BigInt Library Project Report

Mallikarachchi S.D. - 210364F
CS4453

1. Introduction

In modern computing, and particularly in cryptography, there is a frequent need to work with integers much larger than the standard 32-bit or 64-bit types supported by hardware. Algorithms such as RSA and Diffie-Hellman depend on numbers that are 1024 bits, 2048 bits, or even larger, where the size of the integers directly relates to the level of security. To explore this, I have developed a custom BigInt library in C++. The goal of this project is to represent large integers efficiently and to support key modular arithmetic operations such as modular addition, multiplication, and inverse, which are fundamental to cryptographic computations.

Key features include:

- Representation of large integers using a vector array of 32-bit chunks
- Conversion between decimal and hexa-decimal formats
- Arithmetic operations; addition, subtraction and multiplication
- Modular arithmetic operations: addition, multiplication and inverse
- Utility functions for bitwise shifts, bit length calculations, and normalization

2. Design and Implementation

2.1 Number Representation

In this project, a new data type called **BigInt** is introduced to represent numbers larger than the standard 64-bit limit. The approach is to store the number inside a **vector (vector<uint32_t>)** by splitting it into 32-bit chunks. These chunks are arranged in **little-endian order**, meaning the least significant chunk is placed at index 0. To make the representation more efficient, the number is normalized by removing any leading zero chunks.

The underlying structure can be summarized as:

```
vector<uint32_t> chunks;
```

The following example demonstrates how this representation works in practice.

Consider the 128-bit number:

```
n = 0xA3F12B9C4D5E6789ABCDEF0123456789
```

0xA3F12B9C	0x4D5E6789	0xABCDEF01	0x23456789

0x23456789	0xABCDEF01	0x4D5E6789	0xA3F12B9C

Little-endian Format

In the BigInt vector array (little-endian format), the number is stored as:

```
Chunks = [0x23456789, 0xABCDEF01, 0x4D5E6789, 0xA3F12B9C]
```

```
Enter number: 0xA3F12B9C4D5E6789ABCDEF0123456789
Bit length: 128
Number of chunks: 4
chunks[0] = 0x23456789
chunks[1] = 0xABCDEF01
chunks[2] = 0x4D5E6789
chunks[3] = 0xA3F12B9C
```

Figure 1: 128-bit Number Representation

Why use 32-bit chunks instead of 64-bit chunks to represent large numbers?

Using 32-bit chunks instead of 64-bit makes the implementation avoid overflow in intermediate calculations (since additions/multiplications can safely fit into 64-bit temporary variables).

Key functions:

`BigInt(uint64_t value)`	Constructor: Initializes a BigInt from a 64-bit unsigned integer by splitting it into 32-bit chunks.
`normalize()`	Removes leading zero chunks (most significant zeros) from the BigInt
`from_hex(const string &hexString)`	Static function: Converts a hexadecimal string (with or without 0x prefix) into a BigInt.
`to_hex(const BigInt &value)`	Static function: Converts a BigInt into a hexadecimal string with 0x prefix.
`from_decimal(const string &decString)`	Static function: Converts a decimal string into a BigInt.
`to_decimal(const BigInt &value)`	Static function: Converts a BigInt into its decimal string representation.
`compare(const BigInt &a, const BigInt &b)`	Static function: Compares two BigInts; returns -1 if a < b, 0 if a == b, and 1 if a > b.

`bitLength()`	Returns the number of bits required to represent the BigInt (0 for zero).
`is_zero()`	Checks if the BigInt is zero.
`is_even()`	Static function: Checks if the BigInt is divisible by 2.
`is_one()`	Static function: Checks if the BigInt equals 1.
`shr1()`	Performs a single-bit right shift (divides the BigInt by 2).
`shlBits(size_t numBits)`	Performs a left shift by a specified number of bits (numBits) on the BigInt.

