Report on the Implementation of the BigNum Library

Author

- Index Number: 200193U
- Name: Chathura Gunasekara

Introduction

The `BigNum` library provides a custom implementation of large integers for mathematical operations that exceed the typical limitations of primitive data types in C++. This library supports operations such as addition, subtraction, multiplication, division, and modulus with large integers represented as a vector of digits. It also includes functions for modular arithmetic, including modular addition, multiplication, and inversion. The library is designed to handle both positive and negative integers.

Class Structure

The `BigNum` class is the core of the library, encapsulating a vector of digits to represent arbitrarily large numbers. Each digit is stored in a `std::vector<int>`, and additional member variables track the sign of the number.

Constructors

- **Default Constructor**: Initializes the `BigNum` object with a single digit of zero and a positive sign.
- **Parameterized Constructor (long long)**: Takes a `long long` integer as input, determines its sign, and converts the integer into a vector of digits. If the number is negative, its sign is stored, and the absolute value is processed.
- **Parameterized Constructor (string)**: Takes a string representation of a number as input, determines its sign, and converts it into a vector of digits. The string can include a leading negative sign.

Utility Functions

- **getModulus(const int mod_bit_length)**: Returns the modulus of the number based on the specified bit length. The modulus is defined as $2^{bitlength}$.

Helper Functions

- **removeLeadingZeros()**: Removes any leading zeros from the vector of digits to maintain a compact representation.
- **add()**: Implements addition between two `BigNum` objects. It supports the addition of both positive and negative numbers by checking the signs and adjusting accordingly.
- **subtract()**: Implements subtraction by considering the signs of the operands. If both numbers have the same sign, a simple digit-wise subtraction is performed; otherwise, addition is used after adjusting the signs.
- **multiply()**: Implements multiplication using a digit-by-digit multiplication algorithm, storing the result in the `BigNum` object.
- **divide()**: Performs integer division, returning both the quotient and the remainder as a pair of `BigNum` objects. It uses a binary search within each digit to find the appropriate quotient digit.

Operator Overloading

The library overloads several operators to provide intuitive usage:

- **Arithmetic Operators**: Overloads `+`, `-`, `*`, `/`, `%` to perform the respective operations using the helper functions.
- **Comparison Operators**: Overloads `==`, `!=`, `<`, `>`, `<=`, `>=` to compare two `BigNum` objects based on their digit representation and sign.
- **Unary Operators**: Overloads `-` to return the negation of a `BigNum` object.
- **Assignment Operator**: Overloads `=` to perform deep copying of `BigNum` objects, ensuring self-assignment is correctly handled.

Modular Arithmetic Functions

- **modAdd()**: Computes modular addition of two `BigNum` objects. The modulus is defined as (2^{\text{mod_bit_length}}). This function adds the two numbers and then applies the modulus operation.
- **modMultiply()**: Computes modular multiplication of two `BigNum` objects. It multiplies the numbers and applies the modulus.
- **modInverse()**: Computes the modular inverse of a `BigNum` object using the Extended Euclidean Algorithm. This is useful in cryptographic applications and number theory.

Implementation

The project structure is designed as follows:

The project structure is designed as follows:

```
├── src/                        // Source files
│   ├── BigNum.cpp
│   └── BigNum.h
├── bin/                        // Compiled executables
│   └── BigNumTest.exe
├── tests/                      // Test files
│   ├── test_BigNum.cpp
│   └── Makefile                // Makefile for compiling tests
└── docs/                       // Documentation
    └── Report.md
```

The `src` directory contains the implementation files for the `BigNum` library: `BigNum.cpp` and `BigNum.h`. These files define the class structure, constructors, utility functions, helper functions, operator overloading, and modular arithmetic functions.

The `tests` directory contains tests for each function of the `BigNum` library. The `test_BigNum.cpp` file contains the tests for `addtion`, `subtraction`, `modular operations`, etc., to validate the functionality of each function.

The `docs` directory contains the `Report.md` markdown file that contains the documentation and explanation of the `BigNum` library, including the introduction, class structure, implementation details, error handling, testing, and conclusion.

Testing

Exhaustive test cases were developed and run to validate the functionality of the library. The functions subjected to testing are: addtion, subtraction, multiplication, division, modulus, modular addition, modular multiplication, modular inversion.

