Practical 1

Implement Caesar cipher encryption-decryption.

Code:

```
//Encryption
plaintext = input("Enter the value of Plaintext: ")
key = int(input("Enter the value of key: "))

# Encryption of Plaintext to Cipher text
def encrypt(p, k):
    cipher = ""
    for i in p:
        if i.isupper():
            cipher += chr((((ord(i) + k) - 65) % 26) + 65)
        else:
            cipher += chr((((ord(i) + k) - 97) % 26) + 97)
    return cipher

ciphertext = encrypt(plaintext, key)
print("Ciphertext for the plain text:", plaintext, "is", ciphertext)
```

OutPut:

Enter the value of Plaintext: information
Enter the value of key: 4
Ciphertext for the plain text: information is mrjsvqexmsr

```
#Decryption
import nltk
from nltk.corpus import wordnet
```

```
# Download WordNet data (you only need to do this once)
# nltk.download('wordnet')
#Decryption
def is_english_word(word):
    # Check if the word exists in WordNet
    return len(wordnet.synsets(word)) > 0

def crypt_analysis(char):
    for j in range(1,27):
        plain=""
        for i in char:
            if i.isupper():
                plain += chr((((ord(i) - j) - 65) % 26) + 65)
            else:
                plain += chr((((ord(i) - j) - 97) % 26) + 97)
        print("The key value is :", j)
        print("The Decoded text is:",plain)

        is_english_word(plain)

        if is_english_word(plain):
            print("\n")
            print(f"'{plain}' is a valid English word.")

        print("------------------------------")
char=input("Enter The Text To be Decoded:")
crypt_analysis(char)
```

Output:

```
Enter The Text To be Decoded:mrjsvqexmsr
The key value is : 1
The Decoded text is: lqirupdwlrq
------------------------------
The key value is : 2
The Decoded text is: kphqtocvkqp
------------------------------
The key value is : 3
The Decoded text is: jogpsnbujpo
------------------------------
The key value is : 4
The Decoded text is: information


'information' is a valid English word.
------------------------------
The key value is : 5
The Decoded text is: hmenqlzshnm
------------------------------
The key value is : 6
The Decoded text is: gldmpkyrgml
------------------------------
The key value is : 7
The Decoded text is: fkclojxqflk
------------------------------
The key value is : 8
The Decoded text is: ejbkniwpekj
------------------------------
The key value is : 9
The Decoded text is: diajmhvodji
------------------------------
The key value is : 10
The Decoded text is: chzilguncih
------------------------------
The key value is : 11
The Decoded text is: bgyhkftmbhg
------------------------------
```

The key value is : 12
The Decoded text is: afxgjeslagf

The key value is : 13
The Decoded text is: zewfidrkzfe

The key value is : 14
The Decoded text is: ydvehcqjyed

The key value is : 15
The Decoded text is: xcudgbpixdc

The key value is : 16
The Decoded text is: wbtcfaohwcb

The key value is : 17
The Decoded text is: vasbezngvba

The key value is : 18
The Decoded text is: uzradymfuaz

The key value is : 19
The Decoded text is: tyqzcxletzy

The key value is : 20
The Decoded text is: sxpybwkdsyx

The key value is : 21
The Decoded text is: rwoxavjcrxw

The key value is : 22
The Decoded text is: qvnwzuibqwv

The key value is : 23
The Decoded text is: pumvythapvu

The key value is : 24
The Decoded text is: otluxsgzout

```
The key value is : 25
The Decoded text is: nsktwrfynts
-------------------------------
The key value is : 26
The Decoded text is: mrjsvqexmsr
-----------------------------
```

Practical 2

Implement Monoalphabetic cipher encryption-decryption.

Code:

```
s = input("Enter the string: ")
key = ['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'a', 'b', 'c']

def encrypt(s):
    result = ""
    for char in s:
        if char.isalpha():
            r = (ord(char) - 97)
            result += key[r]
        else:
            result += char
    return result

encrypted_string = encrypt(s)
print("Encrypted string:", encrypted_string)

def decrypt(m):
    result = ""
    for char in m:
        if char.isalpha():
            for i in range(len(key)):
                if key[i] == char:
                    r = (i + 97)
```

```
                    result += chr(r)
                    break
        else:
            result += char
    return result


decrypted_string = decrypt(encrypted_string)
print("Decrypted string:", decrypted_string)
```

OutPut:

Enter the string: information
Encrypted string: lqirupdwlrq
Decrypted string: information

Practical 3

Implement Playfair cipher encryption-decryption.

