**Practical No.2**

* **Substitution Technique:-**

1. Ceaser Cipher –

Code:

def caesar\_encrypt(plaintext, k):

    ciphertext = ""

    for char in plaintext:

        if char.isalpha():

            base = ord('A') if char.isupper() else ord('a')

            shifted = (ord(char) - base + k) % 26 + base

            ciphertext += chr(shifted)

        else:

            ciphertext += char

    return ciphertext

if \_\_name\_\_ == "\_\_main\_\_":

    text = "THIS IS  SECURITY LAB"

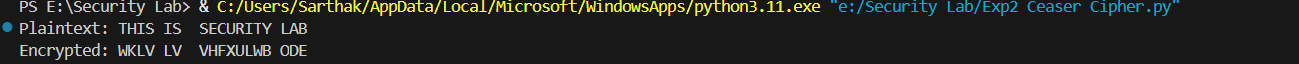
    k = 3

    encrypted = caesar\_encrypt(text, k)

    print("Plaintext:", text)

    print("Encrypted:", encrypted)

Output:



1. Modified Ceaser Cipher –

Code:

def modified\_caesar\_encrypt(plaintext, k):

    modifiedciphertext = ""

    for char in plaintext:

        if char.isalpha():

            base = ord('A') if char.isupper() else ord('a')

            shifted = (ord(char) - base + k) % 26 + base

            modifiedciphertext += chr(shifted)

        else:

            modifiedciphertext += char

    return modifiedciphertext

if \_\_name\_\_ == "\_\_main\_\_":

    text = "THIS IS SECURITY LAB"

    k = 4

    encrypted = modified\_caesar\_encrypt(text, k)

    print("Plaintext:", text)

    print("Encrypted:", encrypted)

Output:



1. Monoalphabetic Cipher –

Code:

def generate\_substitution\_dict(key):

    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    key = key.upper()

    encrypt\_dict = {}

    decrypt\_dict = {}

    for a, k in zip(alphabet, key):

        encrypt\_dict[a] = k

        decrypt\_dict[k] = a

    return encrypt\_dict, decrypt\_dict

def monoalphabetic\_encrypt(plaintext, key):

    encrypt\_dict, \_ = generate\_substitution\_dict(key)

    ciphertext = []

    for char in plaintext:

        if char.isalpha():

            is\_upper = char.isupper()

            sub\_char = encrypt\_dict[char.upper()]

            ciphertext.append(sub\_char if is\_upper else sub\_char.lower())

        else:

            ciphertext.append(char)

    return ''.join(ciphertext)

if \_\_name\_\_ == "\_\_main\_\_":

    plaintext = "THIS IS SECURITY LAB"

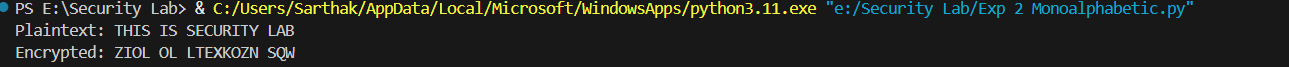
    key = "QWERTYUIOPASDFGHJKLZXCVBNM"

    encrypted = monoalphabetic\_encrypt(plaintext, key)

    print("Plaintext:", plaintext)

    print("Encrypted:", encrypted)

Output:



4.Homophonic Cipher –

Code:

def generate\_substitution\_dict(key):

    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    key = key.upper()

    encrypt\_dict = {}

    decrypt\_dict = {}

    for a, k in zip(alphabet, key):

        encrypt\_dict[a] = k

        decrypt\_dict[k] = a

    return encrypt\_dict, decrypt\_dict

def monoalphabetic\_encrypt(plaintext, key):

    encrypt\_dict, \_ = generate\_substitution\_dict(key)

    ciphertext = []

    for char in plaintext:

        if char.isalpha():

            is\_upper = char.isupper()

            sub\_char = encrypt\_dict[char.upper()]

            ciphertext.append(sub\_char if is\_upper else sub\_char.lower())

        else:

            ciphertext.append(char)

    return ''.join(ciphertext)

if \_\_name\_\_ == "\_\_main\_\_":

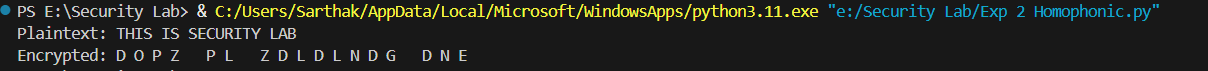
    plaintext = "THIS IS SECURITY LAB"

    key = "QWERTYUIOPASDFGHJKLZXCVBNM"

    encrypted = monoalphabetic\_encrypt(plaintext, key)

    print("Plaintext:", plaintext)

    print("Encrypted:", encrypted)

