

CS4453 - Information Security and Cryptography

Implementation of BigInteger 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 BigInteger 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 **BigInteger** 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 **BigInteger**.

```
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: 67925529321993341972441053392579865124983743438227044144265232738415139671811773275475788861305360793165960264029209773235657982871742356434819199604765
modulus : 512bits
Calculated : 796728662396498536396591425555595575363209506593129957561407810457685484195614751976899313974065913884608824443386374772406331941284789896899671676766
Expected : 796728662396498536396591425555595575363209506593129957561407810457685484195614751976899313974065913884608824443386374772406331941284789896899671676766
-----
Number 1: 211719599954312219327159734591309276566236375892077028430010578211126913609163472307732996622952473929318897372696063019780521300890575585538376038115502922848881825278846190570048346090519764529
84933264714083598738790014487001568691651326578867087973323242769874902344568522847443006422922857740695323511933
Number 2: 87598250036046328361573979508975797874567090991747491484232548410953461660727163243736934012890860403499259336212441047842264330033846334749008919591268211051890805664510363608528773389654028456
847404069298932532799340737057688466273193289779684968204195588940852651911426296135682164641491365677360037
modulus : 1024bits
Calculated : 1893182091795894505548455768518106725531190728580955194327233606232066153021643510474510233975193307796431149070890850406762278563092442192028738495770677113390077988192394982665533608193017
3792986940668784012534004118007894372145576314058511764884822391446965463843197220434273739142105087499232061900871970
Expected : 1893182091795894505548455768518106725531190728580955194327233606232066153021643510474510233975193307796431149070890850406762278563092442192028738495770677113390077988192394982665533608193017
3792986940668784012534004118007894372145576314058511764884822391446965463843197220434273739142105087499232061900871970
-----
Number 1: 63364610244545559321170346204262694063020505382933261977550131267845191230233698111049196951012049643784817430814393694230215244175519531222130232692829667142562287096243628322523438995495994212
11257846966019294993256428068066417781276049853962701429928570870771954437930885587018396967627039218836341239669738187249936361750849275680499417799924226112282864357763848820437847885942205533599260768
60301485630738946346563073676127296205972382122072844562807769639314660968962515720578673603341778906771795017931255354644165072457872540416933043098134687329554138452327854826474572352974723349456501
11500043
Number 2: 8449622564999510979567534763743016245490938299513553476367623540087314463825916726840358732482936440889930346596821913481071961410816871120244501807999353933848478294159522781471451625278146742
6530656940354328227681607830019779225445605579596949394335665743480939166764264382055813223795291812482309713276855923623293747990901630923196923343680165916581280884904350107313795245032896273282414976598
44645056329786560672421812715695637184448491972338445789657983242753594665481837447062623221765446367991127080220190511152934125615745223030022819619373510586709349763934140950152672791257061744860763527528
67291324
modulus : 2048bits
Calculated : 14786083589454066911684569384169285651094143353428869961431248607657663761282537970580507019925313405277412089678261282904893485828368403423747378082902107641076538965958060410089062877
4140954765631616249239332445719761265542993265082426423661425289360469514994075762920579953436594312567355330549579311161660012368000191752167940817770410952848198859551817372389232032861801
4237367849465419605255070138052548936965952441723131944112903524657528826625563444435901911990995779886146781804259721983636688398542122991010284064512923683324055442305177793737356353475698059149209411
298402978791367
Expected : 14786083589454066911684569384169285651094143353428869961431248607657663761282537970580507019925313405277412089678261282904893485828368403423747378082902107641076538965958060410089062877
4140954765631616249239332445719761265542993265082426423661425289360469514994075762920579953436594312567355330549579311161660012368000191752167940817770410952848198859551817372389232032861801
4237367849465419605255070138052548936965952441723131944112903524657528826625563444435901911990995779886146781804259721983636688398542122991010284064512923683324055442305177793737356353475698059149209411
298402978791367
-----
```

