**Python Implementation:**

**RSA Encryption and Decryption**

**REPORT**

**Mani Bharathi**

**CS22B1026**

**Implementation**

The RSA encryption algorithm relies on concepts from modular arithmetic and number theory. Its implementation involves the following major steps:

* Randomly selecting two distinct prime numbers, p and q
* Calculating n = p × q and Euler’s totient function ϕ(n) = (p–1)(q–1)
* Choosing a public exponent e such that gcd(e, ϕ(n)) = 1
* Determining the private exponent d where e × d ≡ 1 (mod ϕ(n))

Encryption and decryption are performed using the standard RSA formulas:

* Ciphertext is generated as M^e mod n
* Plaintext is recovered as C^d mod n

Key functionalities implemented in the code include:

* Computing GCD and using the Extended Euclidean Algorithm to find modular inverses.
* Performing modular exponentiation efficiently.
* Encrypting and decrypting messages character by character.

**Code**

import random

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

def xgcd(a, b):

x, old\_x = 0, 1

y, old\_y = 1, 0

while b != 0:

quotient = a // b

a, b = b, a - quotient \* b

old\_x, x = x, old\_x - quotient \* x

old\_y, y = y, old\_y - quotient \* y

return a, old\_x, old\_y

def mod\_pow(base, exponent, mod):

result = 1

base %= mod

while exponent > 0:

if exponent % 2 == 1:

result = (result \* base) % mod

exponent //= 2

base = (base \* base) % mod

return result

def choose\_keys():

with open('primes-to-100k.txt', 'r') as f:

primes = list(map(int, f.read().splitlines()))

filtered\_primes = [p for p in primes if p >= 17]

while True:

p = random.choice(filtered\_primes)

q = random.choice(filtered\_primes)

if p != q:

n = p \* q

if n >= 256:

break

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

while True:

e = random.randrange(2, phi)

if gcd(e, phi) == 1:

break

\_, d, \_ = xgcd(e, phi)

if d < 0:

d += phi

return p, q, n, phi, e, d

def encrypt\_char(char, e, n):

return mod\_pow(ord(char), e, n)

def decrypt\_char(cipher, d, n):

return chr(mod\_pow(cipher, d, n))

def main():

print("Generating RSA keys...")

p, q, n, phi, e, d = choose\_keys()

print("\nRSA Key Details:")

print(f"p = {p}")

print(f"q = {q}")

print(f"n = {n}")

print(f"phi = {phi}")

print(f"e = {e}")

print(f"d = {d}\n")

message = input("Enter the message to encrypt: ").strip()

if not message:

print("No message entered. Exiting.")

return

encrypted = [encrypt\_char(c, e, n) for c in message]

print("\nEncrypted message (list of integers):")

print(encrypted)

decrypt\_now = input("\nDo you want to decrypt the message? (y/n): ").lower()

if decrypt\_now == 'y':

decrypted = ''.join([decrypt\_char(val, d, n) for val in encrypted])

print("\nDecrypted message:")

print(decrypted)

else:

print("Decryption skipped.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Results**

The implementation was tested with example plaintexts. The ciphertext obtained through encryption and the restored plaintext after decryption matched correctly, validating the accuracy of the RSA system.

