CSE 101
Algorithms and Abstract Data Types
Programming Assignment 3

In this assignment you will create a BigInteger ADT capable of performing arithmetic operations on arbitrarily large signed integers. The underlying data structure for this ADT will be a List of longs. For this purpose, it will be necessary to alter your List ADT once again. The easiest way to do this is to go back to your List ADT from pa1 and change the element type from `int` to `long`. The `copyList()` function from that version should be retained, since there is now no need to make a "deep" copy.

```
        // copyList()
        // Returns a new List representing the same integer sequence as L.  The
        // cursor in the new list is undefined, regardless of the state of the
        // cursor in L. The List L is unchanged.
        List copyList(List L);
```

One new function called set() should be added to the List ADT.

```
        // set()
        // Overwrites the cursor element with x. Pre: length()>0, index()>=0
        void set(List L, long x);
```

This function allows the client to change the `long` value at the cursor position without having to call `insert()` and `delete()`. Once these small changes are made, thoroughly test the new version of List with your own ListTest client.

The BigInteger ADT will represent a signed integer by encapsulating two objects: an `int` (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 print out the base 10 representation of such a number, we need only concatenate the base 10 digits of each of its base b digits, then strip off any leading zeros. To go the opposite direction and parse a string of base 10 digits into a List of base b digits, we 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

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

represents the number

$$N = c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + \cdots\cdots\cdots + c_2 b^2 + c_1 b^1 + c_0 b^0.$$

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

Example

$$\begin{array}{lrcr}
\text{carry:} & \ \ 1\ \ \ 1\ \ \ \ \ \ \ & \qquad & -1\ \ \ \ \ \ \ \ \ \ \\
 & (35\ \ 57\ \ 97) & & (35\ \ 57\ \ 97) \\
 & \underline{+\ \ (14\ \ 90\ \ 82)} & & \underline{-\ \ (14\ \ 90\ \ 82)} \\
 & (50\ \ 48\ \ 79) & & (20\ \ 67\ \ 15)
\end{array}$$

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: $355797 + 149082 = 504879$ ann $355797 - 149082 = 206715$. 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 in the range $0 \le x < b$, i.e. base b digits.

Example

$$\begin{array}{lrrrcrrr}
 & (35 & 57 & 97) & \qquad & (35 & 57 & 97) \\
+ & (14 & 90 & 82) & & -\ (14 & 90 & 82) \\
\hline
 & (49 & 147 & 179) & & (21 & -33 & 15) \\
 & & & & & & & \\
\text{carry:} & 1 & 1 & 0 & & -1 & 0 & 0 \\
\text{normalize:} & (49 & 147 & 179) & & (21 & -33 & 15) \\
 & & -100 & -100 & & & 100 & \\
 & & & & & & & \\
\text{result:} & (50 & 48 & 79) & & (20 & 67 & 15)
\end{array}$$

