**Ceaser cipher:**

def caesar\_encrypt(text, shift):

encrypted\_text = ""

for char in text:

if char.isalpha():

is\_upper = char.isupper()

char = char.lower()

shifted = (ord(char) - ord('a') + shift) % 26

shifted\_char = chr(ord('a') + shifted)

if is\_upper:

shifted\_char = shifted\_char.upper()

encrypted\_text += shifted\_char

else:

encrypted\_text += char

return encrypted\_text

def caesar\_decrypt(encrypted\_text, shift):

decrypted\_text = ""

for char in encrypted\_text:

if char.isalpha():

is\_upper = char.isupper()

char = char.lower()

shifted = (ord(char) - ord('a') - shift) % 26

shifted\_char = chr(ord('a') + shifted)

if is\_upper:

shifted\_char = shifted\_char.upper()

decrypted\_text += shifted\_char

else:

decrypted\_text += char

return decrypted\_text

def brute\_force\_decrypt(ciphertext):

for shift in range(26):

decrypted\_text = caesar\_decrypt(ciphertext, shift)

print(f"Shift {shift}: {decrypted\_text}")

# Example usage:

text = "Hello, World!"

shift = 3

encrypted\_text = caesar\_encrypt(text, shift)

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

print("Brute Force Decryption:")

brute\_force\_decrypt(encrypted\_text)

**Railfence cipher:**

def rail\_fence\_encrypt(text, key):

return ''.join(text[i::2\*(key-1)] for i in range(key))

plaintext = "HELLOWORLD"

key = 3

encrypted\_message = rail\_fence\_encrypt(plaintext, key)

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

def rail\_fence\_decrypt(text, key):

indexes = [i for i in range(key)] + [i for i in range(key-2, 0, -1)]

return ''.join(text[i::2\*(key-1)] for i in indexes)

encrypted\_message = "HOLELWRDLOO"

key = 3

decrypted\_text = rail\_fence\_decrypt(encrypted\_message, key)

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

**Vernam cipher:**

def vernam\_encrypt(message, key):

if len(message) != len(key):

raise ValueError("Message and key must have the same length")

encrypted\_message = ""

for i in range(len(message)):

encrypted\_bit = str(int(message[i]) ^ int(key[i])) # XOR operation

encrypted\_message += encrypted\_bit

return encrypted\_message

def vernam\_decrypt(encrypted\_message, key):

if len(encrypted\_message) != len(key):

raise ValueError("Encrypted message and key must have the same length")

decrypted\_message = ""

for i in range(len(encrypted\_message)):

decrypted\_bit = str(int(encrypted\_message[i]) ^ int(key[i])) # XOR operation

decrypted\_message += decrypted\_bit

return decrypted\_message

# Example usage:

message = "110010101"

key = "100110010"

encrypted\_message = vernam\_encrypt(message, key)

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

decrypted\_message = vernam\_decrypt(encrypted\_message, key)

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

**Deffie Helman :**

p = 23 # A prime number

g = 5 # A primitive root modulo p

# Alice generates a private key

def generate\_private\_key():

return int(input("Enter a private key (an integer between 1 and p-1): "))

# Validate private keys

def validate\_private\_key(key):

return 1 <= key < p

# Input and validate private keys for Alice and Bob

private\_key\_alice = 0

while not validate\_private\_key(private\_key\_alice):

private\_key\_alice = generate\_private\_key()

private\_key\_bob = 0

while not validate\_private\_key(private\_key\_bob):

private\_key\_bob = generate\_private\_key()

# Alice and Bob exchange their public keys

public\_key\_alice = (g \*\* private\_key\_alice) % p

public\_key\_bob = (g \*\* private\_key\_bob) % p

# Alice and Bob calculate the shared secret key

shared\_secret\_key\_alice = (public\_key\_bob \*\* private\_key\_alice) % p

shared\_secret\_key\_bob = (public\_key\_alice \*\* private\_key\_bob) % p

print(f"Public key Alice: {public\_key\_alice}")

print(f"Public key Bob: {public\_key\_bob}")

print(f"Shared secret key Alice: {shared\_secret\_key\_alice}")

print(f"Shared secret key Bob: {shared\_secret\_key\_bob}")

# Both Alice and Bob now have the same shared secret key

**DES :**

(Way 1)

import secrets

import binascii

def generate\_des\_key():

# Generate a random 56-bit key

key = secrets.token\_bytes(7)

return key

# Convert the generated key to a hexadecimal string for display

des\_key = generate\_des\_key()

hex\_key = binascii.hexlify(des\_key).decode()

print(f"Generated DES Key: {hex\_key}")

(Way 2)

import random

def generate\_des\_key():

# Initialize an empty 56-bit key

key = []

