PRACTICAL FILE

Name : Ujjawal kumar

College roll no. :20201441

Exam roll no. :20020570038

Subject : Information Security

Course : BSc. (H) Computer Science

Semester : 6th Sem, 3rd Year

Submitted to :Mr. Sahil Sir

Date :30th April,2023

(Ramanujan College)

Q1) Implement the error correcting code.

Code

```
def encode(msg):
    # Determine the number of parity bits needed
    n = len(msg)
    k = 1
    while 2**k < n + k + 1:
        k += 1
    m = n + k

    # Initialize the encoded message
    code = [0] * m

    # Copy the message into the encoded message, skipping parity bit positions
    j = 0
    for i in range(m):
        if i+1 not in [2**p for p in range(k)]:
            code[i] = int(msg[j])
            j += 1

    # Calculate the parity bits
    for p in range(k):
        bit = 0
        for i in range(1, m+1):
            if i & (2**p) == (2**p):
                bit ^= code[-i]
        code[2**p - 1] = bit

    # Return the encoded message
    return ''.join(str(bit) for bit in code)


def decode(code):
    # Determine the number of parity bits used
    m = len(code)
    k = 1
    while 2**k < m + 1:
        k += 1
    n = m - k

    # Initialize the decoded message
    msg = [0] * n

    # Calculate the syndrome
    syndrome = []
    for p in range(k):
        bit = 0
```

```python
        for i in range(1, m+1):
            if i & (2**p) == (2**p):
                bit ^= int(code[-i])
        syndrome.append(bit)
    syndrome.reverse()
    syndrome_num = int(''.join(str(bit) for bit in syndrome), 2)

    # Correct errors
    if syndrome_num != 0:
        code_list = list(code)
        code_list[syndrome_num-1] = str(int(code[syndrome_num-1]) ^ 1)
        code = ''.join(code_list)

    # Copy the message from the decoded code
    j = 0
    for i in range(m):
        if i+1 not in [2**p for p in range(k)]:
            msg[j] = int(code[i])
            j += 1

    # Return the decoded message
    return ''.join(str(bit) for bit in msg)

msg = '1011'
code = encode(msg)
print("Encoded Message:-",code)

decoded_msg = decode(code)
print("Decoded Message:-",decoded_msg)
```

Output

```
PROBLEMS    OUTPUT    TERMINAL    ···                Code  +  ···

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal
r\Desktop\is\1.py"
Encoded Message:- 0110011
Decoded Message:- 1011
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q2) Implement the error detecting code.

Code

```python
def calcRedundantBits(m):

    for i in range(m):
        if(2**i >= m + i + 1):
            return i

def posRedundantBits(data, r):

    # Redundancy bits are placed at the positions
    # which correspond to the power of 2.
    j = 0
    k = 1
    m = len(data)
    res = ''

    # If position is power of 2 then insert '0'
    # Else append the data
    for i in range(1, m + r+1):
        if(i == 2**j):
            res = res + '0'
            j += 1
        else:
            res = res + data[-1 * k]
            k += 1


    return res[::-1]


def calcParityBits(arr, r):
    n = len(arr)

    for i in range(r):
        val = 0
        for j in range(1, n + 1):

            if(j & (2**i) == (2**i)):
                val = val ^ int(arr[-1 * j])
                # -1 * j is given since array is reversed

        arr = arr[:n-(2**i)] + str(val) + arr[n-(2**i)+1:]
    return arr
```

```python
def detectError(arr, nr):
    n = len(arr)
    res = 0

    # Calculate parity bits again
    for i in range(nr):
        val = 0
        for j in range(1, n + 1):
            if(j & (2**i) == (2**i)):
                val = val ^ int(arr[-1 * j])

        # Create a binary no by appending
        # parity bits together.

        res = res + val*(10**i)

    # Convert binary to decimal
    return int(str(res), 2)


# Enter the data to be transmitted
data = '1011001'

# Calculate the no of Redundant Bits Required
m = len(data)
r = calcRedundantBits(m)

# Determine the positions of Redundant Bits
arr = posRedundantBits(data, r)

# Determine the parity bits
arr = calcParityBits(arr, r)

# Data to be transferred
print("Data transferred is " + arr)

# Stimulate error in transmission by changing
# a bit value.
# 10101001110 -> 11101001110, error in 10th position.

arr = '11101001110'
print("Error Data is " + arr)
correction = detectError(arr, r)
if(correction==0):
    print("There is no error in the received message.")
else:
    print("The position of error is ",len(arr)-correction+1,"from the left")
```

Output

```
PROBLEMS    OUTPUT    TERMINAL    ...                Code  +  ...

● PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal ku
  r\Desktop\is\2.py"
  Data transferred is 10101001110
  Error Data is 11101001110
  The position of error is  2 from the left
○ PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q3) Implement caeser cipher substitution operation.

Code

```python
def caesar_cipher(message, key):
    """
    Applies the Caesar cipher substitution operation to the given message with
the given key.
    """
    cipher_text = ""

    # Iterate over each character in the message.
    for char in message:
        # Check if the character is a letter.
        if char.isalpha():
            # Shift the character by the key amount.
            shifted_char = chr((ord(char.lower()) - 97 + key) % 26 + 97)

            # Preserve the case of the original character.
            if char.isupper():
                cipher_text += shifted_char.upper()
            else:
                cipher_text += shifted_char
        else:
            # Preserve non-letter characters as-is.
            cipher_text += char

    return cipher_text

message = "hello duniya"
key = 3
cipher_text = caesar_cipher(message, key)
```

```python
print("Original message:", message)
print("Cipher text:", cipher_text)

print("...........Decryption.........")
cipher_text = "Khoor gxqlbd"
shift = 3

plain_text = caesar_cipher(cipher_text, -shift)
print("Plain Text:", plain_text)
```

Output

```
PROBLEMS    OUTPUT    TERMINAL    ...                    Code  +  ∨  ⊞  🗑  ...

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal k
r\Desktop\is\3.py"
Original message: hello duniya
Cipher text: khoor gxqlbd
............Decryption.........
Plain Text: Hello duniya
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q4) Implement monoalphabetic and polyalphabetic cipher substitution operation.

Code

MonoAlphabetic

```python
import random
alpha = "abcdefghijklmnopqrstuvwxyz"

#Encrypts the plain text message

def encrypt(original, key=None):
    if key is None:
        l = list(alpha)
        random.shuffle(l)
        key = "".join(l)
        new = []
    for letter in original:
        new.append(key[alpha.index(letter)])
```

```
    return ["".join(new), key]

#Decrypts the encrypted message

def decrypt(cipher, key=None):
    if key is not None:
        new = []
    for letter in cipher:
        new.append(alpha[key.index(letter)])
    return "".join(new)



e = encrypt("monoalphabetic", None)

print("Encrypted Message:",e) #Prints encrypted message

print("Decrypted Message:",decrypt(e[0], e[1])) #Decodes the message and
prints it
```

Polyalphabetic

```
# Define the Vigenère cipher function
def vigenere_cipher(plaintext, key):
    ciphertext = ""
    key_index = 0
    for letter in plaintext:
        if letter.isalpha():
            # Convert the letter to its alphabetical index (A=0, B=1, etc.)
            letter_index = ord(letter.upper()) - ord('A')

            # Convert the key letter to its alphabetical index
            key_letter = key[key_index % len(key)]
            key_index += 1
            key_index %= len(key)
            key_letter_index = ord(key_letter.upper()) - ord('A')

            # Add the letter and key letter indices, and take the result mod
26
            cipher_index = (letter_index + key_letter_index) % 26

            # Convert the cipher index back to a letter and append it to the
ciphertext
            cipher_letter = chr(cipher_index + ord('A'))
            ciphertext += cipher_letter
        else:
```

```
            # If the character isn't a letter, just append it to the
ciphertext
            ciphertext += letter
    return ciphertext


# Test the function
plaintext = "HELLO WORLD"
key = "KEY"
ciphertext = vigenere_cipher(plaintext, key)
print("Ciphertext:-",ciphertext)


# Define the Vigenère cipher decryption function
def vigenere_decipher(ciphertext, key):
    plaintext = ""
    key_index = 0
    for letter in ciphertext:
        if letter.isalpha():
            # Convert the letter to its alphabetical index (A=0, B=1, etc.)
            letter_index = ord(letter.upper()) - ord('A')

