by using a simple form of polyalphabetic substitution. It uses a keyword, where each letter of the keyword shifts corresponding letters of the plaintext by a number of positions in the alphabet. This cipher is more secure than the Caesar cipher as it uses multiple shifting patterns based on the keyword.

Working of Vigenère Cipher

1. **Encryption**:
    - The plaintext is aligned with the keyword, repeated or truncated as necessary to match its length.
    - Each letter in the plaintext is shifted by the position of the corresponding letter in the keyword (e.g., if the keyword letter is 'B', the plaintext letter is shifted by 1 position).
    - The result is the ciphertext.
2. **Decryption**:
    - The ciphertext is aligned with the keyword.
    - Each letter in the ciphertext is shifted backward by the position of the corresponding letter in the keyword.
    - The result is the original plaintext.

Improved Vigenère Cipher

The **Improved Vigenère Cipher** enhances the traditional Vigenère Cipher by introducing an additional step to modify the keyword:

1. **Key Modification**:
    - The original key is extended to match the length of the plaintext.
    - The key is then rearranged by taking the letters at specific intervals and reversing the sequence, making it harder to predict.

2. **Encryption & Decryption**:
    - The process follows similar steps to the standard Vigenère Cipher but uses the modified key, adding an extra layer of security.

Comparison

- **Standard Vigenère Cipher**: Simpler and easier to implement, but more vulnerable to frequency analysis attacks if the keyword is short or the ciphertext is long.
- **Improved Vigenère Cipher**: Offers enhanced security by making the keyword more complex, making it more resistant to attacks but slightly more complex to implement.

Both methods demonstrate the basic principles of polyalphabetic substitution, providing a foundation for understanding more advanced encryption techniques.

Input:

```
class VignereCipher:

    def __init__(self, key):
        self.key = key  # Initialize the key for the cipher

    def setKey(self, key):
        self.key = key  # Method to set a new key for the cipher

    def encrypt(self, plaintext):
        ciphertext = ""  # Initialize the ciphertext as an empty string
        for i in range(len(plaintext)):
            chNum = ord(plaintext[i]) - ord('A')  # Convert plaintext character to
0-25 range
            chNum += ord(self.key[i % len(self.key)]) - ord('A')  # Add
corresponding key character value
            chNum %= 26  # Wrap around if the sum exceeds 25
            ciphertext += chr(chNum + ord('A'))  # Convert back to a character and
append to ciphertext
        return ciphertext  # Return the encrypted text

    def decrypt(self, ciphertext):
        plaintext = ""  # Initialize the plaintext as an empty string
        for i in range(len(ciphertext)):
            chNum = ord(ciphertext[i]) - ord('A')  # Convert ciphertext character
to 0-25 range
            chNum -= ord(self.key[i % len(self.key)]) - ord('A')  # Subtract
corresponding key character value
            chNum %= 26  # Wrap around if the result is negative
```

```
            plaintext += chr(chNum + ord('A'))  # Convert back to a character and
append to plaintext
        return plaintext  # Return the decrypted text


class ImprovedVignereCipher:

    def __init__(self, key):
        self.key = key  # Initialize the key for the improved cipher

    def setKey(self, key):
        self.key = key  # Method to set a new key for the improved cipher

    def encrypt(self, plaintext):
        newKey = ""  # Initialize the new key as an empty string

        # Generate a new key that matches the length of the plaintext
        for i in range(len(plaintext)):
            newKey += self.key[i % len(self.key)]

        # Sort the new key in reverse order
        newKeyOne = ""
        for i in range(len(self.key)):
            j = i
            while j < len(newKey):
                newKeyOne += newKey[j]  # Rearrange the new key in reverse order
                j += len(self.key)

        newKey = ""
        for i in range(len(newKeyOne) - 1, -1, -1):
            newKey += newKeyOne[i]  # Reverse the newKeyOne to form the final
newKey

        ciphertext = ""  # Initialize the ciphertext as an empty string

        # Encrypt using the modified key
        for i in range(len(plaintext)):
            chNum = ord(plaintext[i]) - ord('A')  # Convert plaintext character to
0-25 range
            chNum += ord(newKey[i % len(newKey)]) - ord('A')  # Add corresponding
key character value
            chNum %= 26  # Wrap around if the sum exceeds 25
            ciphertext += chr(chNum + ord('A'))  # Convert back to a character and
append to ciphertext
        return ciphertext  # Return the encrypted text

    def decrypt(self, ciphertext):
        newKey = ""  # Initialize the new key as an empty string

        # Generate a new key that matches the length of the ciphertext
        for i in range(len(ciphertext)):
```

```
            newKey += self.key[i % len(self.key)]

        # Sort the new key in reverse order
        newKeyOne = ""
        for i in range(len(self.key)):
            j = i
            while j < len(newKey):
                newKeyOne += newKey[j]  # Rearrange the new key in reverse order
                j += len(self.key)

        newKey = ""
        for i in range(len(newKeyOne) - 1, -1, -1):
            newKey += newKeyOne[i]  # Reverse the newKeyOne to form the final
newKey

        plaintext = ""  # Initialize the plaintext as an empty string

        # Decrypt using the modified key
        for i in range(len(ciphertext)):
            chNum = ord(ciphertext[i]) - ord('A')  # Convert ciphertext character
to 0-25 range
            chNum -= ord(newKey[i % len(newKey)]) - ord('A')  # Subtract
corresponding key character value
            chNum %= 26  # Wrap around if the result is negative
            plaintext += chr(chNum + ord('A'))  # Convert back to a character and
append to plaintext

        return plaintext  # Return the decrypted text


# Main code to demonstrate the Vignere Cipher and Improved Vignere Cipher
print("Vignere Cipher\n")
key = input("Enter key: ")  # Get the key from user input
key = key.upper().replace(" ", "")  # Convert key to uppercase and remove spaces
vignereCipher = VignereCipher(key)  # Create a VignereCipher object
text = input("Enter text to encrypt: ")  # Get the plaintext from user input
text = text.upper().replace(" ", "")  # Convert plaintext to uppercase and remove
spaces
encryptedText = vignereCipher.encrypt(text)  # Encrypt the plaintext
print(f"\n\nEncrypted Text: {encryptedText}")  # Display the encrypted text
decryptedText = vignereCipher.decrypt(encryptedText)  # Decrypt the encrypted text
print(f"Decrypted Text: {decryptedText}")  # Display the decrypted text

print("\nImproved Vignere Cipher\n")
improvedVignereCipher = ImprovedVignereCipher(key)  # Create an
ImprovedVignereCipher object
encryptedText = improvedVignereCipher.encrypt(text)  # Encrypt the plaintext using
the improved cipher
print(f"Encrypted Text: {encryptedText}")  # Display the encrypted text from the
improved cipher
```

```
decryptedText = improvedVignereCipher.decrypt(encryptedText)  # Decrypt the
encrypted text from the improved cipher
print(f"Decrypted Text: {decryptedText}\n\n")  # Display the decrypted text from
the improved cipher
```

Output:

```
Vignere Cipher

Enter key: key
Enter text to encrypt: vignere


Encrypted Text: FMEXIPO
Decrypted Text: VIGNERE

Improved Vignere Cipher

Encrypted Text: TGKROBO
Decrypted Text: VIGNERE
```

Practical-5

Q1. Implement RailFence Cipher and Improvised RailFence Cipher

A1. **Rail Fence Cipher:**

The Rail Fence Cipher is a type of transposition cipher where the characters of the plaintext are arranged in a zigzag pattern across multiple "rails" or rows. The message is then read off row by row to create the ciphertext.

Steps in the Rail Fence Cipher:

1. **Matrix Construction:**
    - A matrix is created with a number of rows equal to the key (the number of rails) and columns equal to the length of the plaintext.
    - The plaintext is placed in the matrix in a zigzag pattern. The direction changes when the top or bottom of the matrix is reached.
2. **Encryption:**
    - Once the matrix is filled, the characters are read off row by row to generate the ciphertext.
3. **Decryption:**
    - The decryption process involves reversing the encryption steps.
    - The matrix is reconstructed by filling the zigzag pattern with placeholders and then replacing them with the characters from the ciphertext. The plaintext is then read in the zigzag order to retrieve the original message.

