# PRACTICAL: 5

## AIM:

RSA algorithm is a public key encryption technology. It is considered the most secure way of encryption to secure sensitive data, particularly when sent over an insecure network such as the Internet. The public and private key generation algorithm is the most complex part of the RSA algorithm. The strength of RSA is the difficulty of factoring large integers that are the product of two large prime numbers, which is considered infeasible due to the time it would take using even today's highly configured computers. Implement the RSA algorithm.

## THEORY:

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem, one of the oldest widely used for secure data transmission. The initialism "RSA" comes from the surnames of Ron Rivest, Adi Shamir and Leonard Adleman, who publicly described the algorithm in 1977. An equivalent system was developed secretly in 1973 at Government Communications Headquarters (GCHQ), the British signals intelligence agency, by the English mathematician Clifford Cocks. That system was declassified in 1997

In a public-key cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private). An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decrypted by someone who knows the private key

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used

RSA is a relatively slow algorithm. Because of this, it is not commonly used to directly encrypt user data. More often, RSA is used to transmit shared keys for symmetric-key cryptography, which are then used for bulk encryption–decryption.

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

A basic principle behind RSA is the observation that it is practical to find three very large positive integers e, d, and n, such that for all integers m (0<= m< n), both $(m^e)^d$ and m have the same remainder when divided by n (they are congruent modulo n):

$$(m^e)^d \equiv m \pmod{n}.$$

**Key generation**

The keys for the RSA algorithm are generated in the following way:

1. Choose two large prime numbers *p* and *q*.

- To make factoring harder, *p* and *q* should be chosen at random, be both large and have a large difference. For choosing them the standard method is to choose random integers and use a primality test until two primes are found.
- *p* and *q* are kept secret.

2. Compute n = pq.
   - n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
   - n is released as part of the public key.
3. Compute λ(n), where λ is Carmichael's totient function. Since n = pq, λ(n) = lcm(λ(p), λ(q)), and since p and q are prime, λ(p) = φ(p) = p − 1, and likewise λ(q) = q − 1. Hence λ(n) = lcm(p − 1, q − 1).
   - The lcm may be calculated through the Euclidean algorithm, since lcm(a, b) = |ab|/gcd(a, b).
   - λ(n) is kept secret.
4. Choose an integer e such that 1 < e < λ(n) and gcd(e, λ(n)) = 1; that is, e and λ(n) are coprime.
   - e having a short bit-length and small Hamming weight results in more efficient encryption – the most commonly chosen value for e is 216 + 1 = 65537. The smallest (and fastest) possible value for e is 3, but such a small value for e has been shown to be less secure in some settings.
   - e is released as part of the public key.
5. Determine d as d ≡ e−1 (mod λ(n)); that is, d is the modular multiplicative inverse of e modulo λ(n).
   - This means: solve for d the equation de ≡ 1 (mod λ(n)); d can be computed efficiently by using the extended Euclidean algorithm, since, thanks to e and λ(n) being coprime, said equation is a form of Bézout's identity, where d is one of the coefficients.
   - d is kept secret as the private key exponent.

**Key distribution**

Suppose that Bob wants to send information to Alice. If they decide to use RSA, Bob must know Alice's public key to encrypt the message, and Alice must use her private key to decrypt the message.

To enable Bob to send his encrypted messages, Alice transmits her public key (n, e) to Bob via a reliable, but not necessarily secret, route. Alice's private key (d) is never distributed.

**Encryption**

After Bob obtains Alice's public key, he can send a message M to Alice.

To do it, he first turns M (strictly speaking, the un-padded plaintext) into an integer m (strictly speaking, the padded plaintext), such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c, using Alice's public key e, corresponding to

$$c \equiv m^e \pmod{n}.$$

This can be done reasonably quickly, even for very large numbers, using modular exponentiation. Bob then transmits *c* to Alice. Note that at least nine values of *m* will yield a ciphertext *c* equal to *m*, but this is very unlikely to occur in practice.

**Decryption**

Alice can recover *m* from *c* by using her private key exponent *d* by computing

$$c^d \equiv (m^e)^d \equiv m \pmod{n}.$$

Given *m*, she can recover the original message *M* by reversing the padding scheme.

