NIST COLLEGE

Banepa, Kavre

Bachelor of Computer Science & Information Technology

Fifth Semester

Example Programs on Cryptography

Compiled by

Er. Dinesh Ghemosu

Instructions to Students

1. Students are required to bring an initial lab sheet on the lab period. An initial lab sheet should contain:
    i. A Lab Number
    ii. A Lab Title
    iii. Objectives
    iv. Theory
    v. Algorithms
    vi. Pseudo codes
2. Students are obliged to bring required own materials, notes, books, laptop(if you have, that will be better), pencil, pen, eraser etc.
3. After finishing the lab of that respective day, students need to bring final lab sheet which contains outputs, discussions and conclusion in next lab period. Late submission will not be entertained.
4. Students are expected to maintain discipline in the lab.
5. Students need to appear in lab exam for lab evaluation.

LAB-1

FAMILIARIZATION CLASSICAL CIPHER

Objectives:

1. To encrypt and decrypt a Ceaser cipher.
2. To perform cyytoanalysis of Ceaser cipher.
3. To encrypt and decrypt a message using Monoalphabetic cipher.
4. To encrypt and decrypt a message using Playfair technique.

Program 1

Program 1: WAP to reverse an input message.

```
# Reverse Cipher
msg = input("Input your message: ")
n = len(msg)
reverse_msg = ''
i = n - 1
while (i >= 0):
    reverse_msg = reverse_msg + msg[i]
    i = i-1
print("Revers cipher of message is: ", reverse_msg)
```

Program 2

Program 1: WAP to encrypt a message using Julis Ceaser Cipher.

```
alpha ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print("Lenght of alpha: {}".format(len(alpha)))

# input in capitalize
str_in = input("Enter a word, like HELLO:")
print("str_in = ", str_in)

msg_cipher = " "
n = len(str_in)
print("n =", n)

for i in range(n):
    c = str_in[i]
    loc = alpha.find(c)
    print(i , c, loc) #This line can be omitted, used only to see intermediate result
    newloc = (loc+3)% 26
    msg_cipher += alpha[newloc]

print(:Plain text: ", str_in)
print("Cipher text: ", msg_cipher)
```

Listing 1: Ceaser Cipher.

Program 3

Program 3: WAP to encrypt a message using Ceaser Cipher with shift value.

```python
alpha ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print("Lenght of alpha: {}".format(len(alpha)))

# input a word and it will change to uppercase
str_in = input("Enter a word, like HELLO:").upper()

# input a number in integer
shift = int(input("Input Shift value, like 3: "))

n = len(str_in)
msg_cipher = " "

for i in range(n):
    c = str_in[i]
    loc = alpha.find(c)
    print(i , c, loc) # used to see intermediate result, can be omitted
    newloc = (loc+shift) % 26
    msg_cipher += alpha[newloc]

print("Plain text: ", str_in)
print("Shift Value: ", shift)
print("Cipher text: ", msg_cipher)
```

Listing 2: Ceaser Cipher with shift value.