Code Explanation for Rail Fence Cipher:

- **Initialization:**
    - The class RailFence initializes with a key, which determines the number of rows (rails).
- **Matrix Creation (getRailFence method):**
    - The matrix is filled with the plaintext in a zigzag pattern using the key.
    - The dir_down boolean variable controls the direction of filling, switching direction at the top and bottom rails.
- **Encryption (encrypt method):**
    - The matrix is read row by row to create the ciphertext.
- **Decryption (decrypt method):**
    - The ciphertext is placed back into the matrix, filling the zigzag pattern.

- The plaintext is then reconstructed by reading the matrix in the zigzag order.

Improved Rail Fence Cipher:

The Improved Rail Fence Cipher is an extension of the basic Rail Fence Cipher with added complexity to increase security. In this version, the zigzag pattern is more elaborate, and the columns are shuffled based on a random seed generated from the key.

Steps in the Improved Rail Fence Cipher:

1. **Matrix Construction with Phases:**
   - The plaintext is placed in a matrix with the number of rows equal to the length of the plaintext and columns equal to the key.
   - The filling pattern is more complex, involving multiple phases that change the direction and column arrangement.
2. **Column Shuffling:**
   - After filling the matrix, the columns are shuffled randomly based on a seed derived from the key.
   - This step introduces an additional layer of complexity, making the ciphertext more secure.
3. **Encryption:**
   - The ciphertext is generated by reading the matrix column by column after the shuffle.
4. **Decryption:**
   - During decryption, the matrix is filled again in the shuffled column order.
   - The plaintext is reconstructed by reading the matrix in the order of the filling phases.

Code Explanation for Improved Rail Fence Cipher:

- **Initialization:**
  - The ImprovedRailFence class initializes with a key similar to the basic version.
- **Matrix Creation (getRailFence method):**
  - The matrix is filled with the plaintext in a more complex zigzag pattern involving different phases (phase_value controls the phase).
- **Column Shuffling:**
  - After constructing the matrix, columns are shuffled using Python's random.shuffle() with a seed set by the key.
- **Encryption (encrypt method):**

- The matrix is read in the shuffled column order to generate the ciphertext.

- **Decryption (decrypt method):**
  - The matrix is filled with the shuffled columns, and the plaintext is read by reversing the filling phases.

Conclusion:

The basic Rail Fence Cipher is a straightforward transposition cipher that rearranges the characters in a zigzag pattern, while the Improved Rail Fence Cipher adds complexity through multiple phases of filling and column shuffling, making it more secure. The improved version is harder to crack because of the randomization and the more intricate filling pattern.

Code:

```
import random

def printMat(mat):
    # Print the matrix row by row
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            if(mat[i][j] == "\t"):
                print("*\t", end="")
            else:
                print(mat[i][j], end="")
        print()

class RailFence:
    def __init__(self, key):
        self.key = key  # Initialize the key

    def setKey(self, key):
        self.key = key  # Set a new key

    def getRailFence(self, plaintext, key):
        # Create a matrix with 'key' rows and length of plaintext columns
        railFenceMat = [["\t" for i in range(len(plaintext))] for j in range(key)]

        dir_down = False  # Direction control
        row, col = 0, 0

        # Fill the matrix with characters in a zigzag pattern
        for i in range(len(plaintext)):
            if row == 0 or row == key - 1:
                dir_down = not dir_down  # Change direction at the top or bottom

            railFenceMat[row][col] = plaintext[i] + "\t"
            col += 1
```

```
            if dir_down:
                row += 1
            else:
                row -= 1

        return railFenceMat

    def encrypt(self, plaintext):
        # Get the Rail Fence matrix for the given plaintext
        railFenceMat = self.getRailFence(plaintext, self.key)

        ciphertext = ""

        # Read the matrix row by row to get the encrypted text
        for i in range(self.key):
            for j in range(len(plaintext)):
                if railFenceMat[i][j] != "\t":
                    ciphertext += railFenceMat[i][j][0]

        return ciphertext

    def decrypt(self, ciphertext):
        # Get the Rail Fence matrix for the given ciphertext
        railfenceMat = self.getRailFence(ciphertext, self.key)

        plaintext = ""

        dir_down = False
        row = 0
        col = 0
        lookahead = 0

        # Place the characters from the ciphertext back into the matrix
        for i in range(self.key):
            for j in range(len(ciphertext)):
                if railfenceMat[i][j] != "\t":
                    railfenceMat[i][j] = ciphertext[lookahead] + "\t"
                    lookahead += 1

        dir_down = False
        row = 0
        col = 0

        # Read the matrix in a zigzag pattern to decrypt the text
        for j in range(len(ciphertext)):
            if row == 0 or row == self.key - 1:
                dir_down = not dir_down

            if railfenceMat[row][col] != "\t":
                plaintext += railfenceMat[row][col][0]
```

```
                col += 1

            if dir_down:
                row += 1
            else:
                row -= 1

        return plaintext

class ImprovedRailFence:
    def __init__(self, key):
        self.key = key  # Initialize the key

    def setKey(self, key):
        self.key = key

    def getRailFence(self, plaintext, key):
        # Create a matrix with 'key' rows and length of plaintext columns
        railFenceMat = [["\t" for i in range(key)] for j in range(len(plaintext))]

        phase_value = 0
        end = False
        # Two Left To Right, then right to left


        row, col = 0, 0
        i = 0

        while not end:

            if phase_value == 0: # Left to Right, Top to Bottom
                for _ in range(self.key):
                    railFenceMat[row][col] = plaintext[i] + "\t"
                    i += 1
                    if i == len(plaintext): # If all characters are placed
                        end = True
                        break
                    col = (col + 1) % self.key # Change the column
                    row += 1
                phase_value = (phase_value + 1) % self.key # Change the phase value

            elif phase_value == 1: # Left to Right, Bottom to Top
                for _ in range(self.key):
                    railFenceMat[row][col] = plaintext[i] + "\t" # Place the
character in the matrix
                    i += 1
                    if i == len(plaintext):
                        end = True
                        break
                    col = (col + 1) % self.key # Change the column
                    row += 1
```

```
                phase_value = (phase_value + 1) % self.key # Change the phase value

            elif phase_value == 2: # Right to Left, Bottom to Top
                for _ in range(self.key):
                    col = self.key - 1
                    railFenceMat[row][col] = plaintext[i] + "\t"
                    i += 1
                    if i == len(plaintext):
                        end = True
                        break
                    col = (col - 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

            elif phase_value == 3: # Right to Left, Top to Bottom
                for _ in range(self.key):
                    col = self.key - 1
                    railFenceMat[row][col] = plaintext[i] + "\t"
                    i += 1
                    if i == len(plaintext):
                        end = True
                        break
                    col = (col - 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

        return railFenceMat

    def encrypt(self, plaintext):

        railFenceMat = self.getRailFence(plaintext, self.key) # Get the Rail Fence
matrix

        random.seed(self.key) # Seed the random number generator
        column_order = list(range(self.key)) # Create a list of columns
        random.shuffle(column_order) # Shuffle the columns

        cipher = ""

        for col in column_order: # Read the matrix column by column
            for row in range(len(plaintext)): # Read the matrix row by row
                if railFenceMat[row][col] != "\t": # If the character is not a
placeholder
                    cipher += railFenceMat[row][col][0] # Append the character to
the ciphertext

        return cipher

    def decrypt(self, ciphertext):

        temp = "*" * len(ciphertext) # Create a string of '*' characters
```

```
        railFenceMat = self.getRailFence(temp, self.key) # Get the Rail Fence
matrix

        random.seed(self.key) # Seed the random number generator
        column_order = list(range(self.key)) # Create a list of columns
        random.shuffle(column_order)     # Shuffle the columns

        plaintext = "" # Initialize the plaintext
        lookahead = 0 # Initialize the lookahead

        for col in column_order: # Read the matrix column by column
            for row in range(len(ciphertext)): # Read the matrix row by row
                if railFenceMat[row][col] == "*\t": # If the character is a
placeholder
                    railFenceMat[row][col] = ciphertext[lookahead] + "\t" # Replace
the placeholder with the character
                    lookahead += 1

        phase_value = 0 # Initialize the phase value
        end = False # Initialize the end flag
        row, col = 0, 0 # Initialize the row and column

        i = 0 # Initialize the index

        while not end:

            if phase_value == 0: # Left to Right, Top to Bottom
                for _ in range(self.key):
                    plaintext += railFenceMat[row][col][0] # Append the character
to the plaintext
                    i += 1
                    if i == len(ciphertext):
                        end = True
                        break
                    col = (col + 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

            elif phase_value == 1: # Left to Right, Bottom to Top
                for _ in range(self.key):
                    plaintext += railFenceMat[row][col][0] # Append the character
to the plaintext
                    i += 1
                    if i == len(ciphertext):
                        end = True
                        break
                    col = (col + 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

            elif phase_value == 2: # Right to Left, Bottom to Top
```

```
                for _ in range(self.key):
                    col = self.key - 1
                    plaintext += railFenceMat[row][col][0] # Append the character
to the plaintext
                    i += 1
                    if i == len(ciphertext):
                        end = True
                        break
                    col = (col - 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

            elif phase_value == 3: # Right to Left, Top to Bottom
                for _ in range(self.key):
                    col = self.key - 1
                    plaintext += railFenceMat[row][col][0] # Append the character
to the plaintext
                    i += 1
                    if i == len(ciphertext):
                        end = True
                        break
                    col = (col - 1) % self.key
                    row += 1
                phase_value = (phase_value + 1) % self.key

        return plaintext



# Example usage
rf = RailFence(3)

# Print the Rail Fence matrix for the given plaintext
print()
printMat(rf.getRailFence("HELLO", 3))

# Encrypt the plaintext
enc = rf.encrypt("HELLO")
print("\nEncrypted using Rail Fence: ", enc, end="\n")  # Output the encrypted text

# Decrypt the ciphertext
dec = rf.decrypt(enc)
print("\nDecrypted using Rail Fence: ", dec, end="\n")  # Output the decrypted text

imp = ImprovedRailFence(3) # Initialize the Improved Rail Fence object

print()
printMat(imp.getRailFence("HELLO", 3)) # Print the Rail Fence matrix for the given
plaintext
```

```
enc = imp.encrypt("HELLO") # Encrypt the plaintext
print("\nEncrypted using Improved Rail Fence: ", enc, end="\n") # Output the
encrypted text

dec = imp.decrypt(enc) # Decrypt the ciphertext
print("\nDecrypted using Improved Rail Fence: ", dec, end="\n") # Output the
decrypted text
```

