**Practical No. 1**

**Q.1 Write a program to implement symmetric encryption using Ceaser Cipher algorithm.**

**Code:**

# Implementation of Caesar Cipher

def encrypt(text, shift):

result = ""

for i in range(len(text)):

char = text[i]

if(char == " "):

result += " "

else:

if(char.isupper()):

result += chr((ord(char) + ord(shift) - 65) % 26 + 65)

else:

result += chr((ord(char) + ord(shift) - 97) % 26 + 97)

return result

text = input("Enter text to encrypt : ")

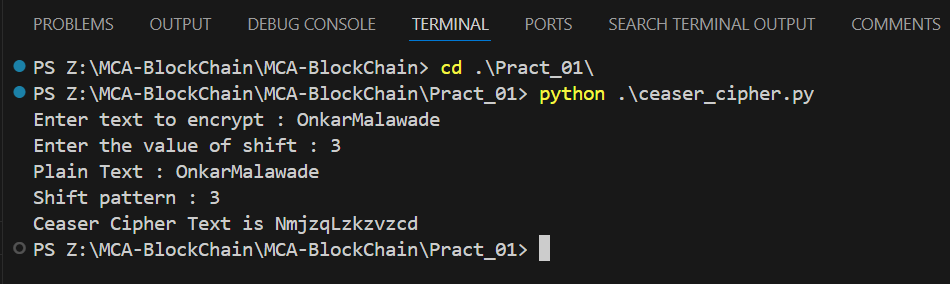
shift = input("Enter the value of shift : ")

print("Plain Text : " + text)

print("Shift pattern : " + shift)

print("Ceaser Cipher Text is " + encrypt(text, shift))

**Output:**

****

**Q.2 Write a program to implement asymmetric encryption using RSA algorithm. Generate both keys public key and private key and store it in file. Also encrypt and decrypt the message using keys.**

**Code:**

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.PublicKey import RSA

from binascii import hexlify

message = b"Public and Private keys Encryption"

private\_key = RSA.generate(1024)

public\_key = private\_key.publickey()

print(type(private\_key), type(public\_key))

private\_pem = private\_key.export\_key().decode()

public\_pem = public\_key.export\_key().decode()

print(type(private\_pem), type(public\_pem))

with open('public\_pem.pem','w') as pu:

    pu.write(public\_pem)

with open('private\_pem.pem','w') as pr:

    pr.write(private\_pem)

pr\_key = RSA.import\_key(open('private\_pem.pem','r').read())

pu\_key = RSA.import\_key(open('public\_pem.pem','r').read())

print(type(pr\_key),type(pu\_key))

cipher = PKCS1\_OAEP.new(key=pu\_key)

cipher\_text = cipher.encrypt(message)

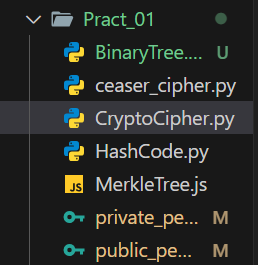
print(cipher\_text)

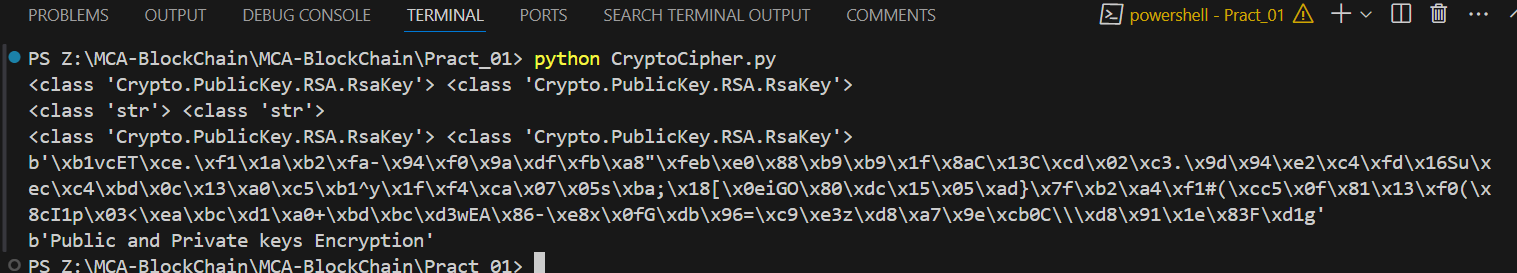
decyText = PKCS1\_OAEP.new(key=pr\_key)

