**Caeser Cipher**

def ceaser\_cipher(text,Shift):

    temp = ''

    for char in text:

        enc = (ord(char)+Shift)%256

        temp+=chr(enc)

    return temp

Shift = int(input('Enter the shift value : '))

pt = input('Enter the string to perform the encryption : ')

b = ceaser\_cipher(pt,Shift).upper()

print("After encryption the cipher text is : ",b)

decrypt = ceaser\_cipher(b,-Shift)

print("After decryption the string is : ", decrypt)

**Hill cipher**

import numpy as np

def generate\_key\_matrix(key):

    key = key.replace(" ", "").upper()

    key\_len = len(key)

    matrix\_size = int(key\_len \*\* 0.5)

    key\_matrix = [ord(char) - 65 for char in key]

    while len(key\_matrix) % matrix\_size != 0:

        key\_matrix.append(0)

    return np.array(key\_matrix).reshape(matrix\_size, matrix\_size)

def hill\_encrypt(message, key):

    key\_matrix = generate\_key\_matrix(key)

    matrix\_size = key\_matrix.shape[0]

    message = message.replace(" ", "").upper()

    while len(message) % matrix\_size != 0:

        message += 'X'

    message\_matrix = [ord(char) - 65 for char in message]

    message\_matrix = np.array(message\_matrix).reshape(-1, matrix\_size)

    encrypted\_text = ""

    for row in message\_matrix:

        row = np.dot(row, key\_matrix) % 26

        encrypted\_text += ''.join([chr(char + 65) for char in row])

    return encrypted\_text

def mod\_mat\_inv(matrix, modulus):

    det = int(np.round(np.linalg.det(matrix)))

    det\_inv = pow(det, -1, modulus)

    adjugate = np.round(det \* np.linalg.inv(matrix)).astype(int) % modulus

    inverse = (det\_inv \* adjugate) % modulus

    return inverse

def hill\_decrypt(ciphertext, key):

    key\_matrix = generate\_key\_matrix(key)

    key\_inverse = mod\_mat\_inv(key\_matrix, 26)

    matrix\_size = key\_matrix.shape[0]

    ciphertext = ciphertext.replace(" ", "").upper()

    ciphertext\_matrix = [ord(char) - 65 for char in ciphertext]

    ciphertext\_matrix = np.array(ciphertext\_matrix).reshape(-1, matrix\_size)

    decrypted\_text = ""

    for row in ciphertext\_matrix:

        row = np.dot(row, key\_inverse) % 26

        decrypted\_text += ''.join([chr(char + 65) for char in row])

    decrypted\_text = decrypted\_text.rstrip('X')

    return decrypted\_text

message = input("Enter the Plain text : ")

key = input("Enter the Key string : ")

encrypted\_message = hill\_encrypt(message, key)

print("Encrypted:", encrypted\_message)

decrypted\_message = hill\_decrypt(encrypted\_message, key)

print("Decrypted:", decrypted\_message)

**Simple DES**

FIXED\_IP = [2, 6, 3, 1, 4, 8, 5, 7]

FIXED\_EP = [4, 1, 2, 3, 2, 3, 4, 1]

FIXED\_IP\_INVERSE = [4, 1, 3, 5, 7, 2, 8, 6]

FIXED\_P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]

FIXED\_P8 = [6, 3, 7, 4, 8, 5, 10, 9]

FIXED\_P4 = [2, 4, 3, 1]

S0 = [[1, 0, 3, 2],

      [3, 2, 1, 0],

      [0, 2, 1, 3],

      [3, 1, 3, 2]]