            # Convert the key letter to its alphabetical index
            key_letter = key[key_index % len(key)]
            key_index += 1
            key_index %= len(key)
            key_letter_index = ord(key_letter.upper()) - ord('A')

            # Subtract the key letter index from the cipher index, and take
the result mod 26
            cipher_index = (letter_index - key_letter_index) % 26

            # Convert the cipher index back to a letter and append it to the
plaintext
            plain_letter = chr(cipher_index + ord('A'))
            plaintext += plain_letter
        else:
            # If the character isn't a letter, just append it to the plaintext
            plaintext += letter
    return plaintext


# Test the function
ciphertext = "RIJVS UYVJN"
key = "KEY"
plaintext = vigenere_decipher(ciphertext, key)
print("Plaintext:-",plaintext)
```

Output

Monoalphabetic

```
PROBLEMS    OUTPUT    TERMINAL    ...                Code  +  ∨

● PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kuma
  r\Desktop\is\4.py"
  Encrypted Message: ['djojbhfcbwmsxy', 'bwyzmetcxrqhdojfugkspnliav']
  Decrypted Message: monoalphabetic
○ PS C:\Users\Ujjawal kumar\Desktop\is>
```

Polyalphabetic

```
PROBLEMS    OUTPUT    TERMINAL    ...                Code  +  ∨

● PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kuma
  r\Desktop\is\4.1.py"
  Ciphertext:- RIJVS UYVJN
  Plaintext:- HELLO WORLD
○ PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q5) Implement playfair cipher substitution operation.

Code

```
def toLowerCase(text):
    return text.lower()

# Function to remove all spaces in a string


def removeSpaces(text):
    newText = ""
```

```python
    for i in text:
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText

# Function to group 2 elements of a string
# as a list element


def Diagraph(text):
    Diagraph = []
    group = 0
    for i in range(2, len(text), 2):
        Diagraph.append(text[group:i])

        group = i
    Diagraph.append(text[group:])
    return Diagraph

# Function to fill a letter in a string element
# If 2 letters in the same string matches


def FillerLetter(text):
    k = len(text)
    if k % 2 == 0:
        for i in range(0, k, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    return new_word


list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

```
# Function to generate the 5x5 key square matrix


def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
            key_letters.append(i)

    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)

    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]

    return matrix


def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j


def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]

    char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
    else:
        char2 = matr[e2r][e2c+1]

    return char1, char2


def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):
```

```
    char1 = ''
    if e1r == 4:
        char1 = matr[0][e1c]
    else:
        char1 = matr[e1r+1][e1c]

    char2 = ''
    if e2r == 4:
        char2 = matr[0][e2c]
    else:
        char2 = matr[e2r+1][e2c]

    return char1, char2


def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    char1 = matr[e1r][e2c]

    char2 = ''
    char2 = matr[e2r][e1c]

    return char1, char2


def encryptByPlayfairCipher(Matrix, plainList):
    CipherText = []
    for i in range(0, len(plainList)):
        c1 = 0
        c2 = 0
        ele1_x, ele1_y = search(Matrix, plainList[i][0])
        ele2_x, ele2_y = search(Matrix, plainList[i][1])

        if ele1_x == ele2_x:
            c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
            # Get 2 letter cipherText
        elif ele1_y == ele2_y:
            c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x,
ele2_y)
        else:
            c1, c2 = encrypt_RectangleRule(
                Matrix, ele1_x, ele1_y, ele2_x, ele2_y)

        cipher = c1 + c2
        CipherText.append(cipher)
    return CipherText


text_Plain = 'instruments'
```

```python
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)

print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ""
for i in CipherList:
    CipherText += i
print("CipherText:", CipherText)
```

Output

```
PROBLEMS    OUTPUT    TERMINAL    ...

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kumar\Desktop\is\5.py"
Key text: Monarchy
Plain Text: instruments
CipherText: gatlmzclrqtx
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q6) Implement hill cipher substitution operation.

Code

```python
keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
```

```python
# key matrix for the key string
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

# Following function encrypts the message
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                    messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

    # Get key matrix from the key string
    getKeyMatrix(key)

    # Generate vector for the message
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    # Following function generates
    # the encrypted vector
    encrypt(messageVector)

    # Generate the encrypted text
    # from the encrypted vector
    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

    # Finally print the ciphertext
    print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

    # Get the message to
    # be encrypted
    message = "ACT"

    # Get the key
```

```python
    key = "GYBNQKURP"

    HillCipher(message, key)

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

Output

```
PROBLEMS    OUTPUT    TERMINAL    ...

