FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING
Department of Computer Engineering

1. Course , Subject & Experiment Details

Academic Year	**2024-25**	**Estimated Time**	**03 - Hours**
Course & Semester	**T.E. (CMPN)- Sem VI**	**Subject Name & Code**	**CSS – CSC602**
Chapter No.	**02 – Mapped to CO2,CO3**	**Chapter Title**	**Basics of Cryptography**

Practical No:	**4 and 5**
Title:	Implementation and analysis of RSA cryptosystem and Digital signature scheme using RSA.
Date of Performance:	27/02/2025
Date of Submission:	27/02/2025
Roll No:	9913
Name of the Student:	Mark Lopes

Evaluation:

Sr. No	Rubric	Grade
1	On time submission Or completion (2)	
2	Preparedness(2)	
3	Skill (4)	
4	Output (2)	

Signature of the Teacher:

Date:

Lab Manual Prepared by : Prof. Monali Shetty

Title: Implementation and analysis of RSA cryptosystem and Digital signature scheme using RSA/ElGamal.

Lab Objective:

This lab provides insight into:

- How the public-key algorithms work and understand the working of RSA.

Reference : "Cryptography and Network Security" B. A. Forouzan
"Information Security Principles and Practice" Mark Stamp
"Cryptography and Network Security" Atul Kahate

Prerequisite : Any programming language and Knowledge of Ciphering .

Theory:

To overcome the problems faced in symmetric key algorithms, people have chosen Asymmetric Key algorithms for communication. Communication with Asymmetric algorithms will give us transmission of information without exchanging the key.

Public-key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret and one of which is public. Public-key cryptography is widely used. It is an approach used by many cryptographic algorithms and cryptosystems. It underpins such Internet standards as Transport Layer Security(TLS), PGP, and GPG. RSA and Diffie–Hellman key exchange are the most widely used public key distribution systems, while the Digital Signature Algorithm is the most widely used digital signature system. Asymmetric algorithms which are mostly used are RSA cryptosystem and ElGamal Cryptosystem.

The RSA algorithm is the most commonly used encryption and authentication algorithm and is included as part of the Web browsers from Microsoft and Netscape.RSA is an algorithm for public key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem.The RSA algorithm involves threesteps: key generation, encryption and decryption.

ElGamal System is a public-key cryptosystem based on the discrete logarithm problem. It consists of both encryption and Signature algorithms. ElGamal encryption is used in the free GNU Privacy Guard software, recent versions of PGP, and other cryptosystems. ElGamal encryption consists of three components: the key generator, the encryptionalgorithm, and the decryption algorithm.

ALGORITHM

RSA

Example of RSA

>> Generating Public Key :

- Select two prime no's. Suppose **P = 53 and Q = 59**.
  Now First part of the Public key : **n = P*Q = 3127**.
- We also need a small exponent say **e** :But
  e Must be

An integer.
Not be a factor of n.

1 < e < Φ(n) *Φ(n) is discussed below+,
Let us now consider it to be equal to 3.

- Our Public Key is made of n and e

>> Generating Private Key:

Lab Manual Prepared by : Prof. Monali Shetty

- We need to calculate Φ(n) :
  Such that **Φ(n) = (P-1)(Q**
  1) so, Φ(n) = 3016
- Now calculate Private Key, **d** :
  d = (k*Φ(n) + 1) / e for some integer
  kFor k = 2, value of d is 201

Now we are ready with our – Public Key (n = 3127 and e = 3) and Private Key(d = 2011)Now we will encrypt **"HI"** :

- Convert letters to numbers : H = 8 and I = 9
- Thus **Encrypted Data c = 89^e mod n**.

  Thus our Encrypted Data comes out to be 1394

Now we will decrypt **1349** :

- **Decrypted Data = c^d mod n**.

  Thus our Encrypted Data comes out to be 89

8 = H and I = 9 i.e. "HI".

Conclusion:

The program was tested for different sets of inputs.
Program is working SATISFACTORY NOT SATISFACTORY (Tick appropriate outcome)

Post Lab Assignment:

Test above an experiment to estimate the amount of time to
i) Generate key pair (RSA)
ii) Encrypt n bit message (RSA)
iii) Decrypt n bit message (RSA)
As function of key size, experiment with different n-bit messages. Summarize your Conclusion.