S1 = [[0, 1, 2, 3],

      [2, 0, 1, 3],

      [3, 0, 1, 0],

      [2, 1, 0, 3]]

def permutate(original, fixed\_key):

    new = ''

    for i in fixed\_key:

        new += original[i - 1]

    return new

def left\_half(bits):

    return bits[:len(bits)//2]

def right\_half(bits):

    return bits[len(bits)//2:]

def shift(bits):

    rotated\_left\_half = left\_half(bits)[1:] + left\_half(bits)[0]

    rotated\_right\_half = right\_half(bits)[1:] + right\_half(bits)[0]

    return rotated\_left\_half + rotated\_right\_half

def key1(key):

    return permutate(shift(permutate(key, FIXED\_P10)), FIXED\_P8)

def key2(key):

    return permutate(shift(shift(shift(permutate(key, FIXED\_P10)))), FIXED\_P8)

def xor(bits, key):

    new = ''

    for bit, key\_bit in zip(bits, key):

        new += str(((int(bit) + int(key\_bit)) % 2))

    return new

def lookup\_in\_sbox(bits, sbox):

    row = int(bits[0] + bits[3], 2)

    col = int(bits[1] + bits[2], 2)

    return '{0:02b}'.format(sbox[row][col])

def f\_k(bits, key):

    L = left\_half(bits)

    R = right\_half(bits)

    bits = permutate(R, FIXED\_EP)

    bits = xor(bits, key)

    bits = lookup\_in\_sbox(left\_half(bits), S0) + \

        lookup\_in\_sbox(right\_half(bits), S1)

    bits = permutate(bits, FIXED\_P4)

    return xor(bits, L)

def encrypt(plain\_text, key):

    bits = permutate(plain\_text, FIXED\_IP)

    temp = f\_k(bits, key1(key))

    bits = right\_half(bits) + temp

    bits = f\_k(bits, key2(key))

    encrypted = permutate(bits + temp, FIXED\_IP\_INVERSE)

    print("Encrypted: ", encrypted)

    return encrypted

def decrypt(cipher\_text, key):

    bits = permutate(cipher\_text, FIXED\_IP)

    temp = f\_k(bits, key2(key))

    bits = right\_half(bits) + temp

    bits = f\_k(bits, key1(key))

    decrypted = permutate(bits + temp, FIXED\_IP\_INVERSE)

    print("Decrypted: ", decrypted)

    return decrypted

def main():

    plain\_text = input("Enter plaintext (8 bits): ")

    key = input("Enter key (10 bits): ")

    encrypted = encrypt(plain\_text, key)

    decrypted = decrypt(encrypted, key)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**RSA**

def generate\_keypair(p, q):

    n = p \* q

    phi = (p - 1) \* (q - 1)

    e = 65537

    d = pow(e, -1, phi)

    return ((e, n), (d, n))

def encrypt(public\_key, plaintext):

    e, n = public\_key

    cipher = [pow(ord(char), e, n) for char in plaintext]

    return cipher

def decrypt(private\_key, ciphertext):

    d, n = private\_key

    plain = [chr(pow(char, d, n)) for char in ciphertext]

    return ''.join(plain)

plaintext = input("Enter the plaintext: ")

p = int(input("Enter the value of p (a prime number): "))

q = int(input("Enter the value of q (a prime number): "))

public\_key, private\_key = generate\_keypair(p, q)

encrypted\_msg = encrypt(public\_key, plaintext)

print("Encrypted message:", encrypted\_msg)

decrypted\_msg = decrypt(private\_key, encrypted\_msg)

print("Decrypted message:", decrypted\_msg)

**Diffie Hellman Key exchange Algorithm**

def diffie\_hellman(p, g, a, b):

  A = pow(g, a, p)

  B = pow(g, b, p)

  s = pow(B, a, p)

  return s

p = 17

g = 5

a = 4

b = 6

shared\_key = diffie\_hellman(p, g, a, b)

print(f"Shared secret key: {shared\_key}")

**Vernam Cipher**

def encrypt(key, pt):

    temp = ""

    for i in range(len(pt)):

        char = pt[i]

        temp += chr((ord(char) - 97 + ord(key[i]) - 97) % 26 + 97)

    return temp

def decrypt(key, enc):

    temp = ""

    for i in range(len(enc)):

        char = enc[i]  # Corrected from pt[i]

        temp += chr((ord(char) - ord(key[i]) + 26) % 26 + 97)

    return temp

if \_\_name\_\_ == "\_\_main\_\_":

    while True:

        pt = input("Enter the plain text: ")

        key = input("Enter the key: ")

        if len(pt)!=len(key):

            print("The length of the key and the plain text has to be same")

        else:

            enc = encrypt(key, pt)

            print("After encryption: ", enc)

            dec = decrypt(key, enc)

            print("After decryption: ", dec)

