**PANDIT DEENDAYAL ENERGY UNIVERSITY**

**SCHOOL OF TECHNOLOGY**

****

**Course: Information Security**

**Course Code: 20CP304P**

**LAB MANUAL**

**B.Tech. (Computer Engineering)**

**Semester 5**

|  |  |
| --- | --- |
| **Submitted To:** | **Submitted By:** |
| Dr. Utkarsh Tiwari | Dev Mehta |
|  | 22BCP282 |
|  | G8 batch |

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **List of experiments** | **Date** | **Sign** |
| **1** | Study and Implement a program for Caesar Cipher | 25/07/24 |  |
| **2** | Study and Implement a program for 5x5 Playfair Cipher to encrypt and decrypt the message. | 01/08/24 |  |
| **3** | Study and Implement a program for Rail Fence Cipher with columnar transposition | 22/08/24 |  |
| **4** | Study and implement a program for columnar Transposition Cipher | 29/08/24 |  |
| **5** | Study and implement a program for Vigenère Cipher | 05/09/24 |  |
| **6** | Study and Implement a program for n-gram Hill Cipher | 12/09/24 |  |
| **7** | Study and Use of RSA algorithm (encryption and decryption) | 26/09/24 |  |
| **8** | Study and implement a program of the Digital Signature with RSA algorithm (Reverse RSA) | 03/10/24 |  |
| **9** | Study and Use of Diffie-Hellman Key Exchange | 10/10/24 |  |
| **10** | Design cipher with detailed explanation. Sample as follows.   1. Write a program to encrypt the plaintext with the given key. E.g. plaintext GRONSFELD with the key 1234. Add 1 to G to get H (the letter 1 rank after G is H in the alphabet), then add 2 to C or E (the letter 2 ranks after C is E), and so on. Use smallest letter from plaintext as filler. 2. Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD | 17/10/24 |  |

**Experiment 1**

**AIM:** Study and Implement a program for Caesar Cipher

**Introduction :** In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code, or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on. The method is named after Julius Caesar, who used it in his private correspondence.

**Example :** The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places, equivalent to a right shift of 23 (the shift parameter is used as the [key](https://en.wikipedia.org/wiki/Key_(cryptography))):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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** | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line.

Plaintext: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Deciphering is done in reverse, with a right shift of 3.

The encryption can also be represented using [modular arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic) by first transforming the letters into numbers, according to the scheme, A → 0, B → 1, ..., Z → 25.

**Source Code:**

plainText = input("Enter the text:")

key = 5

uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

lowercase = "abcdefghijklmnopqrstuvwxyz"

def encrypt(plaintext,key):

cipherText = ""

for i in plaintext:

if i in uppercase:

cipherText += uppercase[(uppercase.index(i) + key) % 26]

elif i in lowercase:

cipherText += lowercase[(lowercase.index(i) + key) % 26]

else:

cipherText += i

return cipherText

cipherText = encrypt(plainText,key)

def decrypt(ciphertext,key):

decryptedText = ""

for i in ciphertext:

if i in uppercase:

decryptedText += uppercase[(uppercase.index(i) - key) % 26]

elif i in lowercase:

decryptedText += lowercase[(lowercase.index(i) - key) % 26]

else:

decryptedText += i

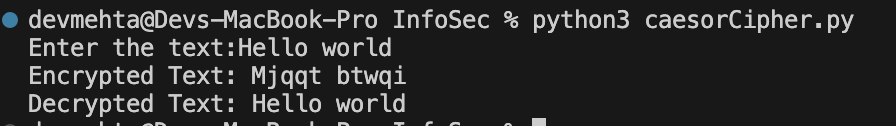
return decryptedText

decryptedText = decrypt(cipherText,key)

print("Encrypted Text:",cipherText)

print("Decrypted Text:",decryptedText)

**Output screenshot:**



**Revised Approach**

**Introduction:**

Advancement in Caesar cipher using the approach of randomization making the cipher more secure and tough to decrypt by the hacker or cryptanalyst. By adding this algorithm, it’s not easy for attacker to crack the cipher because the character replaced is randomly generated. Brute force attacker will also not be able to crack it because the characters are replaced by the other character according to randomization approach from key table.

There is a randomly organised key table containing uppercase, lowercase, blank spaces, special character etc which is used in these approach.

**NO. OF ENCRYPTION LETTERS FOR SINGLE ALPHABET OF PLAIN TEXT: KEY\*KEY**

**ENCRYPTION:**

Ei = x +(i\*K)

X -> current position of alphabet

(i=1 to i=key\*key)

**DECRYPTION:**

Using a loop where the character whose number%(Key\*key)==0,

Shift the number of that character key times left from the current position.

**Example:**

PLAIN TEXT: BAD

KEY : 2

NO. OF ENCRYPTION LETTERS FOR SINGLE ALPHABET OF PLAIN TEXT: 2\*2 = 4

Ei = x +(i\*K)

(i=1 to i=key\*key)

B -> Pris

A->DJ%C

D->GLZB

Encrypted Text: PrisDJ%CGLZB

Decrypted Text: BAD

**Source Code:**

plainText = input("Enter the plainText:")

keyTable = [

'W', 'Q', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'M', 'A', 'S', 'D', 'F', 'G', 'X', 'J', 'K', 'L',"$","%","&", 'Z', 'H', 'C', 'V', 'B', 'N', 'P',

'w', 'q', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'm', 'a', 's', 'd', 'f', 'g', 'x', 'j', 'k', 'l', 'z', 'h', 'c', 'v', 'b', 'n', 'p'," ","!","@","#"

]

key = 3

def encrypt(plaintext,key):

ciphertext = ""

for i in plaintext:

if i in keyTable:

index = keyTable.index(i)

for j in range(1,key\*key+1):

ciphertext+=keyTable[(index+j\*key)%len(keyTable)]

index+=key

else:

ciphertext+=i

return ciphertext

def decrypt(ciphertext,key):

plaintext=""

i=0

while(i<len(ciphertext)):

if ciphertext[i] in keyTable:

if i%(key\*key) == 0:

value = ciphertext[i]

plaintext+=keyTable[(keyTable.index(value)-key)%len(keyTable)]

else:

plaintext+=ciphertext[i]

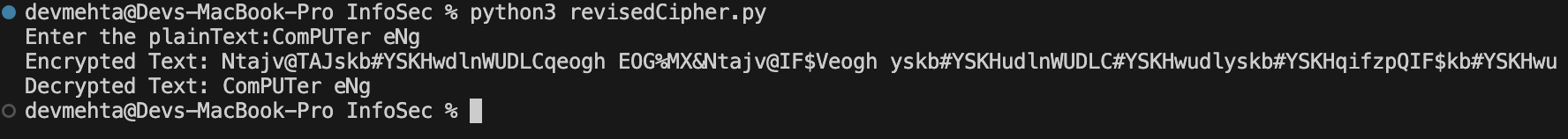
print(plaintext)

i+=1

return plaintext

print("Encrypted Text:", encrypt(plainText,key))

print("Decrypted Text:", decrypt(encrypt(plainText,key),key))

**Output screenshot:**

**Comparative Analysis of original and revised approaches:**

* **Flaw in original approach:**

In Original approach the major flaw was if the key was somehow exposed to external source, the attacker can easily find the pattern and crack the code by brute force method.

**Revised Approach’s betterment:**

Even if the key is exposed to the attacker, they cannot predict the pattern easily and it will take too long for brute force to crack the code as here each letter of plain text is encrypted by key\*key characters and that too not only alphabets but numbers, blank spaces and special characters are also included.

* **Flaw in original approach:**

The number of keys are limited to 26 as there is fixed table of 26 alphabets from where the encrypted text is generated.

**Revised approach’s betterment:**

Here there is no such fixed size limitation, as the key table is randomly organized and is mixture of multiple literals.

**Conclusion:**

Cryptography plays an important role for safe transmission of data. Data is encrypted and decrypted by many techniques. Caesar cipher is important technique which has less complex methodology for encryption. Many advancements are done in Caesar cipher to make it more secure. This **approach of randomization** can also be considered as a modified cipher or **advanced Caesar cipher**.

**References:**

[A. Rajan and D. Balakumaran, "Advancement in caesar cipher by randomization and delta formation," International Conference on Information Communication and Embedded Systems (ICICES2014), Chennai, India, 2014, pp. 1-4, doi: 10.1109/ICICES.2014.7033998. keywords: {Ciphers;Encryption;Educational institutions;Arrays;Computers;Computational complexity},](https://ieeexplore.ieee.org/document/7033998)

**Experiment 2**

**AIM:** Study and Implement a program for 5x5 Playfair Cipher to encrypt and decrypt the message.

**Introduction :** The Playfair cipher or Playfair square or Wheatstone–Playfair cipher is a manual [symmetric](https://en.wikipedia.org/wiki/Symmetric_key_algorithm) [encryption](https://en.wikipedia.org/wiki/Encryption) technique and was the first literal [digram substitution](https://en.wikipedia.org/wiki/Polygraphic_substitution" \o "Polygraphic substitution) cipher. The scheme was invented in 1854 by [Charles Wheatstone](https://en.wikipedia.org/wiki/Charles_Wheatstone), but bears the name of [Lord Playfair](https://en.wikipedia.org/wiki/Lord_Playfair) for promoting its use.

The technique encrypts pairs of letters ([*bigrams*](https://en.wikipedia.org/wiki/Bigram) or *digrams*), instead of single letters as in the simple [substitution cipher](https://en.wikipedia.org/wiki/Substitution_cipher) and rather more complex [Vigenère cipher](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher) systems then in use. The Playfair cipher is thus significantly harder to break since the [frequency analysis](https://en.wikipedia.org/wiki/Frequency_analysis) used for simple substitution ciphers does not work with it. The frequency analysis of bigrams is possible, but considerably more difficult.

**Example :** For the encryption process let us consider the following example:



**The Playfair Cipher Encryption Algorithm:**  
The Algorithm consists of 2 steps: 

1. **Generate the key Square(5×5):**
   * The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
   * The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
2. **Algorithm to encrypt the plain text:** The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.   
   **For example:**

**PlainText**: "instruments"   
**After Split:** 'in' 'st' 'ru' 'me' 'nt' 'sz

**Plain Text:** "instrumentsz"  
**Encrypted Text:** gatlmzclrqtx

𝐸𝑛(𝑥)=(𝑥+𝑛)mod26.

**Original Approach**

**Source Code:**

plain\_text = input("Enter the Plain text:")

key = "KEYWORD"

alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"

space\_list=[]

j\_list = []

double\_list = []

Matrix = []

for k in key:

if k not in Matrix:

Matrix.append(k)

for letter in alphabet:

if letter not in Matrix:

Matrix.append(letter)

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

print(playfair\_matrix)

def find\_position(matrix, char):

for row in range(5):

for col in range(5):

if matrix[row][col] == char:

return row, col

return None

def encrypt(plain\_text, matrix):

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

if (plain\_text[i] == "J"):

j\_list.append(i)

if (plain\_text[i] == " "):

space\_list.append(i)

for i in range(len(plain\_text)-1):

if (plain\_text[i]==plain\_text[i+1]):

double\_list.append(i)

plain\_text = plain\_text[:i] + "X" + plain\_text[i+1:]

if (plain\_text[i]=="X" and plain\_text[i+1]== "X"):

double\_list.append(i)

plain\_text = plain\_text[:i] + "Z" + plain\_text[i+1:]

plain\_text = plain\_text.upper().replace("J", "I")

plain\_text = plain\_text.upper().replace(" ", "")

if len(plain\_text) % 2 != 0:

plain\_text += "Z"

cipher\_text = ""

print(j\_list)

print(space\_list)

i = 0

while i < len(plain\_text):

a = plain\_text[i]

b = plain\_text[i + 1]

row1, col1 = find\_position(matrix, a)

row2, col2 = find\_position(matrix, b)

if row1 == row2:

cipher\_text += matrix[row1][(col1 + 1) % 5]

cipher\_text += matrix[row2][(col2 + 1) % 5]

elif col1 == col2:

cipher\_text += matrix[(row1 + 1) % 5][col1]

cipher\_text += matrix[(row2 + 1) % 5][col2]

else:

cipher\_text += matrix[row1][col2]

cipher\_text += matrix[row2][col1]

i += 2

return cipher\_text

def decrypt(ciphertext,matrix):

plaintext=""

i=0

while i<len(ciphertext):

a = ciphertext[i]

b = ciphertext[i+1]

row1,col1 = find\_position(matrix,a)

row2,col2 = find\_position(matrix,b)

if row1 == row2:

plaintext+=matrix[row1][(col1-1)%5]

plaintext+=matrix[row2][(col2-1)%5]

elif col1 == col2:

plaintext+=matrix[(row1-1)%5][col1]

plaintext+=matrix[(row2-1)%5][col2]

else:

plaintext+=matrix[row1][col2]

plaintext+=matrix[row2][col1]

i+=2

for index in space\_list:

plaintext = plaintext[:index] + " " + plaintext[index:]

for index in j\_list:

plaintext = plaintext[:index] + "J" + plaintext[index + 1:]

for index in double\_list:

plaintext = plaintext[:index] + plaintext[index+1] + plaintext[index + 1:]

if (plain\_text[-1]!= plaintext[-1]):

plaintext = plaintext[:-1]

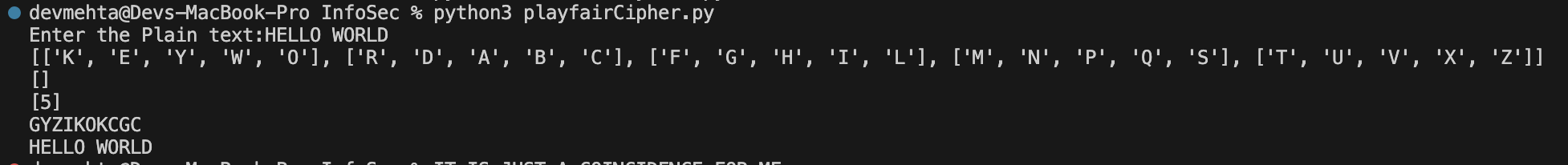
return plaintext

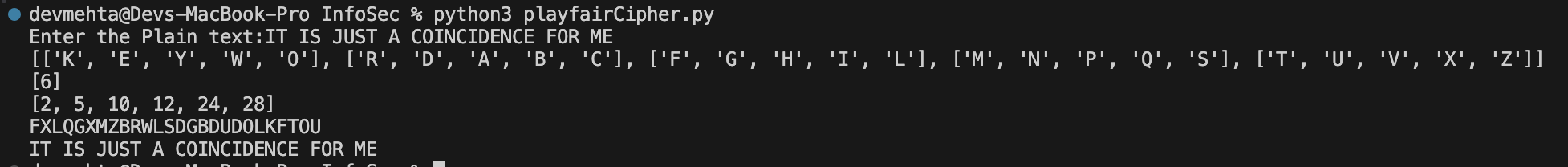
cipher\_text=encrypt(plain\_text,playfair\_matrix)

print(cipher\_text)

print(decrypt(cipher\_text,playfair\_matrix))

**Output screenshot:**



****

**Crypt Analysis:**

* Frequency Analysis:

1. Issue: Common digraphs (pairs of letters) in the plaintext create patterns in the ciphertext, making it vulnerable to frequency analysis.

Exploit: An attacker can match frequent digraphs in the ciphertext with common digraphs in the plaintext, gradually revealing the key.

* Known Plaintext Attack:

1. Issue: If an attacker knows a part of the plaintext, they can use it to deduce the positions of letters in the key square.

Exploit: By analyzing how specific plaintext digraphs are encrypted, the attacker can reconstruct parts of the key.

* Chosen Plaintext Attack:

1. Issue: If an attacker can choose the plaintext, they can systematically break the cipher by encrypting specific pairs of letters.

