LIST OF PROGRAMS 4

**1.Write a C program for Caesar cipher involves replacing each letter of the alphabet with the letter standing k placed further down the alphabet, for k in the range 1 through 25.**

**2. Write a C program for monoalphabetic substitution cipher maps a plaintext alphabet to a ciphertext alphabet, so that each letter of the plaintext alphabet maps to a single unique letter of the ciphertext alphabet.**

**3. Write a C program for Playfair algorithm is based on the use of a 5 X 5 matrix of letters constructed using a keyword. Plaintext is encrypted two letters at a time using this matrix.**

**4. Write a C program for polyalphabetic substitution cipher uses a separate monoalphabetic substitution cipher for each successive letter of plaintext, depending on a key.**

**5. Write a C program for generalization of the Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3. a. Are there any limitations on the value of b? b. Determine which values of a are not allowed.**

**6. Write a C program for ciphertext has been generated with an affine cipher. The most frequent letter of the ciphertext is “B,” and the second most frequent letter of the ciphertext is “U.”Break this code.**

**7. Write a C program for the following ciphertext was generated using a simple substitution algorithm. 53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\* ;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81 (‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?; Decrypt this message. 1. As you know, the most frequently occurring letter in English is e. Therefore, the first or second (or perhaps third?) most common character in the message is likely to stand for e. Also, e is often seen in pairs (e.g., meet, fleet, speed, seen, been, agree, etc.). Try to find a character in the ciphertext that decodes to e. 2. The most common word in English is “the.” Use this fact to guess the characters that stand for t and h. 3. Decipher the rest of the message by deducing additional words.**

**8. Write a C program for monoalphabetic cipher is that both sender and receiver must commit the permuted cipher sequence to memory. A common technique for avoiding this is to use a keyword from which the cipher sequence can be generated. For example, using the keyword CIPHER, write out the keyword followed by unused letters in normal order and match this against the plaintext letters: plain: a b c d e f g h i j k l m n o p q r s t u v w x y z cipher: C I P H E R A B D F G J K L M N O Q S T U V W X Y Z**

**9. Write a C program for PT-109 American patrol boat, under the command of Lieutenant John F. Kennedy, was sunk by a Japanese destroyer, a message was received at an Australian wireless station in Playfair code: KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBNT GONEY CUZWR GDSON SXBOU YWRHE BAAHY USEDQ**

**10. Write a C program for Playfair matrix: M F H I/J K U N O P Q Z V W X Y E L A R G D S T B C Encrypt this message: Must see you over Cadogan West. Coming at once.**

**11. Write a C program for possible keys does the Playfair cipher have? Ignore the fact that some keys might produce identical encryption results. Express your answer as an approximate power of 2. a. Now take into account the fact that some Playfair keys produce the same encryption results. How many effectively unique keys does the Playfair cipher have?**

**12. a. Write a C program to Encrypt the message “meet me at the usual place at ten rather than eight oclock” using the Hill cipher with the key. 9 4 5 7 a. Show your calculations and the result. b. Show the calculations for the corresponding decryption of the ciphertext to recover the original plaintext.**

**13. Write a C program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted.**

**14. Write a C program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on. a. Encrypt the plaintext send more money with the key stream 9 0 1 7 23 15 21 14 11 11 2 8 9 b. Using the ciphertext produced in part (a), find a key so that the cipher text decrypts to the plaintext cash not needed.**

**15. Write a C program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**16. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**17. Write a C program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used in reverse order. Design a key-generation scheme with the appropriate shift schedule for the decryption process.**

**18. Write a C program for DES the first 24 bits of each subkey come from the same subset of 28 bits of the initial key and that the second 24 bits of each subkey come from a disjoint subset of 28 bits of the initial key.**

**19. Write a C program for encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the definition of CBC. Which of the two would you choose: a. For security? b. For performance?**

**20. Write a C program for ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error in the transmitted C1 obviously corrupts P1 and P2. a. Are any blocks beyond P2 affected? b. Suppose that there is a bit error in the source version of P1. Through how many ciphertext blocks is this error propagated? What is the effect at the receiver?**

**21. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?**

**22. Write a C program for Encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.**

**23. Write a C program for Encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES. Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.**

**24. Write a C program for RSA system, the public key of a given user is e = 31, n = 3599. What is the private key of this user? Hint: First use trial-and-error to determine p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).**

**25. Write a C program for set of blocks encoded with the RSA algorithm and we don’t have the private key. Assume n = pq, e is the public key. Suppose also someone tells us they know one of the plaintext blocks has a common factor with n. Does this help us in any way?**

**26. Write a C program for RSA public-key encryption scheme, each user has a public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than generating a new modulus, he decides to generate a new public and a new private key. Is this safe?**

**27. Write a C program for Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 (A S 0, c, Z S 25) and then encrypting each number separately using RSA with large e and large n. Is this method secure? If not, describe the most efficient attack against this encryption method.**

**28. Write a C program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?**

**29. Write a C program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix. 30. Write a C program for CBC MAC of a oneblock message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.**

**31. Write a C program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. a. What constants are needed for block sizes of 64 and 128 bits? b. How the left shift and XOR accomplishes the desired result.**

**32. Write a C program for DSA, because the value of k is generated for each signature, even if the same message is signed twice on different occasions, the signatures will differ. This is not true of RSA signatures. Write a C program for the implication of this difference?**

**33. Writing a C program for Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. Implement in C programming.**

**34. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?**

**35. Write a C program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.**

**36. Write a C program for Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.**

**37. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**38. Write a C program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted. Implement in C programming.**

**39. Write a C program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**40. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

SOLUTIONS:

1.

def caesar\_encrypt(plaintext, k):

"""Encrypt the plaintext using the Caesar cipher."""

ciphertext = ""

for char in plaintext:

if char.isalpha():

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

ciphertext += chr((ord(char) - base + k) % 26 + base)

else:

ciphertext += char # Non-alphabet characters remain unchanged

return ciphertext

def caesar\_decrypt(ciphertext, k):

"""Decrypt the ciphertext using the Caesar cipher."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

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

plaintext += chr((ord(char) - base - k + 26) % 26 + base)

else:

plaintext += char # Non-alphabet characters remain unchanged

return plaintext

def main():

print("Caesar Cipher Program")

print("1. Encrypt")

print("2. Decrypt")

choice = int(input("Enter your choice (1 or 2): "))

if choice not in [1, 2]:

print("Invalid choice. Exiting.")

return

k = int(input("Enter the shift value (k) between 1 and 25: "))

if k < 1 or k > 25:

print("Invalid shift value. Exiting.")

return

text = input("Enter the text: ")

if choice == 1:

result = caesar\_encrypt(text, k)

print("Encrypted text:", result)

elif choice == 2:

result = caesar\_decrypt(text, k)

print("Decrypted text:", result)

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

main()

2.

import string

def generate\_cipher\_alphabet(keyword):

"""

Generates a monoalphabetic cipher alphabet using a keyword.

The keyword is followed by unused letters of the alphabet.

"""

keyword = "".join(dict.fromkeys(keyword.lower())) # Remove duplicate letters

remaining\_letters = "".join([ch for ch in string.ascii\_lowercase if ch not in keyword])

return keyword + remaining\_letters

def encrypt(plaintext, cipher\_alphabet):

"""

Encrypts the plaintext using the cipher alphabet.

"""

ciphertext = ""

for char in plaintext:

if char.isalpha():

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

ciphertext += cipher\_alphabet[ord(char.lower()) - ord('a')].upper() if char.isupper() else cipher\_alphabet[ord(char) - ord('a')]

else:

ciphertext += char # Non-alphabet characters remain unchanged

return ciphertext

def decrypt(ciphertext, cipher\_alphabet):

"""

Decrypts the ciphertext using the cipher alphabet.

"""

reverse\_alphabet = {cipher\_alphabet[i]: string.ascii\_lowercase[i] for i in range(len(cipher\_alphabet))}

plaintext = ""

for char in ciphertext:

if char.isalpha():

if char.isupper():

plaintext += reverse\_alphabet[char.lower()].upper()

else:

plaintext += reverse\_alphabet[char]

else:

plaintext += char # Non-alphabet characters remain unchanged

return plaintext

def main():

print("Monoalphabetic Substitution Cipher")

# User input

keyword = input("Enter the keyword: ").strip()

plaintext = input("Enter the plaintext message: ").strip()

# Generate cipher alphabet

cipher\_alphabet = generate\_cipher\_alphabet(keyword)

print("\nGenerated Cipher Alphabet:")

print(f"Plain: {string.ascii\_lowercase}")

print(f"Cipher: {cipher\_alphabet}")

# Encrypt the plaintext

ciphertext = encrypt(plaintext, cipher\_alphabet)

print("\nEncrypted Message:")

print(ciphertext)

# Decrypt the ciphertext

decrypted\_message = decrypt(ciphertext, cipher\_alphabet)

print("\nDecrypted Message:")

print(decrypted\_message)

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

main()

3.

def generate\_matrix(keyword):

"""

Generate a 5x5 Playfair matrix using the given keyword.

Combine the keyword with remaining unused letters of the alphabet.