## CODE:

```
import sympy
import random
import psutil
import time

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def extended_gcd(a, b):
    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

    return old_r, old_s, old_t

def mod_inverse(a, m):
    gcd_value, x, y = extended_gcd(a, m)
    if gcd_value != 1:
        raise ValueError(f"{a} and {m} are not coprime, modular inverse does not exist.")
    else:
        return x % m

def generate_large_prime(decimal_digits):
    return sympy.randprime(10**(decimal_digits-1), 10**decimal_digits)

def generate_public_exponent(phi_n, decimal_digits):
    e = random.randint(10**(decimal_digits-1), 10**decimal_digits - 1)
    while gcd(e, phi_n) != 1:
```

```
        e = random.randint(10**(decimal_digits-1), 10**decimal_digits - 1)
    return e

def generate_keys(decimal_digits):
    cpu_before, ram_before = get_system_utilization()
    p = generate_large_prime(decimal_digits)
    q = generate_large_prime(decimal_digits)
    n = p * q
    phi_n = (p - 1) * (q - 1)
    e = generate_public_exponent(phi_n, decimal_digits)
    cpu_during_gen, ram_during_gen = get_system_utilization()
    d = mod_inverse(e, phi_n)
    cpu_after, ram_after = get_system_utilization()
    print("\nKey Generation Utilization:")
    print(f"CPU Before Key Generation: {cpu_before}% | RAM Before Key Generation:
{ram_before}%")
    print(f"CPU During Key Generation: {cpu_during_gen}% | RAM During Key Generation:
{ram_during_gen}%")
    print(f"CPU After Key Generation: {cpu_after}% | RAM After Key Generation: {ram_after}%")

    return ((n, e), (n, d), p, q)

def encrypt(message, pub_key):
    n, e = pub_key
    cipher_text = [pow(ord(char), e, n) for char in message]
    return cipher_text

def decrypt(cipher_text, priv_key):
    n, d = priv_key
    message = ''.join([chr(pow(char, d, n)) for char in cipher_text])
    return message

def get_system_utilization():
    cpu = psutil.cpu_percent(interval=1)
    ram = psutil.virtual_memory().percent
    return cpu, ram

def rsa_algorithm(message, decimal_digits):
    start_time = time.time()

    pub_key, priv_key, p, q = generate_keys(decimal_digits)

    cpu_before_enc, ram_before_enc = get_system_utilization()

    cipher_text = encrypt(message, pub_key)

    cpu_during_enc, ram_during_enc = get_system_utilization()

    decrypted_message = decrypt(cipher_text, priv_key)
```

```
    cpu_after_enc, ram_after_enc = get_system_utilization()

    end_time = time.time()

    print(f"\nOriginal Message: {message}")
    print(f"Ciphertext: {cipher_text}")
    print(f"Decrypted Message: {decrypted_message}")

    print("\nSystem Utilization During Encryption/Decryption:")
    print(f"CPU Before Encryption: {cpu_before_enc}% | RAM Before Encryption:
{ram_before_enc}%")
    print(f"CPU During Encryption: {cpu_during_enc}% | RAM During Encryption:
{ram_during_enc}%")
    print(f"CPU After Encryption: {cpu_after_enc}% | RAM After Encryption: {ram_after_enc}%")

    print(f"Execution Time: {end_time - start_time:.6f} seconds")
    print(f"Generated Prime p: {p}")
    print(f"Generated Prime q: {q}")
    print(f"Public Exponent e: {pub_key[1]}")
    print(f"Private Exponent d: {priv_key[1]}")

if __name__ == "__main__":
    message = input("Enter the message to encrypt: ")
    decimal_digits = int(input("Enter the number of decimal digits for p, q, and e: "))
    rsa_algorithm(message, decimal_digits)
```

## OUTPUT:

```
PROBLEMS  COMMENTS  PORTS  DEBUG CONSOLE  OUTPUT  TERMINAL

C:\Users\HP\Desktop\SEM-6\CRNS\PR5>python3 Practical-7.py
Enter the message to encrypt: Trushangkumar sumanbhai patel
Enter the number of decimal digits for p, q, and e: 10

Key Generation Utilization:
CPU Before Key Generation: 5.7% | RAM Before Key Generation: 83.5%
CPU During Key Generation: 4.4% | RAM During Key Generation: 83.5%
CPU After Key Generation: 8.7% | RAM After Key Generation: 83.9%

Original Message: Trushangkumar sumanbhai patel
Ciphertext: [4453532882523342408, 6399500532632042452, 1440956188479928763, 604955520960206071, 5159882049969110120, 2069423041551465604, 93542483
3781236276, 2260547122366746860, 8317109596080542601, 1440956188479928763, 3969103403446493610, 2069423041551465604, 6399500532632042452, 29259686
07955983467, 604955520960206071, 1440956188479928763, 3969103403446493610, 2069423041551465604, 9354248337812362 76, 3905712531264677894, 515988204
9969110120, 2069423041551465604, 6577943649377056241, 2925968607955983467, 541775565799116804, 2069423041551465604, 5669203044497453248, 339769707
0105920668, 3925536414680111882]
Decrypted Message: Trushangkumar sumanbhai patel

System Utilization During Encryption/Decryption:
CPU Before Encryption: 5.9% | RAM Before Encryption: 83.3%
CPU During Encryption: 7.8% | RAM During Encryption: 83.3%
CPU After Encryption: 5.4% | RAM After Encryption: 83.2%
Execution Time: 6.045281 seconds
Generated Prime p: 1562305163
Generated Prime q: 6239118947
Public Exponent e: 1452316119
Private Exponent d: 9201074065718868455

C:\Users\HP\Desktop\SEM-6\CRNS\PR5>
```

