Amir Sotoodeh
CS256
5/30/18

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;
/**
* Constructor that initializes a big number to 0.
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber::MyBigNumber() {
      digits.push_back(0);
      isNegative = false;
}

/**
* Constructor that initializes a big number to string n.
* @param n is a string
* @return Returns a BigNumber
*/
BigNumber::MyBigNumber::MyBigNumber(string n) {
      string number = n;
      if (number[0] == '-') {
```

```cpp
			isNegative = true;
		}
		else {
			isNegative = false;
		}

		if (isNegative) {
			//if negative, delete negative sign
			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);
			}
		}

	}

	/** @brief isSmaller
	* function that returns true if the first parameter is smaller than second
	parameter (compares vector digits)
	* @return Returns a bool
	*/
	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;
}

/** @brief isSmallerOrEqual
* function that returns true if the first parameter is smaller than or equal
to second parameter (compares vector digits)
* @return Returns a bool
*/
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;
}

/** @brief divideBy10
* function divides a vector by 10 (helper function)
* @return return void
*/
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);
}

/** @brief less
* function that returns true if the first parameter is smaller than second
parameter (compares vector digits)
* @return Returns a bool
*/
bool BigNumber::MyBigNumber::less(vector< int > first, vector< int > second)
{
		bool i = 0;
		int j;
		int leftOperandSize = first.size();
		int 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;
	}

	/** @brief lessEqual
	* 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;
}

/** @brief print
* function that prints each digit of the BigNumber (loops through vector)
* @return return void
*/
void BigNumber::MyBigNumber::MyBigNumber::print()
{
      if (isNegative == true)
            cout << "-";
      else
            cout << "";

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

/** @brief maximum
* if left parameter is bigger than the right parameter, return true
* @return returns a bool
*/
bool BigNumber::MyBigNumber::maximum(vector <int> & x, vector <int> & y) {
      if (x.size() > y.size())
            return true;
      else return false;
}

/** @brief reverse
* reverses a given vector (helper function)
* @return returns void
*/
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;
```

```
	}
}

/** @brief add
* a function that adds two vectors and stores into third parameter
* @return returns void
*/
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);

}

/** @brief subtract
* a function that subtracts two vectors and stores into third parameter
* @return returns void
*/
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);

}

/** @brief operator overload /
* operator overloading that divides two BigNumbers.  EXAMPLE:
BigNumber/BigNumber
* @return returns BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator /(MyBigNumber & x) {
      MyBigNumber answer;
      MyBigNumber quotient;
      MyBigNumber one("1");

      if (x.digits[0] == 0) {
            cout << "Cannot Divide by 0." << endl;
            return quotient;
      }

      vector<int> afterSubtracting = digits;

      while (isSmaller(x.digits, afterSubtracting)) {
            vector<int> hold = afterSubtracting;
            subtract(hold, x.digits, afterSubtracting);
            quotient = quotient + one;

      }



      return quotient;
}

/** @brief operator overload +
* operator overloading that adds two BigNumbers.  EXAMPLE: BigNumber+BigNumber
* @return returns BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator +(MyBigNumber & x) {
      BigNumber::MyBigNumber answer;
      answer.digits.clear();
      bool pre; //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);
      }
      else {
            pre = maximum(digits, x.digits);
            //if first bignumber is greater than second big number, then
first-second
            if (pre) {
                  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&&pre) {
                  answer.isNegative = true;
            }

            //if the first big number is positive and is smaller in magnitude,
            if ((!isNegative) && (!pre)) {
                  answer.isNegative = true;
            }
      }
      while (answer.digits[0] == 0) {
            answer.digits.erase(answer.digits.begin());
      }
      return answer;
}

/** @brief operator overload -
* operator overloading that subtracts two BigNumbers.  EXAMPLE:
BigNumber-BigNumber
* @return returns BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator -(MyBigNumber & x) {
      MyBigNumber answer;
      answer.digits.clear();
      bool pre = !isSmaller(digits, x.digits);
      if (isNegative != x.isNegative) {
            if (pre && !isNegative) {
                  add(digits, x.digits, answer.digits);
                  answer.isNegative = false;
            }
            else if (pre && isNegative) {
                add(digits, x.digits, answer.digits);
                answer.isNegative = true;
            }
            else if (!pre && x.isNegative) {
                add(x.digits, digits, answer.digits);
                answer.isNegative = false;
            }
            else if (!pre && !x.isNegative) {
                add(x.digits, digits, answer.digits);
                answer.isNegative = false;
            }
        }
        else if (isNegative&&x.isNegative) {
            if (pre) {
                subtract(digits, x.digits, answer.digits);
                answer.isNegative = true;
            }
            else {
                subtract(x.digits, digits, answer.digits);
                answer.isNegative = false;
            }
        }
        else {
            if (pre) {
                answer.isNegative = false;
                subtract(digits, x.digits, answer.digits);
            }
            else {
                answer.isNegative = true;
                subtract(x.digits, digits, answer.digits);
            }
        }
        while (answer.digits[0] == 0) {
            answer.digits.erase(answer.digits.begin());
        }
        return answer;
    }

