CISS245: Advanced Programming
Assignment 11

Name: _____

Objectives

1. Design a class with supporting methods and functions
2. Design a class with object composition
3. Declare objects
4. Access member variables of an object
5. Write member functions/operators
6. Overload operators
7. Translate algorithms to code

As always read the whole document carefully before diving into coding. While many previous assignments are practice on syntax, this one requires problem solving. Start early ... and be careful.

Q1. The goal is to implement a class and functions for integer computations, not at the level of the C/C++ `int`, but rather integers with an extremely huge number of digits. We will call this the `LongInt` class. Such computations occur frequently in cryptography, physics, etc. This is, conceptually speaking, probably one of the first problem we have to solve. After all, if we can't operate with integers of any size, how can we handle real numbers, complex numbers, functions, etc.?

With this class, you will be able to compute, for instance, the factorial of 1000 (i.e. 1000!) and more. (Check the web for the factorial of 1000.) Also, you can modify your `Rational` class to use `LongInt` instead of `int` for numerator and denominator. With that, you can compute with fractions that look like this:

```
1231231241541231236547679 8 / 123598347693429857485202845
```

and you can check if

```
123123124154123123654767 91
```

is prime. This is helpful in many area of math and CS. For instance modern cryptography (example RSA cryptography) uses extremely huge prime numbers of several hundred digits in length (at this point in time). RSA cryptography depends on the difficulty of factoring numbers. In particular, it involves numbers such as

```
22601385262034057849416540486101975135080389157197767183211977681094456418 17
 96667660859312130658257725063156288667697044807000181114971186300211248792 81
 99487482066070131066586646083327982803560379205391980139946496955261
```

This is a product of two primes. The goal is to find these two primes. Clearly you can't even begin if your computer cannot work with numbers of this size. If you can factorize the above number into two primes, then ciphertext (i.e., encrypted text) that is encrypted using RSA and using the above number is completely broken and you can gain access to the encrypted data.

The fact that C/C++ natively can only handle integers with 10 digits is not a drawback of the language. No one expects a programming language to supply all the possible types and classes in the world. It's more important for a language to be extensible, i.e., that it provides features to create new types of values. That's why object-oriented programming is important. Now back to our "long integer" class.

Note that the integers in C/C++ are limited in size. Specifically an `int` value is usually about -2 billion to 2 billion. The goal of this assignment is to write a class to compute integers outside of this range. This can be achieved using arrays. For instance for the integer 1352 (the numeric concept of "one thousand, three hundred and fifty-two"), can be stored in an array, `size`, and `sign` variables:

```
x[0] = 2 x[1] = 5 x[2] = 3 x[3] = 1 size = 4 sign = 1 // sign
equals 1 means the number being modeled // is \positive"
```

and the integer -237 (the numeric concept of "negative two hundred and thirty-seven") can be modeled using:

```
x[0] = 7 x[1] = 3 x[2] = 2 size = 3 sign = -1 // sign equals -1
means the number being modeled // is \negative"
```

Note that the "ones" digit go into `x[0]`, the "10s" digit go into `x[1]`, etc. Note that 0 (the concept of "zero") can be modeled using

```
x[0] = 0 size = 1 sign = 1
```

Design a class for such objects. The name of the class is `LongInt`. For the integer array, you should use the STL `vector` class – see section below for information on the `vector` class. This is similar to our `IntDynArr` class. (The point of working on `IntDynArr` is to gain an understanding of the internal workings of `std::vector`.) Note that each `vector` object already maintains a `size` instance variable. Therefore for this class, you only need to add `sign` to the class. Hence the class should look like this:

```
class LongInt { private: std::vector< int > x; int sign; };
```

Note that we are using object composition. By the way, do not confuse the C++ STL `vector` with vectors from math or our `vec2d` class. In computer science, a dynamic array of values is also called a vector.

Note the curious syntax of the class we're using for the dynamic array of digits:

```
std::vector< int >
```

This basically tells C++ that you want a vector of `int` values. (For details refer to notes on templates.)