Output:

```
H       *       *       *       O
*       E       *       L       *
*       *       L       *       *

Encrypted using Rail Fence:  HOELL

Decrypted using Rail Fence:  HELLO

H       *       *
*       E       *
*       *       L
L       *       *
*       O       *

Encrypted using Improved Rail Fence:  EOLHL

Decrypted using Improved Rail Fence:  HELLO
```

Practical-6

Q1. Implement Columnar Transposition Cipher and Improvised Columnar Transposition Cipher

A1.

Encryption Process (Columnar Transposition):

1. **Matrix Creation**:
    - The plaintext is divided into rows and columns based on the length of the encryption key. If necessary, padding ('X') is added to ensure the matrix is complete.
    - The getColumnarMatrix() function creates this grid, where each character of the plaintext is filled row by row.
2. **Column Rearrangement**:
    - The columns of the matrix are rearranged based on the alphabetical order of the characters in the key. This is done by sorting the key and reordering the corresponding columns.
    - The encrypt() function reads the matrix in column-major order based on the sorted key and generates the ciphertext by concatenating characters column by column.

Decryption Process (Columnar Transposition):

1. **Reconstruct the Matrix**:
    - The decrypt() function first reconstructs the matrix by placing the characters of the ciphertext into columns in the order determined by the sorted key.
    - Once the matrix is reconstructed, it reads the matrix row by row to retrieve the original plaintext.
2. **Removing Padding**:
    - Any padding ('X') added during encryption is removed from the decrypted plaintext before returning the final result.

Improved Columnar Transposition Encryption Process:

1. **Matrix Creation**:

- Similar to the basic Columnar Transposition, the plaintext is arranged into a matrix of rows and columns based on the key length. Padding ('X') is also added if necessary.

2. **Row Rotation**:
   - Each row of the matrix is rotated by a shift value calculated from the sum of the ASCII values of the characters in the key. This adds an extra layer of complexity to the encryption.
   - The rotateRows() function rotates each row by a specific amount.
3. **Reordered Column Reading**:
   - The columns are then read based on a randomly generated order using the key. If the column index is even, it is read top to bottom; if odd, it is read bottom to top. This further scrambles the plaintext.
   - The encrypt() function handles this column reordering and forms the final ciphertext.

Improved Columnar Transposition Decryption Process:

1. **Reconstruct the Matrix**:
   - The decrypt() function first reconstructs the matrix based on the random column order generated during encryption. It refills the matrix with characters from the ciphertext.
2. **Reverse Row Rotation**:
   - After the matrix is filled, the rows are rotated back to their original positions using the reverseRotateRows() function.
3. **Reading the Matrix**:
   - Finally, the plaintext is reconstructed by reading the matrix row by row. Any padding ('X') is removed before returning the decrypted message.

Time Complexity Analysis

For the **basic Columnar Transposition cipher**, the time complexity for both encryption and decryption is **O(n * m * log m)**, where n is the length of the plaintext and m is the length of the key. This is because we first arrange the plaintext into an n / m matrix, which takes **O(n)** time, and then sort the key to rearrange the columns, which takes **O(m log m)** time. For each sorted column, we traverse all rows, leading to a total complexity of **O(n * m * log m)**.

For the **Improved Columnar Transposition cipher**, the time complexity is similar but with added complexity for row rotation. The encryption and decryption both involve generating a

matrix in **O(n)**, sorting the columns in **O(m log m)**, and performing row rotation in **O(n)**, resulting in an overall complexity of **O(n * m * log m)**. The additional row rotation and its reversal slightly increase the constant factors but do not affect the overall time complexity compared to the basic version.

