RSA Cryptosystem

Introduction to Cryptography

Student

Nguyễn Tấn Lộc 23127406
Nguyễn Văn Minh 23127423
Lê Thanh Phong 23127452
23CNTThuc2

Lecturers

Nguyễn Đình Thúc
Nguyễn Thị Hường
Nguyễn Văn Quang Huy

Ho Chi Minh City, November 16th, 2025

Information

Group

1. Nguyễn Tấn Lộc 23127406
2. Nguyễn Văn Minh 23127423
3. Lê Thanh Phong 23127452

Abstract

This project implements the RSA cryptosystem entirely in C++ using only the standard library. The implementation consists of three main components: prime number checking, RSA key generation, and RSA encryption and decryption.

Project Structure

```
23127406_23127423_23127452.zip
├── big_int
│   └── big_int.hpp
├── project_01_01
│   └── main.cpp
├── project_01_02
│   └── main.cpp
└── project_01_03
    └── main.cpp
```

Library

The project uses only the standard C++ libraries. Below is the complete list:

- `<cstdint>` – fixed-width integer types (`uint32_t`, `uint64_t`).
- `<cstddef>` – standard definitions.
- `<string>` – string processing.
- `<iostream>` – input/output streams.
- `<iomanip>` – formatted I/O (e.g., `std::setw`, `std::setfill`).
- `<cstring>` – memory operations (e.g., `memset`).
- `<sstream>` – string streams for converting BigInt to hex.
- `<random>` – random number generation for Miller–Rabin.
- `<fstream>` – file input/output.
- `<algorithm>` – algorithms such as `std::reverse` and `std::max`.

Contents

List of Figures

List of Tables

1 Big Integer

1.1 Overview

The `BigInt` class is a header-only library designed to handle large integers up to 2048 bits. Each number is stored using an array of 64 `uint32_t` limbs in little-endian order, where `data[0]` represents the least significant 32 bits.

This representation allows efficient arithmetic using 64-bit intermediate operations while avoiding the need for compiler-specific 128-bit types.

The class provides:

- Constructors from integers and hexadecimal strings.
- Arithmetic operators: addition, subtraction, multiplication, division, modulo.
- Bitwise operations: left shift, right shift.
- Comparison operators.
- Stream input/output and string conversion.
- Utility functions for bit manipulation and random number generation.

1.2 Implementation

1. **Data Representation** The number is represented as 64 `uint32_t` limbs, storing a total of 2048 bits. Little-endian storage simplifies carry propagation in addition/subtraction and bitwise operations, starting from the least significant limb. 64-bit arithmetic is used internally to avoid overflow.

2. **Constructors**
   - **Default constructor:** initializes all limbs to zero.
   - **64-bit integer constructor:** stores the low and high 32 bits in `data[0]` and `data[1]`.
   - **Hexadecimal string constructor:** parses a big-endian string into little-endian limb array. Used for reading input from files.

3. **Arithmetic Operations**
   - **Addition** (+, +=): adds limb-by-limb with 64-bit intermediate carry.
   - **Subtraction** (−): subtracts limb-by-limb with borrow propagation.
   - **Multiplication** (*): long multiplication across 64 limbs, using 64-bit temporary variables to handle carries.
   - **Division** (/) and **Modulo** (%, %=): implemented as bitwise long division from the most significant bit down.

4. **Bitwise Operations**
   - **Left shift** («): shifts bits across limbs and within limbs.
   - **Right shift** (»): same as left shift, in the opposite direction.

5. **Comparison Operations**

   - Equality/Inequality (==, `!=`) compares all limbs sequentially.
   - Less/Greater/Equal (<, >, <=, >=) iterate from the most significant limb to the least until a difference is found.

6. **Input/Output Operations**

   - **Output stream** («): converts limbs to a hexadecimal string, skipping leading zeros.
   - **Input stream** (»): reads a hexadecimal string and converts it to the internal representation.
   - **to_string()**: returns a hexadecimal string for debugging and logging.