Now let me describe what features we want for this class ...

With your class, you should be able to do the following:

```
LongInt i; // i models 0 with default constructor std::cout << i << '\n';; //
 Prints 0 LongInt j(123); // Constructor call with int value std::cout << j <<
 '\n'; // Prints 123 LongInt k("9876543210"); // Constructor call with C-string
 std::cout << k << '\n'; // Prints 9876543210 LongInt l("-9876543210");
 std::cout << l << '\n'; // Prints -9876543210 LongInt m = l; // Copy
 constructor: m models the same // integer that l models std::cout << m <<
```

```
'\n'; // Prints -9876543210
```

The above describes the various constructors for this class. Of course you need to implement all the arithmetic operators:

```
LongInt i("1000000000000"); LongInt j(3); i += j; std::cout << i <<
'\n'; // prints 1000000000003 i -= j; std::cout << i << '\n'; // prints
1000000000000
```

[Important debugging suggestion: It's a good idea while testing your code to temporarily print your `LongInt` objects in a slightly different way:

```
LongInt i("123456789"); LongInt j("-123456789"); std::cout << i <<
'\n'; // prints < + 1 2 3 4 5 6 7 8 9 > std::cout << j << '\n'; // prints < -
1 2 3 4 5 6 7 8 9 >
```

Why? Because if the output is for instance -123 it could be that the values in the `vector` is 3, -12 and the `sign` is 1 or it could be 3, 2, -1 and the `sign` is 1. Once you know your class works correctly, you change the output back to what it should be.]

Read the following very carefully ...

Note that if your class `LongInt` class has a pointer to a dynamic array of integers in the heap for the digits (like the `IntDynArr` class) then you would need to define your own destructor, and copy constructor, and `operator=`. However we are using the STL `vector` class (which contains a pointer for the same purpose). The `vector` class already has a destructor to deallocate the memory used by the pointer in `vector` objects. Therefore when your `LongInt` default destructor is called, C++ will call the destructor of `vector` for you, doing the right thing. Likewise when you call `operator=` for your `LongInt` object like this

```
i = j;
```

where `i` and `j` are `LongInt` objects, C++ will call `operator=` on the STL `vector` object inside the `LongInt` objects, again doing the right thing. So you do not have to implement `operator=` for the `LongInt` class. (But just because I say you don't have to, it does not mean you don't have to know why.)

Other augmented operators (besides the `+=` and `-=` from above) you must have are the `*=`, `/=`, `%=`. Of course there are the binary operators `+`, `-`, `*`, `/`, `%`.

Your `LongInt` class must be able to inter-operate with `int` values too. For instance

```
LongInt i(2); i += 3; std::cout << i << std::endl; // prints 5 std::cout << (i
 + 1) << std::endl; // prints 6 std::cout << (1 + i) << std::endl; // prints 7
 int j = 2; j += i; std::cout << j << std::endl; std::cout << (1 + j) <<
 std::endl; std::cout << (1 + j) << std::endl;
```

The same applies to `-=`, `*=`, `%=`, `-`, `*`, `/`. (For this, you should refer to your `Rational` class for review and hints.)

All these operators behave in the usual mathematical way. For instance you have to be careful about carries:

```
LongInt i("10099999"); i = i + 2; std::cout << i << std::endl; // prints
 10100001
```

and you have to be careful with borrows during subtraction:

```
LongInt i("3000000000000"); i = i - 1; std::cout << i << std::endl; // prints
 2999999999999
```

To allow easy assignment for extremely long integers you must overload operator= to be able to do this:

```
LongInt i; i = "9876543210"; std::cout << i << std::endl; // prints 9876543210
 i = "-9876543210"; std::cout << i << std::endl; // prints -9876543210
```

Your class must also allow comparisons among objects and with integers using `==`, `!=`, `<`, `<=`, `>`, `>=`. For instance

```
LongInt i(123), j(74); bool b0 = (i == j); // false bool b1 = (i < 42); //
 false bool b1 = (42 <= j); // true bool b2 = (-1 == j); // false
```

Etc.