The following are the tests that were performed:

- **Addition**: Test cases for addition of positive and negative numbers.

```
void test_addition()
{
    assertEqual(num_1 + num_2, BigNum("68056473384187692692674921486353642913"), "Addition Test 1");
    assertEqual(num_1 + num_3, BigNum("34028236692093846346452752893637505843 3"), "Addition Test 2");
    assertEqual(num_3 + num_4, BigNum("2305843009213693951"), "Addition Test 3");
}
```

- **Subtraction**: Test cases for subtraction of positive and negative numbers.

```
void test_subtraction()
{
    assertEqual(num_1 - num_2, BigNum("1"), "Subtraction Test 4");
    assertEqual(num_1 - num_3, BigNum("340282366920938463462221685927161364481"), "Subtraction Test 5");
    assertEqual(num_3 - num_4, BigNum("1"), "Subtraction Test 6");
}
```

- **Multiplication**: Test cases for multiplication of positive and negative numbers.

```
void test_multiplication()
{
    assertEqual(num_1 * num_2, BigNum("115792089237316195423570985008687907853610267032561502502920958615344897851392"), "Multiplication Test 7");
    assertEqual(num_1 * num_3, BigNum("392318858461667547739736838950479151007550136783609004032"), "Multiplication Test 8");
    assertEqual(num_3 * num_4, BigNum("1329227995784915871750885555673497600"), "Multiplication Test 9");
}
```

- **Division**: Test cases for division of positive and negative numbers.

```cpp
void test_division()
{
    assertEqual(num_3 / num_1, BigNum("0"), "Division Test 10");
    assertEqual(num_1 / num_2, BigNum("1"), "Division Test 11");
    assertEqual(num_3 / num_4, BigNum("1"), "Division Test 12");
}
```

- **Modulus**: Test cases for modulus of positive and negative numbers.

```cpp
void test_modulus()
{
    assertEqual(num_3 % num_1, BigNum("1152921504606846976") % num_1, "Modulus Test 13");
    assertEqual(num_3 % num_4, BigNum("1"), "Modulus Test 14");
}
```

- **Modular Addition**: Test cases for modular addition of positive and negative numbers.

```cpp
void test_modular_arithmetic()
{
    assertEqual(modAdd(num_3, num_4, 128), BigNum("2305843009213693951"), "Modular Addition Test 15");
    assertEqual(modAdd(num_3, num_4, 256), BigNum("2305843009213693951"), "Modular Addition Test 16");
    assertEqual(modMultiply(num_3, num_4, 128), BigNum("1329227995784915871750885555673497600"), "Modular Multiplication Test 17");
    assertEqual(modMultiply(num_1, num_2, 256), BigNum("340282366920938463463374607431768211456"), "Modular Multiplication Test 18");

    // Modular Inversion Tests
    try
    {
        assertEqual(modInverse(num_4, 128), BigNum("338953138925153547589317878866881019903"), "Modular Inversion Test 19");
    }
    catch (const std::exception &e)
    {
        std::cout << "Modular Inversion Test 19 failed: " << e.what() << std::endl;
    }

    try
    {
        assertEqual(modInverse(num_4, 256), BigNum("115790322390251417039239869215646299045894469606720753195441786934024766226431"), "Modular Inversion Test 20");
    }
    catch (const std::exception &e)
    {
        std::cout << "Modular Inversion Test 20 failed: " << e.what() << std::endl;
    }
}
```

Note - The tests were created using the Wolfram Alpha online calculator to verify the correctness of the results.

Test Results

The tests were run successfully, and all the test cases passed, validating the correctness of the `BigNum` library functions.

```
Running tests...
--------------------------
Running addition tests...
Addition Test 1 passed.
Addition Test 2 passed.
Addition Test 3 passed.
--------------------------
Running subtraction tests...
Subtraction Test 4 passed.
Subtraction Test 5 passed.
Subtraction Test 6 passed.
--------------------------
Running multiplication tests...
Multiplication Test 7 passed.
Multiplication Test 8 passed.
Multiplication Test 9 passed.
--------------------------
Running division tests...
Division Test 10 passed.
Division Test 11 passed.
Division Test 12 passed.
--------------------------
Running modulus tests...
Modulus Test 13 passed.
Modulus Test 14 passed.
--------------------------
Running modular arithmetic tests...
Modular Addition Test 15 passed.
Modular Addition Test 16 passed.
Modular Multiplication Test 17 passed.
Modular Multiplication Test 18 passed.
Modular Inversion Test 19 passed.
Modular Inversion Test 20 passed.
--------------------------
```

Compilation and Execution

The `BigNum` library can be compiled and tested using the provided `Makefile`. The following commands can be used to compile the library and run the tests from the `tests` directory:

```
make clean
make all
cd ../bin
./BigNumTest.exe
```

Discussion

Handling of Large Numbers

The library handles large numbers by storing each digit in a vector and performing operations digit by digit. This approach allows the library to manage arbitrarily large integers without overflow, unlike standard integer types. It can manage both positive and negative numbers by separately tracking the sign.

Error Handling

The library checks for division by zero error in the `divide()` function and throws an `std::invalid_argument` exception if such an errorneous attempt is made.

Also, in the `modInverse()` function, the library checks if the number is not relatively prime to the modulus and throws an `std::invalid_argument` exception if the modular inverse does not exist.

Conclusion

The `BigNum` library provides a robust framework for handling large integer arithmetic and modular arithmetic. By leveraging standard data structures and C++ features like operator overloading and exception handling, the library offers a versatile toolset for mathematical operations beyond the capacity of primitive data types. This makes it suitable for applications in cryptography, scientific computing, and anywhere large integer calculations are required.