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()

# Determine the length of the keyword and calculate number of columns

num\_cols = len(keyword)

# Calculate the number of rows needed

num\_rows = len(plaintext) // num\_cols + (len(plaintext) % num\_cols > 0)

# Create a grid to hold the characters

grid = [''] \* num\_cols

# Fill the grid with characters from plaintext

for i in range(len(plaintext)):

col = i % num\_cols

grid[col] += plaintext[i]

# Create a list of (key\_char, index) pairs and sort by key\_char

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

# Generate the ciphertext by reading columns in the order of sorted key

ciphertext = ''

for index in key\_indices:

ciphertext += grid[index]

return ciphertext

keyword = "KEYWORD"

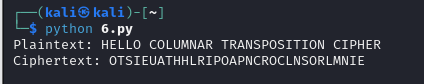
plaintext = "HELLO COLUMNAR TRANSPOSITION CIPHER"

# Encrypt the plaintext using Columnar Transposition with the keyword

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

print("Plaintext:", plaintext)

print("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()

# Determine the number of columns based on the keyword length

num\_cols = len(keyword)

# Calculate the number of rows needed

num\_rows = len(plaintext) // num\_cols + (len(plaintext) % num\_cols > 0)

# Create a grid to hold the characters

grid = [''] \* num\_cols

# Fill the grid with characters from plaintext

for i in range(len(plaintext)):

col = i % num\_cols

grid[col] += plaintext[i]

# Create a list of (key\_char, index) pairs and sort by key\_char

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

# Generate the ciphertext by reading columns in the order of sorted key

ciphertext = ''

for index in key\_indices:

ciphertext += grid[index]

return ciphertext

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

# First encryption using the first keyword

intermediate\_ciphertext = columnar\_transposition\_encrypt(plaintext, keyword1)

# Second encryption using the second keyword

final\_ciphertext = columnar\_transposition\_encrypt(intermediate\_ciphertext, keyword2)

return final\_ciphertext

keyword1 = "KEYWORD"

keyword2 = "ANOTHER"

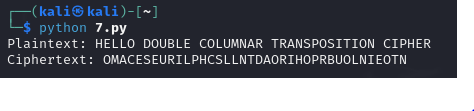
plaintext = "HELLO DOUBLE COLUMNAR TRANSPOSITION CIPHER"

# Encrypt the plaintext using Double Columnar Transposition

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

print("Plaintext:", plaintext)

print("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):

# Clean up the plaintext: remove spaces and convert to uppercase

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

keyword = keyword.upper()

ciphertext = []

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

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

if p\_char.isalpha(): # Ensure we only encrypt alphabetic characters

# Calculate the shift

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

# Encrypt the character

encrypted\_char = chr((ord(p\_char) - ord('A') + shift) % 26 + ord('A'))

ciphertext.append(encrypted\_char)

else:

ciphertext.append(p\_char) # Non-alphabetic characters remain unchanged

return ''.join(ciphertext)

plaintext = "She is listening"

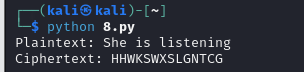
keyword = "PASCAL"

# Encrypt the plaintext using Vigenère Cipher

ciphertext = vigenere\_encrypt(plaintext, keyword)

print("Plaintext:", plaintext)

print("Ciphertext:", ciphertext)



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

import numpy as np

def create\_matrix(key):

return np.array(key).reshape(2, 2)

def mod26(x):

return x % 26

def gcd(a, b):

while b:

a, b = b, a % b

return a

def hill\_encrypt(plaintext, key):

key\_matrix = create\_matrix(key)

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

if len(plaintext) % 2 != 0:

plaintext += 'X' # Padding with 'X' if length is odd

ciphertext = ""

for i in range(0, len(plaintext), 2):

block = np.array([ord(plaintext[i]) - ord('A'), ord(plaintext[i + 1]) - ord('A')])

encrypted\_block = np.dot(key\_matrix, block)

encrypted\_block = mod26(encrypted\_block)

ciphertext += chr(encrypted\_block[0] + ord('A')) + chr(encrypted\_block[1] + ord('A'))

return ciphertext

def hill\_decrypt(ciphertext, key):

key\_matrix = create\_matrix(key)

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

if gcd(det, 26) != 1:

raise ValueError("The key matrix is not invertible under modulo 26. Choose a different key.")

