Diffie Hellman

```
import random

def power_mod(base, exponent, modulus):
    result = 1
    while exponent > 0:
    if exponent % 2 == 1:
    result = (result * base) % modulus
    base = (base * base) % modulus
    exponent //= 2
    return result

def diffie_hellman(p, g):
    print(f"Prime number (p): {p}")
    print(f"Base (g): {g}")
    a = random.randint(2, p-2)
    b = random.randint(2, p-2)
    print(f"Alice's private key (a): {a}")
    print(f"Bob's private key (b): {b}")
    A = power_mod(g, a, p)
    print(f"Alice's public value (A = g^a mod p): {A}")
    B = power_mod(g, b, p)
    print(f"Bob's public value (B = g^b mod p): {B}")
    K_A = power_mod(B, a, p)
    print(f"Alice's shared secret key (K_A = B^a mod p): {K_A}")
    K_B = power_mod(A, b, p)
    print(f"Bob's shared secret key (K_B = A^b mod p): {K_B}")
    if K_A == K_B:
    print(f"Shared secret key: {K_A}")
    else:
    print("Something went wrong. The shared keys don't match!")

if __name__ == "__main__":
    p = 23
    g = 5
    diffie_hellman(p, g)
```

```
Prime number (p): 23
Base (g): 5
Alice's private key (a): 10
Bob's private key (b): 13
Alice's public value (A = g^a mod p): 9
Bob's public value (B = g^b mod p): 16
Alice's shared secret key (K_A = B^a mod p): 6
Bob's shared secret key (K_B = A^b mod p): 6
Shared secret key: 6
```

SHA-1

```
import hashlib

def sha1_hash(input_string):
    sha1 = hashlib.sha1()
    sha1.update(input_string.encode('utf-8'))
    return sha1.hexdigest()

if __name__ == "__main__":
    input_string = "hello world"
    print(f"Input: {input_string}")
    print(f"SHA-1 Hash: {sha1_hash(input_string)}")
```

```
Input: hello world
SHA-1 Hash: 2ef7bde608ce5404e97d5f042f95f89f1c232871
```

MD5

```
import hashlib

def md5_hash(input_string):
    md5 = hashlib.md5()
    md5.update(input_string.encode('utf-8'))
    return md5.hexdigest()

if __name__ == "__main__":
    input_string = "hello world"
    print(f"Input: {input_string}")
    print(f"MD5 Hash: {md5_hash(input_string)}")
```

```
Input: hello world
MD5 Hash: 5eb63bbbe01eeed093cb22bb8f5acdc3
```

Simple Columnar Transposition

```
def simple_columnar_transposition_encrypt(plaintext, key):
    n = len(key)
    matrix = ['' for _ in range(n)]
    for i in range(len(plaintext)):
    matrix[i % n] += plaintext[i]
    ciphertext = ''.join(matrix)
    return ciphertext

def simple_columnar_transposition_decrypt(ciphertext, key):
```

```
    n = len(key)
    rows = len(ciphertext) // n
    cols = n
    matrix = ['' for _ in range(cols)]
    idx = 0
    for i in range(cols):
    for j in range(rows):
    matrix[i] += ciphertext[idx]
    idx += 1
    plaintext = ''.join(matrix)
    return plaintext

if __name__ == "__main__":
    plaintext = "HELLOTHISISANEXAMPLE"
    key = "43125"
    print(f"Plaintext: {plaintext}")
    simple_ciphertext = simple_columnar_transposition_encrypt(plaintext, key)
    print(f"Simple Columnar Transposition Ciphertext: {simple_ciphertext}")
    simple_decrypted = simple_columnar_transposition_decrypt(simple_ciphertext, key)
    print(f"Simple Columnar Transposition Decrypted: {simple_decrypted}")
```

```
Plaintext: HELLOTHISISANEXAMPLE
Key: 43125
```

utput:

```text
Plaintext: HELLOTHISISANEXAMPLE
Simple Columnar Transposition Ciphertext: HTAESLNPIEXMLIOHSS
Simple Columnar Transposition Decrypted: HELLOTHISISANEXAMPLE
```