Handling integers with less than 64-bits

`BigInt(678)`

Handling integers with more than 64-bits

For very large integers, the library provides string-based constructors. It can accept either decimal or hexadecimal strings as input. These strings are then converted into the internal vector-of-32-bit-chunks representation in little-endian order.

```
static BigInt from_decimal(const string &s)
    {
        BigInt result;
        for (char c : s)
        {
            if (c < '0' || c > '9')
                continue;
            int digit = c - '0';
            result = BigInt::mul(result, BigInt(10));
            result = BigInt::add(result, BigInt(digit));
        }
        return result;
    }
```

```
BigInt::from_decimal("337190594831935655935496053709200529395")
```

```
    static BigInt from_hex(const string &hex)
    {
        BigInt result;
        string s = hex;
        if (s.size() >= 2 && s[0] == '0' && (s[1] == 'x' || s[1] ==
'X'))
```

```
            s = s.substr(2);

        reverse(s.begin(), s.end()); // process from least significant
digit
        uint64_t current = 0;
        int bits = 0;
        for (char c : s)
        {
            int val;
            if (c >= '0' && c <= '9')
                val = c - '0';
            else if (c >= 'a' && c <= 'f')
                val = c - 'a' + 10;
            else if (c >= 'A' && c <= 'F')
                val = c - 'A' + 10;
            else
                continue;

            current |= (uint64_t)val << bits;
            bits += 4;
            if (bits >= 32)
            {
                result.chunks.push_back((uint32_t)(current &
0xFFFFFFFFu));
                current >>= 32;
                bits -= 32;
            }
        }
        if (bits > 0)
            result.chunks.push_back((uint32_t)current);
        result.normalize();
        return result;
    }
```

```
BigInt::from_hex("0xA3F12B9C4D5E6789ABCDEF0123456789")
```

2.2 Arithmetic Operations

This section presents the implementation of arithmetic operations for arbitrary-precision integers (BigInt). Large numbers are represented as arrays of 32-bit chunks, and addition, subtraction, and multiplication are performed in a chunk-wise manner with proper handling of carry and borrow. These classical multi-precision algorithms form the foundation for modular arithmetic and cryptographic computations.

Function	Description	Time Complexity
`add(const BigInt &a, const BigInt &b)`	Chunk-wise addition with carry propagation	O(n)
`subtract(const BigInt &a, const BigInt &b)`	Chunk-wise subtraction with borrow handling; throws exception for negative results	O(n)
`mul(const BigInt &a, const BigInt &b)`	Grade-school multiplication with carry propagation	O(n · m)

2.3 Modular Arithmetic Operations

Modular arithmetic is a fundamental component of many cryptographic algorithms, enabling computations within a finite number system defined by a modulus m of bit length n. In this context, all operations addition, subtraction, multiplication, and reduction are performed modulo m, ensuring results always remain within the range 0 to m−1. These operations are essential for implementing secure cryptographic primitives such as RSA, Diffie-Hellman. The following section describes the key modular operations implemented in the BigInt library and their computational characteristics.

Function	Modular Arithmetic Operation	Description	Time Complexity
`mod(const BigInt &a, const BigInt &m)`	Modular Reduction $a \bmod m$	Computes by repeated subtraction/addition until the value is within [0, m-1]. Simple "naive reduction" algorithm.	$O(n^2)$
`modAdd(const BigInt &a, const BigInt &b, const BigInt &mod)`	Modular Addition $a + b \bmod m$	Computes by first performing multi-precision addition, then applying modular reduction using mod.	$O(n^2)$
`modSub(const BigInt &a, const BigInt &b, const BigInt &m)`	Modular Subtraction $a - b \bmod m$	Computes with wrap-around if a < b. Uses subtraction and optional addition to ensure result in [0, m-1].	O(n)
`modMul(const BigInt &a, const BigInt &b, const BigInt &mod)`	Modular Multiplication $a * b \bmod m$	Computes using classical "schoolbook" multiplication, then applies mod.	$O(n^2)$

`modInverse(BigInt a, const BigInt &m)`	Modular Inverse $a^{-1}\ (mod\ m)$	Computes modular inverse using Binary Extended GCD algorithm: repeatedly halves even numbers, subtracts smaller from larger, and updates coefficients.	$O(n^2)$

Binary Extended GCD Algorithm

The modulo inverse operation involves finding an integer x such that;

$$a\,.\,x\ \equiv\ 1\ (mod\ n)$$

A modulo inverse exists only if $gcd(a,\ n)\ =\ 1$. There are several algorithms available to compute the modulo inverse, including the Extended Euclidean Algorithm, the Binary Extended GCD Algorithm, and Euler's Phi Function [1]. However, in this implementation of **BigInt**, I have used **Binary Extended GCD Algorithm** [2] [3] to compute the modulo inverse of a given integer with respect to a modulus m. This approach is efficient even for very large integers, such as those with 512-bit or 1024-bit lengths.
This method is conceptually simple and efficient for large numbers. Traditional Extended Euclidean Algorithm uses division and modulo operations repeatedly, which are costly for large integers. The binary version replaces division by 2 with bit-shifts, which are extremely fast at the hardware level. All operations are addition, subtraction, and bit-shifts, avoiding the expensive general division operations that BigInt arithmetic would otherwise require.