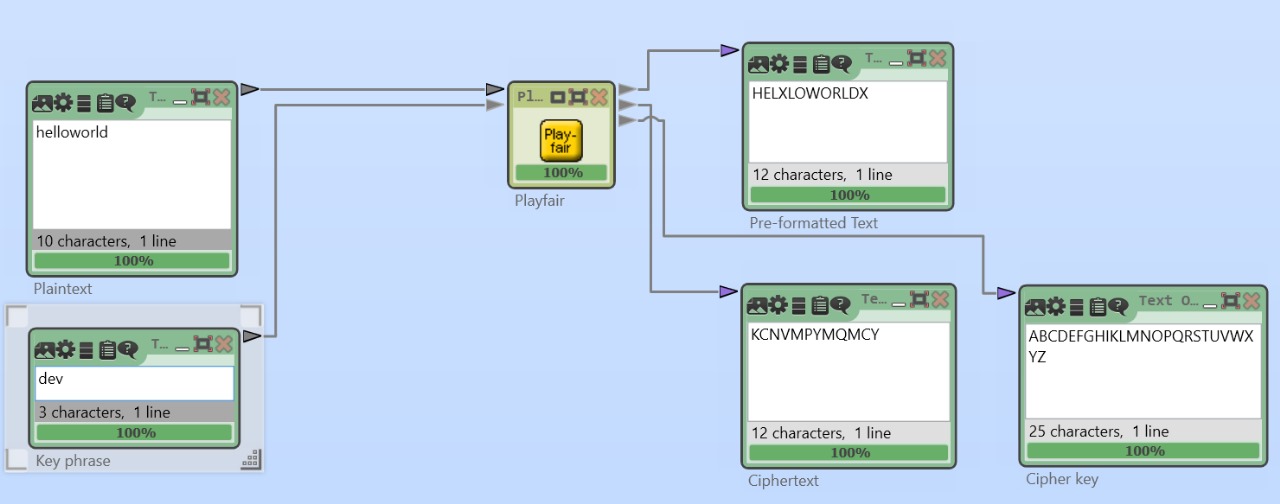
Exploit: Encrypting chosen digraphs reveals the structure of the key square.

* Limited Key Space:

1. Issue: The 5x5 matrix results in a limited number of possible key squares, which is vulnerable to brute-force attacks.

Exploit: With enough computational power, an attacker could try all possible key squares to decrypt the ciphertext.

**Crypt Tool output:**

****

**Revised Approach**

**Introduction:**

The algorithm is designed to secure text by encrypting them using a modified version of the Playfair cipher. The Playfair cipher is a classic encryption technique that usually works with a 5x5 grid of letters, but this algorithm uses a 8x8 grid (Playfair matrix) to accommodate more characters, including uppercase and lowercase letters, numbers, and some special characters.

**Encryption Process**

1. **Dividing the Text:**
   * The original text (plaintext) is split into sections or pairs of characters. If two characters in a pair are the same (like "aa"), a filler character (an extra, random character) is added between them to make them different (like "axa").
2. **Encrypting with Playfair Matrix:**
   * Each pair of characters is encrypted using the 8x8 Playfair matrix. This matrix is generated in a special way (called the "Octonary Quadrate Pattern") that differs from the traditional method, making it more secure.
   * If the row or column number is at even index it takes right shift for encrypting whereas if row or column is at odd index it takes left shift for encryption.
3. **Storing Filler Information:**
   * The number and positions of filler characters used during encryption are also encoded into the final encrypted message (ciphertext). This information is necessary for decrypting the message later.

**Decryption Process**

The decryption process is essentially the reverse of encryption:

1. **Decrypting Pairs**:
   * The pairs of characters are decrypted using the same 8x8 Playfair matrix that was used for encryption.
   * If the row or column number is at even index it takes left shift for decrypting whereas if row or column is at odd index it takes right shift for decryption.
2. **Removing Fillers**:
   * The filler characters that were added during encryption are removed based on the information stored in the ciphertext. This restores the original plaintext.

**Example:**

Encrypting and Decrypting "My name is Dev"

Step 1: Dividing the Text

* Plaintext: "My name is Dev"
* Split into pairs: My, na, me, is, De, vX (Note: A filler "X" is added to make the final pair)

Step 2: Encrypting with Playfair Matrix

* Assume positions for "M" and "y" in the matrix.
* Applying the modified Playfair rules:
  + "My" becomes "Ab"
  + "na" becomes "Bc"
  + "me" becomes "Cd"
  + "is" becomes "Ef"
  + "De" becomes "Gh"
  + "vX" becomes "Ij"
* Ciphertext: "AbBcCdEfGhIj"

Step 3: Decrypting the Ciphertext

* Ciphertext: "AbBcCdEfGhIj"
* Split into pairs: Ab, Bc, Cd, Ef, Gh, Ij
* Reverse the encryption process:
  + "Ab" decrypts to "My"
  + "Bc" decrypts to "na"
  + "Cd" decrypts to "me"
  + "Ef" decrypts to "is"
  + "Gh" decrypts to "De"
  + "Ij" decrypts to "vX"
* Decrypted Text: "My name is Dev"

**Source Code:**

import random

def generate\_playfair\_matrix(key):

all\_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789[]"

matrix = []

shuffled\_chars = list(all\_chars)

random.shuffle(shuffled\_chars)

for char in key:

if char not in matrix:

matrix.append(char)

for char in shuffled\_chars:

if char not in matrix:

matrix.append(char)

if len(matrix) == 64:

break

return [matrix[i:i+8] for i in range(0, 64, 8)]

def find\_position(char, matrix):

for row in range(8):

for col in range(8):

if matrix[row][col] == char:

return row, col

return None

def encrypt\_pair(pair, matrix):

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

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

if row1 == row2 and row1%2==0:

return matrix[row1][(col1 + 1) % 8] + matrix[row2][(col2 + 1)%8]

elif row1 == row2 and row1%2!=0:

return matrix[row1][(col1 - 1) % 8] + matrix[row2][(col2 - 1)%8]

elif col1 == col2 and col1%2==0:

return matrix[(row1 + 1) % 8][col1] + matrix[(row2 + 1)%8][col2]

elif col1 == col2 and col1%2!=0:

return matrix[(row1 - 1) % 8][col1] + matrix[(row2 - 1)%8][col2]

else:

return matrix[row1][col2] + matrix[row2][col1]

def decrypt\_pair(pair, matrix):

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

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

if row1 == row2 and row1%2==0:

return matrix[row1][(col1 - 1) % 8] + matrix[row2][(col2 - 1)%8]

elif row1 == row2 and row1%2!=0:

return matrix[row1][(col1 + 1) % 8] + matrix[row2][(col2 + 1)%8]

elif col1 == col2 and col1%2==0:

return matrix[(row1 - 1) % 8][col1] + matrix[(row2 - 1)%8][col2]

elif col1 == col2 and col1%2!=0:

return matrix[(row1 + 1) % 8][col1] + matrix[(row2 + 1)%8][col2]

else:

return matrix[row1][col2] + matrix[row2][col1]

def prepare\_text(text):

for i in text:

if i == ' ':

text = text.replace(' ', '')

prepared = ''

i = 0

while i < len(text):

if i == len(text) - 1:

prepared += text[i] + 'X'

i += 1

elif text[i] == text[i + 1]:

prepared += text[i] + 'X'

i += 1

else:

prepared += text[i] + text[i + 1]

i += 2

return prepared

def encrypt\_text(plaintext, matrix):

prepared\_text = prepare\_text(plaintext)

ciphertext = ''

for i in range(0, len(prepared\_text), 2):

ciphertext += encrypt\_pair(prepared\_text[i:i+2], matrix)

return ciphertext

def decrypt\_text(ciphertext, matrix):

decrypted\_text = ''

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

decrypted\_text += decrypt\_pair(ciphertext[i:i+2], matrix)

return decrypted\_text.replace('X', '')

key = "keyWordQQ"

matrix = generate\_playfair\_matrix(key)

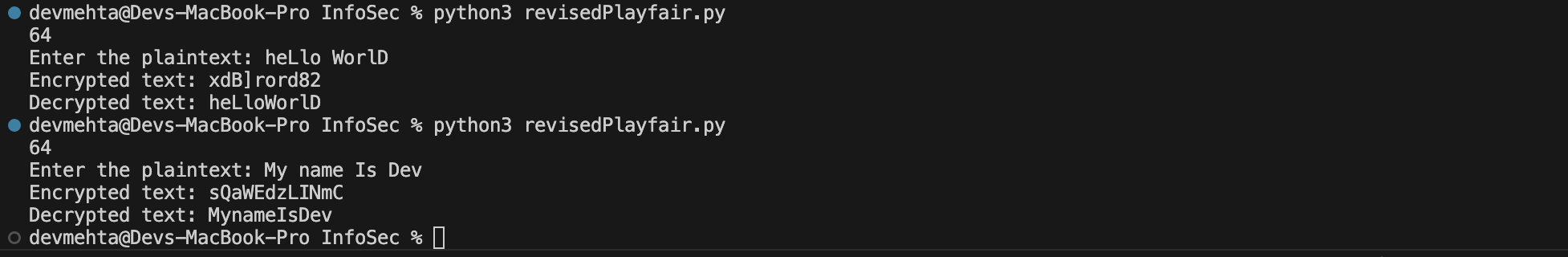
plaintext = input("Enter the plaintext: ")

ciphertext = encrypt\_text(plaintext, matrix)

print("Encrypted text:", ciphertext)

decrypted\_text = decrypt\_text(ciphertext, matrix)

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

**Output screenshot:**

**Comparative Analysis of original and revised approaches:**

* **Flaw in original approach:**
  + Uses a 5x5 matrix.
  + Covers only 26 uppercase English letters (I and J are usually combined).
  + Excludes digits, special characters, and lowercase letters.
  + Vulnerable to frequency analysis due to the predictability of the matrix
  + The key (keyword) is sensitive, but once known, the matrix can be easily reconstructed.

**Revised Approach’s betterment:**

* The modified cipher expands the character set, making it suitable for modern text, which includes a mix of uppercase, lowercase, numbers, and special characters.
* The use of a less predictable matrix generation method makes the cipher more secure and difficult to decrypt.
* The randomness in matrix generation and filler selection increases key sensitivity.

**Conclusion:**

The modified Playfair cipher with the Octonary Quadrate Pattern offers significant improvements over the original Playfair cipher, addressing its flaws by expanding the character set, enhancing matrix generation, introducing randomization in filler characters. These modifications increase the cipher's security, making it more resistant to modern cryptanalytic techniques while also making it more versatile and applicable to a wider range of text types.

**References:**

**Citations:**

[Sugirtham, N., Jenny, R.S., Thiyaneswaran, B. et al. Modified Playfair for Text File Encryption and Meticulous Decryption with Arbitrary Fillers by Septenary Quadrate Pattern. Int J Netw Distrib Comput 12, 108–118 (2024). https://doi.org/10.1007/s44227-02](https://doi.org/10.1007/s44227-023-00019-4)

**Experiment 3**

**AIM:** Study and Implement a program for Rail Fence Cipher with columnar transposition

**Introduction :** The rail fence cipher (also called a zigzag cipher) is a [classical](https://en.wikipedia.org/wiki/Classical_cipher) type of [transposition cipher](https://en.wikipedia.org/wiki/Transposition_cipher). It derives its name from the manner in which encryption is performed, in analogy to a fence built with horizontal rails.

**Example :** In the rail fence cipher, the plaintext is written downwards diagonally on successive "rails" of an imaginary fence, then moving up when the bottom rail is reached, down again when the top rail is reached, and so on until the whole plaintext is written out. The ciphertext is then read off in rows.

For example, to encrypt the message 'WE ARE DISCOVERED. RUN AT ONCE.' with 3 "rails", write the text as:

W . . . E . . . C . . . R . . . U . . . O . . .

. E . R . D . S . O . E . E . R . N . T . N . E

. . A . . . I . . . V . . . D . . . A . . . C .

(Note that spaces and punctuation are omitted.) Then read off the text horizontally to get the ciphertext:

WECRUO ERDSOEERNTNE AIVDAC

**Original Approach**

**Source Code:**

plaintext = input("Enter the plaintext:")

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

key = 3

def encrypt(plaintext,key):

matrix = [['\n'for i in range(len(plaintext))] for j in range(key)]

row,col = 0,0

flag = False

for i in plaintext:

if(row == 0) or (row == key-1):

flag = not flag

matrix[row][col] = i

col+=1

if flag:

row+=1

else:

row-=1

result = []

for i in range(key):

for j in range(len(plaintext)):

if(matrix[i][j] != '\n'):

result.append(matrix[i][j])

return ("".join(result))

def decrypt(ciphertext,key):

matrix = [['\n' for i in range(len(ciphertext))] for j in range(key)]

row,col = 0,0

flag = 0

for i in range(len(ciphertext)):

if row == 0:

flag = True

if row == key - 1:

flag = False

matrix[row][col] = '\*'

col += 1

if flag:

row += 1

else:

row -= 1

index = 0

for i in range(key):

for j in range(len(ciphertext)):

if ((matrix[i][j] == '\*') and

(index < len(ciphertext))):

matrix[i][j] = ciphertext[index]

index += 1

result = []

row, col = 0, 0

flag = False

for i in range(len(ciphertext)):

if row == 0:

flag = True

if row == key-1:

flag = False

if (matrix[row][col] != '\*'):

result.append(matrix[row][col])

col += 1

if flag:

row += 1

else:

row -= 1

return("".join(result))

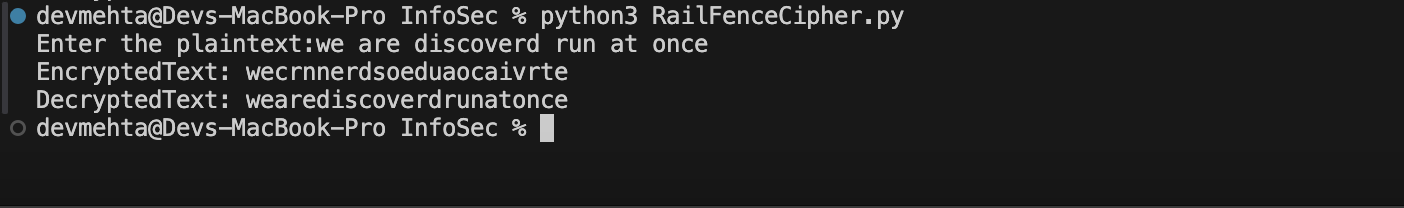
ciphertext = encrypt(plaintext,key)

decyptedtext = decrypt(ciphertext,key)

print("EncryptedText:",ciphertext)

print("DecryptedText:",decyptedtext)