Advanced Columnar Transposition

```
def advanced_columnar_transposition_encrypt(plaintext, key):
  n = len(key)
  sorted_key_indices = sorted(range(n), key=lambda x: key[x])
  rows = (len(plaintext) + n - 1) // n
  matrix = ['' for _ in range(n)]
  for i in range(len(plaintext)):
    col = i % n
    matrix[col] += plaintext[i]
  ciphertext = ''.join(matrix[i] for i in sorted_key_indices)
  return ciphertext

def advanced_columnar_transposition_decrypt(ciphertext, key):
```

```
    n = len(key)
    sorted_key_indices = sorted(range(n), key=lambda x: key[x])
    rows = (len(ciphertext) + n - 1) // n
    column_lengths = [rows + 1 if i < len(ciphertext) % n else rows for i in range(n)]
    matrix = ['' for _ in range(n)]
    idx = 0
    for i in sorted_key_indices:
        matrix[i] = ciphertext[idx:idx + column_lengths[i]]
        idx += column_lengths[i]
    plaintext = ''
    for i in range(rows):
        for j in range(n):
            if i < len(matrix[j]):
                plaintext += matrix[j][i]
    return plaintext

if __name__ == "__main__":
    plaintext = "HELLOTHISISANEXAMPLE"
    key = "43125"
    print(f"Plaintext: {plaintext}")
    advanced_ciphertext = advanced_columnar_transposition_encrypt(plaintext, key)
    print(f"Advanced Columnar Transposition Ciphertext: {advanced_ciphertext}")
    advanced_decrypted = advanced_columnar_transposition_decrypt(advanced_ciphertext,
key)
    print(f"Advanced Columnar Transposition Decrypted: {advanced_decrypted}")
```

```
Plaintext: HELLOTHISISANEXAMPLE
Key: 43125
```

Output:

```
text

Plaintext: HELLOTHISISANEXAMPLE
Advanced Columnar Transposition Ciphertext: HTAESLNPIEXMLIOHSS
Advanced Columnar Transposition Decrypted: HELLOTHISISANEXAMPLE
```

Euclidean & Advanced Euclidean

```
def euclidean_algorithm(a, b):
        while b:
        a, b = b, a % b
        return a
```

```
def extended_euclidean_algorithm(a, b):
    if b == 0:
    return a, 1, 0
    gcd, x1, y1 = extended_euclidean_algorithm(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

if __name__ == "__main__":
    a = 56
    b = 98
    print(f"Euclidean Algorithm GCD: {euclidean_algorithm(a, b)}")
    gcd, x, y = extended_euclidean_algorithm(a, b)
    print(f"Extended Euclidean Algorithm GCD: {gcd}, x: {x}, y: {y}")
```

```
Euclidean Algorithm GCD: 14
Extended Euclidean Algorithm GCD: 14, x: 2, y: -1
```

Triple DES

```
from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
import binascii

def encrypt_3des(plain_text, key):
    cipher = DES3.new(key, DES3.MODE_CBC)
    padded_data = pad(plain_text.encode(), DES3.block_size)
    cipher_text = cipher.encrypt(padded_data)
    return cipher.iv + cipher_text

def decrypt_3des(cipher_text, key):
    iv = cipher_text[:DES3.block_size]
    cipher_text = cipher_text[DES3.block_size:]
    cipher = DES3.new(key, DES3.MODE_CBC, iv)
    decrypted_data = unpad(cipher.decrypt(cipher_text), DES3.block_size)
    return decrypted_data.decode()

def generate_key():
    return get_random_bytes(24)

key = generate_key()
plain_text = "This is a secret message!"

cipher_text = encrypt_3des(plain_text, key)
print("Cipher Text (Hex):", binascii.hexlify(cipher_text))
```

```
decrypted_message = decrypt_3des(cipher_text, key)
print("Decrypted Message:", decrypted_message)
```

```
========================= RESTART: /home/mtechlab/c.py =========================
Cipher Text (Hex): b'e699e08e867b65fa9dfc6982e09bda6521b9d82a7ae28d2f299883498b33f01591f7c0909383897a'
Decrypted Message: This is a secret message!
>>>|
```

