blow fish

```
from Crypto.Cipher import Blowfish
from Crypto.Util.Padding import pad, unpad
import base64

def get_user_input():
    key = input("Enter the key (in hex format, e.g., '0123456789abcdef'): ")
    plaintext = input("Enter the plaintext message: ")

    # Convert the hex key to bytes
    key_bytes = bytes.fromhex(key)
    # Ensure the key length is appropriate for Blowfish (up to 56 bytes)
    if len(key_bytes) > 56:
        raise ValueError("Key length should be between 1 and 56 bytes.")

    return key_bytes, plaintext.encode()

def encrypt_message(key_bytes, plaintext_bytes):
    cipher = Blowfish.new(key_bytes, Blowfish.MODE_CBC)
    # Pad plaintext to be a multiple of block size (8 bytes for Blowfish)
    padded_plaintext = pad(plaintext_bytes, Blowfish.block_size)
    ciphertext_bytes = cipher.encrypt(padded_plaintext)
    # Encode the result as base64 to make it easy to display
    iv = base64.b64encode(cipher.iv).decode('utf-8')
    ciphertext = base64.b64encode(ciphertext_bytes).decode('utf-8')
    return iv, ciphertext

def decrypt_message(key_bytes, iv, ciphertext):
    cipher = Blowfish.new(key_bytes, Blowfish.MODE_CBC, iv=base64.b64decode(iv))
    ciphertext_bytes = base64.b64decode(ciphertext)
    padded_plaintext = cipher.decrypt(ciphertext_bytes)
    # Unpad plaintext
    plaintext_bytes = unpad(padded_plaintext, Blowfish.block_size)
    return plaintext_bytes.decode()

def main():
    key_bytes, plaintext_bytes = get_user_input()

    # Encrypt the message
    iv, ciphertext = encrypt_message(key_bytes, plaintext_bytes)
    print(f"IV (Base64): {iv}")
    print(f"Ciphertext (Base64): {ciphertext}")

    # Decrypt the message
    decrypted_message = decrypt_message(key_bytes, iv, ciphertext)
    print(f"Decrypted message: {decrypted_message}")

if __name__ == "__main__":
    main()
```

Caesar cipher def caesar_cipher(text, shift):

```
    result = ""
    for char in text:
        if char.isalpha():
            # Determine whether it's an uppercase or lowercase letter
            base = ord('A') if char.isupper() else ord('a')
            shifted_char = chr((ord(char) - base + shift) % 26 + base)
            result += shifted_char
        else:
            # Non-alphabetic characters remain unchanged
            result += char
    return result
```

```
def main():
    try:
        plaintext = input("Enter the text you want to encrypt: ")
        shift_amount = int(input("Enter the shift value (positive for
encryption, negative for decryption): "))
        encrypted_text = caesar_cipher(plaintext, shift_amount)
        print(f"Encrypted text: {encrypted_text}")
    except ValueError:
        print("Invalid input. Please enter a valid shift value (an integer).")

if __name__ == "__main__":
    main()
```

cbc

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

# Step 1: AES Encryption using CBC mode
def encrypt_cbc(key, plaintext, iv):
    # Padding the plaintext to be a multiple of the block size (AES block size
is 128 bits)
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plaintext.encode('utf-8')) + padder.finalize()

    # Create a cipher object
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
    encryptor = cipher.encryptor()

    # Perform the encryption
    ciphertext = encryptor.update(padded_data) + encryptor.finalize()
    return ciphertext

# Step 2: AES Decryption using CBC mode
def decrypt_cbc(key, ciphertext, iv):
    # Create a cipher object for decryption
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
    decryptor = cipher.decryptor()

    # Perform the decryption
    padded_plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    # Unpadding the plaintext
    unpadder = padding.PKCS7(128).unpadder()
    plaintext = unpadder.update(padded_plaintext) + unpadder.finalize()
    return plaintext.decode('utf-8')

# Get input from user
plaintext = input("Enter a message to encrypt: ")

# Step 3: Generate a random key and IV (Initialization Vector)
key = os.urandom(32)  # AES-256, 32 bytes key
iv = os.urandom(16)   # AES block size is 16 bytes

# Step 4: Encrypt the message
ciphertext = encrypt_cbc(key, plaintext, iv)
print(f"Encrypted message (in bytes): {ciphertext}")
```

