**1.1 Caesar Cipher**

def caesar\_encrypt(text, shift):

    result = ""

    for char in text:

        if char.isalpha():

            base = ord('A') if char.isupper() else ord('a')

            result += chr((ord(char) - base + shift) % 26 + base)

        else:

            result += char

    return result

def caesar\_decrypt(cipher, shift):

    return caesar\_encrypt(cipher, -shift)

plain\_text = "Hello, World!"

shift\_key = 3

encrypted = caesar\_encrypt(plain\_text, shift\_key)

decrypted = caesar\_decrypt(encrypted, shift\_key)

print("Plain Text :", plain\_text)

print("Encrypted  :", encrypted)

print("Decrypted  :", decrypted)

**1.2 Playfair Cipher**

def create\_matrix(key):

    key = key.lower().replace('j','i')

    s = "".join(dict.fromkeys([c for c in key if c.isalpha()]))

    alphabet = "abcdefghiklmnopqrstuvwxyz"

    matrix = (s + "".join([c for c in alphabet if c not in s]))

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

def find\_pos(matrix, c):

    for i in range(5):

        for j in range(5):

            if matrix[i][j] == c:

                return i,j

def prepare(text):

    text = text.lower().replace('j','i')

    text = "".join([c for c in text if c.isalpha()])

    i, res = 0, ""

    while i < len(text):

        a = text[i]

        b = text[i+1] if i+1 < len(text) and text[i+1] != a else 'x'

        res += a + b

        i += 2 if b != 'x' else 1

    if len(res) % 2: res += 'x'

    return res

def playfair(text, matrix, enc=True):

    text = prepare(text) if enc else text

    res = ""

    step = 1 if enc else -1

    for i in range(0, len(text), 2):

        r1,c1 = find\_pos(matrix, text[i])

        r2,c2 = find\_pos(matrix, text[i+1])

        if r1 == r2:

            res += matrix[r1][(c1+step)%5] + matrix[r2][(c2+step)%5]

        elif c1 == c2:

            res += matrix[(r1+step)%5][c1] + matrix[(r2+step)%5][c2]

        else:

            res += matrix[r1][c2] + matrix[r2][c1]

    return res

key = input("Key: ")

text = input("Text: ")

m = create\_matrix(key)

print("Matrix:")

for row in m: print(" ".join(row))

enc = playfair(text, m)

dec = playfair(enc, m, False)

print("Encrypted:", enc)

print("Decrypted:", dec)

**2. DES**

IP = [

    58, 50, 42, 34, 26, 18, 10, 2,

    60, 52, 44, 36, 28, 20, 12, 4,

    62, 54, 46, 38, 30, 22, 14, 6,

    64, 56, 48, 40, 32, 24, 16, 8,

    57, 49, 41, 33, 25, 17, 9, 1,

    59, 51, 43, 35, 27, 19, 11, 3,

    61, 53, 45, 37, 29, 21, 13, 5,

    63, 55, 47, 39, 31, 23, 15, 7

]

DES\_ROUNDS = 16

def permute(block: int, table: list) -> int:

    result = 0

    for i, pos in enumerate(table):

        bit = (block >> (64 - pos)) & 1

        result |= bit << (63 - i)

    return result

def feistel\_function(half\_block: int, key: int) -> int:

    return half\_block ^ key

def key\_schedule(key: int) -> list:

    return [key ^ (i + 1) for i in range(DES\_ROUNDS)]

def des\_round(block: int, key: int) -> int:

    left = (block >> 32) & 0xFFFFFFFF

    right = block & 0xFFFFFFFF

    new\_right = feistel\_function(right, key)

    return (new\_right << 32) | left  # Swap halves

def des\_encrypt(block: int, key: int) -> int:

    round\_keys = key\_schedule(key)

    block = permute(block, IP)

    for rk in round\_keys:

        block = des\_round(block, rk)

    return block

def string\_to\_uint64(text: str) -> int:

    bytes\_text = text.encode('utf-8')