**Output screenshot:**



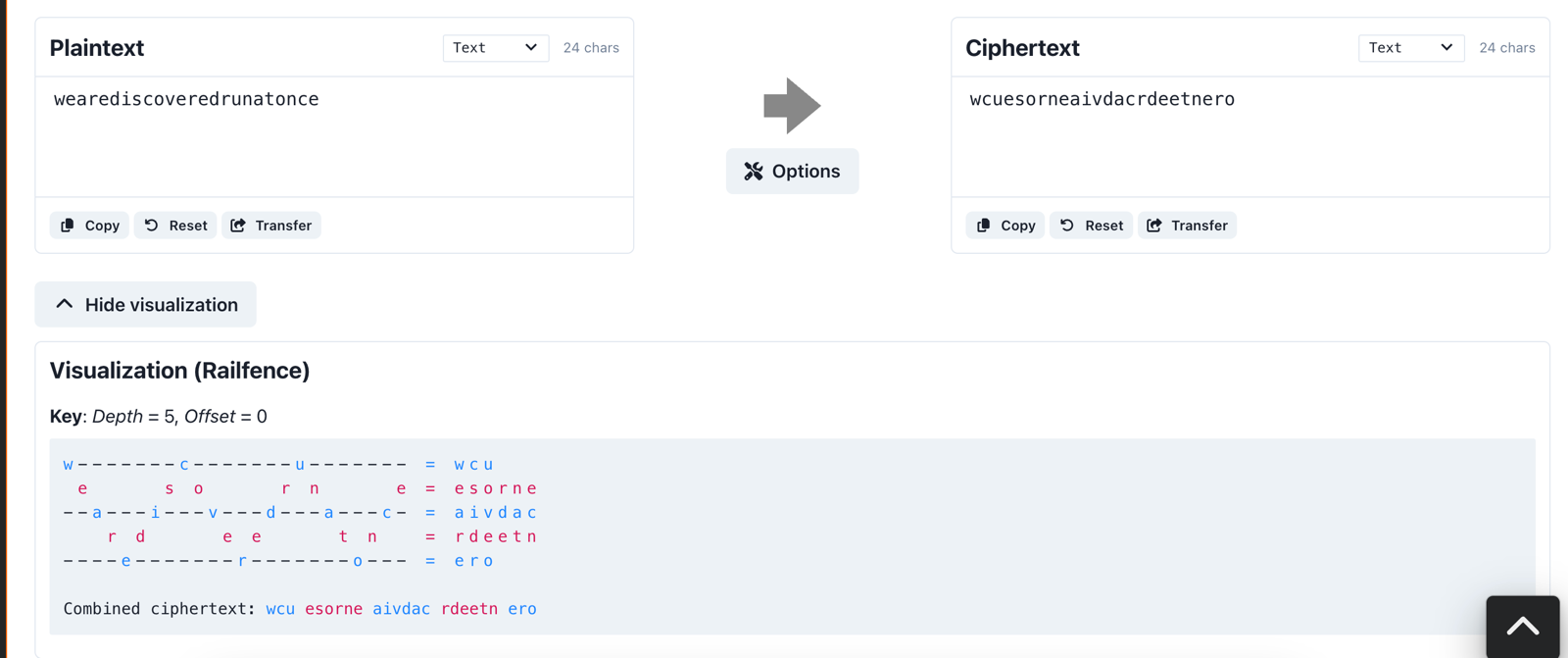
**Crypt Analysis:**

The original Rail Fence Cipher is a simple transposition cipher that rearranges the characters of the plaintext in a zigzag pattern across multiple rows. Its main vulnerabilities include:

1. Pattern Recognition: The regularity of the text makes it easy to recognize patterns and reconstruct the original message.
2. Brute-Force Attack: The limited key space (number of rows) allows for quick brute-force decryption by trying all possible keys.
3. Known-Plaintext Attack: If part of the plaintext is known, the cipher can be easily broken.
4. Frequency Analysis: Since letter frequencies remain unchanged, comparing them to expected language patterns can help reveal the key.

Overall, the Rail Fence Cipher is considered weak and easily breakable, making it unsuitable for secure communication.

**Crypt Tool output:**

****

**Revised Approach**

**Introduction:**

The algorithm is designed to secure text by encrypting them using a modified version of the Rail fence cipher. The Rail Fence cypher is a kind of transposition cypher where the plain text can be represented in a zig zag manner and then rearranged, but this algorithm uses map table where the text is not only rearranged by zig zag manner but also after that it is rearranged by using mathematical and ascii concepts.

**Encryption Process**

* The characters of the plaintext are arranged in a zigzag pattern across multiple rows determined by the key.
* The zigzag pattern is formed by moving downward and upward through the rows, filling the matrix.
* Once the matrix is filled, the ciphertext is created by reading the characters row by row.
* **Swapping Characters:**
* To add an extra layer of complexity, the first and last characters of the resulting string from the Rail Fence step are swapped.
* **Character Substitution Using a Mapping Table:**
* Each character in the modified ciphertext is then substituted using a predefined mapping table. The mapping is done using the formula:
* The position of a character in the alphabet is calculated as:

*j=ord(i.upper())−65*

**Decryption Process**

* **Reversing Character Substitution:**
* The first step is to reverse the character substitution. Each character in the ciphertext is converted back to its original character using a reverse mapping table:

*Original Character=chr(reversed\_map[i]+65).lower()*

* **Swapping Characters Back:**
* The first and last characters of the string are swapped back to their original positions.
* **Rail Fence Decryption:**
* The modified ciphertext is rearranged back into the zigzag pattern using the key. A matrix is created with placeholders, and then the characters are placed in their respective positions.
* The plaintext is reconstructed by reading the matrix in the original zigzag pattern.

**Example:**

Consider the plaintext "HELLO" with a key of 3:

1. Remove Spaces: "HELLO" (no spaces to remove).
2. **Rail Fence Encryption:**
   * Ciphertext (before swap): "HLOEL"
   * Ciphertext (after swap): "OLHEL"
3. Character Substitution:
   * 'O' -> y
   * 'L' -> b
   * 'H' -> r
   * 'E' -> y
   * 'L' -> u
   * Final ciphertext: "raqpa"
4. **Decryption:**
   * Reverse Substitution:
     + 'y' -> O
     + 'b' -> L
     + 'r' -> H
     + 'y' -> E
     + 'u' -> L
     + Result: "OLHEL"
   * Swap Characters Back:
     + "OLHEL" -> "HLOEL"

**Source Code:**

plaintext = input("Enter the plaintext:")

key = 3

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

map\_table = {

0:'&',1:'o',2:'p',

3:'q',4:'r',5:'s',

6:'t',7:'u',8:'v',

9:'w',10:'x',11:'y',

12:'z',13:'a',14:'b',

15:'c',16:'d',17:'e',

18:'f',19:'g',20:'h',

21:'i',22:'j',23:'k',

24:'l',25:'m',26:'n'

}

reversed\_map = {value: key for key, value in map\_table.items()}

def encrypt(plaintext,key):

matrix = [['\n'for i in range(len(plaintext))] for j in range(key)]

row,col = 0,0

flag = False

for i in plaintext:

if(row == 0) or (row == key-1):

flag = not flag

matrix[row][col] = i

col+=1

if flag:

row+=1

else:

row-=1

result = []

for i in range(key):

for j in range(len(plaintext)):

if(matrix[i][j] != '\n'):

result.append(matrix[i][j])

temp = result[0]

result[0] = result[len(result)-1]

result[len(result)-1] = temp

mid\_cipher = "".join(result)

cipher=""

for i in mid\_cipher:

j = ord(i.upper()) - 65

cipher += map\_table[j]

return cipher

def decrypt(ciphertext,key):

mid\_cipher=""

for i in ciphertext:

j = chr(reversed\_map[i]+65).lower()

mid\_cipher+=j

list\_cipher = list(mid\_cipher)

temp = list\_cipher[0]

list\_cipher[0] = list\_cipher[len(list\_cipher)-1]

list\_cipher[len(list\_cipher)-1] = temp

mid\_cipher="".join(list\_cipher)

matrix = [['\n' for i in range(len(mid\_cipher))] for j in range(key)]

row,col = 0,0

flag = False

for i in range(len(mid\_cipher)):

if row == 0:

flag = True

if row == key - 1:

flag = False

matrix[row][col] = '\*'

col += 1

if flag:

row += 1

else:

row -= 1

index = 0

for i in range(key):

for j in range(len(mid\_cipher)):

if ((matrix[i][j] == '\*') and

(index < len(mid\_cipher))):

matrix[i][j] = mid\_cipher[index]

index += 1

result = []

row, col = 0, 0

for i in range(len(mid\_cipher)):

if row == 0:

dir\_down = True

if row == key-1:

dir\_down = False

if (matrix[row][col] != '\*'):

result.append(matrix[row][col])

col += 1

if dir\_down:

row += 1

else:

row -= 1

return("".join(result))

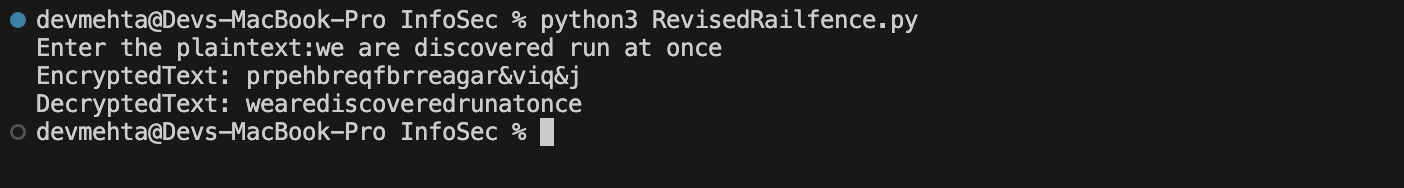
ciphertext = encrypt(plaintext,key)

decyptedtext = decrypt(ciphertext,key)

print("EncryptedText:",ciphertext)

print("DecryptedText:",decyptedtext)

**Output screenshot:**



**Comparative Analysis of original and revised approaches:**

**Security:**

* Classical Rail Fence Cipher: Vulnerable to pattern recognition and frequency analysis.
* Modified Rail Fence Cipher: More secure due to character substitution and additional steps, making frequency analysis less effective.

**Reversibility:**

* Both ciphers are reversible if the key is known, but the modified version requires reversing the substitution and Rail Fence processes.

**Performance:**

* The classical version is faster to implement, but the performance difference is minimal in practical applications. The added security of the modified version justifies the slight increase in complexity.

**Conclusion:**

The modified Rail Fence Cipher enhances the classical version by incorporating formula-based character substitution and additional character swapping. These modifications significantly increase the cipher's security by adding layers of complexity that are absent in the classical version. This makes the modified cipher more resistant to cryptographic attacks, while still maintaining the simplicity and efficiency of the original Rail Fence Cipher. For applications where added security is needed without overly complicating the encryption process, this modified version offers a balanced and effective solution.

**References:**

[Debolina Dalui, Sudipta Sahana, “Rail Fence Cipher Based Encryption Technique For Secure Data Transfer,” *International Journal of Computer Sciences and Engineering*, Vol.7, Issue.4, pp.910-914, 2019.](https://www.ijcseonline.org/full_paper_view.php?paper_id=4140)

**Experiment 4**

**AIM:** Study and implement a program for columnar Transposition Cipher

**Introduction :** The columnar transposition cipher is a fairly simple, easy to implement cipher. It is a transposition cipher that follows a simple rule for mixing up the characters in the plaintext to form the ciphertext. Although weak on its own, it can be combined with other ciphers, such as a substitution cipher, the combination of which can be more difficult to break than either cipher on it's own.

**Example :** The key for the columnar transposition cipher is a keyword e.g. GERMAN. The row length that is used is the same as the length of the keyword. To encrypt a piece of text, e.g.

defend the east wall of the castle

we write it out in a special way in a number of rows (the keyword here is GERMAN):

G E R M A N

d e f e n d

t h e e a s

t w a l l o

f t h e c a

s t l e x x

In the above example, the plaintext has been padded so that it neatly fits in a rectangle. This is known as a regular columnar transposition. An irregular columnar transposition leaves these characters blank, though this makes decryption slightly more difficult. The columns are now reordered such that the letters in the key word are ordered alphabetically.

A E G M N R

n e d e d f

a h t e s e

l w t l o a

c t f e a h

x t s e x l

The ciphertext is read off along the columns:

nalcxehwttdttfseeleedsoaxfeahl

**Original Approach**

**Source Code:**

import math

def encrypt(key,plaintext):

key\_len = len(key)

num\_cols = math.ceil(len(plaintext)/key\_len)

matrix=[[' 'for \_ in range(num\_cols)]for \_ in range(key\_len)]

index = 0

for i in range(key\_len):

for j in range(num\_cols):

if(index<len(plaintext)):

matrix[i][j] = plaintext[index]

index+=1

else:

matrix[i][j] = 'X'

index = 0

key\_list = sorted(list(key))

col\_order = [key.index(k) for k in key\_list]

ciphertext = ""

for col in col\_order:

for row in range(num\_cols):

ciphertext += matrix[col][row]

return ciphertext

def decrpyt(key,ciphertext):

key\_len = len(key)

num\_cols = math.ceil(len(ciphertext)/key\_len)

matrix=[[' 'for \_ in range(num\_cols)]for \_ in range(key\_len)]

index = 0

key\_list = sorted(list(key))

col\_order = [key.index(k) for k in key\_list]

for col in col\_order:

for row in range(num\_cols):

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

index+=1

plaintext=''

for i in range(key\_len):

for j in range(num\_cols):

plaintext+=matrix[i][j]

for i in plaintext:

if(i == 'X'):

plaintext = plaintext.replace('X','')

for i in space\_list:

plaintext = plaintext[:i] + " " + plaintext[i:]

return plaintext

plaintext = input("Enter the text:")

space\_list =[]

for i in range(len(plaintext)):

if(plaintext[i]==" "):

space\_list.append(i)

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

key = "GERMAN"

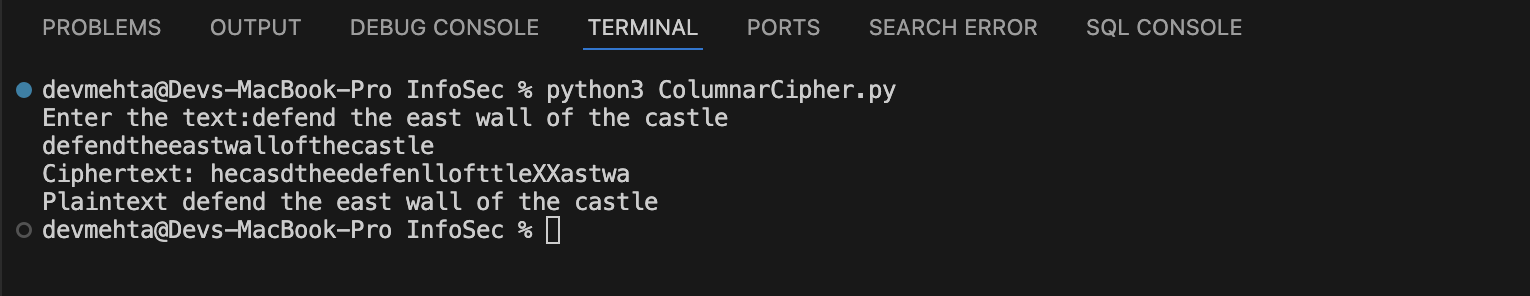
print(plaintext)

ciphertext = encrypt(key,plaintext)

print("Ciphertext:",ciphertext)

print("Plaintext",decrpyt(key,ciphertext))

**Output screenshot:**

****

**Crypt Analysis:**

* **Frequency Analysis:**

Description: Analyze the frequency of letters or patterns in the ciphertext. While this cipher doesn’t alter letter frequencies, patterns in ciphertext can reveal the key length or structure.

* **Brute Force:**

Description: Attempt all possible keys until the correct one is found. This method is feasible only for short keys due to the exponential increase in permutations with longer keys.

* **Pattern Recognition:**

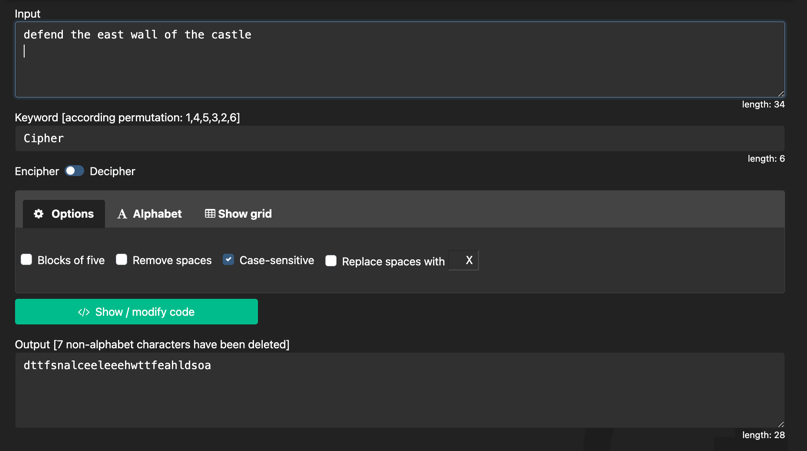
Description: Identify and analyze repeating patterns in the ciphertext to infer the key length and column order. This involves finding clues in the ciphertext’s structure.

