Amir Sotoodeh
CS256
6/1/18

Final Part 1: BigNumber Library

Github:
BigNumber.cpp:
https://github.com/excisionhd/CS256/blob/master/FinalPart1/BigNumbersStaticLibrary/BigNumbersStaticLibrary/BigNumber.cpp
Static Library BigNumber.h
https://github.com/excisionhd/CS256/blob/master/FinalPart1/BigNumbersStaticLibrary/BigNumbersStaticLibrary/BigNumber.h
Dynamic Library BigNumber.h
https://github.com/excisionhd/CS256/blob/master/FinalPart1/BigNumberDynamicLibrary/BigNumberDynamicLibrary/BigNumber.h

```
/** @file BigNumber.cpp
* @author Amir Sotoodeh
* @date 5/30/18
* @brief BigNumber.cpp file is the implementation of the BigNumber class.
*/
#include "stdafx.h"
#include "BigNumber.h"
#include "math.h"
#include <string>
#include <stdexcept>
#include <iostream>
#include <sstream>
#include <iterator>

using namespace std;

/**
* DEFAULT CONSTRUCTOR: initializes a BigNumber with Digits = 0.
* @return Returns an instance of a BigNumber
*/
BigNumber::MyBigNumber::MyBigNumber() {
	digits.push_back(0);
	isNegative = false;
}

/**
* CONSTRUCTOR: initializes a BigNumber based on given string n.
* @param Single parameter, string n.
* @return Returns an instance BigNumber.
*/
BigNumber::MyBigNumber::MyBigNumber(string n) {
	string number = n;
```

```
	if (n.at(0) == '-') {
		isNegative = true;
	}
	else {
		isNegative = false;
	}

	if (isNegative) {

		number.erase(0,1);

		for (int i = number.length()-1; i >= 0; i--) {
			char c = number[i];
			int digit = c - '0';
			digits.insert(digits.begin(), digit);

		}

	}
	else {
		for (int i = number.length()-1; i >= 0; i--) {
			char c = number[i];
			int digit = c - '0';
			digits.insert(digits.begin(), digit);
		}
	}

}

//! setNegative Function
/*!
* Modifies the isNegative member variable based on input v.
* @param Single parameter, bool v.
*/
void BigNumber::MyBigNumber::setNegative(bool v) {
	isNegative = v;
}

//! getNegative Function
/*!
* Returns a boolean value based on isNegative member variable.
*/
bool BigNumber::MyBigNumber::getNegative() {
	return isNegative;
}

//! isSmaller Function
/*! Function that returns true if the first parameter is smaller than second
```

```
parameter (compares vector digits).
*  @return Returns a boolean value.
*/
bool isSmaller(vector<int> first, vector<int> second)
{
	int n1 = first.size(), n2 = second.size();

	if (n1 < n2)
		return true;
	if (n2 < n1)
		return false;

	for (int i = 0; i < n1; i++) {
		if (first[i] < second[i])
			return true;
		else if (first[i] > second[i])
			return false;
	}

	return false;
}

//! isSmallerOrEqual Function
/*! Function that returns true if the first parameter is smaller than OR EQUAL
TO second parameter.
*  @return Returns a boolean value.
*/
bool isSmallerOrEqual(vector<int> first, vector<int> second)
{
	int n1 = first.size(), n2 = second.size();

	if (n1 < n2)
		return true;
	if (n2 < n1)
		return false;

	for (int i = 0; i < n1; i++) {
		if (first[i] < second[i])
			return true;
		else if (first[i] > second[i])
			return false;
	}

	return true;
}

//! convertToString Function
/*! Function that converts a given vector into a string (helper function).
```

```
*  @return Returns the string version of the vector.
*/
string BigNumber::MyBigNumber::convertToString(vector<int> x) {
	string s1;

	for (int i = 0; i < x.size(); i++) {
		s1 += x[i] + '0';
	}
	return s1;

}

//! isASmallerString Function
/*! Function that returns true if first parameter string is greater in length
(based on numerical values).
*  @return Returns a bool.
*/
bool BigNumber::MyBigNumber::isASmallerString(string str1, string str2)
{
	int n1 = str1.length(), n2 = str2.length();

	if (n1 < n2)
		return true;
	if (n2 < n1)
		return false;

	for (int i = 0; i<n1; i++)
		if (str1[i] < str2[i])
			return true;
		else if (str1[i] > str2[i])
			return false;

	return false;
}

//! divideBy10 Function
/*! Function that divides a vector by 10.