Code:

```
import math
import random

class ColumnarTransposition:

    # Constructor to initialize with the provided key
    def __init__(self, key):
        self.key = key

    # Function to update the key if needed
    def setKey(self, key):
        self.key = key

    # Function to create the columnar matrix for encryption and decryption
    def getColumnarMatrix(self, plaintext):
        # Calculate the target length by rounding up to fill the grid
        targetLength = math.ceil(len(plaintext) / len(self.key)) * len(self.key)
        plainList = list(plaintext)
        # Pad the plaintext with 'X' if necessary to complete the matrix
        plainList += ['X' for _ in range(targetLength - len(plaintext))]
        plaintext = ''.join(plainList)

        mat = []

        totalRow = int(len(plaintext) / len(self.key))
        look = 0

        # Populate the matrix row by row
        for i in range(totalRow):
            tempString = ''
            tempList = []
            for j in range(len(self.key)):
                tempList.append(plaintext[look])
                look += 1
            mat.append(tempList)

        return mat

    # Encryption function using the key to rearrange the matrix columns
    def encrypt(self, plaintext):
        mat = self.getColumnarMatrix(plaintext)
```

```
        enuList = list(enumerate(self.key))

        # Sort the key to get the new column order
        sortedEnuList = sorted(enuList, key= lambda x : x[1])

        cipher = ''

        totalRow = int(len(plaintext) / len(self.key))

        # Read the columns in sorted key order to generate the ciphertext
        for index, _ in sortedEnuList:
            for row in range(totalRow + 1):
                cipher += mat[row][index]

        return cipher

    # Decryption function that reverses the encryption process
    def decrypt(self, cipher):
        totalRow = math.ceil(len(cipher) / len(self.key))  # Number of rows in the
matrix

        # Create a matrix with empty strings to store the reordered columns
        mat = [['' for _ in range(len(self.key))] for _ in range(totalRow)]

        # Sort the key to find the column order
        enuList = list(enumerate(self.key))
        sortedEnuList = sorted(enuList, key=lambda x: x[1])

        look = 0

        # Refill the matrix columns in the sorted order
        for index, _ in sortedEnuList:
            for row in range(totalRow):
                mat[row][index] = cipher[look]
                look += 1

        plainText = ''

        # Rebuild the plaintext by reading the matrix row by row
        for row in mat:
            plainText += ''.join(row)

        return plainText.rstrip('X')  # Remove padding (X) if any

class ImprovedCol:

    # Constructor to initialize with the provided key
    def __init__(self, key):
        self.key = key
```

```
        # Seed the random module using the sum of ASCII values of characters in the
key
        seed_value = sum(ord(char) for char in self.key)
        random.seed(seed_value)

        # Random order of columns:
        self.order = list(range(len(self.key)))
        random.shuffle(self.order)

    # Rotate the rows based on key
    def rotateRows(self, mat):
        for i, row in enumerate(mat):
            shift = sum(ord(char) for char in self.key) % len(row)
            mat[i] = row[shift:] + row[:shift]
        return mat

    # Reverse the row rotation during decryption
    def reverseRotateRows(self, mat):
        for i, row in enumerate(mat):
            shift = sum(ord(char) for char in self.key) % len(row)
            mat[i] = row[-shift:] + row[:-shift]
        return mat

    # Create the columnar matrix for the given plaintext
    def getColumnarMatrix(self, plaintext):
        # Calculate the target length by rounding up to fill the grid
        targetLength = math.ceil(len(plaintext) / len(self.key)) * len(self.key)
        plainList = list(plaintext)

        # Pad the plaintext with 'X' if necessary to complete the matrix
        plainList += ['X' for _ in range(targetLength - len(plaintext))]
        plaintext = ''.join(plainList)

        mat = []
        totalRow = int(len(plaintext) / len(self.key))
        look = 0

        # Populate the matrix row by row
        for i in range(totalRow):
            tempList = []
            for j in range(len(self.key)):
                tempList.append(plaintext[look])
                look += 1
            mat.append(tempList)

        return mat

    # Encrypt the plaintext
    def encrypt(self, plaintext):
        mat = self.getColumnarMatrix(plaintext)
```

```
        # Rotate the rows
        mat = self.rotateRows(mat)

        orderEnu = list(enumerate(self.order))
        sortedEnuList = sorted(orderEnu, key=lambda x: x[1])

        cipher = ''

        # If column is even, go from top to bottom, otherwise bottom to top
        for index, _ in orderEnu:
            if index % 2 == 0:
                for row in range(len(mat)):
                    cipher += mat[row][index]
            else:
                for row in range(len(mat) - 1, -1, -1):
                    cipher += mat[row][index]

        return cipher

    # Decrypt the ciphertext
    def decrypt(self, cipher):
        totalRow = math.ceil(len(cipher) / len(self.key))
        mat = [['' for _ in range(len(self.key))] for _ in range(totalRow)]

        look = 0

        orderEnu = list(enumerate(self.order))
        sortedEnuList = sorted(orderEnu, key=lambda x: x[1])

        # If column is even, go top-down, otherwise bottom-up to fill the matrix
        for index, _ in orderEnu:
            if index % 2 == 0:
                for row in range(len(mat)):
                    mat[row][index] = cipher[look]
                    look += 1
            else:
                for row in range(len(mat) - 1, -1, -1):
                    mat[row][index] = cipher[look]
                    look += 1

        # Reverse the row rotation
        mat = self.reverseRotateRows(mat)

        plainText = ''.join(''.join(row) for row in mat)

        return plainText.rstrip('X')



col = ColumnarTransposition('keys')
```

```
cipher = col.encrypt('hellohowareyouiamfine'.upper())
print(f"Cipher: {cipher}")
print(f"Decrypted: {col.decrypt(cipher)}")

impro = ImprovedCol('keys')
cipher = impro.encrypt('hellohowareyouiamfine'.upper())
print(f"Cipher: {cipher}")
print(f"Decrypted: {impro.decrypt(cipher)}")
```

Output:

```
Cipher: HOAOMEXFURHELOEIIXXNAYWL
Decrypted: HELLOHOWAREYOUIAMFINE
```

Basic Columnar Transposition

```
Cipher: EHRUFXHOAOMELWYANXLOEIIX
Decrypted: HELLOHOWAREYOUIAMFINE
```

Improvised Columnar Transposition

Practical-7

Q1. Implement RSA Cipher and Improvised RSA Cipher.

A1.

1. Normal RSA Implementation

Overview

The normal RSA implementation uses two large prime numbers ppp and qqq to generate a public and private key pair. The algorithm is primarily based on the mathematical properties of prime factorization and modular arithmetic. The key steps in the normal RSA algorithm are as follows:

1. **Key Generation**:
    - Select two large prime numbers ppp and qqq.
    - Compute n=p×qn = p \times qn=p×q.
    - Calculate Euler's totient function ϕ(n)=(p−1)(q−1)\phi(n) = (p - 1)(q - 1)ϕ(n)=(p−1)(q−1).
    - Choose an integer eee such that 1<e<ϕ(n)1 < e < \phi(n)1<e<ϕ(n) and gcd(e,ϕ(n))=1\text{gcd}(e, \phi(n)) = 1gcd(e,ϕ(n))=1.
    - Compute ddd as the modular inverse of eee modulo ϕ(n)\phi(n)ϕ(n).
2. **Encryption**:
    - Convert the plaintext message into an integer MMM.
    - Compute the ciphertext C=Memod nC = M^e \mod nC=Memodn.
3. **Decryption**:
    - Compute the plaintext M=Cdmod nM = C^d \mod nM=Cdmodn.

Time Complexity

- **Key Generation**:
    - Prime generation is O(k3)O(k^3)O(k3) for kkk-bit numbers (using probabilistic tests).
    - Key computation (multiplication and GCD) is O(k2)O(k^2)O(k2).
- **Encryption and Decryption**:
    - Both encryption and decryption are O(k3)O(k^3)O(k3) due to the modular exponentiation.

Overall, the time complexity for the normal RSA implementation is dominated by the key generation phase, leading to **O(k3)O(k^3)O(k3)**.

---

2. Modified RSA Implementation

Overview

The modified RSA implementation introduces randomness and adjustments to the prime generation process based on the ASCII value of the message. It incorporates checks to ensure the generated numbers are prime, and if not, it finds the nearest primes. The steps include:

1. **Key Generation**:
    - Use the ASCII value of the message to seed a random number generator.
    - Generate two random numbers ppp and qqq.
    - Check if ppp and qqq are prime; if not, adjust them to the nearest lower prime (for ppp) and the nearest greater prime (for qqq).
    - Compute n=p×qn = p \times qn=p×q and ϕ(n)\phi(n)ϕ(n).
    - Choose eee such that gcd(e,ϕ(n))=1\text{gcd}(e, \phi(n)) = 1gcd(e,ϕ(n))=1.
    - Compute ddd as the modular inverse of eee modulo ϕ(n)\phi(n)ϕ(n).
2. **Encryption and Decryption**:
    - The same as in normal RSA.

Time Complexity

- **Key Generation**:
    - Prime checking involves O(p)O(\sqrt{p})O(p) for each number, which can be significant depending on how large the numbers are.
    - Finding the nearest primes also involves iterating, leading to an additional O(k)O(k)O(k) complexity for each adjustment.
    - Thus, the overall complexity for key generation might become **O(k2)+O(k⋅p)O(k^2) + O(k \cdot \sqrt{p})O(k2)+O(k⋅p)**.
- **Encryption and Decryption**:
    - As with normal RSA, both operations remain at O(k3)O(k^3)O(k3) due to modular exponentiation.

The modified RSA implementation thus has a time complexity of approximately **O(k3)O(k^3)O(k3)**, similar to the normal implementation but with additional overhead during key generation for prime checking and adjustment.

Conclusion

Both normal and modified RSA implementations provide robust encryption methods, with time complexities primarily dictated by the key generation and modular exponentiation processes. The modifications made in the RSA implementation offer flexibility and randomness, albeit with slightly more overhead in the key generation phase.

