KURUKULASOORIYA KVR 200332X

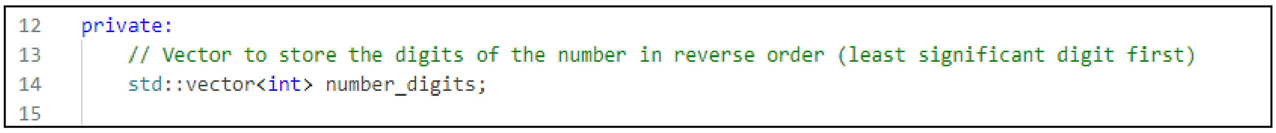
IMPLEMENTATION OF A BigNum LIBRARY
REPORT

INTRODUCTION

Modern cryptography is based on public key cryptosystems, which allow safe communication over untrusted networks. These systems work with very big integers in their arithmetic operations; these numbers are usually in the range of 512 bits to 2048 bits or more. The majority of computer languages' normal data types are insufficient for performing these computations, hence big-num libraries specialised libraries designed to handle enormous numbers must be used.
In this project, a BigNum library in C++ was used to facilitate the necessary arithmetic operations. The library is designed to handle integers of various sizes, as defined by the argument n, which indicates the modulus' bit length. The implementation implements important modular arithmetic operations such as addition, multiplication, and inversion, which are required for public key cryptosystems to function properly.

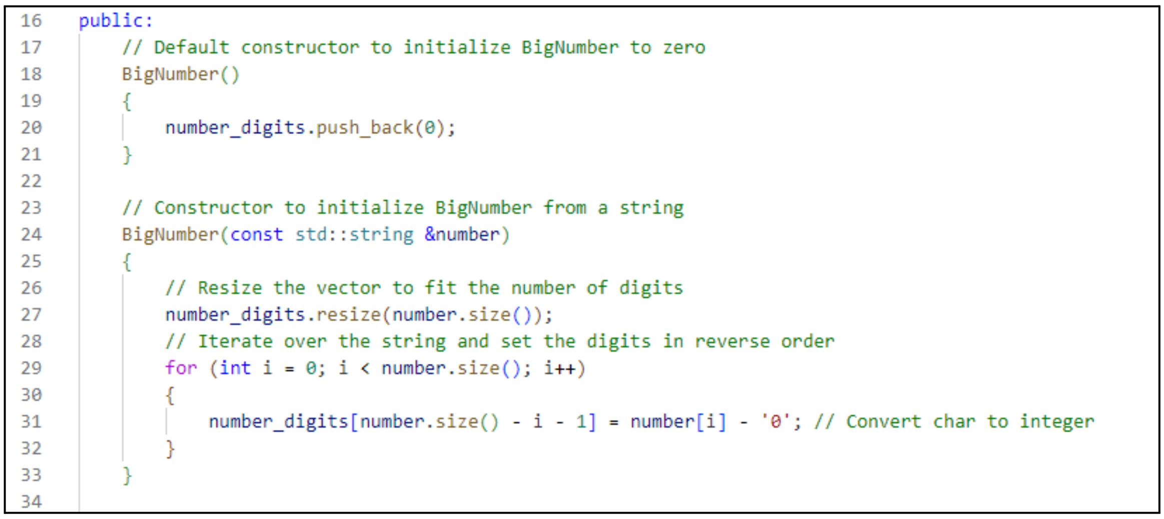
DESIGN AND IMPLEMENTATION

The BigNumber class is intended to handle large numbers that exceed the size limits for standard integer types. The method utilises a vector-based approach to store digits of a number in reverse order, allowing for efficient arithmetic operations thereby eliminating the need to manually handle carry propagation from the most significant to the least significant digit.



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

```

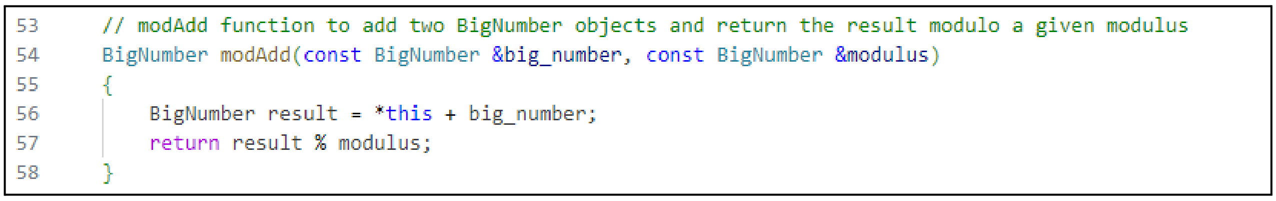
There are 2 constructors used to define the BigNumber either to zero or as an input taken from a string.



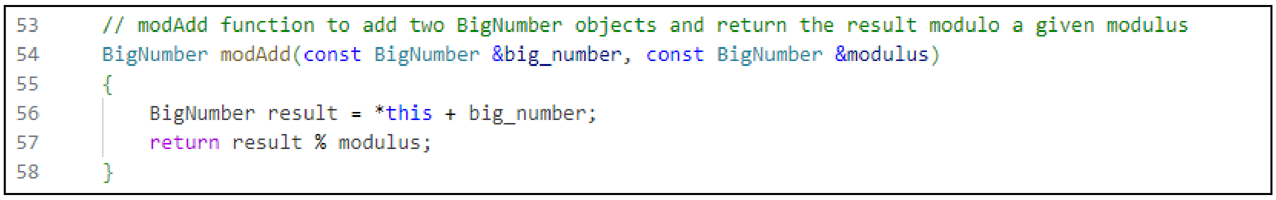
```cpp
public:
    // Default constructor to initialize BigNumber to zero
    BigNumber()
    {
        number_digits.push_back(0);
    }

    // Constructor to initialize BigNumber from a string
    BigNumber(const std::string &number)
    {
        // Resize the vector to fit the number of digits
        number_digits.resize(number.size());
        // Iterate over the string and set the digits in reverse order
        for (int i = 0; i < number.size(); i++)
        {
            number_digits[number.size() - i - 1] = number[i] - '0'; // Convert char to integer
        }
    }

```

Modular Arithmetic

modAdd and modMultiplication functions solve modular addition and multiplication respectively. The result of the arithmetic operation is first computed using the overloaded + and * operators, and then return the result modulo a given modulus using the % operator.

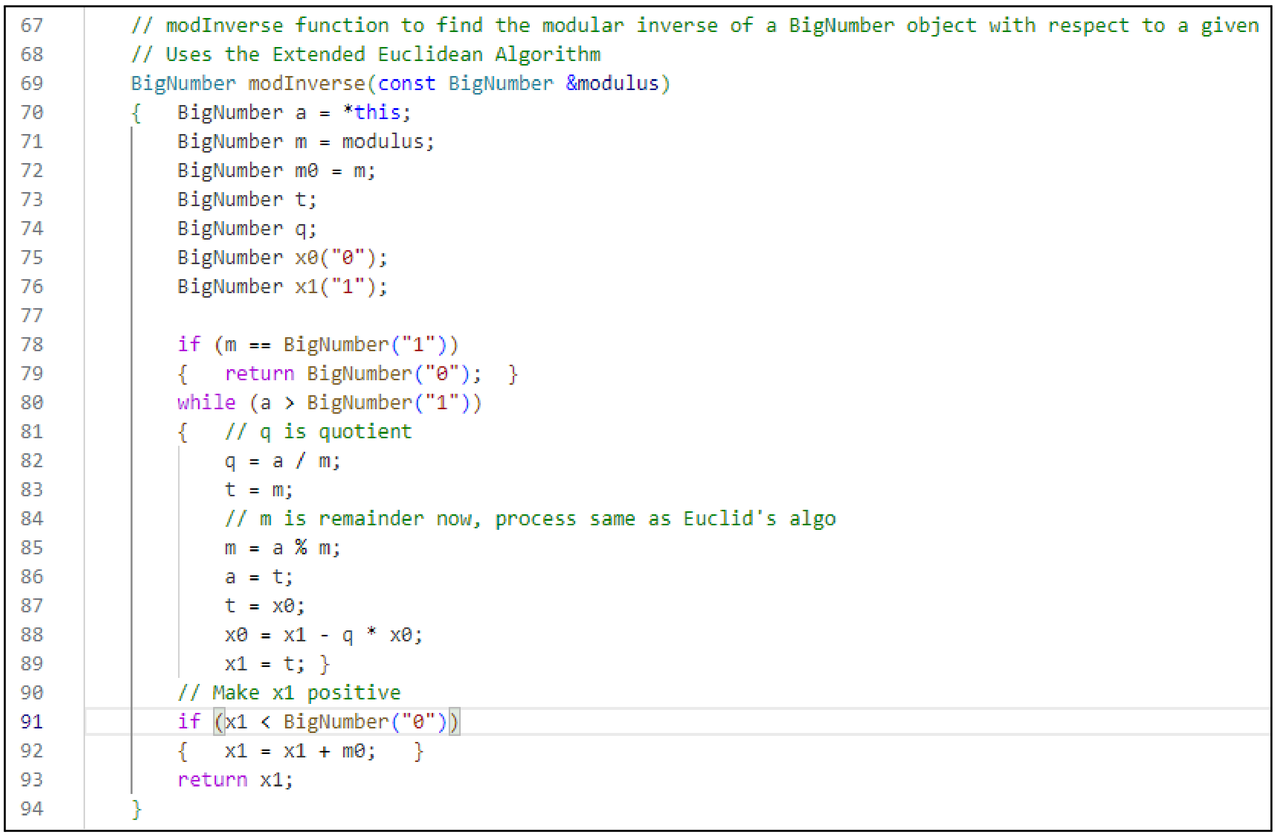