*Figure 1: 10 Decimal Places for Prime Numbers in RSA: Key Generation, Encryption, and Decryption Performance*

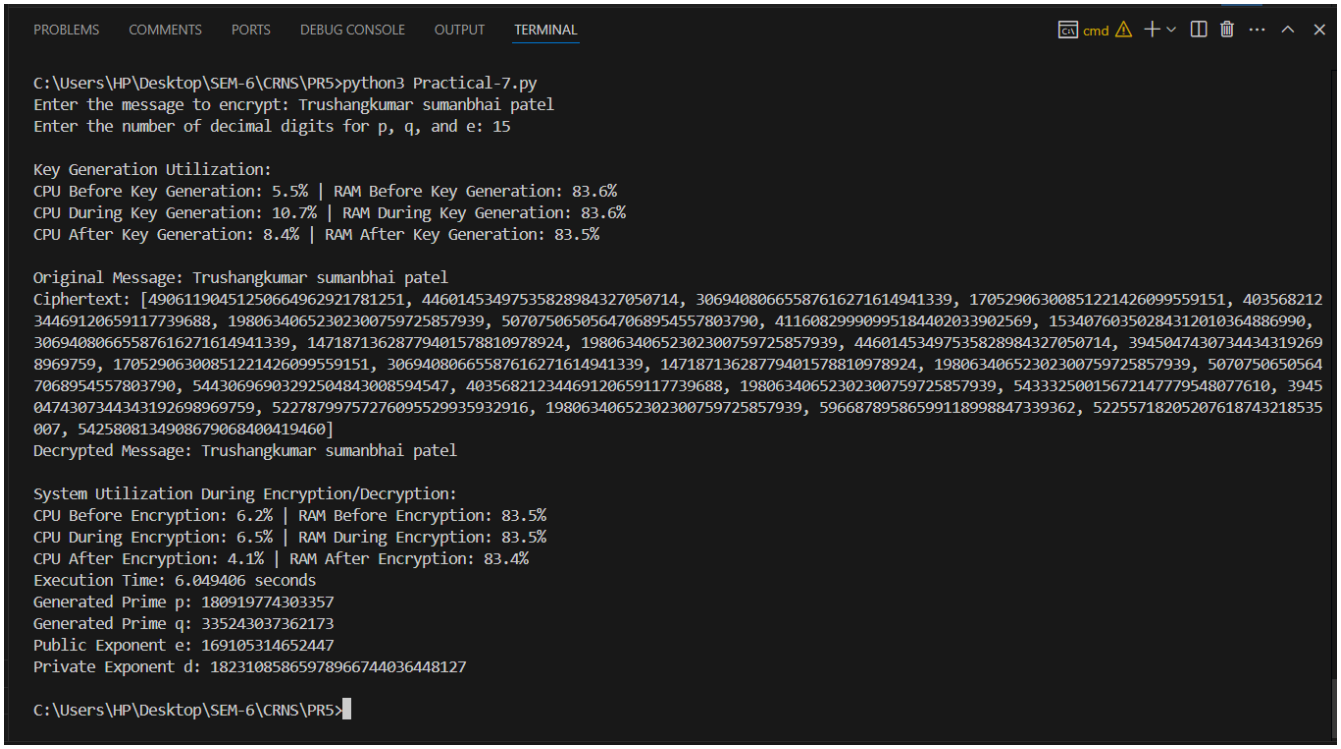


*Figure 2: 15 Decimal Places for Prime Numbers in RSA: Key Generation, Encryption, and Decryption Performance*

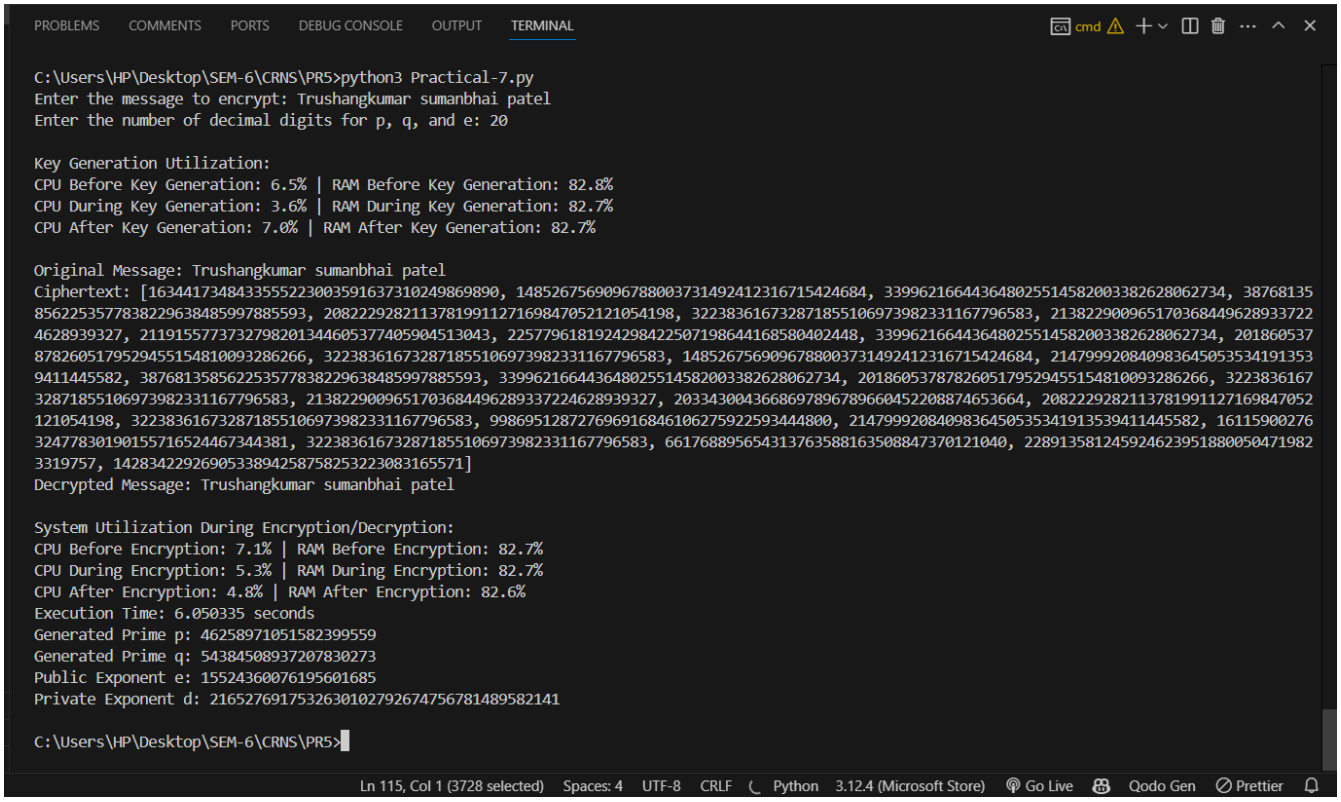


*Figure 3: 20 Decimal Places for Prime Numbers in RSA: Key Generation, Encryption, and Decryption Performance*

| Decimal Places | Stage | CPU Utilization (%) | RAM Utilization (%) |
|---|---|---|---|
| 10 | Before Key Generation | 5.7 | 83.5 |
| | During Key Generation | 4.4 | 83.5 |
| | After Key Generation | 8.7 | 83.9 |
| | Before Encryption | 5.9 | 83.3 |
| | During Encryption | 7.8 | 83.3 |
| | After Encryption | 5.4 | 83.2 |
| 15 | Before Key Generation | 5.5 | 83.6 |
| | During Key Generation | 10.7 | 83.6 |
| | After Key Generation | 8.4 | 83.5 |
| | Before Encryption | 6.2 | 83.5 |
| | During Encryption | 6.5 | 83.5 |
| | After Encryption | 4.1 | 83.4 |
| 20 | Before Key Generation | 6.5 | 82.8 |
| | During Key Generation | 3.6 | 82.7 |
| | After Key Generation | 7 | 82.7 |
| | Before Encryption | 7.1 | 82.7 |
| | During Encryption | 5.3 | 82.7 |
| | After Encryption | 4.8 | 82.6 |

## LATEST APPLICATIONS:

- Secure Web Communications (SSL/TLS)
- Digital Signatures
- Cryptographic Key Management
- Email Encryption (S/MIME)
- Blockchain & Cryptocurrencies
- Authentication Systems

## LEARNING OUTCOME:

In this practical, I tested the RSA algorithm with primes of 10, 15, and 20 decimal places to assess system utilization. CPU and RAM usage remained stable, with slight increases during key generation. The highest resource usage occurred during key generation. Overall, RSA demonstrated efficient performance with minimal impact on system resources, even with larger primes.

## REFERENCES:

1. YouTube: https://www.youtube.com/watch?v=VF3AHG0T9ec
2. Wikipedia: https://en.wikipedia.org/wiki/RSA_(cryptosystem)
3. ChatGPT: https://chatgpt.com/