Walchand College of Engineering, Sangli ***(Government Aided Autonomous Institute)***	
AY 2025-26	
Course Information	
Programme	B.Tech. (Computer Science and Engineering)
Class, Semester	Final Year B. Tech., Sem VII
Course Code	6CS451
Course Name	Cryptography and Network Security Lab
PRN	22510016

Experiment No. 07

Title - Implementation of RSA Algorithm.

Objectives:

1. ***To understand the theoretical foundation of the RSA algorithm***
   Explore the mathematical principles behind RSA, including prime numbers, Euler's theorem, and modular exponentiation.
2. ***To implement the RSA algorithm from scratch using a programming language (e.g., Python, Java, C++)***
   Develop modules for key generation, encryption, and decryption.
3. ***To ensure secure key generation***
   Implement a method to generate large, random prime numbers and compute public and private key pairs.
4. ***To demonstrate the encryption and decryption process***
   Encrypt a plaintext message using the public key and decrypt it using the private key.
5. ***To validate the correctness and performance of the implementation***
   Test the algorithm with different input sizes and ensure accurate decryption of encrypted messages.
6. ***To explore the limitations and possible optimizations of RSA***
   Analyze time complexity, key size limitations, and potential vulnerabilities in naïve implementations.

Problem Statement:

In the current era of digital communication and data exchange, ensuring the confidentiality, integrity, and authenticity of information is critically important. Traditional symmetric encryption methods face challenges in secure key distribution and scalability. To address these issues, public key cryptography provides a robust solution where encryption and decryption are performed using separate keys. The ***RSA algorithm*** is one of the most widely used public key cryptosystems that enables secure data transmission over insecure networks.

However, understanding and implementing RSA from scratch requires a deep understanding of number theory, modular arithmetic, and key generation processes. There is a need for a clear, educational, and functional implementation of the RSA algorithm that demonstrates its core principles and operations, including key generation, encryption, and decryption.

Equipment/Tools:

Python 3.8+ (recommended 3.10+)
Standard library only (no external packages required) — uses `secrets` and `math`.
(Optional) sympy for prime utilities if you prefer convenience: `pip install sympy`.
Text editor / IDE (VS Code, PyCharm) and terminal.

Theory:

RSA is a public-key cryptosystem based on number theory:

Choose two large primes p and q.
Compute n = p * q (the modulus).
Compute Euler's totient phi = (p-1)*(q-1).
Choose public exponent e such that 1 < e < phi and `gcd(e, phi) = 1`.
Compute private exponent d as the modular inverse of e modulo `phi` (`d ≡ e^{-1} (mod phi)`).

Public key: (n, e) — used for encryption. Private key: (n, d) — used for decryption.

Encryption (numeric): c = m^e mod n. Decryption: m = c^d mod n.

Important notes:

Messages must be converted to integers smaller than n before encryption.
Real-world RSA requires padding (PKCS#1 v1.5 or OAEP) for security; raw RSA is vulnerable to many attacks.
Use sufficiently large key sizes (2048 bits or larger for real applications).
Use CRT optimization for faster decryption (optional).

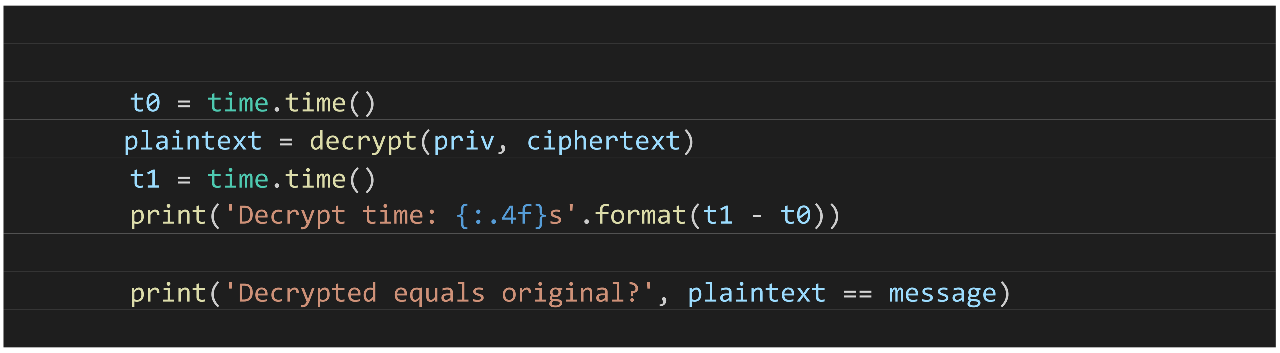
Procedure:

Implement a fast probable-prime test (Miller-Rabin).
Generate large random primes p and q of desired bit length.
Compute n, phi, choose e (commonly 65537), compute d (modular inverse).
Provide functions to convert text ↔ integer and perform encryption/decryption.
Test with short messages and verify correctness.
Measure time for key generation and encryption/decryption for different key sizes.

