Ceaser cipher

```python
#A python program to illustrate Caesar Cipher Technique
def encrypt(text,s):
    result = ""

    # traverse text
    for i in range(len(text)):
        char = text[i]

        # Encrypt uppercase characters
        if (char.isupper()):
            result += chr((ord(char) + s-65) % 26 + 65)

        # Encrypt lowercase characters
        else:
            result += chr((ord(char) + s - 97) % 26 + 97)

    return result

#check the above function
text = "ATTACKATONCE"
s = 4
print ("Text : " + text)
print ("Shift : " + str(s))
print ("Cipher: " + encrypt(text,s))
```

Playfair Cipher

```python
# Python program to implement Playfair Cipher

# Function to convert the string to lowercase


def toLowerCase(text):
    return text.lower()

# Function to remove all spaces in a string


def removeSpaces(text):
    newText = ""
    for i in text:
```

```python
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText

# Function to group 2 elements of a string
# as a list element


def Diagraph(text):
    Diagraph = []
    group = 0
    for i in range(2, len(text), 2):
        Diagraph.append(text[group:i])

        group = i
    Diagraph.append(text[group:])
    return Diagraph

# Function to fill a letter in a string element
# If 2 letters in the same string matches


def FillerLetter(text):
    k = len(text)
    if k % 2 == 0:
        for i in range(0, k, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    return new_word
```

```python
list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

# Function to generate the 5x5 key square matrix


def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
            key_letters.append(i)

    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)

    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]

    return matrix


def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j


def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]

    char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
```

```python
    else:
        char2 = matr[e2r][e2c+1]

    return char1, char2


def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1r == 4:
        char1 = matr[0][e1c]
    else:
        char1 = matr[e1r+1][e1c]

    char2 = ''
    if e2r == 4:
        char2 = matr[0][e2c]
    else:
        char2 = matr[e2r+1][e2c]

    return char1, char2


def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    char1 = matr[e1r][e2c]

    char2 = ''
    char2 = matr[e2r][e1c]

    return char1, char2


def encryptByPlayfairCipher(Matrix, plainList):
    CipherText = []
    for i in range(0, len(plainList)):
        c1 = 0
        c2 = 0
        ele1_x, ele1_y = search(Matrix, plainList[i][0])
        ele2_x, ele2_y = search(Matrix, plainList[i][1])

        if ele1_x == ele2_x:
            c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
            # Get 2 letter cipherText
        elif ele1_y == ele2_y:
            c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
```

```
        else:
            c1, c2 = encrypt_RectangleRule(
                Matrix, ele1_x, ele1_y, ele2_x, ele2_y)

        cipher = c1 + c2
        CipherText.append(cipher)
    return CipherText


text_Plain = 'instruments'
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)

print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ""
for i in CipherList:
    CipherText += i
print("CipherText:", CipherText)
```

Hill Cipher

```
# Python3 code to implement Hill Cipher

keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
```

```python
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

# Following function encrypts the message
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                       messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

    # Get key matrix from the key string
    getKeyMatrix(key)

    # Generate vector for the message
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    # Following function generates
    # the encrypted vector
    encrypt(messageVector)

    # Generate the encrypted text
    # from the encrypted vector
    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

    # Finally print the ciphertext
    print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

    # Get the message to
    # be encrypted
    message = "ACT"
```

```python
    # Get the key
    key = "GYBNQKURP"

    HillCipher(message, key)

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

Vigenere Cipher

```python
# Python code to implement
# Vigenere Cipher

# This function generates the
# key in a cyclic manner until
# it's length isn't equal to
# the length of original text
def generateKey(string, key):
    key = list(key)
    if len(string) == len(key):
        return(key)
    else:
        for i in range(len(string) -
                    len(key)):
            key.append(key[i % len(key)])
    return("" . join(key))

# This function returns the
# encrypted text generated
# with the help of the key
def cipherText(string, key):
    cipher_text = []
    for i in range(len(string)):
        x = (ord(string[i]) +
            ord(key[i])) % 26
        x += ord('A')
        cipher_text.append(chr(x))
    return("" . join(cipher_text))

# This function decrypts the
# encrypted text and returns
# the original text
def originalText(cipher_text, key):
    orig_text = []
    for i in range(len(cipher_text)):
```

```
        x = (ord(cipher_text[i]) -
            ord(key[i]) + 26) % 26
        x += ord('A')
        orig_text.append(chr(x))
    return("" . join(orig_text))

# Driver code
if __name__ == "__main__":
    string = "GEEKSFORGEEKS"
    keyword = "AYUSH"
    key = generateKey(string, keyword)
    cipher_text = cipherText(string,key)
    print("Ciphertext :", cipher_text)
    print("Original/Decrypted Text :",
        originalText(cipher_text, key))
```

DSA

```
# Python3 code for the above approach

# Hexadecimal to binary conversion


def hex2bin(s):
    mp = {'0': "0000",
        '1': "0001",
        '2': "0010",
        '3': "0011",
        '4': "0100",
        '5': "0101",
        '6': "0110",
        '7': "0111",
        '8': "1000",
        '9': "1001",
        'A': "1010",
        'B': "1011",
        'C': "1100",
        'D': "1101",
        'E': "1110",
        'F': "1111"}
    bin = ""
    for i in range(len(s)):
        bin = bin + mp[s[i]]
    return bin
```

```python
# Binary to hexadecimal conversion


def bin2hex(s):
    mp = {"0000": '0',
        "0001": '1',
        "0010": '2',
        "0011": '3',
        "0100": '4',
        "0101": '5',
        "0110": '6',
        "0111": '7',
        "1000": '8',
        "1001": '9',
        "1010": 'A',
        "1011": 'B',
        "1100": 'C',
        "1101": 'D',
        "1110": 'E',
        "1111": 'F'}
    hex = ""
    for i in range(0, len(s), 4):
        ch = ""
        ch = ch + s[i]
        ch = ch + s[i + 1]
        ch = ch + s[i + 2]
        ch = ch + s[i + 3]
        hex = hex + mp[ch]

    return hex

# Binary to decimal conversion


def bin2dec(binary):

    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal
```

```python
# Decimal to binary conversion


def dec2bin(num):
    res = bin(num).replace("0b", "")
    if(len(res) % 4 != 0):
        div = len(res) / 4
        div = int(div)
        counter = (4 * (div + 1)) - len(res)
        for i in range(0, counter):
            res = '0' + res
    return res

# Permute function to rearrange the bits


def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation

# shifting the bits towards left by nth shifts


def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s = s + k[j]
        s = s + k[0]
        k = s
        s = ""
    return k

# calculating xow of two strings of binary number a and b


def xor(a, b):
    ans = ""
    for i in range(len(a)):
        if a[i] == b[i]:
            ans = ans + "0"
        else:
            ans = ans + "1"
    return ans
```

