5.RSA ALGORITHM

```
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.util.Random;
import java.util.Scanner;

public class RSABytes {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random rnd = new
Random(System.currentTimeMillis());

        System.out.println("Enter two prime
numbers (or enter 0 0 to auto-generate
suitable primes):");
        BigInteger p = sc.nextBigInteger();
        BigInteger q = sc.nextBigInteger();

        // Auto-generate primes if user
enters 0 0
        if (p.equals(BigInteger.ZERO) &&
q.equals(BigInteger.ZERO)) {
            int bitLength = 16;
            do {
                p =
BigInteger.probablePrime(bitLength, rnd);
                q =
BigInteger.probablePrime(bitLength, rnd);
                bitLength++;
            } while
(p.multiply(q).compareTo(BigInteger.valueOf(
256)) <= 0);

            System.out.println("Auto-
generated primes:");
            System.out.println("p = " + p);
            System.out.println("q = " + q);
        }

        BigInteger n = p.multiply(q);
        BigInteger phi =
(p.subtract(BigInteger.ONE)).multiply(q.subt
ract(BigInteger.ONE));

        // Ensure modulus is large enough
        if
(n.compareTo(BigInteger.valueOf(256)) <= 0)
{
```

```
            System.out.println("Error:
modulus n = p*q is too small for byte-wise
encryption. Use larger primes.");
            sc.close();
            return;
        }

        // Choose public exponent e
        BigInteger e =
BigInteger.valueOf(65537);
        if
(!phi.gcd(e).equals(BigInteger.ONE)) {
            e = BigInteger.valueOf(3);
            while
(!phi.gcd(e).equals(BigInteger.ONE)) {
                e = e.add(BigInteger.TWO);
            }
        }

        // Compute private exponent d
        BigInteger d;
        try {
            d = e.modInverse(phi);
        } catch (ArithmeticException ex) {
            e = BigInteger.valueOf(3);
            while
(!phi.gcd(e).equals(BigInteger.ONE)) {
                e = e.add(BigInteger.TWO);
            }
            d = e.modInverse(phi);
        }

        System.out.println("\nPublic key
(e): " + e);
        System.out.println("Private key (d):
" + d);
        System.out.println("Modulus (n): " +
n);

        sc.nextLine(); // consume leftover
newline
        System.out.println("\nEnter the
message (whole line allowed):");
        String plaintext = sc.nextLine();

        // Convert to UTF-8 bytes
        byte[] plainBytes =
plaintext.getBytes(StandardCharsets.UTF_8);

        // Encrypt each byte
        BigInteger[] cipher = new
BigInteger[plainBytes.length];
        for (int i = 0; i <
plainBytes.length; i++) {
            int unsigned = plainBytes[i] &
0xFF;
            BigInteger m =
BigInteger.valueOf(unsigned);
            cipher[i] = m.modPow(e, n);
        }

        System.out.println("\nCiphertext
(space-separated integers):");
        StringBuilder cb = new
StringBuilder();
        for (BigInteger c : cipher) {
            cb.append(c.toString()).append("
");
        }

System.out.println(cb.toString().trim());

        // Decrypt back to message
        byte[] decryptedBytes = new
byte[cipher.length];
```

```
        for (int i = 0; i < cipher.length;
i++) {
            BigInteger m =
cipher[i].modPow(d, n);
            int val = m.intValue();
            decryptedBytes[i] = (byte) (val
& 0xFF);
        }

        String decrypted = new
String(decryptedBytes,
StandardCharsets.UTF_8);
        System.out.println("\nDecrypted
message: " + decrypted);

        sc.close();
    }
}
```