CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 6

In this assignment you will create a BigInteger ADT in C++ that is capable of performing arithmetic operations on arbitrarily large signed integers. The underlying data structure for this ADT will be a List of longs. It will therefore be necessary to alter your List ADT from pa5 slightly. Fortunately, this can be done by changing a single line of code in List.h, namely line 15, which changes from

```
typedef int ListElement;
```

to

```
typedef long ListElement;
```

Make this change, then test your List again using your ListTest.cpp (and my ListClient.cpp if you like).

The BigInteger ADT will represent a signed integer by encapsulating two pieces of data: an int (which will be either 1, -1, or 0) giving its sign, and a List of non-negative longs representing its magnitude. Each List element will be a single digit in the base b positional numbering system, where b is a power of 10: $b = 10^p$. For reasons that will become clear, we restrict the exponent p to the range $1 \leq p \leq 9$. The BigInteger arithmetic operations will utilize the `long` arithmetic built into the C++ language to perform operations on single (base b) digits, and build up the standard arithmetic operations (add, subtract, multiply) out of these (base b) digit operations. The reason we chose b to be a power of 10 is to facilitate the conversion between base 10 and base b. In the case $p = 2$, we have $b = 100$, each base b digit consists of 2 base 10 digits.

$$\text{Base 100 digits} = \{00, 01, 02, \ldots\ldots\ldots, 97, 98, 99\}$$

The 52-digit base 10 number

$$N = 6523485630758234007488392857982374523487612398700554,$$

(too large for any built-in C++ data type) becomes the following List of 26 base 100 digits.

$$L = (65\ 23\ 48\ 56\ 30\ 75\ 82\ 34\ 00\ 74\ 88\ 39\ 28\ 57\ 98\ 23\ 74\ 52\ 34\ 87\ 61\ 23\ 98\ 70\ 05\ 54)$$

where we have separated digits by a space for the sake of readability. The same number in base 1,000 (i.e., $p = 3$) would have 18 digits:

$$L = (006\ 523\ 485\ 630\ 758\ 234\ 007\ 488\ 392\ 857\ 982\ 374\ 523\ 487\ 612\ 398\ 700\ 554),$$

and in base 1,000,000,000 ($p = 9$) it would have 6 digits:

$$L = (006523485\ \ 630758234\ \ 007488392\ \ 857982374\ \ 523487612\ \ 398700554).$$

Observe that to obtain the base 10 representation of such a number, we can merely concatenate the base 10 digits of each of its base b digits, then strip off any leading zeros. To go in the opposite direction and parse a string of base 10 digits into a List of base b digits, we can separate the string into groups of p characters, working from right to left. The final, leftmost, base b digit may be parsed from fewer than p characters. In all of these Lists, we regard the front as being the right end, and the back as the left. With this convention, the List index of a base b digit is the corresponding power on b. Thus

$$L = (c_{n-1}\ \ c_{n-2}\ \ \cdots\cdots\ \ c_2\ \ c_1\ \ c_0)$$

represents the number

$$N = c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + \cdots\cdots + c_2b^2 + c_1b^1 + c_0b^0.$$

It is instructive at this point to do some arithmetic examples using the above representation. To illustrate, we take $p = 2$, so $b = 100$.

Example

carry:	$1\ \ 1$		-1	
	$(35\ \ 57\ \ 97)$		$(35\ \ 57\ \ 97)$	
$+$	$(14\ \ 90\ \ 82)$	$-$	$(14\ \ 90\ \ 82)$	
	$(50\ \ 48\ \ 79)$		$(20\ \ 67\ \ 15)$	

As we can see, it is very useful to think of a borrow in subtraction as nothing more than a negative carry. One checks easily that in base 10: $355797 + 149082 = 504879$ and $355797 - 149082 = 206715$. Another way to do these examples is to add and subtract Lists as vectors, then "normalize" the results, by working from right to left in each vector, carrying and borrowing as needed to obtain a List of longs *x* in the range $0 \le x < b$, i.e., base b digits.