```python
# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [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]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
        6, 7, 8, 9, 8, 9, 10, 11,
        12, 13, 12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21, 20, 21,
        22, 23, 24, 25, 24, 25, 26, 27,
        28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

        [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

        [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],
```

```
        [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

        [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

        [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

        [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
        [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
        [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
        [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

        [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
        [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
        [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
        [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
            39, 7, 47, 15, 55, 23, 63, 31,
            38, 6, 46, 14, 54, 22, 62, 30,
            37, 5, 45, 13, 53, 21, 61, 29,
            36, 4, 44, 12, 52, 20, 60, 28,
            35, 3, 43, 11, 51, 19, 59, 27,
            34, 2, 42, 10, 50, 18, 58, 26,
            33, 1, 41, 9, 49, 17, 57, 25]


def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)

    # Initial Permutation
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))

    # Splitting
```

```
    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        # Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)

        # XOR RoundKey[i] and right_expanded
        xor_x = xor(right_expanded, rkb[i])

        # S-boxex: substituting the value from s-box table by calculating row and
column
        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(
                int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] +
xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str = sbox_str + dec2bin(val)

        # Straight D-box: After substituting rearranging the bits
        sbox_str = permute(sbox_str, per, 32)

        # XOR left and sbox_str
        result = xor(left, sbox_str)
        left = result

        # Swapper
        if(i != 15):
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
            " ", bin2hex(right), " ", rk[i])

    # Combination
    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text


pt = "123456ABCD132536"
key = "AABB09182736CCDD"

# Key generation
```

```python
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
            2, 2, 2, 2,
            1, 2, 2, 2,
            2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
    # Shifting the bits by nth shifts by checking from shift table
    left = shift_left(left, shift_table[i])
    right = shift_left(right, shift_table[i])

    # Combination of left and right string
    combine_str = left + right
```

```
    # Compression of key from 56 to 48 bits
    round_key = permute(combine_str, key_comp, 48)

    rkb.append(round_key)
    rk.append(bin2hex(round_key))

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("Decryption")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)

# This code is contributed by Aditya Jain
```

RSA

```
from math import gcd

# defining a function to perform RSA approch
def RSA(p: int, q: int, message: int):
    # calculating n
    n = p * q

    # calculating totient, t
    t = (p - 1) * (q - 1)

    # selecting public key, e
    for i in range(2, t):
        if gcd(i, t) == 1:
            e = i
            break

    # selecting private key, d
    j = 0
    while True:
        if (j * e) % t == 1:
            d = j
```

```
            break
        j += 1

    # performing encryption
    ct = (message ** e) % n
    print(f"Encrypted message is {ct}")

    # performing decryption
    mes = (ct ** d) % n
    print(f"Decrypted message is {mes}")

# Testcase - 1
RSA(p=53, q=59, message=89)

# Testcase - 2
RSA(p=3, q=7, message=12)
```

Deffie Hellman with man in the middle attack

```
import random

# public keys are taken
# p is a prime number
# g is a primitive root of p
p = int(input('Enter a prime number : '))
g = int(input('Enter a number : '))


class A:
    def __init__(self):
        # Generating a random private number selected by alice
        self.n = random.randint(1, p)

    def publish(self):
        # generating public values
        return (g**self.n)%p

    def compute_secret(self, gb):
        # computing secret key
        return (gb**self.n)%p


class B:
    def __init__(self):
        # Generating a random private number selected for alice
```

```python
        self.a = random.randint(1, p)
        # Generating a random private number selected for bob
        self.b = random.randint(1, p)
        self.arr = [self.a,self.b]

    def publish(self, i):
        # generating public values
        return (g**self.arr[i])%p

    def compute_secret(self, ga, i):
        # computing secret key
        return (ga**self.arr[i])%p


alice = A()
bob = A()
eve = B()

# Printing out the private selected number by Alice and Bob
print(f'Alice selected (a) : {alice.n}')
print(f'Bob selected (b) : {bob.n}')
print(f'Eve selected private number for Alice (c) : {eve.a}')
print(f'Eve selected private number for Bob (d) : {eve.b}')

# Generating public values
ga = alice.publish()
gb = bob.publish()
gea = eve.publish(0)
geb = eve.publish(1)
print(f'Alice published (ga): {ga}')
print(f'Bob published (gb): {gb}')
print(f'Eve published value for Alice (gc): {gea}')
print(f'Eve published value for Bob (gd): {geb}')

# Computing the secret key
sa = alice.compute_secret(gea)
sea = eve.compute_secret(ga,0)
sb = bob.compute_secret(geb)
seb = eve.compute_secret(gb,1)
print(f'Alice computed (S1) : {sa}')
print(f'Eve computed key for Alice (S1) : {sea}')
print(f'Bob computed (S2) : {sb}')
print(f'Eve computed key for Bob (S2) : {seb}')
```

AES

```
AES 256 Encryption in Python
#Generate the initial matrix from the input data
def gen_matrix(data):
    matrix = [[0] * 4 for x in range(4)]
    #Generate the 4x4 input matrix
    row = 3
    col = 3
    for x in range(15,-1,-1):
        matrix[3-row][3-col] =  (data >> (8 * (x))) & 0xFF
        if x % 4 == 0:
            col -= 1
        row = (row - 1) % 4
    return matrix

#Add round key operation
def add_round_key(key):
    global matrix
    key_matrix = gen_matrix(key)
    for r in range(4):
        for c in range(4):
            matrix[r][c] = matrix[r][c] ^ key_matrix[r][c]
    dumpMatrix("add round keys")

for round in range(len(round_keys)-1):
    #Byte substitution
    byte_sub()

    #Shift rows
    shift_rows()

    # Final round doesn't include mix columns
    if round < 13:
        #Mix columns
        mix_columns()

    #Add round key
    add_round_key(round_keys[round+1])

#Perform the byte substitution layer
def byte_sub():
    global matrix
    for r in range(4):
        for c in range(4):
```

```
            t = matrix[r][c]
            matrix[r][c] = sbox(matrix[r][c])
    dumpMatrix("Byte sub.")

