CAESAR CIPHER

```
def caesar_cipher(text, shift, mode='encrypt'):
    result = ""
    shift = shift if mode == 'encrypt' else -shift
    for char in text:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            result += chr((ord(char) - shift_base + shift) % 26 + shift_base)
        else:
            result += char
    return result

text = "HelloWorld"
shift = 3
encrypted = caesar_cipher(text, shift, mode='encrypt')
decrypted = caesar_cipher(encrypted, shift, mode='decrypt')
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

SUBSTITUTION CIPHER

```
def substitution_cipher(text, key, mode='encrypt'):
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    key_map = dict(zip(alphabet, key)) if mode == 'encrypt' else dict(zip(key, alphabet))
    result = ''.join([key_map[char] if char in key_map else char for char in text.lower()])
    return result

key = 'qwertyuiopasdfghjklzxcvbnm'
text = "hello"
encrypted = substitution_cipher(text, key, 'encrypt')
decrypted = substitution_cipher(encrypted, key, 'decrypt')
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

HILL CIPHER

```
import numpy as np

def hill_cipher(text, key, mode='encrypt'):
    n = len(key)
    text = text.lower().replace(' ', '')
    if mode == 'encrypt':
        text += 'x' * ((n - len(text) % n) % n)  # Padding
        text_vector = [ord(char) - ord('a') for char in text]
        encrypted = (np.dot(key, text_vector) % 26).astype(int)
        return ''.join([chr(num + ord('a')) for num in encrypted])
    else:  # Decrypt
        key_inv = np.linalg.inv(key).astype(int) % 26
        text_vector = [ord(char) - ord('a') for char in text]
        decrypted = (np.dot(key_inv, text_vector) % 26).astype(int)
        return ''.join([chr(num + ord('a')) for num in decrypted])

key = np.array([[6, 24, 1], [13, 16, 10], [20, 17, 15]])  # Example key matrix
text = "hello"
encrypted = hill_cipher(text, key, 'encrypt')
# decrypted = hill_cipher(encrypted, key, 'decrypt') (Hill requires valid decryption key)
print("Encrypted:", encrypted)
```

DES

```
from Crypto.Cipher import DES
import os

def des_encrypt_decrypt(data, key, mode='encrypt'):
    des = DES.new(key, DES.MODE_ECB)
    data = data.ljust(8)  # Padding
    if mode == 'encrypt':
        encrypted = des.encrypt(data.encode())
        return encrypted
    else:
        decrypted = des.decrypt(data)
        return decrypted.decode().strip()

key = b"abcdefgh"
data = "plaintext"
encrypted = des_encrypt_decrypt(data, key, 'encrypt')
decrypted = des_encrypt_decrypt(encrypted, key, 'decrypt')
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

BLOWFISH

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

def blowfish_encrypt_decrypt(data, key, mode='encrypt'):
    cipher = Blowfish.new(key, Blowfish.MODE_ECB)
    if mode == 'encrypt':
        encrypted = cipher.encrypt(pad(data.encode(), Blowfish.block_size))
        return encrypted
    else:
        decrypted = unpad(cipher.decrypt(data), Blowfish.block_size)
        return decrypted.decode()

key = b"secretkey"
data = "HelloWorld"
encrypted = blowfish_encrypt_decrypt(data, key, 'encrypt')
decrypted = blowfish_encrypt_decrypt(encrypted, key, 'decrypt')
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

AES

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def aes_encrypt_decrypt(data, key, mode='encrypt'):
    cipher = AES.new(key, AES.MODE_ECB)
    if mode == 'encrypt':
        encrypted = cipher.encrypt(pad(data.encode(), AES.block_size))
        return encrypted
    else:
        decrypted = unpad(cipher.decrypt(data), AES.block_size)
        return decrypted.decode()

key = b"thisisaverysecretkey!!"[:16]
data = "SecretMessage"
encrypted = aes_encrypt_decrypt(data, key, 'encrypt')
decrypted = aes_encrypt_decrypt(encrypted, key, 'decrypt')
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

RC4

```
from Crypto.Cipher import ARC4

def rc4_encrypt_decrypt(data, key):
    cipher = ARC4.new(key)
    encrypted = cipher.encrypt(data)
    return encrypted