Program 4

Program 4: WAP to encrypt and decrypt a message using Ceaser Cipher.

```python
# Ceaser Cipher
alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

# define a functionn that checks if input message contains alhabets
    onlys
def contains_only_alphabets(input_str):
    for char in input_str:
        if not char.isalpha():
            return False
    return True

#input plaintext containing only alphabets
while True:
    try:
        msg = input("Input message: ")

        #checking if input string contains alphabets only:dines
        if contains_only_alphabets(msg):
            break
        else:
            print("Input message does not contain alphabets only!!.")
    except ValueError:
        print("Input is not an alphabetic. Try again!!!")

#input a postive integer (between 1 to 26) as key(shift)
while True:
    try:
        key = int(input("Enter a number between 1 and 26: "))
        if 1 <= key <= 26:
```

```
                break  # Exit the loop if the input is valid
            else:
                print("Input is not between 1 and 26. Try again.")
        except ValueError:
            print("Input is not an integer. Try again.")

def ceasar_encryption(text, shift):
    encrypted_text =''

    #changing text to upper (or lower) case
    text = text.upper()

    #lenght of text
    n = len(text)

    #encrypt text
    for i in range(n):
        c = text[i]
        loc = alpha.find(c)
        newloc = (loc + shift) % 26
        encrypted_text += alpha[newloc]

    return encrypted_text


def ceasar_decryption(encrypted_text, shift):
    decrypted_text =''


    #lenght of encrypted_text
    n = len(encrypted_text)

    #decrypt text
    for i in range(n):
        c = encrypted_text[i]
        loc = alpha.find(c)
        newloc = (loc - shift) % 26
        decrypted_text += alpha[newloc]

    return decrypted_text

ciphertext = ceasar_encryption(msg, key)
decrypted_text = ceasar_decryption(ciphertext, key)
print("Plaintext: ", msg)
print("Ciphertext: ", ciphertext)
print("Decrypted text: ", decrypted_text)
```

Program 5

Program 5: WAP to perform bruteforce of a Ceaser cipher.

```
#caesar brute force

alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def ceasar_decryption(encrypted_text, shift):
    decrypted_text =''
```

```python


    #lenght of encrypted_text
    n = len(encrypted_text)

    #decrypt text
    for i in range(n):
        c = encrypted_text[i]
        loc = alpha.find(c)
        newloc = (loc - shift) % 26
        decrypted_text += alpha[newloc]

    return decrypted_text

def caesar_brute_force(ciphertext):
    for shift in range(26):
        decrypted_text = ceasar_decryption(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_text}")

#Example usage
ciphertext = 'ZRUOG'
caesar_brute_force(ciphertext)
```

Program 6

Program 6: WAP to perform ecncryption and decryption using Ceaser cipher.

```python
def caesar_encrypt(text, shift):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            char_code = ord(char) #return unicode of character
            encrypted_char = chr(((char_code - 97 + shift) % 26) + 97)
            if is_upper:
                encrypted_char = encrypted_char.upper()
            encrypted_text += encrypted_char
        else:
            encrypted_text += char
    return encrypted_text

def caesar_decrypt(encrypted_text, shift):
    decrypted_text = ""
    for char in encrypted_text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            char_code = ord(char)
            decrypted_char = chr(((char_code - 97 - shift) % 26) + 97)
            if is_upper:
                decrypted_char = decrypted_char.upper()
            decrypted_text += decrypted_char
        else:
            decrypted_text += char
    return decrypted_text

```

```python
# Example usage:
plaintext = input("Input a message: ")
shift = 3 #change this value (1-26)
encrypted_text = caesar_encrypt(plaintext, shift)
decrypted_text = caesar_decrypt(encrypted_text, shift)

print("Original Text:", plaintext)
print("Cipher Text:", encrypted_text)
print("Decrypted Text:", decrypted_text)
```

