import hashlib

input\_string = input("Enter a string to hash: ")

hash\_object = hashlib.sha1(input\_string.encode())

hash\_hex = hash\_object.hexdigest()

print("SHA-1 Hash:", hash\_hex)

import hashlib

input\_string = input("Enter a string: ")

hash\_object = hashlib.md5(input\_string.encode())

md5\_hash = hash\_object.hexdigest()

print("MD5 Hash:", md5\_hash)

import random

import math

def simple\_columnar\_transposition(message, key, num\_rows, num\_cols):

padded\_message = message.ljust(num\_rows \* num\_cols)

matrix = [list(padded\_message[i \* num\_cols:(i + 1) \* num\_cols]) for i in range(num\_rows)]

return ''.join(matrix[r][i] for i in key for r in range(num\_rows) if matrix[r][i] != " ")

def advanced\_columnar\_transposition(message, key, num\_rows, num\_cols, rounds):

for \_ in range(rounds):

message = simple\_columnar\_transposition(message, key, num\_rows, num\_cols)

return message

def generate\_random\_key(num\_cols):

key = list(range(num\_cols))

random.shuffle(key)

return key

def main():

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

num\_cols = random.randint(2, 10)

num\_rows = math.ceil(len(message) / num\_cols)

key = generate\_random\_key(num\_cols)

encrypted\_message = simple\_columnar\_transposition(message, key, num\_rows, num\_cols)

print(f"Simple Columnar Transposition: {encrypted\_message}")

rounds = random.randint(1, 5)

advanced\_encrypted\_message = advanced\_columnar\_transposition(message, key, num\_rows, num\_cols, rounds)

print(f"Advanced Columnar Transposition (Rounds: {rounds}): {advanced\_encrypted\_message}")

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

    main()

import random

import math

def simple\_columnar\_transposition(message, key, num\_rows, num\_cols):

padded\_message = message.ljust(num\_rows \* num\_cols)

matrix = []

x = 0

for i in range(num\_rows):

row = []

for j in range(num\_cols):

row.append(padded\_message[x])

x += 1

matrix.append(row)

encrypted = []

for i in key:

for r in range(num\_rows):

if matrix[r][i] != " ":

encrypted.append(matrix[r][i]) # No need for index correction since key starts from 0

return ''.join(encrypted)

def advanced\_columnar\_transposition(message, key, num\_rows, num\_cols, rounds):

transposed\_message = message

for \_ in range(rounds):

transposed\_message = simple\_columnar\_transposition(transposed\_message, key, num\_rows, num\_cols)

return transposed\_message

def generate\_random\_key(num\_cols):

key = list(range(num\_cols))

random.shuffle(key)

return key

def main():

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

num\_cols = random.randint(2, 10) # Random number of columns between 2 and 10

num\_rows = math.ceil(len(message) / num\_cols)

key = generate\_random\_key(num\_cols)

encrypted\_message = simple\_columnar\_transposition(message, key, num\_rows, num\_cols)

print(f"Simple Columnar Transposition: {encrypted\_message}")

rounds = random.randint(1, 5)

advanced\_encrypted\_message = advanced\_columnar\_transposition(message, key, num\_rows, num\_cols, rounds)

print(f"Advanced Columnar Transposition (Rounds: {rounds}): {advanced\_encrypted\_message}")

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

main()

# Euclidean algorithm

def gcd(a, b):

if a == 0:

return b

return gcd(b % a, a)

# Driver code for GCD

a, b = 10, 15

print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 35, 10

print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 31, 2

print(f"GCD({a}, {b}) = {gcd(a, b)}")

# Extended Euclidean algorithm

def gcd\_extended(a, b):

if a == 0:

return b, 0, 1

gcd, x1, y1 = gcd\_extended(b % a, a)

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

y = x1

return gcd, x, y

# Driver Program for Extended Euclidean Algorithm

a = 35

b = 15

gcd, x, y = gcd\_extended(a, b)

print(f"gcd({a}, {b}) = {gcd}, x = {x}, y = {y}")

**1. Cryptlib**

* **Description**: An open-source, cross-platform toolkit for cryptographic security in applications. It supports encryption, digital signatures, secure data transport, and more.
* **License**: Sleepycat License (compatible with GPL) or a commercial license.
* **Recent Updates**: Added support for modern algorithms like AES-GCM and Curve25519, and improved secure multi-party computation.
* **Use Cases**: Suitable for mobile, web, and server applications needing embedded cryptographic functions.