```cpp
// modAdd function to add two BigNumber objects and return the result modulo a given modulus
BigNumber modAdd(const BigNumber &big_number, const BigNumber &modulus)
{
    BigNumber result = *this + big_number;
    return result % modulus;
}
```



```cpp
// modAdd function to add two BigNumber objects and return the result modulo a given modulus
BigNumber modAdd(const BigNumber &big_number, const BigNumber &modulus)
{
    BigNumber result = *this + big_number;
    return result % modulus;
}
```

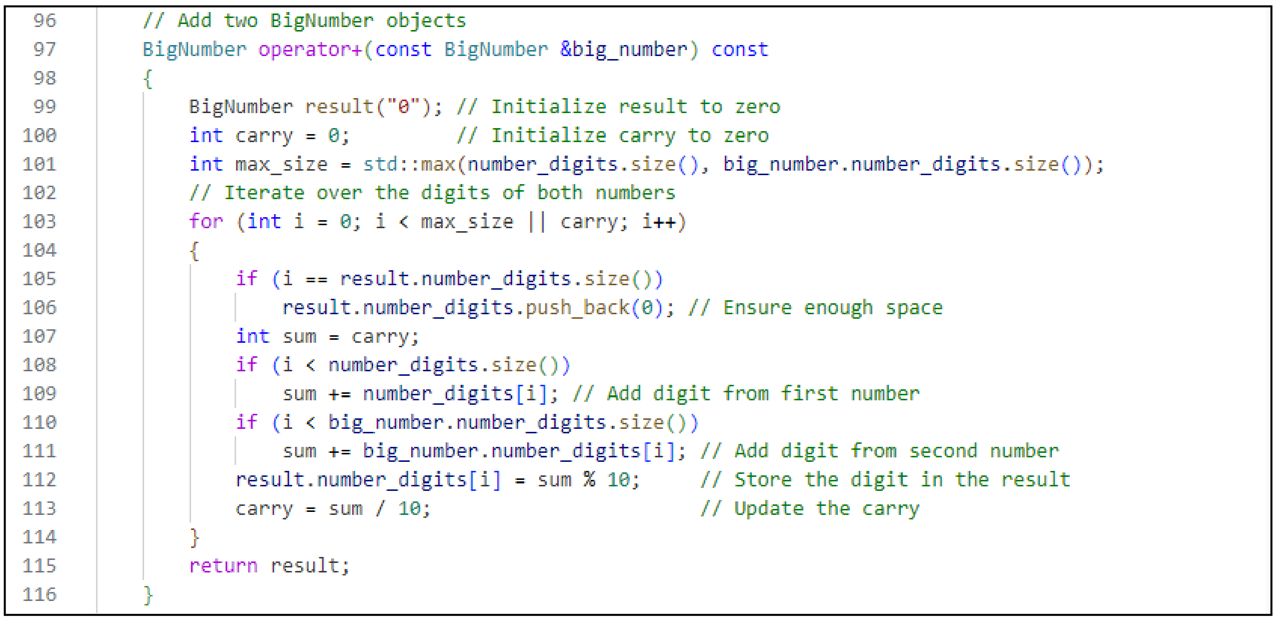
modInverse function uses the Extended Euclidean Algorithm to compute the modular inverse ensures that the result is always non-negative by adding the modulus when needed.



```cpp
// modInverse function to find the modular inverse of a BigNumber object with respect to a given
// Uses the Extended Euclidean Algorithm
BigNumber modInverse(const BigNumber &modulus)
{   BigNumber a = *this;
    BigNumber m = modulus;
    BigNumber m0 = m;
    BigNumber t;
    BigNumber q;
    BigNumber x0("0");
    BigNumber x1("1");

    if (m == BigNumber("1"))
    {   return BigNumber("0");  }
    while (a > BigNumber("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 < BigNumber("0"))
    {   x1 = x1 + m0;   }
    return x1;
}
```

Supportive Operators

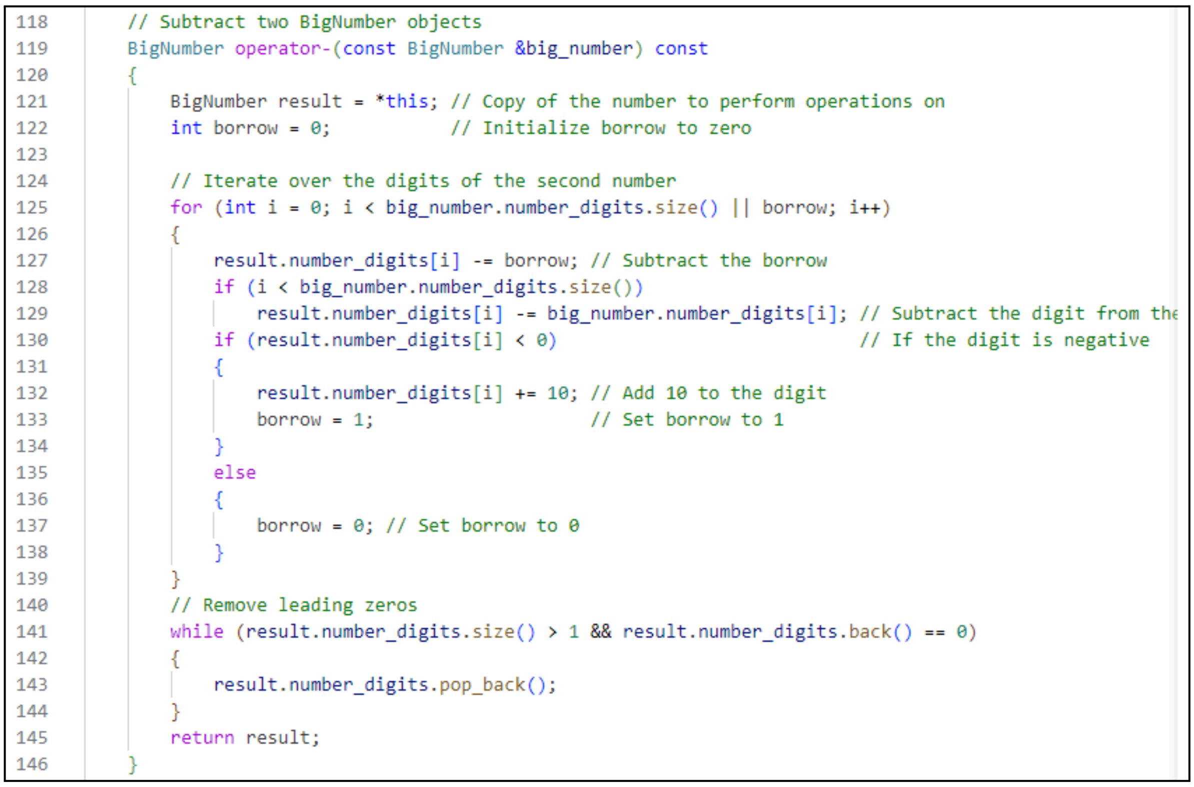
I) Addition (operator +):



