**Practical No.1: Implementing Substitution and Transposition Ciphers**

**Aim: To study and implement the Substitution and Transposition Ciphers**

Code: Python code for implementing Caesar Cipher

def encrypt(msg, key):

enc\_msg = ""

for ch in msg:

if ch.isupper():

new\_ch = chr((ord(ch) - ord("A") + key) % 26 + ord("A"))

enc\_msg += new\_ch

elif ch.islower():

new\_ch = chr((ord(ch) - ord("a") + key) % 26 + ord("a"))

enc\_msg += new\_ch

else:

enc\_msg += ch # Preserve non-alphabet characters

return enc\_msg

def decrypt(msg, key):

dec\_msg = ""

for ch in msg:

if ch.isupper():

new\_ch = chr((ord(ch) - ord("A") - key) % 26 + ord("A"))

dec\_msg += new\_ch

elif ch.islower():

new\_ch = chr((ord(ch) - ord("a") - key) % 26 + ord("a"))

dec\_msg += new\_ch

else:

dec\_msg += ch # Preserve non-alphabet characters

return dec\_msg

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

key = int(input("Enter the key for encryption: "))

enc = encrypt(text, key)

dec = decrypt(enc, key)

print(f"\nThe encrypted text is: {enc}\nThe decryption of encrypted text is: {dec}")

**Code: Python code for implementing Railfence Cipher**

**Railfence\_cipher.py**

string = input("Enter a string: ")

def rail\_fence(txt):

even\_chars = ""

odd\_chars = ""

# Separate characters into even and odd indexed strings

for i in range(len(txt)):

if i % 2 == 0:

even\_chars += txt[i]

else:

odd\_chars += txt[i]

# Combine the even and odd indexed characters

return even\_chars + odd\_chars

print(rail\_fence(string))

**Practical No.2: RSA Encryption and Decryption**

**Aim: To study and implement the RSA Encryption and Decryption**

**Code:**

import math

class RSA:

def \_\_init\_\_(self, prime1, prime2) -> None:

self.p = prime1

self.q = prime2

self.N = self.p \* self.q

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

self.e = self.getE()

self.d = self.getD()

def isPrime(self, n):

if n <= 1:

return False

for i in range(2, int(math.sqrt(n)) + 1):

if n % i == 0:

return False

return True

def getE(self):

for num in range(2, self.phi):

if math.gcd(num, self.phi) == 1: # e should be coprime with phi

return num

return None # Fallback if no suitable e is found

def getD(self):

d = 1

while (d \* self.e) % self.phi != 1:

d += 1

return d

def encrypt(self, msg):

enc\_msg = (msg \*\* self.e) % self.N

return enc\_msg

def decrypt(self, msg):

dec\_msg = (msg \*\* self.d) % self.N

return dec\_msg

# Example usage

p, q = 7, 17

r = RSA(p, q)

print(f"The value of p is: {r.p}\nThe value of q is: {r.q}")

print(f"The value of N is: {r.N}\nThe value of phi is: {r.phi}")

print(f"The value of E is: {r.e}\nThe value of D is: {r.d}")

print(f"The encryption of PT is: {r.encrypt(10)}")

print(f"The decryption of CT is: {r.decrypt(r.encrypt(10))}") # Correctly decrypting the encrypted message

**Practical No.3: Message Authentication Codes (MAC)**

**Aim: To study and implement the Message Authentication Code for ensuring the message integrity and**

**Authenticity**

**Code:**

import hashlib

def MAC(msg):

result = hashlib.sha1(msg.encode())

return result.hexdigest()

def encrypt(msg, key):

enc\_msg = ""

for ch in msg:

if ch.isupper():

new\_ch = chr((ord(ch) - ord("A") + key) % 26 + ord("A"))

enc\_msg += new\_ch

elif ch.islower():

new\_ch = chr((ord(ch) - ord("a") + key) % 26 + ord("a"))

enc\_msg += new\_ch

else:

enc\_msg += ch # Preserve non-alphabetical characters

mac = MAC(enc\_msg)

return [enc\_msg, mac]

def decrypt(msg, key, mac):

dmac = MAC(msg)

if dmac == mac:

print("Message is not changed")

else:

print("Message is changed")

dec\_msg = ""

for ch in msg:

if ch.isupper():

new\_ch = chr((ord(ch) - ord("A") - key) % 26 + ord("A"))

dec\_msg += new\_ch

elif ch.islower():

new\_ch = chr((ord(ch) - ord("a") - key) % 26 + ord("a"))

dec\_msg += new\_ch

else:

dec\_msg += ch # Preserve non-alphabetical characters

return dec\_msg

# Example usage

msg = input("Enter the message: ")

enc = encrypt(msg, 2)

print(f"Encrypted Message: {enc[0]}")

print(f"MAC: {enc[1]}")

