Big Integer Library

by

Jane Alam Jan

Introduction

Often we need to write codes which can handle big integer operations. Suppose we have to add two numbers which can have 1000 digits. Then none of the usual data types would help. Though it's not that tough to code, but it's boring and tiresome.

Here we are giving a Big Integer library which has most of the operations including some mathematical and conditional operations. Mainly we have focused on some important facts.

1) The size of the code is an issue since if the size is too big then it would take so much time just to write it in real contests.
2) Negative numbers should be supported.
3) Number of digits shouldn't be an issue. If we need more digits we should allocate them dynamically.
4) Some exceptions like division by zero should be reported properly such that related coding bugs can be found.
5) The code should produce the same result as the basic C++ mathematical operators. For example, we should output -1 for (-10 % 3) which is also the result found using C++.

So, we will give the code at first. After that we will describe it fully and its merits and drawbacks.

Code

```
// header files

#include <cstdio>
#include <string>
#include <algorithm>
#include <iostream>

using namespace std;

struct Bigint {
    // representations and structures
    string a; // to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1 otherwise

    // constructors
    Bigint() {} // default constructor
    Bigint( string b ) { (*this) = b; } // constructor for string

    // some helpful methods
    int size() { // returns number of digits
        return a.size();
    }
    Bigint inverseSign() { // changes the sign
        sign *= -1;
        return (*this);
    }
    Bigint normalize( int newSign ) { // removes leading 0, fixes sign
        for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
            a.erase(a.begin() + i);
        sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
        return (*this);
    }

    // assignment operator
    void operator = ( string b ) { // assigns a string to Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse( a.begin(), a.end() );
        this->normalize( b[0] == '-' ? -1 : 1 );
    }

    // conditional operators
    bool operator < ( const Bigint &b ) const { // less than operator
        if( sign != b.sign ) return sign < b.sign;
        if( a.size() != b.a.size() )
            return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
        for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
            return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
        return false;
    }
    bool operator == ( const Bigint &b ) const { // operator for equality
        return a == b.a && sign == b.sign;
    }
```

```
    // mathematical operators
    Bigint operator + ( Bigint b ) { // addition operator overloading
        if( sign != b.sign ) return (*this) - b.inverseSign();
        Bigint c;
        for(int i = 0, carry = 0; i<a.size() || i<b.size() || carry; i++ ) {
            carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-48 : 0);
            c.a += (carry % 10 + 48);
            carry /= 10;
        }
        return c.normalize(sign);
    }
    Bigint operator - ( Bigint b ) { // subtraction operator overloading
        if( sign != b.sign ) return (*this) + b.inverseSign();
        int s = sign; sign = b.sign = 1;
        if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(-s);
        Bigint c;
        for( int i = 0, borrow = 0; i < a.size(); i++ ) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }
    Bigint operator * ( Bigint b ) { // multiplication operator overloading
        Bigint c("0");
        for( int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48 ) {
            while(k--) c = c + b; // ith digit is k, so, we add k times
            b.a.insert(b.a.begin(), '0'); // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }
    Bigint operator / ( Bigint b ) { // division operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0"), d;
        for( int j = 0; j < a.size(); j++ ) d.a += "0";
        int dSign = sign * b.sign; b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }
    Bigint operator % ( Bigint b ) { // modulo operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0");
        b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b;
        }
        return c.normalize(sign);
    }
```

```
    // output method
    void print() {
        if( sign == -1 ) putchar('-');
        for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
    }
};

int main() {
    Bigint a, b, c; // declared some Bigint variables

    ////////////////////////
    // taking Bigint input //
    ////////////////////////
    string input; // string to take input

    cin >> input; // take the Big integer as string
    a = input; // assign the string to Bigint a

    cin >> input; // take the Big integer as string
    b = input; // assign the string to Bigint b

    ///////////////////////////////////
    // Using mathematical operators //
    ///////////////////////////////////

    c = a + b; // adding a and b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a - b; // subtracting b from a
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a * b; // multiplying a and b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a / b; // dividing a by b
    c.print(); // printing the Bigint
    puts(""); // newline

    c = a % b; // a modulo b
    c.print(); // printing the Bigint
    puts(""); // newline

    ///////////////////////////////////
    // Using conditional operators //
    ///////////////////////////////////

    if( a == b ) puts("equal"); // checking equality
    else puts("not equal");

    if( a < b ) puts("a is smaller than b"); // checking less than operator

    return 0;
}
```

Documentation

Here we will describe the full idea behind the code. Mostly some parts of the codes are compact just to make it relatively small. Before going the details of the operators, we first describe the representations and structures.