**2. Crypto++**

* **Description**: A free, open-source C++ library offering a comprehensive suite of cryptographic algorithms, including AES, DES, RSA, DSA, ECC, and hashing functions.
* **License**: Public domain software.
* **Recent Updates**: Enhanced to support ChaCha20-Poly1305, BLAKE3 hashing, and EdDSA.
* **Use Cases**: Widely used in security-sensitive applications requiring high performance and portability.

**3. LibreSSL**

* **Description**: An open-source TLS protocol implementation by the OpenBSD project, forked from OpenSSL with the goal of simplifying and securing the codebase.
* **License**: ISC License.
* **Recent Updates**: Focused on removing deprecated algorithms, supporting modern standards like TLS 1.3, SHA-3, and Ed25519, and improving performance.
* **Use Cases**: Primarily used for secure communication in UNIX-like systems.

**4. OpenSSL**

* **Description**: A popular open-source library implementing SSL and TLS protocols, along with a variety of cryptographic functions.
* **License**: Apache License 2.0.
* **Recent Updates**: OpenSSL 3.0 introduced a modular "Provider" architecture, post-quantum cryptography support, and updated TLS 1.3.
* **Use Cases**: Used for HTTPS, VPNs, and general cryptographic operations.

**5. PyCryptodome**

* **Description**: A Python library of cryptographic primitives, forked from PyCrypto, supporting AES, RSA, DSA, and various hash functions.
* **License**: MIT License.
* **Recent Updates**: Added hardware-accelerated encryption and improved support for AES-GCM and RSA-PSS signatures.
* **Use Cases**: Ideal for Python-based projects requiring cryptographic operations, such as file encryption and digital signatures.

**6. Tink (by Google)**

* **Description**: A Google-developed open-source cryptographic library offering simple, secure APIs for cryptographic functions and key management.
* **License**: Apache License 2.0.
* **Recent Updates**: Enhanced support for hybrid encryption, cloud KMS integration, and AES-SIV.
* **Use Cases**: Useful for secure application development, cloud-based encryption, and secure data storage.

# Python3 program for the RC4 algorithm

plain\_text = "001010010010"

key = "101001000001"

n = 3 # Number of bits to consider at a time

S = [i for i in range(0, 2\*\*n)]

key\_list = [int(key[i:i + n], 2) for i in range(0, len(key), n)]

pt = [int(plain\_text[i:i + n], 2) for i in range(0, len(plain\_text), n)]

# Adjust key\_list length to match S

if len(S) != len(key\_list):

key\_list += key\_list[:len(S) - len(key\_list)]

# Key Scheduling Algorithm (KSA)

def KSA(S, key\_list):

j = 0

for i in range(len(S)):

j = (j + S[i] + key\_list[i]) % len(S)

S[i], S[j] = S[j], S[i]

return S

# Pseudo-Random Generation Algorithm (PRGA)

def PRGA(S, text\_length):

i = j = 0

key\_stream = []

for \_ in range(text\_length):

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

j = (j + S[i]) % len(S)

S[i], S[j] = S[j], S[i]

t = (S[i] + S[j]) % len(S)

key\_stream.append(S[t])

return key\_stream

# XOR between generated key stream and input text

def XOR(input\_text, key\_stream):

return [input\_text[i] ^ key\_stream[i] for i in range(len(input\_text))]

# Encryption function

def encryption():

print("Plain text:", plain\_text)

print("Key:", key)

S\_init = KSA(S[:], key\_list) # Initial permutation

key\_stream = PRGA(S\_init, len(pt))

cipher\_text = XOR(pt, key\_stream)

encrypted\_to\_bits = ''.join(f"{bin(c)[2:]:0>{n}}" for c in cipher\_text)

print("Cipher text:", encrypted\_to\_bits)

return cipher\_text

# Decryption function

def decryption(cipher\_text):

S\_init = KSA(S[:], key\_list)

key\_stream = PRGA(S\_init, len(pt))

original\_text = XOR(cipher\_text, key\_stream)

decrypted\_to\_bits = ''.join(f"{bin(p)[2:]:0>{n}}" for p in original\_text)

print("Decrypted text:", decrypted\_to\_bits)

# Driver Code

cipher\_text = encryption()

print("---------------------------------------------------------")

decryption(cipher\_text)