7. **Utility Functions**

   - `is_zero()`: returns true if all limbs are zero.
   - `is_odd()`: returns true if the least significant bit is 1.
   - `getBit(int idx)`: retrieves a specific bit (0 or 1).
   - `getBitLen()`: returns the effective bit-length of the integer.
   - `randomBigInt(bits)`: returns a random BigInt of the specified bit-length.
   - `randomBigIntRange(low, high)`: returns a random BigInt in the range $[low, high]$.
   - `randomBase(n)`: returns a random integer in the range $[2, n-2]$, useful for primality tests.

1.3 Application

1. **Prime Number Checking**
   Large integers for Miller-Rabin or other probabilistic primality tests are handled by `BigInt`, including modular exponentiation and random base generation.

2. **Key Generation**
   RSA key generation relies on `BigInt` arithmetic for:

   - Modulus: $N = p \cdot q$
   - Euler's totient: $\phi(N) = (p-1) \cdot (q-1)$
   - Modular inverse computation for the private key d.

3. **Encryption and Decryption**
   RSA operations $C = M^e \bmod N$ and $M = C^d \bmod N$ are implemented using modular exponentiation. All multiplication, modulo, and bit-shifting operations utilize `BigInt` functions.

2 Prime Number Checking

2.1 Problem

The goal is to check whether a given integer is prime. A prime is an integer has only two positive divisors are 1 and itself.

$$\text{n is prime} \Leftrightarrow \forall d \in \mathbb{Z}^+, 1 < d < n \Rightarrow d \nmid n$$

Input: A reversed hexadecimal string, then stored in `BigInt` class.
Output:

- 1 if the number is prime.
- 0 if it is not.

2.2 Algorithm

2.2.1 Brute Force

Brute force testing:

$$\text{n is prime } \Leftrightarrow 1 < d < \sqrt{n}, \forall d \nmid n$$

This is impossible for big integer like 512-bit integer. Because $\sqrt{n} \approx 256$ bits, so the total division required is about 2^{256}. The run time will exceed the limit is 60 seconds.

2.2.2 Fermat's Little Theorem

If p is prime and a is any integer that $\gcd(a, p) = 1$, then

$$a^{p-1} \equiv 1 \pmod{p}$$

If the congruence is not correct with any base a, the number is composite.

A composite number is an integer greater than 1 that is not prime and has at least one positive divisor other than 1 and itself. In short, a composite number is a number that can be factored into smaller integers.

Although Fermat' little theorem is the fast way to reject many composite number, but it fails to detect Carmichael numbers that satisfy:

$$a^{n-1} \equiv 1 \pmod{n}, \forall a \text{ coprime with } n$$

[2]

2.2.3 Strong Probable Prime Test - Miller-Rabin

To improve accuracy, we use the Miller-Rabin primality test, based on group theory.
It can detect all Fermat liars by checking the structure:

$$a^m, a^{2m}, a^{4m}, \ldots$$

Instead of only checking a^{n-1}
Miller-Rabin breaks this exponent apart:

$$n - 1 = 2^r.m \text{ with } m \text{ odd}$$

Now, we consider the value:

$$x = a^m \bmod n$$

if n is prime, $a^{n-1} \equiv 1$, then the group $(\mathbb{Z}/n\mathbb{Z})^x$ is cyclic.
In cyclic group, equation $x^2 \equiv 1 \pmod{n}$ has exactly two solutions:

$$x = 1 \text{ and } x = n - 1$$

If n is composite, the group is not cyclic, so its value other than ± 1.
[3] [6]

2.3 Implementation

Computing the $x.y$ directly can be lead to overflow. To avoid it, we will use modulo n to reduce after each step.

Algorithm 1: Modular Multiplication `mulMod(x, y, n)`

$p \leftarrow 0$;
$x \leftarrow x \bmod n$;
$L \leftarrow$ bitLength(y) ;
for $i \leftarrow 0$ **to** $L - 1$ **do**
    if *y has bit i set* **then**
        $p \leftarrow p + x$;
        if $p \geq n$ **then**
            $p \leftarrow p - n$;
    $x \leftarrow x \ll 1$;
    if $x \geq n$ **then**
        $x \leftarrow x - n$;
return $p \bmod n$;

[6]

In this function, we use binary exponentaion that compute x^n in $O(\log(p))$ time instead of $O(\log(p)$.