*/
void divideBy10(vector< int > &num)
{
	int size = num.size();
	for (int i = 1; i < size; ++i)
		num[i - 1] = num[i];

	num.resize(size - 1);
}

//! less Function
```

```
/*! Function that returns true if first parameter is smaller than the second
parameter numerically.
* @return Returns a bool.
*/
bool BigNumber::MyBigNumber::less(vector< int > first, vector< int > second)
{
	bool i = 0;
	int j, leftOperandSize = first.size(), rightOperandSize = second.size();

	if (leftOperandSize < rightOperandSize)
	{
		i = 1;
	}

	if (leftOperandSize == rightOperandSize)
	{
		for (j = leftOperandSize - 1; j >= 0; j--)
		{
			if (first[j] != second[j])
			{
				if (first[j] < second[j])
				{
					i = 1;
				}
				break;
			}
		}
	}
	return i;
}

//! lessEqual Function
/*! Function that returns true if the first parameter is smaller than or equal
to second parameter (compares vector digits)
* @return Returns a bool.
*/
bool BigNumber::MyBigNumber::lessEqual(vector< int > first, vector< int >
second)
{
	bool i = 1;
	int j;
	int leftOperandSize = first.size();
	int rightOperandSize = second.size();
	if (leftOperandSize > rightOperandSize)
	{
		i = 0;
	}
	if (leftOperandSize == rightOperandSize)
```

```
	{
		for (j = leftOperandSize - 1; j >= 0; j--)
		{
			if (first[j] != second[j])
			{
				if (first[j] > second[j])
				{
					i = 0;
				}
				break;
			}
		}
	}
	return i;
}

//! computeSignAndValue Function
/*! Function that computes the sign and the value for the difference of two
big numbers.
* @return Returns a BigNumber.
*/
BigNumber::MyBigNumber
BigNumber::MyBigNumber::computeSignAndValue(BigNumber::MyBigNumber n1,
BigNumber::MyBigNumber n2) {
	MyBigNumber answer;

	bool rightIsBigger = !isSmaller(n1.digits, n2.digits);
	if (n1.isNegative != n2.isNegative) {
		if (rightIsBigger && !n1.isNegative) {
			add(n1.digits, n2.digits, answer.digits);
			answer.isNegative = false;
			answer.digits.pop_back();
		}
		else if (rightIsBigger && n1.isNegative) {
			add(n1.digits, n2.digits, answer.digits);
			answer.isNegative = true;
		}
		else if (!rightIsBigger && n2.isNegative) {
			add(n2.digits, n1.digits, answer.digits);
			answer.isNegative = false;
			answer.digits.pop_back();
		}
		else if (!rightIsBigger && !n2.isNegative) {
			add(n2.digits, n1.digits, answer.digits);
			answer.isNegative = false;
			answer.digits.pop_back();
		}
	}
```

```
	else if (isNegative&&n2.isNegative) {
		if (rightIsBigger) {
			subtract(n1.digits, n2.digits, answer.digits);
			answer.isNegative = true;
		}
		else {
			subtract(n2.digits, n1.digits, answer.digits);
			answer.isNegative = false;
		}
	}
	else {
		if (rightIsBigger) {
			subtract(n1.digits, n2.digits, answer.digits);
			answer.isNegative = false;
		}
		else {
			answer.isNegative = true;
			subtract(n2.digits, n1.digits, answer.digits);
		}
	}
	return answer;
}


//! print Function
/*! Function that prints the digits of a vector in a BigNumber.
*/
void BigNumber::MyBigNumber::MyBigNumber::print()
{
	if (isNegative == true)
		cout << "-";
	else
		cout << "";

	for (int i = 0; i < digits.size(); i++) {
		cout << digits[i];
	}
}

//! max Function
/*! Function that returns true if left parameter is greater than the right
parameter.
* @return Returns a bool.
*/
bool BigNumber::MyBigNumber::max( vector <int> & x,  vector <int> & y) {
	if (x.size() > y.size())
		return true;
	else return false;
```

```
}

//! reverse Function
/*! Function that reverses the digits of a vector; this is necessary because
arithmetic will produce a reversed vector.
*/
void BigNumber::MyBigNumber::reverse(vector <int> & z) {
      int temp, j = z.size() - 1;
      for (int i = 0; i < z.size() / 2; i++)
      {
            temp = z[i];
            z[i] = z[j];
            z[j] = temp;
            j -= 1;
      }
}

//! add Function
/*! Function that assists the operator overload + in adding two vectors.
*/
void BigNumber::MyBigNumber::add( vector <int> & x, vector <int> & y, vector
<int> & z)
      {
            vector <int> first, second;
            first = x;
            second = y;


            if (first.size() > second.size())
                  swap(first, second);

            int n1 = first.size(), n2 = second.size();
            int diff = n2 - n1;

            int carry = 0;

            for (int i = n1 - 1; i >= 0; i--)
            {
                  int sum = ((first[i]) +
                        (second[i + diff]) +
                        carry);
                  z.push_back(sum % 10);
                  carry = sum / 10;
            }

            for (int i = n2 - n1 - 1; i >= 0; i--)
            {
                  int sum = ((second[i]) + carry);
```

```
                z.push_back(sum % 10);
                carry = sum / 10;
        }

        if (carry)
                z.push_back(carry);

        reverse(z);

    }

//! subtract Function
/*! Function that assists the operator overload - in subtracting two vectors.
*/
void BigNumber::MyBigNumber::subtract( vector <int> & x,  vector <int> & y,
vector <int> & z) {
        z.clear();
        vector <int> first, second;
        first = x;
        second = y;

        if (isSmaller(first, second))
                swap(first, second);

        int n1 = first.size(), n2 = second.size();
        int diff = n1 - n2;

        int carry = 0;

        for (int i = n2 - 1; i >= 0; i--)
        {

                int sub = ((first[i + diff]) -
                        (second[i]) -
                        carry);
                if (sub < 0)
                {
                        sub = sub + 10;
                        carry = 1;
                }
                else
                        carry = 0;

                z.push_back(sub);
        }

        for (int i = n1 - n2 - 1; i >= 0; i--)
        {
```

```
                if (first[i] == 0 && carry)
                {
                        z.push_back('9');
                        continue;
                }
                int sub = ((first[i]) - carry);
                if (i>0 || sub>0)
                        z.push_back(sub);
                carry = 0;

        }

        reverse(z);

    }

//! subtractReturnVector Function
/*! Function that subtracts the digits of two vectors
* @return Returns a vector of integers
*/
vector<int> BigNumber::MyBigNumber::subtractReturnVector(vector <int> & x,
vector <int> & y) {

    string s1 = convertToString(x);
    string s2 = convertToString(y);
    vector<int> answer;

    if (isASmallerString(s1, s2))
        swap(s1, s2);


    string result = "";


    int n1 = s1.length(), n2 = s2.length();


    std::reverse(s1.begin(), s1.end());
    std::reverse(s2.begin(), s2.end());

    int carry = 0;

    for (int i = 0; i<n2; i++)
    {

        int sub = ((s1[i] - '0') - (s2[i] - '0') - carry);

        if (sub < 0)
```

```
            {
                  sub = sub + 10;
                  carry = 1;
            }
            else
                  carry = 0;

            result.push_back(sub + '0');
      }

      for (int i = n2; i<n1; i++)
      {
            int sub = ((s1[i] - '0') - carry);

            if (sub < 0)
            {
                  sub = sub + 10;
                  carry = 1;
            }
            else
                  carry = 0;

            result.push_back(sub + '0');
      }

      std::reverse(result.begin(), result.end());
      for (int i = 0; i < result.length(); i++) {
            answer.push_back(result[i] - '0');
      }

      while (answer.at(0) == 0) {
            answer.erase(answer.begin());
      }
      return answer;
}

//! OPERATOR OVERLOAD /
/*! OVERLOADING THE OPERATOR / FOR THE DIVISION OF TWO BIG NUMBERS.  EX:
MyBigNumber / MyBigNumber
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator /(MyBigNumber & x) {
      MyBigNumber quotient;
      vector<int> firstDigits = digits, secondDigits = x.digits;


      if (firstDigits.at(0) == 0 || secondDigits.at(0) == 0) {
            cout << "ERROR: DIVIDE BY ZERO" << endl;
```

```
            return quotient;
      }



      MyBigNumber one("1");

      std::cout << "Please wait, division may take awhile..." << std::endl;
      while (isSmallerOrEqual(secondDigits, firstDigits)) {
            vector<int> result = subtractReturnVector(firstDigits,
secondDigits);
            firstDigits = result;
            quotient = quotient + one;
      }

      if (isNegative != x.isNegative) {
            quotient.setNegative(true);
      }

      return quotient;
}

//! OPERATOR OVERLOAD +
/*! OVERLOADING THE OPERATOR + FOR THE ADDITION OF TWO BIG NUMBERS.  EX:
MyBigNumber + MyBigNumber
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator +(MyBigNumber & x) {
      BigNumber::MyBigNumber answer;

      bool leftIsBigger; //if the front one is bigger return true.