Other operators include `++` and `--`. Note that there are two types of `++` and `--`. C++ differentiates between the pre- and post-increment operators in the following way:

`++i`	same as	`i.operator++()`
`i++`	same as	`i.operator++(0)`

(Refer to the notes on operator overloading.) The difference between the pre- and post-increment operator is that the pre- version all return a reference to the object after some form of operator. Here's a simple experiment on ++ for int type:

```
int i = 0; int & j = (++i); // i becomes 1 and j references i
```

The post version actually returns a clone of the old value of `i`

```
int i = 0; int j = (i++); // i becomes 1 but j is set to 0, the original //
 value of i
```

Therefore in term of implementation in a class say C, the pre- and post-increment operators would look like this:

```
class C { public: C & operator++() // pre-increment. Note return type.  { //
 some code to change object *this return *this; } C operator++(int i) //
 post-increment. Note return type.  { C old_object = (*this); // some code to
 change object *this return old_object; } };
```

You should also implement the "negative of" operator:

```
LongInt i(123); LongInt j = -i; // j is the same as LongInt(-123)
```

Note that

```
-i same as i.operator-();
```

This is the unary "negative" operator (which is not the same as the subtraction binary operator) which is not the same as `i - j` which is `i.operator-(j)`, i.e. the two operators have different signatures.

Of course there's also the unary positive operator:

```
+i same as i.operator+();
```

(Refer to the `Rational` class for a quick review.)

You should also implement an `abs()` method that returns the absolute value of the LongInt object:

```
LongInt i(-123); LongInt j = i.abs(); // j is the same as LongInt(123)
```

For convenience, you should also have a non-member function `abs()`:

```
LongInt i(-123); LongInt j = abs(i); // j is the same as LongInt(123)
```

With all the above methods you can more or less treat `LongInt` the same as `int`. For instance you can print the factorial of 1000:

```
LongInt product(1); for (int i = 1; i <= 1000; i++) { product *= i; std::cout
 << i << ' ' << product << std::endl; }
```

The factorial of 1000 (i.e. 1000!) is an extremely long integer. You can check the correctness of your output by searching for it on the web. Of course you can even declare i to be a `LongInt` object:

```
LongInt product(1); for (LongInt i = 1; i <= 1000; i++) { product *= i;
 std::cout << i << ' ' << product << std::endl; }
```

but of course you would expect `int` variables to perform computations faster than `LongInt` objects.

Other Requirements

As always:

- Methods must be constant wherever possible
- Object parameters passed by reference whenever possible; they must also be made constant wherever possible
- Code duplication must be kept to a minimal. In general the augmented assignment operators should be implemented first, with the non-augmented ones depending on the augmented ones. Similarly, `operator!=` is just the opposite of `operator==`.
- You must have a header file `LongInt.h` and a class implementation file `LongInt.cpp`.
- You must include a test file `testLongInt.cpp` that tests all the functions and methods of your `LongInt` library.

The STL Vector Class

Although we already have an `IntDynArr` class that can be used to model our `LongInt` class, we will use the STL (standard template library) `vector` class. Note that although C++ compilers comes with the `vector` class which you can use right away, it is still important to implement your version of the `vector` class (which we call `IntDynArr`). Only then will you understand the performance differences between "container" classes for storing values – the `vector` class is not the only class used for storing values. One class might be faster in a certain scenario while another might be better in a different scenario. Also in the real world, there are situations where you have to modify the classes that come with your C++ compiler. This means that you cannot restrict yourself to being a user of classes supplied by your C++ compiler. You have to know the inner guts of these classes.