#Helper function for a leftward rotation of a matrix row
def l_rotate_row(rowNum, shiftCount):
    for x in range(shiftCount):
        global matrix
        temp_byte = matrix[rowNum][0]
        matrix[rowNum][0] = matrix[rowNum][1]
        matrix[rowNum][1] = matrix[rowNum][2]
        matrix[rowNum][2] = matrix[rowNum][3]
        matrix[rowNum][3] = temp_byte

#Shift rows operation
def shift_rows():
    global matrix
    l_rotate_row(1, 1)
    l_rotate_row(2, 2)
    l_rotate_row(3, 3)
    dumpMatrix("shift rows")

#Mix columns operation
def mix_columns():
    global matrix
    for c in range(4):
        col = [
                matrix[0][c],
                matrix[1][c],
                matrix[2][c],
                matrix[3][c]
        ]
        col = mmult(col)
        matrix[0][c] = col[0]
        matrix[1][c] = col[1]
        matrix[2][c] = col[2]
        matrix[3][c] = col[3]
    dumpMatrix("mix columns")

#Matrix multiplication done in GF(2^8)
def mmult(matb):
    c = [
            None,
            None,
            None,
```

```
            None
        ]
    c[0] = gf2mult(2, matb[0]) ^ gf2mult(3, matb[1]) ^ matb[2] ^ matb[3]

    c[1] = matb[0] ^ gf2mult(2, matb[1]) ^ gf2mult(3, matb[2]) ^ matb[3]

    c[2] = matb[0] ^ matb[1] ^ gf2mult(2, matb[2]) ^ gf2mult(3, matb[3])

    c[3] = gf2mult(3, matb[0]) ^ matb[1] ^ matb[2] ^ gf2mult(2, matb[3])

    return c

#GF(2^8) multiplication using AES irreducible polynomial
def gf2mult(x, y):
    ret = 0
    for i in range(8):
        if (y & 1) != 0:
            ret = ret ^ x
        b = (x & 0x80)
        x = (x << 1) & 0xFF
        if b:
            x = x ^ 0x1B
        y = (y >> 1) & 0xFF
    return ret

    #Reconstruct the 128-bit cipher text from the matrix
    cipher = 0
    for c in range(4):
        for r in range(4):
            cipher = cipher << 8
            b = matrix[r][c]
            cipher |= b
```

Date: 10-02-2024.

CRYPTOGRAPHY AND NETWORK SECURITY LAB

DIGITAL ASSIGNMENT 1

Submitted by:
Mukund Agarwal
21BCI0003
7037662404
mukund.agarwal2021@vitstudent.ac.in

Submitted to:
Prakash G.

1. **Convert hex to base64**

The string:

```
49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f7
573206d757368726f6f6d
```

Should produce:

```
SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
```

So go ahead and make that happen. You'll need to use this code for the rest of the exercises.

CODE:

```java
// code by
// Mukund Agarwal
// 21BCI0003

import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.Scanner;

public class hex_to_base {

    public static void main(String[] args) {
        // Create a Scanner object to get user input
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter a hexadecimal string
        System.out.print("Enter a hexadecimal string: ");
        String hexString = scanner.nextLine();

        // Close the scanner to avoid resource leaks
        scanner.close();

        // Convert hex to bytes
        byte[] hexBytes = hexStringToBytes(hexString);

        // Encode bytes to base64
        String base64String = bytesToBase64(hexBytes);
```

```java
        // Print the result
        System.out.println("Base64 Encoded: " + base64String);
    }

    private static byte[] hexStringToBytes(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];

        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(hexString.charAt(i), 16) <<
4)
                    + Character.digit(hexString.charAt(i + 1), 16));
        }

        return data;
    }

    private static String bytesToBase64(byte[] bytes) {
        return Base64.getEncoder().encodeToString(bytes);
    }
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\hex_to_base.java
Enter a hexadecimal string: 49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f7573206d7573
68726f6f6d
Base64 Encoded: SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
```

2. **Fixed XOR**

Write a function that takes two equal-length buffers and produces their XOR combination.

If your function works properly, then when you feed it the string:

```
1c0111001f010100061a024b53535009181c
```

... after hex decoding, and when XOR'd against:

```
686974207468652062756c6c277320657965
```

... should produce:

```
746865206b696420646f6e277420706c6179
```

CODE:

```
// code by
// Mukund Agarwal
// 21BCI0003


import java.util.Scanner;

public class XORBuffer {

    public static void main(String[] args) {
        // Create a Scanner object to get user input
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter the first buffer in hexadecimal format
        System.out.print("Enter the first hexadecimal buffer: ");
        String hexBuffer1 = scanner.nextLine();

        // Prompt the user to enter the second buffer in hexadecimal format
        System.out.print("Enter the second hexadecimal buffer: ");
        String hexBuffer2 = scanner.nextLine();

        // Close the scanner to avoid resource leaks
        scanner.close();
```

```
        // Convert hex to bytes
        byte[] buffer1 = hexStringToBytes(hexBuffer1);
        byte[] buffer2 = hexStringToBytes(hexBuffer2);

        // Check if the buffers are of equal length
        if (buffer1.length != buffer2.length) {
            System.out.println("Error: Buffers must be of equal length.");
            return;
        }

        // Perform XOR operation on the buffers
        byte[] result = xorBuffers(buffer1, buffer2);

        // Print the result as a hexadecimal string
        System.out.println("XOR Result: " + bytesToHexString(result));
    }

    private static byte[] hexStringToBytes(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];

        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(hexString.charAt(i), 16) <<
4)
                    + Character.digit(hexString.charAt(i + 1), 16));
        }

        return data;
    }

    private static byte[] xorBuffers(byte[] buffer1, byte[] buffer2) {
        byte[] result = new byte[buffer1.length];

        for (int i = 0; i < buffer1.length; i++) {
            result[i] = (byte) (buffer1[i] ^ buffer2[i]);
        }

        return result;
    }

    private static String bytesToHexString(byte[] bytes) {
        StringBuilder hexStringBuilder = new StringBuilder();
        for (byte b : bytes) {

            hexStringBuilder.append(String.format("%02X", b));
        }
        return hexStringBuilder.toString();
    }
```

```
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\crypto_q2.java
Enter the first hexadecimal buffer: 1c0111001f010100061a024b53535009181c
Enter the second hexadecimal buffer: 686974207468652062756c6c277320657965
XOR Result: 746865206B696420646F6E277420706C6179
```

3. **Single-byte XOR cipher**

The hex-encoded string:

`1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736`

... has been XOR'd against a single character. Find the key, decrypt the message.

You can do this by hand. But don't: write code to do it for you.

How? Devise some method for "scoring" a piece of English plaintext. Character frequency is a good metric. Evaluate each output and choose the one with the best score.

CODE:

```
// code by
// Mukund Agarwal
// 21BCI0003


import java.util.Scanner;

public class SingleByteXORDecryption {

    public static void main(String[] args) {
        // Create a Scanner object to get user input
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter the hex-encoded string
        System.out.print("Enter the hex-encoded string: ");
        String hexString = scanner.nextLine();

        // Close the scanner to avoid resource leaks
        scanner.close();

        // Convert hex to bytes
        byte[] hexBytes = hexStringToBytes(hexString);

        // Find the key and decrypt the message
        findKeyAndDecrypt(hexBytes);
    }

    private static void findKeyAndDecrypt(byte[] hexBytes) {
        // Iterate over possible single-byte XOR keys (ASCII 0 to 255)
```

```
        for (int key = 0; key <= 255; key++) {
            byte[] decryptedBytes = new byte[hexBytes.length];
            for (int i = 0; i < hexBytes.length; i++) {
                decryptedBytes[i] = (byte) (hexBytes[i] ^ key);
            }

            // Convert decrypted bytes to a string
            String decryptedString = new String(decryptedBytes);