Modulo addition with negative numbers

```
-----
Number 1: -2171995909354312219327159734591309276566236375892077028430010578211126913609163472307732996622952473929318897372696063019780521300890575585538376038115502922848881825278846190570048346090519764529
984933264714083598738790014487001568691651326578867087973323242769874902344568522847443006422922857740695323511933
Number 2: 87598250036046328361573979508975797874567090991747491484232548410953461660727163243736934012890860403499259336212441047842264330033846334749008919591268211051890805664510363608528773389654028456
847404069298932532799340737057688466273193289779684968204195588940852651911426296135682164641491365677360037
modulus : 1024bits
Calculated : 6587829094250320616830238215984487021794345402539788641231490589840770298810816012963634050602813010567369596351637802061743029143270749210632881475765288203016231366257445515239389749134
26392695413935845336526573978920369008193011405354030708875744961425714035800883388578853129259241783750671253848104
Expected : 6587829094250320616830238215984487021794345402539788641231490589840770298810816012963634050602813010567369596351637802061743029143270749210632881475765288203016231366257445515239389749134
26392695413935845336526573978920369008193011405354030708875744961425714035800883388578853129259241783750671253848104
-----
Number 1: -63364610244545559321170346204262694063020505382933261977550131267845191230233698111049196951012049643784817430814393694230215244175519531222130232692829667142562287096243628322523438995495994212
211257846966019294993256428068066417781276049853962701429928570870771954437930885587018396967627039218836341239669738187249936361750849275680499417799924226112282864357763848820437847885942205533599260768
60301485630738946346563073676127296205972382122072844562807769639314660968962515720572867360334177890677179501793125535464416507245787254041693304309813468732955413845232785482647457235297472334945650
11500043
Number 2: 8449622564999510979567534763743016245490938299513553476367623540087314463825916726840358732482936440889930346596821913481871961410816871120244501807999353933848478294159522781471451625278146742
6530656940354328227681607830019779225445605579596949394335665743480939166764264382055813223795291812482309713276855923623293747990901630923196923343680165916581280884904350107313795245032896273282414976598
44645056329786560672421812715695637184448491972338445789657983242753594665481837447062623221765446367991127080220190511152934125615745223030022819619373510586709349763934140950152672791257061744860763527528
67291324
modulus : 2048bits
Calculated : 2113161540544955047495080143316746839887733245684220856591222354086625515235797457298667629727217465284486835153025406885046899326493398881142683951696867912861911971723239538934012629782
1525305404782237529393324457197612655722007249644070948727089847288333788233539735270325878059867630080015854175079438438158138166391929165680923655458452241326711619109416766173476217946815
715829843453706990476143267931075414226751247246705702560122685021360343893369651931587500533648573202589204490071839738561746970910849943577598112631506369711797639435008890816466999045535311937627414
07102755791281
Expected : 2113161540544955047495080143316746839887733245684220856591222354086625515235797457298667629727217465284486835153025406885046899326493398881142683951696867912861911971723239538934012629782
1525305404782237529393324457197612655722007249644070948727089847288333788233539735270325878059867630080015854175079438438158138166391929165680923655458452241326711619109416766173476217946815
715829843453706990476143267931075414226751247246705702560122685021360343893369651931587500533648573202589204490071839738561746970910849943577598112631506369711797639435008890816466999045535311937627414
07102755791281
-----
```

3.2. Modulo Multiplication Testing

Modulo multiplication with 0

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

Modulo multiplication with large numbers

```
Number 1: 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
Number 2: 6792592329354197244105845654208764936695958768703025559807810571944652062027784502491704296059281380780868630776397416292312812100127446982717232354202589
modulus : 512bits
Calculated : 1251507578769795846564208764936695958768703025559807810571944652062027784502491704296059281380780868630776397416292312812100127446982717232354202589
Expected : 1251507578769795846564208764936695958768703025559807810571944652062027784502491704296059281380780868630776397416292312812100127446982717232354202589
-----
Number 1: 21719590935412129327159734591309275662367358920770284300105782112691360916347230772996229524739291889737269063019780521300890575585538376038115502922848881825278846190570048346090519764529
8493264714083597387900144870015686916513257886708797323242769874902344568522847443086422922857740695323511933
Number 2: 8759825003604632836157379595897578745670909917474914842325484109534616607271632437369304128980680409239336212441047842264330033846334749080919591268211051890856645103636085287735839654028456
34474040699203526537299348073705768846627319328977968490682041955889408526519114262961356821646419136557736037
modulus : 1024bits
Calculated : 15264245273198623669176001307083905790281677127764393906990283449097776672043828637404358079874281543874744654441710745092308809477853689364527779589727853823541879156184592058996453332815
6300906484139615965359723103599336673177921183210677367844775988437824304626454338204989960259307509467366738045073
Expected : 15264245273198623669176001307083905790281677127764393906990283449097776672043828637404358079874281543874744654441710745092308809477853689364527779589727853823541879156184592058996453332815
6300906484139615965359723103599336673177921183210677367844775988437824304626454338204989960259307509467366738045073
-----
Number 1: 633646109244555932117034620426940560320593829332619775501312678451912302336981110491969510120496437848174308143936942302152441755195312221302326928296671425622870962436283222537438995495994212
1125784696601929499325642860866417781276048539627014299285708707719544379308855870183969676270392188363341239669738187249936361750849275680499417799924226112282864357763848820437847885942205533599260768
603014856307389463465630736712729620597238212220728445628077696393146609689625215720572867360334177890677179501793125535464416507245787254041693304309813468732955413845232785482674745723529747233349456501
11580043
Number 2: 8449622564990180979553746734730162454908382995135534767637633400873144638259167260493923248239264400898930346596821913481871961410816807112024450108079993539330484702934159522701471451625278146742
65385694054328272601687038019792254567959694934336657434809391676426438028558122279528181248230971327685592362293747999091630923186923343680165916581280884904350107313795245032898273282414976598
446450563297865606724218127156956371844484919723384457896579832427539465648183744706262322176544367991127080220190511152934125615745223830022819619373510586709349763934140950152672791257061744860763527528
67291324
modulus : 2048bits
Calculated : 11807412577886306517701956074495441833451608630036189153928631218647481794687152847063576128691643698612674558618132676589131369797979268045438720170912103981961688102415909103594970837
255651535104410981841610120116907584789835021091517602720704372861082538543046084343011149293378669194648502174693443287499065830240506524362760378363626405957086124353966547971814031593901215971125279
55911313725857622584126846830628739341051187408596570963232915218363693359455609997450417367922853065299075949111967198447091594750587782318206221074717763151937411467602364789852041731702358509132538
280547616365588
Expected : 118074125778862051770195607449544183345160863003610315393286312186474817946871528470635761286916436986126745586181326265891311369797979268045438720170912103981961688102415909103594970837
2556515351044109818416101201169075847898350210915176027207043728610825385430460843430111492933786691946485021746934432874990658302405065243626037836326450907096124353966547971814031593901215971125279
5591131372585762258412684683062873934105107408596570963232915218363693359455609997450417367922853065299075949111967198447091594750587782318206221074717763151937411467602364789852041731702358509132538
280547616365588
```