    /** @brief operator overload *
    * operator overloading that multiplies two BigNumbers.  EXAMPLE:
    BigNumber/BigNumber
    * @return returns BigNumber
    */
    BigNumber::MyBigNumber BigNumber::MyBigNumber::operator *(MyBigNumber & x) {
        MyBigNumber answer;
        if (isNegative != x.isNegative) {
            answer.isNegative = true;
        }
```

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

string s1, s2;

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

for (int i = 0; i < x.digits.size(); i++) {
      s2 += x.digits[i] + '0';
}
if (s1 == "0" || s2 == "0")
      return answer;

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)
```

```cpp
		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[0] == 0) {
		answer.digits.erase(answer.digits.begin());
	}


	return answer;
}

/** @brief operator overload ++
* prefix operator overloading that adds 1 to a BigNumber.  EXAMPLE:
++BigNumber
* @return returns BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator ++() {
	MyBigNumber answer;
	MyBigNumber x("1");
	add(digits, x.digits, answer.digits);
	cout << answer.digits[0] << endl;
	return answer;
}

/** @brief operator overload %
* prefix operator overloading that takes the modulus of two BigNumbers.
EXAMPLE: BigNumber%BigNumber
* Unfortunately, this functionality has not been fully implemented due to
processing power constraints.
* @return returns BigNumber
*/
BigNumber::MyBigNumber BigNumber::MyBigNumber::operator %(MyBigNumber & r) {
	cout << "Unfortunately, modulus for big numbers requires heavy computing
power, please use an integer as an operand." << endl;
	return MyBigNumber();
}

/** @brief operator overload %
* operator overloading that takes modulus of a BigNumber and and int.
EXAMPLE: BigNumber%int
* @return returns BigNumber
```

```
*/
int BigNumber::MyBigNumber::operator %(int a)
{
      string s1;

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

      int res = 0;

      // One by one process all digits of 'num'
      for (int i = 0; i < s1.length(); i++)
            res = (res * 10 + (int)s1[i] - '0') % a;

      return res;
}
```

```
/** @file BigNumber.h
* @author Amir Sotoodeh
* @date 5/30/18
* @brief BigNumber.h headerfile that outlines functions used.
*/
#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);
            bool max( std::vector <int> & x,  std::vector <int> & y);
            void reverse(std::vector <int> & r);
            bool less(std::vector<int> leftOperand, std::vector<int>
rightOperand);
            bool lessEqual(std::vector< int > leftOperand, std::vector< int >
rightOperand);

      public:
            MyBigNumber(std::string n);
            MyBigNumber();
            MyBigNumber operator +( MyBigNumber & r);
            MyBigNumber operator -( MyBigNumber & r);
            MyBigNumber operator *( MyBigNumber & r);
            MyBigNumber operator /( MyBigNumber & r);
            MyBigNumber operator ++();
            MyBigNumber operator %( MyBigNumber & r);
            int operator %(int a);

            void print();

      };

}
```

```cpp
/** @file BigNumber.h
* @author Amir Sotoodeh
* @date 5/30/18
* @brief BigNumber.h headerfile that outlines functions used.
*/
#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);
		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);
	public:
		BIGNUMBER_API MyBigNumber(std::string n);
		BIGNUMBER_API MyBigNumber();
		BIGNUMBER_API MyBigNumber operator +(MyBigNumber & r);
		BIGNUMBER_API MyBigNumber operator -(MyBigNumber & r);
		BIGNUMBER_API MyBigNumber operator *(MyBigNumber & r);
		BIGNUMBER_API MyBigNumber operator /(MyBigNumber & r);
		BIGNUMBER_API MyBigNumber operator ++();
		BIGNUMBER_API MyBigNumber operator %(MyBigNumber & r);
		BIGNUMBER_API int operator %(int a);
		BIGNUMBER_API void print();

	};

}
```