Academic Year: 2025-26	**Programme: BTECH-Cyber (CSE)**
Year: 2nd	**Semester: IV**
Student Name: Mishitha	**Batch: K2**
Roll No: K073	**Date of experiment: 17/2/25**
Faculty: Rejo Mathew	**Signature with Date:**

Experiment 6: RSA algorithm

Aim: Write a program to implement RSA algorithm.

Learning Outcomes:
After completion of this experiment, student should be able to
1. Differentiate between symmetric and asymmetric key cryptography.
2. Describe working of RSA algorithm.
3. Understand application of RSA along with its advantage and limitations.

Theory:

Algorithm for RSA is given below.
- Choose two large prime numbers p, q
- Let $n = p * q$; then $\varphi(n) = (p-1) * (q-1)$ [where φ is Euler's totient function]
- Choose $e < \varphi(n)$ such that e is relatively prime to $\varphi(n)$.

Choose an integer e such that $1 < e < \varphi(pq)$, and e and $\varphi(pq)$ share no divisors other than 1 (i.e., e and $\varphi(pq)$ are coprime.).

- e is released as the public key exponent.

Determine $d = e^{-1} \bmod (\varphi)$

- d is kept as the private key exponent.

- Compute d such that $e\, d \bmod \varphi(n) = 1$
- Public key: (e, n); private key: (d, n)
- Encipher: $c = m^e \bmod n$
- Decipher: $m = c^d \bmod n$

Example:
- Take $p = 7$, $q = 11$, so $n = 77$ and $\varphi(n) = 60$
- Alice chooses $e = 17$, making $d = 53$

- Bob wants to send Alice secret message HELLO (07 04 11 11 14)
- $07^{17} \bmod 77 = 28$
- $04^{17} \bmod 77 = 16$

- 11^{17} mod 77 = 44
- 11^{17} mod 77 = 44
- 14^{17} mod 77 = 42
- Bob sends 28 16 44 44 42
- Alice receives 28 16 44 44 42

Alice uses private key, *d* = 53, to decrypt message:

- 28^{53} mod 77 = 07
- 16^{53} mod 77 = 04
- 44^{53} mod 77 = 11
- 44^{53} mod 77 = 11
- 42^{53} mod 77 = 14
- Alice translates message to letters to read HELLO

Algorithm:

1. Accept two integer numbers from user.
2. Validate the input provided by user is a prime number. If not, ask user to re-enter prime number.
3. Generate public key and private key.
4. Display public key and private key.
5. Ask user to input message for encryption.
6. Display the cipher text.
7. Ask user to input cipher text for decryption.
8. Display the plain text.

