CS4453 - Information Security and Cryptography

Implementation of BigNumber Library

200727M - Udara Malinda Wijesinghe

---

1. Introduction

Public key cryptosystems rely on computations using extremely big numbers, typically ranging from 512 to 2048 bits. These processes necessitate specialist handling to ensure efficiency and precision. Therefore most of these operations are handled by the libraries that are implemented to various programming languages.
This report describes the design and implementation of a BigNum library that enables large integer arithmetic, such as

- Modulo Addition
- Modulo Multiplication
- Modular inversion.

The library is written in C++ and is intended to handle numbers with different bit lengths, making it adaptable to various cryptographic needs. Key design decisions, including number representation and modular arithmetic operations, are briefly described together with test results and performance analysis, which demonstrate the library's capacity to process huge quantities efficiently and accurately.

2. Design

2.1. Big Number Representation

Create a **BigNumber** object with two attributes. The first attribute, **number_digits**, stores the digits of the given integer as characters in reversed order. The second attribute, **isNegative**, stores the sign of the given big number. This is how we represent the given **BigNumber**.

```
private:
    // Vector to store the digits of the number in reverse order (least
significant digit first)
    std::vector<int> number_digits;
    bool isNegative = false;
```

2.2. Basic Comparison Operators

2.2.1. “ < ” - Operator

It first checks if their signs differ, returning **True** if this number is negative. Then, it compares the sizes of their digit vectors, returning based on size and sign. Finally, it compares corresponding digits from the most significant to the least significant, returning based on sign and digit value.

```
bool operator<(const BigNumber &other_number) const{
        if (isNegative != other_number.isNegative){
            return isNegative;
        }
        if (number_digits.size() != other_number.number_digits.size()){
            return (isNegative ? number_digits.size() >
other_number.number_digits.size() : number_digits.size() <
other_number.number_digits.size());
        }

        for (int i = number_digits.size() - 1; i >= 0; i--){
            if (number_digits[i] != other_number.number_digits[i]){
                return (isNegative ? number_digits[i] >
other_number.number_digits[i] : number_digits[i] <
other_number.number_digits[i]);
            }
        }
        return false;
    }
```

2.2.2. “ > ” - Operator

This **>** function checks if the current **BigNumber** is greater than **other_number** by reversing the logic of the **<** function. It returns **True** if **other_number** is less than the current object.

```
bool operator>(const BigNumber &other_number) const{
        return other_number < *this;
    }
```

2.2.3. “ <= ” - Operator

```
    bool operator<=(const BigNumber &other_number) const{
        return !(*this > other_number);
    }
```

2.2.4. “ >= ” - Operator

```
    bool operator>=(const BigNumber &other_number) const{
        return !(*this < other_number);
    }
```

2.2.5. “ == ” - Operator

```
    bool operator==(const BigNumber &other_number) const{
        return number_digits == other_number.number_digits && isNegative == 
other_number.isNegative;
    }
```

2.2.6. “ != ” - Operator

```
    bool operator!=(const BigNumber &other_number) const{
        return !(*this == other_number);
    }
```