**Rail Fence Cipher**

def encrypt\_rail\_fence(text, key):

    rail = [['\n' for \_ in range(len(text))] for \_ in range(key)]

    dir\_down, row, col = False, 0, 0

    for char in text:

        if (row == 0) or (row == key - 1):

            dir\_down = not dir\_down

        rail[row][col] = char

        col += 1

        row += 1 if dir\_down else -1

    return ''.join(char for row in rail for char in row if char != '\n')

def decrypt\_rail\_fence(cipher, key):

    rail = [['\n' for \_ in range(len(cipher))] for \_ in range(key)]

    dir\_down, row, col = None, 0, 0

    for i in range(len(cipher)):

        if row == 0:

            dir\_down = True

        if row == key - 1:

            dir\_down = False

        rail[row][col] = '\*'

        col += 1

        row += 1 if dir\_down else -1

    index = 0

    for i in range(key):

        for j in range(len(cipher)):

            if rail[i][j] == '\*' and index < len(cipher):

                rail[i][j] = cipher[index]

                index += 1

    result = []

    row, col = 0, 0

    for i in range(len(cipher)):

        if row == 0:

            dir\_down = True

        if row == key - 1:

            dir\_down = False

        if rail[row][col] != '\*':

            result.append(rail[row][col])

            col += 1

        row += 1 if dir\_down else -1

    return ''.join(result)

# Example usage with user input

plaintext = input("Enter your plain text: ")

key = int(input("Enter your key: "))

encrypted\_text = encrypt\_rail\_fence(plaintext, key)

print("Encrypted Text:", encrypted\_text)

decrypted\_text = decrypt\_rail\_fence(encrypted\_text, key)

print("Decrypted Text:", decrypted\_text)

**Row Column Transposition Cipher**

import math

def encrypt\_message(message, key):

    col = len(key)

    row = int(math.ceil(len(message) / col))

    fill\_null = row \* col - len(message)

    message += '\_' \* fill\_null

    matrix = [message[i:i+col] for i in range(0, len(message), col)]

    cipher = ''.join(matrix[i][key.index(k)] for k in sorted(key) for i in range(row))

    return cipher

def decrypt\_message(cipher, key):

    col = len(key)

    row = int(math.ceil(len(cipher) / col))

    key\_sorted = sorted(key)

    matrix = [[''] \* col for \_ in range(row)]

    index = 0

    for k in key\_sorted:

        col\_index = key.index(k)

        for j in range(row):

            matrix[j][col\_index] = cipher[index]

            index += 1

    decrypted\_message = ''.join(''.join(row) for row in matrix).rstrip('\_')

    return decrypted\_message

# Driver Code

msg = "Geeks for Geeks"

key = "HACK"

cipher = encrypt\_message(msg, key)

print("Encrypted Message:", cipher)

decrypted\_msg = decrypt\_message(cipher, key)

print("Decrypted Message:", decrypted\_msg)

**Miller rabin**

import random

def is\_prime(n, k=5):

    """

    Miller-Rabin primality test.

    Parameters:

    - n: The number to be tested for primality.

    - k: The number of rounds of testing. Higher values of k increase the accuracy.

    Returns:

    - True if n is likely to be prime, False otherwise.

    """

    if n <= 1:

        return False

    if n == 2 or n == 3:

        return True

    if n % 2 == 0:

        return False

    # Write n as 2^r \* d + 1

    r, d = 0, n - 1

    while d % 2 == 0:

        r += 1

        d //= 2

    # Witness loop

    for \_ in range(k):

        a = random.randint(2, n - 2)

        x = pow(a, d, n)

        if x == 1 or x == n - 1:

            continue

        for \_ in range(r - 1):

            x = pow(x, 2, n)

            if x == n - 1:

                break

        else:

            return False  # Not prime

    return True  # Likely prime

# Example usage

number\_to\_test = 1031

rounds\_of\_testing = 5

if is\_prime(number\_to\_test, rounds\_of\_testing):

    print(f"{number\_to\_test} is likely to be a prime number.")

else:

    print(f"{number\_to\_test} is not a prime number.")