# Simulating message not modified

dec = decrypt(enc[0], 2, enc[1])

print(f"Decrypted Message: {dec}")

# Simulating message modified

modified\_msg = "hello"

enc[0] = modified\_msg

dec = decrypt(enc[0], 2, enc[1])

**Practical No.4: Digital Signatures**

**Aim: To study and implement the Digital Signature algorithm**

**Code:**

import math

import hashlib

class RSA:

def \_\_init\_\_(self, p, q):

self.p = p

self.q = q

self.N = p \* q

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

self.e = self.getE()

self.d = self.getD()

def isPrime(self, n):

if n <= 1:

return False

if n <= 3:

return True

if n % 2 == 0 or n % 3 == 0:

return False

i = 5

while i \* i <= n:

if n % i == 0 or n % (i + 2) == 0:

return False

i += 6

return True

def getE(self):

for num in range(2, self.phi):

if self.isPrime(num) and self.phi % num != 0:

return num

return -1

def getD(self):

d = 1

while (d \* self.e) % self.phi != 1:

d += 1

return d

def encrypt(self, msg):

letters = ""

for i in msg:

if not i.isdigit():

posi = ord(i) - ord("a")

ct = pow(posi, self.e, self.N) % 26

letters += chr(ct + 97)

else:

letters += i

return letters

def decrypt(self, msg):

letters = ""

for i in msg:

if not i.isdigit():

posi = ord(i) - ord("a")

pt = pow(posi, self.d, self.N) % 26

letters += chr(pt + 97)

else:

letters += i

return letters

def hashcode(msg, key):

msgKey = msg + key

hashed = hashlib.sha256(msgKey.encode("UTF-8")).hexdigest()

return hashed

# Main execution

p = int(input("Enter Prime number 1: "))

q = int(input("Enter Prime number 2: "))

rsa = RSA(p, q)

if rsa.isPrime(p) and rsa.isPrime(q):

print(f"Value of P: {rsa.p}, Value of Q: {rsa.q}\nValue of E: {rsa.e}, Value of D: {rsa.d}")

msg = input("Enter the message: ")

key = input("Enter key: ")

choice = int(input("1) Encryption\n2) Decryption\nYour Choice: "))

if choice == 1:

hash\_value = hashcode(msg, key)

digi\_sign = rsa.encrypt(hash\_value)

print(f"Digital signature: {digi\_sign}\nHash value of message: {hash\_value}")

elif choice == 2:

digi\_sign = input("Enter the digital signature: ")

hash\_value = rsa.decrypt(digi\_sign)

print(f"Decrypted Hash value: {hash\_value}")

hashValueOfMsg = hashcode(msg, key)

if hash\_value == hashValueOfMsg:

print("Connection is safe")

else:

print("Connection is not safe")

else:

print("Numbers are not prime")

**Practical No.5: Key Exchange using Diffie-Hellman**

**Aim: To study and implement the Diffie-Hellman key exchange algorithm for secure exchange of keys**

**between two entities**

**Code:**

from random import randint

if \_\_name\_\_ == '\_\_main\_\_':

P = 23

G = 9

print('The Value of P is :%d'%(P))

print('The Value of G is :%d'%(G))

a = 4

print('Secret Number for Alice is :%d'%(a))

x = int(pow(G,a,P))

b = 6

print('Secret Number for Bob is :%d'%(b))

y = int(pow(G,b,P))

ka = int(pow(y,a,P))

kb = int(pow(x,b,P))

print('Secret key for the Alice is : %d'%(ka))

print('Secret Key for the Bob is : %d'%(kb))