* **Key Length Estimation:**

Description: Estimate the key length by analyzing the ciphertext’s structure and repeating patterns. Once the key length is guessed, test different column orders to decrypt the message.

* These methods leverage the predictable structure of the Columnar Transposition Cipher, making it vulnerable to various forms of cryptanalysis if the key or key length is not properly protected.

**Crypt Tool output:**

****

**Revised Approach**

**Introduction:**

The Columnar Transposition Cipher is a classical encryption technique that arranges plaintext into a grid based on a keyword and reads it column-by-column to produce ciphertext. In the modified approach, we combined the Columnar Transposition Cipher with a Caesar Cipher to enhance encryption complexity. The Caesar Cipher introduces a shift-based transformation of characters, while the Columnar Transposition Cipher arranges these characters into a grid based on the key.

**Encryption Process**

* **Caesar Cipher Encryption:**

Apply Shift: Shift each letter in the plaintext by a fixed number of positions in the alphabet (e.g., +3).

Transform Characters: Non-alphabetic characters remain unchanged.

* **Columnar Transposition Encryption:**

Grid Formation: Write the shifted plaintext into a grid with columns based on the length of the keyword.

Column Order: Read the columns in the order specified by sorting the keyword alphabetically.

**Decryption Process**

* **Columnar Transposition Decryption:**

Reconstruct Grid: Arrange the ciphertext into a grid according to the column order derived from the keyword.

Read Plaintext: Extract plaintext by reading the grid row by row.

* **Caesar Cipher Decryption:**

Reverse Shift: Shift each letter back by the fixed number of positions in the alphabet (e.g., -3).

Transform Characters: Non-alphabetic characters remain unchanged.

**Example:**

Plaintext: "HELLO WORLD"  
Caesar Key: 3  
Columnar Key: "KEY"

Caesar Encryption:

Plaintext: "HELLO WORLD" → "KHOOR ZRUOG"

Columnar Encryption:

Grid (for keyword "KEY"):

K H O

O R Z

R U O

G X X

Column Order for "KEY": 2 1 3

Ciphertext: Read columns in order: "ORU R H Z O X X G"

Ciphertext: "ORURHZOOXXG"

Decryption:

Columnar Decryption: Reconstruct grid and read row by row to get "KHOOR ZRUOG".

Caesar Decryption: Shift back to get "HELLO WORLD"

**Source Code:**

import math

def caesar\_cipher\_encrypt\_text(text, key):

encrypted\_text = ''

for char in text:

if char.isalpha():

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

encrypted\_char = chr((ord(char) - base + key) % 26 + base)

else:

encrypted\_char = char

encrypted\_text += encrypted\_char

return encrypted\_text

def caesar\_cipher\_decrypt\_text(encrypted\_text, key):

decrypted\_text = ''

for char in encrypted\_text:

if char.isalpha():

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

decrypted\_char = chr((ord(char) - base - key) % 26 + base)

else:

decrypted\_char = char

decrypted\_text += decrypted\_char

return decrypted\_text

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

key\_len = len(key)

num\_cols = math.ceil(len(plaintext) / key\_len)

matrix = [[' ' for \_ in range(num\_cols)] for \_ in range(key\_len)]

index = 0

for i in range(num\_cols):

for j in range(key\_len):

if index < len(plaintext):

matrix[j][i] = plaintext[index]

index += 1

else:

matrix[j][i] = 'X'

key\_list = sorted(list(key))

col\_order = [key.index(k) for k in key\_list]

ciphertext = ''

for col in col\_order:

for row in range(num\_cols):

ciphertext += matrix[col][row]

return ciphertext

def columnar\_transposition\_decrypt(key, ciphertext):

key\_len = len(key)

num\_cols = math.ceil(len(ciphertext) / key\_len)

matrix = [[' ' for \_ in range(num\_cols)] for \_ in range(key\_len)]

key\_list = sorted(list(key))

col\_order = [key.index(k) for k in key\_list]

index = 0

for col in col\_order:

for row in range(num\_cols):

if index < len(ciphertext):

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

index += 1

plaintext = ''

for row in range(num\_cols):

for col in range(key\_len):

if matrix[col][row] != 'X':

plaintext += matrix[col][row]

return plaintext

text = input("Enter the text: ")

key = "GERMAN"

caesar\_key = 3

encrypted\_text = caesar\_cipher\_encrypt\_text(text, caesar\_key)

encrypted\_text = columnar\_transposition\_encrypt(key, encrypted\_text)

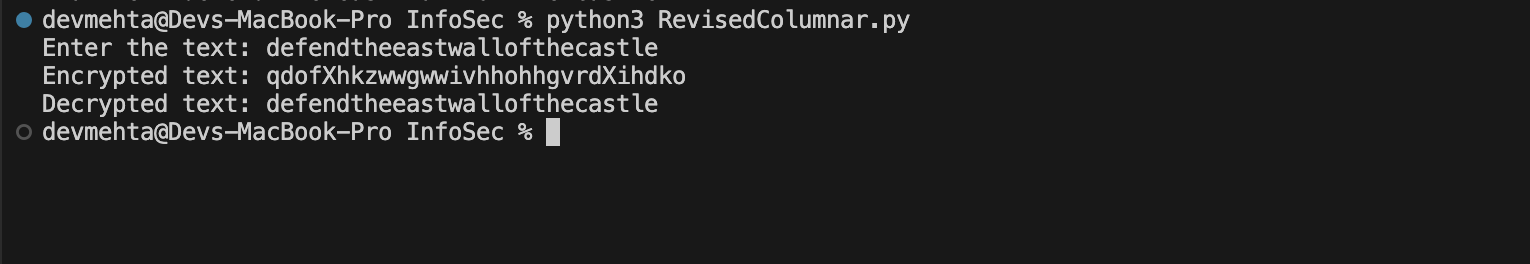
print("Encrypted text:", encrypted\_text)

decrypted\_text = columnar\_transposition\_decrypt(key, encrypted\_text)

decrypted\_text = caesar\_cipher\_decrypt\_text(decrypted\_text, caesar\_key)

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

**Output screenshot:**

****

**Comparative Analysis of original and revised approaches:**

* **Original Approach:**
  + Strengths: Simple to implement; effective against basic attacks.
  + Weaknesses: Vulnerable to frequency analysis and pattern recognition; easier to break with known plaintext attacks.
* **Modified Approach:**
  + Strengths: Adds an additional layer of security with the Caesar Cipher, complicating cryptanalysis.
  + Weaknesses: Still vulnerable to sophisticated attacks if key length or Caesar shift is known; more complex than the original but still has vulnerabilities if key management isn’t robust.

**Conclusion:**

The modified Columnar Transposition Cipher, when combined with Caesar Cipher encryption, enhances security by introducing an additional layer of encryption. While it improves upon the original approach by adding complexity, it remains susceptible to cryptanalysis if key details are discovered. For improved security, especially in sensitive applications, more advanced encryption methods are recommended. The combination of these classical techniques provides a learning tool for understanding encryption principles but may not be suitable for high-security requirements.

**References:**

[Rachmawati, Dian, Sri Melvani Hardi, and Raju Partogi Pasaribu. "Combination of columnar transposition cipher caesar cipher and lempel ziv welch algorithm in image security and compression." Journal of Physics: Conference Series. Vol. 1339. No. 1. IOP Pub](https://iopscience.iop.org/article/10.1088/1742-6596/1339/1/012007)

**Experiment 5**

**AIM:** Study and implement a program for Vigenère Cipher

**Introduction :** The Vigenère cipher (French pronunciation: [[viʒnɛːʁ]](https://en.wikipedia.org/wiki/Help:IPA/French)) is a method of [encrypting](https://en.wikipedia.org/wiki/Encryption) [alphabetic](https://en.wikipedia.org/wiki/Alphabetic) text where each letter of the [plaintext](https://en.wikipedia.org/wiki/Plaintext) is encoded with a different [Caesar cipher](https://en.wikipedia.org/wiki/Caesar_cipher), whose increment is determined by the corresponding letter of another text, the [key](https://en.wikipedia.org/wiki/Key_(cryptography)). The Vigenère cipher is therefore a special case of a [polyalphabetic substitution](https://en.wikipedia.org/wiki/Polyalphabetic_cipher).

**Example :** If the plaintext is attacking tonight and the key is OCULORHINOLARINGOLOGY, then

* the first letter a of the plaintext is shifted by 14 positions in the alphabet (because the first letter O of the key is the 14th letter of the alphabet, counting from zero), yielding o;
* the second letter t is shifted by 2 (because the second letter C of the key means 2) yielding v;
* the third letter t is shifted by 20 (U) yielding n, with wrap-around;

and so on; yielding the message ovnlqbpvt hznzouz. If the recipient of the message knows the key, they can recover the plaintext by reversing this process.

**Original Approach**

**Source Code:**

plaintext = input("Enter the plain text:")

plaintext=plaintext.upper()

for i in plaintext:

if i == " ":

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

keyword = "VIGENERE"

key = ""

i=0

while(len(key)!=len(plaintext)):

key += keyword[i]

i=(i+1)%len(keyword)

def encrypt(plaintext,key):

ciphertext = ""

for i in range(len(plaintext)):

char = plaintext[i]

if char.isupper():

encrypted\_char = chr((ord(char) + ord(key[i]) - 2 \* ord('A')) % 26 + ord('A'))

elif char.islower():

encrypted\_char = chr((ord(char) + ord(key[i]) - 2 \* ord('a')) % 26 + ord('a'))

else:

encrypted\_char = char

ciphertext+=encrypted\_char

return ciphertext

def decrypt(ciphertext,key):

plaintext = ""

for i in range(len(ciphertext)):

char = ciphertext[i]

if char.isupper():

decrypted\_char = chr((ord(char) - ord(key[i]) +26) % 26 + ord('A'))

else:

decrypted\_char = char

plaintext+=decrypted\_char

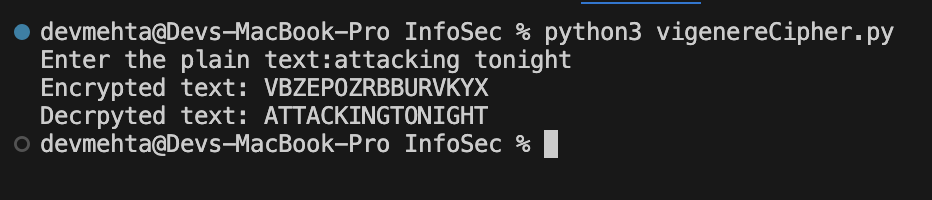
return plaintext

return

print("Encrypted text:",encrypt(plaintext,key))

print("Decrpyted text:",decrypt(encrypt(plaintext,key),key))

**Output screenshot:**



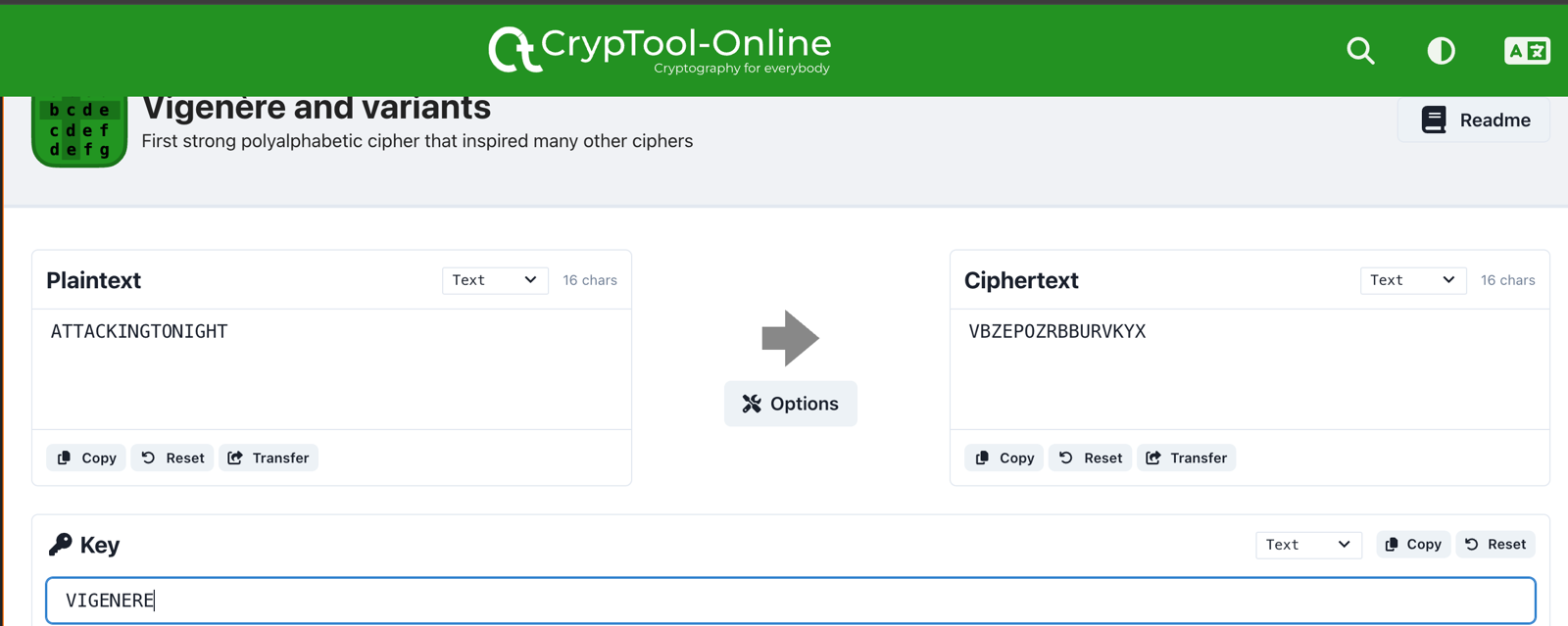
**Crypt Analysis:**

The original Vigenère Cipher, despite its historical significance, has notable vulnerabilities:

1. Frequency Analysis: Although it disguises letter frequency through polyalphabetic substitution, it is still susceptible to frequency analysis. Repeating patterns in the ciphertext can reveal periodic structures, allowing attackers to determine the key length and eventually break the cipher.
2. Kasiski Examination: This technique involves analyzing repeated sequences in the ciphertext to deduce the length of the keyword. Once the key length is known, the ciphertext can be segmented into columns corresponding to each character in the key, making it easier to apply frequency analysis to each segment.
3. Friedman Test: This statistical method estimates the key length by calculating the index of coincidence in the ciphertext. This test helps in identifying the repetition of characters and deducing the key length.

Overall, while the Vigenère Cipher provides a basic level of encryption, its vulnerability to these cryptanalytic methods makes it less secure by modern standards.

**Crypt Tool output:**



**Revised Approach**

**Introduction:**

The extended Vigenère Cipher is an improvement over the traditional cipher. It encrypts not only letters but also numbers, and special characters. The encryption and decryption process uses a repeating keyword that matches the length of the input message, allowing secure encoding of a wider set of characters. The algorithm operates on modulo 95 to handle all printable ASCII characters (ranging from space to ~).

**Encryption Process:**

1. Prepare the Plaintext:
   * Remove spaces from the plaintext (spaces won't be encrypted).
2. Generate the Key:
   * Choose a keyword (e.g., "VIGENERE").
   * Repeat the keyword until its length matches the length of the plaintext.
   * For example, if the plaintext is "HELLO123!" and the keyword is "VIGENERE", the repeated key will be "VIGENEREVI".
3. Encrypt the Message:
   * For each character in the plaintext, convert it to its numeric ASCII value.
   * Do the same for the corresponding character in the key.
   * Add the numeric equivalents of the plaintext and the key characters.
   * Take the sum modulo 95 to ensure the result stays within the ASCII printable range (32 to 126).
   * Convert the result back to a character.
   * Continue this for every character in the plaintext.