Following is the algorithm that describes the Binary Extended GCD Algorithm used in BigInt implementation

```
function modInverseBEGCD(a, m):
    if m == 0:
        "Modulus cannot be zero"

    a = a mod m
    if a == 0:
        "Inverse does not exist"

    u = a, v = m, r = 1, s = 0
    while u != 0:
        while u is even:
            u = u / 2
            if r is odd:
                r = r + m
            r = r / 2

        while v is even:
            v = v / 2
            if s is odd:
                s = s + m
```

```
            s = s / 2

        if u >= v:
            u = u - v
            r = (r - s) mod m
        else:
            v = v - u
            s = (s - r) mod m

    if v != 1:
        "Inverse does not exist; gcd(a, m) != 1"

    return s mod m
```

2.4 User Interaction

The CLI menu provides a comprehensive interface for testing various operations on large integers, including:

1. Number Representation
2. Modular Reduction
3. Modular Addition
4. Modular Multiplication
5. Modular Inverse

Numbers can be entered in decimal or hexadecimal format
The modulus can be either manually specified or randomly generated as an n-bit number, supporting very large sizes (e.g. 512-bit, 1024-bit).

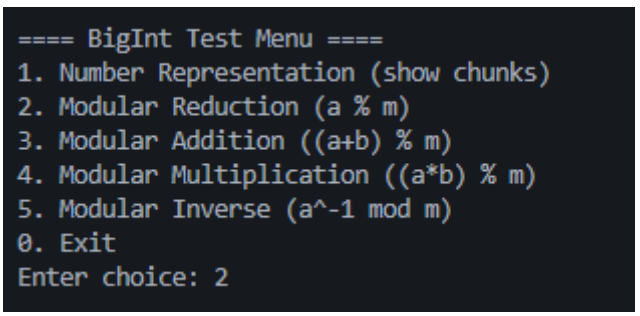


Figure 2: Test Menu

3. Testing and Results

3.1 Test Cases Overview

32-bit case

a = 4294967301
b = 4294967310
m = 8589934592

128-bit case

a = 337190594831935655935496053709200529395
b = 288756596177826426709875773173293015027
m = 175574363680769459853239029457437464990

256-bit case

a = 57896044618658097711785492504343953926634992332820282019728792003956564819968
b = 66557428065377613318815206097065113698063007866445527445797167599395468629518
m = 88270267499276418354887995034676036393148284829267936218739034982122126195791

512-bit case

a = 12504602026837434806806871092399794629700837573435919370164193815133280893713076379969817476236825624373345618801565763443321839601920800038070056366494893
b = 85134307300367665350592797992651294834388509439823617183590139031461889722933332426274905640723558802352458415435749697460655526855863464262999404956250 74
c = 72053704913568361731923055759676792265204868996930906449581450371017958309899861407652250017316683604786363808343346179893663714435528035364 48139201083028

Operation	Case	a	b	m	Expected Value	Observed Value	Pass/ Fail
Modulo Reduction	Normal case	17	-	8	1	1	Pass
	32-bit	a	-	m	4294967301	4294967301	Pass
	128-bits	a	-	m	161616231151166196082257024251763064405	161616231151166196082257024251763064405	Pass
	256-bits	a	-	m	578960446186580977117854925043439539266349923328202820197287920039565648199 68	578960446186580977117854925043439539266349923328202820197287920039565648199 68	Pass