Program 7

Program 7: WAP to perform ecncryption and decryption using Monoalphabetic cipher.

```python
import random

# Define the alphabet
alphabet = "abcdefghijklmnopqrstuvwxyz"

# Create a random permutation of the alphabet
random.seed(42)  # You can change the seed for a different permutation
cipher_alphabet = list(alphabet)
random.shuffle(cipher_alphabet)
cipher_alphabet = ''.join(cipher_alphabet)

print("Alphabet: ", alphabet)
print("Shuffled alphabet:", cipher_alphabet)

# Create a dictionary for the encryption and decryption mappings
encryption_mapping = {}
decryption_mapping = {}

for i in range(len(alphabet)):
    encryption_mapping[alphabet[i]] = cipher_alphabet[i]
    decryption_mapping[cipher_alphabet[i]] = alphabet[i]

print("Encryption mapping:\n ", encryption_mapping)
print("Decrytion mapping: \n", decryption_mapping)

# Encrypt a message using the monoalphabetic cipher


def encrypt(message):
    encrypted_message = ""

    for char in message:
        if char.isalpha():
            # Preserve the case of the character
            if char.isupper():
                encrypted_message += encryption_mapping[char.lower()].upper()
            else:
                encrypted_message += encryption_mapping[char]
        else:
            encrypted_message += char

    return encrypted_message
```

```python


# Decrypt a message using the monoalphabetic cipher
def decrypt(ciphertext):
    decrypted_message = ""

    for char in ciphertext:
        if char.isalpha():
            # Preserve the case of the character
            if char.isupper():
                decrypted_message += decryption_mapping[char.lower()].upper()
            else:
                decrypted_message += decryption_mapping[char]
        else:
            decrypted_message += char

    return decrypted_message


# read message to encrpt:
message = input("Input message to encrypt: ")

# Encryption
cipher_text = encrypt(message)

# Decryption
decrypted_message = decrypt(cipher_text)

print("Plaintext: ", message)
print("Cipher text: ", cipher_text)
print("Decrypted text: ", decrypted_message)
```

Program 8

Program 8: WAP to perform ecncryption and decryption Playfair method.

```python
def create_playfair_matrix(key):
    key = key.replace(" ", "").upper()
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  # The letter 'J' is omitted

    # Initialize the Playfair matrix with zeros
    matrix = [['' for _ in range(5)] for _ in range(5)]

    # Fill the matrix with the key
    used_letters = set()
    row, col = 0, 0

    for letter in key:
        if letter not in used_letters:
            matrix[row][col] = letter
            used_letters.add(letter)
            col += 1
            if col == 5:
                col = 0
                row += 1

```

```
    # Fill the remaining empty cells with the alphabet
    for letter in alphabet:
        if letter not in used_letters:
            matrix[row][col] = letter
            used_letters.add(letter)
            col += 1
            if col == 5:
                col = 0
                row += 1

    return matrix


def prepare_text(text):
    text = text.replace(" ", "").upper()
    # Replace 'J' with 'I'
    text = text.replace("J", "I")
    # Break the text into digrams
    digrams = []
    i = 0
    while i < len(text):
        if i + 1 < len(text) and text[i] == text[i + 1]:
            digrams.append(text[i] + 'X')
            i += 1
        else:
            digrams.append(text[i] + text[i + 1])
            i += 2
    return digrams


def encrypt_playfair(plaintext, key):
    matrix = create_playfair_matrix(key)
    plaintext = prepare_text(plaintext)
    ciphertext = []

    for digram in plaintext:
        row1, col1 = None, None
        row2, col2 = None, None

        # Find the positions of the two letters in the matrix
        for row in range(5):
            for col in range(5):
                if matrix[row][col] == digram[0]:
                    row1, col1 = row, col
                if matrix[row][col] == digram[1]:
                    row2, col2 = row, col

        # Handle the same row or column case
        if row1 == row2:
            col1 = (col1 + 1) % 5
            col2 = (col2 + 1) % 5
        elif col1 == col2:
            row1 = (row1 + 1) % 5
            row2 = (row2 + 1) % 5
        else:
            col1, col2 = col2, col1  # Swap columns

        ciphertext.append(matrix[row1][col1] + matrix[row2][col2])
```

```python

    return ''.join(ciphertext)


def decrypt_playfair(ciphertext, key):
    matrix = create_playfair_matrix(key)
    ciphertext = prepare_text(ciphertext)
    plaintext = []

    for digram in ciphertext:
        row1, col1 = None, None
        row2, col2 = None, None

        # Find the positions of the two letters in the matrix
        for row in range(5):
            for col in range(5):
                if matrix[row][col] == digram[0]:
                    row1, col1 = row, col
                if matrix[row][col] == digram[1]:
                    row2, col2 = row, col

        # Handle the same row or column case
        if row1 == row2:
            col1 = (col1 - 1) % 5
            col2 = (col2 - 1) % 5
        elif col1 == col2:
            row1 = (row1 - 1) % 5
            row2 = (row2 - 1) % 5
        else:
            col1, col2 = col2, col1  # Swap columns

        plaintext.append(matrix[row1][col1] + matrix[row2][col2])

    return ''.join(plaintext)


# Example usage
key = "KEYWORD"
plaintext = "HELLO WORLD"
ciphertext = encrypt_playfair(plaintext, key)
decrypted_text = decrypt_playfair(ciphertext, key)

print("Original Text:", plaintext)
print("Encrypted Text:", ciphertext)
print("Decrypted Text:", decrypted_text)
```