**Decryption Process:**

1. Prepare the Ciphertext:
   * Take the encrypted message (ciphertext) and ensure that spaces or any extra characters (added for formatting) are removed.
2. Generate the Key:
   * Use the same key that was used for encryption, and repeat it to match the length of the ciphertext.
3. Decrypt the Message:
   * For each character in the ciphertext, convert it to its numeric ASCII value.
   * Convert the corresponding character in the key to its numeric value as well.
   * Subtract the key character's numeric value from the ciphertext character's numeric value.
   * If the result is negative, add 95 before taking the modulo 95 operation.
   * Convert the result back to a character to obtain the original plaintext.
   * Repeat this process for every character in the ciphertext.

**Example:**

* Plaintext: "HELLO123!"
* Keyword: "VIGENERE"

Step 1: Remove spaces (plaintext)

plaintext

HELLO123!

Step 2: Generate the repeating key

key

VIGENEREVI

Step 3: Encryption

We take each letter from the plaintext and the corresponding letter from the key:

1. H (72) + V (86) = 158, 158 % 95 = 63 → ASCII ?
2. E (69) + I (73) = 142, 142 % 95 = 47 → ASCII /
3. L (76) + G (71) = 147, 147 % 95 = 52 → ASCII 4
4. L (76) + E (69) = 145, 145 % 95 = 50 → ASCII 2
5. O (79) + N (78) = 157, 157 % 95 = 62 → ASCII >
6. 1 (49) + E (69) = 118, 118 % 95 = 23 → ASCII 7
7. 2 (50) + R (82) = 132, 132 % 95 = 37 → ASCII &
8. 3 (51) + E (69) = 120, 120 % 95 = 25 → ASCII 9
9. ! (33) + V (86) = 119, 119 % 95 = 24 → ASCII 8

Ciphertext: "?/42>7&98"

Step 4: Decryption

To decrypt, we reverse the encryption steps:

1. ? (63) - V (86) = -23, (-23 + 95) % 95 = 72 → ASCII H
2. / (47) - I (73) = -26, (-26 + 95) % 95 = 69 → ASCII E
3. 4 (52) - G (71) = -19, (-19 + 95) % 95 = 76 → ASCII L
4. 2 (50) - E (69) = -19, (-19 + 95) % 95 = 76 → ASCII L
5. > (62) - N (78) = -16, (-16 + 95) % 95 = 79 → ASCII O
6. 7 (23) - E (69) = -46, (-46 + 95) % 95 = 49 → ASCII 1
7. & (37) - R (82) = -45, (-45 + 95) % 95 = 50 → ASCII 2
8. 9 (25) - E (69) = -44, (-44 + 95) % 95 = 51 → ASCII 3
9. 8 (24) - V (86) = -62, (-62 + 95) % 95 = 33 → ASCII !

Decrypted Plaintext: "HELLO123!"

**Source Code:**

def char\_to\_num(char):

return ord(char) - 32

def num\_to\_char(num):

return chr(num + 32)

def encrypt(plaintext, key):

ciphertext = ""

for i in range(len(plaintext)):

p\_num = char\_to\_num(plaintext[i])

k\_num = char\_to\_num(key[i])

encrypted\_num = (p\_num + k\_num) % 95

ciphertext += num\_to\_char(encrypted\_num)

return ciphertext

def decrypt(ciphertext, key):

plaintext = ""

for i in range(len(ciphertext)):

c\_num = char\_to\_num(ciphertext[i])

k\_num = char\_to\_num(key[i])

decrypted\_num = (c\_num - k\_num + 95) % 95

plaintext += num\_to\_char(decrypted\_num)

return plaintext

plaintext = input("Enter the plain text: ")

keyword = "VIGENERE"

key = ""

i=0

while(len(key)!=len(plaintext)):

key += keyword[i]

i=(i+1)%len(keyword)

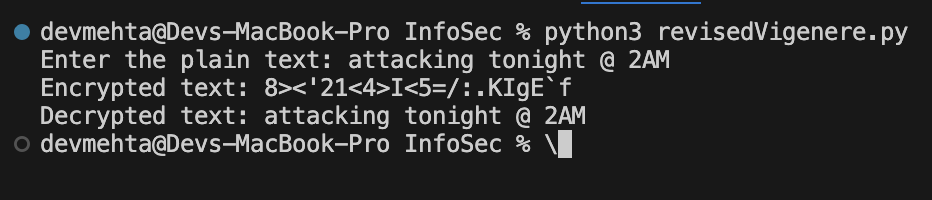
ciphertext = encrypt(plaintext, key)

print("Encrypted text:", ciphertext)

decrypted\_text = decrypt(ciphertext, key)

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

**Output screenshot:**

****

**Comparative Analysis of original and revised approaches:**

**Original Vigenère Cipher:**

* Strengths:
  + Simple to implement.
  + Effective against basic attacks due to polyalphabetic substitution.
* Weaknesses:
  + Vulnerable to frequency analysis and pattern recognition.
  + Easier to break with known plaintext attacks, especially with short keys.

**Modified Extended Vigenère Cipher:**

* Strengths:
  + Supports a wide range of characters (letters, numbers, symbols), making it versatile for modern text.
  + Adds complexity with a larger character set and table, increasing security.
* Weaknesses:
  + Still vulnerable to sophisticated attacks if key management isn’t robust.
  + More complex to implement and manage, and can be broken if the key is weak or exposed.

**Conclusion:**

The practical demonstrated that the original cipher is simple and effective for basic text encryption but is vulnerable to attacks due to its limited character set and susceptibility to frequency analysis. The modified version enhances security by supporting a wider range of characters and a larger table, making it more suitable for modern encryption needs. However, it remains complex and requires careful key management to prevent sophisticated attacks.

**References:**

[Nahar, Khairun, and Partha Chakraborty. "A modified version of Vigenere cipher using 95× 95 table." *International Journal of Engineering and Advanced Technology (IJEAT)* 9.5 (2020): 1144-1148.](https://www.ijeat.org/wp-content/uploads/papers/v9i5/E9941069520.pdf)

**Experiment 6**

**AIM:** Study and Implement a program for n-gram Hill Cipher

**Introduction :** Hill cipher is a polygraphic substitution cipher based on linear algebra. Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.  
The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

**Example :** Input : Plaintext: ACT

Key: GYBNQKURP

Output : Ciphertext: POH

**Original Approach**

**Source Code:**

def conversion():

letter\_to\_num, num\_to\_letter = {}, {}

for idx in range(26):

letter = chr(idx + 65)

letter\_to\_num[letter] = idx

num\_to\_letter[idx] = letter

return letter\_to\_num, num\_to\_letter

def calc\_determinant\_2x2(matrix):

return (matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]) % 26

def find\_inverse\_2x2(matrix):

determinant = calc\_determinant\_2x2(matrix)

if determinant == 0:

return None

for num in range(26):

if (determinant \* num) % 26 == 1:

modular\_inverse = num

break

else:

return None

return [

[(matrix[1][1] \* modular\_inverse) % 26, (-matrix[0][1] \* modular\_inverse) % 26],

[(-matrix[1][0] \* modular\_inverse) % 26, (matrix[0][0] \* modular\_inverse) % 26]

]

def matrix\_vector\_mult(matrix, vec):

result = [0, 0]

for row\_idx in range(2):

for col\_idx in range(2):

result[row\_idx] += matrix[row\_idx][col\_idx] \* vec[col\_idx]

result[row\_idx] %= 26

return result

def convert\_key\_matrix\_to\_nums(matrix, mapping):

return [[mapping[char] for char in row] for row in matrix]

def encrypt\_hill\_cipher(plain\_text, key\_matrix):

letter\_to\_num, num\_to\_letter = convertion()

numeric\_key = convert\_key\_matrix\_to\_nums(key\_matrix, letter\_to\_num)

plain\_text = ''.join([char for char in plain\_text.upper() if char.isalpha()])

if len(plain\_text) % 2 != 0:

plain\_text += 'X'

encrypted\_text = ""

for idx in range(0, len(plain\_text), 2):

pair = plain\_text[idx:idx+2]

vec = [letter\_to\_num[pair[0]], letter\_to\_num[pair[1]]]

encrypted\_vec = matrix\_vector\_mult(numeric\_key, vec)

encrypted\_text += num\_to\_letter[encrypted\_vec[0]] + num\_to\_letter[encrypted\_vec[1]]

return encrypted\_text

def decrypt\_hill\_cipher(cipher\_text, key\_matrix):

letter\_to\_num, num\_to\_letter = convertion()

numeric\_key = convert\_key\_matrix\_to\_nums(key\_matrix, letter\_to\_num)

inv\_key\_matrix = find\_inverse\_2x2(numeric\_key)

if inv\_key\_matrix is None:

return "Key matrix is not invertible."

decrypted\_text = ""

for idx in range(0, len(cipher\_text), 2):

pair = cipher\_text[idx:idx+2]

vec = [letter\_to\_num[pair[0]], letter\_to\_num[pair[1]]]

decrypted\_vec = matrix\_vector\_mult(inv\_key\_matrix, vec)

decrypted\_text += num\_to\_letter[decrypted\_vec[0]] + num\_to\_letter[decrypted\_vec[1]]

return decrypted\_text

key\_grid = [['D', 'D'], ['C', 'F']]

plain\_message = "EXAMOVER"

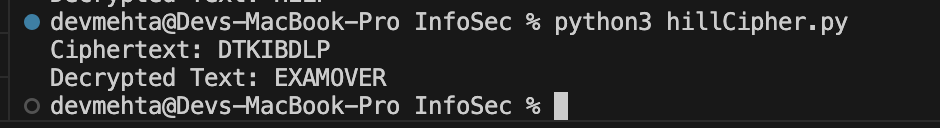
cipher\_output = encrypt\_hill\_cipher(plain\_message, key\_grid)

print(f"Ciphertext: {cipher\_output}")

decrypted\_output = decrypt\_hill\_cipher(cipher\_output, key\_grid)

print(f"Decrypted Text: {decrypted\_output}")

**Output screenshot:**



**Crypt Analysis:**

Cryptanalysis of the Hill cipher primarily focuses on attacking the key matrix used in encryption. Since the Hill cipher operates on blocks of letters, analyzing larger amounts of ciphertext can make it vulnerable, especially if you know part of the plaintext (known as a known-plaintext attack). Here's a deeper look into some methods of cryptanalysis for the Hill cipher:

**1. Known-Plaintext Attack:**

If an attacker has both the plaintext and corresponding ciphertext, they can break the Hill cipher. Suppose you know n plaintext-ciphertext pairs. You can set up a system of linear equations to solve for the key matrix. Since the Hill cipher relies on matrix multiplication, this system can be solved using standard techniques like Gaussian elimination, provided the matrix is invertible mod 26.

For example, suppose the plaintext HEL encrypts to XDP. Converting letters to numbers:

* H = 7, E = 4, L = 11 (Plaintext vector)
* X = 23, D = 3, P = 15 (Ciphertext vector)

Using multiple such pairs, the attacker can solve for the key matrix via linear algebra.

**2. Ciphertext-Only Attack:**

While more difficult, a ciphertext-only attack is possible under certain conditions:

* **Frequency Analysis**: The Hill cipher doesn't hide letter frequencies as effectively as more modern ciphers, so a frequency analysis could reveal common letters. For example, 'E' is the most frequent letter in English, and an attacker might analyze letter frequencies in the ciphertext to make educated guesses about the plaintext.
* **Brute Force on Small Key Sizes**: For small n-grams (e.g., 2x2 or 3x3 matrices), brute-force attacks are feasible. There are a limited number of possible matrices, and the attacker could try all combinations to find the correct key.

**3. Chosen Plaintext Attack:**

If an attacker can choose the plaintext, they can craft plaintext blocks in a way that simplifies solving for the key matrix. For instance, if the attacker chooses the plaintext as AAA, BBB, etc., the ciphertext generated from these blocks would make it easier to break the cipher, as the structure of the key matrix would be more exposed.

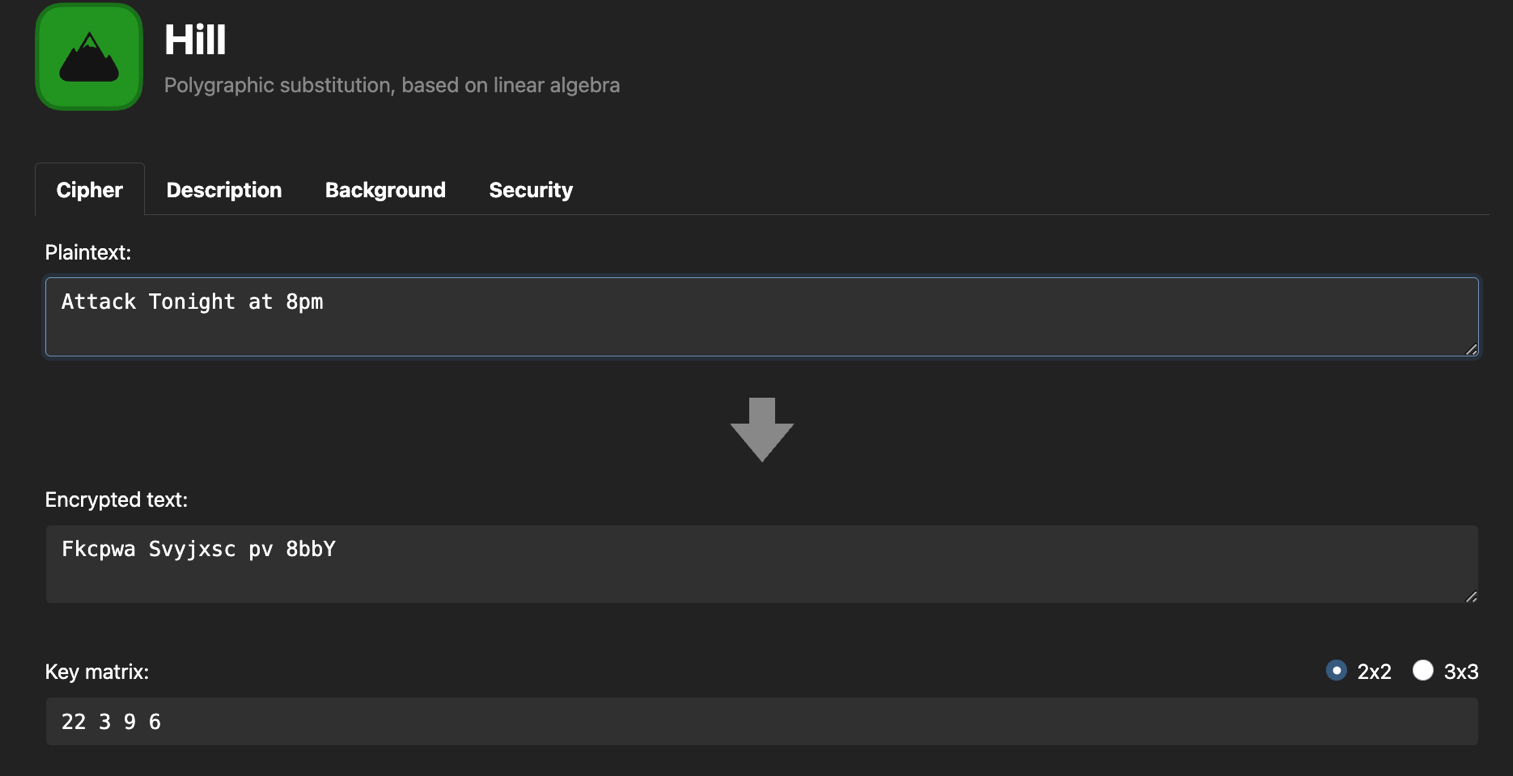
**Example:**

If the plaintext ACT encrypts to POH, and CAT encrypts to XYZ, the attacker can create a system of equations to solve for the key matrix and decrypt any future ciphertext.

**4. Matrix Inversion Challenges:**

A major part of cryptanalysis for Hill cipher is finding the inverse of the key matrix. If the matrix is not invertible (i.e., its determinant is 0 mod 26), then the cipher is unusable. But once the matrix is found, the inverse modulo 26 reveals the decryption key, enabling the attacker to decrypt any ciphertext.