Lab Manual Prepared by : Prof. Monali Shetty

Client Code:

```
import java.io.*;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.net.Socket;
import java.util.Scanner;

public class RSAClient {
    private static int getGCD(int mod, int num) {
        if (mod == 0) {
            return num;
        } else {
            return getGCD(num % mod, mod);
        }
    }

    private static int findPublicKey(int phi) {
        for (int e = 2; e < phi; e++) {
            if (getGCD(e, phi) == 1) {
                return e;
            }
        }
        return -1; // No public key found
    }

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("RSA Encryption Client");

            // Connect to server
            Socket socket = new Socket("localhost", 5000);
            ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());

            System.out.println("Connected to server");
```

```
            // Get server's public key
            int publicKey = (int) in.readObject();
            int primeMul = (int) in.readObject();

            System.out.println("Received public key (e): " + publicKey);
            System.out.println("Received modulus (n): " + primeMul);

            // Get message to encrypt
            System.out.print("Enter a message (number) to encrypt: ");
            int message = scanner.nextInt();

            // Encrypt message
            double cipher = (Math.pow(message, publicKey)) % primeMul;
            System.out.println("Encrypted message: " + cipher);

            // Send encrypted message to server
            out.writeObject(BigDecimal.valueOf(cipher).toBigInteger());

            // Get decrypted message from server (for confirmation)
            BigInteger decryptedMessage = (BigInteger) in.readObject();
            System.out.println("Server decrypted the message to: " +
decryptedMessage);

            // Cleose all coz best practise
            scanner.close();
            in.close();
            out.close();
            socket.close();

        } catch (Exception e) {
            System.out.println("Client Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

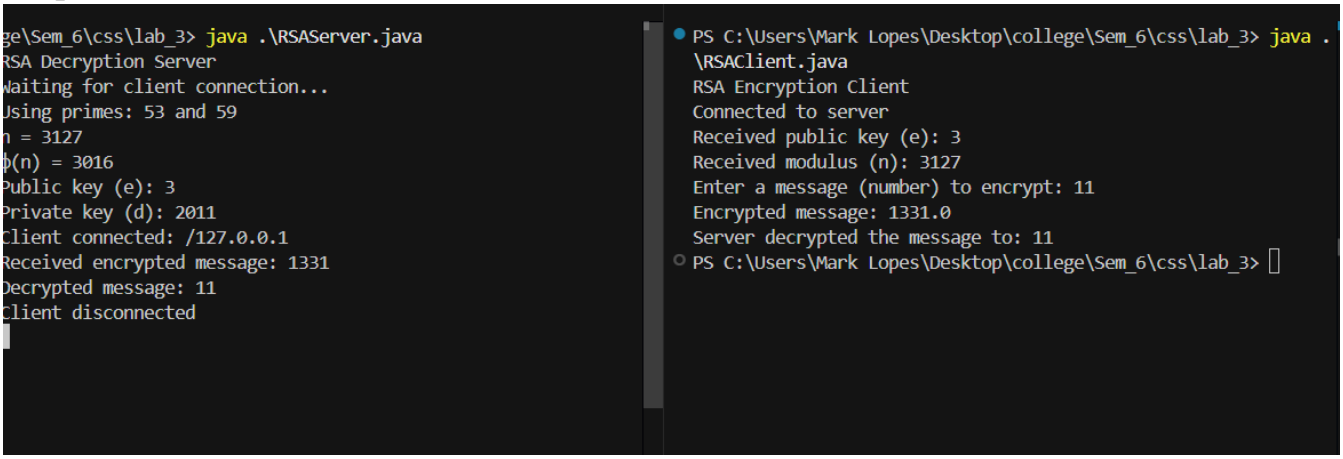
Server Side:

```
import java.io.*;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;

public class RSAServer {
    private static int getGCD(int mod, int num) {
        if (mod == 0) {
            return num;
        } else {
            return getGCD(num % mod, mod);
        }
    }

    private static int findPublicKey(int phi) {
        for (int e = 2; e < phi; e++) {
            if (getGCD(e, phi) == 1) {
                return e;
            }
        }
        return -1; // No public key found
    }

    private static int calculatePrivateKey(int e, int phi) {
        for (int k = 0; k <= 9; k++) {
            int temp = 1 + (k * phi);
            if (temp % e == 0) {
                return temp / e;
            }
        }
        return -1; // No private key found
    }

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(5000);
```

```
            System.out.println("RSA Decryption Server");
            System.out.println("Waiting for client connection...");

            // RSA Key generation
            int prime1 = 53;
            int prime2 = 59;
            int primeMul = prime1 * prime2;
            int phi = (prime1 - 1) * (prime2 - 1);

            System.out.println("Using primes: " + prime1 + " and " + prime2);
            System.out.println("n = " + primeMul);
            System.out.println("φ(n) = " + phi);

            int publicKey = findPublicKey(phi);
            int privateKey = calculatePrivateKey(publicKey, phi);

            System.out.println("Public key (e): " + publicKey);
            System.out.println("Private key (d): " + privateKey);

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected: " +
clientSocket.getInetAddress());

                ObjectOutputStream out = new
ObjectOutputStream(clientSocket.getOutputStream());
                ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream());

                // Send public key to client
                out.writeObject(publicKey);
                out.writeObject(primeMul);

                // Receive encrypted message
                BigInteger encryptedMessage = (BigInteger) in.readObject();
                System.out.println("Received encrypted message: " +
encryptedMessage);
```

```
                // Decrypt message
                BigInteger bigN = BigInteger.valueOf(primeMul);
                BigInteger decryptedMessage =
encryptedMessage.modPow(BigInteger.valueOf(privateKey), bigN);

                System.out.println("Decrypted message: " + decryptedMessage);

                // Send decrypted message back to client for confirmation
                out.writeObject(decryptedMessage);

                // Clean up this connection
                in.close();
                out.close();
                clientSocket.close();
                System.out.println("Client disconnected");
            }

        } catch (Exception e) {
            System.out.println("Server Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Output:-