Code:

```
play = [['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']]
p = input("Enter the text: ")

k = 0
i = 0
plain = ""
while k < len(p) - 1:
    if p[k] == p[k + 1]:
        plain += p[k]
        plain += 'x'
        k = k + 1
    else:
        plain += p[k]
        plain += p[k + 1]
        k = k + 2
if p[len(p) - 1] != plain[len(plain) - 1] or len(p) % 2 == 0:
    plain += p[k]

if len(plain) % 2 != 0:
    plain += 'x'

p = plain
print("Intermediate text:", p)

cipher = ""
for k in range(0, len(p), 2):
```

```
    for i in range(0, 5):
        for j in range(0, 5):
            if play[i][j] == p[k] or (p[k] == 'j' and play[i][j] == 'i'):
                row1 = i
                column1 = j
            if play[i][j] == p[k + 1] or (p[k] == 'j' and play[i][j] == 'i'):
                row2 = i
                column2 = j
    if row1 == row2:
        cipher += play[row1][(column1 + 1) % 5]
        cipher += play[row2][(column2 + 1) % 5]
    elif column1 == column2:
        cipher += play[(row1 + 1) % 5][column1]
        cipher += play[(row2 + 1) % 5][column1]
    else:
        cipher += play[row1][column2]
        cipher += play[row2][column1]

print("The cipher is:", cipher)
```

Output:
Enter the text: information
Intermediate text: informationx
The cipher is: hoilwrdqotsc

Practical 4

Implement Polyalphabetic cipher encryption-decryption.

```
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)

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)

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)

if __name__ == "__main__":
    string = input("Enter the string: ")
    keyword = input("Enter the keyword: ")
```

```
key = generateKey(string, keyword)
cipher_text = cipherText(string, key)
print("Ciphertext:", cipher_text)
print("Original/Decrypted Text:", originalText(cipher_text, key))
```

Output:

Enter The String For Encryption:INFORMATIONSECURITY
Enter The keyword:COMPUTER
Ciphertext : KBRDLFEKKCZHYVYIKHK
Original/Decrypted Text : INFORMATIONSECURITY

Practical 5

Implement Hill cipher encryption-decryption..