LAB-2

FAMILIARIZATION HILL CIPHER

Objectives:

1. To be familiar with matrix operations, multiplication determinant, and inverse.
2. To encrypt a text using Hill cipher.

Program 9

Program 9: WAP to find determinant and inverse of a matrix using python library.

```
import numpy as np

# Define your matrix as a 2D NumPy array
# matrix = np.array([[2, 3, 1],
#                     [4, 1, 7],
#                     [0, 5, 2]])


matrix = np.array([[17, 17, 5],
                   [21, 18, 21],
                   [2, 2, 19]])


def determinant_matrix(matrix):
    # Calculate the determinant
    determinant = np.linalg.det(matrix)
    return determinant


def inverse_matrix(matrix):
    # Use the np.linalg.inv() function to find the inverse
    inverse_matrix = np.linalg.inv(matrix)
    return inverse_matrix


print("Determinant of a matrix:\n", determinant_matrix(matrix))
print("Inverse of a matrix:\n", inverse_matrix(matrix))
```

Program 10

Program 10: WAP to find determinant of a matrix.

```
'''
# 3 by 3 matrix
| a  b  c |
| d  e  f |
| g  h  i |

determinant

det = a(ei - fh) - b(di - fg) + c(dh - eg)

'''
import numpy as np
```

```

# Define your matrix as a 2D NumPy array
# matrix = np.array([[2, 3, 1],
#                     [4, 1, 7],
#                     [0, 5, 2]])

matrix = np.array([[6, 24, 1],
                   [13, 16, 10],
                   [20, 17, 15]])


def determinant_matrix(matrix):
    # Calculate the determinant
    # Calculate the determinant using the formula
    det = ((matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1])) -
           (matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0])) +
           (matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0])))

    return det


def inverse_matrix(matrix):
    # Use the np.linalg.inv() function to find the inverse
    inverse_matrix = np.linalg.inv(matrix)
    return inverse_matrix


# Print the determinant
print("Determinant of a matrix:\n", determinant_matrix(matrix))
print("Inverse of a matrix:\n", inverse_matrix(matrix))

a = np. dot(determinant_matrix(matrix), matrix)

inv = inverse_matrix(a)
inv = (inv * determinant_matrix(matrix)) % 26
print(inv)
```

Program 11

Program 11: WAP to find inverse of a 2 by 2 matrix.

```
'''
# 2 by 2 matrix

# invers is given
1 / (ad - bc) * |  d  -b |
                | -c   a |


'''

import numpy as np

# Define your 2x2 matrix as nested lists
```

```python
matrix = [[5, 10],
          [9, 7]]

a = matrix[0][0]
b = matrix[0][1]
c = matrix[1][0]
d = matrix[1][1]

# Calculate the determinant

det = a * d - b * c


print("Matrix is:\n")
for row in matrix:
    print(row)

print("Determinant is: ", det)


# Check if the determinant is zero (matrix is singular)
if det == 0:
    print("The matrix is singular, and its inverse does not exist.")
else:
    # Calculate the inverse
    inverse = [[d / det, -b / det],
               [-c / det, a / det]]

    # Print the inverse matrix
    print("Inverse Matrix:")
    for row in inverse:
        print(row)
```