"""

keyword = keyword.lower().replace("j", "i") # Treat 'j' as 'i'

matrix = []

seen = set()

alphabet = "abcdefghiklmnopqrstuvwxyz" # 'j' is excluded

# Add keyword to matrix

for char in keyword:

if char not in seen and char in alphabet:

seen.add(char)

matrix.append(char)

# Add remaining letters to matrix

for char in alphabet:

if char not in seen:

matrix.append(char)

# Convert list to 5x5 matrix

return [matrix[i:i + 5] for i in range(0, 25, 5)]

def find\_position(matrix, char):

"""

Find the row and column of a character in the matrix.

"""

for row\_idx, row in enumerate(matrix):

if char in row:

return row\_idx, row.index(char)

return None

def preprocess\_plaintext(plaintext):

"""

Prepare the plaintext for encryption:

- Replace 'j' with 'i'

- Insert 'x' between repeating letters in a digraph

- Add 'x' at the end if plaintext length is odd

"""

plaintext = plaintext.lower().replace("j", "i")

processed = ""

i = 0

while i < len(plaintext):

char1 = plaintext[i]

char2 = plaintext[i + 1] if i + 1 < len(plaintext) else "x"

if char1.isalpha():

if char1 == char2:

processed += char1 + "x"

i += 1

else:

processed += char1 + char2

i += 2

else:

i += 1

if len(processed) % 2 != 0:

processed += "x" # Add 'x' padding

return processed

def encrypt\_digraph(matrix, digraph):

"""

Encrypt a pair of letters using the Playfair matrix.

"""

r1, c1 = find\_position(matrix, digraph[0])

r2, c2 = find\_position(matrix, digraph[1])

if r1 == r2: # Same row

return matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]

elif c1 == c2: # Same column

return matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]

else: # Rectangle

return matrix[r1][c2] + matrix[r2][c1]

def decrypt\_digraph(matrix, digraph):

"""

Decrypt a pair of letters using the Playfair matrix.

"""

r1, c1 = find\_position(matrix, digraph[0])

r2, c2 = find\_position(matrix, digraph[1])

if r1 == r2: # Same row

return matrix[r1][(c1 - 1) % 5] + matrix[r2][(c2 - 1) % 5]

elif c1 == c2: # Same column

return matrix[(r1 - 1) % 5][c1] + matrix[(r2 - 1) % 5][c2]

else: # Rectangle

return matrix[r1][c2] + matrix[r2][c1]

def playfair\_encrypt(plaintext, keyword):

"""

Encrypt the plaintext using the Playfair cipher with the given keyword.

"""

matrix = generate\_matrix(keyword)

plaintext = preprocess\_plaintext(plaintext)

ciphertext = ""

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

ciphertext += encrypt\_digraph(matrix, plaintext[i:i + 2])

return ciphertext

def playfair\_decrypt(ciphertext, keyword):

"""

Decrypt the ciphertext using the Playfair cipher with the given keyword.

"""

matrix = generate\_matrix(keyword)

plaintext = ""

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

plaintext += decrypt\_digraph(matrix, ciphertext[i:i + 2])

return plaintext

def main():

print("Playfair Cipher")

# User inputs

keyword = "keyword"

plaintext = "hello world"

print(f"Keyword: {keyword}")

print(f"Plaintext: {plaintext}")

# Encrypt

ciphertext = playfair\_encrypt(plaintext, keyword)

print("\nEncrypted Message:")

print(ciphertext)

# Decrypt

decrypted\_message = playfair\_decrypt(ciphertext, keyword)

print("\nDecrypted Message:")

print(decrypted\_message)

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

main()

4.

def vigenere\_encrypt(plaintext, key):

"""

Encrypts plaintext using the Vigenère cipher.

"""

ciphertext = []

key = key.lower()

key\_index = 0

for char in plaintext:

if char.isalpha():

shift = ord(key[key\_index % len(key)]) - ord('a')

if char.islower():

encrypted\_char = chr((ord(char) - ord('a') + shift) % 26 + ord('a'))

else:

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

ciphertext.append(encrypted\_char)

key\_index += 1

else:

ciphertext.append(char)

return ''.join(ciphertext)

def vigenere\_decrypt(ciphertext, key):

"""

Decrypts ciphertext using the Vigenère cipher.

"""

plaintext = []

key = key.lower()

key\_index = 0

for char in ciphertext:

if char.isalpha():

shift = ord(key[key\_index % len(key)]) - ord('a')

if char.islower():

decrypted\_char = chr((ord(char) - ord('a') - shift) % 26 + ord('a'))

else:

decrypted\_char = chr((ord(char) - ord('A') - shift) % 26 + ord('A'))

plaintext.append(decrypted\_char)

key\_index += 1

else:

plaintext.append(char)

return ''.join(plaintext)

def main():

print("Polyalphabetic Substitution Cipher (Vigenère Cipher)")

plaintext = input("Enter plaintext: ")

key = input("Enter key: ")

# Encryption

ciphertext = vigenere\_encrypt(plaintext, key)

print("\nEncrypted Message:")

print(ciphertext)

# Decryption

decrypted\_message = vigenere\_decrypt(ciphertext, key)

print("\nDecrypted Message:")

print(decrypted\_message)

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

main()

5.

from math import gcd

def is\_coprime(a, m):

"""Check if a and m are coprime."""

return gcd(a, m) == 1

def find\_invalid\_a\_values(modulo):

"""Find values of 'a' that are not coprime with the given modulo."""

invalid\_a = [a for a in range(modulo) if gcd(a, modulo) != 1]

return invalid\_a

def affine\_encrypt(plaintext, a, b):

"""Encrypt the plaintext using the Affine Caesar Cipher."""

if not is\_coprime(a, 26):

raise ValueError(f"Key 'a' = {a} is not coprime with 26.")

ciphertext = ""

for char in plaintext:

if char.isalpha():

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

p = ord(char) - base

c = (a \* p + b) % 26

ciphertext += chr(base + c)

else:

ciphertext += char

return ciphertext

def affine\_decrypt(ciphertext, a, b):

"""Decrypt the ciphertext using the Affine Caesar Cipher."""

if not is\_coprime(a, 26):

raise ValueError(f"Key 'a' = {a} is not coprime with 26.")

a\_inverse = pow(a, -1, 26) # Modular multiplicative inverse of a under modulo 26

plaintext = ""

for char in ciphertext:

if char.isalpha():

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

c = ord(char) - base

p = (a\_inverse \* (c - b)) % 26

plaintext += chr(base + p)

else:

plaintext += char

return plaintext

# Main execution

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

print("Affine Caesar Cipher Analysis")

# Part (a): Are there any limitations on b?

print("\n(a) Analysis of 'b':")

print("There are no limitations on 'b'. It can take any value since it is a shift, and the modulo operation ensures wrapping.")

# Part (b): Determine invalid values of 'a'

print("\n(b) Invalid values of 'a':")

invalid\_a = find\_invalid\_a\_values(26)

print(f"Invalid 'a' values (not coprime with 26): {invalid\_a}")

# Example encryption and decryption

plaintext = "software"

a, b = 5, 8 # Example keys

try:

print("\nExample Encryption and Decryption:")

ciphertext = affine\_encrypt(plaintext, a, b)

print(f"Plaintext: {plaintext}")

print(f"Encrypted: {ciphertext}")

decrypted\_text = affine\_decrypt(ciphertext, a, b)

print(f"Decrypted: {decrypted\_text}")

except ValueError as e:

print(e)

6.

from math import gcd

def mod\_inverse(a, m):

"""Find the modular inverse of a under modulo m."""

for x in range(1, m):

if (a \* x) % m == 1:

return x

return None

def encrypt\_affine(plaintext, a, b):

"""Encrypt a plaintext using an affine cipher."""

if gcd(a, 26) != 1:

raise ValueError(f"'a' must be coprime with 26. '{a}' is invalid.")

ciphertext = ""

for char in plaintext:

if char.isalpha():

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

p = ord(char) - base

c = (a \* p + b) % 26

ciphertext += chr(base + c)

else:

ciphertext += char

return ciphertext

def decrypt\_affine(ciphertext, a, b):

"""Decrypt a ciphertext encrypted using an affine cipher."""

a\_inverse = mod\_inverse(a, 26)

if not a\_inverse:

raise ValueError(f"No modular inverse for 'a = {a}' under mod 26.")

plaintext = ""

for char in ciphertext:

if char.isalpha():

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

c = ord(char) - base

p = (a\_inverse \* (c - b)) % 26

plaintext += chr(base + p)

else:

plaintext += char

return plaintext

def break\_affine\_cipher(ciphertext, most\_freq\_cipher='B', second\_freq\_cipher='U'):

"""Break an affine cipher given letter frequencies."""

print(f"Breaking ciphertext: {ciphertext}")

# Frequencies given in the question

cipher\_most\_freq = ord(most\_freq\_cipher) - ord('A') # 1 for 'B'

cipher\_second\_freq = ord(second\_freq\_cipher) - ord('A') # 20 for 'U'

# Assuming these map to 'E' (4) and 'T' (19)

plain\_most\_freq = 4

plain\_second\_freq = 19

# Solve for a and b using:

# C1 = (a \* P1 + b) % 26

# C2 = (a \* P2 + b) % 26

# Resulting in two linear equations:

# 1 = (a \* 4 + b) % 26

# 20 = (a \* 19 + b) % 26

a\_candidates = []

for a in range(1, 26):

if gcd(a, 26) == 1: # Ensure 'a' is coprime with 26

b = (cipher\_most\_freq - a \* plain\_most\_freq) % 26

if (a \* plain\_second\_freq + b) % 26 == cipher\_second\_freq:

a\_candidates.append((a, b))

if not a\_candidates:

print("No valid key pair (a, b) found.")

return

print("\nPossible Key Pairs (a, b):", a\_candidates)

for a, b in a\_candidates:

try:

plaintext = decrypt\_affine(ciphertext, a, b)

print(f"\nDecrypted with (a={a}, b={b}): {plaintext}")

except ValueError as e:

print(e)

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

# Encryption example

plaintext = "ramcharan"

a, b = 15, 23 # Encryption keys

print(f"Plaintext: {plaintext}")

ciphertext = encrypt\_affine(plaintext, a, b)

print(f"Encrypted (a={a}, b={b}): {ciphertext}")

# Decryption example

decrypted\_text = decrypt\_affine(ciphertext, a, b)

print(f"Decrypted: {decrypted\_text}")

# Breaking a cipher

example\_ciphertext = "BJUWBJUDWH"

break\_affine\_cipher(example\_ciphertext)

7.

import collections

def frequency\_analysis(ciphertext):

"""

Perform frequency analysis on the ciphertext to identify the most common characters.