key = b"key123"
data = b"Hello world"
encrypted = rc4_encrypt_decrypt(data, key)
print("Encrypted:", encrypted)
```

RSA

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

key = RSA.generate(2048)
cipher_rsa = PKCS1_OAEP.new(key)

data = "Hello RSA"
encrypted = cipher_rsa.encrypt(data.encode())
decrypted = cipher_rsa.decrypt(encrypted).decode()

print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

DIFFIE HELLMAN KEY EXCHANGE

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Diffie-Hellman Key Exchange</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      margin-top: 50px;
    }
    input {
      margin: 10px;
      padding: 10px;
      font-size: 16px;
    }
    button {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
    }
    .output {
      margin-top: 20px;
      font-weight: bold;
    }
  </style>
</head>
<body>
  <h1>Diffie-Hellman Key Exchange</h1>
  <p>Enter the values for prime number \( p \), generator \( g \), and secret keys for Alice
and Bob:</p>
  <form onsubmit="return calculateKeys()">
    <label for="prime">Prime Number (p):</label>
    <input type="number" id="prime" required><br>
    <label for="generator">Generator (g):</label>
    <input type="number" id="generator" required><br>
    <label for="aliceSecret">Alice's Secret Key (a):</label>
    <input type="number" id="aliceSecret" required><br>
    <label for="bobSecret">Bob's Secret Key (b):</label>
    <input type="number" id="bobSecret" required><br>
    <button type="submit">Calculate Shared Keys</button>
  </form>

  <div class="output" id="output"></div>
```

```
    <script>
        function calculateKeys() {
            // Get user inputs
            const p = parseInt(document.getElementById('prime').value);
            const g = parseInt(document.getElementById('generator').value);
            const a = parseInt(document.getElementById('aliceSecret').value);
            const b = parseInt(document.getElementById('bobSecret').value);

            // Calculate public keys
            const A = Math.pow(g, a) % p;
            const B = Math.pow(g, b) % p;

            // Calculate shared keys
            const sharedKeyAlice = Math.pow(B, a) % p;
            const sharedKeyBob = Math.pow(A, b) % p;

            // Display results
            const outputDiv = document.getElementById('output');
            outputDiv.innerHTML = `
                <p>Alice's Public Key: <strong>${A}</strong></p>
                <p>Bob's Public Key: <strong>${B}</strong></p>
                <p>Shared Secret Key (Alice): <strong>${sharedKeyAlice}</strong></p>
                <p>Shared Secret Key (Bob): <strong>${sharedKeyBob}</strong></p>
            `;

            // Prevent form submission
            return false;
        }
    </script>
</body>
</html>
```

SHA-1

```
import hashlib

data = "Hello World"
sha1_digest = hashlib.sha1(data.encode()).hexdigest()
print("SHA-1 Digest:", sha1_digest)
```

MD5

```
import hashlib

data = "Hello World"
md5_digest = hashlib.md5(data.encode()).hexdigest()
print("MD5 Digest:", md5_digest)
```

COLUMNAR TRANSPOSITION

```
def columnar_encrypt(text, key):
    n = len(key)
    columns = sorted(range(n), key=lambda x: key[x])
    matrix = [text[i:i + n] for i in range(0, len(text), n)]
    while len(matrix[-1]) < n:  # Padding
        matrix[-1] += 'X'
    encrypted = ''.join([''.join(row[col] for row in matrix) for col in columns])
    return encrypted

def columnar_decrypt(text, key):
    n = len(key)
    num_rows = len(text) // n
    columns = sorted(range(n), key=lambda x: key[x])
    reverse_columns = sorted(range(n), key=lambda x: columns[x])
    cols = [text[i * num_rows:(i + 1) * num_rows] for i in range(n)]
    matrix = [''.join(cols[col][row] for col in reverse_columns) for row in range(num_rows)]
    return ''.join(matrix).rstrip('X')

key = [3, 1, 4, 2]
text = "HELLO WORLD"
encrypted = columnar_encrypt(text, key)
decrypted = columnar_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

ADVANCED COLUMNAR CIPHER

```
def advanced_columnar_encrypt(text, key):
    n = len(key)
    key_map = sorted([(k, i) for i, k in enumerate(key)], key=lambda x: x[0])
    columns = ["" for _ in key]
    for i, char in enumerate(text):
        columns[i % n] += char
    encrypted = ''.join(columns[key_map[i][1]] for i in range(n))
    return encrypted

def advanced_columnar_decrypt(text, key):
    n = len(key)
    key_map = sorted([(k, i) for i, k in enumerate(key)], key=lambda x: x[0])
    rows = len(text) // n
    cols = [text[i * rows:(i + 1) * rows] for i in range(n)]
    decrypted = ''.join(''.join(cols[key_map[j][1]][i] for j in range(n)) for i in range(rows))
    return decrypted.rstrip('X')

key = [3, 1, 4, 2]
text = "HELLO WORLD"
encrypted = advanced_columnar_encrypt(text, key)
decrypted = advanced_columnar_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

EUCLEIIAN

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

a, b = 252, 105
print("GCD using Euclidean Algorithm:", euclidean_algorithm(a, b))
```

ADVANCED EUCLIDIAN

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

a, b = 252, 105
gcd, x, y = extended_euclidean(a, b)
print(f"GCD: {gcd}, x: {x}, y: {y}")
```