1. Caesar Cipher

```
def caesar_encrypt(plaintext, shift):
    ciphertext = ""
    for char in plaintext:
        if char.isupper():
            ciphertext += chr((ord(char) + shift - 65) % 26 + 65)
        elif char.islower():
            ciphertext += chr((ord(char) + shift - 97) % 26 + 97)
        else:
            ciphertext += char
    return ciphertext

def caesar_decrypt(ciphertext, shift):
    return caesar_encrypt(ciphertext, -shift)

def brute_force_caesar(ciphertext):
    print("Brute-force attack results:")
    for shift in range(26):
        decrypted = caesar_decrypt(ciphertext, shift)
        print(f"Shift {shift:2}: {decrypted}")

def main():
    while True:
        print("\nCaesar Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Brute-Force Attack")
        print("4. Exit")
```

```
        choice = input("Enter your choice (1-4): ")

        if choice == '1':
            plaintext = input("Enter plaintext: ")
            shift = int(input("Enter shift value (0-25): "))
            encrypted = caesar_encrypt(plaintext, shift)
            print(f"Encrypted text: {encrypted}")

        elif choice == '2':
            ciphertext = input("Enter ciphertext: ")
            shift = int(input("Enter shift value (0-25): "))
            decrypted = caesar_decrypt(ciphertext, shift)
            print(f"Decrypted text: {decrypted}")

        elif choice == '3':
            ciphertext = input("Enter ciphertext to brute-force: ")
            brute_force_caesar(ciphertext)
        elif choice == '4':
            print("Exiting program...")
            break
        else:
            print("Invalid choice. Please try again.")

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

2. PlayFair Cipher

```
def prepare_input(text):
    # Convert to uppercase and remove non-letters
    text = ''.join(filter(str.isalpha, text.upper()))

    # Replace J with I
    text = text.replace('J', 'I')

    # Create digraphs (pairs)
    i = 0
    digraphs = []
    while i < len(text):
        a = text[i]
        b = ''
        if i + 1 < len(text):
            b = text[i + 1]
            if a == b:
                b = 'X'
                i += 1
            else:
                i += 2
        else:
            b = 'X'
            i += 1
        digraphs.append(a + b)
    return digraphs
```

```
def create_matrix(key):
    # Remove duplicates and replace J with I
    key = key.upper().replace('J', 'I')
    seen = set()
    matrix = []

    for char in key:
        if char not in seen and char.isalpha():
            seen.add(char)
            matrix.append(char)

    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':
        if char not in seen:
            seen.add(char)
            matrix.append(char)

    return [matrix[i*5:(i+1)*5] for i in range(5)]

def find_position(matrix, letter):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == letter:
                return i, j
    return None

def encrypt_pair(pair, matrix):
    a_row, a_col = find_position(matrix, pair[0])
    b_row, b_col = find_position(matrix, pair[1])
```

```
    if a_row == b_row:
        return matrix[a_row][(a_col + 1) % 5] + matrix[b_row][(b_col +
1) % 5]
    elif a_col == b_col:
        return matrix[(a_row + 1) % 5][a_col] + matrix[(b_row + 1) %
5][b_col]
    else:
        return matrix[a_row][b_col] + matrix[b_row][a_col]


def decrypt_pair(pair, matrix):
    a_row, a_col = find_position(matrix, pair[0])
    b_row, b_col = find_position(matrix, pair[1])


    if a_row == b_row:
        return matrix[a_row][(a_col - 1) % 5] + matrix[b_row][(b_col -
1) % 5]
    elif a_col == b_col:
        return matrix[(a_row - 1) % 5][a_col] + matrix[(b_row - 1) %
5][b_col]
    else:
        return matrix[a_row][b_col] + matrix[b_row][a_col]


def playfair_encrypt(plaintext, key):
    matrix = create_matrix(key)
    digraphs = prepare_input(plaintext)
    return ''.join([encrypt_pair(pair, matrix) for pair in digraphs])


def playfair_decrypt(ciphertext, key):
    matrix = create_matrix(key)
```

```
    digraphs = prepare_input(ciphertext)
    return ''.join([decrypt_pair(pair, matrix) for pair in digraphs])

