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):
```

```python
        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
```