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kuma
r\Desktop\is\6.py"
Ciphertext:  POH
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q7) Implement rail fence cipher transposition operation.

Code

```python
# function to encrypt a message
def encryptRailFence(text, key):


    rail = [['\n' for i in range(len(text))]
                for j in range(key)]

    # to find the direction
    dir_down = False
    row, col = 0, 0

    for i in range(len(text)):

        # check the direction of flow
        # reverse the direction if we've just
        # filled the top or bottom rail
        if (row == 0) or (row == key - 1):
            dir_down = not dir_down
```

```
        # fill the corresponding alphabet
        rail[row][col] = text[i]
        col += 1

        # find the next row using
        # direction flag
        if dir_down:
            row += 1
        else:
            row -= 1
    # now we can construct the cipher
    # using the rail matrix
    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
    return("" . join(result))

# This function receives cipher-text
# and key and returns the original
# text after decryption
def decryptRailFence(cipher, key):

    # create the matrix to cipher
    # plain text key = rows ,
    # length(text) = columns
    # filling the rail matrix to
    # distinguish filled spaces
    # from blank ones
    rail = [['\n' for i in range(len(cipher))]
                for j in range(key)]

    # to find the direction
    dir_down = None
    row, col = 0, 0

    # mark the places with '*'
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key - 1:
            dir_down = False

        # place the marker
        rail[row][col] = '*'
        col += 1
```

```
        # find the next row
        # using direction flag
        if dir_down:
            row += 1
        else:
            row -= 1

    # now we can construct the
    # fill the rail matrix
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((rail[i][j] == '*') and
            (index < len(cipher))):
                rail[i][j] = cipher[index]
                index += 1

    # now read the matrix in
    # zig-zag manner to construct
    # the resultant text
    result = []
    row, col = 0, 0
    for i in range(len(cipher)):

        # check the direction of flow
        if row == 0:
            dir_down = True
        if row == key-1:
            dir_down = False

        # place the marker
        if (rail[row][col] != '*'):
            result.append(rail[row][col])
            col += 1

        # find the next row using
        # direction flag
        if dir_down:
            row += 1
        else:
            row -= 1
    return("".join(result))

# Driver code
if __name__ == "__main__":
    print(encryptRailFence("attack at once", 2))
    print(encryptRailFence("helloworld ", 3))
    print(encryptRailFence("defend the west wall", 3))
```

```
# Now decryption of the
# same cipher-text
print(decryptRailFence("holelwrdlo ", 3))
print(decryptRailFence("atc toctaka ne", 2))
print(decryptRailFence("dnheweedtews alf  tl", 3))
```

Output

```
PROBLEMS    OUTPUT    TERMINAL    ...                    Code  +  

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kuma
r\Desktop\is\7.py"
atc toctaka ne
holelwrdlo
dnheweedtews alf  tl
helloworld
attack at once
defend the west wall
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q8) Implement row transposition cipher transposition operation.

Code

```
import math
key=input("Enter keyword text (Contains unique letters only):
").lower().replace(" ", "")
plain_text = input("Enter plain text (Letters only): ").lower().replace(" ",
"")

len_key = len(key)
len_plain = len(plain_text)
row = int(math.ceil(len_plain / len_key))
matrix = [ ['X']*len_key for i in range(row) ]

# print(matrix)


t = 0
for r in range(row):
  for c,ch in enumerate(plain_text[t : t+ len_key]):
    matrix[r][c] = ch
  t += len_key
```

```
# print(matrix)
sort_order = sorted([(ch,i) for i,ch in enumerate(key)])  #to make
alphabetically order of chars
# print(sort_order)

cipher_text = ''
for ch,c in sort_order:
  for r in range(row):
    cipher_text += matrix[r][c]

print("Encryption")
print("Plain text is :",plain_text)
print("Cipher text is:",cipher_text)



matrix_new = [ ['X']*len_key for i in range(row) ]
key_order = [ key.index(ch) for ch in sorted(list(key))]  #to make original
key order when we know keyword
# print(key_order)

t = 0
for c in key_order:
  for r,ch in enumerate(cipher_text[t : t+ row]):
    matrix_new[r][c] = ch
  t += row
# print(matrix_new)

p_text = ''
for r in range(row):
  for c in range(len_key):
    p_text += matrix_new[r][c] if matrix_new[r][c] != 'X' else ''

print("Decryption")
print("Cipher text is:",cipher_text)
print("Plain text is :",p_text)
```

Output