Output: 

* **Transposition Technique:-**

1. Simple Columnar Transposition Technique:

Code:  
def columnar\_encrypt(plaintext, keyword):

    plaintext = ''.join(filter(str.isalpha, plaintext.upper()))

    keyword = keyword.upper()

    n\_cols = len(keyword)

    n\_rows = len(plaintext) // n\_cols + (1 if len(plaintext) % n\_cols else 0)

    padded\_len = n\_rows \* n\_cols

    plaintext += 'X' \* (padded\_len - len(plaintext))

    matrix = [list(plaintext[i \* n\_cols:(i + 1) \* n\_cols]) for i in range(n\_rows)]

    order = sorted(range(len(keyword)), key=lambda k: keyword[k])

    ciphertext = ''

    for col in order:

        for row in range(n\_rows):

            ciphertext += matrix[row][col]

    return ciphertext

def columnar\_decrypt(ciphertext, keyword):

    ciphertext = ciphertext.upper()

    keyword = keyword.upper()

    n\_cols = len(keyword)

    n\_rows = len(ciphertext) // n\_cols

    order = sorted(range(len(keyword)), key=lambda k: keyword[k])

    matrix = [ [''] \* n\_rows for \_ in range(n\_cols) ]

    index = 0

    for col in order:

        for row in range(n\_rows):

            matrix[col][row] = ciphertext[index]

            index += 1

    plaintext = ''

    for row in range(n\_rows):

        for col in range(n\_cols):

            plaintext += matrix[col][row]

    return plaintext.rstrip('X')

if \_\_name\_\_ == "\_\_main\_\_":

    plaintext = "WELCOME TO LAB"

    keyword = "NET"

    encrypted = columnar\_encrypt(plaintext, keyword)

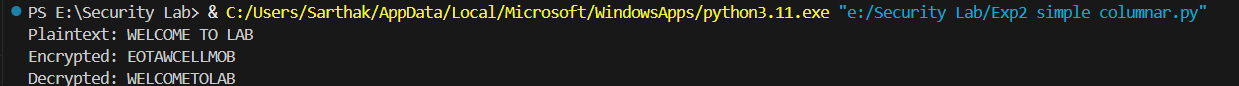
    decrypted = columnar\_decrypt(encrypted, keyword)

    print("Plaintext:", plaintext)

    print("Encrypted:", encrypted)

    print("Decrypted:", decrypted)

Output:



1. Simple Columnar Transposition Technique with multiple rounds:

Code:

def columnar\_encrypt(plaintext, keyword):

    plaintext = ''.join(filter(str.isalpha, plaintext.upper()))

    keyword = keyword.upper()

    n\_cols = len(keyword)

    n\_rows = len(plaintext) // n\_cols + (1 if len(plaintext) % n\_cols else 0)

    padded\_len = n\_rows \* n\_cols

    plaintext += 'X' \* (padded\_len - len(plaintext))

    matrix = [list(plaintext[i \* n\_cols:(i + 1) \* n\_cols]) for i in range(n\_rows)]

    order = sorted(range(len(keyword)), key=lambda k: keyword[k])

    ciphertext = ''

    for col in order:

        for row in range(n\_rows):

            ciphertext += matrix[row][col]

    return ciphertext

def columnar\_decrypt(ciphertext, keyword):

    ciphertext = ciphertext.upper()

    keyword = keyword.upper()

    n\_cols = len(keyword)

    n\_rows = len(ciphertext) // n\_cols

    order = sorted(range(len(keyword)), key=lambda k: keyword[k])

    matrix = [ [''] \* n\_rows for \_ in range(n\_cols) ]

    index = 0

    for col in order:

        for row in range(n\_rows):

            matrix[col][row] = ciphertext[index]

            index += 1

    plaintext = ''

    for row in range(n\_rows):

        for col in range(n\_cols):

            plaintext += matrix[col][row]

    return plaintext.rstrip('X')

if \_\_name\_\_ == "\_\_main\_\_":

    plaintext = "THIS IS SL LAB"

    keyword = "NET"

    encrypted = columnar\_encrypt(plaintext, keyword)