Code: ***type or copy your completed working code here***

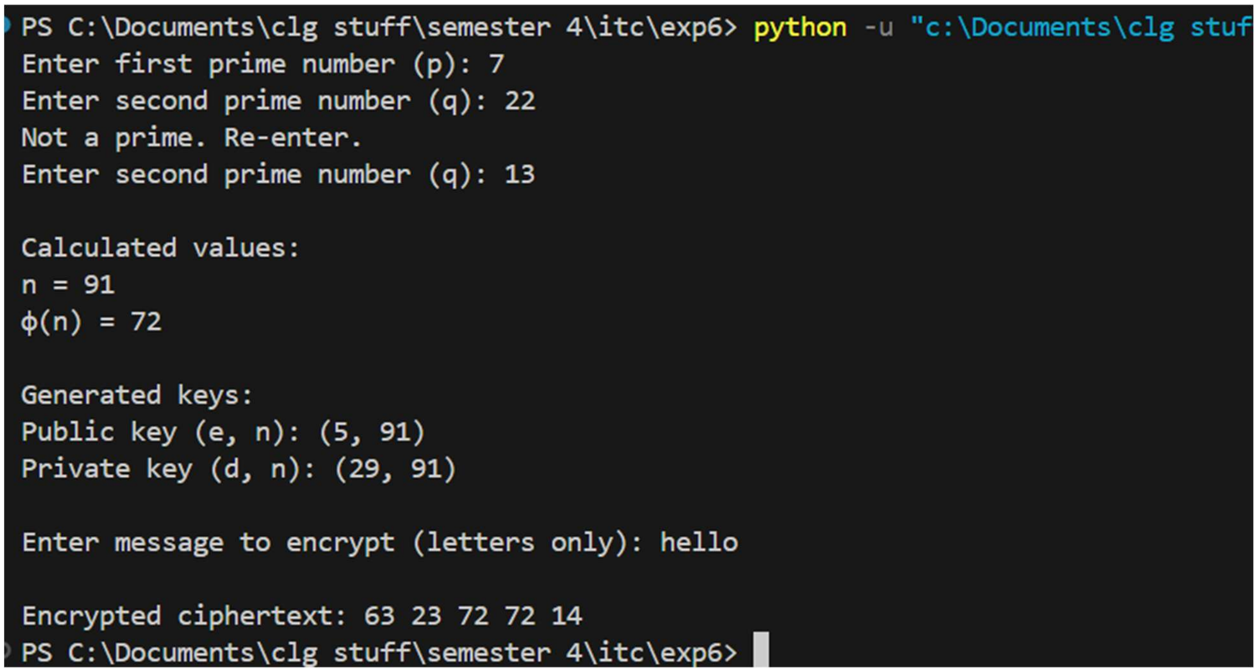
Note: Code should have proper comments

- **Encryption**
  Code

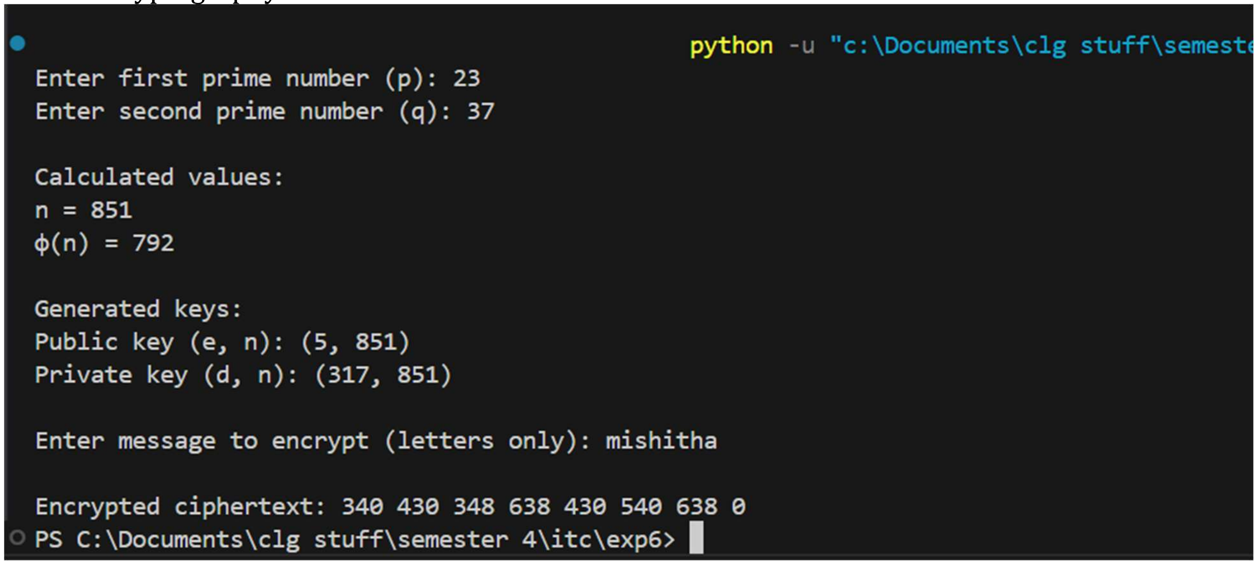
```
# Step 1: Accept two prime numbers from the user
p = int(input("Enter first prime number (p): "))
while True:
    if p < 2:
        is_prime = False
    else:
        is_prime = True
        # Check divisibility up to sqrt(p)
        for i in range(2, int(p**0.5) + 1):
            if p % i == 0:
                is_prime = False
                break
    if is_prime:
        break
    else:
        print("Not a prime. Re-enter.")
        p = int(input("Enter first prime number (p): "))

q = int(input("Enter second prime number (q): "))
while True:
    if q < 2:
        is_prime = False
    else:
        is_prime = True
        # Check divisibility up to sqrt(q)
        for i in range(2, int(q**0.5) + 1):
            if q % i == 0:
                is_prime = False
                break
    if is_prime:
        break
    else:
        print("Not a prime. Re-enter.")
        q = int(input("Enter second prime number (q): "))

# Step 2: Compute n and ϕ(n)
n = p * q
phi = (p - 1) * (q - 1)
print("\nCalculated values:")
print(f"n = {n}")
print(f"ϕ(n) = {phi}")

# Step 3: Find e starting from 3, checking coprimality
e = 3
found = False
while e < phi:
    # Check gcd(e, p-1) == 1
    a, b = e, (p-1)
    while b != 0:
        a, b = b, a % b
    gcd_e_p1 = a

    # Check gcd(e, q-1) == 1
    a, b = e, (q-1)
    while b != 0:
        a, b = b, a % b
    gcd_e_q1 = a

    # Check gcd(e, phi) == 1
    a, b = e, phi
    while b != 0:
        a, b = b, a % b
    gcd_e_phi = a

    if gcd_e_p1 == 1 and gcd_e_q1 == 1 and gcd_e_phi == 1:
        found = True
        break
    e += 2  # Increment to next odd number

if not found:
    print("\nError: No suitable 'e' found. Try different primes.")
    exit()

# Step 4: Compute d (modular inverse of e mod phi)
# Using Extended Euclidean Algorithm
a, b = e, phi
old_r, r = a, b
old_s, s = 1, 0
old_t, t = 0, 1

while r != 0:
    quotient = old_r // r
    old_r, r = r, old_r - quotient * r
    old_s, s = s, old_s - quotient * s
    old_t, t = t, old_t - quotient * t

if old_r != 1:
    print("\nError: Modular inverse does not exist. Choose different primes.")
    exit()
else:
    d = old_s % phi  # Ensure d is positive

print("\nGenerated keys:")
print(f"Public key (e, n): ({e}, {n})")
print(f"Private key (d, n): ({d}, {n})")

# Step 5: Encrypt message
message = input("\nEnter message to encrypt (letters only): ")
message = message.upper()
```

```
cipher = []
for char in message:
    if not char.isalpha():
        continue
    # Convert character to 0-based index (A=0, B=1, ..., Z=25)
    m = ord(char) - ord('A')

    # Compute c = m^e mod n using manual modular exponentiation
    c = 1
    base = m % n
    exp = e
    while exp > 0:
        if exp % 2 == 1:
            c = (c * base) % n
        exp = exp // 2
        base = (base * base) % n
    cipher.append(str(c))

print("\nEncrypted ciphertext:", " ".join(cipher))
```