It works by considering p in binary form and using the rule:

$$x^{2k} = (x^k)^2,\ x^{2k+1} = x.(x^k)^2$$

Instead of multiplying x repeatedly, we only square x at each bit and only multiply when that bit is 1.

Algorithm 2: Modular Exponentiation `powerMod(x, p, n)`

```
result ← 1 ;
L ← bitLength(p) ;
for i ← 0 to L − 1 do
    if p has bit i set then
        result ← mulMod(result, x, n) ;
    x ← mulMod(x, x, n) ;
return result ;
```

[6]

Algorithm 3: Miller–Rabin Primality Test

Input: n: integer, k: number of iterations
Result: true if n is probably prime, false if composite

```
if n < 2 or n is even then
    return false ;
m ← n − 1 ;
r ← 0 ;
while m is even do
    m ← m/2 ;
    r ← r + 1 ;
for i ← 1 to k do
    Choose random a in [2, n − 2] ;
    x ← powerMod(a, m, n) ;
    if x = 1 or x = n − 1 then
        continue ;
    composite ← true ;
    for j ← 1 to r − 1 do
        x ← mulMod(x, x, n) ;
        if x = n − 1 then
            composite ← false ;
            break ;
    if composite then
        return false ;
return true ;
```

[6]

2.4 Result

2.4.1 Test Case With Official Output File

All test case is passed. Here is the run time of each case:

Table 1: Test Case Result in Problem 1 (Test 00 - 09)

Test Case	Runtime (s)
test_00.inp	0.0954
test_01.inp	0.0165
test_02.inp	0.2138
test_03.inp	0.0589
test_04.inp	1.5655
test_05.inp	0.2207
test_06.inp	2.2298
test_07.inp	0.3449
test_08.inp	4.4752
test_09.inp	0.9797

2.4.2 Test Case Without Output File

Table 2: Test Case Result in Problem 1 (Test 10 - 19)

Test Case	Output	Runtime (s)
test_10.inp	0	0.0048
test_11.inp	1	0.2289
test_12.inp	0	0.0050
test_13.inp	1	1.0689
test_14.inp	0	0.1634
test_15.inp	1	2.0648
test_16.inp	0	0.2259
test_17.inp	1	3.2712
test_18.inp	0	0.4591
test_19.inp	1	9.6914

3 RSA Key Generation

3.1 Problem

The goal is to generate the RSA private key exponent d from the given values:

$$p, \quad q, \quad e$$

The modulus of the RSA system is:

$$N = pq$$

and Euler's totient function is:

$$\varphi(N) = (p-1)(q-1)$$

To produce a valid RSA key pair, the private exponent d must satisfy the congruence:

$$ed \equiv 1 \pmod{\varphi(N)}$$

In other words, d is the modular multiplicative inverse of e modulo $\varphi(N)$.
The output is:

- d if the inverse exists,
- -1 if e has no inverse modulo $\varphi(N)$.

3.2 Algorithm

3.2.1 Euler's Totient Function

For two distinct primes p and q:

$$\varphi(N) = (p-1)(q-1)$$

This counts the number of integers in $[1, N]$ that are coprime to N. [6]

3.2.2 Definition of Private Exponent d

RSA requires that:

$$ed \equiv 1 \pmod{\varphi(N)}$$

Equivalently:

$$d = e^{-1} \bmod \varphi(N)$$

Thus d exists if and only if:

$$\gcd(e, \varphi(N)) = 1$$

3.2.3 Bezout's Identity and Modular Inverse

Bézout's identity states that for any integers a and b, there exist integers x and y such that:

$$ax + by = \gcd(a, b)$$

- x and y are called Bezout coefficients.
- This identity guarantees that if $\gcd(a, b) = 1$, then a solution x exists such that $ax \equiv 1 \pmod b$.

In RSA:

$$ex + \varphi(N)y = 1$$

Taking modulo $\varphi(N)$ gives:

$$ex \equiv 1 \pmod{\varphi(N)}$$

Thus $x \bmod \varphi(N)$ is the private exponent d.