2.3. Basic Arithmetic Operations

2.3.1. “ + ” - Addition

The overloaded **operator+** for **BigNumber** adds two numbers by handling both their absolute values and signs. If the two numbers have the same sign, it calls the **addAbsoluteValues** function, which sums the corresponding digits from both numbers, taking care of the carry, and the result inherits the common sign. If the numbers have different signs, it compares their absolute values to determine which one is larger. The function then subtracts the smaller absolute value from the larger using **subtractAbsoluteValues**, ensuring that the result carries the sign of the larger absolute value. Both helper functions ensure that the resulting **BigNumber** is properly formatted by removing any leading zeros before returning it.

```
// Helper function to add absolute values of two BigNumbers
    BigNumber addAbsoluteValues(const BigNumber &a, const BigNumber &b) 
const{
        BigNumber result;
        result.isNegative = false;
        int carry = 0;
        size_t max_size = std::max(a.number_digits.size(), 
b.number_digits.size());
        result.number_digits.clear(); // Clear the result vector

        for (size_t i = 0; i < max_size || carry != 0; ++i){
            int digit_sum = carry;
            if (i < a.number_digits.size())
                digit_sum += a.number_digits[i];
            if (i < b.number_digits.size())
                digit_sum += b.number_digits[i];
            result.number_digits.push_back(digit_sum % 10);
            carry = digit_sum / 10;
        }
        result.trimLeadingZeros();
        return result;
    }
```

```
    // Helper function to subtract absolute values of two BigNumbers (a - b)
assuming |a| >= |b|
    BigNumber subtractAbsoluteValues(const BigNumber &a, const BigNumber &b)
const{
        BigNumber result;
        result.isNegative = false;
        int borrow = 0;
        result.number_digits.clear(); // Clear the result vector
        for (size_t i = 0; i < a.number_digits.size(); ++i){
            int digit_diff = a.number_digits[i] - borrow;
            if (i < b.number_digits.size()){digit_diff -=
b.number_digits[i];}
            if (digit_diff < 0){
                digit_diff += 10;
                borrow = 1;
            }
            else{ borrow = 0;}
            result.number_digits.push_back(digit_diff);
        }
        result.trimLeadingZeros();
        return result;
    }
    // Overload the + operator for BigNumber
    BigNumber operator+(const BigNumber &other_number) const{
        if (isNegative == other_number.isNegative){
            // If both numbers have the same sign, add their absolute values
            BigNumber result = addAbsoluteValues(*this, other_number);
            result.isNegative = isNegative; // The result will have the same
sign as both numbers
            return result;
        }
        else{
            // If the signs are different, subtract the smaller absolute value
from the larger
            if (absolute() >= other_number.absolute()){
                BigNumber result = subtractAbsoluteValues(*this,
other_number);
                result.isNegative = isNegative; // Keep the sign of the
larger absolute value
                return result;
            }
            else{
                BigNumber result = subtractAbsoluteValues(other_number,
*this);
                result.isNegative = other_number.isNegative; // Keep the sign
of the larger absolute value
                return result;
```

```
            }
        }
    }
```

2.3.2. “ - ” - Subtraction

The **operator-** function performs subtraction by negating the sign of **other_number** and then adding it to ***this**. It creates a copy of **other_number**, flips its sign, and uses the previously defined **operator+** to compute the result, effectively turning subtraction into addition.

```
 BigNumber operator-(const BigNumber &other_number) const{
        // Subtraction is addition with the opposite sign
        BigNumber negated_other = other_number;
        negated_other.isNegative = !other_number.isNegative; // Flip the sign
        return *this + negated_other;
    }
```

2.3.3. “ * ” - Multiplication

The **operator*** function multiplies two **BigNumber** objects by performing digit-by-digit multiplication, similar to how multiplication is done manually. It first initializes the **result** with enough space to hold the maximum possible number of digits and determines the sign of the result based on whether the input numbers have different signs. The function then iterates through each digit of the first number, multiplying it by each digit of the second number. The partial products are added to the appropriate position in the **result** vector, managing carry values as needed. After the multiplication, leading zeros are trimmed to ensure the result is correctly formatted before returning it.

```
    BigNumber operator*(const BigNumber &other_number) const{
        BigNumber result;
        result.number_digits.resize(number_digits.size() +
other_number.number_digits.size(), 0); // Clear the result vector
        result.isNegative = (isNegative != other_number.isNegative); // The
sign of the result is negative if the signs are different
        // Multiply the absolute values of the two numbers
        for (size_t i = 0; i < number_digits.size(); ++i){
            for (size_t j = 0; j < other_number.number_digits.size(); ++j){
                int product = number_digits[i] *
other_number.number_digits[j];
                int position = i + j;
                while (product > 0){
                    if (position == result.number_digits.size()){
                        result.number_digits.push_back(0);
                    }
                    product += result.number_digits[position];
                    result.number_digits[position] = product % 10;
                    product /= 10;
                    position++;
                }
```

```
            }
        }
        result.trimLeadingZeros();
        return result;
    }
```

2.3.4. “ / ” & “ % ” - Division & Modulus

The **operator/** and **operator%** functions are used to perform division and modulus operations on two **BigNumber** objects. Both operations rely on the **divide** helper function, which performs the actual division. The **operator/** returns the quotient by extracting the first element from the pair returned by **divide**, while the **operator%** returns the remainder by extracting the second element of the pair.