Many of the methods you have implemented in the `IntDynArr` class actually appear in this STL `vector` class. This `vector` class comes with most C++ compilers. Make sure you try the following examples to understand some methods available in this class.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector< int > v; // default constructor
    std::cout << "size:" << v.size() << '\n';

    v.push_back(5); // expand v by inserting 5 to the end of v
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << '\n';

    v.push_back(-54);
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";

    v.push_back(13542);
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";
```

```
    v.resize(5); // change the size to 5 so that we also have v[3], v[4]
    v[3] = 3;
    v[4] = 4;
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";

    v.resize(0);
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";

    v.resize(5);
    for (int i = 0; i < 5; i++) v[i] = i;
    std::vector< int > u = v; // invoke copy constructor
    std::cout << "size:" << u.size() << "    array:";
    for (int i = 0; i < u.size(); i++)
        std::cout << u[i] << ' ';
    std::cout << "\n";

    u[0] = -13579;
    u[1] = -24680;
    for (int i = 2; i < u.size(); i++) u[i] = 0;
    v = u; // Invoking the assignment operator
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";

    v.resize(2);
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << "\n";

    return 0;
}
```

You can also find information on the vector class on the web. Here are some references:

- http://www.cppreference.com/wiki/stl/vector/start
- http://www.cplusplus.com/reference/stl/vector/

It's important for you (at this point in your CS career) to be comfortable using resources on the web – without plagiarism of course – for references and to learn new things. You can learn almost anything you want as long as you have internet access. But this is especially the case for CS since CS people are heavy users of the internet. After all we did invent the internet. Therefore you will find lots of CS resources on the web. Nonetheless, it's still good to have a copy of well-written CS books. Sometimes the problem with the web is that there's TOO much information and you spend time differentiating between good and up-to-date information from mis-information.

The vector class is usually mentioned in most C++ textbooks.

Note that you should only use online resources only when you are allowed to do so. Otherwise you are plagiarizing. Plagiarism is a serious academic misconduct – think academic crime.

Unsigned Integers

In the short tutorial on STL vector class above, you would notice that the compiler might give you a warning here:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector< int > v; // default constructor
    std::cout << "size:" << v.size() << '\n';

    v.push_back(5); // expand v by inserting 5 to the end of v
    std::cout << "size:" << v.size() << "    array:";
    for (int i = 0; i ¡ v.size(); i++)
        std::cout << v[i] << ' ';
    std::cout << '\n';


    return 0;
}
```

saying that you're comparing `unsigned int` with `int`.

There are many integer types in C++. (Refer to your CISS240 notes). The `int` is the most common.

```
int x = 0;
```

In this case, assuming a 32-bit machine and compiler, your `x` can hold an integer value from -2^{31} to $2^{31}-1$, i.e., roughly -2 billion to +2 billion. An `unsigned int` is declared like this:

```
unsigned int x = 0;
```

or

```
unsigned x = 0;
```

In this case `x` can hold an integer value from 0 to $2^{32}-1$, i.e., from 0 to roughly 4 billion. That's all there is to `unsigned int`.

If `v` is a STL vector object, then `v.size()` is an `unsigned int`. In the code above `i` is declared to be an `int`. That's why there was a warning. It's not necessarily

an error. This is not an issue when `i` and `v.size()` have values in their common ranges, i.e., 0 to $2^{31} - 1$. To be absolutely safe and to avoid having compiler warning messages, you do this:

```
...

int main()
{
    ...

    for (unsigned int i = 0; i < v.size(); i++)
        std::cout << v[i] << ' ';

    ...
}
```

Test code

You are now on your own: you should design a complete test like previous assignments. Desiging test cases and trying to break your code (or someone else's) is an extremely important discipline.

For instance when you run your `main()` (in `testLongInt.cpp`) the user enters option 0 to test the constructor that accepts an int and prints the LongInt constructor with the int, the user enters option 1 to test the constructor that accepts a C-string and prints the LongInt constructor with the C-string, etc. Also, see section on Ultimate test cases.

Temporarily, you can use the following ad-hoc and incomplete test code. The test code is incomplete; you should add code to test all methods/functions to this code. The `multeq_digit()` method multiplies a digit to the object. The `multeq_tenpower()` multiplies a tenpower to object. For instance if a is `LongInt(123)`, then `a.multieq_digit(2)` will make `a` the same as `LongInt(246)` and `a.multeq_tenpower(4)` will make a the same as `LongInt(1230000)`.