            // Print potential decryption with the current key
            System.out.println("Key: " + key + " | Decrypted: " +
decryptedString);
        }
    }

    private static byte[] hexStringToBytes(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];

        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(hexString.charAt(i), 16) <<
4)
                    + Character.digit(hexString.charAt(i + 1), 16));
        }

        return data;
    }
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\crypto_q3.java
Enter the hex-encoded string: 1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736
```

```
Key: 82 | Decrypted: Ieeacdm*GI-y*fcao*k*zedn*el*hkied
```

4. **Detect single-character XOR**

One of the 60-character strings in [this file] has been encrypted by single-character XOR.

Find it.

(Your code from #3 should help.)

CODE:

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class XORDecryptor {

    public static void main(String[] args) {
        try {
            // Read the contents of the file
            List<String> ciphertexts =
Files.readAllLines(Paths.get("encrypt.txt"));

            // Convert each ciphertext from hexadecimal to bytes
            List<byte[]> ciphertextBytes = new ArrayList<>();
            for (String ciphertext : ciphertexts) {
                ciphertextBytes.add(hexStringToBytes(ciphertext.trim()));
            }

            // Find the key and the plaintext with the minimum Hamming
distance
            int minDistance = Integer.MAX_VALUE;
            int bestKey = 0;
            byte[] bestPlaintext = null;

            for (byte[] ciphertext : ciphertextBytes) {
                for (int key = 0; key < 256; key++) { // Try all possible 8-
bit keys
                    byte[] plaintext = xorDecrypt(ciphertext, (byte) key);
```

```
                    if (scorePlaintext(plaintext)) {
                        int distance = hammingDistance(plaintext, ciphertext);

                        if (distance < minDistance) {
                            minDistance = distance;
                            bestKey = key;
                            bestPlaintext = plaintext;
                        }
                    }
                }
            }

            // Print the result
            System.out.println("Key: " + bestKey);
            System.out.println("Plaintext: " + new String(bestPlaintext,
StandardCharsets.UTF_8));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static byte[] xorDecrypt(byte[] ciphertext, byte key) {
        byte[] plaintext = new byte[ciphertext.length];
        for (int i = 0; i < ciphertext.length; i++) {
            plaintext[i] = (byte) (ciphertext[i] ^ key);
        }
        return plaintext;
    }

    private static int hammingDistance(byte[] s1, byte[] s2) {
        int distance = 0;
        for (int i = 0; i < s1.length && i < s2.length; i++) {
            int xorResult = s1[i] ^ s2[i];
            distance += Integer.bitCount(xorResult);
        }
        return distance;
    }

    private static boolean scorePlaintext(byte[] text) {
        Set<Integer> printable = new HashSet<>();
        for (int i = 32; i < 127; i++) {
            printable.add(i);
        }

        for (byte b : text) {
            if (!printable.contains((int) b)) {
                return false;
            }
```

```
        }
        return true;
    }

    private static byte[] hexStringToBytes(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];

        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(hexString.charAt(i), 16) <<
4)
                    + Character.digit(hexString.charAt(i + 1), 16));
        }

        return data;
    }
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\crypto_q4.java
Key: 96
Plaintext: W1[Mj.^2qWJZaS,=1cl&&^W6IlmnrB
```

5. **Implement repeating-key XOR**

Here is the opening stanza of an important work of the English language:

```
Burning 'em, if you ain't quick and nimble
I go crazy when I hear a cymbal
```

Encrypt it, under the key "ICE", using repeating-key XOR.

In repeating-key XOR, you'll sequentially apply each byte of the key; the first byte of plaintext will be XOR'd against I, the next C, the next E, then I again for the 4th byte, and so on.

It should come out to:

```
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272
a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e27282f
```

Encrypt a bunch of stuff using your repeating-key XOR function. Write the code for the above scenario.

CODE:

```
// code by
// Mukund Agarwal
// 21BCI0003

public class RepeatingKeyXOREncryption {

    public static void main(String[] args) {
        String plaintext = "Burning 'em, if you ain't quick and nimble\nI go
crazy when I hear a cymbal";
        String key = "ICE";

        byte[] plaintextBytes = plaintext.getBytes();
        byte[] keyBytes = key.getBytes();
        byte[] encryptedBytes = repeatingKeyXOR(plaintextBytes, keyBytes);

        StringBuilder encryptedHex = new StringBuilder();
        for (byte b : encryptedBytes) {
            encryptedHex.append(String.format("%02x", b));
        }

        System.out.println(encryptedHex.toString());
    }

    private static byte[] repeatingKeyXOR(byte[] text, byte[] key) {
```

```java
        byte[] encryptedBytes = new byte[text.length];
        int keyLength = key.length;

        for (int i = 0; i < text.length; i++) {
            encryptedBytes[i] = (byte) (text[i] ^ key[i % keyLength]);
        }

        return encryptedBytes;
    }
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\crypto_q5.java
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272a282b2f20430a652e2c652a3124333a653e2b20
27630c692b20283165286326302e27282f
```

6. **Compute the following Discrete logarithms using any programming language:**

(a) $\log_{10}(22)$ for the prime $p = 47$.

(b) $\log_{627}(608)$ for the prime $p = 941$.

CODE:

```
// code by
// Mukund Agarwal
// 21BCI0003

import java.util.Scanner;

public class DiscreteLogarithm {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the base: ");
        int base = scanner.nextInt();

        System.out.print("Enter the number: ");
        int number = scanner.nextInt();

        System.out.print("Enter the modules (prime) : ");
        int modulus = scanner.nextInt();

        scanner.close();

        int result = findDiscreteLogarithm(base, number, modulus);

        System.out.println("The discrete logarithm is: " + result);
    }

    private static int findDiscreteLogarithm(int base, int number, int
modulus) {
        int result = -1;

        for (int i = 0; i < modulus; i++) {
            // Calculate base^i mod modulus
            int power = modPow(base, i, modulus);

            // Check if base^i mod modulus equals the given number
            if (power == number) {
                result = i;
```

```java
                break;
            }
        }

        return result;
    }

    private static int modPow(int base, int exponent, int modulus) {
        if (exponent == 0) {
            return 1;
        }

        long result = 1;
        long baseValue = base % modulus;

        while (exponent > 0) {
            if (exponent % 2 == 1) {
                result = (result * baseValue) % modulus;
            }

            baseValue = (baseValue * baseValue) % modulus;
            exponent /= 2;
        }

        return (int) result;
    }
}
```

OUTPUT:

```
PS C:\CODING_PROGRAMS\JAVA\DSA_apna_college> java .\crypto_q6a.java
Enter the base: 10
Enter the number: 22
Enter the modules (prime) : 47
The discrete logarithm is: 11
```

```
Enter the base: 627
Enter the number: 608
Enter the modules (prime) : 941
The discrete logarithm is: 18
```

−x−x−x−x−x−

21BCI0003
MUKUND AGARWAL
11th MARCH 2024
Cryptography and Network Security Lab DA

Exercise: Encrypting and Decrypting with RSA

In this exercise, you will encrypt and decrypt numbers using a simple version of the RSA algorithm. Each team should have two members. ***Each*** team-member should complete the exercise as Bob, then pass along some information (but not the whole sheet!) to the other team-member, who will complete the exercise as Alice. In this way, both team-members will play both roles in the exercise. You should conceal your actual numbers from your team-members. You can consider your code for this exercise "prototype" code – you can throw it away and start design when you start the lab.