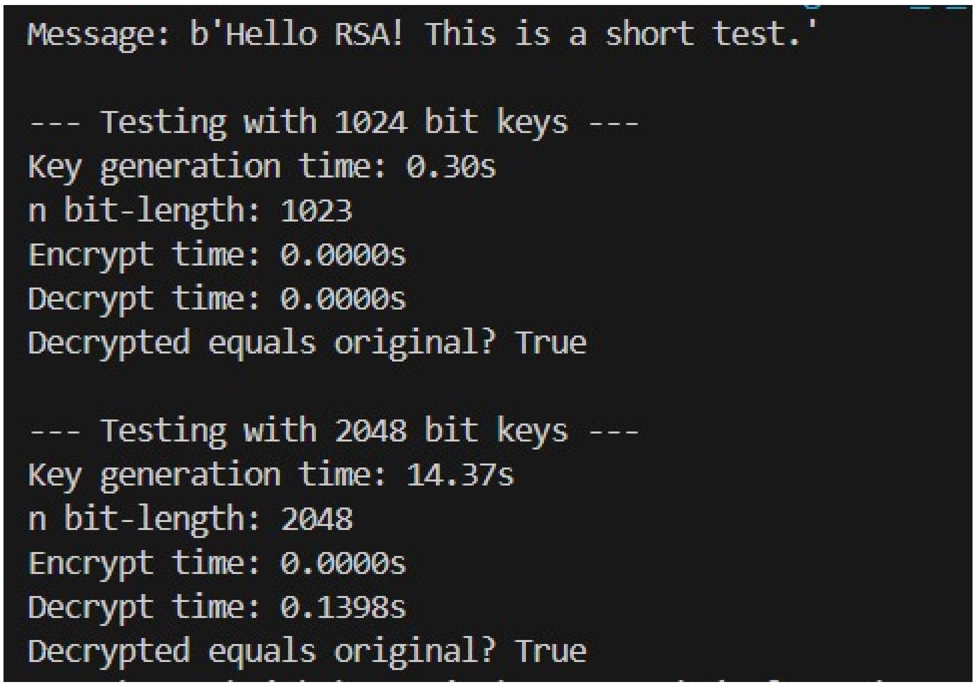
Steps:

```
import math
import secrets
import time

def is_prime(n, k=40):
    if n < 2:
        return False
    if n % 2 == 0:
        return n == 2
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = secrets.randbelow(n - 3) + 2
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

def generate_large_prime(bits):
    while True:
        candidate = secrets.randbits(bits)
        candidate |= (1 << bits - 1)|1
        if is_prime(candidate):
            return candidate

def egcd(a, b):
    if a == 0:
        return b, 0, 1
    g, y, x = egcd(b % a, a)
    return g, x - (b // a) * y, y
```

```python
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modularinversedoesnotexist')
    return x % m

def generate_keypair(bits=1024):
    p = generate_large_prime(bits // 2)
    q = generate_large_prime(bits // 2)
    n = p * q
    phi=(p   -1)*(q-1)
    e = 65537
    if math.gcd(e, phi) != 1:
        e = 3
        while math.gcd(e, phi) != 1:
            e +=2
    d = modinv(e, phi)
    public_key = (n, e)
    private_key = (n, d)
    return public_key, private_key, (p, q)

def encrypt(public_key, plaintext_bytes):
    n, e = public_key
    m=int.from_bytes(plaintext_bytes,byteorder='big')
    if m >= n:
        raise ValueError('message toolargeforthekey size')
    c = pow(m, e, n)
    return c

def decrypt(private_key, ciphertext_int):
    n, d = private_key
    m = pow(ciphertext_int, d, n)
    byte_length = (m.bit_length() + 7) // 8
    return m.to_bytes(byte_length, byteorder='big')

if __name__ == '__main__':
    message = b"Hello RSA! This is a short test."
    print('Message:', message)

    for bits in (1024, 2048):
        print('\n--- Testingwith',bits,   'bitkeys---')
        t0 = time.time()
        pub, priv, primes = generate_keypair(bits)
        t1 = time.time()
        print('Keygenerationtime:   {:.2f}s'.format(t1 - t0))
        n, e = pub
        print('n bit-length:', n.bit_length())

        t0 = time.time()
        ciphertext = encrypt(pub, message)
        t1 = time.time()
        print('Encrypttime:  {:.4f}s'.format(t1-t0))
```



```
        t0 = time.time()
        plaintext = decrypt(priv, ciphertext)
        t1 = time.time()
        print('Decrypt time: {:.4f}s'.format(t1 - t0))

        print('Decrypted equals original?', plaintext == message)
```



```
Message: b'Hello RSA! This is a short test.'

--- Testing with 1024 bit keys ---
Key generation time: 0.30s
n bit-length: 1023
Encrypt time: 0.0000s
Decrypt time: 0.0000s
Decrypted equals original? True

--- Testing with 2048 bit keys ---
Key generation time: 14.37s
n bit-length: 2048
Encrypt time: 0.0000s
Decrypt time: 0.1398s
Decrypted equals original? True
```

Observations and Conclusion:

1. ***Key Generation***
   - o For ***1024-bit keys***, key generation took noticeably less time compared to ***2048-bit keys***.
   - o As the key size increases, the generation of large prime numbers becomes slower due to primality testing.
2. ***Encryption and Decryption***
   - o Encryption was consistently faster than decryption, since the public exponent e is small (commonly 65537).
   - o Decryption, which uses the private key with a large exponent d, takes more computation time.
   - o The decrypted message matched the original plaintext, verifying correctness.
3. ***Performance Trends***
   - o Larger key sizes (e.g., 2048 bits) increase security but also increase computational cost.
   - o Even with 2048-bit keys, the program was able to encrypt and decrypt short messages successfully within reasonable time.

The RSA algorithm was successfully implemented in Python, demonstrating ***key generation, encryption, and decryption***.
The program validated the theoretical foundations of RSA, including prime generation, modular arithmetic, and modular exponentiation.
The results confirm that RSA ensures ***confidentiality and correctness***, as decrypted outputs matched the original messages.
However, RSA is computationally intensive for large keys and is not efficient for encrypting long messages directly.
In practice, RSA is often combined with symmetric algorithms (like AES) in a ***hybrid cryptosystem***, where RSA secures the symmetric key.