`a11q01/skel/testLongInt.cpp`

```
#include <iostream>
#include <cmath>
#include "LongInt.h"

typedef LongInt Z; // see note on typedef

int main()
{
    {
        Z z;
        std::cout << "Testing constructor: You should see 0\n" << z << '\n';
    }
    {
        std::cout << "Testing constructor: You should see -20 -19 ... 19 20\n";
        for (int i = -20; i <= 20; i++)
        {
            Z z(i);
            std::cout << z << ' ';
        }
        std::cout << '\n';
    }
    {
        std::cout << "Testing LongInt(char []) ...\n";
        if (Z("0") != Z(0)) std::cout << "fail for \"0\"\n";
        if (Z("1") != Z(1)) std::cout << "fail for \"1\"\n";
        if (Z("-1") != Z(-1)) std::cout << "fail for \"-1\"\n";
```

```
        if (Z("54321") != Z(54321)) std::cout << "fail for \"54321\"\n";
        if (Z("-54321") != Z(-54321)) std::cout << "fail for \"-54321\"\n";
    }
    {
        std::cout << "Testing == and != ... " << std::endl;
        for (int i = -1000; i < 1000; i++)
        {
            for (int j = -1000; j < 1000; j++)
            {
                LongInt I(i), J(j);
                if ((i == j) != (I == J))
                {
                    std::cout << "== error for i:"
                              << i << ", " << "j:" << j << std::endl;
                }
                if ((i != j) != (I != J))
                {
                    std::cout << "!= error for i:"
                              << i << ", " << "j:" << j << std::endl;
                }
            }
        }
    }

    {
        std::cout << "Testing < ...\n";
        for (int i = -1000; i < 1000; i++)
        {
            for (int j = -1000; j < 1000; j++)
            {
                if (
                    ((i < j) != (Z(i) < Z(j))) &&
                    ((i < j) != (Z(i) < j)) &&
                    ((i < j) != (i < Z(j)))
                    )
                {
                    std::cout << "< error for " << i << ' ' << j << '\n';
                }
            }
        }
    }

    {
        std::cout << "Testing += and + ...\n";
        for (int i = -1000; i < 1000; i++)
        {
            for (int j = -1000; j < 1000; j++)
            {
                LongInt I(i), J(j);
                LongInt a = (I += J);
                LongInt b = LongInt(i) + LongInt(j);
```

```
            if ((i + j) != a || I != (i + j))
            {
                std::cout << "+= error for ";
                std::cout << i << ' ' << j << '\n';
            }
            if ((i + j) != b)
            {
                std::cout << "+ error for ";
                std::cout << i << ' ' << j << '\n';
            }
        }
    }
}

{
    std::cout << "Testing -= and - ...\n";
    for (int i = -1000; i < 1000; i++)
    {
        for (int j = -1000; j < 1000; j++)
        {
            LongInt I(i), J(j);
            LongInt a = (I -= J);
            LongInt b = LongInt(i) - LongInt(j);
            if ((i - j) != a || (i - j) != I)
            {
                std::cout << "-= error for " << i << ' ' << j << '\n';
            }
            if ((i - j) != b)
            {
                std::cout << "- error for " << i << ' ' << j << '\n';
            }
        }
    }
}

{
    std::cout << "Testing multeq_tenpower ...\n";
    for (int i = -1000; i < 1000; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            LongInt a(i);
            a.multeq_tenpower(j);
            if (i * int(pow(10, j)) != a)
            {
                std::cout << i << ' ' << j << ' ' << a << '\n';
            }
        }
    }
}
```

```
    {
        std::cout << "Testing multeq_digit ...\n";
        for (int i = -10000; i < 10000; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                LongInt a(i);
                a.multeq_digit(j);
                if (i * j != a)
                {
                    std::cout << "error for " << i << ' ' << j << '\n';
                }
            }
        }
    }

    {
        std::cout << "Testing *= and * ...\n";
        for (int i = -10000; i < 10000; i++)
        {
            for (int j = -10000; j < 10000; j++)
            {
                LongInt a(i);
                LongInt b(j);
                LongInt c = a * b;
                a *= b;
                if (i * j != a)
                {
                    std::cout << "error for *= for " << i << ' ' << j << '\n';
                }
                if (i * j != c)
                {
                    std::cout << "error for * for " << i << ' ' << j << '\n';
                }
            }
        }
    }

    {
        std::cout << "Testing pre ++ ...\n";
        for (int i = -1000000; i < 1000000; i++)
        {
            LongInt a(i);
            if (i + 1 != (++a))
            {
                std::cout << "error for " << i << '\n';
            }
        }
    }

    {
```

```
        std::cout << "Testing post ++ ...\n";
        for (int i = -1000000; i < 1000000; i++)
        {
            LongInt I(i);
            LongInt c = (I++);
            if (i != c || i + 1 != I)
            {
                std::cout << "error for " << i << '\n';
            }
        }
    }


    {
        std::cout << "Testing / and % ...\n";
        int i,j;
        for (j = 1; j < 1000000; j++)
        {
            for (i = 1; i < 1000000; i++)
            {
                if (i % 1000 == 0)
                    std::cout << "i:" << i << "  j:" << j << std::endl;
                LongInt a(i), b(j);
                LongInt q = a / b, r = a % b;
                if (i / j != q)
                {
                    std::cout << "/ error for "
                              << i << ' ' << j << std::endl;
                    return 0;
                }
                if (i % j != r)
                {
                    std::cout << "% error in "
                              << i << ' ' << j << std::endl;
                    return 0;

                }
            }
        }
    }

    return 0;
}
```