"""

frequency = collections.Counter(ciphertext)

sorted\_frequency = frequency.most\_common()

return sorted\_frequency

def substitution\_mapping(ciphertext, mapping):

"""

Apply a substitution mapping to the ciphertext to decode it.

"""

decoded\_text = []

for char in ciphertext:

if char in mapping:

decoded\_text.append(mapping[char])

else:

decoded\_text.append(char)

return ''.join(decoded\_text)

def main():

# Ciphertext provided in the problem

ciphertext = ("53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:"

"‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956"

"\*2(5\*—4)8¶8\*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;"

"4)485†528806\*81(‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;")

print("Ciphertext:\n", ciphertext)

# Step 1: Frequency Analysis

print("\nStep 1: Frequency Analysis")

freq = frequency\_analysis(ciphertext)

for char, count in freq:

print(f"'{char}': {count} times")

# Step 2: Initial Mapping Based on Observations

# Based on English letter frequencies and guessing common words like "the"

mapping = {

'‡': 'e',

'†': 't',

'8': 'h',

'4': 'o',

'5': 'a',

'6': 'n',

'0': 'r',

'3': 's',

'\*': 'i',

'(': 'l',

')': 'd',

';': 'u',

':': 'y',

'9': 'w',

'1': 'c',

'2': 'm',

'¶': 'f',

'—': 'g',

']': 'b',

'?': 'p'

}

# Step 3: Decode the Ciphertext

print("\nStep 3: Decoding Using Substitution")

decoded\_message = substitution\_mapping(ciphertext, mapping)

print("\nDecoded Message:\n", decoded\_message)

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

main()

8.

import string  
  
 def generate\_cipher\_alphabet(key):  
 *"""  
 Generates a ciphertext alphabet based on the given key.  
 """* key = "".join(dict.fromkeys(key)) # Remove duplicates from key  
 remaining\_letters = "".join([char for char in string.ascii\_lowercase if char not in key])  
 return key + remaining\_letters  
  
  
 def encrypt(plaintext, cipher\_alphabet):  
 *"""  
 Encrypts the plaintext using the given cipher alphabet.  
 """* plaintext = plaintext.lower()  
 ciphertext = ""  
 plaintext\_alphabet = string.ascii\_lowercase  
  
 for char in plaintext:  
 if char in plaintext\_alphabet:  
 index = plaintext\_alphabet.index(char)  
 ciphertext += cipher\_alphabet[index]  
 else:  
 ciphertext += char # Non-alphabetic characters remain unchanged  
 return ciphertext  
  
  
 def decrypt(ciphertext, cipher\_alphabet):  
 *"""  
 Decrypts the ciphertext using the given cipher alphabet.  
 """* ciphertext = ciphertext.lower()  
 plaintext = ""  
 plaintext\_alphabet = string.ascii\_lowercase  
  
 for char in ciphertext:  
 if char in cipher\_alphabet:  
 index = cipher\_alphabet.index(char)  
 plaintext += plaintext\_alphabet[index]  
 else:  
 plaintext += char # Non-alphabetic characters remain unchanged  
 return plaintext  
  
  
 def main():  
 print("Monoalphabetic Substitution Cipher")  
 key = input("Enter a key (unique sequence of letters): ").lower()  
 cipher\_alphabet = generate\_cipher\_alphabet(key)  
  
 print(f"Cipher Alphabet: {cipher\_alphabet}")  
  
 plaintext = input("Enter the plaintext: ")  
 ciphertext = encrypt(plaintext, cipher\_alphabet)  
 print(f"Encrypted Text: {ciphertext}")  
  
 decrypted\_text = decrypt(ciphertext, cipher\_alphabet)  
 print(f"Decrypted Text: {decrypted\_text}")  
  
  
 if \_\_name\_\_ == "\_\_main\_\_":  
 main()

9.

# Function to create the Playfair matrix

def create\_playfair\_matrix(key):

# Remove duplicates and make all letters uppercase, replace 'J' with 'I'

key = key.upper().replace('J', 'I')

matrix = ""

# Fill the matrix with the key, avoiding repeated characters

for char in key:

if char not in matrix and char.isalpha():

matrix += char

# Add remaining letters of the alphabet (excluding 'J')

alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"

for char in alphabet:

if char not in matrix:

matrix += char

# Create a 5x5 matrix (list of lists)

playfair\_matrix = [list(matrix[i:i + 5]) for i in range(0, len(matrix), 5)]

return playfair\_matrix

# Function to find the position of a letter in the matrix

def find\_position(matrix, char):

for i, row in enumerate(matrix):

for j, letter in enumerate(row):

if letter == char:

return i, j

return None

# Function to prepare the message (split into digraphs)

def prepare\_message(message):

message = message.replace('J', 'I').upper() # Replace 'J' with 'I' and convert to uppercase

digraphs = []

i = 0

while i < len(message):

if i + 1 < len(message) and message[i] != message[i + 1]:

digraphs.append(message[i:i + 2])

i += 2

else:

digraphs.append(message[i] + 'X') # Add 'X' if the letters are the same or last single letter

i += 1

return digraphs

# Function to encrypt the message

def encrypt(message, matrix):

digraphs = prepare\_message(message)

encrypted\_message = []

for digraph in digraphs:

r1, c1 = find\_position(matrix, digraph[0])

r2, c2 = find\_position(matrix, digraph[1])

# Case 1: Same row

if r1 == r2:

encrypted\_message.append(matrix[r1][(c1 + 1) % 5])

encrypted\_message.append(matrix[r2][(c2 + 1) % 5])

# Case 2: Same column

elif c1 == c2:

encrypted\_message.append(matrix[(r1 + 1) % 5][c1])

encrypted\_message.append(matrix[(r2 + 1) % 5][c2])

# Case 3: Rectangle (opposite corners)

else:

encrypted\_message.append(matrix[r1][c2])

encrypted\_message.append(matrix[r2][c1])

return ''.join(encrypted\_message)

# Function to decrypt the message

def decrypt(message, matrix):

digraphs = prepare\_message(message)

decrypted\_message = []

for digraph in digraphs:

r1, c1 = find\_position(matrix, digraph[0])

r2, c2 = find\_position(matrix, digraph[1])

# Case 1: Same row

if r1 == r2:

decrypted\_message.append(matrix[r1][(c1 - 1) % 5])

decrypted\_message.append(matrix[r2][(c2 - 1) % 5])

# Case 2: Same column

elif c1 == c2:

decrypted\_message.append(matrix[(r1 - 1) % 5][c1])

decrypted\_message.append(matrix[(r2 - 1) % 5][c2])

# Case 3: Rectangle (opposite corners)

else:

decrypted\_message.append(matrix[r1][c2])

decrypted\_message.append(matrix[r2][c1])

return ''.join(decrypted\_message)

# Main function to demonstrate Playfair cipher encryption and decryption

def main():

key = 'PT109' # The keyword used to create the matrix

message = 'KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBNT GONEY CUZWR GDSON SXBOU YWRHE BAAHY USED' # The message

# Create the Playfair matrix based on the keyword

matrix = create\_playfair\_matrix(key)

# Encrypt the message

encrypted\_message = encrypt(message.replace(' ', ''), matrix)

print("Encrypted message:", encrypted\_message)

# Decrypt the message

decrypted\_message = decrypt(encrypted\_message, matrix)

print("Decrypted message:", decrypted\_message)

# Run the main function

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

main()

10.

def create\_matrix(key):

alphabet = "abcdefghiklmnopqrstuvwxyz"

key = "".join(dict.fromkeys(key + alphabet))

matrix = [key[i:i+5] for i in range(0, 25, 5)]

return matrix

def find\_position(matrix, char):

for row\_idx, row in enumerate(matrix):

if char in row:

return (row\_idx, row.index(char))

return None

def process\_pairs(text, mode, matrix):

pairs = []

text = text.replace("j", "i")

i = 0

while i < len(text):

if i + 1 < len(text) and text[i] != text[i + 1]:

pairs.append(text[i] + text[i + 1])

i += 2

else:

pairs.append(text[i] + "x")

i += 1

result = ""

for pair in pairs:

row1, col1 = find\_position(matrix, pair[0])

row2, col2 = find\_position(matrix, pair[1])

if row1 == row2:

if mode == "encrypt":

result += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]

else:

result += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]

elif col1 == col2:

if mode == "encrypt":

result += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]

else:

result += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]

else:

result += matrix[row1][col2] + matrix[row2][col1]

return result

def playfair\_encrypt\_decrypt():

key = input("Enter the encryption key: ").lower().replace(" ", "")

message = input("Enter the message: ").lower().replace(" ", "")

matrix = create\_matrix(key)

print("\nPlayfair Cipher Matrix:")

for row in matrix:

print(" ".join(row))

cipher\_text = process\_pairs(message, "encrypt", matrix)

print(f"\nEncrypted Message (Cipher Text): {cipher\_text}")

decrypted\_text = process\_pairs(cipher\_text, "decrypt", matrix)

print(f"Decrypted Message: {decrypted\_text.rstrip('x')}")

playfair\_encrypt\_decrypt()

11.

import math

def calculate\_keys(n):

"""

Calculate the total number of possible keys (n!) and an approximation of effectively unique keys.

