CSCI 2270, Fall 2013
HOMEWORK 3, BigNum

Part 1, due Monday 10/7 at 8 am by Moodle
Part 2, due Monday 10/14 at 8 am 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 BigNums. This will introduce you to a class that uses dynamic memory allocation.

You should start by copying the header file BigNum.h, the stubs file BigNum.cxx, and the TestBigNum.cxx 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 BigNum keeps track of the digits and the sign, and imposes no limit on the size of the number. The digits array is somewhat analogous to the bag's item array. We'll actually store the digits in this array backwards, since it makes the rest of the implementation a bit simpler.

Several of these functions use dynamic memory allocation (new and delete). The first of these functions to write are the four constructors. You can make a BigNum up from scratch (think about what might be a good way to initialize a BigNum), or from a string that the user has typed in representing a decimal number, or from a regular integer number, or from another BigNum. You'll also need a destructor to free up memory when you're done using a particular BigNum variable. Finally you'll also need to overload the assignment operator =, which uses a private helper function called resize(). In Part 1 of this assignment, you will write all the memory managing code plus the get/set methods:

```
BigNum();
BigNum(int num);
BigNum(const char strin[]);
BigNum(const BigNum& anotherBigNum);
~BigNum();
void  resize(size_t n);
BigNum& operator=(const BigNum& anotherBigNum);
size_t get_digit(size_t index) const;
void set_digit(size_t digit, size_t index);
size_t get_used() const;
void set_used(size_t new_used);
size_t get_capacity() const;
bool get_positive() const;
void set_positive(bool pos_or_neg);
```

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

```
bool operator>(const BigNum& anotherBigNum);
bool operator>=(const BigNum& anotherBigNum);
bool operator<(const BigNum& anotherBigNum);
bool operator<=(const BigNum& anotherBigNum);
bool operator==(const BigNum& anotherBigNum);
```

```
bool operator!=(const BigNum& anotherBigNum);
```

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

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

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. You should end up with a set of simple arithmetic functions for BigNums, like `+`, `+=`, `-`, `-=`, `*`, `*=`, `++`, and `--`. For extra credit, we'll add the `factorial` function, to get some really big BigNums, and integer division and remainder functions here. 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.

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

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

I'll provide a small test file to start you out exercising these methods, and we'll build a Makefile to compile it in with your code. 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 BigNums, get your routines debugged, and then test the limits with bigger ones. Also, since BigNums are expensive in terms of memory, you'll want to pay attention to when you can free up memory a BigNum is using; not doing this depletes the amount of heap memory you have available as your code runs.