    padded = bytes\_text[:8].ljust(8, b'\x00')

    return int.from\_bytes(padded, byteorder='little')

def uint64\_to\_hex\_string(value: int) -> str:

    return f"0x{value:016X}"

plaintext = input("Enter 8-character plaintext: ")[:8]

key\_text = input("Enter 8-character key: ")[:8]

pt\_block = string\_to\_uint64(plaintext)

key\_block = string\_to\_uint64(key\_text) & 0xFFFFFFFFFFFFFF00  # Mimic C behavior

print(f"\nOriginal: {uint64\_to\_hex\_string(pt\_block)}")

encrypted = des\_encrypt(pt\_block, key\_block)

print(f"Encrypted: {uint64\_to\_hex\_string(encrypted)}")

**3. Diffie-Hellman**

def mod\_exp(base, exp, mod):

    res = 1

    base %= mod

    while exp:

        if exp & 1:

            res = (res \* base) % mod

        base = (base \* base) % mod

        exp >>= 1

    return res

p = int(input("Prime p: "))

g = int(input("Primitive root g: "))

a = int(input("Alice private key a: "))

b = int(input("Bob private key b: "))

A = mod\_exp(g, a, p)

B = mod\_exp(g, b, p)

print(f"Alice's public key: {A}")

print(f"Bob's public key: {B}")

s\_a = mod\_exp(B, a, p)

s\_b = mod\_exp(A, b, p)

print(f"Alice's shared secret: {s\_a}")

print(f"Bob's shared secret: {s\_b}")

print("\nSuccess!" if s\_a == s\_b else "\nFailure.")

**4. SHA**

import struct

def rotr(x, n): return ((x >> n) | (x << (32 - n))) & 0xFFFFFFFF

def generate\_constants():

    k, n = [], 2

    while len(k) < 64:

        if all(n % d for d in range(2, int(n\*\*0.5)+1)):

            k.append(int((n \*\* (1/3)) \* (1 << 32)) & 0xFFFFFFFF)

        n += 1

    return k

def pad\_message(msg):

    ml = len(msg) \* 8

    msg += b'\x80'

    pad\_len = (56 - len(msg) % 64) % 64

    msg += b'\x00' \* pad\_len + ml.to\_bytes(8, 'big')

    return msg

def sha256\_block(chunk, h, k):

    w = list(struct.unpack('>16L', chunk)) + [0]\*48

    for i in range(16, 64):

        s0 = rotr(w[i-15], 7) ^ rotr(w[i-15], 18) ^ (w[i-15] >> 3)

        s1 = rotr(w[i-2], 17) ^ rotr(w[i-2], 19) ^ (w[i-2] >> 10)

        w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFF

    a, b, c, d, e, f, g, h1 = h

    for i in range(64):

        S1 = rotr(e,6) ^ rotr(e,11) ^ rotr(e,25)

        ch = (e & f) ^ (~e & g)

        temp1 = (h1 + S1 + ch + k[i] + w[i]) & 0xFFFFFFFF

        S0 = rotr(a,2) ^ rotr(a,13) ^ rotr(a,22)

        maj = (a & b) ^ (a & c) ^ (b & c)

        temp2 = (S0 + maj) & 0xFFFFFFFF

        a, b, c, d, e, f, g, h1 = (

            (temp1 + temp2) & 0xFFFFFFFF, a, b, c, (d + temp1) & 0xFFFFFFFF, e, f, g

        )

    for i, val in enumerate([a, b, c, d, e, f, g, h1]):

        h[i] = (h[i] + val) & 0xFFFFFFFF

def sha256(msg):

    k = generate\_constants()

    h = [

        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,

        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19

    ]

    msg = pad\_message(msg)

    for i in range(0, len(msg), 64):

        sha256\_block(msg[i:i+64], h, k)

    return b''.join(x.to\_bytes(4, 'big') for x in h)

if \_\_name\_\_ == '\_\_main\_\_':

    msg = input("Enter message: ").encode()

    print("SHA-256 Hash:", sha256(msg).hex())