```
# Step 5: Decrypt the message
decrypted_message = decrypt_cbc(key, ciphertext, iv)
print(f"Decrypted message: {decrypted_message}")
```

cfb

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

# Step 1: AES Encryption using CFB mode
def encrypt_cfb(key, plaintext, iv):
    # Create a cipher object using AES in CFB mode
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv),
backend=default_backend())
    encryptor = cipher.encryptor()

    # Perform the encryption
    ciphertext = encryptor.update(plaintext.encode('utf-8')) +
encryptor.finalize()
    return ciphertext

# Step 2: AES Decryption using CFB mode
def decrypt_cfb(key, ciphertext, iv):
    # Create a cipher object for decryption
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv),
backend=default_backend())
    decryptor = cipher.decryptor()

    # Perform the decryption
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    return plaintext.decode('utf-8')

# Get input from user
plaintext = input("Enter a message to encrypt: ")

# Step 3: Generate a random key and IV (Initialization Vector)
key = os.urandom(32)  # AES-256, 32 bytes key
iv = os.urandom(16)   # AES block size is 16 bytes

# Step 4: Encrypt the message
ciphertext = encrypt_cfb(key, plaintext, iv)
print(f"Encrypted message (in bytes): {ciphertext}")

# Step 5: Decrypt the message
decrypted_message = decrypt_cfb(key, ciphertext, iv)
print(f"Decrypted message: {decrypted_message}")
```

des

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
import base64

def get_user_input():
    key = input("Enter the 8-byte key (in hex format, e.g., '12345678'): ")
    plaintext = input("Enter the plaintext message: ")

    # Convert the hex key to bytes
    key_bytes = bytes.fromhex(key)
    # Ensure the key is exactly 8 bytes for DES
    if len(key_bytes) != 8:
```

```
        raise ValueError("Key must be exactly 8 bytes long.")

    return key_bytes, plaintext.encode()

def encrypt_message(key_bytes, plaintext_bytes):
    cipher = DES.new(key_bytes, DES.MODE_CBC)
    # Pad plaintext to be multiple of block size (8 bytes for DES)
    padded_plaintext = pad(plaintext_bytes, DES.block_size)
    ciphertext_bytes = cipher.encrypt(padded_plaintext)
    # Encode the result as base64 to make it easy to display
    iv = base64.b64encode(cipher.iv).decode('utf-8')
    ciphertext = base64.b64encode(ciphertext_bytes).decode('utf-8')
    return iv, ciphertext

def decrypt_message(key_bytes, iv, ciphertext):
    cipher = DES.new(key_bytes, DES.MODE_CBC, iv=base64.b64decode(iv))
    ciphertext_bytes = base64.b64decode(ciphertext)
    padded_plaintext = cipher.decrypt(ciphertext_bytes)
    # Unpad plaintext
    plaintext_bytes = unpad(padded_plaintext, DES.block_size)
    return plaintext_bytes.decode()

def main():
    key_bytes, plaintext_bytes = get_user_input()

    # Encrypt the message
    iv, ciphertext = encrypt_message(key_bytes, plaintext_bytes)
    print(f"IV (Base64): {iv}")
    print(f"Ciphertext (Base64): {ciphertext}")

    # Decrypt the message
    decrypted_message = decrypt_message(key_bytes, iv, ciphertext)
    print(f"Decrypted message: {decrypted_message}")

if __name__ == "__main__":
    main()

differ

import random

# Step 1: Diffie-Hellman Key Exchange
def diffie_hellman(p, g, private_key):
    # Compute public key
    public_key = pow(g, private_key, p)
    return public_key

# Step 2: Compute the shared secret
def compute_shared_secret(public_key, private_key, p):
    # Compute the shared secret
    shared_secret = pow(public_key, private_key, p)
    return shared_secret

# Get input from user
p = int(input("Enter a large prime number (p): "))
g = int(input("Enter a generator (g): "))

# Private keys (chosen randomly by both parties)
private_key_A = random.randint(1, p - 1)
private_key_B = random.randint(1, p - 1)

# Step 3: Compute public keys for both parties
public_key_A = diffie_hellman(p, g, private_key_A)
public_key_B = diffie_hellman(p, g, private_key_B)
```

```
print(f"Public key of Party A: {public_key_A}")
print(f"Public key of Party B: {public_key_B}")

# Step 4: Compute the shared secret for both parties
shared_secret_A = compute_shared_secret(public_key_B, private_key_A, p)
shared_secret_B = compute_shared_secret(public_key_A, private_key_B, p)