Bezout's identity provides the mathematical guarantee that the modular inverse exists when $\gcd(e, \varphi(N)) = 1$. [6]

3.3 Implementation

Like the modular multiplication in prime number checking to avoid overflow, computing inverse with extended Euclidean is efficient and runs in $O(\log(\varphi(N)))$

Algorithm 4: Extended Euclidean Algorithm

Input: a, b
Output: $\gcd(a, b)$ and coefficients x, y such that $ax + by = \gcd(a, b)$

if $b = 0$ **then**
  $x \leftarrow 1$, $y \leftarrow 0$;
  return a ;
Compute $\gcd(b, a \bmod b)$ recursively ;
Receive (x_1, y_1) from recursion ;
$x \leftarrow y_1$;
$y \leftarrow x_1 - (a/b) \cdot y_1$;
return $\gcd(a, b)$;

[1] [6]

Algorithm 5: Modular Inverse `modInverse(a, m)`

Compute $(\gcd(a, m), x, y)$ using Extended GCD ;
if $\gcd(a, m) \neq 1$ **then**
  return 0 `// Inverse does not exist`
return $(x \bmod m)$;

[4]

Algorithm 6: RSA Key Generation

Input: p, q, e
Output: Private exponent d

$\varphi \leftarrow (p-1)(q-1)$;
$d \leftarrow$ `modInverse`(e, φ) ;
if $d = 0$ **then**
  return -1 `// Invalid key`
return d ;

[5] [6]

3.4 Result

3.4.1 Test Case With Official Output File

All test cases passed. Here is the run time of each case:

Table 3: Test Case Result in Problem 2 (Test 00 - 09)

Test Case	Runtime (s)
test_00.inp	0.0142
test_01.inp	0.0097
test_02.inp	0.0181
test_03.inp	0.0108
test_04.inp	0.0257
test_05.inp	0.0062
test_06.inp	0.0221
test_07.inp	0.0087
test_08.inp	0.0186
test_09.inp	0.0073

3.4.2 Test Case Without Output File

Table 4: Test Case Result in Problem 2 (Test 10 - 19)

Test Case	Output	Runtime (s)
test_10.inp	-1	0.0078
test_11.inp	56F85272	0.0166
test_12.inp	-1	0.0099
test_13.inp	50FD39976E671E62920B0797D9D9983	0.0205
test_14.inp	-1	0.0081
test_15.inp	F771AD3D632AF4AA7F2F1CED5AECDAF26860 7D62A3B8C5FAF2590BF0BFE0BC2	0.0301
test_16.inp	-1	0.0084
test_17.inp	F87514A38B9F242EC24FC08CB746F676D83A3B EBCC292CD69EE6110F5EE48747D275E93101F4 CC97F3CBA57C5A4DEF7	0.0204
test_18.inp	-1	0.0101
test_19.inp	9E6CE5842781F81922EEDC370C5E9B9064756A9 4A38D77A6F8812B8A6B6B805FC76A5DF63831B B60D8DA565C5889095216B7F617FFC99FC64792 984DC1EC8BE5C560343EF046B28958098EDEFB CE3F3683031C8B8AFC566F432B9D75383662D3 A96D6CF1DBA7B5795257A63152102821597EE1 5932D723A73CDDB71E8F1D1AD3	0.0247

4 RSA Encryption and Decryption

4.1 Problem

The goal is to encrypt and decrypt messages using the RSA cryptosystem.
Inputs:

- A message M (integer) to be encrypted.
- Public key (N, e) for encryption.
- Private key (N, d) for decryption.

Outputs:

- Ciphertext $C = M^e \bmod N$ when encrypting.
- Decrypted message $M = C^d \bmod N$ when decrypting.