Example

	$(35$	57	$97)$		$(35$	57	$97)$
$+$	$(14$	90	$82)$	$-$	$(14$	90	$82)$
	$(49$	147	$179)$		$(21$	-33	$15)$
carry:	1	1	0		-1	0	0
normalize:	$(49$	147	$179)$		$(21$	-33	$15)$
		-100	-100			100	
result:	$(50$	48	$79)$		$(20$	67	$15)$

The reader is urged at this point to do a large number of such examples, since these are exactly the algorithms needed to perform BigInteger addition and subtraction. The subtraction example above (in particular the value -33) illustrates why it is useful to use `long` instead of `unsigned long` as our List element data type. It allows us to do signed arithmetic at the level of each base b digit. You should also attempt to do some multiplication problems in this representation. When multiplying, it is necessary for each digit in the first BigInteger to multiply each digit in the second BigInteger. If x and y are two such digits, then $0 \le x < b$ and $0 \le y < b$, hence $0 \le xy < b^2$. In order to guarantee that this product is always computable in type `long`, we must have b^2 no larger than the maximum possible `long` value, i.e.

$$b^2 \le 2^{63} - 1$$

and therefore

$$b \le \sqrt{2^{63} - 1} = 3{,}037{,}000{,}499.97\ldots$$

The largest such b that is also a power of 10 is $b = 1{,}000{,}000{,}000 = 10^9$, which explains why the power p on 10 must be in the range $1 \le p \le 9$.

BigInteger ADT Specifications

The BigInteger ADT will be implemented in files BigInteger.h and BigInteger.cpp. Following our standard practice, BigInteger.h will contain the definition of class BigInteger, along with prototypes of member functions, as well those of overloaded operators. The file BigInteger.h is provided on the webpage under `Examples/pa6`, and should be submitted unaltered with your project. All functions and operators in this file will be implemented by you in BigInteger.cpp. (The one exception is the destructor, which is commented out and marked as optional.)

BigInteger.cpp should define global constants called `base` and `power`, respectively, where `base` is of type `long` (i.e. type `ListElement` from List.h), and `power` is of type `int`. These constants should be defined so that $\text{base} = 10^{\text{power}}$ and $0 \leq \text{power} \leq 9$. During your testing phase, you may choose power to have any value in the range $0 \leq \text{power} \leq 9$. When you submit your project however, be sure to set $\text{power} = 9$ and $\text{base} = 1000000000$ (1 billion), for proper grading.

You will notice that the BigInteger class (as defined in BigInteger.h) has two member fields: an int (1, -1 or 0) specifying the sign of a BigInteger, and a normalized List of longs specifying its magnitude. Here *normalized* means that each List element x is in the range $0 \leq x < \text{base}$, and so constitutes a valid digit in the chosen radix. The sign 0 is reserved for the number 0, whose magnitude is represented by an empty List. This is the zero state, which is created by the no-argument constructor `BigInteger()`. The constructor from long `BigInteger(long x)` will create a BigInteger object representing the integer `x`.

The constructor from string

```
BigInteger(std::string s);
```

will create a BigInteger object representing the integer specified by `s`. The string `s` will begin with an optional sign ('+' or '−'), followed by decimal digit characters $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. If this constructor is called on a non-empty string that cannot be parsed as a signed integer, it will throw an `invalid_argument` exception (from the standard library `<stdexcept>` with the error message:

```
"BigInteger: Constructor: non-numeric string"
```

If called on an empty string, it will throw an `invalid_argument` exception with the error message

```
"BigInteger: Constructor: empty string"
```

The copy constructor, `BigInteger(const BigInteger& N)`, will simply copy the fields of `N` to `this`. The function

```
std::string to_string();
```

will return a string representation of a BigInteger as a base 10 numeral. It will be used to overload the stream insertion operator <<, as was done in the Queue and Stack examples, and in your List ADT from pa5. In a similar manner you will use

```
int compare(const BigInteger& N) const;
```

to overload the comparison operators: ==, <, <=, >, and >=. Likewise, the functions

```
BigInteger add(const BigInteger& N) const;
BigInteger sub(const BigInteger& N) const;
BigInteger mult(const BigInteger& N) const;
```

will be used to overload the arithmetic operators: `+`, `+=`, `-`, `-=`, `*`, and `*=`. This design is intended to separate the task of performing the operation (like addition or comparison) from the task of overloading the operator. Each task presents its own challenges, and it is better to face them separately. Implement the functions first, then overload the operators, *not the other way around*.