Instructions for Bob:

We will be doing B=16-bit RSA.

1. Select the encryption exponent e=17. (In practice, e=65537 would often be used for larger p and q.)
2. Calculate p like this: (Write results in the table below)
   a. Select a (B/2=) 8-bit random number. You can use `random.randint(i,j)` to select an integer x satisfying `i<=`x`<=j`. (You need to `import random` to use `random`.)
   b. Set the two highest bits and the lowest bit (to 1). This forms our tentative p.
   You can look at the binary form of, e.g. p, using `"{0:b}".format(p)`. You can set the highest bit in p using p = p | 0b10000000, and set all three bits similarly, by putting ones in the positions of the bits you wish to set in the 0bNNNNNNNN number used above.
   c. Check if p is prime. If p is not prime, add 2 to p and try again. (You can use an inefficient program to check if the number is prime; e.g., check if all numbers smaller than p do not divide p).
   d. Check if the number (p-1) is co-prime with e=17, i.e., gcd(p-1,e)=1. If not, add 2 to p and try again. (This step is necessary to ensure that we can find a d such that ed = 1 (mod z)
   Note: since e=17 is prime, you can simply check that (p-1) mod e≠0.

Initial random number (decimal)	Initial random number (binary)	*p (decimal)*	*p (binary)*	Is *p* prime?	Is (p-1)%e≠0 ?
170	10101010	195	11000011	No	-
		197	11000101	Yes	Yes

Final:	197	11000101	Yes	Yes

(Instructions for Bob, continued.)

3. Repeat step 2 to select *q*. (Note: *q* must be different from *p*. Start over if *q* will equal *p*.)

Initial random number (decimal)	Initial random number (binary)	*q (decimal)*	*q (binary)*	Is *q* prime?	Is (q-1)%e≠0 ?
181	10110101	181	10110101	Yes	Yes
	Final:	181	10110101		

4. Calculate the modulus $n = p*q$

   $n =$ 11000101×10110101=1111010000111000 1

5. Calculate the totient z = (*p*-1)*(*q*-1)

   z = 11000100×10110100=1111001110001000 0
6. Select the decryption exponent *d* such that (*de*) mod z = 1. You can simply "guess and check" all values of 1 < d < z. Only one value of d will work if e is selected as in step 5. (This would usually be done using the Extended Euclid's Algorithm.) This is your private key. Do not reveal d, p, q, n, or z to Alice or Trudy!

   d = 2753

7. Provide your public key [e;n] (that is, simply the numbers e and n) to Alice and Trudy. (This simulates posting your public key on your personal website…)

   Public key: [17,1111010000111000 1]
8. Wait for Alice to send you a secret message.
9. Once you receive the secret message from Alice, you can decrypt it using your private key. Suppose *c* is the ciphertext. Compute the original message *m* as $m = c^d \bmod n$. For smaller numbers you can simply compute this as (c**d)%n.

   c = 123 m = c^d mod n=(123^2753) mod(1111010000111000 1)

   The original message is correct as it matches with Alice's chose message.

   Don't reveal the secret message to Trudy!

Instructions for Alice:

1. You will receive the public key [e;n] (That is, simply the numbers e and n) from Bob. Write it here:

   e = 17 n = 11110100001110001

2. Select any number 0 <= m < n for your plaintext secret message. If you like, you can encrypt a sequence of ASCII characters as separate messages *m* (that is, using block encryption.)

   m =42

3. Compute the ciphertext c as c = m^e mod n. For smaller numbers you can simply compute this as (m**e)%n.

   c =(42^17)mod(11110100001110001)

4. Give the ciphertext message *c* to Bob and Trudy. This simulates Trudy eavesdropping on the wire.

Code for simple RSA demonstration for Bob and Alice:

```
import random
from math import gcd

# Function to generate a prime number with n bits
def generate_prime(bits):
   while True:
      num = random.getrandbits(bits)
      if num % 2 != 0 and is_prime(num):
         return num

# Function to check if a number is prime
def is_prime(num):
   if num < 2:
      return False
   for i in range(2, int(num ** 0.5) + 1):
      if num % i == 0:
         return False
   return True

# Function to find the modular inverse
def mod_inverse(a, m):
   for x in range(1, m):
      if (a * x) % m == 1:
         return x
   return None
```

```
# Function to generate keys for Bob
def generate_keys():
   # Select the encryption exponent e
   e = 17
   # Generate two random prime numbers for p and q
   p = generate_prime(8)
   q = generate_prime(8)
   n = p * q
   z = (p - 1) * (q - 1)
   # Find the decryption exponent d
   d = mod_inverse(e, z)
   return e, n, d

# Function to encrypt a message for Alice
def encrypt(message, e, n):
   return pow(message, e, n)

# Function to decrypt a message for Bob
def decrypt(ciphertext, d, n):
   return pow(ciphertext, d, n)

# Bob's keys
bob_e, bob_n, bob_d = generate_keys()
print("Bob's public key: [e =", bob_e, ", n =", bob_n, "]")

# Alice's keys
alice_e, alice_n, alice_d = generate_keys()
print("Alice's public key: [e =", alice_e, ", n =", alice_n, "]")

# Message to be sent
message = 12345
print("Original message:", message)

# Encrypt the message for Alice
ciphertext = encrypt(message, alice_e, alice_n)
print("Encrypted message:", ciphertext)

# Decrypt the message for Bob
decrypted_message = decrypt(ciphertext, bob_d, bob_n)
print("Decrypted message:", decrypted_message)
```

Instructions for Trudy: (optional)

(If you have extra time, you may want to play this role – simply get [e;n] and c from another team!)