Typedef

A typedef is just a shorthand for a type. For instance support you are too lazy to type `bool`. You can do this:

```
typedef bool B; // B is the same as bool
B someflag = true;
```

It's important to know that `B` is the same as `bool`. This means for instance that you cannot have the following:

```
void f(bool);
void f(B);
```

since the two functions have the same prototype. When compared with structures:

```
struct W
{
    bool flag;
};

struct X
{
    bool flag;
};
```

the above `W` and `X` are actually different event though they have the same content. Likewise, the following classes are considered different:

```
class Y
{
private:
    bool flag;
};

class Z
{
private:
    bool flag;
};
```

Therefore it's OK to have the following:

```
void f(W);
void f(X);
void f(Y);
```

```
void f(Z);
```

You can have typedefs for arrays, pointers, and references. This is how you do it:

```
typedef int G [100];
typedef int * H;
typedef int & J;
G g;           // g is an array of 100 integers
H h = new int; // h is a pointer to an int value
J j = *h;      // j is a reference to an int value
```

Typedefs are frequently used for really long types or classes. For instance you can do this:

```
typedef LongInt Z;
Z i; // i is a LongInt object
```

SPOILERS: Advice and recursion

The following are some suggestions/advice. You need not follow them. In fact, treat them as spoilers. Look at them only when you're stuck. Also, I'll try not to give you too much help below so that you can learn from your struggles. But if you really need help, read the following and if you need more help, obviously you can talk to me or discuss with fellow students and share ideas. And remember that plagiarism – as in copying code – is not allowed.

Again, implement one small feature at a time, testing it completely before moving on to the next.

Clearly you must implement constructors first. The next thing to implement is printing. After all, if you can't print, you can't debug!

After you have read the requirements above (carefully), you might want to think about how you want to split the logic in your methods into cases before writing methods for all cases at once.

For `operator+` and `operator+=`, remember that you can easily implement `operator+` once you have `operator+=` since for almost any imaginable `class C`, once you have the copy constructor and `operator+=` working correctly, `operator+` will always look like this:

```
C operator+(const C & c) const
{
    return C(*this) += c;
}
```