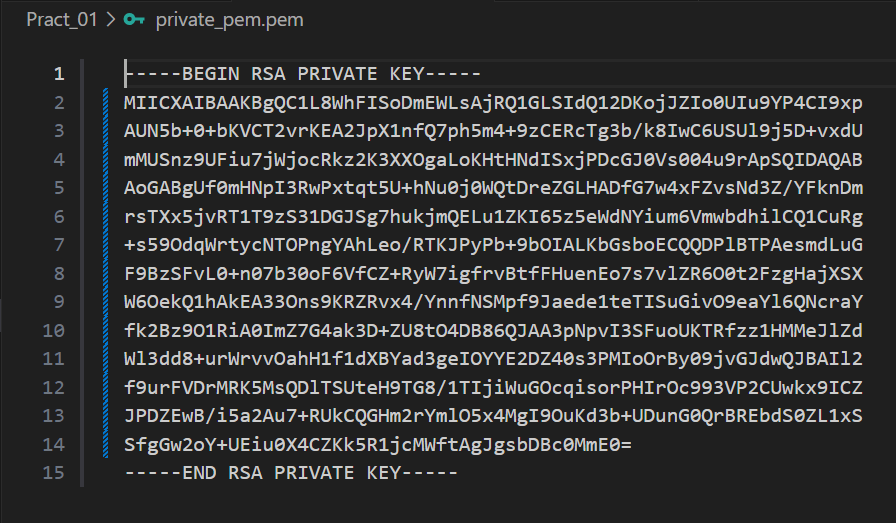
decyText\_text = decyText.decrypt(cipher\_text)

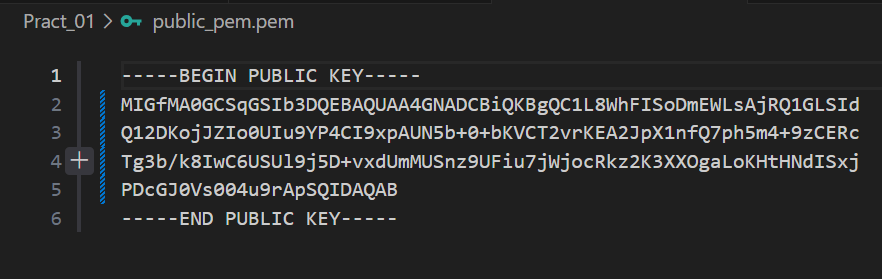
print(decyText\_text)

**Output:**

****

****

****

****

**Q.3 Write a program to demonstrate the use of Hash Functions (SHA-256).**

**Code:**

import hashlib

str = "Hello, How are you?"

encoded = str.encode()

result = hashlib.sha256(encoded)

print("String : " , end = "")

print(str)

print("Hash Value : ", end="")

print(result)

print("Hexadecimal equivalent : ", result.hexdigest())

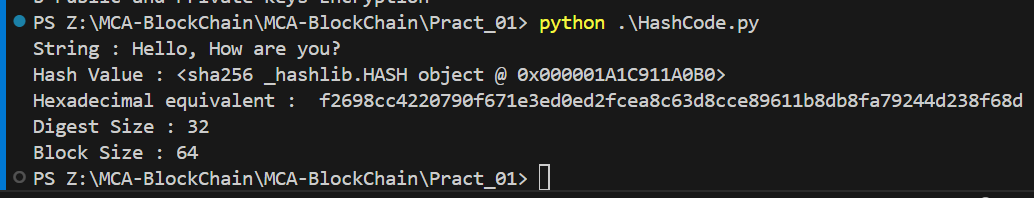
print("Digest Size : ",end="")

print(result.digest\_size)

print("Block Size : ", end = "")

print(result.block\_size)

**Output:**

****

**Q.4 Write a program to demonstrate Merkle Tree.**

**Code:**

// Program to create merkle Tree

var merkle = require('merkle');

var str = 'Fred, Bert, Bill, Bob, Alice, Trent'

var arr = str.split(',');

console.log("Input : \t \t", arr);

var tree = merkle('sha1').sync(arr)

console.log("Root Hash : \t", tree.root());

console.log("Tree Depth : \t", tree.depth());

console.log("Tree Level : \t",tree.levels());

console.log("Tree Nodes : \t",tree.nodes());

var i;