					564819968	564819968	
	512-bits	a	-	m	52992315354805986 33614565516432115 40318035067374282 87252060487780314 85062723090239204 59247450515726389 47092379672311454 53955468158367996 50162191716541186 5	52992315354805986 33614565516432115 40318035067374282 87252060487780314 85062723090239204 59247450515726389 47092379672311454 53955468158367996 50162191716541186 5	
Modulo Addition	Normal Case	7	23	10	0	0	Pass
	a = 0	0	7	17	7	7	Pass
	b = 0	19	0	13	6	6	Pass
	32-bits	a	b	m	19	19	Pass
	128-bits	a	b	m	99224099967453703 08565473851018114 9452	99224099967453703 08565473851018114 9452	Pass
	256-bits	a	b	m	36183205184759292 67571270356673303 12315497153699978 73246786924621229 907253695	36183205184759292 67571270356673303 12315497153699978 73246786924621229 907253695	Pass
	512-bits	a	b	m	66072917741605289 95481539739729565 66009871471803209 97986069176440758 78204026437341066 85803684584478365 13186986764714972 10654649400401539 39147371845995391 1	66072917741605289 95481539739729565 66009871471803209 97986069176440758 78204026437341066 85803684584478365 13186986764714972 10654649400401539 39147371845995391 1	Pass
Modulo Multiplication	Normal case	7	5	12	11	11	Pass
	a = 0	0	9	13	0	0	Pass
	b = 0	27	0	16	0	0	Pass
	32-bits	a	b	m	6442450944	6442450944	Pass

	128-bits	a	b	m	14899772900048969446443424032390980197 5	14899772900048969446443424032390980197 5	Pass
	256-bits	a	b	m	772643715251968916042161895757908544840010879512051470677289131233048293664 74	772643715251968916042161895757908544840010879512051470677289131233048293664 74	
	512 bits	a	b	m	171699585533376588394350405614960628360562517619372634473590010671881086671653591510298555173188936861708001528474788161676094637492335753162349159648957 8	171699585533376588394350405614960628360562517619372634473590010671881086671653591510298555173188936861708001528474788161676094637492335753162349159648957 8	
Modulo Inverse	Normal case	3	-	11	4	4	Pass
	Prime Modulus	10	-	17	12	12	Pass
	a = 1	1	-	19	1	1	Pass
	a and m not coprimes (no inverse)	6	-	15	DNE	DNE	Pass
	32-bits	a	-	b	DNE	DNE	Pass
	128-bits	a	-	b	143249604396967721089630885187562460681	143249604396967721089630885187562460681	Pass

3.2 Number Representation Test

```
Enter number: 0xA3F12B9C4D5E6789ABCDEF0123456789
Bit length: 128
Number of chunks: 4
chunks[0] = 0x23456789
chunks[1] = 0xABCDEF01
chunks[2] = 0x4D5E6789
chunks[3] = 0xA3F12B9C
```

```
=== Number Representation Test ===
Enter a large number
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 2
Enter number: 0x123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
Bit length: 249
Number of chunks: 8
chunks[0] = 0x89ABCDEF
chunks[1] = 0x1234567
chunks[2] = 0x89ABCDEF
chunks[3] = 0x1234567
chunks[4] = 0x89ABCDEF
chunks[5] = 0x1234567
chunks[6] = 0x89ABCDEF
chunks[7] = 0x1234567
```

```
=== Number Representation Test ===
Enter a large number
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 125046020268374348068068710923997946297008375734359193701641938151332808937130763799698174762368256243733456188015657634433218396019208000380700566
366494893
Bit length: 512
Number of chunks: 16
chunks[0] = 0x72CDD4AD
chunks[1] = 0xE29990BA
chunks[2] = 0xB6CCC2C
chunks[3] = 0x797C7465
chunks[4] = 0x13F7BB80
chunks[5] = 0x423732E0
chunks[6] = 0x22FB34A7
chunks[7] = 0xB28238C5
chunks[8] = 0xABDF4BD4
chunks[9] = 0xC7D4ADBD
chunks[10] = 0xBC7967D
chunks[11] = 0xAB74A879
chunks[12] = 0xCCE047E7
chunks[13] = 0x69B672EA
chunks[14] = 0xC9AB5C2C
chunks[15] = 0xEEC138B4
```