So the focus is on `operator+=`. I suggest you first handle the case of adding positive `LongInt` objects. In other words test

```
LongInt(123) + LongInt(456)
```

and

```
LongInt i(123);
i += LongInt(456);
```

and not

```
LongInt i(123);
i += (LongInt(-456)
```

or

```
LongInt(123) + (LongInt(-456)
```

This means that you might (I'm not saying you must) want your `operator+=` to look like this:

```
LongInt & operator+=(const LongInt & I)
{
    if (*this is positive and I is positive)
    {
    }
    // leave the other cases out first
}
```

and if so, you might want to have an "is positive method" in your class, i.e., have your `operator>=` done first. This means that you might as well finish all the boolean operators first:

```
operator==
operator!=
operator<
operator<=
operator>
operator>=
```

(After all these are much simpler.) Once you're done with `operator+=` for positive `LongInt` objects, you should realize after some thought that adding two negative numbers is actually very easy. Why? If you try to add -123 and -456, isn't that the same as adding 123 and 456 and then sticking in a negative sign? Therefore the addition of two negative numbers actually depend on the addition of two position numbers. I'll let you think about that. The pseudocode is then:

```
LongInt & operator+=(const LongInt & I)
{
    if (*this is positive and I is positive)
    {
    }
    else (*this is negative and I is negative)
    {
        let J be the same as *this but with sign = 1
        let K be the same as I but with sign = 1
        then both J and K are positive
        perform J += K (recursion)
        copy J to *this, but set the sign of *this to -1
    }
    // other cases
}
```

But what about the addition of a positive and a negative? For instance 123 and

-456. You can actually think of that as the subtraction $123 - 456$. So in this case, it depends on `operator-=` where the two number are positive. Etc.

Altogether there are 4 cases for `operator+=` and also several cases for `operator-=`. Figure out the simple cases, write and test them, and then get the other cases to call these simple cases (recursively) and it will save you a lot of work.

SPOILERS: Multiplication

You should use the standard column multiplication algorithm which you should be very familiar with. Here's an example. Suppose you want to multiply $A = 123$ with $B = 142$, you would do this:

```
      1 2 3
x     1 4 2
------------
      2 4 6
    4 9 2
+ 1 2 3
------------
  1 7 4 6 6
------------
```

Of course this is just

```
      1 2 3
x     1 4 2
------------
      2 4 6
    4 9 2 0
+ 1 2 3 0 0
------------
  1 7 4 6 6
------------
```

(I've added zeroes.) The rows that you're adding are:

```
      2 4 6
    4 9 2 0
  1 2 3 0 0
```

And where do they come from?

```
      2 4 6 = 123 x 1
    4 9 2 0 = 123 x 4 x 10
  1 2 3 0 0 = 123 x 2 x 100
```

or

```
      2 4 6 = 123 x 1 x 10^0
    4 9 2 0 = 123 x 4 x 10^1
```

```
1 2 3 0 0 = 123 x 2 x 10^2
```

(To make things more regular looking.) If I replace the 123 by `A`, the rows become

```
  2 4 6 = A x 1 x 10^0
4 9 2 0 = A x 4 x 10^1
1 2 3 0 0 = A x 2 x 10^2
```

Of course the 1 and 4 and 2 are the digits of B.

```
  2 4 6 = A x B[0] x 10^0
4 9 2 0 = A x B[1] x 10^1
1 2 3 0 0 = A x B[2] x 10^2
```

This means that as long as you have a way to multiply your `LongInt` objects by a digit and you have a way to multply your `LongInt` object by 10 powers, you can generate the rows in your column multiplication. Suppose we call these functions (or methods – it's up to you) `MULT_BY_DIGIT` and `MULT_BY_TENPOWER`, the above is

```
  2 4 6 = MULT_BY_TENPOWER(MULT_BY_DIGIT(A, B[0]), 0)
4 9 2 0 = MULT_BY_TENPOWER(MULT_BY_DIGIT(A, B[1]), 1)
1 2 3 0 0 = MULT_BY_TENPOWER(MULT_BY_DIGIT(A, B[2]), 2)
```

Clearly this is a loop over the digits of B. Once you have your program printing out the row correctly, you can start to add them. The pseudocode is of course this:

```
sum = 0
for i = 0, 1, 2, ...:
    term = MULT_BY_TENPOWER(MULT_BY_DIGIT(A, B[i]), i)
    sum += term
```

Of course it's clear that you MUST have all the addition operations done before doing multiplication. And of course you have to make the above work for all `LongInt` objects including objects with different sizes. You also have to make sure that you can multiply integers which are negative. This is easy. For instance if you're told in school to multiply 123 and -142, you would just multiply 123 and 142 and then add a negative sign to the product.

