```
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)
```