3.3 Modular Reduction Test

```
=== Modular Reduction Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 17
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 8
a % m = Hex: 0x1
Decimal: 1
```

```
=== Modular Reduction Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 4294967301
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 8589934592
a % m = Hex: 0x100000005
Decimal: 4294967301
```

```
=== Modular Reduction Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 57896044618658097711785492504343953926634992332820282019728792003956564819968
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 882702674992764183548879950346760363931482848292679362187390349821221261957 91
a % m = Hex: 0x8000000000000000000000000000000000000000000000000000000000000000
Decimal: 57896044618658097711785492504343953926634992332820282019728792003956564819968
```

```
=== Modular Reduction Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 125046020268374348068068710923997946297008375734359193701641938151332808937130763799698174762368256243733456188015657634433218396019208000380700563664948 93
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 720537049135683617319230557596767922652048689969309064495814503710179583098986140765225001731668360478636380834334617989366371443552803536448139201083028
a % m = Hex: 0x652E1AABF18968BC5BFB2D4340C8E4D727682CA72EE742052505E0BEEEC7F6C6FA115353D4BBEA7696E69B071C6BE7A645600537B8CC6657B6D91D3B7EA1DE19
Decimal: 52992315354805986336145655164321154031803506737428287252060487780314850627230902392045924745051572638947092379672311454539554681583679965016219171654118 65
```

```
=== Modular Reduction Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 125046020268374348068068710923997946297008375734359193701641938151332808937130763799698174762368256243733456188015657634433218396019208000380700563664948 93
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 720537049135683617319230557596767922652048689969309064495814503710179583098986140765225001731668360478636380834334617989366371443552803536448139201083028
a % m = Hex: 0x652E1AABF18968BC5BFB2D4340C8E4D727682CA72EE742052505E0BEEEC7F6C6FA115353D4BBEA7696E69B071C6BE7A645600537B8CC6657B6D91D3B7EA1DE19
Decimal: 52992315354805986336145655164321154031803506737428287252060487780314850627230902392045924745051572638947092379672311454539554681583679965016219171654118 65
```

3.4 Modular Addition Test

```
=== Modular Addition Test ===
a = 7
b = 23
m = 10
(a + b) % m = Decimal: 0
Hex:     0x0
```

```
=== Modular Addition Test ===
a = 5
b = 3
m = 32
(a + b) % m = Decimal: 8
Hex:     0x8
```

```
=== Modular Addition Test ===
a = 4294967301
b = 4294967310
m = 8589934592
(a + b) % m = Decimal: 19
Hex:     0x13
```

```
=== Modular Addition Test ===
a = 0
b = 7
m = 17
(a + b) % m = Decimal: 7
Hex:     0x7
```

```
=== Modular Addition Test ===
a = 19
b = 0
m = 13
(a + b) % m = Decimal: 6
Hex:     0x6
```

```
=== Modular Addition Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 337190594831935655935496053709200529395
Enter number b
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 288756596177826426709875773173293015027
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 1755743636807694598532390294574374649 90

=== Modular Addition Test ===
(a + b) % m = Hex: 0x4AA5DDD51EFD351E79B2294EAAB1AB0C
Decimal: 992240999674537030856547385101811494 52
```

```
=== Modular Addition Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 57896044618658097711785492504343953926634992332820282019728792003956564819968
Enter number b
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 665574280653776133188152060970651136980630078664455274457971675993954686295 18
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 882702674992764183548879950346760363931482848292679362187390349821221261957 91

=== Modular Addition Test ===
(a + b) % m = Hex: 0x4FFEF7E83A9BEB4C1182C05C89E06A01E49E4126612549D3175537C9C83019BF
Decimal: 361832051847592926757127035667330312315497153699978732467869246212299072536 95
```

```
=== Modular Addition Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 125046020268374348068068710923997946297008375734359193701641938151332808937130763799698174762368256243733456188015657634433218396019208000380700563664948 93
Enter number b
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 851343073003676653505927979926512948343885094398236171835901390314618897229333324262749056407235588023524584154357496974606555268558634642629994049562507 4
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 720537049135683617319230557596767922652048689969309064495814503710179583098986140765225001731668360478636380834334617989366371443552803536448139201083028

=== Modular Addition Test ===
(a + b) % m = Hex: 0x7E27C4FA8EEA6EDA5926CB497F31D83A891166BB4E4DEBADE811E1D48C2ACC5322EBBB52F7916499574FF40FA32B5E867E9FA10BA4784521290 0B66037E0F2F7
Decimal: 660729177416052899548153973972956566009871471803209979860691764407587820402643734106685803684584478365131869867647149721065464940040153939147371845995391 1
```