**Crypt Tool output:**



**Revised Approach**

**Introduction:**

The Hill cipher is a classical encryption technique that utilizes linear algebra to transform plaintext into ciphertext through matrix operations. This revised approach enhances the traditional Hill cipher by incorporating the following features:

1. Character Mapping: Each character is mapped to a numerical value, where 'A' to 'Z' corresponds to 1-26, and space is represented as 27.
2. Block Formation: The plaintext is divided into n×n times n×n blocks, filling any shortfall with spaces to ensure uniformity.
3. Key Matrices: Two square matrices, P (additive key) and Q (multiplicative key, non-singular), are used in the encryption process.
4. Encryption and Decryption: The ciphertext B is obtained using B=(A+P)Q, and the original plaintext AA is recovered with A= BQ^{-1} – P.
5. Message Reconstruction: The numerical values from the decrypted matrix are converted back into characters to reconstruct the original message.

This approach not only retains the core principles of the Hill cipher but also enhances usability and security, making it a practical solution for modern cryptographic applications.

**Source Code:**

import numpy as np

def char\_to\_num(char):

if char == ' ':

return 27

return ord(char.upper()) - ord('A') + 1

def num\_to\_char(num):

if num == 27:

return ' '

return chr(num + ord('A') - 1)

def create\_matrix(block, n):

nums = [char\_to\_num(char) for char in block]

if len(nums) < n \* n:

nums += [27] \* (n \* n - len(nums))

return np.array(nums).reshape(n, n)

def example\_keys(n):

P = np.random.randint(1, 28, size=(n, n))

Q = np.random.randint(1, 28, size=(n, n))

while np.linalg.det(Q) == 0:

Q = np.random.randint(1, 28, size=(n, n))

return P, Q

def encrypt(message, P, Q):

n = P.shape[0]

padded\_message = message.upper()

while len(padded\_message) % (n \* n) != 0:

padded\_message += ' '

encrypted\_blocks = []

for i in range(0, len(padded\_message), n \* n):

block = padded\_message[i:i + n \* n]

A = create\_matrix(block, n)

B = np.dot(A + P, Q) # B = (A + P)Q

encrypted\_blocks.append(B)

return encrypted\_blocks

def decrypt(encrypted\_blocks, P, Q):

Q\_inv = np.linalg.inv(Q)

decrypted\_blocks = []

for B in encrypted\_blocks:

A = np.dot(B, Q\_inv) - P # A = BQ^{-1} - P

decrypted\_blocks.append(A)

return decrypted\_blocks

def matrix\_to\_message(blocks):

message = ''

for matrix in blocks:

for row in matrix:

message += ''.join(num\_to\_char(round(num)) for num in row)

return message.strip()

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

message = input("Enter the plaintext:")

n = 3

P, Q = example\_keys(n)

print("Key Matrix P:\n", P)

print("Key Matrix Q:\n", Q)

encrypted\_blocks = encrypt(message, P, Q)

print("\nCiphertext Matrices B:")

for i, B in enumerate(encrypted\_blocks):

print(f"Block {i + 1}:\n{B}")

decrypted\_blocks = decrypt(encrypted\_blocks, P, Q)

print("\nDecrypted Matrices A:")

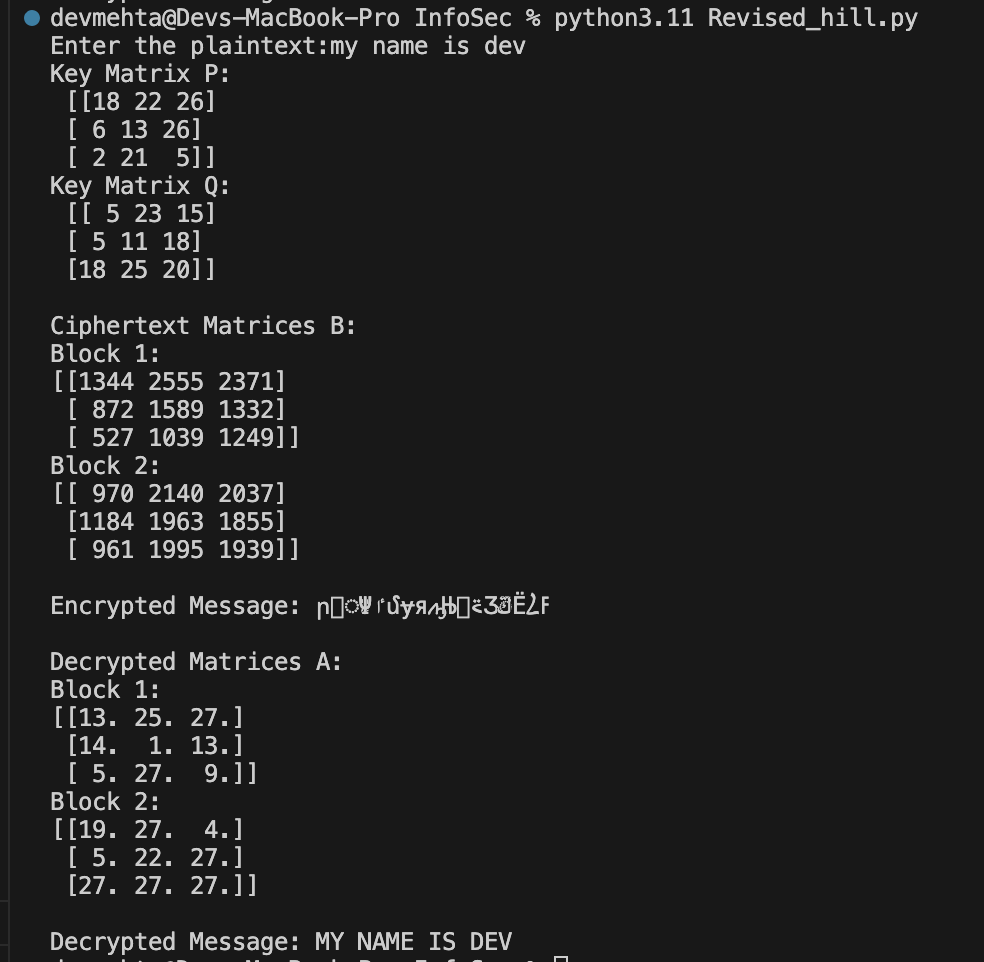
for i, A in enumerate(decrypted\_blocks):

print(f"Block {i + 1}:\n{A}")

decrypted\_message = matrix\_to\_message(decrypted\_blocks)

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

**Output screenshot:**

****

**Comparative Analysis of original and revised approaches:**

* **Basic Concept**:
  + **Original Hill Cipher**: Uses fixed block sizes determined by the key matrix for linear transformations.
  + **Revised Hill Cipher**: Allows flexible block sizes with padding for incomplete blocks, handling spaces explicitly.
* **Padding**:
  + **Original Hill Cipher**: No formal padding; can result in unusable final blocks.
  + **Revised Hill Cipher**: Implements padding with spaces, ensuring all blocks are complete and enhancing encryption consistency.
* **Key Matrix**:
  + **Original Hill Cipher**: Requires a non-singular key matrix, limiting key options.
  + **Revised Hill Cipher**: Utilizes two matrices P and Q for better key management and flexibility.
* **Security**:
  + **Original Hill Cipher**: Vulnerable to known plaintext attacks due to linearity.
  + **Revised Hill Cipher**: Improved security through block-wise processing and padding, making patterns less discernible.
* **Complexity**:
  + **Original Hill Cipher**: Simpler, faster operations, but less secure.
  + **Revised Hill Cipher**: More complex due to additional computations, but offers better security.
* **Error Propagation**:
  + **Original Hill Cipher**: Errors affect all subsequent characters in a block.
  + **Revised Hill Cipher**: Similar propagation, but padding mitigates the impact of incomplete data.
* **Practical Use**:
  + **Original Hill Cipher**: Less suitable for modern applications due to security vulnerabilities.
  + **Revised Hill Cipher**: More applicable in contemporary contexts due to enhanced security and flexibility.

**Conclusion:**

In conclusion, while the Original Hill Cipher is historically significant, it is less secure for modern applications. The Revised Hill Cipher addresses many of its limitations, providing enhanced flexibility, improved security, and a more robust encryption process. However, both ciphers share similar vulnerabilities related to linear transformations, underscoring the importance of complementary cryptographic techniques for ensuring robust security.

**References:**

**Citation:**

[Rekha, G. & Srinivas, V.. (2023). A Novel Approach in Hill Cipher Cryptography. INTERNATIONAL JOURNAL OF MATHEMATICS AND COMPUTER RESEARCH. 11. 10.47191/ijmcr/v11i6.06.](https://www.researchgate.net/publication/372214646_A_NOVEL_APPROACH_IN_HILL_CIPHER_CRYPTOGRAPHY)

**Experiment 7**

**AIM:** Study and Use of RSA algorithm (encryption and decryption)

**Introduction :** RSA (Rivest–Shamir–Adleman) is a [public-key cryptosystem](https://en.wikipedia.org/wiki/Public-key_cryptography), one of the oldest widely used for secure data transmission. The [initialism](https://en.wikipedia.org/wiki/Initialism) "RSA" comes from the surnames of [Ron Rivest](https://en.wikipedia.org/wiki/Ron_Rivest), [Adi Shamir](https://en.wikipedia.org/wiki/Adi_Shamir) and [Leonard Adleman](https://en.wikipedia.org/wiki/Leonard_Adleman), who publicly described the algorithm in 1977. An equivalent system was developed secretly in 1973 at [Government Communications Headquarters](https://en.wikipedia.org/wiki/Government_Communications_Headquarters) (GCHQ), the British [signals intelligence](https://en.wikipedia.org/wiki/Signals_intelligence) agency, by the English mathematician [Clifford Cocks](https://en.wikipedia.org/wiki/Clifford_Cocks). That system was [declassified](https://en.wikipedia.org/wiki/Classified_information) in 1997.

In a public-key [cryptosystem](https://en.wikipedia.org/wiki/Cryptosystem), the [encryption key](https://en.wikipedia.org/wiki/Encryption_key) is public and distinct from the [decryption key](https://en.wikipedia.org/wiki/Decryption_key), which is kept secret (private). An RSA user creates and publishes a public key based on two large [prime numbers](https://en.wikipedia.org/wiki/Prime_number), along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decrypted by someone who knows the private key.

The security of RSA relies on the practical difficulty of [factoring](https://en.wikipedia.org/wiki/Factorization) the product of two large [prime numbers](https://en.wikipedia.org/wiki/Prime_number), the "[factoring problem](https://en.wikipedia.org/wiki/Factoring_problem)". Breaking RSA encryption is known as the [RSA problem](https://en.wikipedia.org/wiki/RSA_problem). Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used.

**Example :** First, let’s assume you calculated your keys as follows:

1. p=17 and q =7. Notice 17 and 7 are both prime numbers
2. n= 17 x 7 = 119
3. f(n) = (17-1)(7-1)=96
4. e=11, notice that gcd(96,11)=1 and 1<11<96
5. d=35

The keys are:

* private key: {35,119}
* public key: {11,119}

Now, you published somehow your public key and I want to send you a message only you can read. The message I want to send you is M=21. Notice that you can always find a numeric representation for any message. At the end of the day, all data in a computer is represented as numbers.

*C*= *Me*mod *n=2111 mod 119 = 98*

When you receive the encrypted message C=45, you use your private key to decrypt it.

*M*= *Cd*mod *n=9835 mod 119 = 21*

**Original Approach**

**Source Code:**

def rsa\_secrets():

p = 61

q = 53

n = p \* q

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

e = 17

d = pow(e, -1, phi)

public\_key = [e,n]

private\_key = [d,n]

return private\_key,public\_key

def encryption(plaintext,public\_key):

encrypted\_text = [(ord(char)\*\*public\_key[0])%public\_key[1] for char in plaintext]

return encrypted\_text

def decryption(ciphertext,private\_key):

decrypted\_text = ''.join([chr((char \*\* private\_key[0]) % private\_key[1]) for char in ciphertext])

return decrypted\_text

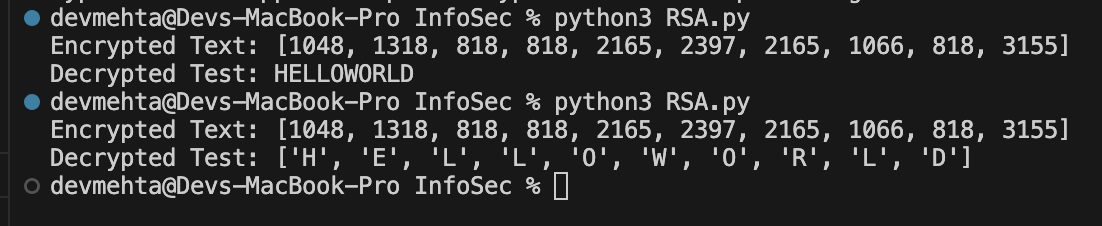
plaintext = "HELLOWORLD"

public\_key,private\_key = rsa\_secrets()

ciphertext = encryption(plaintext,public\_key)

print("Encrypted Text:", ciphertext)

print("Decrypted Test:", decryption(ciphertext,private\_key))

**Output screenshot:**

**Crypt Analysis:**

RSA cryptanalysis aims to break the RSA encryption without knowing the private key. Here are the key methods attackers use:

**1. Factoring the Modulus (n)**

RSA security relies on the difficulty of factoring n=p×qn = p times qn=p×q. If an attacker can factor n, they can compute the private key. Modern algorithms like the **General Number Field Sieve (GNFS)** are used for this.

**Prevention**: Use large key sizes (2048 bits or more).

**2. Chosen Ciphertext Attack (CCA)**

This attack involves sending manipulated ciphertexts to decrypt. Improper padding schemes (e.g., **Padding Oracle Attack**) allow attackers to recover the private key.

**Prevention**: Use secure padding (e.g., OAEP) and constant-time decryption.

**3. Low Exponent Attack**

Using small exponents like e=3e = 3e=3 can make RSA vulnerable, especially if the same message is encrypted for multiple recipients.

**Prevention**: Use a large public exponent (e.g., e=65537e = 65537e=65537) and proper padding.

**4. Timing Attacks**

Attackers analyze the time taken for encryption or decryption to recover the private key.

**Prevention**: Implement constant-time algorithms.

**5. Common Modulus Attack**

If two users share the same modulus but have different exponents, an attacker can decrypt shared messages.