Program 12

Program 12: WAP to find inverse of a 3 by 3 matrix.

```python
'''
For a 3 by 3 matrix:
| a  b  c |
| d  e  f |
| g  h  i |


The inverse is given by:
1 / ((a * e * i) + (b * f * g) + (c * d * h) - (a * f * h) - (b * d * i) - (c * e * g)) * | (e * i - f * h)  (c * h - b * i)  (b * f - c * e) |
                                                                                          | (f * g - d * i)  (a * i - c * g)  (c * d - a * f) |
                                                                                          | (d * h - e * g)  (b * g - a * h)  (a * e - b * d) |


'''

import numpy as np
```

```python

# Set the global configuration for float display
np.set_printoptions(precision=4)

# matrix = np.array([[2, 3, 1],
#                    [4, 1, 7],
#                    [0, 5, 2]])

matrix = np.array([[17, 17, 5],
                   [21, 18, 21],
                   [2, 2, 19]])

a = matrix[0][0]
b = matrix[0][1]
c = matrix[0][2]

d = matrix[1][0]
e = matrix[1][1]
f = matrix[1][2]

g = matrix[2][0]
h = matrix[2][1]
i = matrix[2][2]


# Calculate the determinant
det = (a * e * i + b * f * g + c * d * h) - (a * f * h + b * d * i + c
    * e * g)

print("Matrix is:\n")
for row in matrix:
    print(row)

print("Determinant: ", det)

# Check if the determinant is zero (matrix is singular)
if det == 0:
    print("The matrix is singular, and its inverse does not exist.")
else:
    # Calculate the elements of the inverse matrix
    inv_a = (e * i - f * h) / det
    inv_b = (c * h - b * i) / det
    inv_c = (b * f - c * e) / det
    inv_d = (f * g - d * i) / det
    inv_e = (a * i - c * g) / det
    inv_f = (c * d - a * f) / det
    inv_g = (d * h - e * g) / det
    inv_h = (b * g - a * h) / det
    inv_i = (a * e - b * d) / det

    # Create the inverse matrix
    inverse = [[inv_a, inv_b, inv_c],
               [inv_d, inv_e, inv_f],
               [inv_g, inv_h, inv_i]]


inverse_mod_26 = (inverse) % 26

```

```
    # Print the inverse matrix
    # print("Inverse Matrix:")
    # for row in inverse:
    #     print(row)

    for row in inverse_mod_26:
        print(row)
```

Program 13

Program 13: WAP to find inverse of a 3 by 3 matrix.

```
'''
For a 3 by 3 matrix:
| a  b  c |
| d  e  f |
| g  h  i |


The inverse is given by:
1 / ((a * e * i) + (b * f * g) + (c * d * h) - (a * f * h) - (b * d * i
    ) - (c * e * g)) * | (e * i - f * h)  (c * h - b * i)  (b * f - c *
    e) |
                     | (f * g - d * i)  (a * i - c * g)  (c * d - a * f) |
                     | (d * h - e * g)  (b * g - a * h)  (a * e - b * d) |


'''

import numpy as np

# Set the global configuration for float display
np.set_printoptions(precision=4)

# matrix = np.array([[2, 3, 1],
#                     [4, 1, 7],
#                     [0, 5, 2]])

matrix = np.array([[17, 17, 5],
                   [21, 18, 21],
                   [2, 2, 19]])

a = matrix[0][0]
b = matrix[0][1]
c = matrix[0][2]

d = matrix[1][0]
e = matrix[1][1]
f = matrix[1][2]

g = matrix[2][0]
h = matrix[2][1]
i = matrix[2][2]