Code:

RSA:

```
# Function to compute the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Function to compute the modular inverse using the Extended Euclidean Algorithm
def modinv(e, phi):
    # Initialize the variables for the extended Euclidean algorithm
    A1, A2, A3 = 1, 0, phi
    B1, B2, B3 = 0, 1, e

    # Perform the algorithm in a loop
    while B3 != 0 and B3 != 1:
        Q = A3 // B3  # Integer division
        # Update the variables
        T1, T2, T3 = A1 - Q * B1, A2 - Q * B2, A3 - Q * B3
        A1, A2, A3 = B1, B2, B3
        B1, B2, B3 = T1, T2, T3

    if B3 == 0:
        raise Exception("Modular inverse does not exist")

    # If we get here, B3 is 1, so the inverse exists and is B2
    return B2 % phi

# Function to generate RSA keys
def generate_rsa_keys(p, q):
    # Step 1: Compute n = p * q
    n = p * q

    # Step 2: Compute phi(n) = (p-1) * (q-1)
    phi = (p - 1) * (q - 1)

    # Step 3: Choose e such that 1 < e < phi and gcd(e, phi) = 1 (typically e =
65537)
    e = 65537
```

```
    if gcd(e, phi) != 1:
        raise ValueError("e and phi(n) are not coprime!")

    # Step 4: Compute the modular inverse of e mod phi (this is d)
    d = modinv(e, phi)

    # Public key is (e, n), private key is (d, n)
    return (e, n), (d, n)

# RSA encryption function
def encrypt(plaintext, public_key):
    e, n = public_key
    # Convert plaintext to an integer using ord (assuming plaintext is a single
character)
    plaintext_int = ord(plaintext)
    # Encrypt using ciphertext = plaintext^e mod n
    ciphertext = pow(plaintext_int, e, n)
    return ciphertext

# RSA decryption function
def decrypt(ciphertext, private_key):
    d, n = private_key
    # Decrypt using plaintext = ciphertext^d mod n
    plaintext_int = pow(ciphertext, d, n)
    # Convert integer back to a character
    plaintext = chr(plaintext_int)
    return plaintext

# Example usage
if __name__ == "__main__":
    # Choose two small prime numbers (in real RSA, p and q should be much larger)
    p = 61
    q = 53

    # Generate public and private keys
    public_key, private_key = generate_rsa_keys(p, q)

    print(f"Value of p: {p}")
    print(f"Value of q: {q}")

    # Print the public and private keys
    print(f"Public Key (e, n): {public_key}")
    print(f"Private Key (d, n): {private_key}")

    # Message to encrypt
    message = 'A'  # Single character message

    print(f"Message: {message}")

    # Encrypt the message
    ciphertext = encrypt(message, public_key)
```

```
    print(f"Ciphertext: {ciphertext}")

    # Decrypt the ciphertext
    decrypted_message = decrypt(ciphertext, private_key)
    print(f"Decrypted message: {decrypted_message}")
```

Modified RSA:

```
import random

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

# Function to find the nearest lower prime
def nearest_lower_prime(n):
    while n > 2:
        n -= 1
        if is_prime(n):
            return n
    return 2  # Smallest prime

# Function to find the nearest greater prime
def nearest_greater_prime(n):
    while True:
        n += 1
        if is_prime(n):
            return n

# Function to generate a random prime or adjust if it's not prime
def generate_random_prime(seed):
    random.seed(seed)  # Set the seed for reproducibility
    num = random.randint(50, 200)
    return num

# Function to compute the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Function to compute the modular inverse using the Extended Euclidean Algorithm
def modinv(e, phi):
    # Initialize the variables for the extended Euclidean algorithm
    A1, A2, A3 = 1, 0, phi
```

```
    B1, B2, B3 = 0, 1, e

    # Perform the algorithm in a loop
    while B3 != 0 and B3 != 1:
        Q = A3 // B3  # Integer division
        # Update the variables
        T1, T2, T3 = A1 - Q * B1, A2 - Q * B2, A3 - Q * B3
        A1, A2, A3 = B1, B2, B3
        B1, B2, B3 = T1, T2, T3

    if B3 == 0:
        raise Exception("Modular inverse does not exist")

    # If we get here, B3 is 1, so the inverse exists and is B2
    return B2 % phi

# Function to generate RSA keys
def generate_rsa_keys(message):
    # Step 1: Use the ASCII value of the message as a seed for generating primes
    seed = ord(message)
    p = generate_random_prime(seed)
    q = generate_random_prime(seed + 1)  # Add a small offset to generate a second
distinct prime

    p, q = min(p, q), max(p, q)  # Ensure p < q
    p, q = nearest_lower_prime(p), nearest_greater_prime(q)  # Adjust if not prime

    # Step 2: Compute n = p * q
    n = p * q

    # Step 3: Compute phi(n) = (p-1) * (q-1)
    phi = (p - 1) * (q - 1)

    # Step 4: Choose e such that 1 < e < phi and gcd(e, phi) = 1 (typically e =
65537)
    random.seed(seed + 2)  # Set the seed for reproducibility
    e = random.randint(2, phi - 1)

    e = nearest_lower_prime(e)  # Adjust e to the nearest lower prime

    if gcd(e, phi) != 1:
        raise ValueError("e and phi(n) are not coprime!")

    # Step 5: Compute the modular inverse of e mod phi (this is d)
    d = modinv(e, phi)

    # Public key is (e, n), private key is (d, n)
    return (e, n), (d, n), p, q

# RSA encryption function
def encrypt(plaintext, public_key):
```

```
    e, n = public_key
    # Convert plaintext to an integer using ord (assuming plaintext is a single
character)
    plaintext_int = ord(plaintext)
    # Encrypt using ciphertext = plaintext^e mod n
    ciphertext = pow(plaintext_int, e, n)
    return ciphertext

# RSA decryption function
def decrypt(ciphertext, private_key):
    d, n = private_key
    # Decrypt using plaintext = ciphertext^d mod n
    plaintext_int = pow(ciphertext, d, n)
    # Convert integer back to a character
    plaintext = chr(plaintext_int)
    return plaintext

# Example usage
if __name__ == "__main__":
    # Message to encrypt
    message = 'B'  # Single character message

    # Generate public and private keys using the ASCII value of the message as seed
    public_key, private_key, p, q = generate_rsa_keys(message)

    # Print the public and private keys
    print(f"Public Key (e, n): {public_key}")
    print(f"Private Key (d, n): {private_key}")
    print(f"Generated primes p: {p}, q: {q}")

    # Encrypt the message
    ciphertext = encrypt(message, public_key)
    print(f"Ciphertext: {ciphertext}")

    # Decrypt the ciphertext
    decrypted_message = decrypt(ciphertext, private_key)
    print(f"Decrypted message: {decrypted_message}")
```

Output:

RSA:

```
Value of p: 61
Value of q: 53
Public Key (e, n): (65537, 3233)
Private Key (d, n): (2753, 3233)
Message: A
Ciphertext: 2790
Decrypted message: A
```

Modified RSA:

```
Public Key (e, n): (3821, 4757)
Private Key (d, n): (1301, 4757)
Generated primes p: 67, q: 71
Ciphertext: 66
Decrypted message: B
```

Practical-8

Q1. Implement DSA Cipher and Improvised DSA Cipher.

