Practical 1

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

   Program:

```
def encrypt(text,s):
    result = ""
    for i in range(len(text)):
        char = text[i]
        if(char == " "):
            result += " "
        else:
            if(char.isupper()):
                result += chr((ord(char) + s - 65) % 26 + 65)
            else:
                result += chr((ord(char) + s - 97) % 26 + 97)

    return result

#input text
text =input("Enter Text to encrypt: ")
s = 4

print("Plain Text:  "+ text)
print("Shift Pattern: "+str(s))
print("Cipher: " + encrypt(text,s))
```

   Output:

```
PS D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\tempCodeRunnerFile.py"
Enter Text to encrypt: I am not encrypted
Plain Text:  I am not encrypted
Shift Pattern: 4
Cipher: M eq rsx irgvctxih
```

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.**

Program:-

```
# Import necessary modules for RSA 256 cryptography
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA
from binascii import hexlify

# Define the message to be encrypted and decrypted
message = b"Public and Private Key Encryption"

# Generate a private key with a key size of 1024 bits
private_key = RSA.generate(1024)

# Generate a public key from the private key
public_key = private_key.public_key()

# Print the types of the private and public keys
print(type(private_key), type(public_key))

# Export the private and public keys in PEM format
private_pem = private_key.export_key().decode()
public_pem = public_key.export_key().decode()

# Print the types of the private and public key strings
print(type(private_pem), type(public_pem))

# Write the private and public keys to files
with open('private_pem.pem', 'w') as pr:
    pr.write(private_pem)
with open('public_pem.pem', 'w') as pr:
```

```
    pr.write(public_pem)

# Read the private and public keys from files
pr_key = RSA.import_key(open('private_pem.pem', 'r').read())
pu_key = RSA.import_key(open('public_pem.pem', 'r').read())

# Print the types of the imported private and public keys
print(type(pr_key), type(pu_key))

# Create a cipher suite for encryption using the public key
cipher_suite = PKCS1_OAEP.new(key=pu_key)

# Encrypt the message using the cipher suite
cipher_text = cipher_suite.encrypt(message)

# Print the encrypted cipher text in hexadecimal format
print("Cipher Text: ", hexlify(cipher_text))

# Create a cipher suite for decryption using the private key
plain = PKCS1_OAEP.new(key=pr_key)

# Decrypt the cipher text using the cipher suite
plain_text = plain.decrypt(cipher_text)

# Print the decrypted plain text
print(plain_text)
```

private_pem.pem

-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQCd7iW8drZXZhnOzSk0rYljYuwcWUc3uNYah3ChT8
ZO/+NgP7vl
W9hn9+qRgnE8Y/ZLiuk8+7518+btgXUKKfu9x2D0AeMQkk6q9P73/w91i
o52GstD

dRkv9KoxvSggGXcoIDw5QhjXqgpJyu3TeiL2t91oxSH+SPjWjzk/EPBW6
wIDAQAB
AoGAIOpQjR4E1ORfvp092E/O0Zr9cM8eq7tnTDIsREKXJ0HnxtihtaTVzp
16Ewen
yBlhbM8v21jwki7aU2fm1852O/yKkyc2PQqLSGThwZ97u+YK0Le6wdIjT
u3Rj+00
GgMswdhFCqHF/IyV+qy2731oHygGpybKDfZ4pZe5RhiIN6ECQQC9fmC
1B6W4V1/v
ADDVAgKegsV8cSmpLPjuCUgQk0pHo6XCPn68i/zTXwsqeZ+jxBIW9O
m+wo5aaXez
Lot0rEAxAkEA1VvcrhWbdD7yG3PGtcAXtTYW3axLOMZt62MMq5Zn0
Py0S8HPwpUA
bLpdXqhDuXChpB3e5tz++zyyiV9TfDX92wJAMoV8QSe9zk01XaJeYpw7
mIljH8+H
/PvpjoVY+lpaxojiC8zfu2NTUAOaFYQBxQbkj8xSebKjg4V1DYfOVJgM
MQJBAKGI
pTGf5kxCg+bI5v8f6lMmGnXGRkU75mi6WxNmEj+ls5NPr05wpRuslZhe
6LdzUM4C
V4qOcvYf5EQhSgHWltUCQHFGwcKa/0UpFD9pFuIWLLmixrnJVberD5v
nOS9NFiS5
PdcjpNegm9zFmP6aRawS0vWw8Zz+zaKAKJUPye+8+YY=
-----END RSA PRIVATE KEY-----

public_pem.pem

-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCd7iW8drZXZh
nOzSk0rYljYuwc
WUc3uNYah3ChT8ZO/+NgP7vlW9hn9+qRgnE8Y/ZLiuk8+7518+btgXUK
Kfu9x2D0
AeMQkk6q9P73/w91io52GstDdRkv9KoxvSggGXcoIDw5QhjXqgpJyu3Tei
L2t91o
xSH+SPjWjzk/EPBW6wIDAQAB
-----END PUBLIC KEY-----