```cpp
// Add two BigNumber objects
BigNumber operator+(const BigNumber &big_number) const
{
    BigNumber result("0"); // Initialize result to zero
    int carry = 0;         // Initialize carry to zero
    int max_size = std::max(number_digits.size(), big_number.number_digits.size());
    // Iterate over the digits of both numbers
    for (int i = 0; i < max_size || carry; i++)
    {
        if (i == result.number_digits.size())
            result.number_digits.push_back(0); // Ensure enough space
        int sum = carry;
        if (i < number_digits.size())
            sum += number_digits[i]; // Add digit from first number
        if (i < big_number.number_digits.size())
            sum += big_number.number_digits[i]; // Add digit from second number
        result.number_digits[i] = sum % 10;     // Store the digit in the result
        carry = sum / 10;                       // Update the carry
    }
    return result;
}
```

The two numbers are added digit by digit, beginning with the least significant digit. If the sum exceeds 10, the carry is moved to the next digit. The result is saved in a new BigNumber instance.

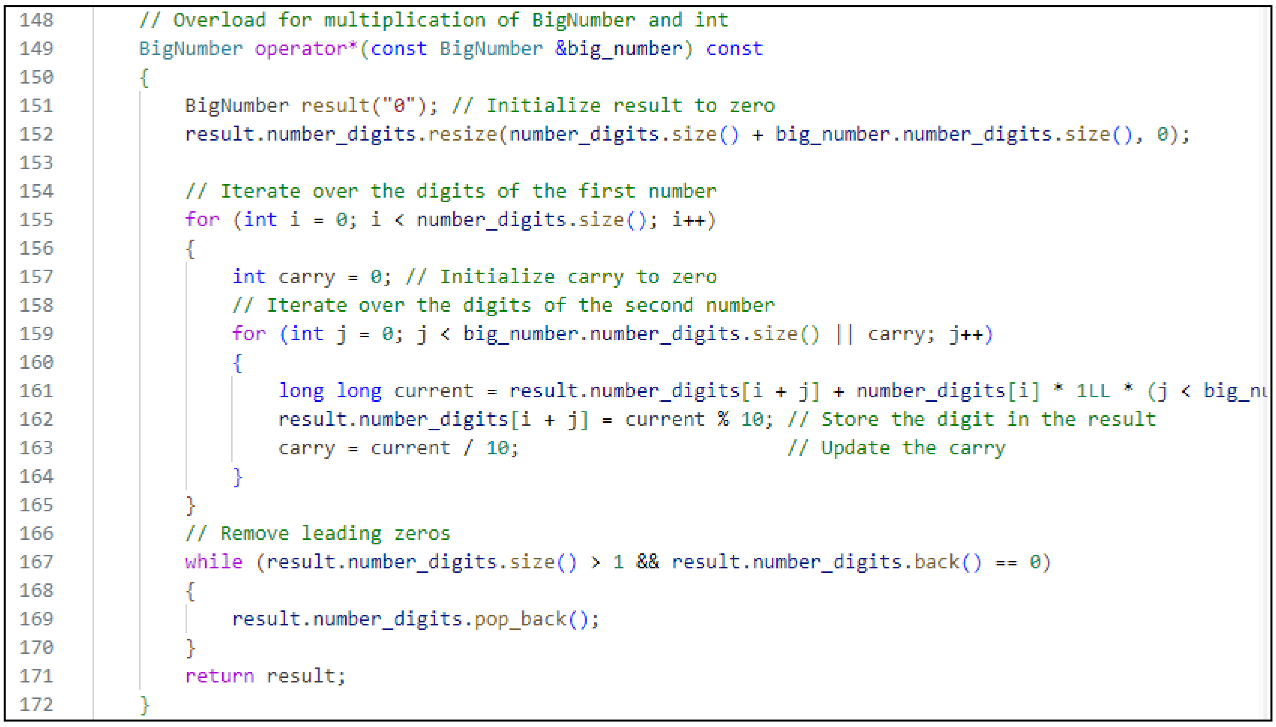
II) Subtraction (Operator-)



```cpp
// Subtract two BigNumber objects
BigNumber operator-(const BigNumber &big_number) const
{
    BigNumber result = *this; // Copy of the number to perform operations on
    int borrow = 0;           // Initialize borrow to zero

    // Iterate over the digits of the second number
    for (int i = 0; i < big_number.number_digits.size() || borrow; i++)
    {
        result.number_digits[i] -= borrow; // Subtract the borrow
        if (i < big_number.number_digits.size())
            result.number_digits[i] -= big_number.number_digits[i]; // Subtract the digit from the
        if (result.number_digits[i] < 0)                             // If the digit is negative
        {
            result.number_digits[i] += 10; // Add 10 to the digit
            borrow = 1;                    // Set borrow to 1
        }
        else
        {
            borrow = 0; // Set borrow to 0
        }
    }
    // Remove leading zeros
    while (result.number_digits.size() > 1 && result.number_digits.back() == 0)
    {
        result.number_digits.pop_back();
    }
    return result;
}
```

This performs digit-by-digit subtraction. If a digit is negative, it borrows from the next higher digit. The output correctly handles borrowing, with no negative digits remaining and leading zeros are deleted after subtraction.

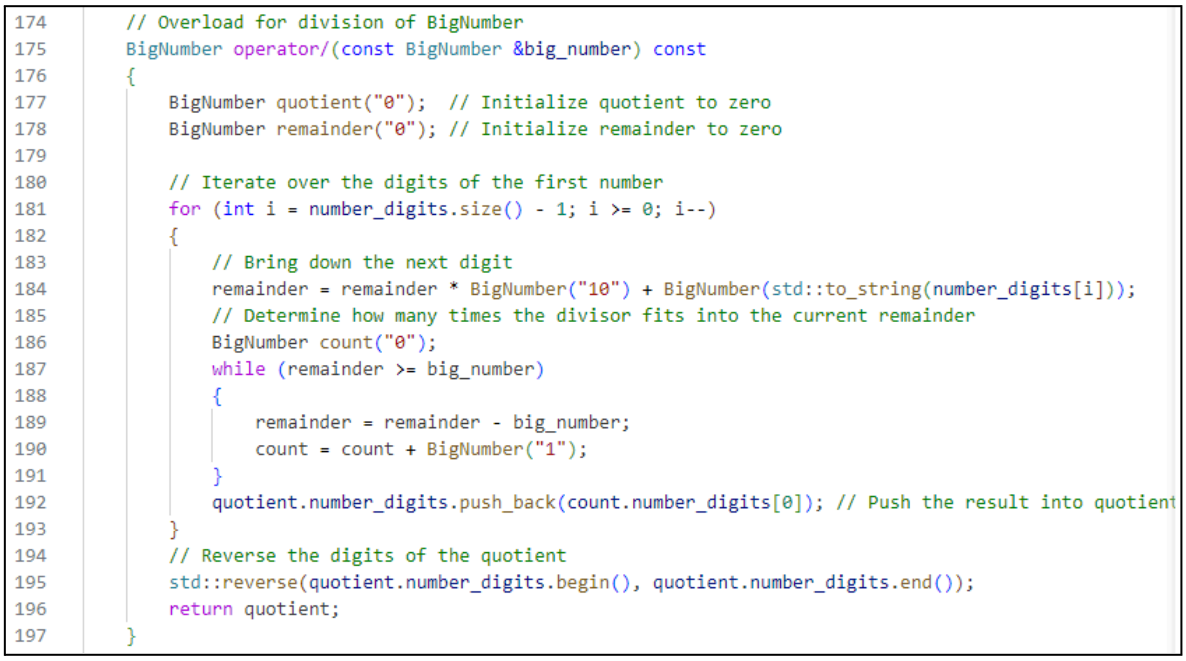
III) Multiplication (Operator*):