A1.

```
import math

# First key generation
class GCDUtil:
    def find_coprime(self, num):
        for i in range(2, num):
            if math.gcd(num, i) == 1:
                return i

    def extended_gcd(self, a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = self.extended_gcd(b % a, a)

        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y


class KeyPairGenerator:
    def is_prime(self, candidate):
        # First consider edge case:
        if candidate <= 1:
            return False
        else:
            # Calculating approx. square root.
            for i in range(2, int(math.sqrt(candidate) + 1)):
                if candidate % i == 0:
                    return False
            return True

    def generate_keys(self, prime1, prime2):
        gcd_util = GCDUtil()
        if self.is_prime(int(prime1)) and self.is_prime(int(prime2)):
            modulus = prime1 * prime2
            totient = (prime1 - 1) * (prime2 - 1)  # Calculate the totient function
φ(n)

            # Now check for a number e in range 1 to φ(n) such that gcd(e, φ(n)) =
1
            exponent = gcd_util.find_coprime(totient)
```

```
            # Use extended Euclidean algorithm to find d such that (d * e) % φ(n) =
1
            gcd, private_key, _ = gcd_util.extended_gcd(exponent, totient)
            private_key = private_key % totient  # Ensure d is positive
            if private_key < 0:
                private_key += totient

            return exponent, private_key, modulus
        else:
            raise ValueError("Both numbers must be prime.")


# Encrypt with d (Private key) and it is called signature
class MessageSender:
    def encrypt(self, message, private_key, modulus):
        # Encrypt each character and return a list of encrypted values
        return [pow(char, private_key, modulus) for char in message]


# Verify With e (Public key) which is announced or receiver had it beforehand
class MessageReceiver:
    def decrypt(self, signature, exponent, modulus):
        # Decrypt each character and return the original message
        return ''.join(chr(pow(char, exponent, modulus)) for char in signature)


# Driver Code
keygen = KeyPairGenerator()
p = int(input("Enter first prime number (p): "))
q = int(input("Enter second prime number (q): "))
e, d, n = keygen.generate_keys(p, q)

# Sender encrypts message with private key d
plain_text = input("Enter message to encrypt: ")
# Convert the message to a list of ASCII values
plain_num = [ord(char) for char in plain_text]

sender = MessageSender()
signature = sender.encrypt(plain_num, d, n)
print(f"The signature generated to send to receiver is: {signature}")

# Receiver decrypts signature with public key e
receiver = MessageReceiver()
message = receiver.decrypt(signature, e, n)
print(f"Message and signature received by receiver: {message}")

# Check if the original message matches the decrypted message
if message == plain_text:
    print("Message is authentic")
else:
    print("Message tampered")
```

```python
import math
import random

# First key generation
class Key:
    def find_coprime(self, num):
        for i in range(2, num):
            if math.gcd(num, i) == 1:
                return i

    def extended_gcd(self, a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = self.extended_gcd(b % a, a)

        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y


class KeyGen:
    def is_prime(self, candidate):
        # First consider edge case:
        if candidate <= 1:
            return False
        else:
            # Calculating approx. square root.
            for i in range(2, int(math.sqrt(candidate) + 1)):
                if candidate % i == 0:
                    return False
            return True

    def generate_keys(self, prime1, prime2):
        backpack = Key()
        if self.is_prime(int(prime1)) and self.is_prime(int(prime2)):
            modulus = prime1 * prime2
            totient = (prime1 - 1) * (prime2 - 1)  # Calculate the totient function
φ(n)

            # Now check for a number e in range 1 to φ(n) such that gcd(e, φ(n)) =
1
            exponent = backpack.find_coprime(totient)

            # Use extended Euclidean algorithm to find d such that (d * e) % φ(n) =
1
            gcd, private_key, _ = backpack.extended_gcd(exponent, totient)
            private_key = private_key % totient  # Ensure d is positive
            if private_key < 0:
                private_key += totient

            return exponent, private_key, modulus
```

```
        else:
            raise ValueError("Both numbers must be prime.")


# Second encrypt with d (Private key) and it is called signature
class Encryptor:
    def encrypt_message(self, msg, private_key, modulus):
        # Encrypt each character using random values seeded with p * q
        random.seed(modulus)  # Seed the random generator with modulus (p * q)
        encrypted_output = []
        for char in msg:
            # Generate a random modifier
            modifier = random.randint(1, 100)
            # Encrypt the character and apply the modifier
            encrypted_char = pow(char + modifier, private_key, modulus)
            encrypted_output.append(encrypted_char)
        return encrypted_output


# Fourth Verify With e (Public key) which is announced or receiver had it
beforehand
class Decryptor:
    def decrypt_signature(self, encrypted_signature, public_key, modulus):
        # Decrypt each character and return the original message
        decrypted_output = []
        for char in encrypted_signature:
            # Decrypt the character
            decrypted_char = pow(char, public_key, modulus)
            decrypted_output.append(decrypted_char)  # Append the decrypted number
        return decrypted_output


# Driver Code
key_gen = KeyGen()
first_prime = int(input("Enter the First prime number for key generation: "))
second_prime = int(input("Enter the Second prime number for key generation: "))
public_key, private_key, modulus = key_gen.generate_keys(first_prime, second_prime)

# Encryptor encrypts message with private key
input_text = input("Enter Message to Encrypt: ")
# Convert the message to a list of ASCII values
ascii_values = [ord(char) for char in input_text]

encryptor = Encryptor()
signature = encryptor.encrypt_message(ascii_values, private_key, modulus)
print(f"The signature generated to send to receiver is: {signature}")
print("Sending Message & Signature to Receiver.............")

# Decryptor decrypts signature with public key
decryptor = Decryptor()
decrypted_signature = decryptor.decrypt_signature(signature, public_key, modulus)
# Convert the decrypted ASCII values back to characters
```

```
final_message = ''.join(chr(num) for num in decrypted_signature)
print(f"Message and Signature Received by Receiver: {final_message}")

# Check if the original message matches the decrypted message
if input_text == final_message:
    print("Verification Successful")
else:
    print("Message Tampered")
```

Output:

```
Enter the First prime number for key generation: 23
Enter the Second prime number for key generation: 11
Enter Message to Encrypt: hell
The signature generated to send to receiver is: [13, 190, 20, 138]
```

Improved

```
Enter first prime number (p): 23
Enter second prime number (q): 11
Enter message to encrypt: hell
The signature generated to send to receiver is: [36, 29, 147, 147]
Message and signature received by receiver: hell
Message is authentic
```

Original

Original RSA-like Algorithm Complexity

1. **Key Generation:**
   - **Primality Testing**: Checking if p and q are prime. Assuming we use the trial division method (which is $O(\sqrt{p})$ and $O(\sqrt{q})$), the overall complexity for both primes is:
   $$O(\sqrt{p} + \sqrt{q})$$
   - **Calculating** n: This is a simple multiplication $n = p \times q$, which is $O(1)$.
   - **Calculating the Totient** $\phi(n)$: This is also a constant time operation:
   $$O(1)$$
   - **Finding** e: This involves finding a coprime number e such that $1 < e < \phi(n)$. The complexity of finding e using the Euclidean algorithm is:
   $$O(\log(\phi(n))) \approx O(\log(p \cdot q)) = O(\log p + \log q)$$
   - **Calculating** d: Using the Extended Euclidean Algorithm also takes:
   $$O(\log(\phi(n))) \approx O(\log p + \log q)$$

   Thus, the overall complexity for the **key generation** step in the original algorithm is:
   $$O(\sqrt{p} + \sqrt{q} + \log p + \log q)$$

2. **Encryption and Decryption:**
   - **Encryption**: The encryption process involves exponentiation $C = M^d \mod n$. This can be efficiently done using modular exponentiation, which has a complexity of:
   $$O(\log d \cdot \log n)$$
   - **Decryption**: Similarly, decryption involves $M = C^e \mod n$ with the same complexity:
   $$O(\log e \cdot \log n)$$

   Assuming d and e are less than $\phi(n)$, the overall complexity for **encryption/decryption** is:
   $$O(\log n \cdot \log(p \cdot q)) = O(\log p \cdot \log q)$$

Improved RSA-like Algorithm Complexity

1. **Key Generation:**
   - **Primality Testing**: As in the original, the complexity for testing if p and q are prime remains:
   $$O(\sqrt{p} + \sqrt{q})$$
   - **Calculating n**: This remains $O(1)$.
   - **Calculating the Totient $\phi(n)$**: This also remains $O(1)$.
   - **Finding e**: The process of finding a coprime e is the same as in the original algorithm:
   $$O(\log p + \log q)$$
   - **Calculating d**: This remains:
   $$O(\log p + \log q)$$

   Thus, the overall complexity for the **key generation** step in the improved algorithm is:
   $$O(\sqrt{p} + \sqrt{q} + \log p + \log q)$$
2. **Encryption and Decryption:**
   - **Encryption**: The encryption in the improved version uses a random modifier for each character and encrypts based on ASCII values. Thus, for a message of length m:
     - The encryption complexity becomes $O(m \cdot (\log d + \log n)) = O(m \cdot (\log p + \log q))$.
   - **Decryption**: Similar to encryption, the decryption complexity also remains:
   $$O(m \cdot (\log e + \log n)) = O(m \cdot (\log p + \log q))$$

Summary of Complexity

- **Original Algorithm Complexity**:
  - Key Generation: $O(\sqrt{p} + \sqrt{q} + \log p + \log q)$
  - Encryption/Decryption: $O(\log p \cdot \log q)$
- **Improved Algorithm Complexity**:
  - Key Generation: $O(\sqrt{p} + \sqrt{q} + \log p + \log q)$
  - Encryption/Decryption: $O(m \cdot (\log p + \log q))$

Note

- In both cases, m is the length of the message being encrypted.
- The complexities regarding primality checking can vary depending on the algorithm used (e.g., Miller-Rabin for larger primes).
- The improved algorithm may have higher encryption/decryption complexity due to the random modifiers added during the encryption process, which introduces a factor related to the message length m.