3.5 Modular Multiplication Test

```
=== Modular Multiplication Test ===
a = 27
b = 0
m = 16
(a * b) % m = Decimal: 0
Hex:     0x0
```

```
=== Modular Multiplication Test ===
a = 7
b = 5
m = 12
(a * b) % m = Decimal: 11
Hex:     0xB
```

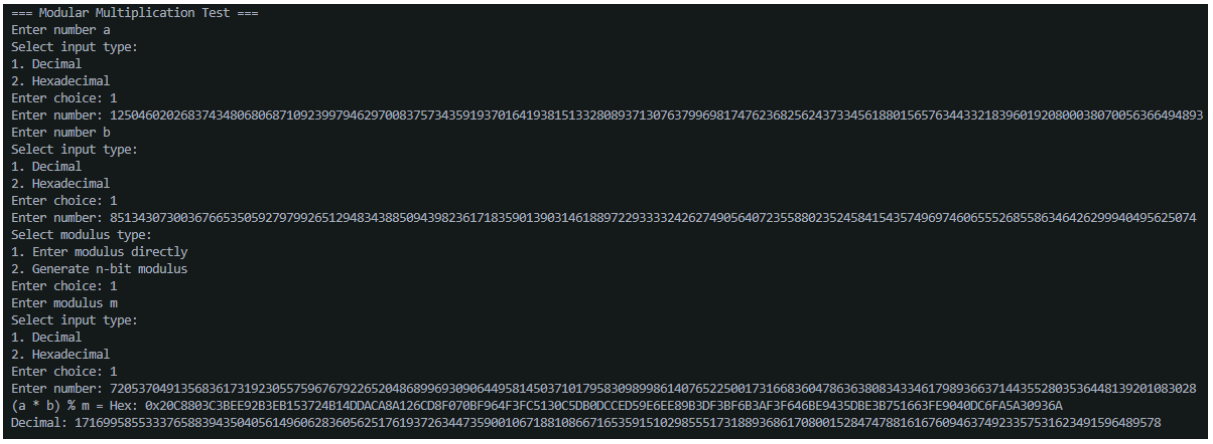
```
=== Modular Multiplication Test ===
a = 3
b = 4
m = 20
(a * b) % m = Decimal: 12
Hex:     0xC
```

```
=== Modular Multiplication Test ===
a = 0
b = 9
m = 13
(a * b) % m = Decimal: 0
Hex:     0x0
```

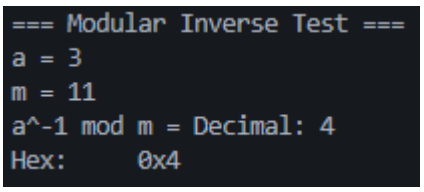
```
=== Modular Multiplication Test ===
a = 4294967296
b = 4294967297
m = 8589934591
(a * b) % m = Decimal: 6442450944
Hex:     0x180000000
```

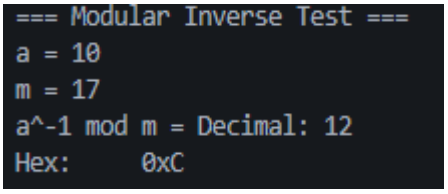
```
=== Modular Multiplication Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 337190594831935655935496053709200529395
Enter number b
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 288756596177826426709875773173293015027
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 1755743636807694598532390294574374649900

=== Modular Multiplication Test ===
(a * b) % m = Hex: 0x7017EB3600D314D66FCD6C79D1CB57F7
Decimal: 14899772900048969446443424032390980197 5
```

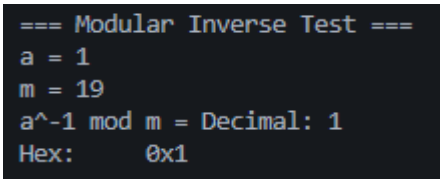
```
=== Modular Multiplication Test ===
Enter number a
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 57896044618658097711785492504343953926634992332820282019728792003956564819968
Enter number b
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 665574280653776133188152060970651136980630078664455274457971675993954686295 18
Select modulus type:
1. Enter modulus directly
2. Generate n-bit modulus
Enter choice: 1
Enter modulus m
Select input type:
1. Decimal
2. Hexadecimal
Enter choice: 1
Enter number: 882702674992764183548879950346760363931482848292679362187390349821221261957 91
(a * b) % m = Hex: 0xAAD215A4E70FB9F7DFB39937E3A0CA1B09C1C2CAD5004F95412ABA5F0FBC30CA
Decimal: 772643715251968916042161895757908544840010879512051470677289131233048293664 74
```