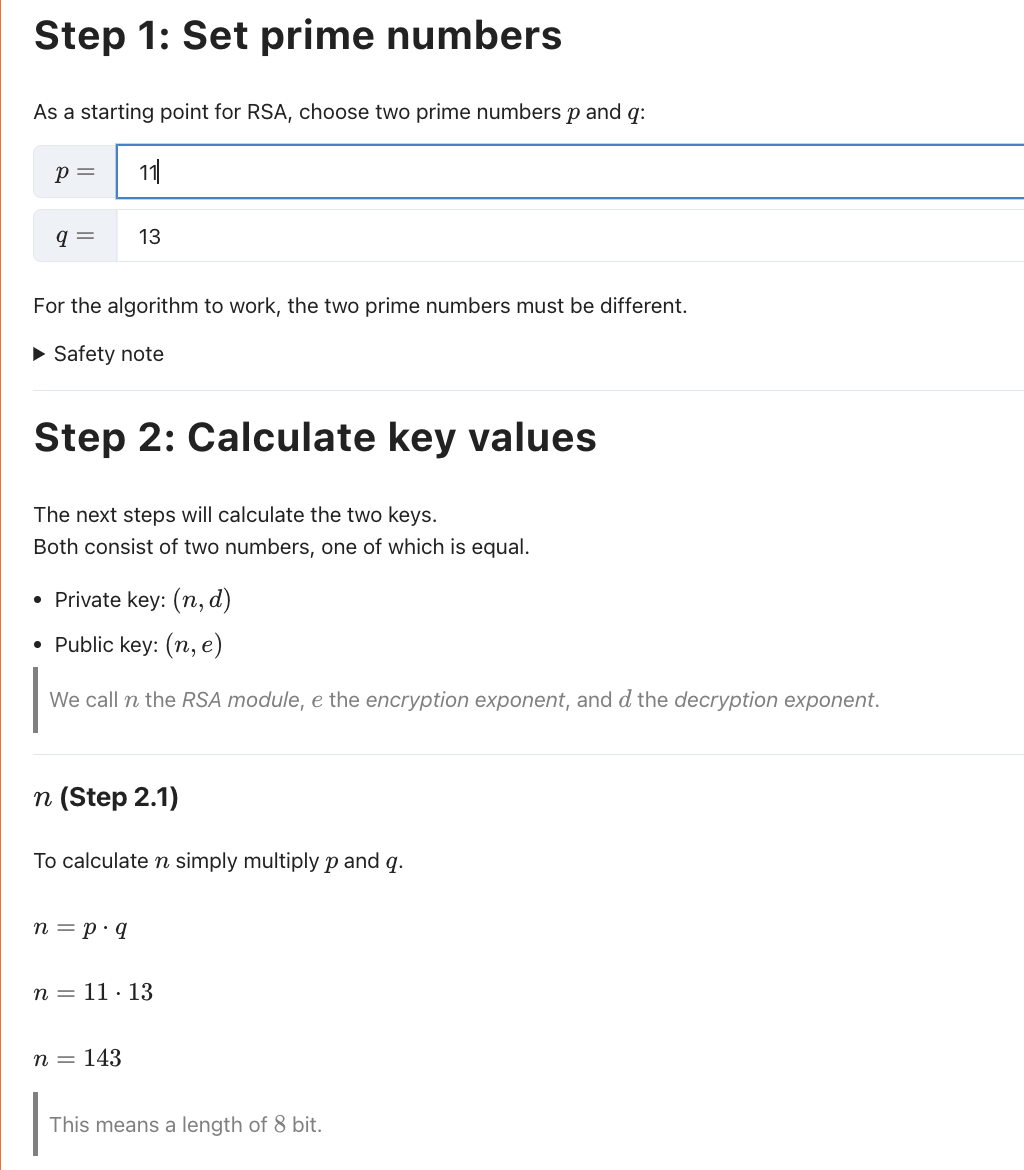
**Prevention**: Ensure each key pair has a unique modulus.

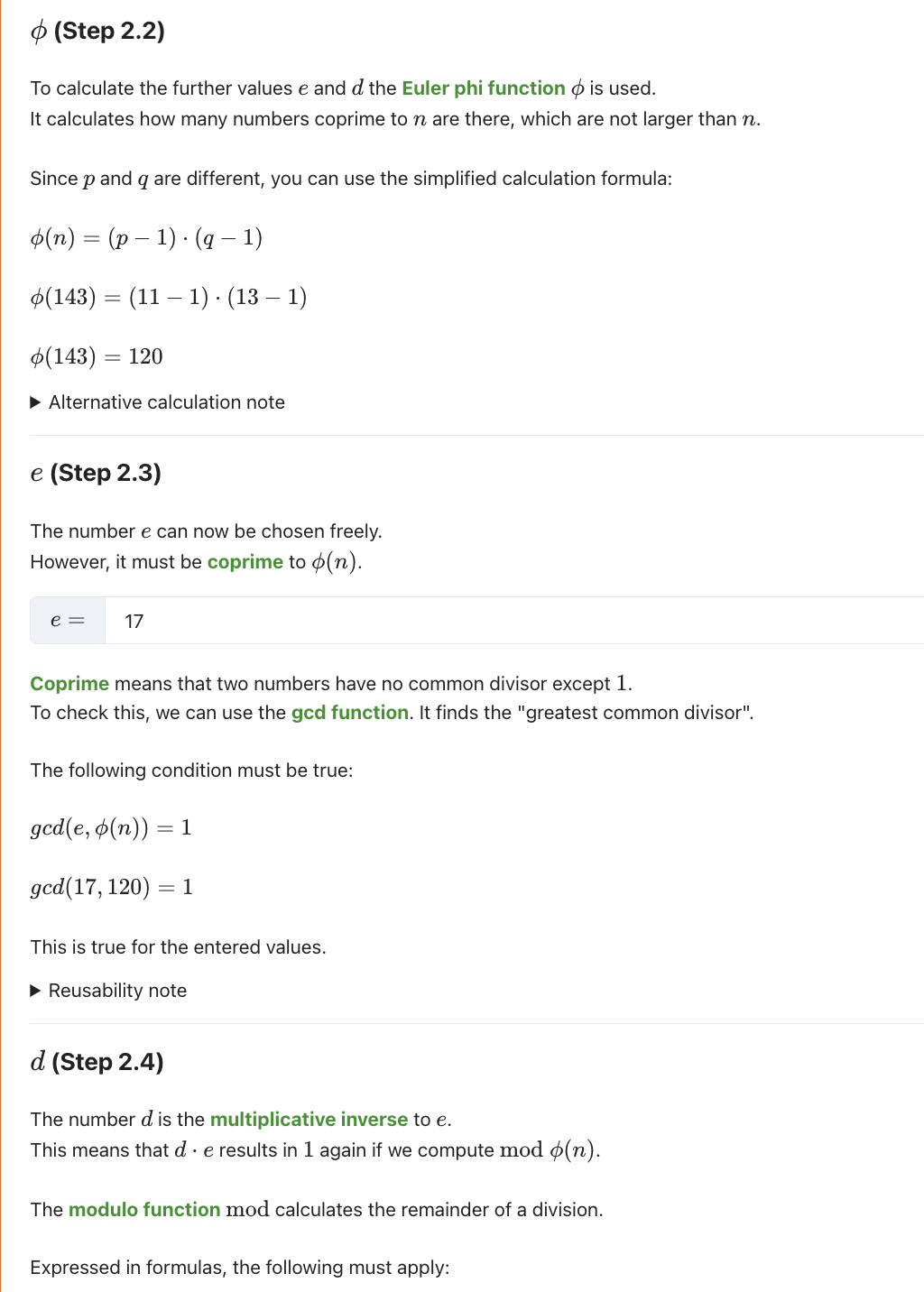
**6. Weak Random Number Generation**

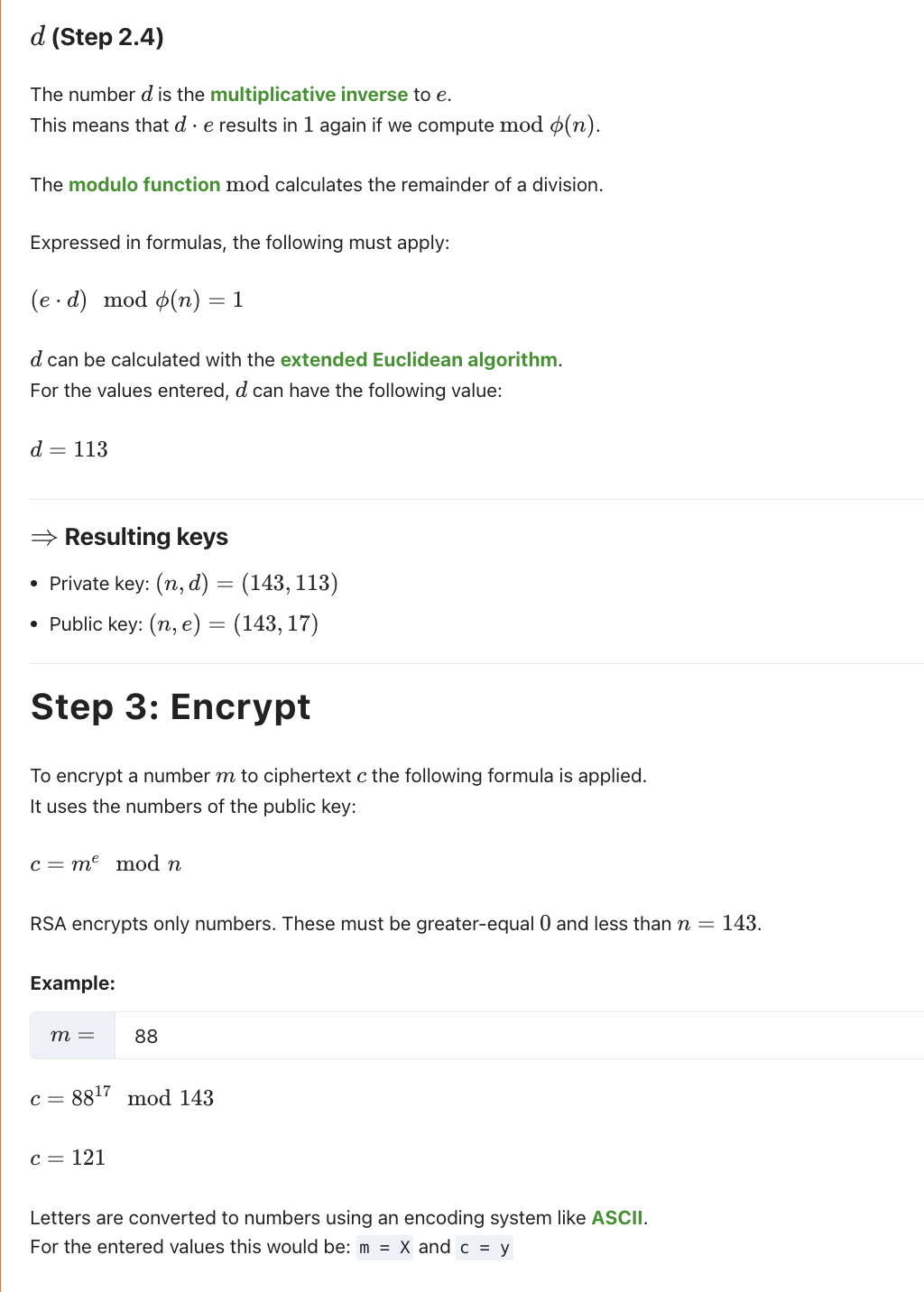
If primes p and q are generated with weak random numbers, an attacker can predict them.

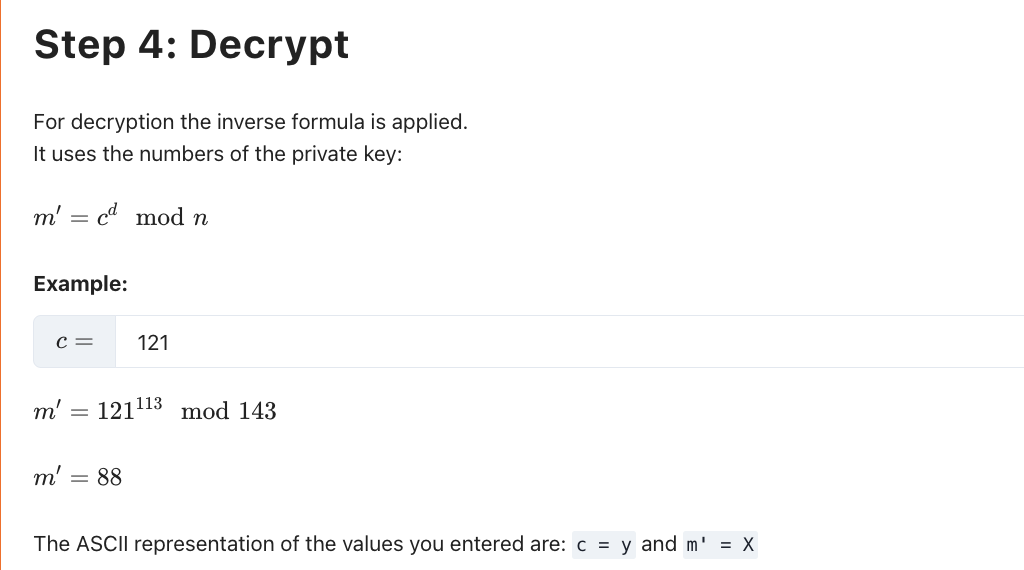
**Prevention**: Use secure random number generators

**Crypt Tool output:**

****

****

****

****

**Conclusion:**

Despite facing cryptanalytic challenges, RSA continues to be a popular and secure public-key cryptosystem. It is crucial to utilize sufficiently large key sizes (such as 2048 bits or more) and to implement suitable countermeasures to guard against known attacks.

**References:**

[Xin Zhou and Xiaofei Tang, "Research and implementation of RSA algorithm for encryption and decryption," Proceedings of 2011 6th International Forum on Strategic Technology, Harbin, Heilongjiang, 2011, pp. 1118-1121, doi: 10.1109/IFOST.2011.6021216. keywords: {Cryptography;Algorithm design and analysis;RSA algorithm;encryption;decryption},](https://ieeexplore.ieee.org/document/6021216)

**Experiment 8**

**AIM:** Study and implement a program of the Digital Signature with RSA algorithm (Reverse RSA)

**Introduction :** Assume that there is a sender (A) and a receiver (B). A wants to send a message (M) to B along with the digital signature (DS) calculated over the message.

Step-1 : Sender A uses SHA-1 Message Digest Algorithm to calculate the message digest (MD1) over the original message M.

Step-2 : A now encrypts the message digest with its private key. The output of this process is called Digital Signature (DS) of A.

Step-3 : Now sender A sends the digital signature (DS) along with the original message (M) to B.

Step-4 : When B receives the Original Message(M) and the Digital Signature(DS) from A, it first uses the same message-digest algorithm as was used by A and calculates its own Message Digest (MD2) for M.

Step-5 : Now B uses A’s public key to decrypt the digital signature because it was encrypted by A’s private key. The result of this process is the original Message Digest (MD1) which was calculated by A.

Step-6 : If MD1==MD2, the following facts are established as follows.

* B accepts the original message M as the correct, unaltered message from A.
* It also ensures that the message came from A and not someone posing as A.

**Original Approach**

**Source Code:**

import hashlib

def rsa\_secrets():

p = 61

q = 53

n = p \* q

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

e = 17

d = pow(e, -1, phi)

public\_key = [e,n]

private\_key = [d,n]

return private\_key,public\_key

def encrypt(text,private\_key):

d = private\_key[0]

n = private\_key[1]

encrypted\_text = [(ord(char)\*\*d)%n for char in text]

return encrypted\_text

def decrypt(text,public\_key):

e = public\_key[0]

n = public\_key[1]

decrypted\_text = ''.join([chr((char\*\*e)%n) for char in text])

return decrypted\_text

plaintext = input("Enter the Plain text:")

msg\_digest = (hashlib.sha1(plaintext.encode())).hexdigest()

print("Message\_digest(sender):",msg\_digest)

private\_key,public\_key = rsa\_secrets()

signature\_list = encrypt(msg\_digest,private\_key)

print(signature\_list)

final\_msg = plaintext+"$"+str(signature\_list)

print("Final Mesaage from sender(A):", final\_msg)

msg\_list = final\_msg.split("$")

print(msg\_list)

received\_plaintext = msg\_list[0]

received\_signature\_list = eval(msg\_list[1])

msg\_digest\_2 = decrypt(received\_signature\_list,public\_key)

print(msg\_digest\_2)

msg\_digest\_3 = (hashlib.sha1(received\_plaintext.encode())).hexdigest()

print(msg\_digest\_3)

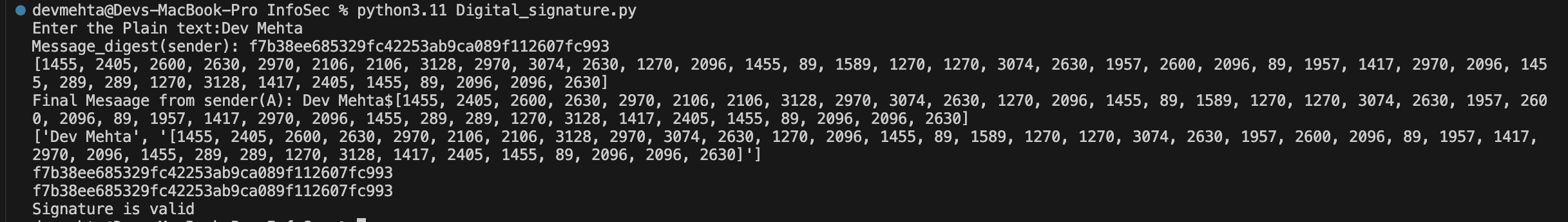
if msg\_digest\_2 == msg\_digest\_3:

print("Signature is valid")

else:

print("Signature is invalid")

**Output screenshot:**



**Cryptanalysis of Reverse RSA Digital Signature:**

1. **Key Size**: Small key sizes are vulnerable to **factorization attacks**. Use at least 2048-bit keys for security.
2. **Public Key Exposure**: The public key is widely known, so weak keys or poor implementation can lead to attacks like **Wiener’s attack** to derive the private key.
3. **Hash Vulnerability**: Using weak hashes like **SHA-1** allows **collision attacks**, letting attackers forge signatures. Use **SHA-256** or stronger.
4. **Replay Attacks**: Without **timestamps** or unique IDs, captured signatures can be reused by attackers.
5. **Message Recovery**: Weak padding schemes or small exponents make the system vulnerable to **message recovery attacks**.