      //if both bignumbers are negative, then guaranteed that the answer will
be negative
      if (isNegative == x.isNegative) {
            answer.isNegative = isNegative;
            MyBigNumber::add(digits, x.digits, answer.digits);
            answer.digits.pop_back();
      }

      else {
            leftIsBigger = max(digits, x.digits);
            //if first bignumber is greater than second big number, then
first-second
            if (leftIsBigger) {
                  subtract(digits, x.digits, answer.digits);
            }
            //else second number is bigger than first number, second-first
```

```
            else {
                  subtract(x.digits, digits, answer.digits);
            }

            //assume the answer is positive
            answer.isNegative = false;

            //if first bignumber is negative and magnitude is bigger, then the
answer will be negative
            if (isNegative&&leftIsBigger) {
                  answer.isNegative = true;
            }

            //if the first big number is positive and is smaller in magnitude,
            if ((!isNegative) && (!leftIsBigger)) {
                  answer.isNegative = true;
            }
      }

      while (answer.digits.at(0) == 0) {
            answer.digits.erase(answer.digits.begin());
      }
      return answer;
}

//! OPERATOR OVERLOAD -
/*! OVERLOADING THE OPERATOR - FOR THE difference OF TWO BIG NUMBERS.  EX:
MyBigNumber - MyBigNumber
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator -(MyBigNumber & x) {

      MyBigNumber ans = computeSignAndValue(*this, x);

      while (ans.digits.at(0) == 0) {
            ans.digits.erase(ans.digits.begin());
      }
      return ans;
}

//! OPERATOR OVERLOAD *
/*! OVERLOADING THE OPERATOR * FOR THE MULTIPLICATION OF TWO BIG NUMBERS.  EX:
MyBigNumber * MyBigNumber
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator *(MyBigNumber & x) {
      MyBigNumber answer;
      string s1 = convertToString(digits);
```

```
string s2 = convertToString(x.digits);

if (isNegative != x.isNegative) {
      answer.setNegative(true);
}

int n1 = digits.size();
int n2 = x.digits.size();



if (s1 == "0" || s2 == "0")
      return MyBigNumber(0);

vector<int> result(n1 + n2, 0);

int i_n1 = 0;
int i_n2 = 0;

for (int i = n1 - 1; i >= 0; i--)
{
      int carry = 0;
      int n1 = s1[i] - '0';

      i_n2 = 0;

      for (int j = n2 - 1; j >= 0; j--)
      {
            int n2 = s2[j] - '0';

            int sum = n1 * n2 + result[i_n1 + i_n2] + carry;

            carry = sum / 10;
            result[i_n1 + i_n2] = sum % 10;

            i_n2++;
      }

      if (carry > 0)
            result[i_n1 + i_n2] += carry;

      i_n1++;
}

int i = result.size() - 1;
while (i >= 0 && result[i] == 0)
      i--;
```

```
	string s = "";
	while (i >= 0)
		s += std::to_string(result[i--]);

	for (int i = 0; i < s.length(); i++) {
		answer.digits.push_back(s[i]-'0');
	}

	while (answer.digits.at(0) == 0) {
		answer.digits.erase(answer.digits.begin());
	}


	return answer;
}

//! OPERATOR OVERLOAD ++
/*! OVERLOADING THE OPERATOR ++ FOR THE INCREMENTING OF TWO BIG NUMBERS.  EX:
++MyBigNumber
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator ++() {
	MyBigNumber answer, x("1");

	add(digits, x.digits, answer.digits);

	return answer;
}

//! OPERATOR OVERLOAD %
/*! OVERLOADING THE OPERATOR % FOR THE MODULUS OF TWO BIG NUMBERS.  EX:
MyBigNumber % MyBigNumber
* NEGATIVE NUMBERS ARE NOT SUPPORTED
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator %(MyBigNumber & x) {
	MyBigNumber quotient;
	vector<int> firstDigits = digits, secondDigits = x.digits;


	if (firstDigits.at(0) == 0 || secondDigits.at(0) == 0) {
		cout << "ERROR: DIVIDE BY ZERO" << endl;
		return quotient;
	}


	MyBigNumber one("1");
```

```
	std::cout << "Please wait, division may take awhile..." << std::endl;
	while (isSmallerOrEqual(secondDigits, firstDigits)) {
		vector<int> result = subtractReturnVector(firstDigits,
secondDigits);
		firstDigits = result;
		quotient = quotient + one;
	}

	MyBigNumber rhs = quotient * x;
	vector<int> modulus = subtractReturnVector(this->digits, rhs.digits);

	MyBigNumber answer;
	answer.digits = modulus;


	return answer;
}

//! OPERATOR OVERLOAD %
/*! OVERLOADING THE OPERATOR % FOR THE MODULUS OF A BIG NUMBER AND INTEGER.