The **divide** function implements long division by iterating through each digit of the dividend, starting from the most significant. It builds the **remainder** by adding the current digit and then determines the largest possible digit for the quotient using binary search. This digit is found by repeatedly multiplying the divisor by midpoints and checking if the product fits within the current **remainder**. The quotient digit is stored, and the corresponding product is subtracted from the **remainder**. This process continues until all digits have been processed, resulting in the final quotient and remainder, which are then trimmed of any leading zeros and returned as a pair.

```
    BigNumber operator%(const BigNumber &other_number) const{
        // Perform division and return the remainder
        return divide(other_number).second;
    }

    BigNumber operator/(const BigNumber &other_number) const{
        // Perform division and return the quotient
        return divide(other_number).first;
    }

    std::pair<BigNumber, BigNumber> divide(const BigNumber &other_number)
const {
        if (other_number == BigNumber("0")){
            std::cout << "Does not exist" << std::endl;
            return {BigNumber("0"), BigNumber("0")};
        }
        BigNumber result, remainder;
        result.number_digits.resize(number_digits.size(), 0);
        result.isNegative = isNegative != other_number.isNegative;
        remainder.isNegative = isNegative;
        for (int i = number_digits.size() - 1; i >= 0; --i){
            remainder.number_digits.insert(remainder.number_digits.begin(),
number_digits[i]);
            remainder.trimLeadingZeros();
```

```
            int quotientDigit = 0;
            int left = 0, right = 10;
            while (left <= right){
                int mid = (left + right) / 2;
                BigNumber candidate = other_number.absolute() *
BigNumber(mid);
                if (candidate <= remainder.absolute()){
                    quotientDigit = mid;
                    left = mid + 1;
                }
                else{
                    right = mid - 1;
                }
            }
            BigNumber product = other_number.absolute() *
BigNumber(quotientDigit);
            remainder = remainder - product;
            result.number_digits.insert(result.number_digits.begin(),
quotientDigit);
        }
        result.trimLeadingZeros();
        remainder.trimLeadingZeros();
        return {result, remainder};
    }
```

2.4. Modulo Arithmetics

2.4.1. Modulo Addition

The **modAddition** function performs modular addition of two **BigNumber** objects. It first adds the current **BigNumber** (***this**) to **other_number** using the overloaded + operator. Then, it calculates the remainder of this sum when divided by **modulus** using the overloaded **%** operator. The function returns this remainder, effectively giving the result of the addition modulo the given **modulus**.

```
BigNumber modAddition(const BigNumber &other_number, const BigNumber
&modulus){
        BigNumber result = *this + other_number;
        return result % modulus;
    }
```

2.4.2. Modulo Multiplication

The **modMultiplication** function performs modular multiplication of two **BigNumber** objects. It first multiplies the current **BigNumber** (***this**) by **other_number** using the overloaded ***** operator. Then, it calculates the remainder of this product when divided by the given **modulus** using the overloaded **%** operator. The function returns this remainder, giving the result of the multiplication modulo the specified **modulus**.

```
    BigNumber modMultiplication(const BigNumber &other_number, const
BigNumber &modulus){
        BigNumber result = *this * other_number;
        return result % modulus;
    }
```

2.4.3. Modulo Inverse

The **modInverse** function calculates the modular inverse of a **BigNumber** relative to a given modulus using the Extended Euclidean Algorithm. This modular inverse is a value **x** such that **(a * x) % m = 1**. The function initializes the variables needed for the algorithm, including the original number **a**, the modulus **m**, and two coefficients **x0** and **x1** used to compute the inverse. It then iteratively applies the algorithm, dividing **a** by **m** to get a quotient **q** and updating **a** and **m** with the remainder and previous value of **m**, while also adjusting the coefficients **x0** and **x1**. Once **a** becomes 1, **x1** holds the modular inverse. If **x1** is negative, it's adjusted to be positive by adding the original modulus **m0**. The function then returns **x1** as the modular inverse.

```
BigNumber modInverse(const BigNumber &modulus){
        BigNumber a = *this;
        BigNumber m = modulus;
        BigNumber m0 = m;
        BigNumber t, q;
        BigNumber x0 = 0, x1 = 1;
        if (m == 1)
            return 0;
        // Apply extended Euclid Algorithm
        while (a > 1){
            // q is quotient
            q = a / m;
            t = m;
            // m is remainder now, process same as Euclid's algo
            m = a % m, a = t;
            t = x0;
            x0 = x1 - q * x0;
            x1 = t;
        }
        // Make x1 positive
        if (x1 < 0)
            x1 = x1 + m0;
        return x1;
    }
```