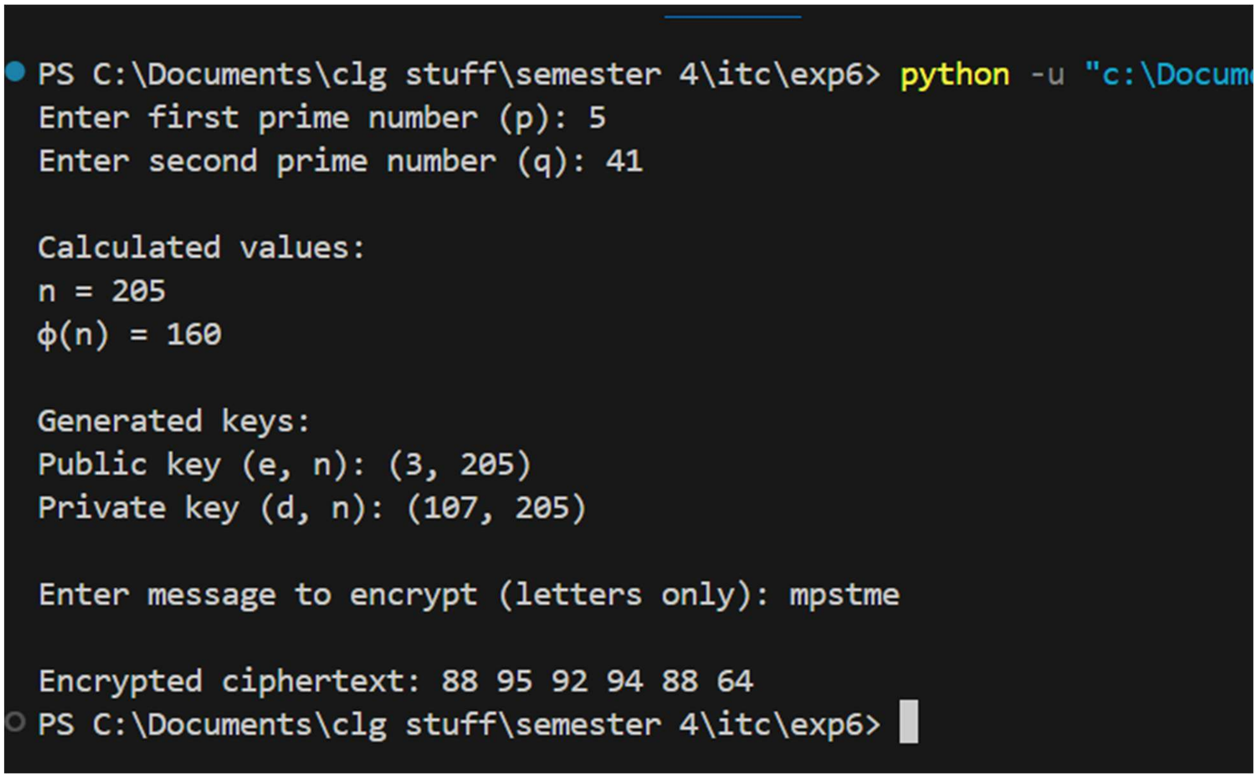
Output:



```
PS C:\Documents\clg stuff\semester 4\itc\exp6> python -u "c:\Documents\clg stuf
Enter first prime number (p): 7
Enter second prime number (q): 22
Not a prime. Re-enter.
Enter second prime number (q): 13

Calculated values:
n = 91
ϕ(n) = 72

Generated keys:
Public key (e, n): (5, 91)
Private key (d, n): (29, 91)

Enter message to encrypt (letters only): hello

Encrypted ciphertext: 63 23 72 72 14
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```



```
                                          python -u "c:\Documents\clg stuff\semeste
Enter first prime number (p): 23
Enter second prime number (q): 37

Calculated values:
n = 851
ϕ(n) = 792

Generated keys:
Public key (e, n): (5, 851)
Private key (d, n): (317, 851)

Enter message to encrypt (letters only): mishitha

Encrypted ciphertext: 340 430 348 638 430 540 638 0
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```