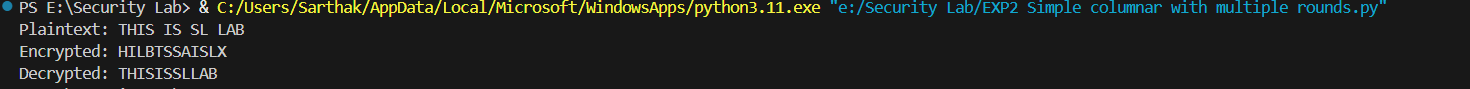
    decrypted = columnar\_decrypt(encrypted, keyword)

    print("Plaintext:", plaintext)

    print("Encrypted:", encrypted)

    print("Decrypted:", decrypted)

Output:



1. Rail Fence Cipher

Code:

def rail\_fence\_encrypt(plain\_text, rails):

    if rails < 2:

        raise ValueError("Number of rails must be at least 2.")

    fence = [[] for \_ in range(rails)]

    rail = 0

    direction = 1  # 1 for down, -1 for up

    for char in plain\_text:

        fence[rail].append(char)

        rail += direction

        if rail == rails - 1 or rail == 0:

            direction = -direction  # Reverse direction at top or bottom rail

    # Combine all rails to form ciphertext

    return ''.join(''.join(row) for row in fence)

def rail\_fence\_decrypt(cipher\_text, rails):

    if rails < 2:

        raise ValueError("Number of rails must be at least 2.")

    n = len(cipher\_text)

    fence = [[''] \* n for \_ in range(rails)]

    # Mark positions where characters should be placed

    rail = 0

    direction = 1

    for i in range(n):

        fence[rail][i] = '\*'  # Mark the position

        rail += direction

        if rail == 0 or rail == rails - 1:

            direction = -direction

    # Fill the fence with ciphertext characters row-wise

    index = 0

    for i in range(rails):

        for j in range(n):

            if fence[i][j] == '\*' and index < n:

                fence[i][j] = cipher\_text[index]

                index += 1

    # Read the fence in zigzag order to reconstruct plaintext

    result = []

    rail = 0

    direction = 1

    for i in range(n):

        result.append(fence[rail][i])

        rail += direction

        if rail == 0 or rail == rails - 1:

            direction = -direction

    return ''.join(result)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    plaintext = "WELCOME TO LAB"

    rails = 3

    encrypted = rail\_fence\_encrypt(plaintext, rails)

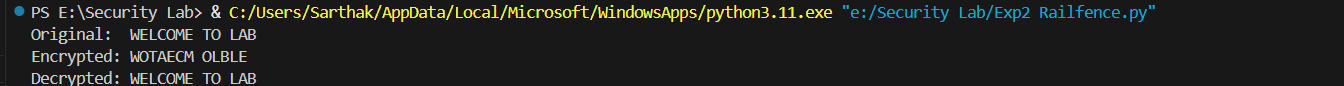
    decrypted = rail\_fence\_decrypt(encrypted, rails)

    print(f"Original:  {plaintext}")

    print(f"Encrypted: {encrypted}")

    print(f"Decrypted: {decrypted}")

Output:



1. Vernam Cipher

Code:

def vernam\_encrypt(plain\_text, key):

    if len(key) < len(plain\_text):

        raise ValueError("Key must be at least as long as plaintext.")

    # Convert plaintext and key to byte arrays

    plain\_bytes = bytearray(plain\_text, 'utf-8')

    key\_bytes = bytearray(key, 'utf-8')[:len(plain\_text)]  # Use only needed key part

    # Perform XOR byte by byte

    cipher\_bytes = bytearray()

    for i in range(len(plain\_bytes)):

        cipher\_bytes.append(plain\_bytes[i] ^ key\_bytes[i])

    return cipher\_bytes.hex()  # Return as hex string

def vernam\_decrypt(cipher\_text\_hex, key):

    # Convert hex string back to bytes

    cipher\_bytes = bytearray.fromhex(cipher\_text\_hex)

    # Convert key to bytes and truncate to ciphertext length

    key\_bytes = bytearray(key, 'utf-8')[:len(cipher\_bytes)]

    # Perform XOR to decrypt

    plain\_bytes = bytearray()

    for i in range(len(cipher\_bytes)):

        plain\_bytes.append(cipher\_bytes[i] ^ key\_bytes[i])

    return plain\_bytes.decode('utf-8')

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    plaintext = "HELLO WORLD"

    key = "MYSECRETKEY"

    encrypted = vernam\_encrypt(plaintext, key)

    decrypted = vernam\_decrypt(encrypted, key)

    print(f"Original:  {plaintext}")

    print(f"Key:       {key}")

    print(f"Encrypted: {encrypted}")

    print(f"Decrypted: {decrypted}")

Output:

