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:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SEARCH TERMINAL OUTPUT    COMMENTS

PS Z:\MCA-BlockChain\MCA-BlockChain> cd .\Pract_01\
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> python .\ceaser_cipher.py
Enter text to encrypt : OnkarMalawade
Enter the value of shift : 3
Plain Text : OnkarMalawade
Shift pattern : 3
Ceaser Cipher Text is NmjzqLzkzvzcd
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01>
```

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:

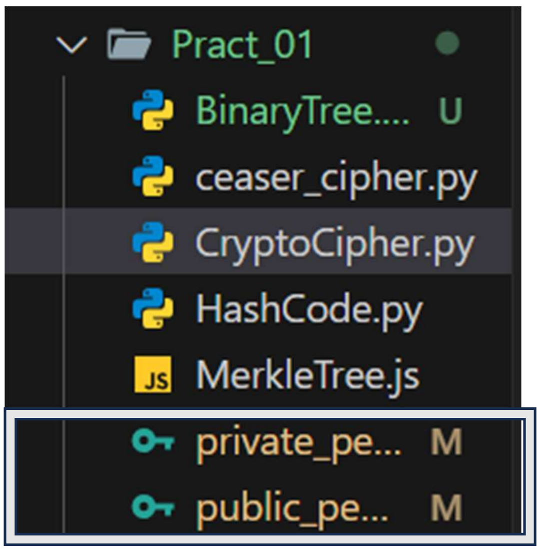
```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA
from binascii import hexlify

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)
```

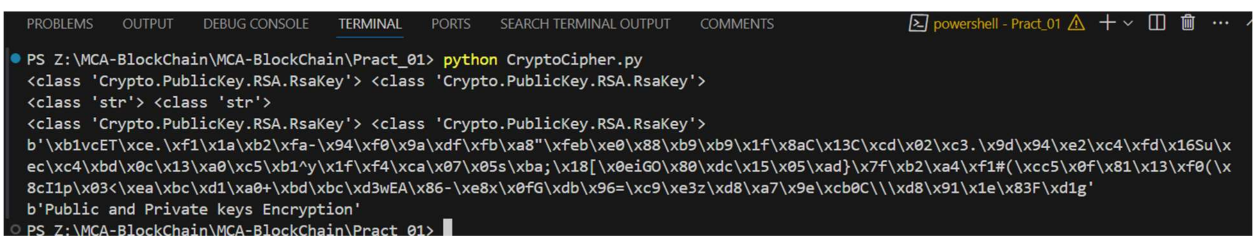
print(cipher_text)

decyText = PKCS1_OAEP.new(key=pr_key)
decyText_text = decyText.decrypt(cipher_text)

print(decyText_text)

Output:





Pract_01 > private_pem.pem


```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQC1L8WhFISoDmEWLsAjRQ1GLSIdQ12DKojJZIo0UIu9YP4CI9xp
AUN5b+0+bKVCT2vrKEA2JpX1nfQ7ph5m4+9zCERcTg3b/k8IwC6USUl9j5D+vxdU
mMUSnz9UFiu7jWjocRkz2K3XXOgaLoKHtHNdISxjPDcGJ0Vs004u9rApSQIDAQAB
AoGABgUf0mHNpI3RwPxtqt5U+hNu0j0WQtDreZGLHADfG7w4xFZvsNd3Z/YFknDm
rsTXx5jvRT1T9zS31DGJSg7hukjmQELu1ZKI65z5eWdNYium6VmwbdhilCQ1CuRg
+s59OdqWrtycNTOPngYAhLeo/RTKJPyPb+9bOIALKbGsboECQQDPlBTPAesmdLuG
F9BzSFvL0+n07b30oF6VfCZ+RyW7igfrvBtfFHuenEo7s7vlZR6O0t2FzgHajXSX
W6OekQ1hAkEA33Ons9KRZRvx4/YnnfNSMpf9Jaede1teTISuGivO9eaYl6QNcraY
fk2Bz9O1RiA0ImZ7G4ak3D+ZU8tO4DB86QJAA3pNpvI3SFuoUKTRfzz1HMMeJlZd
Wl3dd8+urWrvvOahH1f1dXBYad3geIOYYE2DZ40s3PMIoOrBy09jvGJdwQJBAIl2
f9urFVDrMRK5MsQDlTSUteH9TG8/1TIjiWuGOcqisorPHIrOc993VP2CUwkx9ICZ
JPDZEwB/i5a2Au7+RUkCQGHm2rYmlO5x4MgI9OuKd3b+UDunG0QrBREbdS0ZL1xS
SfgGw2oY+UEiu0X4CZKk5R1jcMWftAgJgsbDBc0MmE0=
-----END RSA PRIVATE KEY-----
```


Pract_01 > public_pem.pem