Representations and Structures

We used a structure **`Bigint`** for the big integer operations. In **`Bigint`** we used two variables.

`string a:` stores the big integer digits in decimal number system. It contains digits only for both positive and negative numbers. We used **`string`** because it can hold arbitrary size. So, the number of digits will not be an issue.

`int sign:` represents whether the big integer is negative or not. If the value of **`sign`** is -1 then it contains a negative integer. Otherwise if the value is 1 then the integer can be positive or zero.

When we add, multiply or subtract integers we start from the right most digit or we can say we start from the least significant digit. After that we iterate left for rest of the calculations. That's why we store the digits in reversed form in '**`a`**'. So, to add, subtract or multiply we can start from left.

Instead of using methods, we overloaded some of the mathematical and conditional operators to simplify the usability of the code.

Constructors

There are two constructors.

`Bigint():` is the default constructor.

`Bigint( string b ):` uses assignment operator **`=`** such that the **`Bigint`** contains the string.

Some Helpful Methods

There are three methods in this section.

`size():` returns the number of digits of the Big integer. Since we store the digits in '**`a`**', so the length of '**`a`**' is actually the number of digits of the **`Bigint`**.

`inverseSign():` actually changes the sign from 1 to -1, or from -1 to 1. It has many uses. For example it's helpful for addition between a positive and a negative number. In this case we can use subtraction changing one of the numbers **`sign`**.

`normalize( int newSign ):` normalizes a **`Bigint`**. It actually removes leading zeroes from the big integer as well as it fixes the **`sign`** of the big integer according to **`newSign`**. It's normally used for subtractions or multiplications. Say we subtract 99 from 100. Usual calculations will find 01 as the result. After normalization the result will be 1.

Assignment Operator

`void operator = ( string b ):` it helps to store string b in the **`Bigint`**. What it does it quite simple. If **`b`** is negative then we store **`b[1..end]`** to **`a`**. Otherwise we store whole **`b[0..end]`** to **`a`**. After that we find the correct **`sign`** and we return the normalized **`Bigint`**.

Conditional Operators

We actually implemented two operators. We can easily add conditional operators, but to make the code compact we ignored them.

`bool operator < ( const Bigint &b ) const:` returns true if the current **`Bigint`** is less than **`Bigint b`**. We have used **`const`** since we actually used the reference of **`b`**, so, if we accidentally change **`b`** it would be a problem, but **`const`** will prevent it. This function is defined such that STL **`sort()`** can be used easily.

The idea of this function is simple. First we check the both the **`signs`**. If they are different then we can find the result based on sign. Otherwise we check numbers of digits of them. If they are different, then if they are negative numbers then the **`Bigint`** with greater size will be smaller. Otherwise if they are positive numbers then the smaller sized **`Bigint`** is smaller.

Now if their numbers of digits are also same, then we iterate from right and based on the digit and sign we find the smaller number. If they are positive numbers then the number containing smaller digit is smaller. For negative numbers the case is opposite. If the right most digits are equal then we still iterate to left until we find unequal digits.

`bool operator == ( const Bigint &b ) const:` it checks the equality of two **`Bigints`**. It's quite simple. If both the signs and the containing digits are equal then they are considered as equal.

Only these two conditional operators are listed because we can derive the other operators using these two operators. For example **`(a > b)`** can be written as **`(b < a)`**, **`(a <= b)`** can be written as **`(a < b || a == b)`**, **`(a != b)`** can be written as **`(!(a == b))`**.

If we think that you need more conditional operators to make things easier, you can easily overload some operators as well.

Mathematical Operators

Five of the main mathematical operators are overloaded here such as **+, -, *, /** and **%**.

Bigint operator + (Bigint b): returns the **Bigint** after adding **Bigint b** with the **Bigint** which initiates the operator. Here we have used the basic addition technique which we are using from childhood. Since the digits are stored in reversed fashion we start from left with no carry. After that we add digits and form carry and continue the procedure until we have no digits left. Throughout the process we saved the resulting digits to a new **Bigint** and finally we return that **Bigint**.

Since here we are thinking of both positive and negative numbers, so, before adding we see their signs. If they are different then we change one of the sign and use subtraction. Otherwise we used the addition technique we have just described. We return the normalized **Bigint** for safety and perfection. The complexity for addition is **O(n)**.