# Step 5: Display the shared secrets
print(f"Shared secret (computed by Party A): {shared_secret_A}")
print(f"Shared secret (computed by Party B): {shared_secret_B}")

# Check if the shared secrets match
if shared_secret_A == shared_secret_B:
    print("Shared secret successfully established!")
else:
    print("Shared secret mismatch.")
```

elagamal

```
import random

# Step 1: Modular exponentiation (fast computation of (base^exp) % mod)
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:  # If exp is odd
            result = (result * base) % mod
        exp = exp >> 1    # exp = exp // 2
        base = (base * base) % mod
    return result

# Step 2: Function to find modular inverse
def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Step 3: ElGamal Key Generation
def generate_keys(p, g):
    x = random.randint(1, p - 2)  # Private key x (1 < x < p-1)
    y = mod_exp(g, x, p)          # Public key y = g^x mod p
    return (p, g, y), x

# Step 4: ElGamal Signature Generation
def sign(message, private_key, p, g):
    x = private_key
    k = random.randint(1, p - 2)
    while gcd(k, p - 1) != 1:
        k = random.randint(1, p - 2)

    r = mod_exp(g, k, p)
    k_inv = mod_inverse(k, p - 1)
    s = (k_inv * (message - x * r)) % (p - 1)
    return r, s

# Step 5: ElGamal Signature Verification
def verify(message, signature, public_key):
```

```
    p, g, y = public_key
    r, s = signature
    if not (0 < r < p):
        return False
    v1 = mod_exp(g, message, p)
    v2 = (mod_exp(y, r, p) * mod_exp(r, s, p)) % p
    return v1 == v2

# Step 6: GCD Function
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

# Get input from user
p = int(input("Enter a large prime number (p): "))
g = int(input("Enter a generator (g): "))
message = int(input("Enter a message as an integer: "))

# Key Generation
public_key, private_key = generate_keys(p, g)
print(f"Public Key: {public_key}")
print(f"Private Key: {private_key}")

# Signature Generation
signature = sign(message, private_key, p, g)
print(f"Signature: {signature}")

# Signature Verification
verification = verify(message, signature, public_key)
print(f"Signature valid: {verification}")
```

hash fun

```
import hashlib

# Step 1: Hashing Function
def hash_message(message, algorithm):
    # Select the algorithm
    if algorithm == 'sha256':
        hasher = hashlib.sha256()
    elif algorithm == 'md5':
        hasher = hashlib.md5()
    else:
        raise ValueError("Unsupported algorithm. Use 'sha256' or 'md5'.")

    # Encode the message and update the hasher
    hasher.update(message.encode('utf-8'))
    return hasher.hexdigest()

# Get input from user
message = input("Enter a message to hash: ")
algorithm = input("Choose hashing algorithm ('sha256' or 'md5'): ")

# Generate hash
hashed_message = hash_message(message, algorithm)
print(f"Hashed message using {algorithm}: {hashed_message}")
```

cmac

```
import numpy as np

def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None

def hill_cipher(text, key_matrix, mode='encrypt'):
    text = text.lower().replace(" ", "")
    n = len(key_matrix)
    padded_text = text + 'x' * (n - len(text) % n)
    result = ''

    for i in range(0, len(padded_text), n):
        vector = [ord(c) - ord('a') for c in padded_text[i:i+n]]
        if mode == 'decrypt':
            determinant = int(np.round(np.linalg.det(key_matrix)))
            inverse_matrix = mod_inverse(determinant % 26, 26) *
np.round(determinant * np.linalg.inv(key_matrix)).astype(int) % 26
            key_matrix = inverse_matrix

        transformed_vector = np.dot(key_matrix, vector) % 26
        result += ''.join(chr(int(num) + ord('a')) for num in
transformed_vector)

    return result

# Example usage
text = input("Enter a string: ")
key_matrix = np.array([[3, 3], [2, 5]])

encrypted_text = hill_cipher(text, key_matrix)
decrypted_text = hill_cipher(encrypted_text, key_matrix, mode='decrypt')

print("Encrypted text:", encrypted_text)
print("Decrypted text:", decrypted_text)

monoalphabetic

def create_monoalphabetic_cipher():
    # Define the substitution mapping (you can customize this)
    monoalpha_cipher = {
        'a': 'm', 'b': 'n', 'c': 'b', 'd': 'v', 'e': 'c',
        'f': 'x', 'g': 'z', 'h': 'a', 'i': 's', 'j': 'd',
        'k': 'f', 'l': 'g', 'm': 'h', 'n': 'j', 'o': 'k',
        'p': 'l', 'q': 'p', 'r': 'o', 's': 'i', 't': 'u',
        'u': 'y', 'v': 't', 'w': 'r', 'x': 'e', 'y': 'w',
        'z': 'q', ' ': ' ',  # Space remains unchanged
    }
    return monoalpha_cipher