Output:

```
PS D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\RSACryptography.py"
<class 'Crypto.PublicKey.RSA.RsaKey'> <class 'Crypto.PublicKey.RSA.RsaKey'>
<class 'str'> <class 'str'>
<class 'Crypto.PublicKey.RSA.RsaKey'> <class 'Crypto.PublicKey.RSA.RsaKey'>
Cipher Text:  b'65d5dabb581e6250d4dcfe4998981529d9e1b637a3a714fbd44b84f08a7a71fd699aa44e823d877d2bf55b9b019a04df1d
b64e46221327c90a12577eee8db5cbf332f2594168efbb126a2aac367bd96e8bc36e64c37a87bd7c5f27df7c1269ad10bc5516d53d6bd8db7c
a0c70c813f70e6383a9b19cf5f9e30e951d944782fdd'
b'Public and Private Key Encryption'
```

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

Program:

```
#SHA 256
import hashlib

string = "Hello how are you"

encoded = string.encode()

result = hashlib.sha256(encoded)

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

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

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

print("Digest Size: ",end="")
print(result.digest_size)

print("Block Size: ",end="")
print(result.block_size)
```

Output:

```
PS D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\sha256.py"
String: Hello how are you
Hash: <sha256 _hashlib.HASH object @ 0x0000025695E4A0D0>
Hexadecimal equivalant:  <built-in method hexdigest of _hashlib.HASH object at 0x0000025695E4A0D0>
Digest Size: 32
Block Size: 64
```

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

Program:

```
//Merkle Tree

var merkle = require("merkle")

// Sample data
var str = 'Fred, Bret, Bill, Bob, Alice, Trent';

// Split the string into an array of strings
var arr = str.split(',');

// Create a new merkle tree
console.log("Input \t \t ",arr);

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

console.log("Merkle Root \t \t ",tree.root());
console.log("Tree Depth \t \t ",tree.depth());
console.log("Tree Levels \t \t ",tree.levels());
console.log("Tree Nodes \t \t ",tree.nodes());

var i;
for(i =0; i< tree.levels();i++)
{
   console.log("Level "+i+": \t\t", tree.level(i));
}
```

Output:

```
PS D:\SyMCA Sem3\BlockChain\Prac_1> node "d:\SyMCA Sem3\BlockChain\Prac_1\MerkleTree.js"
Input            [ 'Fred', ' Bret', ' Bill', ' Bob', ' Alice', ' Trent' ]
Merkle Root              CA2DA9FB28EED6DDBF324396FAC88F12D8013986
Tree Depth               3
Tree Levels              4
Tree Nodes               6
Level 0:                [ 'CA2DA9FB28EED6DDBF324396FAC88F12D8013986' ]
Level 1:                [
  '2F454F840BFD7E43E53A398A483C1EE3E8322F37',
  'F245EC856F1ADA0E74AA17A65583D870C25BBD3A'
]
Level 2:                [
  'B8B295ACA2A9CA869C466E25202DE24660249817',
  '0E48BE1A0598E25D6646CDC47C5B3F33C39122B7',
  'F245EC856F1ADA0E74AA17A65583D870C25BBD3A'
]
Level 3:                [
  '48FDE3D64619929F3AB6F64953B06E1D041BF901',
  '48FDE3D64619929F3AB6F64953B06E1D041BF901',
  '48FDE3D64619929F3AB6F64953B06E1D041BF901',
  '2F5B255CBED913AC3612B92178F14C2A601442EA',
  'EE687A2FF9ADA7BA32B182A12D31BE495EA2F9CC',
  '4B9AFD16AF7665CDF13CEEA8DB4E41A81679256D',
  'ABC156FFF43072D35703F3F412C4BF3B25C70AD9',
  '2C2439449D7A51DDD22D02C25D89FF1078160BA9'
]
```