**Prevention:**

* Use strong padding (e.g., PKCS#1).
* Secure with **SHA-256**.
* Employ **session tokens** or **timestamps** for added protection.

**Conclusion:**

In this practical, we successfully implemented RSA-based digital signatures to ensure message integrity and authenticity. By generating a SHA-1 hash of the message, encrypting it using the sender's private key, and verifying it on the receiver's side with the public key, we confirmed that the signature validates the message. The signature verification process ensures that the message has not been altered and was indeed sent by the intended party, demonstrating the effectiveness of RSA in securing communications.

**References:**

[Mansour, Abdelmajid Hassan. "Analysis of RSA digital signature Key generation using strong prime." *International Journal of Computer* 24.1 (2017): 28-36](https://www.researchgate.net/profile/Abdelmajid-Emam/publication/378123668_Analysis_of_RSA_Digital_Signature_Key_Generation_using_Strong_Prime/links/65c7740e79007454976c4645/Analysis-of-RSA-Digital-Signature-Key-Generation-using-Strong-Prime.pdf)

**Experiment 9**

**AIM:** Study and Use of Diffie-Hellman Key Exchange

**Introduction :** The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

* For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.
* P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Step-by-Step explanation is as follows:

| **Alice** | **Bob** |
| --- | --- |
| Public Keys available = P, G | Public Keys available = P, G |
| Private Key Selected = a | Private Key Selected = b |
| Key generated =  𝑥=𝐺𝑎𝑚𝑜𝑑*P* | Key generated =  𝑦=𝐺𝑏𝑚𝑜𝑑𝑃 |
| Exchange of generated keys takes place | |
| Key received = y | key received = x |
| Generated Secret Key =  𝑘𝑎=𝑦𝑎𝑚𝑜𝑑𝑃 | Generated Secret Key =  𝑘𝑏=𝑥𝑏𝑚𝑜𝑑𝑃 |
| Algebraically, it can be shown that  𝑘𝑎=𝑘𝑏​ | |
| Users now have a symmetric secret key to encrypt | |

**Example :**

Step 1: Alice and Bob get public numbers P = 23, G = 9  
Step 2: Alice selected a private key a = 4 and  
 Bob selected a private key b = 3  
Step 3: Alice and Bob compute public values  
Alice: x =(9^4 mod 23) = (6561 mod 23) = 6  
 Bob: y = (9^3 mod 23) = (729 mod 23) = 16  
Step 4: Alice and Bob exchange public numbers  
Step 5: Alice receives public key y =16 and  
 Bob receives public key x = 6  
Step 6: Alice and Bob compute symmetric keys  
 Alice: ka = y^a mod p = 65536 mod 23 = 9  
 Bob: kb = x^b mod p = 216 mod 23 = 9  
Step 7: 9 is the shared secret.

**Source Code:**

import random

from sympy import isprime, primerange

def generate\_prime():

prime\_candidate = random.choice(list(primerange(50, 100)))

return prime\_candidate

def find\_primitive\_root(p):

for g in range(2, p):

if pow(g, (p - 1) // 2, p) != 1:

return g

return None

def alice\_generate\_keys(p, g):

a = random.randint(1, p - 1)

A = pow(g, a, p)

return a, A

def bob\_generate\_keys(p, g):

b = random.randint(1, p - 1)

B = pow(g, b, p)

return b, B

def alice\_compute\_shared\_secret(a, B, p):

shared\_secret\_A = pow(B, a, p)

return shared\_secret\_A

def bob\_compute\_shared\_secret(b, A, p):

shared\_secret\_B = pow(A, b, p)

return shared\_secret\_B

# Generate prime number p and find primitive root g

p = generate\_prime()

g = find\_primitive\_root(p)

print(f"Generated Prime (p): {p}")

print(f"Primitive Root (g): {g}")

# Alice and Bob generate their keys

a, A = alice\_generate\_keys(p, g)

print(f"Alice's Public Key: {A}")

b, B = bob\_generate\_keys(p, g)

print(f"Bob's Public Key: {B}")

# They exchange public keys (A and B)

shared\_secret\_A = alice\_compute\_shared\_secret(a, B, p)

shared\_secret\_B = bob\_compute\_shared\_secret(b, A, p)

print(f"Alice's Shared Secret: {shared\_secret\_A}")

print(f"Bob's Shared Secret: {shared\_secret\_B}")

# Verify if both shared secrets match

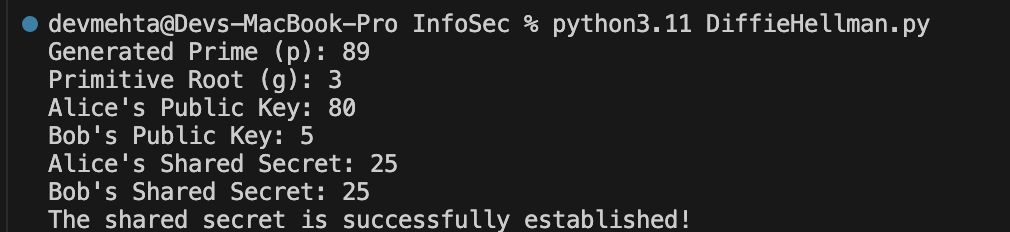
if shared\_secret\_A == shared\_secret\_B:

print("The shared secret is successfully established!")

else:

print("Something went wrong with the key exchange.")

**Output screenshot:**



**Cryptanalysis of Diffie-Hellman key exchange :**

1. **Discrete Logarithm Problem (DLP)**: The security depends on the difficulty of solving Gxmod  PG^x \mod PGxmodP for x (DLP). Large prime sizes (2048+ bits) make this infeasible.
2. **Man-in-the-Middle Attack (MitM)**: An attacker can intercept keys from both parties, creating separate secrets with each. Mitigation: Use authentication like digital signatures.
3. **Precomputation Attacks**: Attacker precomputes tables for reused primes. Mitigation: Use unique, large primes each session.
4. **Brute Force & Algorithmic Attacks**: Methods like Baby-step Giant-step can reduce attack complexity, but large primes make it impractical.
5. **Elliptic Curve Diffie-Hellman (ECDH)**: ECDH offers equivalent security with shorter keys, making brute-force and storage attacks harder.
6. **Side-Channel Attacks**: Analyzing timing, power, or emissions. Mitigation: Use constant-time operations and secure hardware.
7. **Logjam Attack**: Exploits weak 512-bit keys. Mitigation: Use strong, 2048+ bit primes and disable weak ciphers.

Using large, unique primes, incorporating authentication, and considering ECDH help ensure Diffie-Hellman remains secure.

**Conclusion:**

In conclusion, the Diffie-Hellman key exchange securely enables two parties to establish a shared secret over an insecure channel by leveraging the difficulty of the discrete logarithm problem. While it is vulnerable to certain attacks, such as man-in-the-middle and precomputation attacks, these risks can be mitigated through large prime selection, authentication methods, and adopting Elliptic Curve Diffie-Hellman for enhanced security and efficiency.

**References:**

[Li, Nan. "Research on Diffie-Hellman key exchange protocol." *2010 2nd International Conference on Computer Engineering and Technology*. Vol. 4. IEEE, 2010.](https://ieeexplore.ieee.org/abstract/document/5485276)

**Experiment 10**

**A) AIM:** Write a program to encrypt the plaintext with the given key.

E.g. plaintext GRONSFELD with the key 1234. Add 1 to G to get H (the letter 1 rank after G is H in the alphabet), then add 2 to C or E (the letter 2 ranks after C is E), and so on. Use smallest letter from plaintext as filler. In the given example last three letters are filler.

Plain letter G R O N S F E L D D D D

Key (repeated) 1 2 3 4 1 2 3 4 1 2 3 4

Cipher Letter H T R R T H H P E F G H

The encrypted message is HTRRTHHPEFGH.

Input : Plaintext (in capital letters only), key (in numbers only)

Output : ciphertext (in capital letters only)

|  |  |
| --- | --- |
| Test Case 1 | Test Case 2 |

**Source Code:**

def encrypt(plaintext,key):

ciphertext = ""

j=0

for i in plaintext:

if(i.isalpha()):

ciphertext+=chr(ord(i)+key[j])

j+=1

if(j==len(key)):

j=0

return ciphertext

def decrypt(ciphertext,key,filler\_len):

plaintext = ""

j = 0

for i in ciphertext:

if(i.isalpha()):

plaintext+=chr(ord(i)-key[j])

j+=1

if(j==len(key)):

j=0

plaintext = list(plaintext)

while(filler\_len>0):

plaintext.pop()

filler\_len-=1

return "".join(plaintext)

plain\_text = input("Enter the plain text:")

min\_letter = input("Enter the minimum letter:")

plain\_text = plain\_text.upper()

min\_letter = min\_letter.upper()

key = int(input("Enter the key:"))

key = [int(x) for x in str(key)]

filler\_len = len(key)-len(plain\_text)%len(key)

if(len(plain\_text)%len(key)!=0):

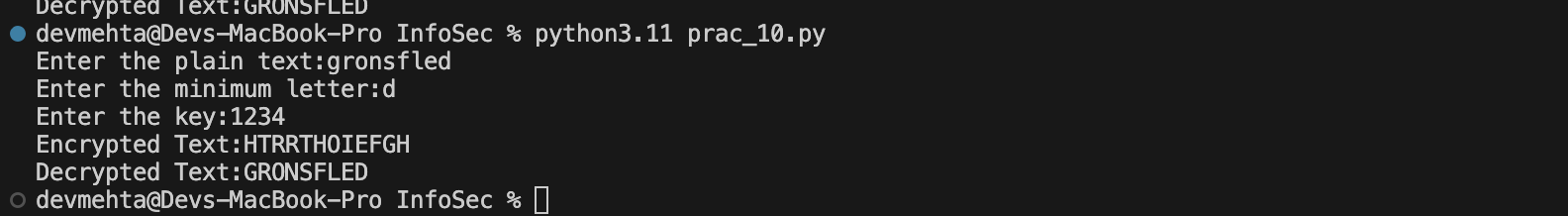
plain\_text += min\_letter.upper() \* filler\_len

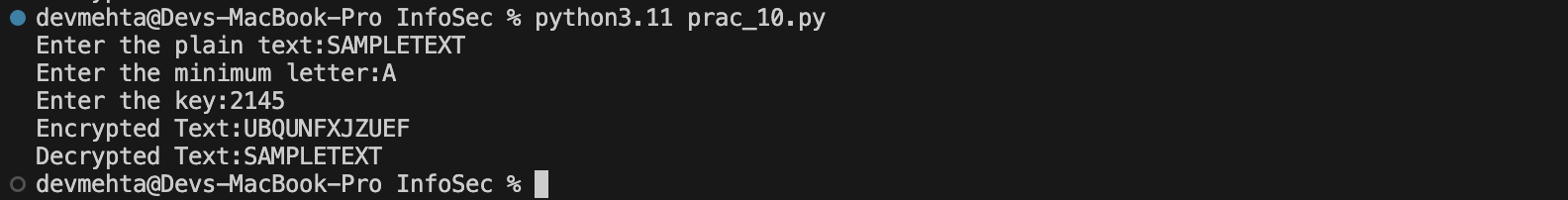
cipher\_text = encrypt(plain\_text,key)

print("Encrypted Text:"+cipher\_text)

print("Decrypted Text:"+decrypt(cipher\_text,key,filler\_len))

**Output**

****

****

**B) AIM:** Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Plain letter | R | A | G |  | B | A | B | Y |
| word i | 1 | 1 | 1 |  | 2 | 2 | 2 | 2 |
| letter j | 0 | 1 | 2 |  | 0 | 1 | 2 | 3 |
| i+j | 1 | 2 | 3 |  | 2 | 3 | 4 | 5 |
| Cipher letter | S | C | J |  | D | D | F | D |

Input : Plaintext (in capital letters only, upto nine words only)

Output : ciphertext (in capital letters only)

Note : Students can use string.h if required.

|  |  |
| --- | --- |
| Test case 1 | Test Case 2 |

Output : ciphertext (in capital letters only)

**Source Code:**

def encrypt(plaintext):

word\_count = 1

letter\_count = 0

ciphertext = ""

key = []

for i in plaintext:

if (i != " "):

val = word\_count + letter\_count

key.append(val)

ciphertext+=chr((ord(i) - ord("A") + val)%26 + ord("A"))

letter\_count += 1

else:

key.append(" ")

ciphertext+=" "

word\_count += 1

letter\_count = 0

return ciphertext,key

def decrypt(ciphertext,key):

plaintext = ""

key\_index = 0

for i in range(len(ciphertext)):

if ciphertext[i] == " ":

plaintext += " "

key\_index += 1

continue

val = key[key\_index]

if(val!=" "):

plaintext += chr(((ord(ciphertext[i]) - ord('A') - val) % 26) + ord('A'))

key\_index += 1

return plaintext

plain\_text = input("Enter the plain text:")

plain\_text = plain\_text.upper()

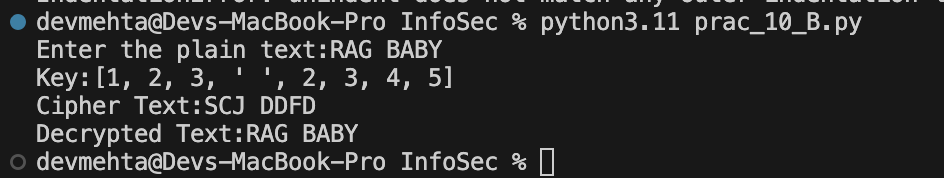
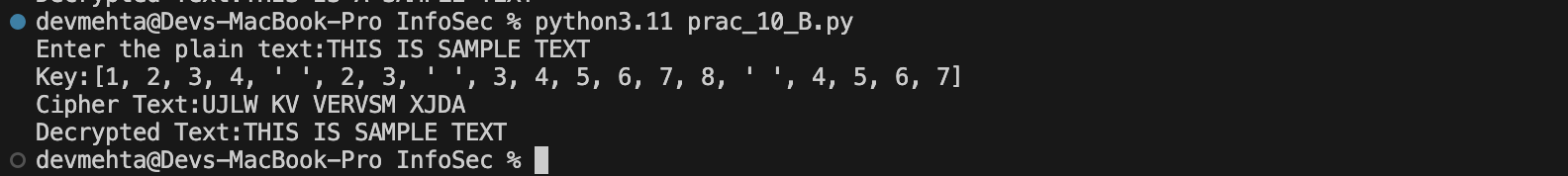
cipher\_text,key = encrypt(plain\_text)

print("Key:"+"".join(str(key)))

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

print("Decrypted Text:"+decrypt(cipher\_text,key))

**Output:**

****

**Conclusion:**

We have successfully implemented both the custom ciphers using the given algorithms in the questions.