Bigint operator - (Bigint b): returns the Bigint after subtracting **b** from the operator initiator **Bigint**. The subtraction idea is also our childhood idea. We start from the leftmost digit (since we saved them in reversed order). After that we calculate borrow and iterate right. Finally we return the new normalized **Bigint**.

Again if both the signs are different then we don't have to use subtraction. We change the sign of **b** and return the addition result with **b**. The complexity for subtraction is **O(n)**.

Bigint operator * (Bigint b): returns the multiplication of the **Bigints**. The idea we have used for multiplication is a bit different than we have used it our childhood. The main theme is same but the implementation is different. The idea is that suppose we want to multiply **a** and **b**. Now we take **c = 0**. After that we start from the rightmost digit of **a**. If this digit is **k** then we add **b** to **c**, **k** times. After that we multiply **b** with **10** and we take the second rightmost digit of **a**. If this digit is **p** then we add **b** to **c**, **p** times. We continue this procedure until no digit is left in **a**. Since in **Bigint** we stored digits in reversed fashion so we start from the leftmost digit and we continue this procedure to find multiplication result. After that we normalized the result with the correct sign.

For example suppose we want to multiply 123 and 459. Now at first we say the result is 0. Now we start from taking the rightmost digit of 123. Since the rightmost digit is 3, we add 459 to result 3 times obtaining 1377 as result. Now we multiply 459 with 10 to get 4590. Now the next rightmost digit of 123 is 2. So, we add 4590 two times with 1377 to get 10557. Now again we multiply 4590 with 10 to get 45900. The last digit of 123 is 1, so, we add 45900 with 10557 to get 56457 which is the correct result. The complexity for multiplication is $O(n^2)$.

Bigint operator / (Bigint b): performs division. Division by zero can occur, that's why if this case occurs we force a division by zero operation such that the code creates the same exception.

However the idea we have used for division is a little bit different from the idea we used in school arithmetic. Say we want to divide **a** by **b**. Let **c** be the reminder throughout the process and **d** is the division result. Initially **c** and **d** both are **0**. Each time we take a digit from left of **a**. We multiply **c** with **10** and then add the digit to form the new reminder. While **c** is greater than or equal to **b** we subtract **b** from **c** and say we can subtract **k** times. Then we multiply **d** with **10** and add **k**. Then after the full iteration, **c** contains the reminder and **d** contains the division result.

For example suppose we want to divide 4567 with 12. Initially **c** and **d** both are zero. Now multiply **c** with 10 and add 4 (from **a**). Now **c** is 4 which is less than 12. So, **k** is zero and thus **d** will be zero. After that **c** will be 45. Now **b** can be subtracted from **c** 3 times. So, **k** = 3. Now **d** will be 3 and **c** will be (45 – 12 – 12 – 12) which is 9. Now in next iteration **c** will be 96 (multiplied with 10 and added the 3rd digit of **a** from left). So, we can subtract **b** from **c** 8 times. So, **k** will be 8. So, **d** will be 38 (multiplied with 10 and added 8) and **c** will be 0. Finally after adding the last digit of **a**, **c** will be 7 which is less than 12. So, **k** will be zero and **d** will be 380 (multiplied with 10 and added 0). So, the division result is 380 and reminder is 7. The Complexity for division is $O(n^2)$.

Bigint operator % (Bigint b): returns the modulo result by **Bigint b**. The procedure is same as the division method. Actually when finding the division result we find the reminder, too. The complexity is same as division, thus $O(n^2)$.

Output Method

void print(): prints the **Bigint**. Actually it prints the sign if needed, and the numbers from **string a** in reversed order. Since we have stored the numbers in reversed order, so, if we print them in reversed order we get the right result.

main() function

In main() function we have shown some examples of how to use the **Bigint**. Since the operators are overloaded, expressions like **(a * b + (d - e))** can be used where the precedence of operators will remain correct.

Drawbacks

1) Don't expect the code to be too efficient. There are faster ways to do multiplications. You may search for **Karatsuba Algorithm** for faster multiplications.
2) The base we have used is 10. So, we can use only 0 to 9. If we change the base of the numbers to store then the length of the numbers will be small. For example if the base is 10000 then 12345678 will need only two digits, where in decimal base it needs 8 digits. If the length is small then multiplications and divisions can be done faster.
3) Some parts are too compact. So, be careful when writing it.
4) We have not overloaded some operators like we haven't even added the assignment operator for integers. If you need it you may convert the integers to strings and then you can use the assignment operator.