```
    // Overload for multiplication of BigNumber and int
    BigNumber operator*(const BigNumber &big_number) const
    {
        BigNumber result("0"); // Initialize result to zero
        result.number_digits.resize(number_digits.size() + big_number.number_digits.size(), 0);

        // Iterate over the digits of the first number
        for (int i = 0; i < number_digits.size(); i++)
        {
            int carry = 0; // Initialize carry to zero
            // Iterate over the digits of the second number
            for (int j = 0; j < big_number.number_digits.size() || carry; j++)
            {
                long long current = result.number_digits[i + j] + number_digits[i] * 1LL * (j < big_nu
                result.number_digits[i + j] = current % 10; // Store the digit in the result
                carry = current / 10;                       // Update the carry
            }
        }
        // Remove leading zeros
        while (result.number_digits.size() > 1 && result.number_digits.back() == 0)
        {
            result.number_digits.pop_back();
        }
        return result;
    }
```

This function does digit-wise multiplication with carry propagation. The result vector is initialised with enough digits to represent the product of two large numbers and enables efficient multiplication even with large inputs.

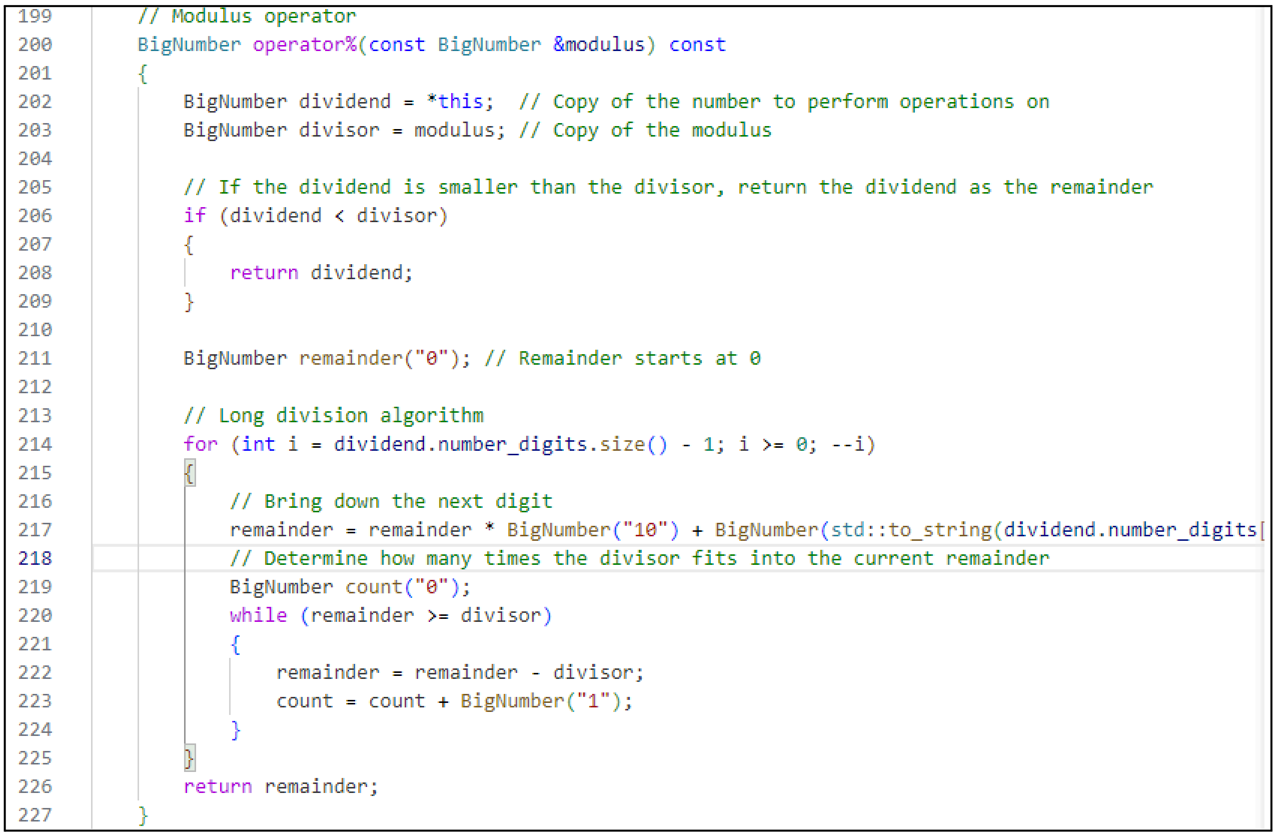
IV) Division (Operator/):



```
    // Overload for division of BigNumber
    BigNumber operator/(const BigNumber &big_number) const
    {
        BigNumber quotient("0");  // Initialize quotient to zero
        BigNumber remainder("0"); // Initialize remainder to zero