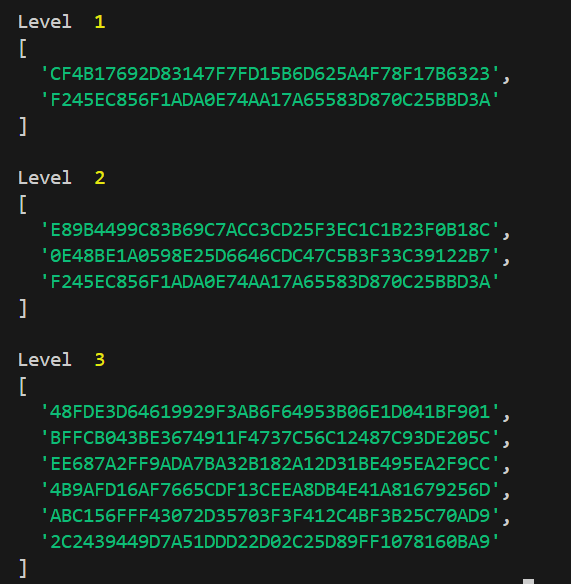
for (i = 0; i < tree.levels(); i++) {

    console.log("\nLevel ", i);

    console.log(tree.level(i));

}

**Output:**

**** ****

**Q.5 Write a program to implement Merkle Tree using JavaScript**

**Code:**

import hashlib

class MerkleTree:

    def \_\_init\_\_(self, leaves):

        self.leaves = [self.\_hash(leaf) for leaf in leaves]

        self.root = self.\_build\_tree(self.leaves)

    def \_build\_tree(self, leaves):

        if len(leaves) == 1:

            return leaves[0]

        next\_level = []

        for i in range(0, len(leaves), 2):

            if i + 1 < len(leaves):

                combined = leaves[i] + leaves[i + 1]

            else:

                combined = leaves[i] + leaves[i]

            next\_level.append(self.\_hash(combined))

        return self.\_build\_tree(next\_level)

    def \_hash(self, data):

        return hashlib.sha256(data.encode()).hexdigest()

    def get\_root(self):

        return self.root

    def get\_proof(self, index):

        proof = []

        current\_index = index

        current\_level = self.leaves

        while len(current\_level) > 1:

            if current\_index % 2 == 0:  # Even index

                if current\_index + 1 < len(current\_level):

                    proof.append(('right', current\_level[current\_index + 1]))

            else:

                proof.append(('left', current\_level[current\_index - 1]))

            current\_index //= 2

            current\_level = self.\_get\_next\_level(current\_level)

        return proof

    def \_get\_next\_level(self, current\_level):

        next\_level = []

        for i in range(0, len(current\_level), 2):

            if i + 1 < len(current\_level):

                combined = current\_level[i] + current\_level[i + 1]

            else:

                combined = current\_level[i] + current\_level[i]  # Handle odd number of nodes by duplicating

            next\_level.append(self.\_hash(combined))

        return next\_level

    def verify\_proof(self, proof, target\_hash):

        current\_hash = target\_hash

        for direction, sibling\_hash in proof:

            if direction == 'left':

                combined = sibling\_hash + current\_hash

            else:

                combined = current\_hash + sibling\_hash

            current\_hash = self.\_hash(combined)

        return current\_hash == self.root

leaves = ["leaf1", "leaf2", "leaf3", "leaf4"]

tree = MerkleTree(leaves)

root = tree.get\_root()

print("Merkle root:", root)

proof = tree.get\_proof(0)

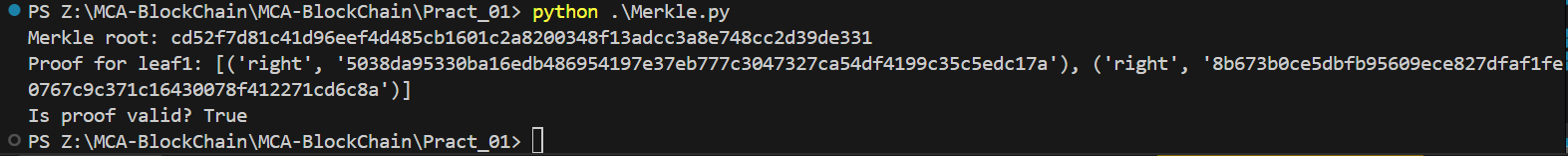
print("Proof for leaf1:", proof)

target\_hash = hashlib.sha256("leaf1".encode()).hexdigest()

is\_valid = tree.verify\_proof(proof, target\_hash)

print("Is proof valid?", is\_valid)

**Output:**



**Q.6 Write a program to implement RSA algorithm**

**Code:**

import random

def gcd(a, b):

while b:

a, b = b, a % b

return a

def multiplicative\_inverse(e, phi):

def extended\_gcd(a, b):

if a == 0:

return b, 0, 1

else:

gcd, x, y = extended\_gcd(b % a, a)

return gcd, y - (b // a) \* x, x

gcd, x, y = extended\_gcd(e, phi)

if gcd != 1:

raise Exception("Modular inverse does not exist")

else:

return x % phi

def generate\_keypair(p, q):

n = p \* q

phi = (p - 1) \* (q - 1)

e = random.randrange(2, phi)

while gcd(e, phi) != 1:

e = random.randrange(2, phi)

d = multiplicative\_inverse(e, phi)

return ((e, n), (d, n))

def encrypt(pk, plaintext):

e, n = pk

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

return ciphertext

def decrypt(pk, ciphertext):

d, n = pk

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

return ''.join(plaintext)

p = 5

q = 7

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

print("Public Key:", public)

print("Private Key:", private)

message = "Hello, World!"