Code:

```
def get_key_matrix():
    key = input("Enter key matrix: ").upper()
    square_root = int(len(key) ** 0.5)
    if square_root * square_root != len(key):
        print("Cannot form a square matrix")
        return None
    key_matrix = [[ord(key[i]) - 65 for i in range(j, j + square_root)] for j in range(0, len(key), square_root)]
    return key_matrix

def is_valid_matrix(key_matrix):
    det = key_matrix[0][0] * key_matrix[1][1] - key_matrix[0][1] * key_matrix[1][0]
    if det % 26 == 0:
        raise Exception("Det equals to zero, invalid key matrix!")

def reverse_matrix(key_matrix):
    det = key_matrix[0][0] * key_matrix[1][1] - key_matrix[0][1] * key_matrix[1][0]
    det_inverse = pow(det, -1, 26)
    adjugate = [[0, 0], [0, 0]]
    adjugate[0][0] = (key_matrix[1][1] * det_inverse) % 26
    adjugate[0][1] = ((-key_matrix[0][1]) * det_inverse) % 26
    adjugate[1][0] = ((-key_matrix[1][0]) * det_inverse) % 26
    adjugate[1][1] = (key_matrix[0][0] * det_inverse) % 26
    return adjugate

def hill_cipher_encrypt(message, key_matrix):
    message = message.replace(" ", "").upper()
```

```
    while len(message) % len(key_matrix) != 0:
        message += "Q"
    encrypted_message = ""
    for i in range(0, len(message), len(key_matrix)):
        message_chunk = message[i:i+len(key_matrix)]
        message_vector = [[ord(char) - 65] for char in message_chunk]
        encrypted_vector = [[0], [0]]
        for j in range(len(key_matrix)):
            for k in range(len(message_vector)):
                encrypted_vector[j][0] += key_matrix[j][k] * message_vector[k][0]
            encrypted_vector[j][0] %= 26
        encrypted_chunk = [chr(char[0] + 65) for char in encrypted_vector]
        encrypted_message += "".join(encrypted_chunk)
    return encrypted_message

def hill_cipher_decrypt(encrypted_message, key_matrix):
    encrypted_message = encrypted_message.replace(" ", "").upper()
    decrypted_message = ""
    reverse_key_matrix = reverse_matrix(key_matrix)
    for i in range(0, len(encrypted_message), len(key_matrix)):
        encrypted_chunk = encrypted_message[i:i+len(key_matrix)]
        encrypted_vector = [[ord(char) - 65] for char in encrypted_chunk]
        decrypted_vector = [[0], [0]]
        for j in range(len(reverse_key_matrix)):
            for k in range(len(encrypted_vector)):
                decrypted_vector[j][0] += reverse_key_matrix[j][k] * encrypted_vector[k][0]
            decrypted_vector[j][0] %= 26
        decrypted_chunk = [chr(char[0] + 65) for char in decrypted_vector]
        decrypted_message += "".join(decrypted_chunk)
    return decrypted_message
```

```
def main():
    print("Hill Cipher Implementation (2x2)")
    print("------------------------")
    print("1. Encrypt text (A=0,B=1,...Z=25)")
    print("2. Decrypt text (A=0,B=1,...Z=25)")
    print("3. Encrypt text (A=1,B=2,...Z=26)")
    print("4. Decrypt text (A=1,B=2,...Z=26)")

    choice = input("Select your choice: ")

    key_matrix = get_key_matrix()
      if key_matrix is None:
        return

    is_valid_matrix(key_matrix)

    if choice == "1":
        phrase = input("Enter phrase to encrypt: ")
        encrypted_phrase = hill_cipher_encrypt(phrase, key_matrix)
        print("Encrypted phrase:", encrypted_phrase)
    elif choice == "2":
        encrypted_phrase = input("Enter phrase to decrypt: ")
        decrypted_phrase = hill_cipher_decrypt(encrypted_phrase, key_matrix)
        print("Decrypted phrase:", decrypted_phrase)
    elif choice == "3":
        phrase = input("Enter phrase to encrypt: ")
        encrypted_phrase = hill_cipher_encrypt(phrase, key_matrix)
        print("Encrypted phrase:", encrypted_phrase)
    elif choice == "4":
        encrypted_phrase = input("Enter phrase to decrypt: ")
        decrypted_phrase = hill_cipher_decrypt(encrypted_phrase, key_matrix)
```

```
        print("Decrypted phrase:", decrypted_phrase)
    else:
        print("Invalid choice")


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

OutPut:

#Encryption
1. Encrypt text (A=0,B=1,...Z=25)
2. Decrypt text (A=0,B=1,...Z=25)
3. Encrypt text (A=1,B=2,...Z=26)
4. Decrypt text (A=1,B=2,...Z=26)

Select your choice: 1
Enter key matrix: info
Enter phrase to encrypt: east
Encrypted phrase: GUBS

#Decryption
Hill Cipher Implementation (2x2)
----
1. Encrypt text (A=0,B=1,...Z=25)
2. Decrypt text (A=0,B=1,...Z=25)
3. Encrypt text (A=1,B=2,...Z=26)
4. Decrypt text (A=1,B=2,...Z=26)

Select your choice: 2
Enter key matrix: info
Enter phrase to decrypt: gubs
Decrypted phrase: EAST

Practical 6

To implement Simple DES or AES.

Code:

```
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_str = ""
    for i in range(len(s)):
        bin_str += mp[s[i]]
    return bin_str

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_str = ""
    for i in range(0, len(s), 4):
        chunk = ""
        chunk += s[i]
        chunk += s[i + 1]
        chunk += s[i + 2]
        chunk += s[i + 3]
        hex_str += mp[chunk]
    return hex_str

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

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

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

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


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


# Tables for encryption
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]

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]

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]

