RSA Algorithm

Aim:

The aim of this program is to implement the RSA algorithm, which is a widely used public-key encryption system. The program generates a key pair (public and private keys), encrypts a message using the public key, and then decrypts it using the private key.

Description:

This program demonstrates RSA encryption and decryption using Java. It covers key generation, encryption, and decryption steps. The program follows these basic steps:

1. Generate two large prime numbers.
2. Compute the modulus n and Euler's totient φ(n).
3. Choose a public exponent e and ensure it is coprime with φ(n).
4. Compute the private exponent d such that e * d ≡ 1 (mod φ(n)).
5. Use the public key (e, n) for encryption and the private key (d, n) for decryption.

Code:

1. RSA.java

```java
import java.math.BigInteger;
import java.security.SecureRandom;

public class RSA {

    // Generate a large prime number
    private static BigInteger generatePrime(int bitLength) {
        SecureRandom rand = new SecureRandom();
        return BigInteger.probablePrime(bitLength, rand);
    }

    // Extended Euclidean Algorithm to find gcd and the modular inverse
    private static BigInteger extendedGCD(BigInteger a, BigInteger b) {
        BigInteger[] result = {BigInteger.ZERO, BigInteger.ONE, a};
        BigInteger[] temp;
        while (b.compareTo(BigInteger.ZERO) > 0) {
            BigInteger[] q = a.divideAndRemainder(b);
            a = b;
            b = q[1];
            temp = result;
            result = new BigInteger[] {temp[1].subtract(q[0].multiply(temp[0])), temp[0], temp[2]};
        }
        return result[1].mod(a); // Modular inverse
    }

    // Generate RSA keys
    public static BigInteger[] generateKeyPair(int bitLength) {
        BigInteger p = generatePrime(bitLength / 2);
```

```
        BigInteger q = generatePrime(bitLength / 2);
        BigInteger n = p.multiply(q);
        BigInteger phi = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));

        // Choose public exponent e (commonly 65537)
        BigInteger e = BigInteger.valueOf(65537);

        // Ensure e and phi(n) are coprime (gcd(e, phi(n)) = 1)
        while (e.gcd(phi).compareTo(BigInteger.ONE) != 0) {
            e = e.add(BigInteger.TWO); // Try next odd number
        }

        // Calculate the private exponent d such that e * d ≡ 1 (mod φ(n))
        BigInteger d = extendedGCD(e, phi);

        // Public key is (e, n), Private key is (d, n)
        return new BigInteger[] {e, n, d};
    }

    // Encrypt a message using the public key (e, n)
    public static BigInteger encrypt(BigInteger message, BigInteger e, BigInteger n) {
        return message.modPow(e, n);
    }

    // Decrypt a message using the private key (d, n)
    public static BigInteger decrypt(BigInteger cipherText, BigInteger d, BigInteger n) {
        return cipherText.modPow(d, n);
    }

    public static void main(String[] args) {
        int bitLength = 512;  // Key size in bits

        // Step 1: Generate RSA keys
        BigInteger[] keys = generateKeyPair(bitLength);
        BigInteger e = keys[0];  // Public exponent
        BigInteger n = keys[1];  // Modulus
        BigInteger d = keys[2];  // Private exponent

        System.out.println("Public Key (e, n): (" + e + ", " + n + ")");
        System.out.println("Private Key (d, n): (" + d + ", " + n + ")");

        // Step 2: Encrypt a message (convert string to BigInteger)
        String message = "Hello RSA!";
        BigInteger messageBigInt = new BigInteger(message.getBytes());
        System.out.println("Original Message: " + message);
        BigInteger cipherText = encrypt(messageBigInt, e, n);
```

```java
        System.out.println("Encrypted Message: " + cipherText);

        // Step 3: Decrypt the message
        BigInteger decryptedMessageBigInt = decrypt(cipherText, d, n);
        String decryptedMessage = new String(decryptedMessageBigInt.toByteArray());
        System.out.println("Decrypted Message: " + decryptedMessage);
    }
}
```

Output:

```
Public Key (e, n): (65537, 551813440433008827971444996955565127977462202495476694049969027286657467905278984692857353040926284225895571311254824563692123776492089455766627575822701)
Private Key (d, n): (0, 551813440433008827971444996955565127977462202495476694049969027286657467905278984692857353040926284225895571311254824563692123776492089455766627575822701)
Original Message: Hello RSA!
Encrypted Message: 2648137819973182600152913478840281652331470051608318799618698523135790100590237729532804203172343185127530772815850679483077162794941890493039809440067793
```

Code Explanation:

1. **Key Generation**:
    - Two large prime numbers p and q are generated randomly.
    - The modulus n is the product of p and q.
    - Euler's totient φ(n) is computed as (p - 1) * (q - 1).
    - The public exponent e is chosen as 65537, a common value in RSA, and adjusted if necessary to ensure it's coprime with φ(n).
    - The private exponent d is computed using the Extended Euclidean Algorithm.
2. **Encryption**:
    - The message is first converted into a BigInteger (from a string).
    - The message is then encrypted using the formula cipherText = message^e % n.
3. **Decryption**:
    - The encrypted message is decrypted using the formula message = cipherText^d % n, retrieving the original message.

Complexity Analysis:

- **Time Complexity**:
    - **Key Generation**: Generating large primes (O(k^4), where k is the bit length) and computing the GCD for modular inverse using the Extended Euclidean Algorithm (O(k^2)).
    - **Encryption and Decryption**: Both involve modular exponentiation, which has a time complexity of O(k^3) due to the use of exponentiation by squaring.
- **Space Complexity**: The space complexity is O(k) as the algorithm stores a few large BigInteger objects (such as the primes and keys).