        // Iterate over the digits of the first number
        for (int i = number_digits.size() - 1; i >= 0; i--)
        {
            // Bring down the next digit
            remainder = remainder * BigNumber("10") + BigNumber(std::to_string(number_digits[i]));
            // Determine how many times the divisor fits into the current remainder
            BigNumber count("0");
            while (remainder >= big_number)
            {
                remainder = remainder - big_number;
                count = count + BigNumber("1");
            }
            quotient.number_digits.push_back(count.number_digits[0]); // Push the result into quotient
        }
        // Reverse the digits of the quotient
        std::reverse(quotient.number_digits.begin(), quotient.number_digits.end());
        return quotient;
    }
```

In this function, long division is implemented with the quotient being built digit by digit. It calculates how many times the divisor fits within the existing remainder and accumulates the quotient at each step.

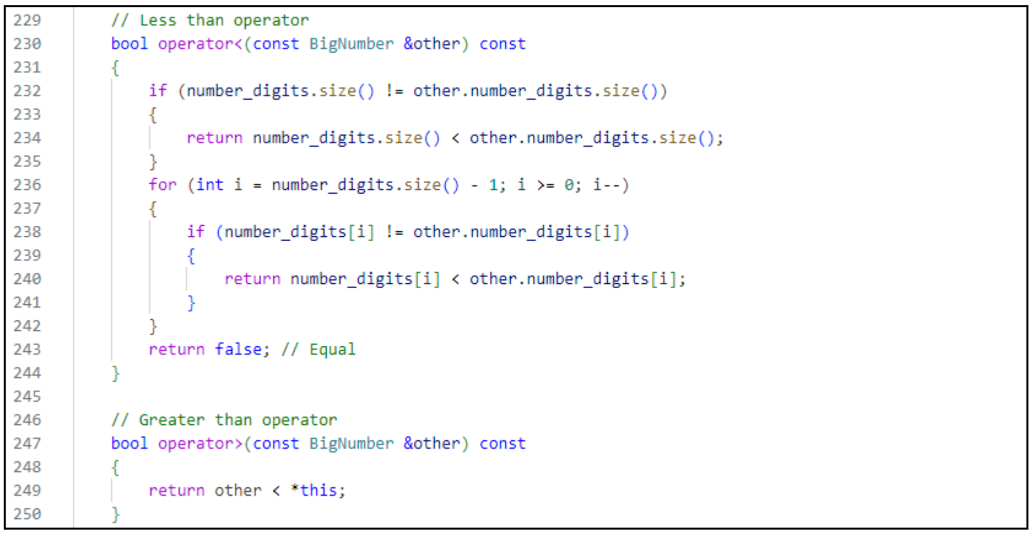
V) Modulus (Operator%)



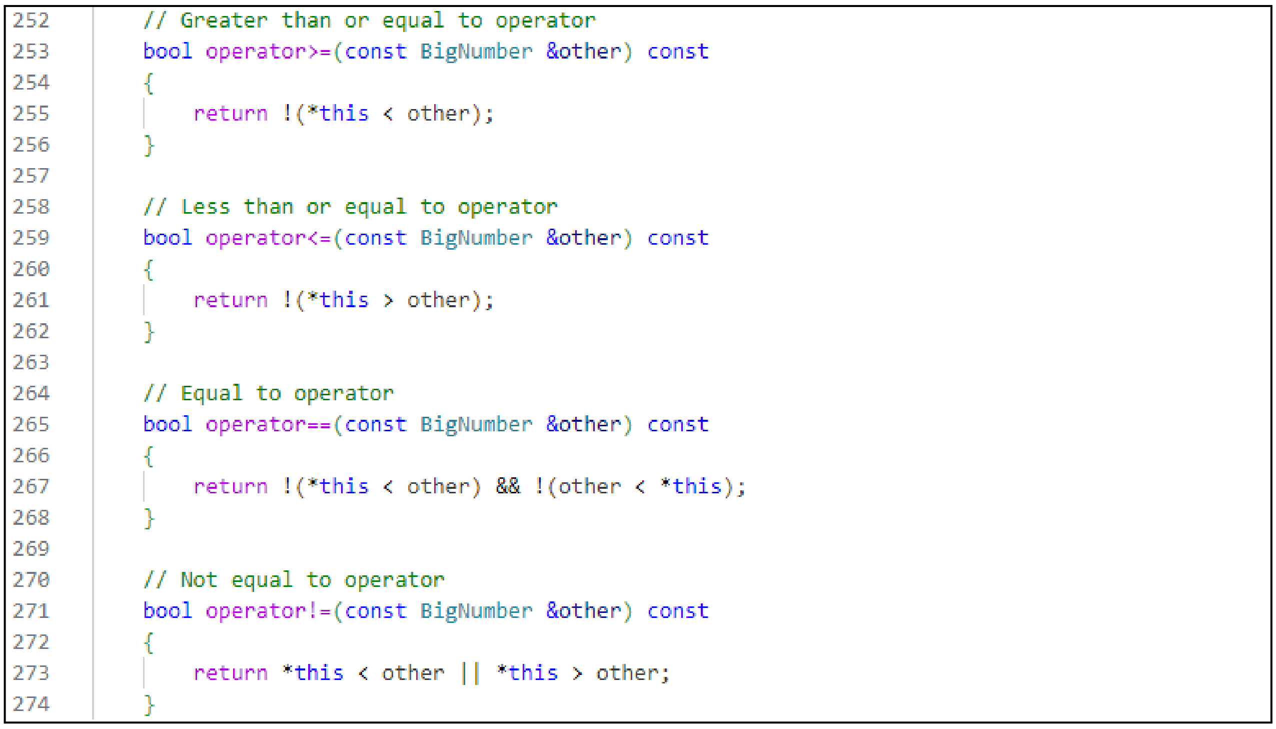
```cpp
// Modulus operator
BigNumber operator%(const BigNumber &modulus) const
{
    BigNumber dividend = *this;  // Copy of the number to perform operations on
    BigNumber divisor = modulus; // Copy of the modulus

    // If the dividend is smaller than the divisor, return the dividend as the remainder
    if (dividend < divisor)
    {
        return dividend;
    }

    BigNumber remainder("0"); // Remainder starts at 0

    // Long division algorithm
    for (int i = dividend.number_digits.size() - 1; i >= 0; --i)
    {
        // Bring down the next digit
        remainder = remainder * BigNumber("10") + BigNumber(std::to_string(dividend.number_digits[
        // Determine how many times the divisor fits into the current remainder
        BigNumber count("0");
        while (remainder >= divisor)
        {
            remainder = remainder - divisor;
            count = count + BigNumber("1");
        }
    }
    return remainder;
}
```

This calculates the remainder when dividing the current BigNumber by the specified modulus. The residual is calculated using a long division technique and if the number is less than the modulus, it is returned.

VI) Comparison (Operator (<, >, <=, >=, ==,!=))



```cpp
// Less than operator
bool operator<(const BigNumber &other) const
{
    if (number_digits.size() != other.number_digits.size())
    {
        return number_digits.size() < other.number_digits.size();
    }
    for (int i = number_digits.size() - 1; i >= 0; i--)
    {
        if (number_digits[i] != other.number_digits[i])
        {
            return number_digits[i] < other.number_digits[i];
        }
    }
    return false; // Equal
}

// Greater than operator
bool operator>(const BigNumber &other) const
{
    return other < *this;
}
```



```cpp
// Greater than or equal to operator
bool operator>=(const BigNumber &other) const
{
    return !(*this < other);
}

// Less than or equal to operator
bool operator<=(const BigNumber &other) const
{
    return !(*this > other);
}

// Equal to operator
bool operator==(const BigNumber &other) const
{
    return !(*this < other) && !(other < *this);
}

// Not equal to operator
bool operator!=(const BigNumber &other) const
{
    return *this < other || *this > other;
}
```

These operators allow for the comparison of two BigNumber instances. First, they compare the size (number of digits), and if they are equal, they compare the digits in order of their significance. This ensures accurate lexicographical comparison for huge numbers.

TESTING

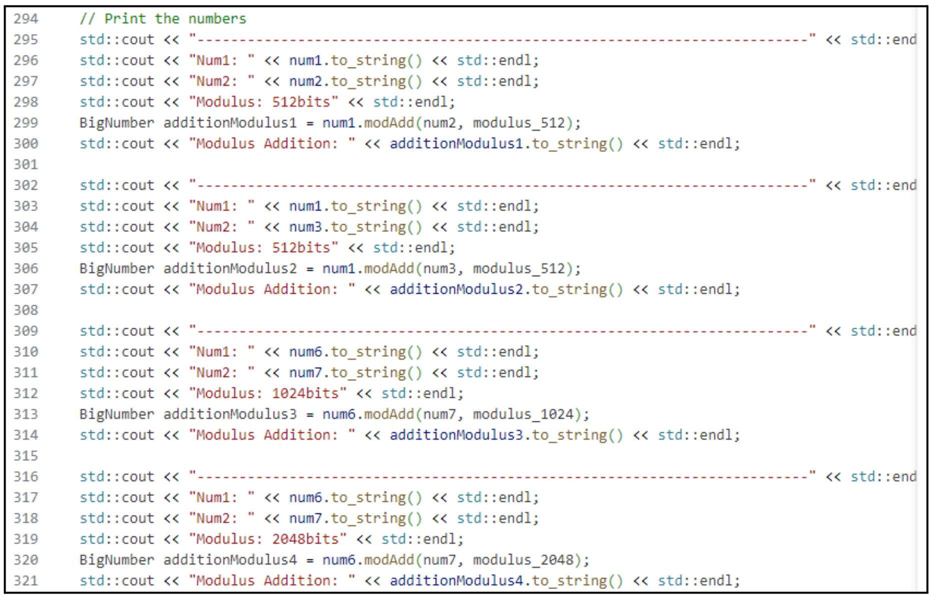
In order to test the correctness of the library following approach was taken:

For modular addition and multiplication,

1) Num1 and Num2 were initialised with large numbers.
2) The size of the moduli was initialised with either 512 bits/ 1024 bits or 2048 bits.
3) The function was called with specified Num1 and 2 inputs and the size of moduli.

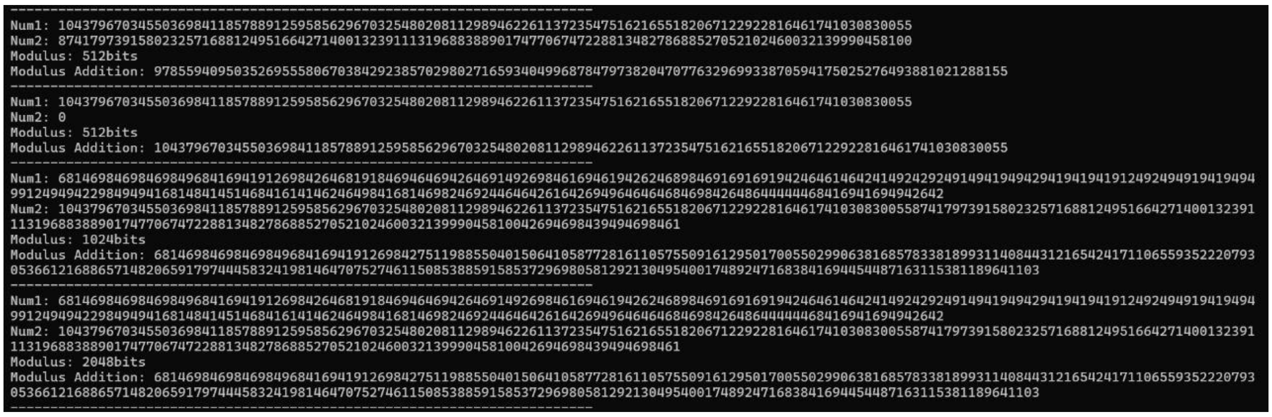
I) Modular Addition

Testing Commands:



```cpp
// Print the numbers
std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num1.to_string() << std::endl;
std::cout << "Num2: " << num2.to_string() << std::endl;
std::cout << "Modulus: 512bits" << std::endl;
BigNumber additionModulus1 = num1.modAdd(num2, modulus_512);
std::cout << "Modulus Addition: " << additionModulus1.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num1.to_string() << std::endl;
std::cout << "Num2: " << num3.to_string() << std::endl;
std::cout << "Modulus: 512bits" << std::endl;
BigNumber additionModulus2 = num1.modAdd(num3, modulus_512);
std::cout << "Modulus Addition: " << additionModulus2.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num6.to_string() << std::endl;
std::cout << "Num2: " << num7.to_string() << std::endl;
std::cout << "Modulus: 1024bits" << std::endl;
BigNumber additionModulus3 = num6.modAdd(num7, modulus_1024);
std::cout << "Modulus Addition: " << additionModulus3.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num6.to_string() << std::endl;
std::cout << "Num2: " << num7.to_string() << std::endl;
std::cout << "Modulus: 2048bits" << std::endl;
BigNumber additionModulus4 = num6.modAdd(num7, modulus_2048);
std::cout << "Modulus Addition: " << additionModulus4.to_string() << std::endl;
```

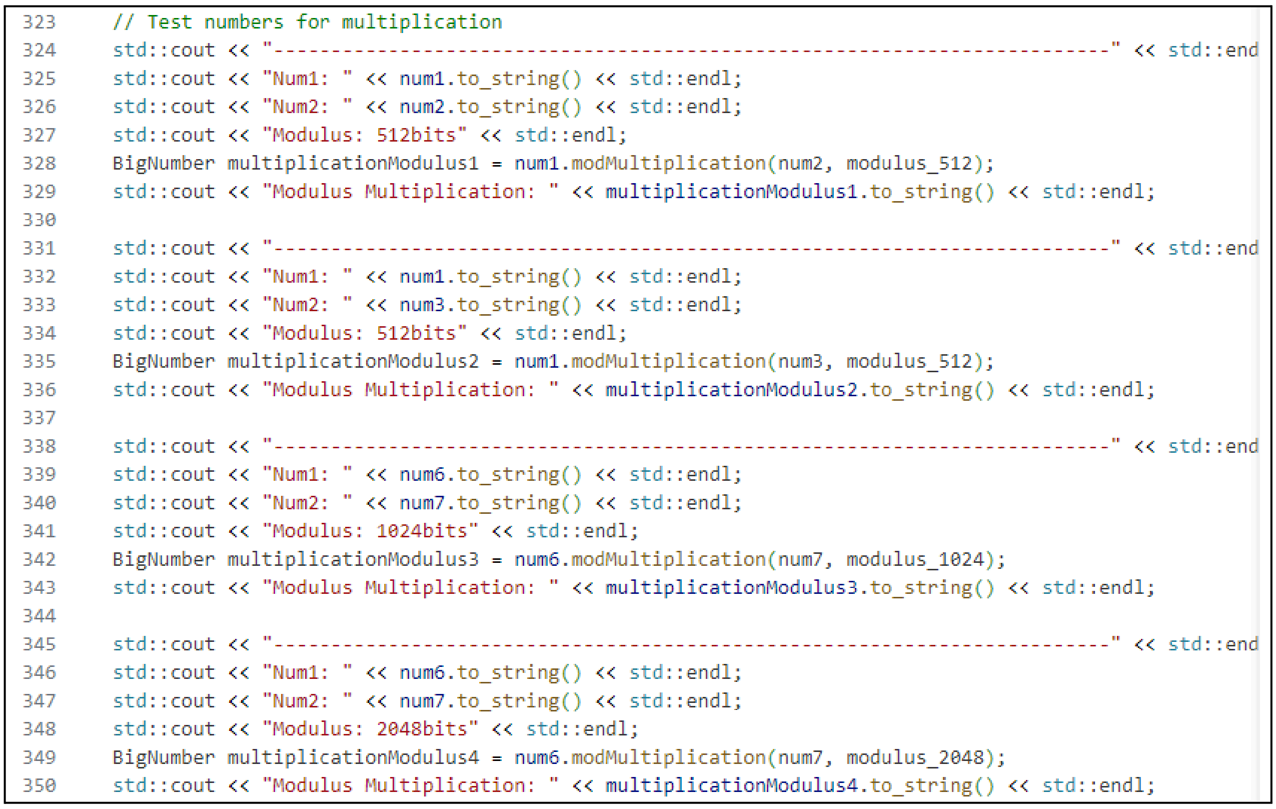
CLI Outputs:



```
------------------------------------------------------------------------
Num1: 10437967034550369841185788912595856296703254802081129894622611372354751621655182067122922816461741030830055
Num2: 87417973915802325716881249516642714001323911131968838890174770674722881348278688527052102460032139990458100
Modulus: 512bits
Modulus Addition: 97855940950352695558067038429238570298027165934049968784797382047077632969933870594175025276493881021288155
------------------------------------------------------------------------
Num1: 10437967034550369841185788912595856296703254802081129894622611372354751621655182067122922816461741030830055
Num2: 0
Modulus: 512bits
Modulus Addition: 10437967034550369841185788912595856296703254802081129894622611372354751621655182067122922816461741030830055
------------------------------------------------------------------------
Num1: 6814698469846984968416941912698426468191846946469426469149269846169461942624689846916916919424646146424149242924914941949429419419419124924949194194949
9124949422984949416814841451468416141462464984168146982469244646426164269496464646846984264864444446841694169494264 2
Num2: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
131968838890174770674722881348278688527052102460032139990458100426946984394946984 61
Modulus: 1024bits
Modulus Addition: 6814698469846984968416941912698427511988550401506410587728161105755091612950170055029906381685783381899311408443121654241711065593522207930
53661216886571482065917974445832419814647075274611508538859158537296980581292130495400174892471683841694454487163115381189641103
------------------------------------------------------------------------
Num1: 6814698469846984968416941912698426468191846946469426469149269846169461942624689846916916919424646146424149242924914941949429419419419124924949194194949
9124949422984949416814841451468416141462464984168146982469244646426164269496464646846984264864444446841694169494264 2
Num2: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
131968838890174770674722881348278688527052102460032139990458100426946984394946984 61
Modulus: 2048bits
Modulus Addition: 6814698469846984968416941912698427511988550401506410587728161105755091612950170055029906381685783381899311408443121654241711065593522207930
53661216886571482065917974445832419814647075274611508538859158537296980581292130495400174892471683841694454487163115381189641103
------------------------------------------------------------------------
```

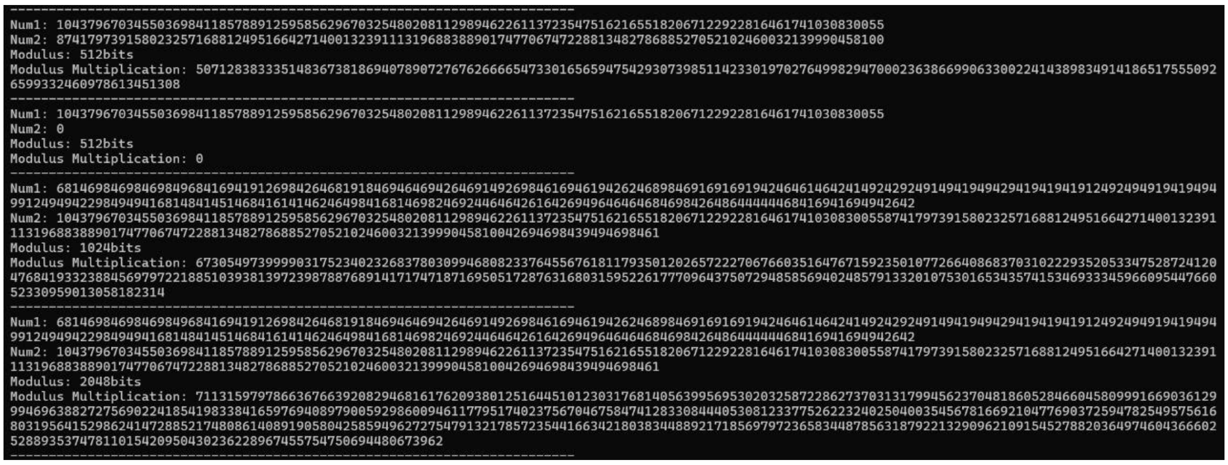
II) Modular Multiplication

Testing Commands:



```
// Test numbers for multiplication
std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num1.to_string() << std::endl;
std::cout << "Num2: " << num2.to_string() << std::endl;
std::cout << "Modulus: 512bits" << std::endl;
BigNumber multiplicationModulus1 = num1.modMultiplication(num2, modulus_512);
std::cout << "Modulus Multiplication: " << multiplicationModulus1.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num1.to_string() << std::endl;
std::cout << "Num2: " << num3.to_string() << std::endl;
std::cout << "Modulus: 512bits" << std::endl;
BigNumber multiplicationModulus2 = num1.modMultiplication(num3, modulus_512);
std::cout << "Modulus Multiplication: " << multiplicationModulus2.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num6.to_string() << std::endl;
std::cout << "Num2: " << num7.to_string() << std::endl;
std::cout << "Modulus: 1024bits" << std::endl;
BigNumber multiplicationModulus3 = num6.modMultiplication(num7, modulus_1024);
std::cout << "Modulus Multiplication: " << multiplicationModulus3.to_string() << std::endl;

std::cout << "------------------------------------------------------------------------" << std::end
std::cout << "Num1: " << num6.to_string() << std::endl;
std::cout << "Num2: " << num7.to_string() << std::endl;
std::cout << "Modulus: 2048bits" << std::endl;
BigNumber multiplicationModulus4 = num6.modMultiplication(num7, modulus_2048);
std::cout << "Modulus Multiplication: " << multiplicationModulus4.to_string() << std::endl;
```

CLI Outputs:



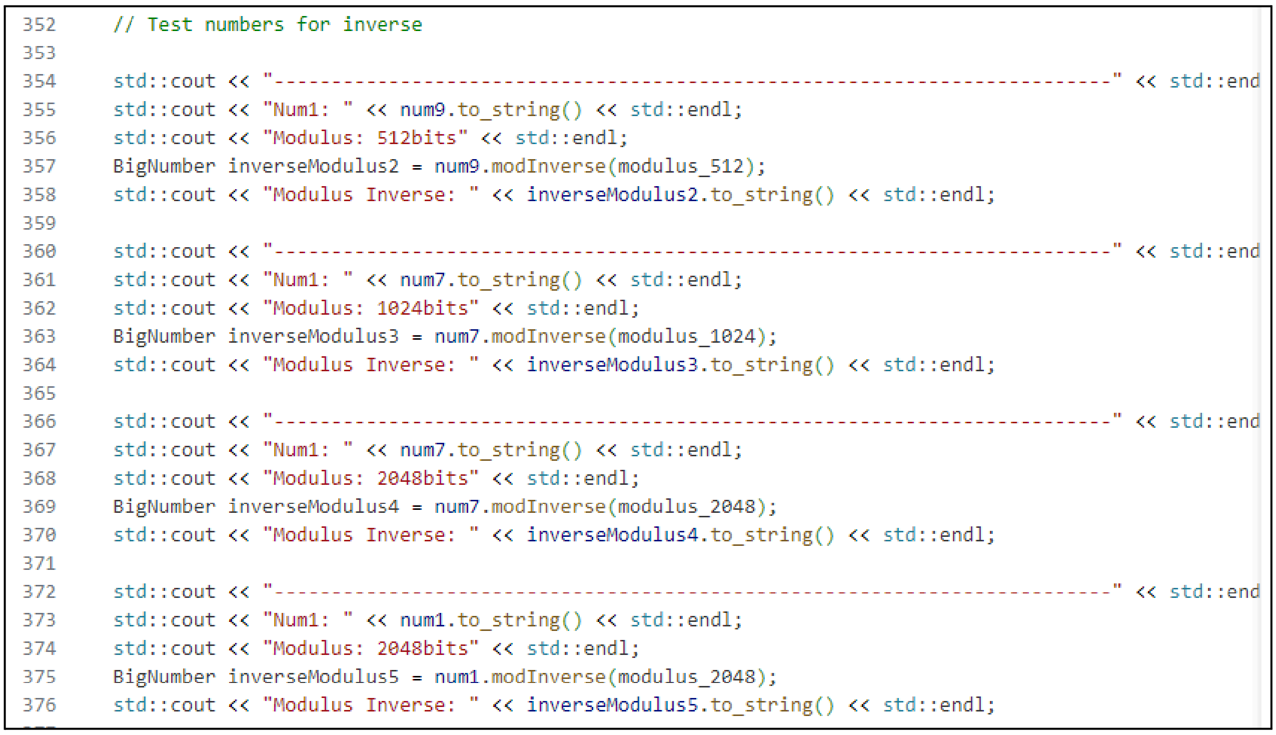
```
------------------------------------------------------------------------
Num1: 10437967034550369841185788912595856296703254802081129894622611372354751621655182067122922816461741030830055
Num2: 87417973915802325716881249516642714001323911131968838890174770674722881348278688527052102460032139990458100
Modulus: 512bits
Modulus Multiplication: 5071283833351483673818694078907276762666654733016565947542930739851142330197027649982947000236386699063300224143898349141865175550926
59933246097861345130 8
------------------------------------------------------------------------
Num1: 10437967034550369841185788912595856296703254802081129894622611372354751621655182067122922816461741030830055
Num2: 0
Modulus: 512bits
Modulus Multiplication: 0
------------------------------------------------------------------------
Num1: 6814698469846984968416941912698426468191846946469426469149269846169461942624689846916916919424646146424149242924914941949429419419419124924949194194949
9124949422984949416814841451468416141462464984168146982469244646426164269496464646846984264864444446841694169494264 2
Num2: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
131968838890174770674722881348278688527052102460032139990458100426946984394946984 61
Modulus: 1024bits
Modulus Multiplication: 6730549739999031752340232683780309946808233764556761811793501202657222706766035164767159235010772664086837031022293520533475287241204
76841933238845697972218851039381397239878876891417174718716950517287631680315952261777096437507294858569402485791332010753016534357415346933345966095447660
52330959013058182314
------------------------------------------------------------------------
Num1: 6814698469846984968416941912698426468191846946469426469149269846169461942624689846916916919424646146424149242924914941949429419419419124924949194194949
9124949422984949416814841451468416141462464984168146982469244646426164269496464646846984264864444446841694169494264 2
Num2: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
131968838890174770674722881348278688527052102460032139990458100426946984394946984 61
Modulus: 2048bits
Modulus Multiplication: 7113159797866367663920829468161762093801251644510123031768140563995695302032587228627370313179945623704818605284660458099916690361299
94696388272756902241854198338416597694089790059298600946117795174023756704675847412833084440530812337752622324025040035456781669210477690372594782549575616
80319564152986241472885217480861408919058042585949627275479132178572354416634218038344889217185697972365834487856318792213290962109154527882036497460436660 2
528893537478110154209504302362289674557547506944806739 62
------------------------------------------------------------------------
```

For modular inverse,

1) Num1 was initialised with a large number
2) The size of the moduli was initialised with either 512 bits/ 1024 bits or 2048 bits.
3) The function was called with specified Num1 and the size of moduli.

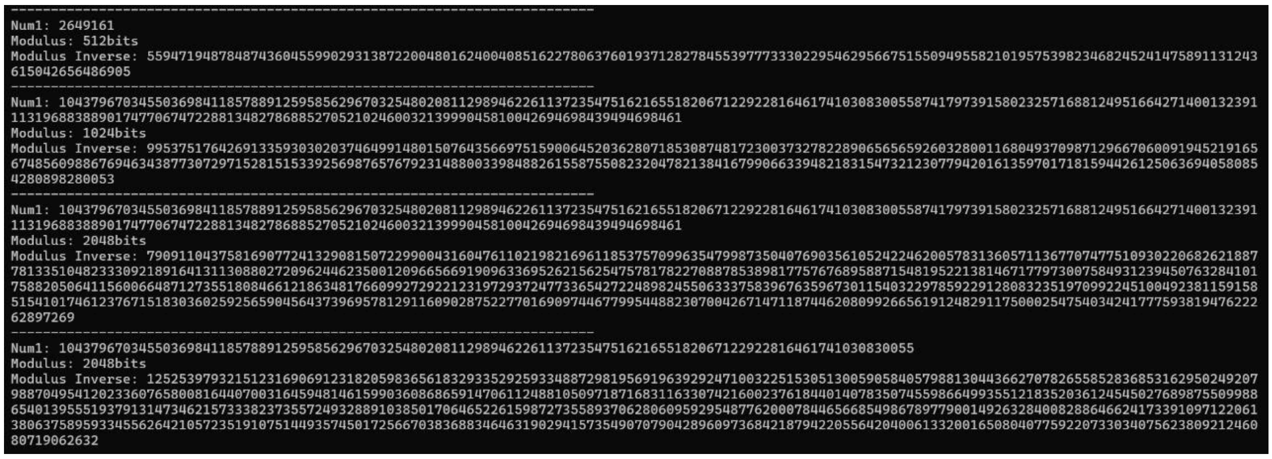
Modular Inverse

Testing Commands:



```
    // Test numbers for inverse

    std::cout << "------------------------------------------------------------------------" << std::end
    std::cout << "Num1: " << num9.to_string() << std::endl;
    std::cout << "Modulus: 512bits" << std::endl;
    BigNumber inverseModulus2 = num9.modInverse(modulus_512);
    std::cout << "Modulus Inverse: " << inverseModulus2.to_string() << std::endl;

    std::cout << "------------------------------------------------------------------------" << std::end
    std::cout << "Num1: " << num7.to_string() << std::endl;
    std::cout << "Modulus: 1024bits" << std::endl;
    BigNumber inverseModulus3 = num7.modInverse(modulus_1024);
    std::cout << "Modulus Inverse: " << inverseModulus3.to_string() << std::endl;

    std::cout << "------------------------------------------------------------------------" << std::end
    std::cout << "Num1: " << num7.to_string() << std::endl;
    std::cout << "Modulus: 2048bits" << std::endl;
    BigNumber inverseModulus4 = num7.modInverse(modulus_2048);
    std::cout << "Modulus Inverse: " << inverseModulus4.to_string() << std::endl;

    std::cout << "------------------------------------------------------------------------" << std::end
    std::cout << "Num1: " << num1.to_string() << std::endl;
    std::cout << "Modulus: 2048bits" << std::endl;
    BigNumber inverseModulus5 = num1.modInverse(modulus_2048);
    std::cout << "Modulus Inverse: " << inverseModulus5.to_string() << std::endl;
```

CLI Outputs:



```
------------------------------------------------------------------------
Num1: 2649161
Modulus: 512bits
Modulus Inverse: 5594719487848743604559902931387220048016240040851622780637601937128278455397773330229546295667515509495582101957539823468245241475891131243
615042656486905
------------------------------------------------------------------------
Num1: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
113196883889017477067472288134827868852705210246003213999045810042694698439494698461
Modulus: 1024bits
Modulus Inverse: 995375176426913359303020374649914801507643566975159006452036280718530874817230037327822890656565926032800116804937098712966706009194521 9165
674856098867694634387730729715281515339256987657679231488003398488261558755082320478213841679906633948218315473212307794201613597017181594426125063694058085
4280898280053
------------------------------------------------------------------------
Num1: 1043796703455036984118578891259585629670325480208112989462261137235475162165518206712292281646174103083005587417973915802325716881249516642714001323911
113196883889017477067472288134827868852705210246003213999045810042694698439494698461
Modulus: 2048bits
Modulus Inverse: 7909110437581690772413290815072299004316047611021982169611853757099635479987350407690356105242246200578313605711367707477510930220682621887
781335104823330921891641311308802720962446235001209665669190963369526215625475781782270887853898177576768958871548195221381467177973007584931239450763284101
758820506411560066487127355180846612186348176609927292212319729372477336542722489824550633375839676359673011540322978592291280832351970992245100492381159158
515410174612376715183036025925659045643739695781291160902875227701690974467799544882307004267147118744620809926656191248291175000254754034241777593819476222
62897269
------------------------------------------------------------------------
Num1: 104379670345503698411857889125958562967032548020811298946226113723547516216551820671229228164617410308300 55
Modulus: 2048bits
Modulus Inverse: 1252539793215123169069123182059836561832933529259334887298195691963929247100322515305130059058405798813044366270782655852836853162950249207
988704954120233607658008164407003164594814615990360868659147061124881050971871683116330742160023761844014078350745598664993551218352036124545027689875509988
654013955519379131473462157333823735572493288910385017064652261598727355893706280609592954877620007844656685498678977900149263284008288646624173391097122061
380637589593345562642105723519107514493574501725667038368834646319029415735490707904289609736842187942205564204006133200165080407759220733034075623809212460
80719062632
```

SUMMARY

The BigNumber class was created with flexibility in mind, allowing for the manipulation of big numbers using basic arithmetic operations. By incorporating modular arithmetic and comparison operators, it becomes ideal for cryptography and number theory applications.