**Lab7**

1. **Write a Python script to encrypt columnar transposition using keyword.**

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

# Remove spaces and convert to uppercase

plaintext = plaintext.replace(" ", "").upper()

# Calculate the number of columns

key\_length = len(keyword)

num\_columns = (len(plaintext) + key\_length - 1) // key\_length # Ceiling division

# Create a grid to hold the characters

grid = [''] \* key\_length

for i in range(len(plaintext)):

grid[i % key\_length] += plaintext[i]

# Sort the keyword and determine the order of columns

sorted\_key = sorted((char, index) for index, char in enumerate(keyword))

order = [index for char, index in sorted\_key]

# Read the columns in the order specified by the sorted key

ciphertext = ''.join(grid[i] for i in order)

return ciphertext

# Input parameters

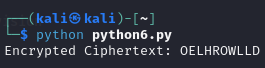
plaintext = "HELLO WORLD"

keyword = "KEYWORD"

# Encrypt the plaintext

ciphertext = columnar\_transposition\_encrypt(plaintext, keyword)

print("Encrypted Ciphertext:", ciphertext)



1. **Write a Python script to encrypt double columnar transposition.**

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

# Remove spaces and convert to uppercase

plaintext = plaintext.replace(" ", "").upper()

# Create the grid

key\_length = len(keyword)

num\_columns = (len(plaintext) + key\_length - 1) // key\_length # Ceiling division

# Fill the grid

grid = [''] \* key\_length

for i in range(len(plaintext)):

grid[i % key\_length] += plaintext[i]

# Sort the keyword and get the order of columns

sorted\_key = sorted((char, index) for index, char in enumerate(keyword))

order = [index for char, index in sorted\_key]

# Read the columns in the order specified by the sorted key

ciphertext = ''.join(grid[i] for i in order)

return ciphertext

def double\_columnar\_transposition\_encrypt(plaintext, keyword1, keyword2):

# First round of encryption

first\_round = columnar\_transposition\_encrypt(plaintext, keyword1)

# Second round of encryption

second\_round = columnar\_transposition\_encrypt(first\_round, keyword2)

return second\_round

# Input parameters

plaintext = "HELLO WORLD"

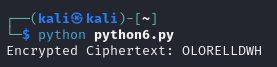
keyword1 = "KEYWORD"

keyword2 = "ANOTHER"

# Encrypt the plaintext using double columnar transposition

ciphertext = double\_columnar\_transposition\_encrypt(plaintext, keyword1, keyword2)

print("Encrypted Ciphertext:", ciphertext)



1. **Write a Python script to encrypt the message “She is listening” using the 6-character keyword “PASCAL” with Vigenere cipher.**

def vigenere\_encrypt(plaintext, keyword):

# Remove spaces and convert to uppercase

plaintext = plaintext.replace(" ", "").upper()

keyword = keyword.upper()

ciphertext = []

keyword\_repeated = (keyword \* (len(plaintext) // len(keyword) + 1))[:len(plaintext)]

for p\_char, k\_char in zip(plaintext, keyword\_repeated):

# Encrypt only if the character is a letter

if p\_char.isalpha():

# Calculate the shifted character

p\_index = ord(p\_char) - ord('A')

k\_index = ord(k\_char) - ord('A')

c\_index = (p\_index + k\_index) % 26

ciphertext.append(chr(c\_index + ord('A')))

else:

# If it's not a letter, just append it

ciphertext.append(p\_char)

return ''.join(ciphertext)

# Input parameters

plaintext = "She is listening"

keyword = "PASCAL"

# Encrypt the plaintext

ciphertext = vigenere\_encrypt(plaintext, keyword)

print("Encrypted Ciphertext:", ciphertext)



1. **Write a Python script to encrypt and decrypt Hill cipher**

import numpy as np

def mod26(x):

"""Perform modulo 26 operation to wrap around alphabet."""

return x % 26

def generate\_key\_matrix(key):

"""Convert key string into a matrix."""