def encrypt_with_monoalphabetic(message, monoalpha_cipher):
    encrypted_message = []
    for char in message.lower():  # Convert to lowercase for consistency
        encrypted_char = monoalpha_cipher.get(char, char)
        encrypted_message.append(encrypted_char)
    return ''.join(encrypted_message)

def main():
    try:
        plaintext = input("Enter the text you want to encrypt: ")
        monoalpha_cipher = create_monoalphabetic_cipher()
        encrypted_text = encrypt_with_monoalphabetic(plaintext,
```

```
monoalpha_cipher)
        print(f"Encrypted text: {encrypted_text}")
    except KeyboardInterrupt:
        print("\nOperation aborted by user.")

if __name__ == "__main__":
    main()


play fair

# Function to prepare the text (removes spaces, handles 'J' as 'I')
def prepare_text(text):
    text = text.upper().replace('J', 'I').replace(' ', '')
    result = ""
    i = 0
    while i < len(text):
        result += text[i]
        if i + 1 < len(text) and text[i] == text[i + 1]:
            result += 'X'
        if i + 1 < len(text):
            result += text[i + 1]
        i += 2
    if len(result) % 2 != 0:
        result += 'X'
    return result

# Create the Playfair cipher square (5x5)
def create_square(key):
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    key = "".join(sorted(set(key.upper().replace('J', 'I')), key=lambda x:
key.index(x)))
    square = ''.join([c for c in key if c in alphabet] + [c for c in alphabet if
c not in key])
    return [list(square[i:i + 5]) for i in range(0, 25, 5)]

# Find position of a letter in the square
def find_position(letter, square):
    for row in range(5):
        if letter in square[row]:
            return row, square[row].index(letter)
    return None

# Encrypt a digraph (pair of letters)
def encrypt_digraph(digraph, square):
    row1, col1 = find_position(digraph[0], square)
    row2, col2 = find_position(digraph[1], square)

    if row1 == row2:
        return square[row1][(col1 + 1) % 5] + square[row2][(col2 + 1) % 5]
    elif col1 == col2:
        return square[(row1 + 1) % 5][col1] + square[(row2 + 1) % 5][col2]
    else:
        return square[row1][col2] + square[row2][col1]

# Playfair cipher encryption
def playfair_encrypt(plaintext, key):
    square = create_square(key)
    plaintext = prepare_text(plaintext)
    ciphertext = ""
    for i in range(0, len(plaintext), 2):
        ciphertext += encrypt_digraph(plaintext[i:i + 2], square)
    return ciphertext
```

```
# Main program
key = input("Enter the key: ")
plaintext = input("Enter the plaintext: ")
ciphertext = playfair_encrypt(plaintext, key)
print("Encrypted message:", ciphertext)
```

rail fence

```
# Function to encrypt using Rail Fence Cipher
def encrypt_rail_fence(text, key):
    fence = [''] * key
    rail = 0
    direction = 1  # 1 means moving down, -1 means moving up

    for char in text:
        fence[rail] += char
        rail += direction

        if rail == 0 or rail == key - 1:
            direction *= -1

    return ''.join(fence)

# Function to decrypt using Rail Fence Cipher
def decrypt_rail_fence(cipher, key):
    fence = [['\n'] * len(cipher) for _ in range(key)]
    rail = 0
    direction = 1
    idx = 0

    # Mark the positions to fill the characters
    for char in cipher:
        fence[rail][idx] = '*'
        idx += 1
        rail += direction
        if rail == 0 or rail == key - 1:
            direction *= -1

    # Fill the rail fence with cipher text
    idx = 0
    for i in range(key):
        for j in range(len(cipher)):
            if fence[i][j] == '*':
                fence[i][j] = cipher[idx]
                idx += 1

    # Read the decrypted message
    result = []
    rail = 0
    direction = 1
    for i in range(len(cipher)):
        result.append(fence[rail][i])
        rail += direction
        if rail == 0 or rail == key - 1:
            direction *= -1

    return ''.join(result)