Practical-9

Q1. Implement Diffie Hillman Key Exchange and Improvised Diffie Hillman Key Exchange.

A1.

Diffie Hillman Key Exchange:

```
# Diffie Hillman Key Exchange

import random

butterLootSpots = ['Akrura Bapu', 'Devashrav Kaka', 'Devak Nana', 'Shursen Dada']

class Person:

    def __init__(self, name):
        self.name = name
        self.private_key = 0

class SecurityExchangeCompany:

    def __init__(self, prime, generator):
        self.prime = prime
        self.generator = generator

    def requestPublicKey(self, person):
        person.private_key = random.randint(1, 100)
        return (self.generator ** person.private_key) % self.prime

    def computeSharedKey(self, person, publicKey):
        return (publicKey ** person.private_key) % self.prime

class Channel:

    def __init__(self, person1, person2, sharedKey):
        self.person1 = person1
        self.person2 = person2
        self.sharedKey = sharedKey % 256

    def messagePassing(self, sender, receiver):
        message = [random.randint(1, 100)]
        temp = "Today's loot spot is " + random.choice(butterLootSpots) + "'s
house."
        for c in temp:
            message.append(ord(c) ^ self.sharedKey)
        message.append(random.randint(1, 100))
```

```
        print("Message sent by ", sender.name, " to ", receiver.name, ": ",
message)
        return message

    def decryptMessage(self, message):
        # Start from index 1 to skip the first random integer
        decrypted_message = []
        for c in message[1:-1]:  # Skip the first and the last element
            decrypted_message.append(chr(c ^ self.sharedKey))  # Correctly decrypt
each character
        return "".join(decrypted_message)  # Join the list into a string


def main():

    prime = 13147
    generator = 5

    krishna = Person("Krishna")
    sudama = Person("Sudama")
    pundirka = Person("Pundirka")
    balram = Person("Balram")

    sec = SecurityExchangeCompany(prime, generator)

    krishnaPublicKey = sec.requestPublicKey(krishna)
    sudamaPublicKey = sec.requestPublicKey(sudama)
    pundirkaPublicKey = sec.requestPublicKey(pundirka)
    balramPublicKey = sec.requestPublicKey(balram)

    print("\n\nKrishna Public Key: ", krishnaPublicKey)
    print("Sudama Public Key: ", sudamaPublicKey)
    print("Pundirka Public Key: ", pundirkaPublicKey)
    print("Balram Public Key: ", balramPublicKey)

    krishnaSudamaSharedKey = sec.computeSharedKey(krishna, sudamaPublicKey)
    pundirkaBalramSharedKey = sec.computeSharedKey(pundirka, balramPublicKey)
    sudamaBalramSharedKey = sec.computeSharedKey(sudama, balramPublicKey)
    krishnaBalramSharedKey = sec.computeSharedKey(krishna, balramPublicKey)

    print("\n-------- Eavesdropper don't know the shared keys -------", end="\n\n")

    print("Krishna-Sudama Shared Key: ", krishnaSudamaSharedKey)
    print("Pundirka-Balram Shared Key: ", pundirkaBalramSharedKey)
    print("Sudama-Balram Shared Key: ", sudamaBalramSharedKey)
    print("Krishna-Balram Shared Key: ", krishnaBalramSharedKey)


    print("\n\n-----------------------------------------\n\n")

    print("Channels available to eavesdrop:")
```

```
    print("1. Krishna-Sudama")
    krishnaSudamaChannel = Channel(krishna, sudama, krishnaSudamaSharedKey)
    print("2. Pundirka-Balram")
    pundirkaBalramChannel = Channel(pundirka, balram, pundirkaBalramSharedKey)
    print("3. Sudama-Balram")
    sudamaBalramChannel = Channel(sudama, balram, sudamaBalramSharedKey)
    print("4. Krishna-Balram")
    krishnaBalramChannel = Channel(krishna, balram, krishnaBalramSharedKey)



    choice = int(input("Enter your choice: "))

    print("\n\n-----------------------------------------\n\n")

    if choice == 1:
        print("Eavesdropper listening to Krishna-Sudama Conversational Channel")
        print("\nAvailable Informations: \n")

        print("Krishna Public Key: ", krishnaPublicKey)
        print("Sudama Public Key: ", sudamaPublicKey)
        print("Prime: ", prime)
        print("Generator: ", generator)

        listened = krishnaSudamaChannel.messagePassing(krishna, sudama)
        print("\nListened Message: ", listened)

        print("\n\nEavesdropper can not decrypt the message as he don't know the
shared key: \n\n")

        print("Decrypted Message: ", krishnaSudamaChannel.decryptMessage(listened))

    elif choice == 2:

        print("Eavesdropper listening to Pundirka-Balram Conversational Channel")
        print("\nAvailable Informations: \n")

        print("Pundirka Public Key: ", pundirkaPublicKey)
        print("Balram Public Key: ", balramPublicKey)
        print("Prime: ", prime)
        print("Generator: ", generator)

        listened = pundirkaBalramChannel.messagePassing(pundirka, balram)
        print("\nListened Message: ", listened)

        print("\n\nEavesdropper can not decrypt the message as he don't know the
shared key: \n\n")

        print("Decrypted Message: ",
pundirkaBalramChannel.decryptMessage(listened))
```

```
    elif choice == 3:

        print("Eavesdropper listening to Sudama-Balram Conversational Channel")
        print("\nAvailable Informations: \n")

        print("Sudama Public Key: ", sudamaPublicKey)
        print("Balram Public Key: ", balramPublicKey)
        print("Prime: ", prime)
        print("Generator: ", generator)

        listened = sudamaBalramChannel.messagePassing(sudama, balram)
        print("\nListened Message: ", listened)

        print("\n\nEavesdropper can not decrypt the message as he don't know the
shared key: \n\n")

        print("Decrypted Message: ", sudamaBalramChannel.decryptMessage(listened))

    elif choice == 4:

        print("Eavesdropper listening to Krishna-Balram Conversational Channel")
        print("\nAvailable Informations: \n")

        print("Krishna Public Key: ", krishnaPublicKey)
        print("Balram Public Key: ", balramPublicKey)
        print("Prime: ", prime)
        print("Generator: ", generator)

        listened = krishnaBalramChannel.messagePassing(krishna, balram)
        print("\nListened Message: ", listened)

        print("\n\nEavesdropper can not decrypt the message as he don't know the
shared key: \n\n")

        print("Decrypted Message: ", krishnaBalramChannel.decryptMessage(listened))

    else:
        print("Exit")

    print("\n\n-----------------------------------------\n\n")

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

Output:

```
Krishna Public Key:  1662
Sudama Public Key:  8056
Pundirka Public Key:  5959
Balram Public Key:  8056

-------- Eavesdropper don't know the shared keys -------

Krishna-Sudama Shared Key:  10466
Pundirka-Balram Shared Key:  1196
Sudama-Balram Shared Key:  8721
Krishna-Balram Shared Key:  10466


-----------------------------------------------


Channels available to eavesdrop:
1. Krishna-Sudama
2. Pundirka-Balram
3. Sudama-Balram
4. Krishna-Balram

Enter your choice: 1


-----------------------------------------------


Eavesdropper listening to Krishna-Sudama Conversational Channel

Available Informations:

Krishna Public Key:  1662
Sudama Public Key:  8056
Prime:  13147
Generator:  5
Message sent by  Krishna  to  Sudama :  [38, 182, 141, 134, 131, 155, 197, 145, 194, 142, 141, 141, 150, 194, 145, 146, 141, 150, 194, 139, 145, 194, 177
38, 151, 144, 145, 135, 140, 194, 166, 131, 134, 131, 197, 145, 194, 138, 141, 151, 145, 135, 204, 25]

Listened Message:  [38, 182, 141, 134, 131, 155, 197, 145, 194, 142, 141, 141, 150, 194, 145, 146, 141, 150, 194, 139, 145, 194, 177, 138, 151, 144, 145,
5, 140, 194, 166, 131, 134, 131, 197, 145, 194, 138, 141, 151, 145, 135, 204, 25]


Eavesdropper can not decrypt the message as he don't know the shared key:


Decrypted Message:  Today's loot spot is Shursen Dada's house.


