CIS 162 Lab 12 Fall 2018

Big Integers

Regular Java integers (i.e., `ints`) take up 32 bits of memory and, therefore, can hold only integers between -2,147,483,648 and 2,147,483,647. Java integers are only 32 bits because, at the time Java was designed, most computers were build to operate on at most 32 bits at one time. (Today, 64-bit machines are most common.) Such 32- and 64-bit operations happen very quickly in hardware. Storing and operating on larger numbers must be done using software.

Today, you are going to implement your own version of Java's `BigInteger` class. This class will allow you to store, add, multiply, compare, and print arbitrarily large integers. One common use of Java's `BigInteger` class is to handle the very large prime numbers used in cryptography.

In order to store a `BigInteger` in Java, you must represent it using a group of "smaller" variables. For this lab, you must use an array of `integers` to store your `BigInteger` Here are two ways you can use your array:

- **Base-10**: Store the integer in the same way you would write it: Each array element contains one digit between 0 and 9. Because you are representing the integer the same way you do on paper, you then perform addition and multiplication using the same technique you use to add and multiply numbers by hand. This technique is the most obvious, and will be the easiest to implement
- **Base-2^32**: Suppose you have an integer so large that it takes 70 bits to represent. You could represent this large integer using three, normal, 32 bit-integers "pushed together". When doing this, you are effectively using your array to represent your `BigInteger` as a "base 2147483648" number. The algorithms to implement some of the methods are *very* difficult (especially the constructor), but they will run much more quickly. If you choose this path, let me know and I will give you more specific instructions.

Writing `BigInteger`

Begin by cloning `https://github.com/KurmasGVSU/BigIntLab.git` (or, by creating a new project and adding `MyBigInteger.java` and `MyBigIntegerTest.java` to it). Implement *and test* the methods *in the order listed below*. This lab will be much easier to write and debug if you follow the directions. Trust me. To reduce complexity, your implementation will (1) handle only positive integers and (2) assume that all `BigIntegers` have 1024 characters (including leading zeros, if necessary).

- `BigInteger(String)`: A constructor taking a `String` as input.
- `String toString()`: Return a `String` representing the `BigInteger` object.
- `static boolean equals(BigInteger n1, BigInteger n2)`: Returns `true` if and only if `n1` is equal to `n2`.
- `static boolean lessThan(BigInteger n1, BigInteger n2)`: Returns `true` if and only if `n1` is less than `n2`.
- `static BigInteger add(BigInteger n1, BigInteger n2)`: Return a `BigInteger` equal to the sum of `n1` and `n2`.
- `static BigInteger multiply(BigInteger n1, BigInteger n2)`: Return a `BigInteger` equal to the product of `n1` and `n2`. (*extra credit*. Remove the `@Ignore` tags from the tests if you decide to attempt this method.)

Testing `MyBigInteger` Test your `MyBigInteger` class using `MyBigIntegerTest.java`. Pay attention to how each method is tested, what test cases are chosen, and what integers are chosen for each tests.

Submisson

Demonstrate to the instructor that your test script runs correctly. Then submit a printout of the code.

For More practice

Here are some exercises that will help you prepare for the Final Exam:

- `static BigInteger subtract(BigInteger n1, BigInteger n2).`
- `static BigInteger divide(BigInteger n1, BigInteger n2).`
- `static BigInteger mod(BigInteger n1, BigInteger m).`
- `double doubleValue().`
- Add the ability to handle negative numbers.
- Make the array `digits` just big enough to hold the number (as opposed to fixing the size at 1024).

Updated Sunday, 11 November 2018, 5:18 PM

W3c Validation