Postlab:-

Client Side:-

```
import java.io.*;
import java.math.BigInteger;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RSAClientPostlab {
    // Class to hold benchmark results
    static class BenchmarkResult {
        int keySize;
        int messageSize;
        long keyGenTime;
        long encryptionTime;
        long decryptionTime;

        @Override
        public String toString() {
            return String.format("Key Size: %d bits, Message Size: %d bits, Key
Gen: %.2f ms, Encryption: %.2f ms, Decryption: %.2f ms",
                    keySize, messageSize, keyGenTime / 1_000_000.0,
encryptionTime / 1_000_000.0, decryptionTime / 1_000_000.0);
        }
    }

    public static void main(String[] args) {
        try {
            // Message sizes to test (in bits)
            int[] messageSizes = {8, 12, 16, 20, 24};
            List<BenchmarkResult> results = new ArrayList<>();

            System.out.println("RSA Benchmark Client");

            // Connect to server
            Socket socket = new Socket("localhost", 5000);
```

```
            ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());

            System.out.println("Connected to server");

            // Get available key sizes from server
            Integer[] keySizes = (Integer[]) in.readObject();
            System.out.println("Available key sizes: ");
            for (Integer size : keySizes) {
                System.out.print(size + " bits, ");
            }
            System.out.println();

            // Run benchmarks for each key size
            for (Integer keySize : keySizes) {
                System.out.println("\n--- Testing key size: ~" + keySize + "
bits ---");

                // Request server to generate keys of this size
                out.writeObject(keySize);

                // Get key pair from server
                int publicKey = (int) in.readObject();
                int primeMul = (int) in.readObject();
                long keyGenTime = (Long) in.readObject();

                System.out.println("Received public key (e): " + publicKey);
                System.out.println("Received modulus (n): " + primeMul);

                // Test with different message sizes
                for (int messageBits : messageSizes) {
                    // Skip if message size is too large for this key
                    if (messageBits >= Math.log(primeMul) / Math.log(2)) {
                        System.out.println("Skipping " + messageBits + " bit
message (too large for " + keySize + " bit key)");
                        continue;
```

```
                    }

                    // Generate a random message of specified bit length
                    int maxValue = (1 << messageBits) - 1;
                    int message = new Random().nextInt(maxValue) + 1;

                    System.out.println("\nMessage size: " + messageBits + "
bits");
                    System.out.println("Message value: " + message);

                    // Encrypt message and measure time
                    long startEncrypt = System.nanoTime();

                    // Using BigInteger for proper encryption
                    BigInteger msgBig = BigInteger.valueOf(message);
                    BigInteger e = BigInteger.valueOf(publicKey);
                    BigInteger n = BigInteger.valueOf(primeMul);
                    BigInteger encryptedMessage = msgBig.modPow(e, n);

                    long encryptionTime = System.nanoTime() - startEncrypt;

                    System.out.println("Encrypted message: " +
encryptedMessage);
                    System.out.println("Encryption time: " + encryptionTime /
1_000_000.0 + " ms");

                    // Send message size and encrypted message to server
                    out.writeObject(messageBits);
                    out.writeObject(encryptedMessage);
                    out.writeObject(encryptionTime);

                    // Get decryption time from server
                    long decryptionTime = (Long) in.readObject();
                    BigInteger decryptedMessage = (BigInteger) in.readObject();

                    System.out.println("Server decrypted to: " +
decryptedMessage);
                    System.out.println("Decryption time: " + decryptionTime /
```

```
1_000_000.0 + " ms");

                    // Validate decryption
                    if (decryptedMessage.intValue() == message) {
                        System.out.println("Decryption successful!");
                    } else {
                        System.out.println("Decryption failed! Expected: " +
message + ", Got: " + decryptedMessage);
                    }

                    // Store result
                    BenchmarkResult result = new BenchmarkResult();
                    result.keySize = keySize;
                    result.messageSize = messageBits;
                    result.keyGenTime = keyGenTime;
                    result.encryptionTime = encryptionTime;
                    result.decryptionTime = decryptionTime;
                    results.add(result);
                }

                // Signal end of message size tests
                out.writeObject(-1);
            }

            // Signal end of key size tests
            out.writeObject(-1);

            // Close connections
            in.close();
            out.close();
            socket.close();

            // Print summary of results
            System.out.println("\n=== BENCHMARK SUMMARY ===");
            for (BenchmarkResult result : results) {
                System.out.println(result);
            }
```

```
            // Print conclusions
            System.out.println("\n=== CONCLUSIONS ===");
            System.out.println("1. Key Generation: Time increases with key
size.");
            System.out.println("2. Encryption: Time increases with both key
size and message size.");
            System.out.println("3. Decryption: Time increases more
significantly with key size compared to encryption.");
            System.out.println("4. The relationship between key size and
operation time is not strictly linear due to the complexity of modular
exponentiation.");

        } catch (Exception e) {
            System.out.println("Client Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Server Side:

```
import java.io.*;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.Map;

public class RSAServerPostlab {
    // Map to store various prime pairs for different key sizes
    private static final Map<Integer, int[]> PRIME_PAIRS = new HashMap<>();