```
PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kumar\Desktop\is\8.py"
Enter keyword text (Contains unique letters only): son
Enter plain text (Letters only): hello guys yellow raam khet
Encryption
Plain text is : helloguysyellowraamkhet
Cipher text is: lgslwahXeoyeoakthluylrme
Decryption
Cipher text is: lgslwahXeoyeoakthluylrme
Plain text is : helloguysyellowraamkhet
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q9) Implement product cipher transposition operation.

Code

```
import random, string, sys
import math


#A custom character map table of 65 characters and which are mapped in 65 int
range
char_std_65 = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
'8': 8, '9': 9,
                'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15, 'G': 16,
'H': 17, 'I': 18,
                'J': 19, 'K': 20, 'L': 21, 'M': 22, 'N': 23, 'O': 24, 'P': 25,
'Q': 26, 'R': 27,
                'S': 28, 'T': 29, 'U': 30, 'V': 31, 'W': 32, 'X': 33, 'Y': 34,
'Z': 35, 'a': 36, 'b': 37,
                'c': 38, 'd': 39, 'e': 40, 'f': 41, 'g': 42, 'h': 43, 'i': 44,
'j': 45, 'k': 46, 'l': 47,
                'm': 48, 'n': 49, 'o': 50, 'p': 51, 'q': 52, 'r': 53,'s': 54,
't': 55, 'u': 56, 'v': 57,
                'w': 58, 'x': 59, 'y': 60, 'z': 61, ' ': 62, ',': 63, '.': 64}

def _getKey(keyName):
    '''
        Function for retrieving character from the char-map table using it's
numeric value
    '''
```

```
    return list(char_std_65.keys())[list(char_std_65.values()).index(keyName)]

class Encryption:
    '''
        MCA0135 Product cipher
    '''
    plain_text = ''
    key = ''
    transposition_key = ''

    def __init__(self, plain_text, key, transposition_key):
        self.plain_text = plain_text
        self.key = key
        self.transposition_key = transposition_key

    def addRoundKey(self, plain_text):
        '''
           The addRoundKey function will xor plain text with key in character
level,
           Then the xore value is wrapped between 0 and 65 to match with our
finite 65 character map table'''
        xored = []
        for i in range(0, len(plain_text)):
            char_in_pt = char_std_65[plain_text[i]]
            char_in_key = char_std_65[self.key[i]]
            xored_value = _getKey((char_in_pt ^ char_in_key) % 65)

            xored.append(xored_value)
        return ''.join(xored)

    def oneTimePad(self, message):
        '''
            The One-Time Pad encrypt function will encrypt a message using the
randomly generated private key that is then decrypted by the receiver using a
matching one-time pad and key
        '''
        cipher = ''
        for c in range(0, len(self.key)):
            #Sum of key and message value is wrapped between 0 and 65 to use
our finite char field
            subst_value = (char_std_65[message[c]] + char_std_65[self.key[c]])
% 65
            cipher = cipher + _getKey(subst_value)
        return cipher

    def rowTransposition(self, message):
        # Each string in ciphertext represents a column in the grid.
        cipher_text = [''] * self.transposition_key
```

```
        # Loop through each column in ciphertext.
        for col in range(self.transposition_key):
            pointer = col
            # Keep looping until pointer goes past the length of the message
            while pointer < len(message):
                # Place the character at pointer in message at the end of the
                # current column in the ciphertext list.
                cipher_text[col] += message[pointer]
                # move pointer over
                pointer += self.transposition_key
        return ''.join(cipher_text)

    def railFenceCipher(self, message):
        '''
            The railFenceCipher function will write message letters out
diagonally
            over a number of rows. Then read off cipher by row.
        '''
        upper_row = ''
        lower_row = ''
        for m in range(1, len(message)+1):
            #Here we are reading from the grid with two rows but usually
            #as many rows as the key is, and as many columns as the length of
the ciphertext.
            if (m % 2 != 0):
                upper_row = upper_row + message[m-1]
            else:
                lower_row = lower_row + message[m-1]
        return upper_row + lower_row

    def endToEndEncryptionProcess(self):
        '''
           The endToEndEncryptionProcess function will execute the whole end
to end execution of
           the algorithm round by round and provide the cipher text.
        '''
        cipher_text = self.addRoundKey(self.plain_text)
        encry_logs = []
        encry_logs.append('Cipher text after addRoundkey:
"{}"'.format(cipher_text))
        '''
           first round - substitution
        '''
        cipher_text = self.oneTimePad(cipher_text)
        encry_logs.append('cipher text after first round(one-time pad):
"{}"'.format(cipher_text))
        '''
           second round - transposition
```

```
        '''
        cipher_text = self.rowTransposition(cipher_text)
        encry_logs.append('Cipher text after rowTransposition in the second 
round: "{}"'.format(cipher_text))
        cipher_text = self.railFenceCipher(cipher_text)
        encry_logs.append('Final cipher text after railFenceCipher in the 