# Generate 56 random bits (0 or 1)

for \_ in range(56):

bit = random.randint(0, 1)

key.append(bit)

# Convert the list of bits to a string

key\_str = ''.join(map(str, key))

# Convert the binary string to bytes

key\_bytes = bytes(int(key\_str[i:i+8], 2) for i in range(0, len(key\_str), 8))

return key\_bytes

# Generate and display a DES key

des\_key = generate\_des\_key()

hex\_key = des\_key.hex()

print(f"Generated DES Key: {hex\_key}")

**RSA :**from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import padding

# Key generation

private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048 # Choose an appropriate key size

)

# Serialization

private\_pem = private\_key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.PKCS8,

encryption\_algorithm=serialization.NoEncryption()

)

public\_key = private\_key.public\_key()

public\_pem = public\_key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

# Encryption

message = b"Hello, RSA!"

ciphertext = public\_key.encrypt(

message,

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

# Decryption

plaintext = private\_key.decrypt(

ciphertext,

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

print("Original Message:", message.decode())

print("Decrypted Message:", plaintext.decode())

(Way 2)

import random

# Function to check if a number is prime

def is\_prime(num):

if num <= 1:

return False

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

if num % i == 0:

return False

return True

# Function to find the greatest common divisor (GCD) of two numbers

def gcd(a, b):

while b:

a, b = b, a % b

return a

# Function to find the modular multiplicative inverse

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

d = 0

x1, x2 = 0, 1

y1, temp\_phi\_n = 1, phi\_n

while e > 0:

temp1 = temp\_phi\_n // e

temp2 = temp\_phi\_n - temp1 \* e

temp\_phi\_n = e

e = temp2

x = x2 - temp1 \* x1

y = d - temp1 \* y1

x2, x1 = x1, x

d, y1 = y1, y

if temp\_phi\_n == 1:

if d < 0:

d += phi\_n

else:

d = -1 # No modular inverse exists

return d

# Key Generation

bits = 1024 # Choose the key size (e.g., 1024 bits)

min\_prime = 2 \*\* (bits - 1)

max\_prime = 2 \*\* bits - 1

# Find two prime numbers

p = random.randint(min\_prime, max\_prime)

while not is\_prime(p):

p = random.randint(min\_prime, max\_prime)

q = random.randint(min\_prime, max\_prime)

while not is\_prime(q) or q == p:

q = random.randint(min\_prime, max\_prime)

n = p \* q

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

# Find the public exponent

e = 65537 # A common choice for the public exponent

# Find the private exponent

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

# Encryption

plaintext = "HELLO"

plaintext = int.from\_bytes(plaintext.encode(), byteorder='big')

ciphertext = pow(plaintext, e, n)

# Decryption

decrypted\_text = pow(ciphertext, d, n)

decrypted\_text = decrypted\_text.to\_bytes((decrypted\_text.bit\_length() + 7) // 8, byteorder='big').decode()

# Display results

print("Original Message:", "HELLO")

print("Encrypted Message:", ciphertext)

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

(Way 3)

import math

def gcd(a, h):

temp = 0

while(1):

temp = a % h

if (temp == 0):

return h

a = h

h = temp

p = int(input("enter the prime number p"))

q = int((input("Enter the prime number q")))

n = p\*q

e = 2

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

while (e < phi):

# e must be co-prime to phi and

# smaller than phi.

if(gcd(e, phi) == 1):

break

else:

e = e+1

def intcheck(phi, e):

k = 1

while True:

d = (1 + (k \* phi)) / e

if d == int(d):

return int(d)

k += 1

# Message to be encrypted

d=intcheck(phi,e)

msg = int(input("Enter the plaintext: "))

print("Message data = ", msg)

# Encryption c = (msg ^ e) % n

c = pow(msg, e)

c = math.fmod(c, n)

print("Encrypted data = ", c)

# Decryption m = (c ^ d) % n

m = pow(c, d)

m = math.fmod(m, n)

print("Original Message Sent = ", m)

**SHA:**

import hashlib

def enc(message):

md5 = hashlib.md5()

md5.update(message.encode('utf-8'))

return md5.hexdigest()

def enc1(message):

sha1 = hashlib.sha1()

sha1.update(message.encode('utf-8'))

return sha1.hexdigest()

message = input("Enter the input string")

print(f"MD5 Hashing String:{enc(message)}")

print(f"SHA-I Hashed String:{enc1(message)}")

**MD5:**

import hashlib

# String to hash

input\_string = "Hello, MD5!"

# Create a hashlib object for MD5

md5\_hash = hashlib.md5()

# Update the hash object with the bytes of the input string

md5\_hash.update(input\_string.encode('utf-8'))

# Get the hexadecimal representation of the hash

hashed\_string = md5\_hash.hexdigest()

print("MD5 hash:", hashed\_string)