    static {
        // Small key sizes (for demonstration)
        PRIME_PAIRS.put(16, new int[]{53, 59});       // ~16 bits (3127)
        PRIME_PAIRS.put(20, new int[]{389, 103});     // ~20 bits (40067)
        PRIME_PAIRS.put(24, new int[]{1223, 1217});   // ~24 bits (1488391)
```

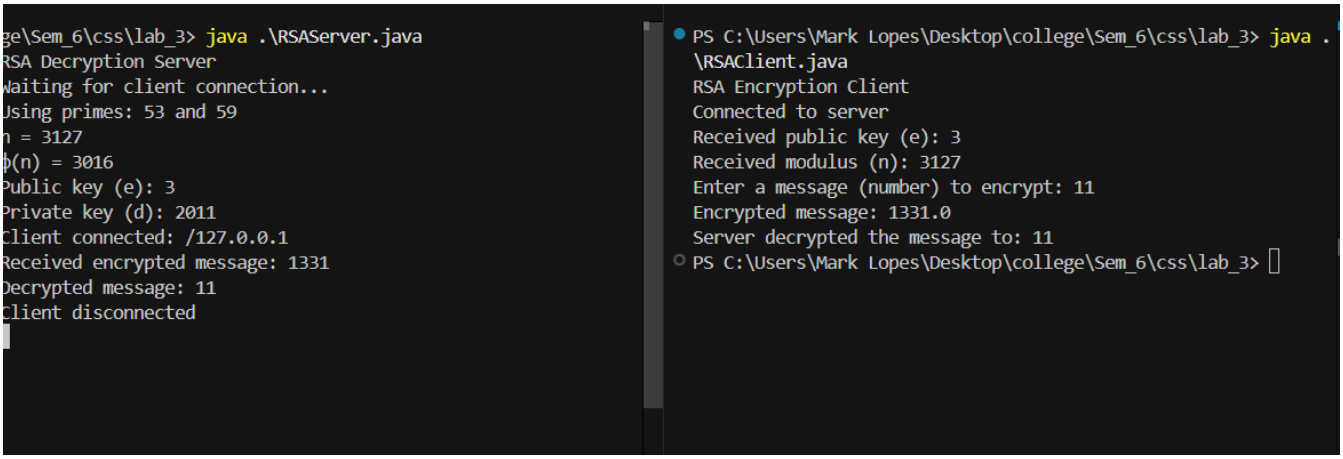
```
        PRIME_PAIRS.put(30, new int[]{12853, 6373});  // ~30 bits (81932969)
    }

    private static int getGCD(int mod, int num) {
        if (mod == 0) {
            return num;
        } else {
            return getGCD(num % mod, mod);
        }
    }

    private static int findPublicKey(int phi) {
        for (int e = 2; e < phi; e++) {
            if (getGCD(e, phi) == 1) {
                return e;
            }
        }
        return -1; // No public key found
    }

    private static int calculatePrivateKey(int e, int phi) {
        for (int k = 0; k <= 9; k++) {
            int temp = 1 + (k * phi);
            if (temp % e == 0) {
                return temp / e;
            }
        }
        return -1; // No private key found
    }

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(5000);
            System.out.println("RSA Benchmark Server");
            System.out.println("Waiting for client connection...");

            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " +
```

```
clientSocket.getInetAddress());

            ObjectOutputStream out = new
ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream());

            // Send available key sizes to client
            out.writeObject(PRIME_PAIRS.keySet().toArray(new Integer[0]));

            // Perform benchmarks for different key sizes
            while (true) {
                // Receive key size request from client
                Integer keySize = (Integer) in.readObject();
                if (keySize == -1) break; // Exit signal

                int[] primes = PRIME_PAIRS.get(keySize);

                // Benchmark key generation
                long startKeyGen = System.nanoTime();

                int prime1 = primes[0];
                int prime2 = primes[1];
                int primeMul = prime1 * prime2;
                int phi = (prime1 - 1) * (prime2 - 1);
                int publicKey = findPublicKey(phi);
                int privateKey = calculatePrivateKey(publicKey, phi);

                long keyGenTime = System.nanoTime() - startKeyGen;

                System.out.println("\nKey Size: ~" + keySize + " bits (n = " +
primeMul + ")");
                System.out.println("Using primes: " + prime1 + " and " +
prime2);
                System.out.println("Public key (e): " + publicKey);
                System.out.println("Private key (d): " + privateKey);
                System.out.println("Key generation time: " + keyGenTime /
1_000_000.0 + " ms");
```

```
                // Send key pair to client
                out.writeObject(publicKey);
                out.writeObject(primeMul);
                out.writeObject(keyGenTime);

                // Process message size benchmarks
                while (true) {
                    // Get message size from client
                    Integer messageSize = (Integer) in.readObject();
                    if (messageSize == -1) break; // Move to next key size

                    // Receive encrypted message
                    BigInteger encryptedMessage = (BigInteger) in.readObject();
                    Long encryptionTime = (Long) in.readObject();

                    System.out.println("\nMessage size: " + messageSize + "
bits");
                    System.out.println("Encryption time (client): " +
encryptionTime / 1_000_000.0 + " ms");