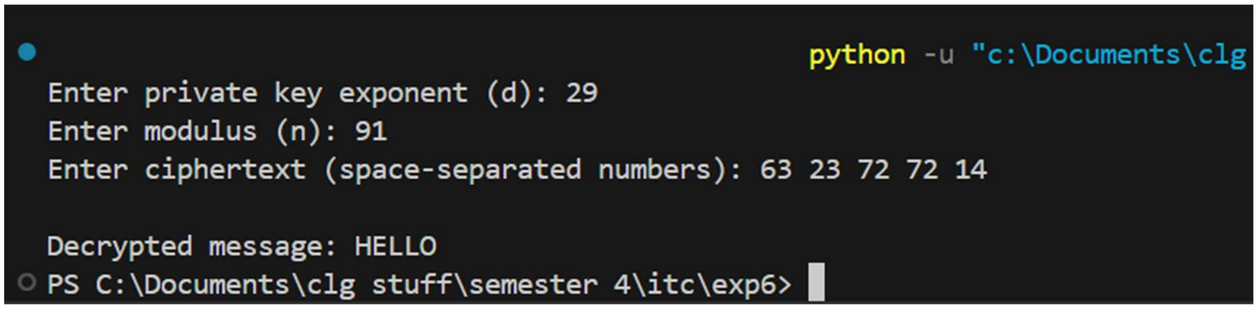
```
PS C:\Documents\clg stuff\semester 4\itc\exp6> python -u "c:\Docume
Enter first prime number (p): 5
Enter second prime number (q): 41

Calculated values:
n = 205
ϕ(n) = 160

Generated keys:
Public key (e, n): (3, 205)
Private key (d, n): (107, 205)

Enter message to encrypt (letters only): mpstme

Encrypted ciphertext: 88 95 92 94 88 64
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```

- **Decryption**
  Code

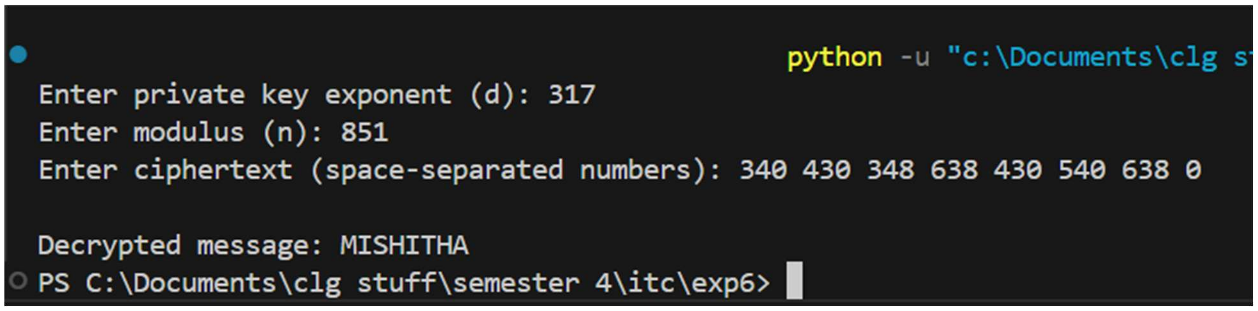
```python
# Step 1: Accept private key (d, n)
d = int(input("Enter private key exponent (d): "))
n = int(input("Enter modulus (n): "))
```

```python
# Step 2: Accept ciphertext
cipher_input = input("Enter ciphertext (space-separated numbers): ")
cipher_list = list(map(int, cipher_input.split()))

# Step 3: Decrypt each number
decrypted = []
for c in cipher_list:
    # Compute m = c^d mod n using manual modular exponentiation
    m = 1
    base = c % n
    exp = d
    while exp > 0:
        if exp % 2 == 1:
            m = (m * base) % n
        exp = exp // 2
        base = (base * base) % n
    # Convert m to character (A=0, B=1, ..., Z=25)
    decrypted_char = chr(m + ord('A'))
    decrypted.append(decrypted_char)

print("\nDecrypted message:", "".join(decrypted))
```

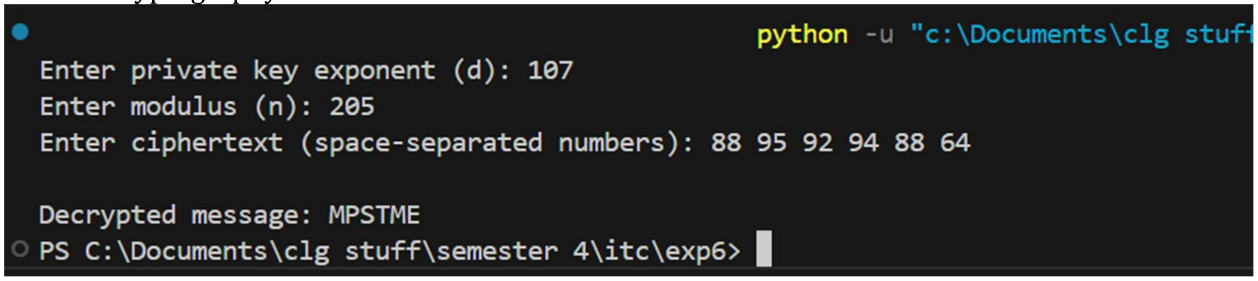
Output



```
                                             python -u "c:\Documents\clg
Enter private key exponent (d): 29
Enter modulus (n): 91
Enter ciphertext (space-separated numbers): 63 23 72 72 14

Decrypted message: HELLO
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```



```
                                             python -u "c:\Documents\clg s
Enter private key exponent (d): 317
Enter modulus (n): 851
Enter ciphertext (space-separated numbers): 340 430 348 638 430 540 638 0

Decrypted message: MISHITHA
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```



```
                                                python -u "c:\Documents\clg stuf
 Enter private key exponent (d): 107
 Enter modulus (n): 205
 Enter ciphertext (space-separated numbers): 88 95 92 94 88 64

 Decrypted message: MPSTME
PS C:\Documents\clg stuff\semester 4\itc\exp6>
```

Questions:

1. Why is it computationally difficult to derive the private key from the public key in RSA?

Ans: RSA encryption is a widely used cryptographic method that relies on the difficulty of factoring large numbers. One of the key reasons why it is computationally difficult to derive the private key from the public key in RSA is because the public key consists of two numbers: **n** and **e**, where **n = p × q**, and **p** and **q** are large prime numbers. To find the private key, we need to calculate **ϕ(n) = (p - 1) × (q - 1)**, which requires knowing **p** and **q**. Since factoring a large number **n** into its prime components is extremely hard with current algorithms, recovering the private key is practically impossible.

2. What happens if two different users select the same prime numbers for their RSA key pairs

Ans: If two different users select the same prime numbers for their RSA key pairs, they will have the same **n** value but may have different public exponents **e**. However, this is a security risk because an attacker can use a mathematical trick called the **Greatest Common Divisor (GCD) method** to compute the private key. This would allow the attacker to decrypt both users' messages and forge their digital signatures, making RSA encryption completely useless for them.

3. How does RSA handle message sizes larger than the modulus n?

Ans: RSA cannot directly handle message sizes larger than **n**, as messages must be smaller than **n** to be encrypted. To overcome this limitation, a technique called **hybrid encryption** is used. First, a **symmetric key** (e.g., an AES key) is generated and encrypted using RSA. Then, the actual message is encrypted with the symmetric key using AES, which is much faster than RSA. Finally, the encrypted symmetric key and the encrypted message are sent together. The receiver first **decrypts the symmetric key with RSA** and then **decrypts the message with AES**. This method ensures that RSA can be used efficiently for large messages.

4. How does the Chinese Remainder Theorem (CRT) optimize RSA decryption?

Ans: The **Chinese Remainder Theorem (CRT)** helps optimize RSA decryption by making it faster. Normally, RSA decryption requires dealing with very large numbers, which is slow. CRT speeds up the process by performing calculations separately for **p** and **q** instead of the full **n**. This means two smaller calculations are done first, and then the results are combined using a special formula. Since the calculations involve smaller numbers, decryption becomes almost **four times faster**, making RSA more efficient in real-world applications.

5. Why does RSA encryption work as a one-way function, and how does the difficulty of integer factorization contribute to its security?

Ans: RSA works as a **one-way function**, meaning it is easy to compute in one direction but very hard to reverse. Encryption is straightforward, as it follows the formula **c = m^e mod**

n. However, decryption requires finding **m = c^d mod n**, which involves computing **d** from **ϕ(n)**. Since **ϕ(n)** depends on **p** and **q**, and finding **p** and **q** requires factoring **n**, it becomes extremely difficult. This is why RSA remains secure—because **factoring a large number is computationally infeasible** with current technology.

Conclusion:

In this lab, I learned how **RSA encryption and decryption work** and why they are secure. The process of encryption is easy using a public key, but decryption requires a private key, which is extremely hard to find without knowing the prime numbers **p** and **q**. If two users accidentally use the same prime numbers, their keys become vulnerable, making their data unsafe. Since RSA cannot handle large messages directly, **hybrid encryption** (a combination of RSA and AES) is used for efficiency. Additionally, the **Chinese Remainder Theorem (CRT)** helps speed up RSA decryption by breaking the computation into smaller parts. Overall, RSA is a secure encryption method because **factoring large numbers is very difficult**, ensuring that encrypted data stays safe from attackers.