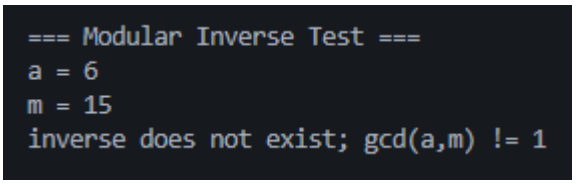


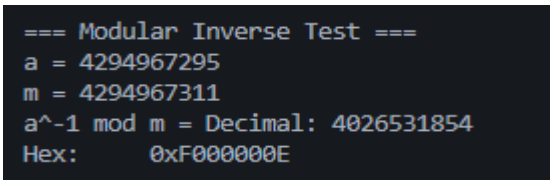
3.6 Modular Inverse Test

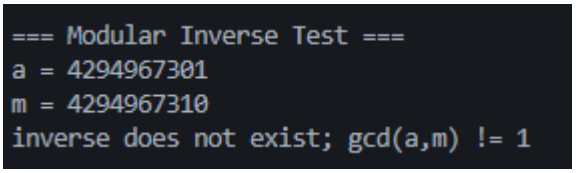


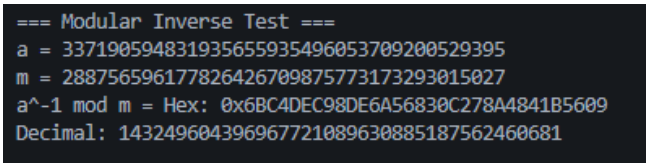












4. Discussion

From the results, it is clear that chunk count directly affects runtime: the higher the count, the longer it takes to complete a single operation. This is due to the increased number of iterations and carry operations required. However, the runtime for modular inverse tests with higher bit lengths indicates that the algorithm is still not efficient for integers exceeding 128 bits. Edge cases such as zero, one, non-invertible moduli are all handled cleanly, ensuring stable behavior.

Multiplication and modular reduction are implemented with basic algorithms, which are sufficient for smaller inputs. However, for extremely large numbers, multiplication operations become a significant bottleneck, while reduction and addition operations maintain relatively good performance. In all tested cases, returned values are proved to be accurate. Repeated-subtraction for modulus and standard long multiplication scale poorly for numbers with hundreds or thousands of bits, resulting in long runtimes. Each BigInt creates temporary objects during operations, increasing memory consumption for large numbers.

Overall, while the current implementation is robust for moderate-sized inputs and demonstrates correct functionality, for extremely large numbers, more advanced techniques such as Karatsuba multiplication or Montgomery reduction could provide substantial performance gains.

5. Conclusion

This project successfully implements a BigInt library supporting large integer representation and core modular arithmetic operations, including addition, multiplication, reduction, and modular inversion. The implementation handles edge cases such as zero, one, and non-invertible elements, ensuring robust and correct behavior. The use of a binary extended GCD algorithm for modular inversion demonstrates efficiency gains compared to naive approaches, particularly for very large numbers.

The library can be extended to support more advanced operations such as modular exponentiation and optimized algorithms for cryptographic applications. Another promising direction is to improve computational efficiency, ensuring that operations not only remain correct but also become significantly faster and less resource-intensive.

6. References

[1] Bufalo, M.; Bufalo, D.; Orlando, G. A Note on the Computation of the Modular Inverse for Cryptography. Axioms 2021, 10, 116. https://doi.org/10.3390/axioms10020116

[2] Pornin, T. (2020). Optimized binary GCD for modular inversion. IACR Cryptology ePrint Archive, 2020/972. https://eprint.iacr.org/2020/972

[3] GeeksforGeeks. (2025, February 12). Stein's algorithm for finding GCD. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/steins-algorithm-for-finding-gcd/

[4] C++ Standard Library Documentation, https://en.cppreference.com/w/

[5] Szyk, B., & Pamuła, H. (Creators), Czernia, D., Bowater, J., & Kuca, B. (Reviewers). (2025, August 16). Modulo calculator. Omni Calculator. https://www.omnicalculator.com/math/modulo