det\_inv = pow(det, -1, 26)

key\_matrix\_inv = (det\_inv \* np.round(det \* np.linalg.inv(key\_matrix)).astype(int)) % 26

plaintext = ""

for i in range(0, len(ciphertext), 2):

block = np.array([ord(ciphertext[i]) - ord('A'), ord(ciphertext[i + 1]) - ord('A')])

decrypted\_block = np.dot(key\_matrix\_inv, block)

decrypted\_block = mod26(decrypted\_block)

plaintext += chr(decrypted\_block[0] + ord('A')) + chr(decrypted\_block[1] + ord('A'))

return plaintext

key = [5, 7, 2, 3] # Valid 2x2 key matrix

plaintext = "HELP"

ciphertext = hill\_encrypt(plaintext, key)

try:

decrypted\_text = hill\_decrypt(ciphertext, key)

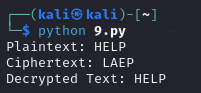
print("Plaintext:", plaintext)

print("Ciphertext:", ciphertext)

print("Decrypted Text:", decrypted\_text)

except ValueError as e:

print(e)



Bonus Point:

1. Write a Python script to perform Kasiski test.

import re

from collections import defaultdict

def find\_repeats(ciphertext, seq\_length):

"""Find repeated sequences of a given length in the ciphertext."""

repeats = defaultdict(list)

for i in range(len(ciphertext) - seq\_length + 1):

seq = ciphertext[i:i + seq\_length]

repeats[seq].append(i)

# Filter out sequences that occur only once

return {seq: positions for seq, positions in repeats.items() if len(positions) > 1}

def calculate\_distances(repeats):

"""Calculate distances between the repeated sequences."""

distances = []

for positions in repeats.values():

for i in range(len(positions) - 1):

distance = positions[i + 1] - positions[i]

distances.append(distance)

return distances

def find\_factors(distances):

"""Find factors for each distance."""

factors = defaultdict(list)

for distance in distances:

for i in range(2, distance + 1):

if distance % i == 0:

factors[i].append(distance)

return factors

def kasiski\_examination(ciphertext, seq\_length=3):

# Clean ciphertext: remove non-alphabetic characters and convert to uppercase

ciphertext = re.sub(r'[^A-Z]', '', ciphertext.upper())

# Step 1: Find repeated sequences

repeats = find\_repeats(ciphertext, seq\_length)

# Step 2: Calculate distances

distances = calculate\_distances(repeats)

# Step 3: Find factors of the distances

factors = find\_factors(distances)

return repeats, distances, factors

ciphertext = "LXFOPVEFRNHRLXFOPVEFRNHRLXFOPVEFRNHR"

repeats, distances, factors = kasiski\_examination(ciphertext)

print("Repeated Sequences:")

for seq, positions in repeats.items():

print(f"Sequence: {seq}, Positions: {positions}")

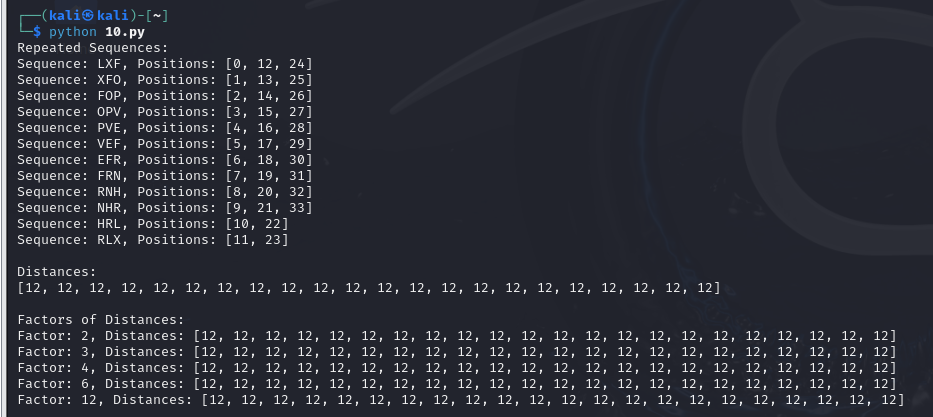
print("\nDistances:")

print(distances)

print("\nFactors of Distances:")

for factor, distances in factors.items():

print(f"Factor: {factor}, Distances: {distances}")



======================================================================================================================================

"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

======================================================================================================================================