"""

# Total number of possible keys (n factorial)

total\_keys = math.factorial(n)

log2\_total\_keys = math.log2(total\_keys)

# Approximation for effectively unique keys (based on equivalence reduction)

effectively\_unique\_keys = 2 \*\* (n - 6) # Simple approximation for larger n

# Results

print(f"\nFor n = {n}:")

print(f"Total possible keys (n!): {total\_keys}")

print(f"Approximate total keys as power of 2: 2^{int(log2\_total\_keys)}")

print(f"Effectively unique keys: 2^{int(n - 6)}")

def simple\_substitution\_encrypt(message, shift=3):

"""

Simple Caesar cipher (substitution cipher) for encryption.

Each letter in the message is shifted by 'shift' positions in the alphabet.

"""

encrypted\_message = []

# Loop through each character in the message

for char in message:

if char.isalpha(): # Only encrypt alphabetic characters

shift\_base = 65 if char.isupper() else 97

encrypted\_char = chr(((ord(char) - shift\_base + shift) % 26) + shift\_base)

encrypted\_message.append(encrypted\_char)

else:

encrypted\_message.append(char) # Non-alphabet characters remain unchanged

return ''.join(encrypted\_message)

def main():

# User input for key space calculation

n = int(input("Enter the value of n (number of elements for key space): "))

calculate\_keys(n)

# Simple message encryption

message = "This is a secret message!"

print(f"\nOriginal Message: {message}")

# Encrypt the message using a Caesar cipher (simple substitution)

encrypted\_message = simple\_substitution\_encrypt(message, shift=3)

print(f"Encrypted Message: {encrypted\_message}")

# Decrypt the message using the inverse of the shift (shift = -3)

decrypted\_message = simple\_substitution\_encrypt(encrypted\_message, shift=-3)

print(f"Decrypted Message: {decrypted\_message}")

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

main()

12.

import numpy as np

# Helper functions

def char\_to\_num(char):

"""Convert a character to a number (A=0, B=1, ..., Z=25)."""

return ord(char) - ord('a')

def num\_to\_char(num):

"""Convert a number to a character (0=A, 1=B, ..., 25=Z)."""

return chr(num + ord('a'))

def hill\_encrypt(plaintext, key):

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

n = key.shape[0] # Size of the key matrix

plaintext = plaintext.replace(" ", "").lower() # Remove spaces and make lowercase

# Pad plaintext to be a multiple of n

while len(plaintext) % n != 0:

plaintext += 'x'

# Convert plaintext to numerical format

plaintext\_nums = [char\_to\_num(char) for char in plaintext]

plaintext\_matrix = np.array(plaintext\_nums).reshape(-1, n).T

# Encrypt using matrix multiplication modulo 26

ciphertext\_matrix = np.dot(key, plaintext\_matrix) % 26

ciphertext = ''.join(num\_to\_char(num) for num in ciphertext\_matrix.T.flatten())

return ciphertext

def hill\_decrypt(ciphertext, key):

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

n = key.shape[0]

ciphertext = ciphertext.lower() # Ensure lowercase

# Convert ciphertext to numerical format

ciphertext\_nums = [char\_to\_num(char) for char in ciphertext]

ciphertext\_matrix = np.array(ciphertext\_nums).reshape(-1, n).T

# Calculate modular inverse of the key matrix

determinant = int(round(np.linalg.det(key))) # Determinant of the key matrix

determinant\_inv = pow(determinant, -1, 26) # Modular inverse of determinant modulo 26

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

# Decrypt using matrix multiplication modulo 26

plaintext\_matrix = np.dot(key\_inv, ciphertext\_matrix) % 26

plaintext = ''.join(num\_to\_char(num) for num in plaintext\_matrix.T.flatten())

return plaintext

def main():

# Define the key matrix

key = np.array([[9, 4], [5, 7]])

# Input plaintext

plaintext = "meet me at the usual place at ten rather than eight oclock"

print("Plaintext:", plaintext)

# Encrypt the plaintext

ciphertext = hill\_encrypt(plaintext, key)

print("Ciphertext:", ciphertext)

# Decrypt the ciphertext

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

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

if \_\_name\_\_

== "\_\_main\_\_":

main()

13.

import numpy as np

# Function to create a 2x2 key matrix from a 4-character key string

def create\_key\_matrix(key\_string):

if len(key\_string) != 4:

print("Key must be 4 characters.")

return None

key\_matrix = np.array([[ord(key\_string[0]) - 65, ord(key\_string[1]) - 65],

[ord(key\_string[2]) - 65, ord(key\_string[3]) - 65]])

return key\_matrix

# Function to encrypt the plaintext

def encrypt(plaintext, key\_matrix):

# Ensure plaintext length is even

if len(plaintext) % 2 != 0:

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

ciphertext = ""

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

# Convert letters to numbers (A=0, B=1, ..., Z=25)

block = [ord(plaintext[i]) - 65, ord(plaintext[i+1]) - 65]

# Matrix multiplication: (key\_matrix \* block) mod 26

encrypted\_block = np.dot(key\_matrix, block) % 26

# Convert the result back to letters

ciphertext += chr(encrypted\_block[0] + 65) + chr(encrypted\_block[1] + 65)

return ciphertext

# Function to decrypt the ciphertext

def decrypt(ciphertext, key\_matrix):

# Calculate the inverse of the key matrix

det = int(np.linalg.det(key\_matrix)) # Determinant of the key matrix

if det == 0:

return "Inverse not possible for this matrix."

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

key\_inv = np.round(key\_inv).astype(int) % 26

# Decrypting the ciphertext

plaintext = ""

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

# Convert letters to numbers

block = [ord(ciphertext[i]) - 65, ord(ciphertext[i+1]) - 65]

# Matrix multiplication: (key\_inv \* block) mod 26

decrypted\_block = np.dot(key\_inv, block) % 26

# Convert the result back to letters

plaintext += chr(decrypted\_block[0] + 65) + chr(decrypted\_block[1] + 65)

return plaintext

# Main function to demonstrate encryption and decryption

def main():

key\_string = "GYBN" # Example 4-character key (2x2 matrix)

key\_matrix = create\_key\_matrix(key\_string)

if key\_matrix is None:

return # Exit if invalid key

plaintext = "HELLO" # Sample plaintext

print(f"Plaintext: {plaintext}")

# Encrypt the plaintext

ciphertext = encrypt(plaintext, key\_matrix)

print(f"Ciphertext: {ciphertext}")

# Decrypt the ciphertext

decrypted\_text = decrypt(ciphertext, key\_matrix)

print(f"Decrypted text: {decrypted\_text}")

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

main()

14.

def char\_to\_num(char):

"""Convert a character to a number (0 for 'a', 1 for 'b', ..., 25 for 'z')."""

return ord(char) - ord('a')

def num\_to\_char(num):

"""Convert a number to a character (0 -> 'a', 1 -> 'b', ..., 25 -> 'z')."""

return chr(num + ord('a'))

def encrypt\_vigenere(plaintext, key):

"""Encrypt plaintext using the Vigenère cipher with the given key."""

plaintext = plaintext.replace(" ", "").lower() # Remove spaces and convert to lowercase

ciphertext = []

for i, char in enumerate(plaintext):

shift = key[i % len(key)]

cipher\_char = (char\_to\_num(char) + shift) % 26

ciphertext.append(num\_to\_char(cipher\_char))

return ''.join(ciphertext)

def decrypt\_vigenere(ciphertext, key):

"""Decrypt ciphertext using the Vigenère cipher with the given key."""

plaintext = []

for i, char in enumerate(ciphertext):

shift = key[i % len(key)]

plain\_char = (char\_to\_num(char) - shift) % 26

plaintext.append(num\_to\_char(plain\_char))

return ''.join(plaintext)

def find\_key\_for\_target(ciphertext, target\_plaintext):

"""Find a key that can decrypt the ciphertext to the target plaintext."""