1. Wait to receive the public key [e;n] (This is simply the numbers e and n) from Bob.

   e = 17

   n = 11110100001110001

2. Factor n to find p and q (Use brute-force Python loop. This is the hard step that makes RSA secure for large numbers.)

```
n = 11110100001110001

for i in range(2, n):
    if n % i == 0:
        p = i
        q = n // i
        break

print("p =", p)
print("q =", q)
```

   p=197

   q=181

3. Compute z = (p-1)*(q-1)

   z=11110011100010000

4. Now compute *d* the same way as Bob did: Select the decryption exponent *d* such that (*de*) mod z = 1. You can simply "guess and check" all values of d < z. Only one value of d will work if e is selected as in step 5. (This would usually be done using the Extended Euclid's Algorithm.)

```
e = 17
z = 11110011100010000

for d in range(2, z):
    if (d * e) % z == 1:
        break

print("d =", d)
```

d= 2753

5. Wait to receive (eavesdrop) on the ciphertext message *c* from Alice to Bob.

   c = 123

6. Decrypt the *c*, ciphertext message: Compute the original message *m* as $m = c^d \bmod n$. For smaller numbers you can simply compute this as (c**e)%n.

   m = (123^2753)mod(11110100001110001)

Consider a sender and receiver who need to exchange data confidentially using symmetric encryption. Write a java program that implements DES encryption and decryption using a 64-bit key size and 64-bit block size.

```
import javax.crypto.Cipher;

import javax.crypto.SecretKey;

import javax.crypto.SecretKeyFactory;

import javax.crypto.spec.DESKeySpec;

import java.util.Scanner;


public class DESExample {


    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);


        // Get user input for plaintext

        System.out.print("Enter plaintext: ");

        String plaintext = scanner.nextLine();


        // Get user input for DES key

        System.out.print("Enter 64-bit DES key (8 characters): ");

        String keyStr = scanner.nextLine();


        try {

            // Convert key string to bytes

            byte[] keyBytes = keyStr.getBytes();


            // Generate DES key from key bytes

            DESKeySpec desKeySpec = new DESKeySpec(keyBytes);

            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");

            SecretKey secretKey = keyFactory.generateSecret(desKeySpec);


            // Create DES cipher instance for encryption

            Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

            cipher.init(Cipher.ENCRYPT_MODE, secretKey);


            // Encrypt plaintext
```

```
            byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());

            System.out.println("Encrypted ciphertext (Base64 encoded): " +
Base64.getEncoder().encodeToString(encryptedBytes));

            // Create DES cipher instance for decryption
            cipher.init(Cipher.DECRYPT_MODE, secretKey);

            // Decrypt ciphertext
            byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

            // Convert decrypted bytes to plaintext
            String decryptedText = new String(decryptedBytes);
            System.out.println("Decrypted plaintext: " + decryptedText);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            scanner.close();
        }
    }
}
```

Consider a sender and receiver who must exchange data confidentially using symmetric encryption. Write a java program with user input that implements AES encryption using a 64/128/256 bits key size and 128-bit block size.

```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.KeyGenerator;
import javax.crypto.spec.SecretKeySpec;
import java.util.Scanner;
import java.util.Base64;

public class AESEncryptionExample {

    public static void main(String[] args) throws Exception {
```

```
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter the plaintext:");
    String plaintext = scanner.nextLine();

    System.out.println("Choose the key size (64, 128, or 256):");
    int keySize = scanner.nextInt();

    // Ensure valid key size selection
    if (keySize != 64 && keySize != 128 && keySize != 256) {
      System.out.println("Invalid key size. Please choose 64, 128, or 256.");
      return;
    }

    String encryptedText = encrypt(plaintext, keySize);
    System.out.println("Encrypted Text: " + encryptedText);

    String decryptedText = decrypt(encryptedText, keySize);
    System.out.println("Decrypted Text: " + decryptedText);
  }

  public static String encrypt(String plaintext, int keySize) throws Exception {
    KeyGenerator keyGen = KeyGenerator.getInstance("AES");
    keyGen.init(keySize);
    SecretKey secretKey = keyGen.generateKey();

    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());
    return Base64.getEncoder().encodeToString(encryptedBytes);
  }

  public static String decrypt(String encryptedText, int keySize) throws Exception {
    byte[] encryptedBytes = Base64.getDecoder().decode(encryptedText);
```

```
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(keySize);
        SecretKey secretKey = keyGen.generateKey();

        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);

        byte[] decryptedBytes = cipher.doFinal(encryptedBytes);
        return new String(decryptedBytes);
    }
}
```

Develop in java programming language with user input, a Public Key Encryption scheme by using the RSA algorithm

```
import java.security.*;
import javax.crypto.*;
import java.util.Base64;
import java.util.Scanner;

public class RSAExample {

    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in);

        // Get user input for plaintext
        System.out.print("Enter plaintext: ");
        String plaintext = scanner.nextLine();

        // Generate RSA key pair
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048); // You can adjust key size as needed
        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();
```

```
        // Encryption
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());

        System.out.println("Encrypted ciphertext (Base64 encoded): " +
Base64.getEncoder().encodeToString(encryptedBytes));

        // Decryption
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

        String decryptedText = new String(decryptedBytes);
        System.out.println("Decrypted plaintext: " + decryptedText);

        scanner.close();
    }
}
```

Design a Diffie Hellman Two-party Key Exchange protocol and perform a Man-in-the-Middle Attack with user input in java

```
import java.math.BigInteger;
import java.security.*;
import java.util.Scanner;

public class DiffieHellmanMITM {

    public static void main(String[] args) throws NoSuchAlgorithmException {
        Scanner scanner = new Scanner(System.in);
```

```
// Generate large prime number p and primitive root g
BigInteger p = BigInteger.probablePrime(512, new SecureRandom());
BigInteger g = new BigInteger("2"); // A primitive root modulo p, often 2 for simplicity

System.out.println("Prime number (p): " + p);
System.out.println("Primitive root (g): " + g);

// Alice's private key
BigInteger aPrivate = generatePrivateKey(p);

// Bob's private key
BigInteger bPrivate = generatePrivateKey(p);

System.out.println("Alice's private key: " + aPrivate);
System.out.println("Bob's private key: " + bPrivate);

// Alice's public key
BigInteger aPublic = g.modPow(aPrivate, p);

// Bob's public key
BigInteger bPublic = g.modPow(bPrivate, p);

System.out.println("Alice's public key: " + aPublic);
System.out.println("Bob's public key: " + bPublic);

// Intercepting communication (MITM attack)
BigInteger interceptedPublicKey = aPublic; // Pretending to be Bob, intercepting Alice's public key
System.out.println("Intercepted public key: " + interceptedPublicKey);

// Compute shared secret keys
BigInteger aliceSharedSecret = interceptedPublicKey.modPow(bPrivate, p);
BigInteger bobSharedSecret = bPublic.modPow(aPrivate, p);

// Both Alice and Bob calculate the same shared secret
System.out.println("Shared secret (Alice's side): " + aliceSharedSecret);
```

```
        System.out.println("Shared secret (Bob's side): " + bobSharedSecret);

        scanner.close();
    }

    // Helper method to generate private key
    private static BigInteger generatePrivateKey(BigInteger p) {
        SecureRandom random = new SecureRandom();
        return new BigInteger(p.bitLength(), random).mod(p.subtract(BigInteger.ONE)).add(BigInteger.ONE);
    }
}
```