3. Testing

All test cases are verified using Python's built-in functions. Each test case has an expected value, obtained by performing the operations in Python. Below are the CLI outputs for each test case.

3.1. Modulo Addition Testing

Modulo addition with 0

```
--------------------------------------------------------------------------------
Number 1: 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
Number 2: 0
modulus : 512bits
Calculated    : 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
Expected      : 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
--------------------------------------------------------------------------------
```

Modulo addition with large numbers

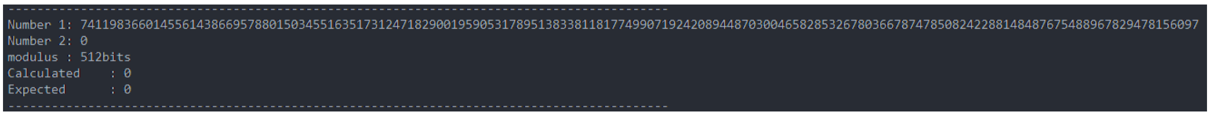
```
--------------------------------------------------------------------------------
Number 1: 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
Number 2: 6792552932193534197244105339257986521498374343822704414426523273384151396718117732757475780861305360793165960264029209773235657982871742356434819199604765
modulus : 512bits
Calculated    : 7967286623964985363396591425555955575363209950593129957561407810457685484195614751976909313974065913884608824443306374277240633194128478989689996716767 66
Expected      : 796728662396498536339659142555595557536320995059312995756140781045768548419561475197690931397406591388460882444330637427724063319412847898968999671676766
--------------------------------------------------------------------------------
Number 1: 217199590935431221932715973459130927656623637589207702843001057821112691360916347230773299962295247392931889737269606301978052130089057558553837603811550292284888182527884619057004834609051976452984932647140835987387900144870015686916513265788670879733232427698749023445685228474430064229228577406953235119 33
Number 2: 875982500360463283615739795058975797874567090991747491484232548410953461660727163243736934012898060403499259333621244104784226433003384633474900891959126821105189805664510363608528773583965402845684474040699289352653279934073705768846627319328977968490682041955889408526519114262961356821646414913665773600 37
modulus : 1024bits
Calculated    : 109318209129589450554845576851810672553119072858095519432723360623206615302164351047451023397519330779643114907089085040676227856309244219202873849577067711339007798819239498266553360819301737929866940668784012534004118007894372145576314058511764884822391446965463843197220434273739142105087499232061900871970
Expected      : 109318209129589450554845576851810672553119072858095519432723360623206615302164351047451023397519330779643114907089085040676227856309244219202873849577067711339007798819239498266553360819301737929866940668784012534004118007894372145576314058511764884822391446965463843197220434273739142105087499232061900871970
--------------------------------------------------------------------------------
Number 1: 633646102445455593211703462042626940560320505382933261977550131267845191230233698111049196951012049643784817430814393694230215244175519531222130232692829667142562287096243628322253743899549599421211257846966019294993256428068066417781276049853962701429928570870771954437930885558701839696762703921883633412396697381872499363617508492756804994177999242261122828643577638488204378478859422055335599260768603014856307389463465630736761272962059723821222072844562807769639314660968962521572057286736033417787906771795017931255354644165072457872540416933043098134687329554138452327854826747457235297472333494565011500043
Number 2: 844962256499951097956753476374301624549093829951355347636762354808731446382591672684035873248239264408989303465968219134818719614108168871120244501087999353933848478293415952278147145162527814674265305669403543228227681607830019779225445605575969949394335665743480939166764264382055813223795291812482309713276855923623293747999091630923196923343680165916581280884904350107313795245032898273282414976598446450563297865606672421812175695637184448491972338445789657983242753594665481837447062623221765443679911270802201905111529341256157452230300228196193735105867093497639341409501526727912570617448607635275286729132 4
modulus : 2048bits
Calculated    : 147860835894540669116845693841692856510941433533428860961431248607657663761282537079508507019925131405277412089678261282904893485828368840234237473378082902107641076538965958060040088906207741409547656351636956252322093803589808619700672165542993265082426423661425289360469514994075765292055799573436594312567355330549579311161660012368000191752167940817770410952848198855951817372389232032861801423736704946541960525507013805254893696859924417231319441129035246575288206825563444435901911990995779886146781804259721983636688398542122991010284064512923683324055442305177779373735635347536980591492094112984029787913 67
Expected      : 147860835894540669116845693841692856510941433533428860961431248607657663761282537079508507019925131405277412089678261282904893485828368840234237473378082902107641076538965958060040088906207741409547656351636956252322093803589808619700672165542993265082426423661425289360469514994075765292055799573436594312567355330549579311161660012368000191752167940817770410952848198855951817372389232032861801423736704946541960525507013805254893696859924417231319441129035246575288206825563444435901911990995779886146781804259721983636688398542122991010284064512923683324055442305177779373735635347536980591492094112984029787913 67
--------------------------------------------------------------------------------
```