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_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)
    pt = permute(pt, initial_perm, 64)
    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        right_expanded = permute(right, exp_d, 48)
        xor_x = xor(right_expanded, rkb[i])
        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)
        sbox_str = permute(sbox_str, per, 32)
        result = xor(left, sbox_str)
        left = result
        if i != 15:
            left, right = right, left
    combine = left + right
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text

pt = "123456ABCD132536"
key = "AABB09182736CCDD"

key = hex2bin(key)
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]

key = permute(key, keyp, 56)
shift_table = [1, 1, 2, 2, 2, 2, 2, 2,
               1, 2, 2, 2, 2, 2, 2, 1]

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]

left = key[0:28]
right = key[28:56]
rkb = []
rk = []
for i in range(0, 16):
    left = shift_left(left, shift_table[i])
    right = shift_left(right, shift_table[i])
    combine_str = left + right
    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)
```

OutPut:

Encryption

After initial permutation 14A7D67818CA18AD

Round 1 18CA18AD 5A78E394 194CD072DE8C

Round 2 5A78E394 4A1210F6 4568581ABCCE

Round 3 4A1210F6 B8089591 06EDA4ACF5B5

Round 4 B8089591 236779C2 DA2D032B6EE3

Round 5 236779C2 A15A4B87 69A629FEC913

Round 6 A15A4B87 2E8F9C65 C1948E87475E

Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0

Round 8 A9FC20A3 308BEE97 34F822F0C66D

Round 9 308BEE97 10AF9D37 84BB4473DCCC

Round 10 10AF9D37 6CA6CB20 02765708B5BF

Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5

Round 12 FF3C485F 22A5963B C2C1E96A4BF3

Round 13 22A5963B 387CCDAA 99C31397C91F

Round 14 387CCDAA BD2DD2AB 251B8BC717D0

Round 15 BD2DD2AB CF26B472 3330C5D9A36D

Round 16 19BA9212 CF26B472 181C5D75C66D

Cipher Text : C0B7A8D05F3A829C

Decryption

After initial permutation 19BA9212CF26B472

Round 1 CF26B472 BD2DD2AB 181C5D75C66D

Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D

Round 3 387CCDAA 22A5963B 251B8BC717D0

Round 4 22A5963B FF3C485F 99C31397C91F

Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3

Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5

Round 7 10AF9D37 308BEE97 02765708B5BF

Round 8 308BEE97 A9FC20A3 84BB4473DCCC

Round 9 A9FC20A3 2E8F9C65 34F822F0C66D

Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0

Round 11 A15A4B87 236779C2 C1948E87475E

Round 12 236779C2 B8089591 69A629FEC913

Round 13 B8089591 4A1210F6 DA2D032B6EE3

Round 14 4A1210F6 5A78E394 06EDA4ACF5B5

Round 15 5A78E394 18CA18AD 4568581ABCCE

Round 16 14A7D678 18CA18AD 194CD072DE8C

Plain Text : 123456ABCD132536

Practical 7

Implement Diffi-Hellmen Key exchange Method.

Code:

```
# Diffie-Hellman Code



def prime_checker(p):
    # Checks If the number entered is a Prime Number or not
    if p < 1:
        return -1
    elif p > 1:
        if p == 2:
            return 1
        for i in range(2, p):
            if p % i == 0:
                return -1
            return 1



def primitive_check(g, p, L):
    # Checks If The Entered Number Is A Primitive Root Or Not
    for i in range(1, p):
        L.append(pow(g, i) % p)
    for i in range(1, p):
        if L.count(i) > 1:
            L.clear()
            return -1
        return 1
```

```
l = []
while 1:
    P = int(input("Enter P : "))
    if prime_checker(P) == -1:
        print("Number Is Not Prime, Please Enter Again!")
        continue
    break

while 1:
    G = int(input(f"Enter The Primitive Root Of {P} : "))
    if primitive_check(G, P, l) == -1:
        print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")
        continue
    break

#  Private Keys
x1, x2 = int(input("Enter The Private Key Of User 1 : ")), int(
    input("Enter The Private Key Of User 2 : "))
while 1:
    if x1 >= P or x2 >= P:
        print(f"Private Key Of Both The Users Should Be Less Than {P}!")
        continue
    break

# Calculate Public Keys
y1, y2 = pow(G, x1) % P, pow(G, x2) % P

# Generate Secret Keys
k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P

print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")
```

```
if k1 == k2:
    print("Keys Have Been Exchanged Successfully")
else:
    print("Keys Have Not Been Exchanged Successfully")
```

Output:

```
Enter P : 23
Enter The Primitive Root Of 23 : 9
Number Is Not A Primitive Root Of 23, Please Try Again!
Enter The Primitive Root Of 23 : 6
Number Is Not A Primitive Root Of 23, Please Try Again!
Enter The Primitive Root Of 23 : 5
Enter The Private Key Of User 1 : 3
Enter The Private Key Of User 2 : 4

Secret Key For User 1 Is 18
Secret Key For User 2 Is 18
```

Practical 8

Implement RSA encryption-decryption algorithm.