ciphertext = ciphertext.lower()

target\_plaintext = target\_plaintext.replace(" ", "").lower()

key = []

for c, t in zip(ciphertext, target\_plaintext):

key.append((char\_to\_num(c) - char\_to\_num(t)) % 26)

return key

def main\_vigenere():

"""Main function to demonstrate encryption, decryption, and key recovery."""

plaintext = "send more money"

key = [9, 0, 1, 7, 23, 15, 21, 14, 11, 11, 2, 8, 9]

print("Original Key:", key)

# Encrypt the plaintext

ciphertext = encrypt\_vigenere(plaintext, key)

print("\nEncrypted Ciphertext:", ciphertext)

# Decrypt to produce the target plaintext "cash not needed"

target\_plaintext = "cash not needed"

chosen\_key = find\_key\_for\_target(ciphertext, target\_plaintext)

print("\nKey to produce target plaintext:", chosen\_key)

decrypted\_text = decrypt\_vigenere(ciphertext, chosen\_key)

print("\nDecrypted Text using chosen key:", decrypted\_text)

# Run the main function

main\_vigenere()

15.

from collections import Counter

# Expected English letter frequencies (normalized)

ENGLISH\_FREQ = {

'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702, 'f': 2.228,

'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153, 'k': 0.772, 'l': 4.025,

'm': 2.406, 'n': 6.749, 'o': 7.507, 'p': 1.929, 'q': 0.095, 'r': 5.987,

's': 6.327, 't': 9.056, 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150,

'y': 1.974, 'z': 0.074

}

def additive\_decrypt(ciphertext, key):

"""Decrypt ciphertext using the additive cipher and key."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

# Shift for lowercase and uppercase separately

shifted = (ord(char.lower()) - ord('a') - key) % 26 + ord('a')

# Preserve the case (upper or lower)

plaintext += chr(shifted).upper() if char.isupper() else chr(shifted)

else:

plaintext += char

return plaintext

def calculate\_score(text):

"""Calculate a score for the text based on English letter frequency."""

text = text.lower()

letter\_count = Counter(char for char in text if char.isalpha())

total\_letters = sum(letter\_count.values())

if total\_letters == 0:

return float('inf') # No letters to compare, return a large score

# Compare text frequency with English frequency

score = 0

for letter, expected\_freq in ENGLISH\_FREQ.items():

observed\_freq = (letter\_count[letter] / total\_letters) \* 100

score += abs(observed\_freq - expected\_freq)

return score

def frequency\_attack(ciphertext, top\_n=10):

"""Perform a frequency attack on the additive cipher."""

results = []

for key in range(26): # Try all possible keys (0-25)

plaintext = additive\_decrypt(ciphertext, key)

score = calculate\_score(plaintext)

results.append((key, plaintext, score))

# Sort results by score (lower is better)

results.sort(key=lambda x: x[2])

return results[:top\_n]

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

# Getting inputs outside the code

ciphertext = input("Enter the ciphertext: ")

top\_n = int(input("Enter the number of top possible plaintexts to display: "))

# Perform the attack

results = frequency\_attack(ciphertext, top\_n)

# Display results

print("\nTop possible plaintexts:")

for rank, (key, plaintext, score) in enumerate(results, start=1):

print(f"{rank}. Key = {key}, Score = {score:.2f}, Plaintext = {plaintext}")

16.

from collections import Counter

# English letter frequencies in percentages

ENGLISH\_FREQ = {

'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702, 'f': 2.228,

'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153, 'k': 0.772, 'l': 4.025,

'm': 2.406, 'n': 6.749, 'o': 7.507, 'p': 1.929, 'q': 0.095, 'r': 5.987,

's': 6.327, 't': 9.056, 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150,

'y': 1.974, 'z': 0.074

}

def shift\_decrypt(ciphertext, key):

"""Decrypt using a Caesar shift with a given key."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

shifted = (ord(char.lower()) - ord('a') - key) % 26 + ord('a')

plaintext += chr(shifted).upper() if char.isupper() else chr(shifted)

else:

plaintext += char

return plaintext

def calculate\_score(text):

"""Score the text based on English letter frequencies."""

text = text.lower()

letter\_count = Counter(char for char in text if char.isalpha())

total\_letters = sum(letter\_count.values())

if total\_letters == 0:

return float('inf') # Penalize empty plaintexts

score = 0

for letter, expected\_freq in ENGLISH\_FREQ.items():

observed\_freq = (letter\_count[letter] / total\_letters) \* 100

score += abs(observed\_freq - expected\_freq)

return score

def frequency\_attack(ciphertext, top\_n=10):

"""Perform a frequency attack on the monoalphabetic cipher."""

results = []

for key in range(26):

plaintext = shift\_decrypt(ciphertext, key)

score = calculate\_score(plaintext)

results.append((key, plaintext, score))

results.sort(key=lambda x: x[2]) # Sort by score (lower is better)

return results[:top\_n]

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

# User input

ciphertext = input("Enter the ciphertext: ")

top\_n = int(input("Enter the number of top possible plaintexts to display: "))

# Perform frequency analysis

results = frequency\_attack(ciphertext, top\_n)

# Output results

print("\nTop possible plaintexts:")

for rank, (key, plaintext, score) in enumerate(results, start=1):

print(f"{rank}. Key = {key}, Score = {score:.2f}, Plaintext = {plaintext}")

17.

# Python3 code for the above approach

# Hexadecimal to binary conversion

def hex2bin(s):

mp = {'0': "0000",

'1': "0001",

'2': "0010",

'3': "0011",

'4': "0100",

'5': "0101",

'6': "0110",

'7': "0111",

'8': "1000",

'9': "1001",

'A': "1010",

'B': "1011",

'C': "1100",

'D': "1101",

'E': "1110",

'F': "1111"}

bin = ""

for i in range(len(s)):

bin = bin + mp[s[i]]

return bin

# Binary to hexadecimal conversion

def bin2hex(s):

mp = {"0000": '0',

"0001": '1',

"0010": '2',

"0011": '3',

"0100": '4',

"0101": '5',

"0110": '6',

"0111": '7',

"1000": '8',

"1001": '9',

"1010": 'A',

"1011": 'B',

"1100": 'C',

"1101": 'D',

"1110": 'E',

"1111": 'F'}

hex = ""

for i in range(0, len(s), 4):

ch = ""

ch = ch + s[i]

ch = ch + s[i + 1]

ch = ch + s[i + 2]

ch = ch + s[i + 3]

hex = hex + mp[ch]

return hex

# Binary to decimal conversion

def bin2dec(binary):

binary1 = binary

decimal, i, n = 0, 0, 0

while(binary != 0):

dec = binary % 10

decimal = decimal + dec \* pow(2, i)

binary = binary//10

i += 1

return decimal

# Decimal to binary conversion

def dec2bin(num):

res = bin(num).replace("0b", "")

if(len(res) % 4 != 0):

div = len(res) / 4

div = int(div)

counter = (4 \* (div + 1)) - len(res)

for i in range(0, counter):

res = '0' + res

return res

# Permute function to rearrange the bits

def permute(k, arr, n):

permutation = ""

for i in range(0, n):

permutation = permutation + k[arr[i] - 1]

return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

s = ""

for i in range(nth\_shifts):

for j in range(1, len(k)):

s = s + k[j]

s = s + k[0]

k = s

s = ""

return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

ans = ""

for i in range(len(a)):

if a[i] == b[i]:

ans = ans + "0"

else:

ans = ans + "1"

return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table

initial\_perm = [58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table

exp\_d = [32, 1, 2, 3, 4, 5, 4, 5,

6, 7, 8, 9, 8, 9, 10, 11,

12, 13, 12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21, 20, 21,

22, 23, 24, 25, 24, 25, 26, 27,

28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table

per = [16, 7, 20, 21,

29, 12, 28, 17,

1, 15, 23, 26,

5, 18, 31, 10,

2, 8, 24, 14,

32, 27, 3, 9,

19, 13, 30, 6,

22, 11, 4, 25]

# S-box Table

sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

[0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

[4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table

final\_perm = [40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25]

def encrypt(pt, rkb, rk):

pt = hex2bin(pt)

# Initial Permutation

pt = permute(pt, initial\_perm, 64)

print("After initial permutation", bin2hex(pt))

# Splitting

left = pt[0:32]

right = pt[32:64]

for i in range(0, 16):

# Expansion D-box: Expanding the 32 bits data into 48 bits

right\_expanded = permute(right, exp\_d, 48)

# XOR RoundKey[i] and right\_expanded

xor\_x = xor(right\_expanded, rkb[i])

# S-boxex: substituting the value from s-box table by calculating row and column

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(

int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

# Straight D-box: After substituting rearranging the bits

sbox\_str = permute(sbox\_str, per, 32)

# XOR left and sbox\_str

result = xor(left, sbox\_str)

left = result

# Swapper

if(i != 15):

left, right = right, left

print("Round ", i + 1, " ", bin2hex(left),

" ", bin2hex(right), " ", rk[i])

# Combination

combine = left + right

# Final permutation: final rearranging of bits to get cipher text

cipher\_text = permute(combine, final\_perm, 64)

return cipher\_text

pt = "123456ABCD132536"

key = "AABB09182736CCDD"

# Key generation

# --hex to binary

key = hex2bin(key)