# Get input from the user
text = input("Enter the text: ")
key = int(input("Enter the key (number of rails): "))

# Encrypt the text
cipher_text = encrypt_rail_fence(text, key)
```

```
print("Encrypted Text:", cipher_text)

# Decrypt the text
decrypted_text = decrypt_rail_fence(cipher_text, key)
print("Decrypted Text:", decrypted_text)
```

row transposition

```
# Function to encrypt using row transposition
def encrypt(plaintext, key):
    columns = [''] * key
    for i, char in enumerate(plaintext):
        columns[i % key] += char

    # Join columns to get encrypted text
    ciphertext = ''.join(columns)
    return ciphertext

# Function to decrypt using row transposition
def decrypt(ciphertext, key):
    num_full_columns = len(ciphertext) // key
    remainder = len(ciphertext) % key

    columns = [''] * key
    index = 0

    # Fill each column for decryption
    for i in range(key):
        length = num_full_columns + (1 if i < remainder else 0)
        columns[i] = ciphertext[index:index + length]
        index += length

    # Recreate the original message
    plaintext = ''
    for i in range(num_full_columns + 1):
        for col in columns:
            if i < len(col):
                plaintext += col[i]

    return plaintext

# Get user input
plaintext = input("Enter the plaintext: ")
key = int(input("Enter the key (number of columns): "))

# Encrypt the plaintext
ciphertext = encrypt(plaintext, key)
print("Encrypted Text:", ciphertext)

# Decrypt the ciphertext
decrypted_text = decrypt(ciphertext, key)
print("Decrypted Text:", decrypted_text)
```

RSA

```
import random
from math import gcd

# Step 1: Function to find modular inverse
def mod_inverse(e, phi):
    for d in range(1, phi):
        if (e * d) % phi == 1:
            return d
```

```
    return None

# Step 2: Function to check if a number is prime
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

# Step 3: RSA key generation
def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError("Both numbers must be prime.")
    elif p == q:
        raise ValueError("p and q cannot be the same.")

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

    # Choose an integer e such that e and phi(n) are coprime
    e = random.choice([i for i in range(2, phi) if gcd(i, phi) == 1])

    # Calculate d such that (e * d) % phi = 1
    d = mod_inverse(e, phi)

    # Public key (e, n) and Private key (d, n)
    return ((e, n), (d, n))

# Step 4: Encryption
def encrypt(public_key, plaintext):
    e, n = public_key
    return [(ord(char) ** e) % n for char in plaintext]

# Step 5: Decryption
def decrypt(private_key, ciphertext):
    d, n = private_key
    return ''.join([chr((char ** d) % n) for char in ciphertext])

# User input
p = int(input("Enter a prime number (p): "))
q = int(input("Enter another prime number (q): "))
plaintext = input("Enter a message to encrypt: ")

# Generate RSA keys
public_key, private_key = generate_keypair(p, q)
print(f"Public Key: {public_key}")
print(f"Private Key: {private_key}")

# Encrypt and Decrypt
ciphertext = encrypt(public_key, plaintext)
print(f"Encrypted message: {ciphertext}")

decrypted_message = decrypt(private_key, ciphertext)
print(f"Decrypted message: {decrypted_message}")
```

sha-256

```
import hashlib

# Step 1: Hashing function using SHA-256
def sha256_hash(message):
```

```
    sha_signature = hashlib.sha256(message.encode('utf-8')).hexdigest()
    return sha_signature

# Get input from user
message = input("Enter a message to hash with SHA-256: ")

# Generate SHA-256 hash
hashed_message = sha256_hash(message)

# Output the hashed message
print(f"SHA-256 hash of the message is: {hashed_message}")


vigenere

def vigenere_encrypt(plain_text, key):
    encrypted_text = ""
    key = key.upper()  # Convert the key to uppercase for consistency
    key_length = len(key)

    for i, char in enumerate(plain_text):
        if char.isalpha():
            base = ord("A") if char.isupper() else ord("a")
            shift = ord(key[i % key_length]) - ord("A")
            shifted_char = chr((ord(char) - base + shift) % 26 + base)
            encrypted_text += shifted_char
        else:
            encrypted_text += char

    return encrypted_text

def main():
    try:
        plaintext = input("Enter the text you want to encrypt: ")
        key = input("Enter the encryption key (keyword or phrase): ")
        encrypted_text = vigenere_encrypt(plaintext, key)
        print(f"Encrypted text: {encrypted_text}")
    except KeyboardInterrupt:
        print("\nOperation aborted by user.")

if __name__ == "__main__":
    main()
```