Rail Fence

```
# Rail Fence Cipher Encryption
def rail_fence_encrypt(text, num_rails):
    if num_rails == 1: return text
    fence = ['' for _ in range(num_rails)]
    row, step = 0, 1

    for char in text:
    fence[row] += char
    if row == 0: step = 1
    elif row == num_rails - 1: step = -1
    row += step

    return ''.join(fence)

# Rail Fence Cipher Decryption
def rail_fence_decrypt(cipher_text, num_rails):
    if num_rails == 1: return cipher_text
    fence = ['' for _ in range(num_rails)]
    row, step = 0, 1
    cipher_len = len(cipher_text)
    char_pos = [0] * num_rails

    for i in range(cipher_len):
    fence[row] += '*'
    if row == 0: step = 1
    elif row == num_rails - 1: step = -1
    row += step

    idx = 0
    for i in range(num_rails):
    for j in range(len(fence[i])):
    if fence[i][j] == '*':
        fence[i] = fence[i][:j] + cipher_text[idx] + fence[i][j+1:]
        idx += 1

    result, row, step = '', 0, 1
```

```
        for i in range(cipher_len):
        result += fence[row][char_pos[row]]
        char_pos[row] += 1
        if row == 0: step = 1
        elif row == num_rails - 1: step = -1
        row += step

        return result

# Example usage
text = "WEAREDISCOVEREDFLEEATONCE"
num_rails = 3
cipher_text = rail_fence_encrypt(text, num_rails)
print(f"Cipher Text: {cipher_text}")
decrypted_text = rail_fence_decrypt(cipher_text, num_rails)
print(f"Decrypted Text: {decrypted_text}")
```

```
Cipher Text: WECRLTEERDSOEEFEAOCAIVDEN
Decrypted Text: WEAREDISCOVEREDFLEEATONCE
```

Vigenere Cipher

```
import string

def vigenere_encrypt(plaintext, key):
        alphabet = string.ascii_uppercase
        key = (key * (len(plaintext) // len(key))) + key[:len(plaintext) % len(key)]  # Repeat key
to match plaintext length
        return ''.join([alphabet[(alphabet.index(p) + alphabet.index(k)) % 26] if p in alphabet
else p
                for p, k in zip(plaintext.upper(), key)])

def vigenere_decrypt(ciphertext, key):
        alphabet = string.ascii_uppercase
        key = (key * (len(ciphertext) // len(key))) + key[:len(ciphertext) % len(key)]  # Repeat
key to match ciphertext length
        return ''.join([alphabet[(alphabet.index(c) - alphabet.index(k)) % 26] if c in alphabet
else c
                for c, k in zip(ciphertext.upper(), key)])

# Example usage
plaintext = "HELLO WORLD"
key = "KEY"

ciphertext = vigenere_encrypt(plaintext, key)
print(f"Ciphertext: {ciphertext}")
```

```
decrypted_text = vigenere_decrypt(ciphertext, key)
print(f"Decrypted Text: {decrypted_text}")
```

```
Ciphertext: RIJVS GSPVH
Decrypted Text: HELLO WORLD
```

Vernam Cipher

```
def vernam_encrypt_decrypt(text, key):
    """Encrypt or decrypt using Vernam Cipher (same process for both)"""
    if len(text) != len(key):
    raise ValueError("Key and text must be of the same length")

    # XOR each character of text with the corresponding character of the key
    return ''.join(chr(ord(t) ^ ord(k)) for t, k in zip(text, key))

# Example usage
plaintext = "HELLO"
key = "XMCKL"  # Key must be the same length as plaintext

ciphertext = vernam_encrypt_decrypt(plaintext, key)
print(f"Ciphertext: {repr(ciphertext)}")  # Use repr to view non-printable characters

# Decrypting is the same as encrypting
decrypted_text = vernam_encrypt_decrypt(ciphertext, key)
print(f"Decrypted Text: {decrypted_text}")
```

```
Ciphertext: '\x10\x08\x0f\x07\x03'
Decrypted Text: HELLO
```