# --parity bit drop table

keyp = [57, 49, 41, 33, 25, 17, 9,

1, 58, 50, 42, 34, 26, 18,

10, 2, 59, 51, 43, 35, 27,

19, 11, 3, 60, 52, 44, 36,

63, 55, 47, 39, 31, 23, 15,

7, 62, 54, 46, 38, 30, 22,

14, 6, 61, 53, 45, 37, 29,

21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

# Number of bit shifts

shift\_table = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits

key\_comp = [14, 17, 11, 24, 1, 5,

3, 28, 15, 6, 21, 10,

23, 19, 12, 4, 26, 8,

16, 7, 27, 20, 13, 2,

41, 52, 31, 37, 47, 55,

30, 40, 51, 45, 33, 48,

44, 49, 39, 56, 34, 53,

46, 42, 50, 36, 29, 32]

# Splitting

left = key[0:28] # rkb for RoundKeys in binary

right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []

rk = []

for i in range(0, 16):

# Shifting the bits by nth shifts by checking from shift table

left = shift\_left(left, shift\_table[i])

right = shift\_left(right, shift\_table[i])

# Combination of left and right string

combine\_str = left + right

# Compression of key from 56 to 48 bits

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

print("Encryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

print("Cipher Text : ", cipher\_text)

print("Decryption")

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

print("Plain Text : ", text)

# This code is contributed by Aditya Jain

18.def generate\_subkeys(initial\_key):

"""

Generate 16 subkeys for DES, ensuring that:

- The first 24 bits of each subkey come from a specific subset of 28 bits.

- The second 24 bits of each subkey come from a disjoint subset of 28 bits.

"""

if len(initial\_key) != 8:

raise ValueError("Initial key must be 8 bytes (64 bits) long")

# Define the shift schedule for each round

shift\_schedule = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# Split the initial key into two 28-bit halves

left\_half = initial\_key[:4] # First 4 bytes

right\_half = initial\_key[4:] # Last 4 bytes

subkeys = []

for shift in shift\_schedule:

# Rotate the halves by the shift amount

left\_half = rotate\_left(left\_half, shift)

right\_half = rotate\_left(right\_half, shift)

# Combine halves to form a 48-bit subkey

subkey = left\_half[:3] + right\_half[:3]

subkeys.append(subkey)

return subkeys

def rotate\_left(data, shift):

"""

Rotate the bits of `data` to the left by `shift` positions.

"""

bits = ''.join(f"{byte:08b}" for byte in data)

rotated\_bits = bits[shift:] + bits[:shift]

rotated\_bytes = [int(rotated\_bits[i:i+8], 2) for i in range(0, len(rotated\_bits), 8)]

return bytes(rotated\_bytes)

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

# Example initial key (8 bytes)

initial\_key = b"12345678"

# Generate subkeys

subkeys = generate\_subkeys(initial\_key)

# Display subkeys

for i, subkey in enumerate(subkeys, start=1):

print(f"Subkey {i}: {subkey.hex()}")

21.

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default\_backend

import os

def pad\_custom(data, block\_size):

"""Custom padding: Add a 1 bit and fill the rest with 0 bits."""

padding\_length = block\_size - (len(data) % block\_size)

padding = b'\x80' + b'\x00' \* (padding\_length - 1) # 1 bit followed by 0 bits

return data + padding

def unpad\_custom(data, block\_size):

"""Remove custom padding."""

# Strip off the padding bytes

return data.rstrip(b'\x00').rstrip(b'\x80')

def ecb\_mode(key, plaintext):

"""Encrypt and decrypt using ECB mode."""

block\_size = algorithms.AES.block\_size // 8

padded\_plaintext = pad\_custom(plaintext, block\_size)

cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, unpad\_custom(decrypted\_data, block\_size)

def cbc\_mode(key, plaintext, iv):

"""Encrypt and decrypt using CBC mode."""

block\_size = algorithms.AES.block\_size // 8

padded\_plaintext = pad\_custom(plaintext, block\_size)

cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, unpad\_custom(decrypted\_data, block\_size)

def cfb\_mode(key, plaintext, iv):

"""Encrypt and decrypt using CFB mode."""

cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, decrypted\_data

# Testing the implementation

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

# Use a 128-bit (16-byte) key and IV

key = b"1234567890abcdef" # Fixed key for demonstration

iv = b"abcdef1234567890" # Fixed IV for CBC and CFB

plaintext = b"HELLO WORLD!!!" # Example plaintext (multiple of block size)

print("Original plaintext:", plaintext)

# ECB Mode

ecb\_ciphertext, ecb\_decrypted = ecb\_mode(key, plaintext)

print("\nECB Mode:")

print("Ciphertext (hex):", ecb\_ciphertext.hex())

print("Decrypted:", ecb\_decrypted)

# CBC Mode

cbc\_ciphertext, cbc\_decrypted = cbc\_mode(key, plaintext, iv)

print("\nCBC Mode:")

print("Ciphertext (hex):", cbc\_ciphertext.hex())

print("Decrypted:", cbc\_decrypted)

# CFB Mode

cfb\_ciphertext, cfb\_decrypted = cfb\_mode(key, plaintext, iv)

print("\nCFB Mode:")

print("Ciphertext (hex):", cfb\_ciphertext.hex())

print("Decrypted:", cfb\_decrypted)

24.

# Extended Euclidean Algorithm to find modular inverse

def extended\_gcd(a, b):

if b == 0:

return a, 1, 0

gcd, x1, y1 = extended\_gcd(b, a % b)

x = y1

y = x1 - (a // b) \* y1

return gcd, x, y

# Function to find the modular inverse

def mod\_inverse(e, phi\_n):

gcd, x, y = extended\_gcd(e, phi\_n)

if gcd != 1:

raise Exception('Inverse does not exist')

else:

return x % phi\_n

# Step 1: Trial-and-error to find p and q

def find\_factors(n):

for i in range(2, int(n \*\* 0.5) + 1):

if n % i == 0:

p = i

q = n // i

return p, q

return None, None

# Given values

e = 31

n = 3599

# Step 2: Find p and q (factors of n)

p, q = find\_factors(n)

if p is None or q is None:

print("Could not find factors of n")

else:

print(f"Factors of n: p = {p}, q = {q}")

# Step 3: Compute Euler's Totient Function φ(n)

phi\_n = (p - 1) \* (q - 1)

print(f"φ(n) = {phi\_n}")

# Step 4: Find the modular inverse of e modulo φ(n) to get d

d = mod\_inverse(e, phi\_n)

print(f"Private key (d) = {d}")

25.def gcd(a, b):

"""Helper function to calculate the greatest common divisor (GCD)."""

while b:

a, b = b, a % b

return a

def main():

# Given values

n = 55 # Example value for n = p \* q, where p = 5, q = 11

e = 3 # Public key

plaintext\_block = 15 # A plaintext block with a common factor with n

print("Given:")

print("n =", n)

print("e =", e)

print("Plaintext block =", plaintext\_block)

# Step 1: Check if the plaintext block has a common factor with n

common\_factor = gcd(plaintext\_block, n)

if common\_factor > 1:

print("\nCommon factor found:", common\_factor)

print("This means n can be factored as:")

print("p =", common\_factor)

print("q =", n // common\_factor)

print("\nRSA security is compromised because the private key can now be derived.")

else:

print("\nNo common factor found. RSA remains secure.")

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

main()

26.import random

# Helper function to calculate the greatest common divisor

def gcd(a, b):

while b:

a, b = b, a % b

return a

# Helper function to find modular inverse

def modinv(a, m):

m0, x0, x1 = m, 0, 1

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

if x1 < 0:

x1 += m0

return x1

# RSA Key Generation

def generate\_keys():

# Generate two small prime numbers for simplicity

p, q = 13, 17 # Small primes for demonstration

n = p \* q

phi\_n = (p - 1) \* (q - 1)

# Choose a public exponent e such that gcd(e, phi\_n) = 1

e = 3

while gcd(e, phi\_n) != 1:

e += 2 # Increment to find next odd number

# Calculate the private key d

d = modinv(e, phi\_n)

return (e, n), (d, n)

# RSA Encryption

def encrypt(message, public\_key):

e, n = public\_key

return [pow(ord(char), e, n) for char in message]

# RSA Decryption

def decrypt(ciphertext, private\_key):

d, n = private\_key

return ''.join([chr(pow(char, d, n)) for char in ciphertext])

# Main Function

def main():

# Step 1: Generate RSA keys

public\_key, private\_key = generate\_keys()

print("Public Key:", public\_key)

print("Private Key:", private\_key)

# Step 2: Encrypt a message

message = "HELLO"

print("\nOriginal Message:", message)

encrypted\_message = encrypt(message, public\_key)

print("Encrypted Message:", encrypted\_message)

# Step 3: Decrypt the message

decrypted\_message = decrypt(encrypted\_message, private\_key)

print("Decrypted Message:", decrypted\_message)

# Step 4: Demonstrate the danger of leaked private key

print("\nLeaking Private Key...")

print("An attacker can use the private key to decrypt the message:", decrypt(encrypted\_message, private\_key))

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

main()