EX: MyBigNumber / int
* @return Returns an int
*/
int BigNumber::MyBigNumber::operator %(int a)
{
	int RESULT = 0;
	string s1 = convertToString(digits);

	for (int i = 0; i < s1.length(); i++)
		RESULT = (RESULT * 10 + (int)s1[i] - '0') % a;

	return RESULT;
}
```

```
/**
* \class BigNumber
*
* \brief BigNumbers can be used in place of primitive data types
* to represent numbers that cannot normally be used in C++.
*
* \author $Author: bv Amir Sotoodeh
*
* \version $Revision: 1.0 $
*
* \date $Date: 6/1/18 $
*/
#pragma once
#include <vector>
#include <string>

namespace BigNumber

{
	class MyBigNumber

	{

	private:
		std::vector <int> digits;
		bool isNegative;
		void add(std::vector <int> & x,  std::vector <int> & y,
std::vector <int> & z);
		void subtract(std::vector <int> &x,  std::vector <int> &y,
std::vector <int> &z);
		std::vector<int> subtractReturnVector(std::vector <int> & x,
std::vector <int> & y);
		bool max(std::vector <int> & x,  std::vector <int> & y);
		void reverse(std::vector <int> & r);
		bool less(std::vector<int> l, std::vector<int> r);
		bool lessEqual(std::vector< int > l, std::vector< int > r);
		bool isASmallerString(std::string str1, std::string str2);
		MyBigNumber computeSignAndValue(MyBigNumber n1, MyBigNumber n2);
		std::string convertToString(std::vector<int> x);
		void setNegative(bool v);
		bool getNegative();


	public:
		MyBigNumber(std::string n);
		MyBigNumber();
		MyBigNumber operator +( MyBigNumber & x);
		MyBigNumber operator -( MyBigNumber & x);
```

```
        MyBigNumber operator *( MyBigNumber & x);
        MyBigNumber operator /( MyBigNumber & x);
        MyBigNumber operator ++();
        MyBigNumber operator %( MyBigNumber & x);
        int operator %(int a);
        void print();

    };

}
```

```
/**
* \class BigNumber
*
* \brief BigNumbers can be used in place of primitive data types
* to represent numbers that cannot normally be used in C++.
*
* \author $Author: bv Amir Sotoodeh
*
* \version $Revision: 1.0 $
*
* \date $Date: 6/1/18 $
*/
#pragma once

#ifdef BIGNUMBER_EXPORTS
#define BIGNUMBER_API __declspec(dllexport)
#else
#define BIGNUMBER_API __declspec(dllimport)
#endif

#include <vector>
#include <string>

namespace BigNumber

{
	class MyBigNumber

	{

	private:
		std::vector <int> digits;
		bool isNegative;
		void BIGNUMBER_API add(std::vector <int> & x, std::vector <int> &
y, std::vector <int> & z);
		void BIGNUMBER_API subtract(std::vector <int> &x, std::vector
<int> &y, std::vector <int> &z);
		std::vector<int> BIGNUMBER_API subtractReturnVector(std::vector
<int> & x, std::vector <int> & y);
		bool BIGNUMBER_API maximum(std::vector <int> & x, std::vector
<int> & y);
		void BIGNUMBER_API reverse(std::vector <int> & r);
		bool BIGNUMBER_API less(std::vector<int> leftOperand,
std::vector<int> rightOperand);
		bool BIGNUMBER_API lessEqual(std::vector< int > leftOperand,
std::vector< int > rightOperand);
		bool BIGNUMBER_API isASmallerString(std::string str1, std::string
str2);
```

```
            MyBigNumber BIGNUMBER_API computeSignAndValue(MyBigNumber n1,
MyBigNumber n2);
            std::string BIGNUMBER_API convertToString(std::vector<int> x);
            void BIGNUMBER_API setNegative(bool v);
            bool BIGNUMBER_API getNegative();

      public:
            BIGNUMBER_API MyBigNumber(std::string n);
            BIGNUMBER_API MyBigNumber();
            MyBigNumber BIGNUMBER_API operator +(MyBigNumber & x);
            MyBigNumber BIGNUMBER_API operator -(MyBigNumber & x);
            MyBigNumber BIGNUMBER_API operator *(MyBigNumber & x);
            MyBigNumber BIGNUMBER_API operator /(MyBigNumber & x);
            MyBigNumber BIGNUMBER_API operator ++();
            MyBigNumber BIGNUMBER_API operator %(MyBigNumber & x);
            int BIGNUMBER_API operator %(int a);
            void BIGNUMBER_API  print();

      };

}
```