-----------------------------------------------
```

Modified Diffie Hellman:

```python
import random

butterLootSpots = ['Akrura Bapu', 'Devashrav Kaka', 'Devak Nana', 'Shursen Dada']

class Person:
    def __init__(self, name):
        self.name = name
        self.private_key = 0
        self.public_key = 0

class SecurityExchangeCompany:
    def __init__(self, prime, generator, noise):
        self.prime = prime
        self.generator = generator
        self.noise = noise  # Quirky noise factor

    def requestPublicKey(self, person):
        person.private_key = random.randint(1, 100)
        raw_public_key = pow(self.generator, person.private_key, self.prime)  #
Generate raw public key
        # Add noise to the public key in a consistent way
        person.public_key = (raw_public_key + self.noise) % self.prime
        return person.public_key
```

```
    def computeSharedKey(self, person, public_key):
        # Compute shared key consistently
        # Here, we remove the noise from the public key to ensure the keys match
        public_key_adjusted = (public_key - self.noise + self.prime) % self.prime
        return pow(public_key_adjusted, person.private_key, self.prime)

class Channel:
    def __init__(self, person1, person2, sharedKey, prime, generator):
        self.person1 = person1
        self.person2 = person2
        self.sharedKey = sharedKey % 256
        self.prime = prime
        self.generator = generator

    def messagePassing(self, sender, receiver):
        message = [random.randint(1, 100)]
        temp = "Today's loot spot is " + random.choice(butterLootSpots) + "'s
house."
        random.seed(self.sharedKey)
        for c in temp:
            message.append(ord(c) ^ self.sharedKey ^ random.randint(1, 100))
        message.append(random.randint(1, 100))
        print(f"Message sent by {sender.name} to {receiver.name}: {message}")
        return message

    def decryptMessage(self, message):
        decrypted_message = []
        random.seed(self.sharedKey)
        for c in message[1:-1]:
            decrypted_message.append(chr(c ^ self.sharedKey ^ random.randint(1,
100)))
        return "".join(decrypted_message)

def main():
    prime = 23  # A small prime number for simplicity
    generator = 5  # Generator value
    noise = 2  # Quirky noise factor

    # Initialize participants
    krishna = Person("Krishna")
    sudama = Person("Sudama")

    # Create the security exchange company
    sec = SecurityExchangeCompany(prime, generator, noise)

    # Exchange public keys
    krishna.public_key = sec.requestPublicKey(krishna)
    sudama.public_key = sec.requestPublicKey(sudama)

    print("\nKrishna's Public Key:", krishna.public_key)
```

```
    print("Sudama's Public Key:", sudama.public_key)

    # Compute shared keys with the corrected formula
    krishnaSudamaSharedKey = sec.computeSharedKey(krishna, sudama.public_key)
    sudamaKrishnaSharedKey = sec.computeSharedKey(sudama, krishna.public_key)

    print("\nKrishna-Sudama Shared Key:", krishnaSudamaSharedKey)
    print("Sudama-Krishna Shared Key:", sudamaKrishnaSharedKey)

    # Create a communication channel and exchange messages
    channel = Channel(krishna, sudama, krishnaSudamaSharedKey, prime, generator)

    message = channel.messagePassing(krishna, sudama)
    print("\nEavesdropped Message:", message)

    decrypted_message = channel.decryptMessage(message)
    print("\nDecrypted Message:", decrypted_message)

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

Output:

```
(base) tirthshah@Tirths-MacBook-Pro PDEU-Sem-5 % python3 -u "/Users/tirthshah/Documents/PDEU-Sem-5/Lab/Information Security/Lect9/Modifie

Krishna's Public Key: 10
Sudama's Public Key: 14

Krishna-Sudama Shared Key: 9
Sudama-Krishna Shared Key: 9
Message sent by Krishna to Sudama: [22, 97, 41, 93, 75, 98, 54, 45, 40, 73, 39, 90, 51, 34, 81, 62, 41, 39, 47, 62, 75, 63, 19, 88, 38, 7
, 103, 104, 61, 98, 115, 77, 56, 84, 28, 116, 74, 60, 87]

Eavesdropped Message: [22, 97, 41, 93, 75, 98, 54, 45, 40, 73, 39, 90, 51, 34, 81, 62, 41, 39, 47, 62, 75, 63, 19, 88, 38, 75, 110, 126,
1, 98, 115, 77, 56, 84, 28, 116, 74, 60, 87]

Decrypted Message: Today's loot spot is Akrura Bapu's house.
```

Normal Code

1. **Public Key Calculation**:
    - Public key is calculated using self.generator ** person.private_key % self.prime, which is straightforward but lacks any additional factors.
2. **Shared Key Calculation**:
    - The shared key is computed directly as (publicKey ** person.private_key) % self.prime, which may lead to mismatches if keys aren't consistent.
3. **Message Encryption**:
    - Messages are encrypted using a simple XOR operation with the shared key. No randomness is involved in the process, which may make it easier to decipher.
4. **Message Decryption**:

    - The decryption process skips the first and last elements of the message for randomness but does not incorporate consistent factors for reconstruction.
5. **User Interaction**:
    - The user is asked to choose which conversation channel to eavesdrop on, but the experience is linear and straightforward.

Improved Code

1. **Enhanced Public Key Calculation**:
    - The public key is adjusted with a quirky noise factor ((raw_public_key + self.noise) % self.prime). This adds complexity but ensures the randomness of keys.
2. **Robust Shared Key Calculation**:
    - The shared key computation removes the noise from the public key before calculating it, ensuring that both parties arrive at the same shared key consistently.
3. **Advanced Message Encryption**:
    - Messages are encrypted using XOR operations with both the shared key and a random number, making it less predictable and more secure.
4. **Improved Decryption Process**:
    - Decryption utilizes the same random seed based on the shared key, enabling a more consistent and reliable reconstruction of the original message.
5. **Increased Interaction**:
    - The improved version can provide a more engaging interaction for users by allowing them to select different channels while also highlighting the complexity added to key exchanges.

Summary of Improvements

- **Security**: The use of a noise factor in the public key and consistent removal of that noise when calculating the shared key enhances the security model.
- **Encryption Complexity**: The incorporation of additional randomization in message passing and decryption makes it more difficult for eavesdroppers to decipher messages without knowledge of the shared key.
- **Code Structure**: The separation of concerns in handling public key exchanges and channel management improves code readability and maintenance.

Overall Time Complexity

Combining all the above, the overall time complexity of the **normal** and **improved** versions can be summarized as:

- **For Key Generation**: O(log p) because of power function and binary exponentiation
- **For Shared Key Computation**: O(2 * log p) because two person share their keys
- **For Message Passing and Decryption**: O(n) n is the length of message and encrypting them character by character takes O(n) time.

Thus, the overall complexity is dominated by the shared key computations and would generally be expressed as:

Total Time Complexity: O(2 * log p + n)

Where:

- **k** is the number of participants.
- **p** is the prime number used in the computations.
- **n** is the number of characters in the message.