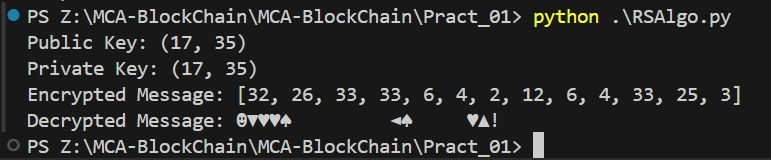
encrypted\_message = encrypt(public, message)

print("Encrypted Message:", encrypted\_message)

decrypted\_message = decrypt(private, encrypted\_message)

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

**Output:**

****

**Q.7 Implement Binary Tree using python**

**Code:**

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, value):

if self.root is None:

self.root = Node(value)

else:

self.\_insert\_recursive(self.root, value)

def \_insert\_recursive(self, current\_node, value):

if value < current\_node.value:

if current\_node.left is None:

current\_node.left = Node(value)

else:

self.\_insert\_recursive(current\_node.left, value)

else:

if current\_node.right is None:

current\_node.right = Node(value)

else:

self.\_insert\_recursive(current\_node.right, value)

def inorder\_traversal(self):

result = []

self.\_inorder\_traversal\_recursive(self.root, result)

return result

def \_inorder\_traversal\_recursive(self, current\_node, result):

if current\_node:

self.\_inorder\_traversal\_recursive(current\_node.left, result)

result.append(current\_node.value)

self.\_inorder\_traversal\_recursive(current\_node.right, result)

def preorder\_traversal(self):

result = []

self.\_preorder\_traversal\_recursive(self.root, result)

return result

def \_preorder\_traversal\_recursive(self, current\_node, result):

if current\_node:

result.append(current\_node.value)

self.\_preorder\_traversal\_recursive(current\_node.left, result)

self.\_preorder\_traversal\_recursive(current\_node.right, result)

def postorder\_traversal(self):

result = []

self.\_postorder\_traversal\_recursive(self.root, result)

return result

def \_postorder\_traversal\_recursive(self, current\_node, result):

if current\_node:

self.\_postorder\_traversal\_recursive(current\_node.left, result)

self.\_postorder\_traversal\_recursive(current\_node.right, result)

result.append(current\_node.value)

def delete(self, value):

self.root = self.\_delete\_recursive(self.root, value)

def \_delete\_recursive(self, current\_node, value):

if current\_node is None:

return current\_node

if value < current\_node.value:

current\_node.left = self.\_delete\_recursive(current\_node.left, value)

elif value > current\_node.value:

current\_node.right = self.\_delete\_recursive(current\_node.right, value)

else:

if current\_node.left is None:

return current\_node.right

elif current\_node.right is None:

return current\_node.left

min\_value\_node = self.\_find\_min\_value\_node(current\_node.right)

current\_node.value = min\_value\_node.value

current\_node.right = self.\_delete\_recursive(current\_node.right, min\_value\_node.value)

return current\_node

def \_find\_min\_value\_node(self, current\_node):

while current\_node.left is not None:

current\_node = current\_node.left

return current\_node

tree = BinaryTree()

tree.insert(5)

tree.insert(3)

tree.insert(7)

tree.insert(2)

tree.insert(4)

tree.insert(6)

tree.insert(8)

# L N R

print("Inorder traversal:", tree.inorder\_traversal())

# N L R

print("Preorder traversal:", tree.preorder\_traversal())

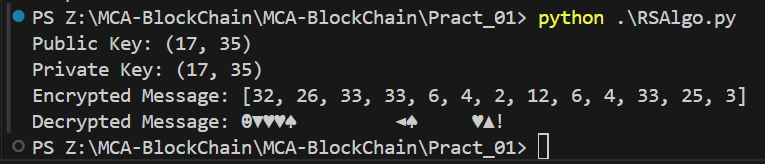
# L R N

print("Postorder traversal:", tree.postorder\_traversal())

tree.delete(4)

print("Inorder traversal after deletion:", tree.inorder\_traversal())

**Output:**

****