5. Write a program to implement Merkle Tree using python

Program:

```
import hashlib

class MerkleTree:
    """Represents a Merkle tree."""
    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):
        """Recursively builds the Merkle tree."""
        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]  # Handle odd number of leaves
by duplicating
            next_level.append(self._hash(combined))

        return self._build_tree(next_level)

    def _hash(self, data):
        """Hashes the input data using SHA-256."""
        return hashlib.sha256(data.encode()).hexdigest()

    def get_root(self):
        """Returns the root of the Merkle tree."""
        return self.root

    def get_proof(self, index):
        """Returns the proof for the leaf at the specified 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:  # Odd index
                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):
        """Returns the next level of the Merkle tree."""
        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):
        """Verifies the proof for the 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

# Example usage:
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 D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\MerkleTree.py"
Merkle root: cd52f7d81c41d96eef4d485cb1601c2a8200348f13adcc3a8e748cc2d39de331
Proof for leaf1: [('right', '5038da95330ba16edb486954197e37eb777c3047327ca54df4199c35c5edc17a'), ('right', '8b673b0ce5dbfb9560
9ece827dfaf1fe0767c9c371c16430078f412271cd6c8a')]
Is proof valid? True
```

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

Program:

```
import random

def gcd(a, b):
    """Compute the greatest common divisor of a and b"""
    while b:
        a, b = b, a % b
    return a

def multiplicative_inverse(e, phi):
    """Compute the multiplicative inverse of e modulo 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):
    """Generate a public and private key pair"""
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)

    # Compute d such that d*e = 1 (mod phi)
    d = multiplicative_inverse(e, phi)

    # Return public and private key pairs
    return ((e, n), (d, n))

def encrypt(pk, plaintext):
    """Encrypt the plaintext using the public key"""
    e, n = pk
    ciphertext = [pow(ord(char), e, n) for char in plaintext]
    return ciphertext

def decrypt(pk, ciphertext):
    """Decrypt the ciphertext using the private key"""
    d, n = pk
    plaintext = [chr(pow(char, d, n)) for char in ciphertext]
    return ''.join(plaintext)

# Example usage:
p = 61
q = 53
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 D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\RsaAlgorithm.py"
Public Key: (2681, 3233)
Private Key: (2921, 3233)
Encrypted Message: [1780, 337, 257, 257, 3161, 543, 2053, 1771, 3161, 2615, 257, 1056, 1375]
Decrypted Message: Hello, World!
```

7. Implement Binary Tree using python

Program:

```
class Node:
    """Represents a node in the binary tree."""
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    """Represents the binary tree itself."""
    def __init__(self):
        self.root = None

    def insert(self, value):
        """Inserts a new value into the binary tree."""
        if self.root is None:
            self.root = Node(value)
```

```
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, current_node, value):
        """Recursively inserts a new value into the binary tree."""
        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):
        """Performs an inorder traversal of the binary tree and returns the values in ascending order."""
        result = []
        self._inorder_traversal_recursive(self.root, result)
        return result

    def _inorder_traversal_recursive(self, current_node, result):
        """Recursively performs an inorder traversal of the binary tree."""
        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):
        """Performs a preorder traversal of the binary tree and returns the values."""
        result = []
```

```
        self._preorder_traversal_recursive(self.root, result)
        return result

    def _preorder_traversal_recursive(self, current_node, result):
        """Recursively performs a preorder traversal of the binary tree."""
        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):
        """Performs a postorder traversal of the binary tree and returns the
values."""
        result = []
        self._postorder_traversal_recursive(self.root, result)
        return result

    def _postorder_traversal_recursive(self, current_node, result):
        """Recursively performs a postorder traversal of the binary tree."""
        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):
        """Deletes a value from the binary tree."""
        self.root = self._delete_recursive(self.root, value)

    def _delete_recursive(self, current_node, value):
        """Recursively deletes a value from the binary tree."""
        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):
        """Finds the node with the minimum value in the binary tree."""
        while current_node.left is not None:
            current_node = current_node.left
        return current_node

# Example usage:
tree = BinaryTree()
tree.insert(5)
tree.insert(3)
tree.insert(7)
tree.insert(2)
tree.insert(4)
tree.insert(6)
tree.insert(8)

print("Inorder traversal:", tree.inorder_traversal())
```

```
print("Preorder traversal:", tree.preorder_traversal())
print("Postorder traversal:", tree.postorder_traversal())

tree.delete(4)
print("Inorder traversal after deletion:", tree.inorder_traversal())
```

Output:

```
PS D:\SyMCA Sem3\BlockChain\Prac_1> python -u "d:\SyMCA Sem3\BlockChain\Prac_1\binaryTree.py"
Inorder traversal: [2, 3, 4, 5, 6, 7, 8]
Preorder traversal: [5, 3, 2, 4, 7, 6, 8]
Postorder traversal: [2, 4, 3, 6, 8, 7, 5]
Inorder traversal after deletion: [2, 3, 5, 6, 7, 8]
```