Modulo addition with negative numbers

```
--------------------------------------------------------------------------------
Number 1: -217199590935431221932715973459130927656623637589207702843001057821112691360916347230773299962295247392931889737269606301978052130089057558553837603811550292284888182527884619057004834609051976452984932647140835987387900144870015686916513265788670879733232427698749023445685228474430064229228577406953235119 33
Number 2: 875982500360463283615739795058975797874567090991747491484232548410953461660727163243736934012898060403499259333621244104784226433003384633474900891959126821105189805664510363608528773583965402845684474040699289352653279934073705768846627319328977968490682041955889408526519114262961356821646414913665773600 37
modulus : 1024bits
Calculated    : 658782909425032061683023821599844870217943453402539788641231490589840770299810816012963634050602813010567369596351637802806174302914327074921063288147576528820301623136625744551523938974913426392699541393558453365265379789203690081930114053540307088757449614257140385000833885788531292592417837506712538481 04
Expected      : 658782909425032061683023821599844870217943453402539788641231490589840770299810816012963634050602813010567369596351637802806174302914327074921063288147576528820301623136625744551523938974913426392699541393558453365265379789203690081930114053540307088757449614257140385000833885788531292592417837506712538481 04
--------------------------------------------------------------------------------
Number 1: -633646102445455593211703462042626940560320505382933261977550131267845191230233698111049196951012049643784817430814393694230215244175519531222130232692829667142562287096243628322253743899549599421211257846966019294993256428068066417781276049853962701429928570870771954437930885558701839696762703921883633412396697381872499363617508492756804994177999242261122828643577638488204378478859422055335599260768603014856307389463465630736761272962059723821222072844562807769639314660968962521572057286736033417787906771795017931255354644165072457872540416933043098134687329554138452327854826747457235297472333494565011500043
Number 2: 844962256499951097956753476374301624549093829951355347636762354808731446382591672684035873248239264408989303465968219134818719614108168871120244501087999353933848478293415952278147145162527814674265305669403543228227681607830019779225445605575969949394335665743480939166764264382055813223795291812482309713276855923623293747999091630923196923343680165916581280884904350107313795245032898273282414976598446450563297865606672421812175695637184448491972338445789657983242753594665481837447062623221765443679911270802201905111529341256157452230300228196193735105867093497639341409501526727912570617448607635275286729132 4
modulus : 2048bits
Calculated    : 211316154054495504745050014331674668398877332456842208565921222354088625551523579745729866762972272147652044860351538254405885043699326493398981142683951696867912861911971723239558934012629782152530540478224375239332344251797619533614441695557220072479644070948727089847288337882335397352703258789059867630088015854175079438438158313816639192916568092365545845224132671161910941676617347621794681571582984343570699047614320679107541442267512472467075026560122685021360343893369651931587500533648573202589200449900718397385617469709108499435775981126315063697117976394350088908164669998045533531997627414071027557912 81
Expected      : 211316154054495504745050014331674668398877332456842208565921222354088625551523579745729866762972272147652044860351538254405885043699326493398981142683951696867912861911971723239558934012629782152530540478224375239332344251797619533614441695557220072479644070948727089847288337882335397352703258789059867630088015854175079438438158313816639192916568092365545845224132671161910941676617347621794681571582984343570699047614320679107541442267512472467075026560122685021360343893369651931587500533648573202589200449900718397385617469709108499435775981126315063697117976394350088908164669998045533531997627414071027557912 81
================================================================================
--------------------------------------------------------------------------------
```