27.# Simple RSA implementation for educational purposes

# Encrypts each letter as an integer (0 to 25) individually

# Function to compute modular inverse of a number a modulo m

def mod\_inverse(a, m):

m0, x0, x1 = m, 0, 1

if m == 1:

return 0

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

if x1 < 0:

x1 += m0

return x1

# RSA Encryption

def rsa\_encrypt(plain, e, n):

return [pow(p, e, n) for p in plain]

# RSA Decryption

def rsa\_decrypt(cipher, d, n):

return [pow(c, d, n) for c in cipher]

# Convert a string message to a list of integers (A=0, B=1, ..., Z=25)

def string\_to\_int(message):

return [ord(c) - ord('A') for c in message.upper()]

# Convert a list of integers back to a string (0=A, 1=B, ..., 25=Z)

def int\_to\_string(int\_list):

return ''.join([chr(i + ord('A')) for i in int\_list])

# Main function to simulate RSA

def main():

# RSA Parameters (for simplicity, using small primes)

p = 61 # Example prime

q = 53 # Example prime

n = p \* q # Modulus n

phi\_n = (p - 1) \* (q - 1) # Euler's totient function

# Public exponent (must be coprime with phi(n))

e = 31 # Example public exponent (large in practice)

# Private exponent (d such that e \* d ≡ 1 (mod phi\_n))

d = mod\_inverse(e, phi\_n)

print(f"Public Key: (e = {e}, n = {n})")

print(f"Private Key: (d = {d}, n = {n})")

# Alice's message (message consists of alphabetic characters)

message = "HELLO" # Example message

# Convert message to integers (A=0, B=1, ..., Z=25)

message\_int = string\_to\_int(message)

print(f"Message as integers: {message\_int}")

# Encrypt the message using RSA

encrypted\_message = rsa\_encrypt(message\_int, e, n)

print(f"Encrypted message: {encrypted\_message}")

# Decrypt the message using RSA

decrypted\_message = rsa\_decrypt(encrypted\_message, d, n)

print(f"Decrypted message (as integers): {decrypted\_message}")

# Convert the decrypted integers back to a string

decrypted\_string = int\_to\_string(decrypted\_message)

print(f"Decrypted message (as string): {decrypted\_string}")

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

main()

28.

import random

# Function to calculate (base^exponent) % modulus

def mod\_exp(base, exponent, modulus):

return pow(base, exponent, modulus)

# Function to perform Diffie-Hellman key exchange

def diffie\_hellman():

# Public parameters

q = 23 # A prime number

a = 5 # A primitive root modulo q

# Alice's private key (secret number)

x\_A = random.randint(1, q-1)

A = mod\_exp(a, x\_A, q) # Alice's public value

# Bob's private key (secret number)

x\_B = random.randint(1, q-1)

B = mod\_exp(a, x\_B, q) # Bob's public value

print(f"Public parameters: a = {a}, q = {q}")

print(f"Alice's public value (A) = {A}")

print(f"Bob's public value (B) = {B}")

# Alice and Bob exchange public values A and B

# Alice computes the shared key

K\_A = mod\_exp(B, x\_A, q)

# Bob computes the shared key

K\_B = mod\_exp(A, x\_B, q)

print(f"Alice's computed shared key (K\_A) = {K\_A}")

print(f"Bob's computed shared key (K\_B) = {K\_B}")

# Check if both keys match

if K\_A == K\_B:

print("Shared key agreed: ", K\_A)

else:

print("Key mismatch!")

# Run the Diffie-Hellman protocol

diffie\_hellman()

29.

import random

def simulate\_sha3\_capacity\_lanes():

# Initializing the state with 25 lanes (5x5) each with 0s

state = [[0] \* 5 for \_ in range(5)]

# Simulating until all capacity lanes (positions 0, 1, 2, 3, 4) have at least one non-zero bit

capacity\_lanes = [(i, j) for i in range(5) for j in range(5) if i == 0 or i == 1] # 0th and 1st rows

steps = 0

while True:

steps += 1

# Randomly flip a bit in the state

i, j = random.choice(capacity\_lanes)

state[i][j] = 1 # Set the bit to 1

# Check if all capacity lanes have at least one non-zero bit

if all(state[i][j] == 1 for i, j in capacity\_lanes):

break

return steps

# Simulating the process

steps\_taken = simulate\_sha3\_capacity\_lanes()

print(f"It took {steps\_taken} steps to make all capacity lanes non-zero.")

31.

31.

def generate\_cmac\_subkeys(block\_size):

# Define constants for block sizes

if block\_size == 64:

Rb = 0x1B # The constant for 64-bit block size

elif block\_size == 128:

Rb = 0x87 # The constant for 128-bit block size

else:

raise ValueError("Block size must be 64 or 128 bits.")

# Initial block of all zero bits

zero\_block = 0x00

cipher\_block = zero\_block # Simulating the encryption step

# Convert cipher\_block to binary representation with correct block size

cipher\_block\_bin = bin(cipher\_block)[2:].zfill(block\_size)

# Left shift the binary representation by 1 bit

subkey1\_bin = cipher\_block\_bin[1:] + '0'

# Convert back to integer

subkey1 = int(subkey1\_bin, 2)

# Conditionally XOR with the constant Rb if MSB of the original block is 1

if cipher\_block\_bin[0] == '1':

subkey1 ^= Rb

# Derive subkey2 by left-shifting subkey1 and conditionally XORing with Rb

subkey1\_bin = bin(subkey1)[2:].zfill(block\_size)

subkey2\_bin = subkey1\_bin[1:] + '0'

subkey2 = int(subkey2\_bin, 2)

if subkey1\_bin[0] == '1':

subkey2 ^= Rb

return subkey1, subkey2

# User Input for block size

block\_size = int(input("Enter the block size (64 or 128 bits): "))

try:

subkey1, subkey2 = generate\_cmac\_subkeys(block\_size)

print(f"For block size {block\_size} bits:")

print(f" Subkey 1: {bin(subkey1)[2:].zfill(block\_size)}")

print(f" Subkey 2: {bin(subkey2)[2:].zfill(block\_size)}")

except ValueError as e:

print(e)

34.from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default\_backend

import os

def pad\_custom(data, block\_size):

"""Custom padding: Add a 1 bit and fill the rest with 0 bits."""

padding\_length = block\_size - (len(data) % block\_size)

padding = b'\x80' + b'\x00' \* (padding\_length - 1) # 1 bit followed by 0 bits

return data + padding

def unpad\_custom(data, block\_size):

"""Remove custom padding."""

# Strip off the padding bytes

return data.rstrip(b'\x00').rstrip(b'\x80')

def ecb\_mode(key, plaintext):

"""Encrypt and decrypt using ECB mode."""

block\_size = algorithms.AES.block\_size // 8

padded\_plaintext = pad\_custom(plaintext, block\_size)

cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, unpad\_custom(decrypted\_data, block\_size)

def cbc\_mode(key, plaintext, iv):

"""Encrypt and decrypt using CBC mode."""

block\_size = algorithms.AES.block\_size // 8

padded\_plaintext = pad\_custom(plaintext, block\_size)

cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, unpad\_custom(decrypted\_data, block\_size)

def cfb\_mode(key, plaintext, iv):

"""Encrypt and decrypt using CFB mode."""

cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()

return ciphertext, decrypted\_data

# Testing the implementation

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

# Use a 128-bit (16-byte) key and IV

key = b"1234567890abcdef" # Fixed key for demonstration

iv = b"abcdef1234567890" # Fixed IV for CBC and CFB

plaintext = b"HELLO WORLD!!!" # Example plaintext (multiple of block size)

print("Original plaintext:", plaintext)

# ECB Mode

ecb\_ciphertext, ecb\_decrypted = ecb\_mode(key, plaintext)

print("\nECB Mode:")

print("Ciphertext (hex):", ecb\_ciphertext.hex())

print("Decrypted:", ecb\_decrypted)

# CBC Mode

cbc\_ciphertext, cbc\_decrypted = cbc\_mode(key, plaintext, iv)

print("\nCBC Mode:")

print("Ciphertext (hex):", cbc\_ciphertext.hex())

print("Decrypted:", cbc\_decrypted)

# CFB Mode

cfb\_ciphertext, cfb\_decrypted = cfb\_mode(key, plaintext, iv)

print("\nCFB Mode:")

print("Ciphertext (hex):", cfb\_ciphertext.hex())

print("Decrypted:", cfb\_decrypted)

35.

import random

import string

# Function to generate a random key of the same length as the plaintext

def generate\_key(plaintext):

# Generate a random key with numbers between 0 and 25 (corresponding to A-Z)

return [random.randint(0, 25) for \_ in range(len(plaintext))]

# Function to encrypt the plaintext using a one-time pad Vigenère cipher

def encrypt(plaintext, key):

ciphertext = []

for i in range(len(plaintext)):

# Only encrypt alphabetic characters

if plaintext[i].isalpha():

# Determine shift based on key and current character

shift = key[i]

# Encrypt character by shifting it

char = plaintext[i].upper()

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

ciphertext.append(encrypted\_char)

else:

# Non-alphabet characters are added to the ciphertext without change

ciphertext.append(plaintext[i])

return ''.join(ciphertext)

# Function to decrypt the ciphertext using the same key

def decrypt(ciphertext, key):

plaintext = []

for i in range(len(ciphertext)):

# Only decrypt alphabetic characters

if ciphertext[i].isalpha():

# Determine shift based on key and current character

shift = key[i]

# Decrypt character by reversing the shift

char = ciphertext[i].upper()

decrypted\_char = chr((ord(char) - ord('A') - shift) % 26 + ord('A'))

plaintext.append(decrypted\_char)

else:

# Non-alphabet characters are added to the plaintext without change

plaintext.append(ciphertext[i])

return ''.join(plaintext)

# Example usage

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

plaintext = input("Enter the plaintext: ")

# Generate a random key for the given plaintext

key = generate\_key(plaintext)

print(f"Generated key: {key}")

# Encrypt the plaintext

encrypted\_text = encrypt(plaintext, key)

print(f"Encrypted text: {encrypted\_text}")

# Decrypt the ciphertext

decrypted\_text = decrypt(encrypted\_text, key)

print(f"Decrypted text: {decrypted\_text}")

36.