second round: "{}"'.format(cipher_text))
        _log('ENCRYPTION', encry_logs)
        return cipher_text

class Decryption:
    cipher_text = ''
    key = ''
    transposition_key = ''

    def __init__(self, cipher_text, key, transposition_key):
        self.cipher_text = cipher_text
        self.key = key
        self.transposition_key = transposition_key

    def reverseRailFenceCipher(self, message):
        '''
            The reverseRailFenceCipher function will decrypt the message.
        '''
        #The middle index for splitting the cipher
        split_index = int(len(message)/2 + 1) if len(message) % 2 != 0 else 
int(len(message)/2)
        reverse_text = ''
        for i in range(0, split_index):
            #Reads the character from the first half and the second half in a
            reverse_text = reverse_text + message[i]
            if (split_index + i) <= len(message)-1:
                reverse_text = reverse_text + message[split_index + i]
        return reverse_text

    def reverseRowTransposition(self, message):
        '''
         The transposition decrypt function will simulate the "columns" and
         "rows" of the grid that the plaintext is written on by using a list
         of strings.
        '''
        #The number of "columns" in our transposition grid:
        numOfColumns = math.ceil(len(message) / self.transposition_key)
        # The number of "rows" in our grid will need:
        numOfRows = self.transposition_key
        # The number of "shaded boxes" in the last "column" of the grid:
        numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
        # Each string in plaintext represents a column in the grid.
```

```
        plaintext = [''] * numOfColumns
        # The col and row variables point to where in the grid the next
character in the encrypted message will go.
        col = 0
        row = 0

        for symbol in message:
            plaintext[col] += symbol
            col += 1 # point to next column
            # If there are no more columns OR we're at a shaded box, go back
to the first column and the next row.
            if (col == numOfColumns) or (col == numOfColumns - 1 and row >=
numOfRows - numOfShadedBoxes):
                col = 0
                row += 1
        return ''.join(plaintext)

    def reverseOneTimePad(self, message):
        plain_text = ''
        for c in range(0, len(self.key)):
            rev_value = (char_std_65[message[c]] + 65) -
char_std_65[self.key[c]]
            if rev_value > 65:
                rev_value = (char_std_65[message[c]] -
char_std_65[self.key[c]])
            plain_text = plain_text + _getKey(rev_value)
        return plain_text

    def reverseAddRoundKey(self, message):
        xored = []
        for i in range(0, len(message)):
            char_in_ct = char_std_65[message[i]]
            char_in_key = char_std_65[self.key[i]]
            if char_in_key == 65 or char_in_key == char_in_ct:
                xored_value = _getKey((char_in_ct + 65 ^ char_in_key))
            else:
                xored_value = _getKey((char_in_ct ^ char_in_key))
            xored.append(xored_value)
        return ''.join(xored)

    def endToEndDecryptionProcess(self):
        rev_text = self.reverseRailFenceCipher(self.cipher_text)
        decry_logs = []
        decry_logs.append('Cipher text after reverseRailFenceCipher operation:
"{}"'.format(rev_text))
        rev_text = self.reverseRowTransposition(rev_text)
        decry_logs.append('Cipher text after reverseRowTransposition
operation: "{}"'.format(rev_text))
```

```
        rev_text = self.reverseOneTimePad(rev_text)
        decry_logs.append('Cipher text after reverseOneTimePad operation:
"{}"'.format(rev_text))
        rev_text = self.reverseAddRoundKey(rev_text)
        decry_logs.append('Plain text after reverseAddRoundKey operation:
"{}"'.format(rev_text))
        _log('DECRYPTION' ,decry_logs)

def _log(title, content):
    '''
       Function for logging all the traces in a wrapped box.
    '''
    msg_size = max(len(word) for word in content) #msg_size/2
    msg_half_size = int((msg_size/2)+1) if msg_size % 2 !=0 else
int(msg_size/2)
    title_size = len(title)
    title_half_size = int(title_size/2)+1 if title_size % 2 !=0 else
int(title_size/2)
    title_pos = (msg_half_size-title_half_size)
    print('+'+'-' * (msg_size + 2)+'+')
    print('|{}{}{}|'.format(' '*(msg_half_size-title_half_size),title, '
'*(msg_size-(title_pos+title_size)+2)))
    for word in content:
        print('| {:<{}} |'.format(word, msg_size))
    print('+'+'-' * (msg_size + 2)+'+')

if __name__ == '__main__':
    plain_text = input('Please enter a message for encryption:')
    key = ''.join(random.choice(string.ascii_uppercase +
string.ascii_lowercase + string.digits) for _ in range(len(plain_text)))
    row_transposition_key = random.randrange(2 , (int(len(key)/2)+1))
    encryption = Encryption(plain_text, key, row_transposition_key)
    print('Plain message for encryption: "{}" & Key: "{}"'.format(plain_text,
key)) #& rowTransposition key: {}
    cipher_text = encryption.endToEndEncryptionProcess()
    decryption = Decryption(cipher_text, key, row_transposition_key)
    decryption.endToEndDecryptionProcess()
```

Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    ...

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kumar\Desktop\is\9.
y"
Please enter a message for encryption:hi i am ujjawal
Plain message for encryption: "hi i am ujjawal" & Key: "PpI87YexcIQm8CZ"
+--------------------------------------------------------------------------------+
|                                  ENCRYPTION                                     |
| Cipher text after addRoundkey: "oViav6O5U,tKoeC"                                |
   |
| Cipher text after reverseRailFenceCipher operation: "AeGH.3 .wi3q.Gl"  |
| Cipher text after reverseRowTransposition operation: "AH i.e..3GG3wql" |
| Cipher text after reverseOneTimePad operation: "oViav6O5U,tKoeC"       |
| Plain text after reverseAddRoundKey operation: "hi i am ujjawal"       |
+------------------------------------------------------------------------+
PS C:\Users\Ujjawal kumar\Desktop\is>
```

Q11) Implement a stream cipher technique.

Code

```python
# Define a secret key
key = b'mysecretkey'

# Define the plaintext message to be encrypted
message = b'This is my secret message.'

# Convert the key and message to binary arrays
key = bytearray(key)
message = bytearray(message)

# Generate the keystream by repeating the key
keystream = bytearray()
while len(keystream) < len(message):
    keystream += key

# Truncate the keystream to the length of the message
keystream = keystream[:len(message)]

# Encrypt the message by XORing it with the keystream
ciphertext = bytearray()
for i in range(len(message)):
    ciphertext.append(message[i] ^ keystream[i])

# Print the encrypted message
print(ciphertext)
```

```python
# Decrypt the message by XORing it with the keystream again
decrypted_message = bytearray()
for i in range(len(ciphertext)):
    decrypted_message.append(ciphertext[i] ^ keystream[i])

# Print the decrypted message
print(decrypted_message)
```

Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    CODEWHISPERER REFERENCE LOG    Code

PS C:\Users\Ujjawal kumar\Desktop\is> python -u "c:\Users\Ujjawal kumar\Desktop\is\11.py"
bytearray(b'9\x11\x1a\x16C\x1b\x16T\x06\x1cY\x1e\x1c\x10\x17\x06\x06E\x19\x0e\x16\n\x0c\x1e\x16K')
bytearray(b'This is my secret message.')
PS C:\Users\Ujjawal kumar\Desktop\is>
```