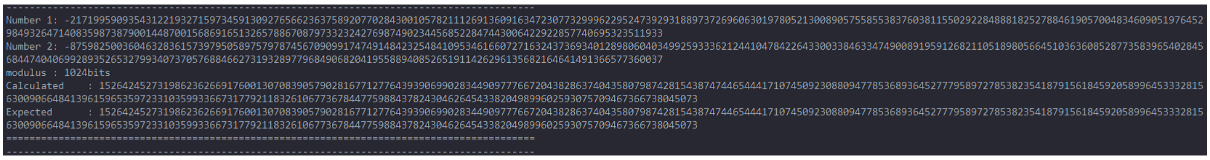
3.2. Modulo Multiplication Testing

Modulo multiplication with 0



Modulo multiplication with large numbers

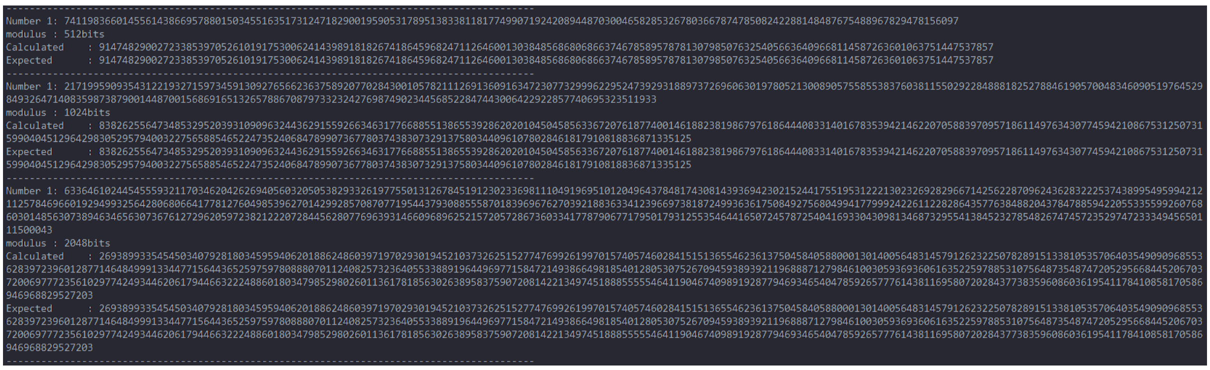
Modulo multiplication with negative numbers



3.3. Modulo Inverse Testing

Modulo inverse with 1

Modulo inverse with large numbers



Modulo inverse with modulo inverse does not exist

```
----------------------------------------------------------------------------------------
Number 1: 6792552932193534197244105339257986521498374343822704414426523273384151396718117732757475780861305360793165960264029209773235657982871742356434819199604762
modulus : 512bits
Does not exist
Does not exist
```

4. Conclusion

This **BigNumber** library provides a comprehensive set of operations for handling large integers with support for arithmetic operations including addition, subtraction, multiplication, and division, as well as modular arithmetic. It includes advanced features such as modular inverse computation and modular operations which are essential for cryptographic applications and other complex mathematical computations. The library efficiently handles large numbers by implementing essential algorithms such as the Extended Euclidean Algorithm for modular inverses and long division for arithmetic operations. By supporting these operations, the library facilitates precise and scalable mathematical computations necessary for a variety of applications, from basic arithmetic to advanced cryptographic algorithms.