```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC1L8WhFISoDmEWLsAjRQ1GLSId
Q12DKojJZIo0UIu9YP4CI9xpAUN5b+0+bKVCT2vrKEA2JpX1nfQ7ph5m4+9zCERc
Tg3b/k8IwC6USUl9j5D+vxdUmMUSnz9UFiu7jWjocRkz2K3XXOgaLoKHtHNdISxj
PDcGJ0Vs004u9rApSQIDAQAB
-----END PUBLIC KEY-----
```

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:

```
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> python .\HashCode.py
String : Hello, How are you?
Hash Value : <sha256 _hashlib.HASH object @ 0x000001A1C911A0B0>
Hexadecimal equivalent :  f2698cc4220790f671e3ed0ed2fcea8c63d8cce89611b8db8fa79244d238f68d
Digest Size : 32
Block Size : 64
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01>
```

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:

```
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> node .\MerkleTree.js
Input :                  [ 'Fred', ' Bert', ' Bill', ' Bob', ' Alice', ' Trent' ]
Root Hash :        989B8BE2CC94200DD9EBFB3512DABCF7BEF7394A
Tree Depth :       3
Tree Level :       4
Tree Nodes :       6

Level  0
[ '989B8BE2CC94200DD9EBFB3512DABCF7BEF7394A' ]

Level  1
[
  'CF4B17692D83147F7FD15B6D625A4F78F17B6323',
  'F245EC856F1ADA0E74AA17A65583D870C25BBD3A'
]
```

```
Level  1
[
  'CF4B17692D83147F7FD15B6D625A4F78F17B6323',
  'F245EC856F1ADA0E74AA17A65583D870C25BBD3A'
]

Level  2
[
  'E89B4499C83B69C7ACC3CD25F3EC1C1B23F0B18C',
  '0E48BE1A0598E25D6646CDC47C5B3F33C39122B7',
  'F245EC856F1ADA0E74AA17A65583D870C25BBD3A'
]

Level  3
[
  '48FDE3D64619929F3AB6F64953B06E1D041BF901',
  'BFFCB043BE3674911F4737C56C12487C93DE205C',
  'EE687A2FF9ADA7BA32B182A12D31BE495EA2F9CC',
  '4B9AFD16AF7665CDF13CEEA8DB4E41A81679256D',
  'ABC156FFF43072D35703F3F412C4BF3B25C70AD9',
  '2C2439449D7A51DDD22D02C25D89FF1078160BA9'
]
```

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:

```
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> python .\Merkle.py
Merkle root: cd52f7d81c41d96eef4d485cb1601c2a8200348f13adcc3a8e748cc2d39de331
Proof for leaf1: [('right', '5038da95330ba16edb486954197e37eb777c3047327ca54df4199c35c5edc17a'), ('right', '8b673b0ce5dbfb95609ece827dfaf1fe
0767c9c371c16430078f412271cd6c8a')]
Is proof valid? True
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01>
```

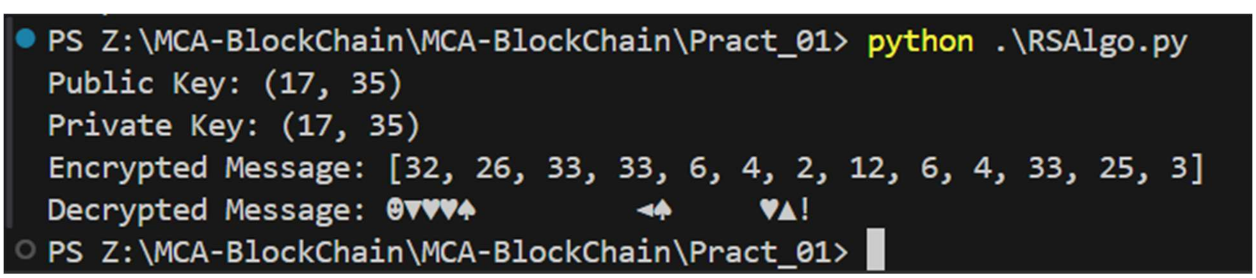
Q.6 Write a program to implement RSA algorithm

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:



```
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> python .\RSAlgo.py
Public Key: (17, 35)
Private Key: (17, 35)
Encrypted Message: [32, 26, 33, 33, 6, 4, 2, 12, 6, 4, 33, 25, 3]
Decrypted Message: ☻▼♥♥♠        ◄♠     ♥▲!
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01>
```

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:
```

```python
        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:

```
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01> python .\RSAlgo.py
Public Key: (17, 35)
Private Key: (17, 35)
Encrypted Message: [32, 26, 33, 33, 6, 4, 2, 12, 6, 4, 33, 25, 3]
Decrypted Message: ☻▼♥♥♠        ◄♠     ♥▲!
PS Z:\MCA-BlockChain\MCA-BlockChain\Pract_01>
```