(The high school multiplication method is actually not the only way to perform multiplication. Its been thought for a very long time that it's the most efficient way. It was only very recent – a couple of decades ago – that a more efficient method was discovered by a Russian around the 60s. This is called the Karatsuba algorithm. See CISS358.)

Division and mod %

I'll leave division and mod (i.e., `%`) to you. You only need to think about `LongInt` division for positive integers.

I have already given you the algorithm in CISS240:

$$\frac{21}{4} = 5\frac{1}{4}$$

where 5 is the integer division of 21 by 4 (also called the quotient) and 1 is the remainder, i.e., 1 is 21%4. And how do you calculate the 5 and the 1?

Pretend you have 21 dollars and you want to give that out evenly to 4 friends. Continually give out 4 dollars to each friend. At some point in time you'll be left with 1. That's the remainder. And each friend has 5 dollars. Therefore the quotient and remainder can be calculated by repeated subtraction.

There's a fastest way based on "long division" method from middle school. For instance if you want to compute quotient and remainder of 3001 by 2 using the repeated subtraction, you would have to perform 1500 subtractions of 2 from 3001 before you stop. A faster way is to realize that since 2 is so small, you can subtract 1000 of 2's. That will bring 3001 down to 1001. Right? That's the idea behind long division from middle school. At this point, your running-quotient is 1000. You can still subtract 2 from 1001. But instead of subtracting a 2 at a time, you can subtract a larger number of 2's. For instance you can subtract 500 2's from 1001 to get it down to 1. At this point your running quotient is $1000 + 500 = 1500$. Since you are now down to 1, you cannot subtract 2 from 1 and stay positive. So the quotient is 1500 and the remainder is 1. Get it? If you don't you should go online and review long division. Andif you have not been paying attention in your K-12 math classes, then you have been missing out on lots of algorithms discovered for the past 4000 years. The general idea is this:

```
INPUT: n and divisor
OUTPUT: quotient and remainder
1. Let quotient = 0.
2. Find some m such that n - m * divisor >= 0. If you cannot, go to 5.
3. Compute n = n - m * divisor and quotient = quotient + m.
4. Go to 2.
5. Return quotient and n.
```

The repeated subtraction is when m is always chosen to be 1.

I leave it to you to decide if you want to implement division and mod using repeated subtraction or long division.

Ultimate test cases

Finally add the following tests to your `main()`:

1. If the user enters test option 100, your program accepts n from the user and then computes and prints $n!$ (factorial of n). You should test your program by printing the factorial of 1000 and factorial of 5000. You can save your computation like this:

   ```
   ./a.out > factorial5000.txt
   ```

   You are strongly encouraged to check your computation with others in the class. For instance how many right-trailing zeroes are there in your 1000!? What is the 200-th digit of 5000!? Etc.
2. If the user enters test option 101, your accepts n from the user and prints the prime factorization of n. If you enter 6 for n, the output is in this format

   ```
   2 3
   ```

   i.e., print the primes in accending order. If you enter 20, the output is

   ```
   2 2 5
   ```

   You should test by factoring 3224380470287.

DIY: The RSA challenges

This is optional.

Go to https://en.wikipedia.org/wiki/RSA_Factoring_Challenge and try to factorize any of the RSA challenges. For instance here's RSA-260:

```
2211282552952966643528108525502623092761208950247001539441374831912882294140
2001986512729726569746599085900330031400051170742204560859276357953757185954
2988389587092292384910067030341246205457845664136645406842143612930176940208
46391065875914794251435144458199
```

You will need to run your program a very very very long time. Think years/centuries/millenia or longer. And you might need multiple computers. You might want to google prime factorization algorithms.