CSCI 2270, Fall 2014
HOMEWORK 2, big_number

Part 1, due Wednesday 10/29 at 11:55 pm by Moodle
Part 2, due Wednesday 11/5 at 11:55 pm by Moodle

There's a limit to how large an integer one can represent with C++'s primitive types. For this assignment, we are going to circumvent this limit in a project called big_numbers. This will introduce you to a class that uses dynamic memory allocation.

You should start by copying the Makefile, the doubly_linked_list.h and doubly_linked_list.cpp file, the header file big_number.h, the class file big_number.cpp, and the test_big_number.cpp code to your own directory (you may want to create a subdirectory for this project in your 2270 directory). Then read it over. The private (internal) representation of a big_number keeps track of the digits and the sign, and imposes no limit on the size of the number. The digits are kept in a doubly linked list. We'll begin with base-10 digits, but extend this eventually to more bases.

To compile this, go to the Geany Build menu and click Make. This runs the compiler commands in your Makefile. You should see it compile with no problems.

In big_number.h, you can find the class definition. Notice that big_numbers track 5 variables:

```
node* head_ptr;
node* tail_ptr;
unsigned int digits;
bool positive;
unsigned int base;
```

Several of the first functions in the class use dynamic memory allocation (new and delete). The first of these functions to write are the four constructors, which are like the init() methods we've used before. You can make a big_number up from scratch (think about what might be a good way to initialize such a big_number), or from a string that the user has typed in, or from a regular integer number, or from another big_number. You'll also need a destructor, which is like the destr() methods we've seen, to free up memory when you're done using a particular big_number variable. Finally you'll also need to overload the assignment operator =. In Part 1 of this assignment, you will write all the memory managing code:

```
big_number();
big_number(int num);
big_number(const string& strin);
big_number(const big_number& another_number);
~big_number();
big_number& operator=(const big_number& another_number);
```

You'll also need to write the relational operators: `==, !=, >, >=, <, and <=`. Again, once you have written the code for some of them, you can probably use those to handle the others.

```
friend bool operator>(const big_number& a, const big_number& b);
```

```
friend bool operator>=(const big_number& a,
     const big_number& b);
friend bool operator<(const big_number& a,
     const big_number& b);
friend bool operator<=(const big_number& a,
     const big_number& b);
friend bool operator==(const big_number& a,
     const big_number& b);
friend bool operator!=(const big_number& a,
     const big_number& b);
```

You'll also write input and output operators for big_numbers.

```
friend std::ostream& operator<<(std::ostream& os,
     const big_number& big_number);
friend std::istream& operator>>(std::istream& is,
     big_number& big_number);
```

Part 2.

It is true that the arithmetic functions are plentiful here, but the list shouldn't worry you too much; many of these functions can be written in terms of other ones. The hard part is usually getting addition, subtraction, and multiplication down. You should end up with a set of simple arithmetic functions for big_numbers, like `+`, `+=`, `-`, `-=`, `*`, `*=`, `++`, and `--`. For extra credit, we'll add the `factorial` function, to get some really big Big_numbers, and integer division and remainder functions as well. I solved the initial addition and subtraction work by writing two helper functions, called `sum()` and `diff()`. If your operator* function is fast, you should program the `factorial` function for extra credit.

```
big_number& operator+=(const big_number& addend);
big_number& operator-=(const big_number& subtractand);
big_number& operator*=(const big_number& multiplicand);
big_number& operator/=(const big_number& divisor);
big_number& operator%=(const big_number& divisor);
big_number& operator++();  // overload prefix increment
big_number& operator--();  // overload prefix decrement

friend big_number operator+(const big_number& addend);
friend big_number operator-(const big_number& subtractand);
friend big_number operator*(const big_number& multiplicand);
friend big_number operator/(const big_number& divisor);
friend big_number operator%(const big_number& divisor);
friend big_number factorial();
```

The test file will be the only one with a main function. It won't be nearly as comprehensive as the test file I'll use in grading, but it'll get you started. You'll probably find it advantageous to start with small big_numbers, get your routines debugged, and then test the limits with bigger ones. Also, since big_numbers are expensive in terms of memory, you'll want to pay attention to when you can free up memory a big_number is using; not doing this depletes the amount of heap memory you have available as your code runs.