1. Project Overview

Objective: Implement RSA encryption and decryption using key sizes of 1024 and 2048 bits.

Requirements:
- Implement the RSA algorithm for public key cryptography, including key generation, encryption, and decryption.
- Use Java, leveraging its BigInteger class for handling large integers essential for RSA.

Importance of Key Sizes:
- The key size determines the security level of RSA. Larger keys (e.g., 2048-bit) are more secure but require more computational resources. Key size affects the strength of encryption and resistance to brute-force attacks, with 2048-bit keys currently recommended for long-term security.

2. Understanding RSA Algorithm

RSA Basics:
- RSA (Rivest-Shamir-Adleman) is a widely used algorithm for secure data transmission. It is based on the mathematical difficulty of factoring large prime numbers.

Key Generation:
- Step 1: Generate two large prime numbers, p and q.
- Step 2: Calculate the modulus $n = p \times q$, which will be used in both encryption and decryption.
- Step 3: Calculate the totient $\phi(n) = (p- 1) \times (q- 1)$.
- Step 4: Choose a public exponent e, commonly 65537, which has properties that balance security and performance.
- Step 5: Calculate the private exponent d such that $d \times e \equiv 1 \, (\text{mod} \, \phi(n))$, using the Extended Euclidean Algorithm to find d.

Encryption:
- Convert the plaintext message to an integer m such that $0 \leq m < n$.
- Compute ciphertext $c = m^e \mod n$.

Decryption:
- Compute the original message $m = c^d \mod n$.
- Convert integer m back to plaintext.

RSA requires that each team member fully understands these processes to ensure the algorithm is implemented correctly.

3. Programming Language and Tools

Chosen Language: Java

Reason:
- Java's BigInteger class natively supports large integers and operations needed for RSA.
- It has robust libraries for secure random number generation and cryptographic functions.

Tools:
- IDE: IntelliJ IDEA or Eclipse for writing and testing the code.
- JDK: Version 8 or higher is recommended due to compatibility with cryptographic libraries and the BigInteger class.

4. Implementation Plan

A. Key Generation Module

1. Prime Number Generation:
- Goal: Generate large prime numbers of 512 bits for 1024-bit RSA and 1024 bits for 2048-bit RSA.
- Method: Use Java's SecureRandom for randomness and the Miller-Rabin primality test to generate probable primes.

- Task: Implement a function to generate primes and verify primality using BigInteger.probablePrime().

2. Modulus (n) and Totient (ϕ(n)) Calculation:
- n Calculation: Multiply the primes p and q to get n.
- ϕ(n) Calculation: Compute $ϕ(n) = (p- 1) \times (q- 1)$.

3. Public Exponent (e):
- Commonly set to 65537 as it is a secure and efficient choice for the public exponent, with a low chance of GCD overlap with $ϕ(n)$.

4. Private Exponent (d):
- Use the Extended Euclidean Algorithm to calculate d, which is the modular inverse of e modulo $ϕ(n)$.

5. Key Storage:
- Store keys in Base64-encoded format for compatibility and security. Consider secure storage practices for the private key, such as encryption or file protection.

B. Encryption Function

1. Message Preparation:
- Convert the plaintext message to a format compatible with RSA, such as a byte array or integer. Apply padding (OAEP with SHA-256 in our case) to prevent predictable encryption results.

2. Encryption Process:
- Compute the ciphertext $c = m^e \mod n$ using Java's Cipher class with RSA/OAEP padding.

C. Decryption Function

1. Decryption Process:
- Compute the original message $m = c^d \mod n$.

2. Message Recovery:
- Convert the integer m back to a readable plaintext format.

5. Testing and Validation

- Unit Tests: Test key generation, encryption, and decryption modules individually.
- Integration Tests: Verify the end-to-end encryption-decryption process using known plaintext.
- Performance Tests: Measure the time taken for key generation, encryption, and decryption for both 1024 and 2048-bit keys.
- Edge Cases: Test boundaries (e.g., very small messages, maximum size messages) to verify handling.

6. Optimization

1. Algorithm Optimization:
- Use efficient modular exponentiation (e.g., square-and-multiply or Montgomery multiplication) for faster encryption and decryption.

2. Code Optimization:
- Profile and optimize the code, especially the prime generation and modular exponentiation functions, to reduce execution time.

3. Library Utilization:
- Use Java's BigInteger and SecureRandom for optimized big integer operations and secure randomness, respectively.

7. Documentation

- Code Documentation: Use inline comments to explain each function's purpose and complex steps.

- Project Report: Include a report detailing the RSA algorithm, chosen methods, challenges encountered, and how each requirement was addressed.

8. README for the Project

Project Title: RSA Encryption & Decryption with Java

Description: This project implements the RSA encryption and decryption algorithm with key sizes of 1024 and 2048 bits. It includes modules for key generation, encryption, decryption, and an example main function to demonstrate usage.

Requirements:
- Java JDK 8 or higher
- IDE: IntelliJ IDEA or Eclipse (optional but recommended for ease of use)

Files:
- RSAKeyPairGenerator.java: Generates public and private keys.
- RSAEncryptor.java: Encrypts messages using the RSA algorithm.
- RSADecryptor.java: Decrypts messages encrypted with RSA.
- Main.java: Demonstrates encryption and decryption with sample messages.

Usage:
1. Setup:
   - Ensure JDK 8 or higher is installed.
   - Save all .java files in the same directory.
2. Compilation:
   javac *.java
3. Execution:
   java Main
4. Sample Output:
   - Shows encrypted and decrypted messages for 1024 and 2048-bit keys.

Security Considerations:
- Utilizes OAEP padding for enhanced security.
- Generates keys with SecureRandom for cryptographic randomness.

Enhancements:
- Hybrid encryption using AES for large messages.
- Key storage and retrieval functionality.

9. Summary for Presentation

- RSA Basics: Explanation of RSA and the significance of public/private keys.
- Key Generation: Process of generating large primes, modulus, totient, and the private exponent.
- Encryption/Decryption: Steps involved in securing a message and retrieving it.
- Testing and Validation: Methods to ensure accurate and efficient encryption/decryption.
- Security & Optimization: Measures taken to secure keys and optimize performance.
- Code Walkthrough: Brief overview of each file and functionality.