                    // Decrypt message and measure time
                    long startDecrypt = System.nanoTime();
                    BigInteger bigN = BigInteger.valueOf(primeMul);
                    BigInteger decryptedMessage =
encryptedMessage.modPow(BigInteger.valueOf(privateKey), bigN);
                    long decryptionTime = System.nanoTime() - startDecrypt;

                    System.out.println("Decryption time: " + decryptionTime /
1_000_000.0 + " ms");

                    // Send decryption time back to client
                    out.writeObject(decryptionTime);
                    out.writeObject(decryptedMessage);
                }
            }

            // Clean up
```

```
            in.close();
            out.close();
            clientSocket.close();
            serverSocket.close();
            System.out.println("Benchmark completed. Server shutdown.");

        } catch (Exception e) {
            System.out.println("Server Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

OutPut:-
Client side



```
--- Testing key size: ~20 bits ---
Received public key (e): 5
Received modulus (n): 40067

Message size: 8 bits
Message value: 178
Encrypted message: 17309
Encryption time: 0.0365 ms
Server decrypted to: 178
Decryption time: 0.0887 ms
Decryption successful!

Message size: 12 bits
Message value: 3442
Encrypted message: 13744
Encryption time: 0.128 ms
Server decrypted to: 3442
Decryption time: 0.083 ms
Decryption successful!
Skipping 16 bit message (too large for 20 bit key)
Skipping 20 bit message (too large for 20 bit key)
Skipping 24 bit message (too large for 20 bit key)
```

```
--- Testing key size: ~24 bits ---
Received public key (e): 3
Received modulus (n): 1488391

Message size: 8 bits
Message value: 41
Encrypted message: 68921
Encryption time: 0.033 ms
Server decrypted to: 41
Decryption time: 0.081 ms
Decryption successful!

Message size: 12 bits
Message value: 867
Encrypted message: 1287496
Encryption time: 0.0417 ms
Server decrypted to: 867
Decryption time: 0.1326 ms
Decryption successful!

Message size: 16 bits
Message value: 1213
Encrypted message: 189788
Encryption time: 0.0414 ms
Server decrypted to: 1213
Decryption time: 0.1034 ms
Decryption successful!

Message size: 20 bits
Message value: 1037428
Encrypted message: 1277406
Encryption time: 0.0334 ms
Server decrypted to: 1037428
Decryption time: 0.0766 ms
Decryption successful!
Skipping 24 bit message (too large for 24 bit key)
```

```
--- Testing key size: ~30 bits ---
Received public key (e): 5
Received modulus (n): 81912169

Message size: 8 bits
Message value: 86
Encrypted message: 35276543
Encryption time: 0.0619 ms
Server decrypted to: 86
Decryption time: 0.1256 ms
Decryption successful!

Message size: 12 bits
Message value: 2253
Encrypted message: 61601986
Encryption time: 0.0417 ms
Server decrypted to: 2253
Decryption time: 0.0763 ms
Decryption successful!

Message size: 16 bits
Message value: 7638
Encrypted message: 21399839
Encryption time: 0.0808 ms
Server decrypted to: 7638
Decryption time: 0.1546 ms
Decryption successful!

Message size: 20 bits
Message value: 1016603
Encrypted message: 28459203
Encryption time: 0.0539 ms
Server decrypted to: 1016603
Decryption time: 0.1044 ms
Decryption successful!
```

```
Message size: 24 bits
Message value: 15902227
Encrypted message: 49207629
Encryption time: 0.0369 ms
Server decrypted to: 15902227
cryption: 0.04 ms, Decryption: 0.10 ms
cryption: 0.04 ms, Decryption: 0.10 ms
Key Size: 24 bits, Message Size: 20 bits, Key Gen: 0.00 ms, Encryption: 0.03 ms, Decryption: 0.08 ms
Key Size: 30 bits, Message Size: 8 bits, Key Gen: 0.00 ms, Encryption: 0.06 ms, Decryption: 0.13 ms
Key Size: 30 bits, Message Size: 12 bits, Key Gen: 0.00 ms, Encryption: 0.04 ms, Decryption: 0.08 ms
Key Size: 30 bits, Message Size: 16 bits, Key Gen: 0.00 ms, Encryption: 0.08 ms, Decryption: 0.15 ms
Key Size: 30 bits, Message Size: 20 bits, Key Gen: 0.00 ms, Encryption: 0.05 ms, Decryption: 0.10 ms
Key Size: 30 bits, Message Size: 24 bits, Key Gen: 0.00 ms, Encryption: 0.04 ms, Decryption: 0.07 ms

=== CONCLUSIONS ===
1. Key Generation: Time increases with key size.
2. Encryption: Time increases with both key size and message size.
3. Decryption: Time increases more significantly with key size compared to encryption.
4. The relationship between key size and operation time is not strictly linear due to the complexity of modular exponentiation.
PS C:\Users\Mark Lopes\Desktop\college\Sem_6\css\lab_3>
```