```

```python
# Calculate the determinant
det = (a * e * i + b * f * g + c * d * h) - (a * f * h + b * d * i + c
    * e * g)

print("Matrix is:\n")
for row in matrix:
    print(row)

print("Determinant: ", det)

# Check if the determinant is zero (matrix is singular)
if det == 0:
    print("The matrix is singular, and its inverse does not exist.")
else:
    # Calculate the elements of the inverse matrix
    inv_a = (e * i - f * h) / det
    inv_b = (c * h - b * i) / det
    inv_c = (b * f - c * e) / det
    inv_d = (f * g - d * i) / det
    inv_e = (a * i - c * g) / det
    inv_f = (c * d - a * f) / det
    inv_g = (d * h - e * g) / det
    inv_h = (b * g - a * h) / det
    inv_i = (a * e - b * d) / det

    # Create the inverse matrix
    inverse = [[inv_a, inv_b, inv_c],
               [inv_d, inv_e, inv_f],
               [inv_g, inv_h, inv_i]]


inverse_mod_26 = (inverse) % 26

    # Print the inverse matrix
    # print("Inverse Matrix:")
    # for row in inverse:
    #     print(row)

    for row in inverse_mod_26:
        print(row)
```

Program 14

Program 14: WAP to peform matrix multiplication using library function.

```python
import numpy as np
# Set the desired number of decimal places globally for matrix value
    display
np.set_printoptions(precision=2)

# Define two matrices as NumPy arrays
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 6]])

matrix2 = np.array([[7, 8],
                    [9, 10],
                    [11, 12]])
```

```

# Perform matrix multiplication
result = np.dot(matrix1, matrix2)

# Print the result
print("Result of Matrix Multiplication:")
print(result)
```

Program 15

Program 15: WAP to peform matrix multiplication using library function.

```
import numpy as np

# Set the desired number of decimal places globally for matrix value
    display
np.set_printoptions(precision=2)

# Define two matrices as 2D lists
matrix1 = [[1, 2, 3],
           [4, 5, 6]]

matrix2 = [[7, 8],
           [9, 10],
           [11, 12]]

# Determine the dimensions of the matrices
rows1 = len(matrix1)
cols1 = len(matrix1[0])
rows2 = len(matrix2)
cols2 = len(matrix2[0])

# Check if matrix multiplication is possible
if cols1 != rows2:
    print("Matrix multiplication is not possible. Number of columns in
    matrix1 must be equal to the number of rows in matrix2.")
else:
    # Create an empty result matrix
    result = [[0 for _ in range(cols2)] for _ in range(rows1)]

    # Perform matrix multiplication using nested loops
    for i in range(rows1):
        for j in range(cols2):
            for k in range(cols1):
                result[i][j] += matrix1[i][k] * matrix2[k][j]

    # Print the result
    print("Result of Matrix Multiplication:")
    for row in result:
        print(row)
```

Program 16

Program 16: WAP to peform matrix multiplication using library function.

```
import numpy as np

```

```python
# Define a function to convert a string to a list of numbers


def text_to_numbers(text):
    text = text.lower().replace(" ", "")
    text_to_num = []
    for c in text:
        loc = alpha.find(c)
#          print(loc)
        text_to_num.append(loc)
    return text_to_num

# Define a function to convert a list of numbers to a string


def numbers_to_text(numbers):
    return ''.join([chr(num + ord('a')) for num in numbers])

# Define a function for Hill Cipher encryption


def hill_cipher_encrypt(plaintext, key_matrix):
    plaintext = plaintext.lower().replace(" ", "")
    while len(plaintext) % 3 != 0:
        plaintext += 'X'  # Padding with 'X' if necessary

    plaintext_numbers = text_to_numbers(plaintext)
    print("Plain text in numbers: ", plaintext_numbers)
    encrypted_numbers = []

    for i in range(0, len(plaintext_numbers), 3):
        block = np.array(plaintext_numbers[i:i+3])
        encrypted_block = np.dot(block, key_matrix) % 26
        print(encrypted_block)
        encrypted_numbers.extend(encrypted_block)
    print(encrypted_numbers)

    return numbers_to_text(encrypted_numbers)


# Define the Hill Cipher key matrix (3x3 matrix)
key_matrix = np.array([[17, 17, 5],
                       [21, 18, 21],
                       [2, 2, 19]])

plaintext = "paymoremoney"
alpha = 'abcdefghijklmnopqrstuvwxyz'

text_to_num = text_to_numbers(plaintext)
print("Indexing of letters in message:\n")
print(text_to_num)

encrypted_text = hill_cipher_encrypt(plaintext, key_matrix)
print("Plain text: ", plaintext)
print("Encrypted:", encrypted_text.upper())
```