Compute the initial subkey and Round 1 subkey for the AES algorithm with 128-bit keys. The main key value is:0f1571c947d9e8590cb7add6af7f6798 in java

```
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;

public class AESKeyExpansion {

    public static void main(String[] args) throws NoSuchAlgorithmException {
        // Main key value as hexadecimal string
        String mainKeyHex = "0f1571c947d9e8590cb7add6af7f6798";

        // Convert main key from hexadecimal to byte array
        byte[] mainKeyBytes = hexStringToByteArray(mainKeyHex);

        // Print the initial subkey
        System.out.println("Initial Subkey (Main Key): " + byteArrayToHexString(mainKeyBytes));

        // Perform key expansion to generate round keys
        SecretKey[] roundKeys = expandAESKey(mainKeyBytes);

        // Print the Round 1 subkey
```

```
        byte[] round1Key = roundKeys[1].getEncoded();
        System.out.println("Round 1 Subkey: " + byteArrayToHexString(round1Key));
    }

    // AES key expansion function
    public static SecretKey[] expandAESKey(byte[] mainKey) {
        SecretKey[] roundKeys = new SecretKey[11]; // AES has 10 rounds, plus the initial key

        roundKeys[0] = new SecretKeySpec(mainKey, "AES");

        for (int i = 1; i < 11; i++) {
            byte[] prevKeyBytes = roundKeys[i - 1].getEncoded();
            byte[] newKeyBytes = new byte[16];

            // Perform key schedule core operations
            for (int j = 0; j < 4; j++) {
                newKeyBytes[j] = (byte) (prevKeyBytes[j] ^ AESKeyExpansion.Rcon(i) ^
AESKeyExpansion.subWord(AESKeyExpansion.rotWord(prevKeyBytes, j * 4)));
                for (int k = 1; k < 4; k++) {
                    newKeyBytes[j + 4 * k] = (byte) (prevKeyBytes[j + 4 * k] ^ newKeyBytes[j + 4 * (k - 1)]);
                }
            }
            roundKeys[i] = new SecretKeySpec(newKeyBytes, "AES");
        }

        return roundKeys;
    }

    // AES key expansion helper functions
    private static byte Rcon(int i) {
        int[] Rcon = {
                0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36
        };
        return (byte) Rcon[i - 1];
    }
```

```java
private static byte[] subWord(byte[] word) {
    byte[] result = new byte[word.length];
    for (int i = 0; i < word.length; i++) {
        result[i] = (byte) AESKeyExpansion.subByte((int) word[i]);
    }
    return result;
}

private static int subByte(int b) {
    int[] SBox = {
            0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
            0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
            0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
            0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
            0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
            0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
            0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
            0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
            0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
            0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
            0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
            0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
            0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
            0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
            0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
            0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
    };
    return SBox[b & 0xFF];
}

private static byte[] rotWord(byte[] word, int start) {
    byte[] result = new byte[word.length];
    for (int i = 0; i < word.length; i++) {
        result[i] = word[(start + i + 1) % word.length];
    }
    return result;
```

```
    }

    // Helper methods for converting byte array to hexadecimal string and vice versa
    public static byte[] hexStringToByteArray(String hexString) {
        int len = hexString.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(hexString.charAt(i), 16) << 4)
                    + Character.digit(hexString.charAt(i + 1), 16));
        }
        return data;
    }

    public static String byteArrayToHexString(byte[] byteArray) {
        StringBuilder result = new StringBuilder();
        for (byte b : byteArray) {
            result.append(String.format("%02X", b));
        }
        return result.toString();
    }
}
```