# Function to calculate the greatest common divisor (GCD)

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Function to find modular inverse of a under modulo m

def mod\_inverse(a, m):

for i in range(1, m):

if (a \* i) % m == 1:

return i

return None

# Affine cipher encryption function

def affine\_encrypt(plaintext, a, b):

ciphertext = ''

for char in plaintext:

if char.isalpha(): # Only encrypt alphabetic characters

# Convert to uppercase and map to 0-25

p = ord(char.upper()) - ord('A')

# Apply affine cipher formula: (a \* p + b) % 26

c = (a \* p + b) % 26

ciphertext += chr(c + ord('A')) # Convert back to letter

else:

ciphertext += char # Non-alphabetic characters remain the same

return ciphertext

# Affine cipher decryption function

def affine\_decrypt(ciphertext, a, b):

plaintext = ''

# Find modular inverse of a

a\_inv = mod\_inverse(a, 26)

if a\_inv is None:

return "Error: a has no modular inverse under 26, decryption impossible!"

for char in ciphertext:

if char.isalpha(): # Only decrypt alphabetic characters

c = ord(char.upper()) - ord('A')

# Apply decryption formula: a\_inv \* (c - b) % 26

p = (a\_inv \* (c - b)) % 26

plaintext += chr(p + ord('A')) # Convert back to letter

else:

plaintext += char # Non-alphabetic characters remain the same

return plaintext

# Main function to test the Affine Cipher

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

# Get the values of a and b (a must be coprime with 26)

a = int(input("Enter the value of 'a' (must be coprime with 26): "))

if gcd(a, 26) != 1:

print("Error: 'a' must be coprime with 26. Exiting...")

exit()

b = int(input("Enter the value of 'b': "))

# Get the plaintext from the user

plaintext = input("Enter the plaintext (uppercase only, no spaces): ")

# Encrypt the plaintext

ciphertext = affine\_encrypt(plaintext, a, b)

print(f"Ciphertext: {ciphertext}")

# Decrypt the ciphertext

decrypted\_text = affine\_decrypt(ciphertext, a, b)

print(f"Decrypted text: {decrypted\_text}")

37.

from collections import Counter

# English letter frequencies in percentages

ENGLISH\_FREQ = {

'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702, 'f': 2.228,

'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153, 'k': 0.772, 'l': 4.025,

'm': 2.406, 'n': 6.749, 'o': 7.507, 'p': 1.929, 'q': 0.095, 'r': 5.987,

's': 6.327, 't': 9.056, 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150,

'y': 1.974, 'z': 0.074

}

def shift\_decrypt(ciphertext, key):

"""Decrypt using a Caesar shift with a given key."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

shifted = (ord(char.lower()) - ord('a') - key) % 26 + ord('a')

plaintext += chr(shifted).upper() if char.isupper() else chr(shifted)

else:

plaintext += char

return plaintext

def calculate\_score(text):

"""Score the text based on English letter frequencies."""

text = text.lower()

letter\_count = Counter(char for char in text if char.isalpha())

total\_letters = sum(letter\_count.values())

if total\_letters == 0:

return float('inf') # Penalize empty plaintexts

score = 0

for letter, expected\_freq in ENGLISH\_FREQ.items():

observed\_freq = (letter\_count[letter] / total\_letters) \* 100

score += abs(observed\_freq - expected\_freq)

return score

def frequency\_attack(ciphertext, top\_n=10):

"""Perform a frequency attack on the monoalphabetic cipher."""

results = []

for key in range(26):

plaintext = shift\_decrypt(ciphertext, key)

score = calculate\_score(plaintext)

results.append((key, plaintext, score))

results.sort(key=lambda x: x[2]) # Sort by score (lower is better)

return results[:top\_n]

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

# User input

ciphertext = input("Enter the ciphertext: ")

top\_n = int(input("Enter the number of top possible plaintexts to display: "))

# Perform frequency analysis

results = frequency\_attack(ciphertext, top\_n)

# Output results

print("\nTop possible plaintexts:")

for rank, (key, plaintext, score) in enumerate(results, start=1):

print(f"{rank}. Key = {key}, Score = {score:.2f}, Plaintext = {plaintext}")

39..from collections import Counter

# Expected English letter frequencies (normalized)

ENGLISH\_FREQ = {

'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702, 'f': 2.228,

'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153, 'k': 0.772, 'l': 4.025,

'm': 2.406, 'n': 6.749, 'o': 7.507, 'p': 1.929, 'q': 0.095, 'r': 5.987,

's': 6.327, 't': 9.056, 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150,

'y': 1.974, 'z': 0.074

}

def additive\_decrypt(ciphertext, key):

"""Decrypt ciphertext using the additive cipher and key."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

# Shift for lowercase and uppercase separately

shifted = (ord(char.lower()) - ord('a') - key) % 26 + ord('a')

# Preserve the case (upper or lower)

plaintext += chr(shifted).upper() if char.isupper() else chr(shifted)

else:

plaintext += char

return plaintext

def calculate\_score(text):

"""Calculate a score for the text based on English letter frequency."""

text = text.lower()

letter\_count = Counter(char for char in text if char.isalpha())

total\_letters = sum(letter\_count.values())

if total\_letters == 0:

return float('inf') # No letters to compare, return a large score

# Compare text frequency with English frequency

score = 0

for letter, expected\_freq in ENGLISH\_FREQ.items():

observed\_freq = (letter\_count[letter] / total\_letters) \* 100

score += abs(observed\_freq - expected\_freq)

return score

def frequency\_attack(ciphertext, top\_n=10):

"""Perform a frequency attack on the additive cipher."""

results = []

for key in range(26): # Try all possible keys (0-25)

plaintext = additive\_decrypt(ciphertext, key)

score = calculate\_score(plaintext)

results.append((key, plaintext, score))

# Sort results by score (lower is better)

results.sort(key=lambda x: x[2])

return results[:top\_n]

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

# Getting inputs outside the code

ciphertext = input("Enter the ciphertext: ")

top\_n = int(input("Enter the number of top possible plaintexts to display: "))

# Perform the attack

results = frequency\_attack(ciphertext, top\_n)

# Display results

print("\nTop possible plaintexts:")

for rank, (key, plaintext, score) in enumerate(results, start=1):

print(f"{rank}. Key = {key}, Score = {score:.2f}, Plaintext = {plaintext}")

40.from collections import Counter

# English letter frequencies in percentages

ENGLISH\_FREQ = {

'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702, 'f': 2.228,

'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153, 'k': 0.772, 'l': 4.025,

'm': 2.406, 'n': 6.749, 'o': 7.507, 'p': 1.929, 'q': 0.095, 'r': 5.987,

's': 6.327, 't': 9.056, 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150,

'y': 1.974, 'z': 0.074

}

def shift\_decrypt(ciphertext, key):

"""Decrypt using a Caesar shift with a given key."""

plaintext = ""

for char in ciphertext:

if char.isalpha():

shifted = (ord(char.lower()) - ord('a') - key) % 26 + ord('a')

plaintext += chr(shifted).upper() if char.isupper() else chr(shifted)

else:

plaintext += char

return plaintext

def calculate\_score(text):

"""Score the text based on English letter frequencies."""

text = text.lower()

letter\_count = Counter(char for char in text if char.isalpha())

total\_letters = sum(letter\_count.values())

if total\_letters == 0:

return float('inf') # Penalize empty plaintexts

score = 0

for letter, expected\_freq in ENGLISH\_FREQ.items():

observed\_freq = (letter\_count[letter] / total\_letters) \* 100

score += abs(observed\_freq - expected\_freq)

return score

def frequency\_attack(ciphertext, top\_n=10):

"""Perform a frequency attack on the monoalphabetic cipher."""

results = []

for key in range(26):

plaintext = shift\_decrypt(ciphertext, key)

score = calculate\_score(plaintext)

results.append((key, plaintext, score))

results.sort(key=lambda x: x[2]) # Sort by score (lower is better)

return results[:top\_n]

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

# User input

ciphertext = input("Enter the ciphertext: ")

top\_n = int(input("Enter the number of top possible plaintexts to display: "))

# Perform frequency analysis

results = frequency\_attack(ciphertext, top\_n)

# Output results

print("\nTop possible plaintexts:")

for rank, (key, plaintext, score) in enumerate(results, start=1):

print(f"{rank}. Key = {key}, Score = {score:.2f}, Plaintext = {plaintext}")