LAB-3

FAMILIARIZATION POLYALPHABETIC CIPHER

Objectives:

1. To perform encryption and decryption using Vigenere cipher and One-Time-Pad method
2. To encrypt and decrypt using Vernam method.

Program 17

Program 17: WAP to implement Vigenere cipher.

```
def vigenere_encrypt(plaintext, keyword):
    plaintext = plaintext.upper()
    keyword = keyword.upper()
    encrypted_text = []
    key_length = len(keyword)

    for i in range(len(plaintext)):
        if plaintext[i].isalpha():
            shift = ord(keyword[i % key_length]) - ord('A')
            encrypted_char = chr(
                ((ord(plaintext[i]) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text.append(encrypted_char)
        else:
            encrypted_text.append(plaintext[i])

    return ''.join(encrypted_text)


def vigenere_decrypt(ciphertext, keyword):
    ciphertext = ciphertext.upper()
    keyword = keyword.upper()
    decrypted_text = []
    key_length = len(keyword)

    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():
            shift = ord(keyword[i % key_length]) - \
                ord('A')  # ord('A') return value 65
            decrypted_char = chr(
                ((ord(ciphertext[i]) - ord('A') - shift + 26) % 26) + ord('A'))
            decrypted_text.append(decrypted_char)
        else:
            decrypted_text.append(ciphertext[i])

    return ''.join(decrypted_text)


# Example usage
plaintext = input("Input a message: ")
keyword = "KEY"  # you can change the key
encrypted_text = vigenere_encrypt(plaintext, keyword)
decrypted_text = vigenere_decrypt(encrypted_text, keyword)
```

```python

print("Plaintext:", plaintext)
print("Encrypted text:", encrypted_text)
print("Decrypted text:", decrypted_text)
```

Program 18

Program 18: WAP to implement One Time Pad cipher.

```python
import random


def generate_one_time_pad(length):
    """Generate a random one-time pad of the specified length."""
    return [random.randint(0, 25) for _ in range(length)]


def otp_encrypt(text, one_time_pad):
    """Encrypt plaintext using the one-time pad."""
    if len(text) != len(one_time_pad):
        raise ValueError(
            "Plaintext and one-time pad must be of the same length")

    encrypted_text = []
    for i in range(len(text)):
        char = text[i].upper()
        if char.isalpha():
            shift = one_time_pad[i]
            encrypted_char = chr(
                ((ord(char) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text.append(encrypted_char)
        else:
            encrypted_text.append(char)

    return ''.join(encrypted_text)


def otp_decrypt(ciphertext, one_time_pad):
    """Decrypt ciphertext using the one-time pad."""
    if len(ciphertext) != len(one_time_pad):
        raise ValueError(
            "Ciphertext and one-time pad must be of the same length")

    decrypted_text = []
    for i in range(len(ciphertext)):
        char = ciphertext[i].upper()
        if char.isalpha():
            shift = one_time_pad[i]
            decrypted_char = chr(
                ((ord(char) - ord('A') - shift + 26) % 26) + ord('A'))
            decrypted_text.append(decrypted_char)
        else:
            decrypted_text.append(char)

    return ''.join(decrypted_text)


```

```python
# Example usage
plaintext = input("Input a message: ")
one_time_pad = generate_one_time_pad(len(plaintext))
encrypted_text = otp_encrypt(plaintext, one_time_pad)
decrypted_text = otp_decrypt(encrypted_text, one_time_pad)


print("Plaintext:", plaintext)
print("Encrypted text:", encrypted_text)
print("Decrypted text:", decrypted_text)
```