```
from decimal import Decimal

def gcd(k, totient):
    while totient != 0:
        c = k % totient
        k = totient
        totient = c
    return k

# Input variables
d = 0
p = eval(input("Enter value of p: "))
q = eval(input("Enter value of q: "))
message = eval(input("Enter message: "))

# Calculate n
n = p * q

# Calculate totient
totient = (p - 1) * (q - 1)

# Calculate K
for k in range(2, totient):
    if gcd(k, totient) == 1:
        break

for i in range(1, 10):
    x = 1 + i * totient
```

```
    if x % k == 0:
        d = int(x / k)
        break

local_cipher = Decimal(0)
local_cipher = pow(message, k)
cipher_text = local_cipher % n

decrypt_t = Decimal(0)
decrypt_t = pow(cipher_text, d)
decrypted_text = decrypt_t % n

print('n = ' + str(n))
print('k = ' + str(k))
print('totient = ' + str(totient))
print('d = ' + str(d))
print('Cipher Text = ' + str(cipher_text))
print('Decrypted Text = ' + str(decrypted_text))
```

Output:

```
Enter value of p: 17
Enter value of q: 19
Enter message: 45
n = 323
k = 5
totient = 288
d = 173
Cipher Text = 163
Decrypted Text = 45
```

PRACTICAL-9

Write a program to generate SHA-1 hash.

```
import hashlib


str_to_hash = input("Enter The string:")
result = hashlib.sha1(str_to_hash.encode())
print("The hexadecimal equivalent of SHA1 is: ", result.hexdigest())
```

Output:

Enter The string:abcdefghijklmnopqrstuvwxyz
The hexadecimal equivalent of SHA1 is: 32d10c7b8cf96570ca04ce37f2a19d84240d3a89

PRACTICAL-10

Implement a digital signature algorithm.

```
# Function to find gcd
# of two numbers
def euclid(m, n):

    if n == 0:
        return m
    else:
        r = m % n
        return euclid(n, r)


# Program to find
# Multiplicative inverse
def exteuclid(a, b):

    r1 = a
    r2 = b
    s1 = int(1)
    s2 = int(0)
    t1 = int(0)
    t2 = int(1)

    while r2 > 0:

        q = r1//r2
        r = r1-q * r2
        r1 = r2
        r2 = r
```

```
        s = s1-q * s2
        s1 = s2
        s2 = s
        t = t1-q * t2
        t1 = t2
        t2 = t

    if t1 < 0:
        t1 = t1 % a

    return (r1, t1)

# Enter two large prime
# numbers p and q
p = 823
q = 953
n = p * q
Pn = (p-1)*(q-1)

# Generate encryption key
# in range 1<e<Pn
key = []

for i in range(2, Pn):

    gcd = euclid(Pn, i)

    if gcd == 1:
        key.append(i)
```

```
# Select an encryption key
# from the above list
e = int(313)

# Obtain inverse of
# encryption key in Z_Pn
r, d = exteuclid(Pn, e)
if r == 1:
    d = int(d)
    print("decryption key is: ", d)

else:
    print("Multiplicative inverse for\
    the given encryption key does not \
    exist. Choose a different encryption key ")


# Enter the message to be sent
M = 19072

# Signature is created by Rahul
S = (M**d) % n

# Rahul sends M and S both to Raj
#Raj generates message M1 using the
# signature S, Rahul's public key e
# and product n.
M1 = (S**e) % n

# If M = M1 only then Raj accepts
# the message sent by Rahul.
```

```
if M == M1:
    print("As M = M1, Accept the\
    message sent by Rahul")
else:
    print("As M not equal to M1,\
    Do not accept the message\
    sent by Rahul ")
```

Output:

decryption key is: 160009
As M = M1, Accept the   message sent by Rahul