# ---------- Example Usage ----------
if __name__ == "__main__":
    key = input("Enter the key: ")
    text = input("Enter the plaintext: ")

    encrypted = playfair_encrypt(text, key)
    print(f"Encrypted: {encrypted}")

    decrypted = playfair_decrypt(encrypted, key)
    print(f"Decrypted: {decrypted}")
```

3. RSA Algorithm

```
import random

# ---------- Extended Euclidean Algorithm ----------
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y
```

```
# ---------- Modular Inverse ----------
def mod_inverse(e, phi):
    gcd, x, _ = extended_gcd(e, phi)
    if gcd != 1:
        raise Exception("Modular inverse doesn't exist.")
    else:
        return x % phi

# ---------- Check for Primality (simple) ----------
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

# ---------- Generate Public and Private Keys ----------
def generate_keys():
    # Choose two distinct prime numbers p and q
    while True:
        p = random.randint(50, 100)
        q = random.randint(50, 100)
        if is_prime(p) and is_prime(q) and p != q:
            break

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

```
    # Choose public exponent e such that 1 < e < phi and gcd(e, phi) = 1
    e = 3
    while True:
        if extended_gcd(e, phi)[0] == 1:
            break
        e += 2

    # Compute private key d
    d = mod_inverse(e, phi)

    return (e, n), (d, n)

# ---------- Encryption ----------
def encrypt(message, public_key):
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in message]
    return cipher

# ---------- Decryption ----------
def decrypt(cipher, private_key):
    d, n = private_key
    decrypted = ''.join([chr(pow(char, d, n)) for char in cipher])
    return decrypted

# ---------- File Operations ----------
def read_file(filename):
    with open(filename, 'r') as file:
```

```
        return file.read()

def write_file(filename, content):
    with open(filename, 'w') as file:
        file.write(str(content))

# ---------- Main Function ----------
if __name__ == "__main__":
    public_key, private_key = generate_keys()

    print(f"Public Key (e, n): {public_key}")
    print(f"Private Key (d, n): {private_key}")

    # Read plaintext from file
    input_file = "plaintext.txt"
    message = read_file(input_file)
    print(f"Original message from file: {message}")

    # Encrypt and save to file
    cipher = encrypt(message, public_key)
    write_file("encrypted.txt", cipher)
    print("Encrypted message saved to encrypted.txt")

    # Decrypt and save to file
    decrypted_message = decrypt(cipher, private_key)
    write_file("decrypted.txt", decrypted_message)
    print("Decrypted message saved to decrypted.txt")
```

```
    # Print results
    print(f"Encrypted data: {cipher}")
    print(f"Decrypted message: {decrypted_message}")
```

4. Diffie Hellman

```
import random

def power_mod(base, exponent, mod):
    """Efficient modular exponentiation"""
    result = 1
    base = base % mod
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % mod
        exponent = exponent >> 1
        base = (base * base) % mod
    return result

def diffie_hellman(p, g, private_a, private_b):
    # Compute public values
    public_a = power_mod(g, private_a, p)
    public_b = power_mod(g, private_b, p)

    # Compute shared secret
    shared_secret_a = power_mod(public_b, private_a, p)
    shared_secret_b = power_mod(public_a, private_b, p)
```

```
    return public_a, public_b, shared_secret_a, shared_secret_b


# Example usage
if __name__ == "_main_":
    # Prime modulus and primitive root
    p = int(input("Enter a prime number (p): "))
    g = int(input("Enter a primitive root modulo p (g): "))

    # Private keys (randomly chosen)
    private_a = random.randint(2, p - 2)
    private_b = random.randint(2, p - 2)

    print(f"Alice's Private Key: {private_a}")
    print(f"Bob's Private Key: {private_b}")

    public_a, public_b, secret_a, secret_b = diffie_hellman(p, g,
private_a, private_b)

    print(f"Alice's Public Key: {public_a}")
    print(f"Bob's Public Key: {public_b}")
    print(f"Alice's Shared Secret: {secret_a}")
    print(f"Bob's Shared Secret: {secret_b}")

    if secret_a == secret_b:
        print("✅ Shared secret successfully established.")
    else:
        print("❌ Shared secret mismatch.")
```