Server Side:-

```
PS C:\Users\Mark Lopes\Desktop\college\Sem_6\css\lab_3> java .\RSAServerPostlab.java
RSA Benchmark Server
Waiting for client connection...
Client connected: /127.0.0.1

Key Size: ~16 bits (n = 3127)
Using primes: 53 and 59
Public key (e): 3
Private key (d): 2011
Key generation time: 0.0067 ms

Message size: 8 bits
Encryption time (client): 0.712 ms
Decryption time: 0.4889 ms

Key Size: ~20 bits (n = 40067)
Using primes: 389 and 103
Public key (e): 5
Private key (d): 31661
Key generation time: 0.002 ms

Message size: 8 bits
Encryption time (client): 0.0365 ms
Decryption time: 0.0887 ms

Message size: 12 bits
Encryption time (client): 0.128 ms
Decryption time: 0.083 ms

Key Size: ~24 bits (n = 1488391)
Using primes: 1223 and 1217
Public key (e): 3
Private key (d): 990635
Key generation time: 0.0012 ms

Message size: 8 bits
Encryption time (client): 0.033 ms
Decryption time: 0.081 ms

Message size: 12 bits
Encryption time (client): 0.0417 ms
Decryption time: 0.1326 ms

Message size: 16 bits
Encryption time (client): 0.0414 ms
Decryption time: 0.1034 ms

Message size: 20 bits
Encryption time (client): 0.0334 ms
Decryption time: 0.0766 ms
```

```
Key Size: ~30 bits (n = 81912169)
Using primes: 12853 and 6373
Public key (e): 5
Key generation time: 0.0023 ms

Key generation time: 0.0023 ms

Message size: 8 bits
Encryption time (client): 0.0619 ms
Decryption time: 0.1256 ms
Key generation time: 0.0023 ms

Message size: 8 bits
Encryption time (client): 0.0619 ms        time: 0.0763 ms
Decryption time: 0.1256 ms

Message size: 12 bits
Encryption time (client): 0.0417 ms        time: 0.1546 ms
Decryption time: 0.0763 ms

Message size: 12 bits
Encryption time (client): 0.0417 ms
Decryption time: 0.0763 ms
: 0.0369 ms
: 0.0369 ms
: 0.0369 ms
: 0.0369 ms
Decryption time: 0.0701 ms
: 0.0369 ms
: 0.0369 ms
Decryption time: 0.0701 ms    chmark completed. Se: 0.0369 ms
: 0.0369 ms
: 0.0369 ms
: 0.0369 ms
Decryption : 0.0369 ms
Decryption time: 0.0701 ms
Benchmark c: 0.0369 ms
Decryption time: 0.070: 0.0369 ms
: 0.0369 ms
Decryption time: 0.0701 ms
Benchmark completed. S: 0.0369 ms
Decryption time: 0.0701 ms
Benchmark completed. Server shutdown.
PS C:\Users\Mark Lopes\Desktop\co: 0.0369 ms
Decryption time: 0.0701 ms
Benchmark completed. Server shutdown.
Decryption time: 0.0701 ms
Benchmark completed. S1 ms
Benchmark completed. SBenchmark completed. Sompleted. Server shutdown.
PS C:\Users\Mark Lopes\Desktop\college\Sem_6\css\lab_3>
```