Program 19

Program 18: WAP to implement Vernam cipher.

```python
import random


def generate_random_key(message_length):
    # Generate a random key of 0s and 1s of the same length as the
    message
    return [random.randint(0, 1) for _ in range(message_length)]


def encrypt(plaintext, key):
    # Ensure the key and plaintext have the same length
    if len(plaintext) != len(key):
        raise ValueError("Key length must match the plaintext length")

    # Perform bitwise XOR to encrypt the plaintext
    ciphertext = [str(int(plaintext[i]) ^ key[i])
                  for i in range(len(plaintext))]

    # Convert the list of bits back to a string
    return ''.join(ciphertext)


def decrypt(ciphertext, key):
    # Ensure the key and ciphertext have the same length
    if len(ciphertext) != len(key):
        raise ValueError("Key length must match the ciphertext length")

    # Perform bitwise XOR to decrypt the ciphertext
    plaintext = [str(int(ciphertext[i]) ^ key[i])
                 for i in range(len(ciphertext))]

    # Convert the list of bits back to a string
    return ''.join(plaintext)


# Example usage
message = "Hello World!"
message_bits = [int(bit) for bit in ''.join(format(ord(char), '08b')
                                             for char in message)]   #
    Convert message to bits
key = generate_random_key(len(message_bits))  # Generate a random key

```

```python
key_stream = ''.join(str(bit) for bit in key)

ciphertext = encrypt(message_bits, key)
decrypted_message_bits = decrypt(ciphertext, key)

# Convert the decrypted bits back to a string
decrypted_message = ''.join(chr(int(
    decrypted_message_bits[i:i+8], 2)) for i in range(0, len(decrypted_message_bits), 8))

print("Original Message:", message)
print("length of message: ", len(message))
print("Key: ",  key_stream)
print("Cipher text (in bits):", ''.join(map(str, ciphertext)))
print("Length of key: ", len(key))
print("Length of encrypted message: ", len(ciphertext))
print("Decrypted text:", decrypted_message)

'''
Descrption: ciphertext = [str(int(plaintext[i]) ^ key[i]) for i in range(len(plaintext))]


This line is responsible for generating the ciphertext, which is the result of encrypting the plaintext using the Vernam cipher. It uses a list comprehension to process each bit of the plaintext and corresponding bit of the key and then combines them to form the ciphertext.

Here's a step-by-step explanation:

1. `for i in range(len(plaintext))`: This part sets up a loop that iterates over the indices
(positions) of each bit in the `plaintext`. `len(plaintext)` returns the length of the plaintext,
and `range(len(plaintext))` creates a sequence of indices from 0 to the length of
the plaintext minus one.

2. `plaintext[i]`: Inside the loop, `plaintext[i]` accesses the `i`-th bit of the plaintext.
This bit is either 0 or 1, representing the binary value of the character in the plaintext.

3. `key[i]`: Similarly, `key[i]` accesses the `i`-th bit of the key.
The key is generated randomly and contains 0s and 1s, just like the plaintext.

4. `int(plaintext[i]) ^ key[i]`: This part performs a bitwise XOR (exclusive OR) operation between the `i`-th bit of the plaintext
and the `i`-th bit of the key. The `int(...)` conversion is used to ensure that
both `plaintext[i]` and `key[i]` are treated as integers. The XOR operation returns 1 if the
bits being compared are different and 0 if they are the same.

5. `str(...)`: After performing the XOR operation,
the result is converted back to a string using `str(...)`.
```

```
This is done because the ciphertext is typically represented as a
    sequence of characters.

So, in summary, this line processes each bit of the plaintext and key,
XORs them together, and converts the result to a string,
effectively creating the ciphertext as a sequence of 0s and 1s.

'''
```