4.2 Algorithm

4.2.1 RSA Encryption

Given a public key (N, e) and a message M such that $0 \leq M < N$, the encryption function is:

$$C \equiv M^e \pmod N$$

This ensures that only someone with the private key can efficiently recover M. [6]

4.2.2 RSA Decryption

Given a private key (N, d) and ciphertext C, the decryption function is:

$$M \equiv C^d \pmod N$$

By the properties of modular exponentiation and Euler's theorem:

$$M^{ed} \equiv M \pmod N$$

because $ed \equiv 1 \pmod{\varphi(N)}$. [6]

4.2.3 Modular Exponentiation

Direct computation of M^e or C^d is inefficient for large numbers. We use **exponentiation by squaring** (modular exponentiation) to compute:

$$base^{exponent} \bmod modulus$$

efficiently in $O(\log exponent)$ time. [5] [6]

4.3 Implementation

Algorithm 7: RSA Encryption `RSA_encrypt(message, e, N)`

Input: $message, e, N$
Output: $ciphertext$

$ciphertext \leftarrow \texttt{powerMod}(message, e, N)$;
return $ciphertext$;

[5]

Algorithm 8: RSA Decryption `RSA_decrypt(ciphertext, d, N)`

Input: $ciphertext, d, N$
Output: $message$

$message \leftarrow \texttt{powerMod}(ciphertext, d, N)$;
return $message$;

[5]

4.4 Result

4.4.1 Test Case With Official Output File

All test cases passed. Here is the run time of each case:

Table 5: Test Case Result in Problem 3 (Test 00 - 09)

Test Case	**Runtime (s)**
test_00.inp	0.0138
test_01.inp	0.0162
test_02.inp	0.0178
test_03.inp	0.0192
test_04.inp	0.0292
test_05.inp	0.0321
test_06.inp	0.0187
test_07.inp	0.0237
test_08.inp	0.0259
test_09.inp	0.0223

4.4.2 Test Case Without Output File

Table 6: Test Case Result in Problem 3 (Test 10 - 19)

Test Case	Output	Runtime (s)
test_10.inp	9D6	0.0173
test_11.inp	66756D5	0.0286
test_12.inp	2CA114EF9B1AE252	0.0221
test_13.inp	370546E6473BB6DCAD17B93F188F091	0.0211
test_14.inp	34E1C0281C392F4C21BEF9B846D0A643330C61 C35F6E1B7	0.0221
test_15.inp	9C4E7C5A09458FFD3CB59D7E677E57964727D6 5DE372839ECB8913DB36568241	0.0249
test_16.inp	8838810FEC61705CD3755996E566A5BE28F9F4D BF4E49B90C83BDF905FBFE3E	0.0254
test_17.inp	4F4D9E7FFFA6397F6E7041178400385DB811E57 DECC8BE13EFDFE24950A6EA6C2B79080FBF8 AC72F5C5931888AD67F	0.0334
test_18.inp	BF2279282F3F050E5FADE50C79946182AC4A35 FF38F9B26F7816E707FD49635F49D7E6EC90AFE 4F6C9BA2FB5ACB26CCD4A291A4BDCED138E 71331438CBC380C1	0.0217
test_19.inp	CEBE7371E6C8D49734C54AFF446A92227249C0 C9BE4F139EA0EDB3D2A5562FC47220A331D77 C450DCCD166DF0D4DD0D3174977416C5D0702 B2EF295058FD83B28DF5D648B59B1E271A97FA 1FA32E767EC0C972769FD1820FA58EC2070C3F B3E394CEE0929C0A2A61AA8BDB25318F705512 979F71FD62F9914CB7B12D0B30EC53	0.0235

References

[1] *Euclidean algorithms (Basic and Extended)*. Accessed: 2025-11-16. URL: https://www.geeksforgeeks.org/dsa/euclidean-algorithms-basic-and-extended/.

[2] *Fermat's little theorem*. Accessed: 2025-11-16. URL: https://www.geeksforgeeks.org/dsa/fermats-little-theorem/.

[3] *Kiểm tra số nguyên tố*. Accessed: 2025-11-16. URL: https://wiki.vnoi.info/algo/algebra/primality_check.md.

[4] *Modular multiplicative inverse*. Accessed: 2025-11-16. URL: https://www.geeksforgeeks.org/dsa/multiplicative-inverse-under-modulo-m/.

[5] *RSA Algorithm in Cryptography*. Accessed: 2025-11-16. URL: https://www.geeksforgeeks.org/computer-networks/rsa-algorithm-cryptography/.

[6] *Untitled Document*. Accessed: 2025-11-16. URL: https://drive.google.com/file/d/1Wl-TRGmICj7CgkmwZ6tGRDPbdI4I1GNN/view?usp=sharing.