Notice that the assignment operator (`operator=()`) is left out. This is because the compiler provides a default assignment operator when `operator=()` is not defined. The default assignment operator simply does a straightforward member assignment, which is appropriate in BigInteger since none of its fields are pointers to dynamic memory. (This is also the reason that the destructor can be left out in this ADT.)

Notice that one important arithmetic operation has been left out of this exercise, namely integer division (with two outputs: quotient and remainder.) You may include functions implementing this operation if you wish, but your grade will in no way depend on it.

A judiciously chosen set of helper functions is essential to the successful completion of this project. The following set is offered as one way to split up the tasks. As you can see, all operate directly on List objects.

```
// negateList()
// Changes the sign of each integer in List L. Used by sub().
void negateList(List& L);

// sumList()
// Overwrites the state of S with A + sgn*B (considered as vectors).
// Used by both sum() and sub().
void sumList(List& S, List A, List B, int sgn);

// normalizeList()
// Performs carries from right to left (least to most significant
// digits), then returns the sign of the resulting integer. Used
// by add(), sub() and mult().
int normalizeList(List& L);

// shiftList()
// Prepends p zero digits to L, multiplying L by base^p. Used by mult().
void shiftList(List& L, int p);

// scalarMultList()
// Multiplies L (considered as a vector) by m. Used by mult().
void scalarMultList(List& L, ListElement m)
```

Since these helpers operate on Lists, and not on BigIngeters, they can and should be implemented as non-member functions, which is why they do not appear in the file BigInteger.h. In any case, these functions are all optional.

Client Module

You will write a top-level client called Arithmetic.cpp that uses the exported operations in BigInteger.cpp. This client will read from an input file containing exactly 3 lines:

line 1: an optional sign character (+ or −) followed by any number of decimal digits
line 2: a blank line
line 3: an optional sign character (+ or −) followed by any number of decimal digits

Arithmetic.cpp will open two files whose names are given on the command line.

```
$ Arithmetic <input file> <output file>
```

It will parse the string on line 1 of the input file as a `BigInteger`, calling it `A`, read and discard line 2, then parse the string on line 3 as a `BigInteger`, calling it `B`. It will then compute the following quantities and write their base 10 digits (including a negative sign, if appropriate) to the output file.

$$A,\ B,\ A+B,\ A-B,\ A-A,\ 3A-2B,\ AB,\ A^2,\ B^2,\ 9A^4+16B^5$$

These quantities will be printed in the above order, each on its own line, separated by blank lines. A set of 5 matched input-output file pairs are posted at `Examples/pa6` on the webpage for testing purposes. The same directory contains two Python programs: Arithmetic.py (our top-level client written in Python) and RandomInput5.py (which creates random matched pairs of input-output files.)

What to Turn In

You will also write your own test clients for both the List and BigInteger ADTs. As usual, these files are your scratch work, showing the tests you performed in building the two modules. Additional test clients will be posted on the webpage as (weak) tests of the two ADTs. These files are for your edification only and are not to be submitted. You will submit the following 9 files in all.

Arithmetic.cpp	Top level client for the project, described above, written by you
BigInteger.cpp	BigInteger implementation file, written by you
BigInteger.h	BigInteger header file, provided, do not alter
BigIntegerTest.cpp	Test suite for the BigInteger ADT, written by you
List.cpp	List implementation file, written by you in pa5
List.h	List header file, provided in pa5, altered slightly by you
ListTest.cpp	Test suite for the List ADT, written by you
Makefile	Included in `Examples/pa6`, alter as you see fit
README.md	Content listing of the project

This project is quite challenging, and should be started as early as possible. As usual, formulate questions and get help early and often.