Modulo multiplication with negative numbers

```
Number 1: -21719590935412129327159734591309275662367358920770284300105782112691360916347230772996229524739291889737269063019780521300890575585538376038115502922848881825278846190570048346090519764529
8493264714083597387900144870015686916513257886708797323242769874902344568522847443086422922857740695323511933
Number 2: -8759825003604632836157379595897578745670909917474914842325484109534616607271632437369304128980680409239336212441047842264330033846334749080919591268211051890856645103636085287735839654028456
34474040699203526537299348073705768846627319328977968490682041955889408526519114262961356821646419136557736037
modulus : 1024bits
Calculated : 15264245273198623669176001307083905790281677127764393906990283449097776672043828637404358079874281543874744654441710745092308809477853689364527779589727853823541879156184592058996453332815
6300906484139615965359723103599336673177921183210677367844775988437824304626454338204989960259307509467366738045073
Expected : 15264245273198623669176001307083905790281677127764393906990283449097776672043828637404358079874281543874744654441710745092308809477853689364527779589727853823541879156184592058996453332815
6300906484139615965359723103599336673177921183210677367844775988437824304626454338204989960259307509467366738045073
-----
```

3.3. Modulo Inverse Testing

Modulo inverse with 1

```
Number 1: 1
modulus : 512bits
Calculated : 1
Expected : 1
```

Modulo inverse with large numbers

```
Number 1: 7411983660145561438669578801503455163517312471829001959053178951383381181774990719242089448703004658285326780366787478508242288148487675488967829478156097
modulus : 512bits
Calculated : 91474829002723385397052610191753006241439891818267418645968247112646001303848568608663746785895787813079850763254056636409668114587263601063751447537857
Expected : 91474829002723385397052610191753006241439891818267418645968247112646001303848568608663746785895787813079850763254056636409668114587263601063751447537857
-----
Number 1: 21719590935412129327159734591309275662367358920770284300105782112691360916347230772996229524739291889737269063019780521300890575585538376038115502922848881825278846190570048346090519764529
8493264714083597387900144870015686916513257886708797323242769874902344568522847443086422922857740695323511933
modulus : 1024bits
Calculated : 83826256547348512952030910909632443629155926634631776688551386553926620204050458563367207618774001461882319867976186444083314016783539421462207058839709571861149763430774594210867531250731
599040851296429830529579400322756588546522473524068478907367780374383073291375803440961078028461817910818836871335125
Expected : 83826256547348512952030910909632443629155926634631776688551386553926620204050458563367207618774001461882319867976186444083314016783539421462207058839709571861149763430774594210867531250731
599040851296429830529579400322756588546522473524068478907367780374383073291375803440961078028461817910818836871335125
modulus : 2048bits
Calculated : 2693899335454504792818034595940620188624860397197029301945210373262515277476992619970157405746028415151365546236175045840588000130140054834145791262322507828915133801535706403549090968553
6283927396012877146484999133447715644365259759788088070112408257323640553889196446977158471493866498185401280530752670945938939211968887127984610030593936061635225978853107564873548747205295668445206703
7200697773561029774249344620617944663224886018034798529802601136178185630263895837590720814221349745188855554641190467409891928779469346540478592657761438116958072028437738359608603619541178410858170586
496968829527203
Expected : 2693899335454504792818034595940620188624860397197029301945210373262515277476992619970157405746028415151365546236175045840588000130140054834145791262322507828915133801535706403549090968553
628392739601287714648499913344771564436525975978808807011240825732364055388919644697715847149386649818540128053075267094593893921196888712798461003059393606163525978853107564873548747205295668445206703
7200697773561029774249344620617944663224886018034798529802601136178185630263895837590720814221349745188855554641190467409891928779469346540478592657761438116958072028437738359608603619541178410858170586
496968829527203
```

Modulo inverse with modulo inverse does not exist

```
lumber 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.