Consider the Diffie-Hellman protocol with q = 3 and p = 353. Alice chooses x = 97 and Bob chooses y = 233. Compute X, Y, and the key in java

```
import java.math.BigInteger;

public class DiffieHellman {

    public static void main(String[] args) {
        // Given parameters
        BigInteger q = BigInteger.valueOf(3);
```

```java
        BigInteger p = BigInteger.valueOf(353);
        BigInteger g = q; // In this case, g equals q

        // Alice's private key
        BigInteger x = BigInteger.valueOf(97);

        // Bob's private key
        BigInteger y = BigInteger.valueOf(233);

        // Compute X and Y
        BigInteger X = g.modPow(x, p);
        BigInteger Y = g.modPow(y, p);

        // Compute the shared key
        BigInteger sharedKey_Alice = Y.modPow(x, p);
        BigInteger sharedKey_Bob = X.modPow(y, p);

        // Print the computed values
        System.out.println("X (Alice's public key): " + X);
        System.out.println("Y (Bob's public key): " + Y);
        System.out.println("Shared key (Alice's side): " + sharedKey_Alice);
        System.out.println("Shared key (Bob's side): " + sharedKey_Bob);
    }
}
```

```java
// Jav code to implement Hill Cipher
class hill {

    // Following function generates the
    // key matrix for the key string
    static void getKeyMatrix(String key, int keyMatrix[][]) {
        int k = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                keyMatrix[i][j] = (key.charAt(k)) % 65;
                k++;
            }
        }
    }

    // Following function encrypts the message
    static void encrypt(int cipherMatrix[][],
            int keyMatrix[][],
            int messageVector[][]) {
        int x, i, j;
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 1; j++) {
                cipherMatrix[i][j] = 0;

                for (x = 0; x < 3; x++) {
                    cipherMatrix[i][j] += keyMatrix[i][x] * messageVector[x][j];
                }

                cipherMatrix[i][j] = cipherMatrix[i][j] % 26;
            }
        }
    }

    // Function to implement Hill Cipher
    static void HillCipher(String message, String key) {
        // Get key matrix from the key string
        int[][] keyMatrix = new int[3][3];
        getKeyMatrix(key, keyMatrix);

        int[][] messageVector = new int[3][1];

        // Generate vector for the message
        for (int i = 0; i < 3; i++)
            messageVector[i][0] = (message.charAt(i)) % 65;

        int[][] cipherMatrix = new int[3][1];

        // Following function generates
        // the encrypted vector
        encrypt(cipherMatrix, keyMatrix, messageVector);

        String CipherText = "";
```

```
        // Generate the encrypted text from
        // the encrypted vector
        for (int i = 0; i < 3; i++)
            CipherText += (char) (cipherMatrix[i][0] + 65);

        // Finally print the ciphertext
        System.out.println("Ciphertext:" + CipherText);
    }

    // Driver code
    public static void main(String[] args) {
        // Get the message to be encrypted
        String message = "CRYPTOGRAPHY";

        // Get the key
        String key = "GYBNQKURP";

        HillCipher(message, key);
    }
}
```

```java
// Java code to implement Vigenere Cipher

class vignere {

    // This function generates the key in
    // a cyclic manner until it's length isi'nt
    // equal to the length of original text
    static String generateKey(String str, String key) {
        int x = str.length();

        for (int i = 0;; i++) {
            if (x == i)
                i = 0;
            if (key.length() == str.length())
                break;
            key += (key.charAt(i));
        }
        return key;
    }

    // This function returns the encrypted text
    // generated with the help of the key
    static String cipherText(String str, String key) {
        String cipher_text = "";

        for (int i = 0; i < str.length(); i++) {
            // converting in range 0-25
            int x = (str.charAt(i) + key.charAt(i)) % 26;

            // convert into alphabets(ASCII)
            x += 'A';

            cipher_text += (char) (x);
        }
        return cipher_text;
    }

    // This function decrypts the encrypted text
    // and returns the original text
    static String originalText(String cipher_text, String key) {
        String orig_text = "";

        for (int i = 0; i < cipher_text.length() &&
                i < key.length(); i++) {
            // converting in range 0-25
            int x = (cipher_text.charAt(i) -
                    key.charAt(i) + 26) % 26;

            // convert into alphabets(ASCII)
            x += 'A';
            orig_text += (char) (x);
```

```java
        }
        return orig_text;
    }

    // This function will convert the lower case character to Upper case
    static String LowerToUpper(String s) {
        StringBuffer str = new StringBuffer(s);
        for (int i = 0; i < s.length(); i++) {
            if (Character.isLowerCase(s.charAt(i))) {
                str.setCharAt(i, Character.toUpperCase(s.charAt(i)));
            }
        }
        s = str.toString();
        return s;
    }

    // Driver code
    public static void main(String[] args) {
        String Str = "CRYPTOGRAPHY";
        String Keyword = "PRATIK";

        String str = LowerToUpper(Str);
        String keyword = LowerToUpper(Keyword);

        String key = generateKey(str, keyword);
        String cipher_text = cipherText(str, key);

        System.out.println("Ciphertext : "
                + cipher_text + "\n");

        System.out.println("Original/Decrypted Text : "
                + originalText(cipher_text, key));
    }
}
```

```java
//A Java Program to illustrate Caesar Cipher Technique
class caesar {
    // Encrypts text using a shift of s
    public static StringBuffer encrypt(String text, int s) {
        StringBuffer result = new StringBuffer();

        for (int i = 0; i < text.length(); i++) {
            if (Character.isUpperCase(text.charAt(i))) {
                char ch = (char) (((int) text.charAt(i) +
                        s - 65) % 26 + 65);
                result.append(ch);
            } else {
                char ch = (char) (((int) text.charAt(i) +
                        s - 97) % 26 + 97);
                result.append(ch);
            }
        }
        return result;
    }

    // Driver code
    public static void main(String[] args) {
        String text = "CRYPTOGRAPHY";
        int s = 4;
        System.out.println("Text : " + text);
        System.out.println("Shift : " + s);
        System.out.println("Cipher: " + encrypt(text, s));
    }
}
```

```java
// Java program to implement Playfair Cipher
import java.util.*;

public class playfair {
    static int SIZE = 30;

    // Function to convert the string to lowercase
    static void toLowerCase(char plain[], int ps) {
        int i;
        for (i = 0; i < ps; i++) {
            if (plain[i] > 64 && plain[i] < 91)
                plain[i] += 32;
        }
    }

    // Function to remove all spaces in a string
    static int removeSpaces(char[] plain, int ps) {
        int i, count = 0;
        for (i = 0; i < ps; i++)
            if (plain[i] != '\u0000')
                plain[count++] = plain[i];

        return count;
    }

    // Function to generate the 5x5 key square
    static void generateKeyTable(char key[], int ks, char keyT[][]) {
        int i, j, k, flag = 0;

        // a 26 character hashmap
        // to store count of the alphabet
        int dicty[] = new int[26];
        for (i = 0; i < ks; i++) {
            if (key[i] != 'j')
                dicty[key[i] - 97] = 2;
        }

        dicty['j' - 97] = 1;

        i = 0;
        j = 0;

        for (k = 0; k < ks; k++) {
            if (dicty[key[k] - 97] == 2) {
                dicty[key[k] - 97] -= 1;
                keyT[i][j] = key[k];
                j++;
                if (j == 5) {
                    i++;
                    j = 0;
                }
```

```
        }
    }

    for (k = 0; k < 26; k++) {
        if (dicty[k] == 0) {
            keyT[i][j] = (char) (k + 97);
            j++;
            if (j == 5) {
                i++;
                j = 0;
            }
        }
    }
}

// Function to search for the characters of a digraph
// in the key square and return their position
static void search(char keyT[][], char a, char b, int arr[]) {
    int i, j;

    if (a == 'j')
        a = 'i';
    else if (b == 'j')
        b = 'i';

    for (i = 0; i < 5; i++) {

        for (j = 0; j < 5; j++) {

            if (keyT[i][j] == a) {
                arr[0] = i;
                arr[1] = j;
            } else if (keyT[i][j] == b) {
                arr[2] = i;
                arr[3] = j;
            }
        }
    }
}

// Function to find the modulus with 5
static int mod5(int a) {
    return (a % 5);
}

// Function to make the plain text length to be even
static int prepare(char str[], int ptrs) {
    if (ptrs % 2 != 0) {
        str[ptrs++] = 'z';
        str[ptrs] = '\0';
    }
    return ptrs;
}
```

```java
    // Function for performing the encryption
    static void encrypt(char str[], char keyT[][], int ps) {
        int i;
        int[] a = new int[4];

        for (i = 0; i < ps; i += 2) {

            search(keyT, str[i], str[i + 1], a);

            if (a[0] == a[2]) {
                str[i] = keyT[a[0]][mod5(a[1] + 1)];
                str[i + 1] = keyT[a[0]][mod5(a[3] + 1)];
            } else if (a[1] == a[3]) {
                str[i] = keyT[mod5(a[0] + 1)][a[1]];
                str[i + 1] = keyT[mod5(a[2] + 1)][a[1]];
            } else {
                str[i] = keyT[a[0]][a[3]];
                str[i + 1] = keyT[a[2]][a[1]];
            }
        }
    }

    // Function to encrypt using Playfair Cipher
    static void encryptByPlayfairCipher(char str[], char key[]) {
        int ps;
        int ks;
        char[][] keyT = new char[5][5];

        // Key
        ks = key.length;
        ks = removeSpaces(key, ks);
        toLowerCase(key, ks);

        // Plaintext
        ps = str.length;
        toLowerCase(str, ps);
        ps = removeSpaces(str, ps);

        ps = prepare(str, ps);

        generateKeyTable(key, ks, keyT);

        encrypt(str, keyT, ps);
    }

    static void strcpy(char[] arr, String s) {
        for (int i = 0; i < s.length(); i++) {
            arr[i] = s.charAt(i);
        }
    }

    // Driver code
    public static void main(String[] args) {
        char str[] = new char[SIZE];
```

```
        char key[] = new char[SIZE];

        // Key to be encrypted

        strcpy(key, "Cryptography");
        System.out.println("Key text: " + String.valueOf(key));

        // Plaintext to be encrypted
        strcpy(str, "network");
        System.out.println("Plain text: " + String.valueOf(str));

        // encrypt using Playfair Cipher
        encryptByPlayfairCipher(str, key);

        System.out.println("Cipher text: " + String.valueOf(str));
    }

}
```