The student 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. The student should also attempt to do some multiplication problems in this representation. In such an example, 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 number that is also a power of 10 is $b = 1{,}000{,}000{,}000 = 10^9$. This 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.c. Following our standard practice BigInteger.c will contain a typedef for BigIntegerObj and BigInteger.h will define the reference type BigInteger as a pointer to BigIntegerObj. It is recommended that BigInteger.c contain either global constants or `#define` constant macros for both BASE and POWER satisfying $\text{BASE} = 10^{\text{POWER}}$. During your testing phase, you may have any value $0 \le \text{POWER} \le 9$, but when you submit your project, be sure to set $\text{POWER} = 9$ and $\text{BASE} = 1000000000$, for proper grading. The BigIntegerObj should have at least two components: an int (1, -1 or 0) specifying the sign, and a normalized List of longs specifying the magnitude. (Here "normalized" means each List element x is in the range $0 \le x < \text{BASE}$.) Your BigInteger ADT will implement the following operations and types.

```
// Exported type  --------------------------------------------------------------

// BigInteger reference type
typedef struct BigIntegerObj* BigInteger;


// Constructors-Destructors ---------------------------------------------------

// newBigInteger()
// Returns a reference to a new BigInteger object in the zero state.
BigInteger newBigInteger();

// freeBigInteger()
// Frees heap memory associated with *pN, sets *pN to NULL.
void freeBigInteger(BigInteger* pN);


// Access functions -----------------------------------------------------------

// sign()
// Returns -1 if N is negative, 1 if N is positive, and 0 if N is in the zero
// state.
int sign(BigInteger N);

// compare()
// Returns -1 if A<B, 1 if A>B, and 0 if A=B.
int compare(BigInteger A, BigInteger B);

// equals()
// Return true (1) if A and B are equal, false (0) otherwise.
int equals(BigInteger A, BigInteger B);
```

```
// Manipulation procedures ----------------------------------------------------

// makeZero()
// Re-sets N to the zero state.
void makeZero(BigInteger N);

// negate()
// Reverses the sign of N: positive <--> negative. Does nothing if N is in the
// zero state.
void negate(BigInteger N);


// BigInteger Arithmetic operations -----------------------------------------------

// stringToBigInteger()
// Returns a reference to a new BigInteger object representing the decimal integer
// represented in base 10 by the string s.
// Pre: s is a non-empty string containing only base ten digits {0,1,2,3,4,5,6,7,8,9}
// and an optional sign {+, -} prefix.
BigInteger stringToBigInteger(char* s);

// copy()
// Returns a reference to a new BigInteger object in the same state as N.
BigInteger copy(BigInteger N);

// add()
// Places the sum of A and B in the existing BigInteger S, overwriting its
// current state:  S = A + B
void add(BigInteger S, BigInteger A, BigInteger B);

// sum()
// Returns a reference to a new BigInteger object representing A + B.
BigInteger sum(BigInteger A, BigInteger B);

// subtract()
// Places the difference of A and B in the existing BigInteger D, overwriting
// its current state:  D = A - B
void subtract(BigInteger D, BigInteger A, BigInteger B);

// diff()
// Returns a reference to a new BigInteger object representing A - B.
BigInteger diff(BigInteger A, BigInteger B);

// multiply()
// Places the product of A and B in the existing BigInteger P, overwriting
// its current state:  P = A*B
void multiply(BigInteger P, BigInteger A, BigInteger B);

// prod()
// Returns a reference to a new BigInteger object representing A*B
BigInteger prod(BigInteger A, BigInteger B);


// Other operations -----------------------------------------------------------

// printBigInteger()
// Prints a base 10 string representation of N to filestream out.
void printBigInteger(FILE* out, BigInteger N);
```

The zero state will be represented by the sign 0 and an empty List. A positive integer will be represented by the sign 1, and a non-empty List containing the (normalized) base b digits its magnitude. A negative integer will be represented by the sign -1 and a non-empty List containing the (normalized) base b digits of its magnitude. Notice that each arithmetic operation has two functions. The *verbs* `add()`, `subtract()` and `multiply()` and the *nouns* `sum()`, `diff()` and `prod()`. The verbs are each defined to take a first argument which is to be overwritten by the result of operating on the other two arguments. Therefore `add(S, A, B)` essentially performs the assignment $S = A + B$. These operations must be defined in such a way that different arguments may refer to one and the same BigInteger object. Thus is must be possible to assign $S = S + A$, or $S = A + S$, or even $S = S + S$, without inadvertently altering the object S during the computation. Similar comments go for the other verbs `subtract()` and `multiply()`. The nouns `sum()`, `diff()` and `prod()` are essentially constructors, since they return references to newly allocated BigInteger objects.

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 private helper functions will be very useful in this project. It is suggested that such a function to "normalize" a List of longs be included in the set.

Client Module
You will write a top level client called Arithmetic.c that uses the exported operations in BigInteger.c. This client will read from an input file containing exactly 4 lines:

line 1: a positive integer a
line 2: an optional sign character ($+$ or $-$) followed by a decimal digits
line 3: a positive integer b
line 4: an optional sign character ($+$ or $-$) followed by b decimal digits

Arithmetic.c will open such an input file, and an output file (to be overwritten if it already exists). It will parse the string on line 2 as a BigInteger A, and the string on line 4 as a BigInteger B. It will then compute the following quantities and write their base 10 digits (including a sign in the negative case) 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 matched input and output files will be posted in the Examples section of the webpage for testing purposes.

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 may be posted on the webpage as (weak) tests of the two ADTs. These files are for your edification only and need not be submitted. You will submit the following 9 files in all.

Arithmetic.c	Top level client for the project, described above
BigInteger.c	BigInteger implementation file
BigInteger.h	BigInteger header file
BigIntegerTest.c	Your test suite for the BigInteger ADT
List.c	List implementation file

List.h	List header file
ListTest.c	Your test suite for the List ADT
README	Contents of the project
Makefile	Included in Examples, alter as you see fit

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