key\_length = int(len(key) \*\* 0.5)

key\_matrix = np.zeros((key\_length, key\_length), dtype=int)

for i in range(key\_length):

for j in range(key\_length):

key\_matrix[i][j] = ord(key[i \* key\_length + j]) - ord('A')

return key\_matrix

def encrypt(plaintext, key):

"""Encrypt the plaintext using the Hill cipher."""

key\_matrix = generate\_key\_matrix(key)

key\_length = key\_matrix.shape[0]

# Prepare plaintext

plaintext = plaintext.replace(" ", "").upper()

while len(plaintext) % key\_length != 0:

plaintext += 'X' # Padding with 'X'

plaintext\_matrix = np.array([ord(char) - ord('A') for char in plaintext]).reshape(-1, key\_length)

# Encrypt

ciphertext\_matrix = (plaintext\_matrix @ key\_matrix) % 26

ciphertext = ''.join(chr(num + ord('A')) for num in ciphertext\_matrix.flatten())

return ciphertext

def decrypt(ciphertext, key):

"""Decrypt the ciphertext using the Hill cipher."""

key\_matrix = generate\_key\_matrix(key)

key\_length = key\_matrix.shape[0]

# Calculate the inverse of the key matrix

det = int(round(np.linalg.det(key\_matrix))) # Determinant

det\_inv = pow(det, -1, 26) # Modular multiplicative inverse

# Calculate the adjugate of the key matrix

key\_matrix\_mod = np.linalg.inv(key\_matrix) \* det

adjugate\_matrix = np.round(key\_matrix\_mod).astype(int) % 26

# Inverse key matrix

inverse\_key\_matrix = (det\_inv \* adjugate\_matrix) % 26

# Prepare ciphertext

ciphertext = ciphertext.replace(" ", "").upper()

ciphertext\_matrix = np.array([ord(char) - ord('A') for char in ciphertext]).reshape(-1, key\_length)

# Decrypt

decrypted\_matrix = (ciphertext\_matrix @ inverse\_key\_matrix) % 26

plaintext = ''.join(chr(num + ord('A')) for num in decrypted\_matrix.flatten())

return plaintext

# Input parameters

plaintext = "HELLO"

key = "GYBNQKURP"

# Encrypt the plaintext

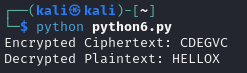
ciphertext = encrypt(plaintext, key)

print("Encrypted Ciphertext:", ciphertext)

# Decrypt the ciphertext

decrypted\_plaintext = decrypt(ciphertext, key)

print("Decrypted Plaintext:", decrypted\_plaintext)



Bonus Point:

**5. Write a Python script to perform Kasiski test.**

from collections import defaultdict

import re

def kasiski\_test(ciphertext, min\_length=3):

"""Perform Kasiski examination on the given ciphertext."""

# Find repeated sequences

sequences = defaultdict(list)

# Extract sequences of the specified minimum length

for length in range(min\_length, min\_length + 1):

for i in range(len(ciphertext) - length):

sequence = ciphertext[i:i + length]

next\_occurrence = ciphertext.find(sequence, i + length)

if next\_occurrence != -1:

distance = next\_occurrence - i

sequences[sequence].append(distance)

# Filter out sequences that don't have multiple occurrences

filtered\_sequences = {seq: dists for seq, dists in sequences.items() if len(dists) > 1}

return filtered\_sequences

# Input ciphertext

ciphertext = "LXFOPVEFRNHR"

# Perform Kasiski test

repeated\_sequences = kasiski\_test(ciphertext)

# Display results

if repeated\_sequences:

print("Repeated Sequences and Distances:")

for seq, distances in repeated\_sequences.items():

print(f"Sequence: {seq}, Distances: {distances}")

else:

print("No repeated sequences found.")



**======================================================================================================================================**

**"Obstacles can't stop you. Problems can't stop you. Most of all, other people can't stop you. Only you can stop you." —Jeffrey Gitomer**

**======================================================================================================================================**