COMP 250 Assignment 2 (Fall 2019)

Prepared by Prof. Michael Langer

Posted: Tuesday Oct. 8
Due: Tuesday Oct 22 at 23:59 PM.

General Instructions

- The T.A.s handing this assignment are:

    - Abhisek (abhisek.konar@mail.mcgill.ca)
    - Brennan (brennan.nichyporuk@mail.mcgill.ca)
    - Charles (peiyong.liu@mail.mcgill.ca)
    - Akshatha (akshatha.arodi@mail.mcgill.ca)
    - Anand (anand.kamat@mail.mcgill.ca)

    Their extra office hours for this assignment will be announced on mycourses.

- *Do not change any of the starter code that is given to you except (1) Add code only instructed, namely* in the "ADD CODE HERE" block, and (2) add helper methods if you need them.

- You can use whatever package name you like. (Unlike in Assignment 1, here you do not need to use the default package in Eclipse.) Make sure the package name is the first line of the file. The grader code will ignore your package name declaration. To learn more about packages, see the Java tutorials.

- The starter code includes a tester class that you can run to test if your methods are correct. Your code will be tested on a more extensive and challenging set of examples. We encourage you modify this tester code and to share your tester code with other students on the discussion board. You do not need to hand in your tester code.

    Your code will be tested on valid inputs only. For example, "12000101" is an *invalid* base 2 representation. Another example is "0004753" which is *invalid* because of the leading 0's.

Submission Instructions:

Submit a single zipped file **A2.zip** that contains the modified **MyBigInteger.java** file to the myCourses assignment A2 folder. *Do not just submit the java file on its own because this will cause problems with Minerva when we download the files.* Include your name and student ID number within the comment at the top of the **MyBigInteger.java** file.

If you submit a java file or if you accidently submit the starter code or a class file instead of a valid java solution file, you will receive a 0. We will try to detect such mistakes before the deadline so that you have a chance to resubmit before the deadline. However, we do not guarantee this.

If you submit late, then you will receive a late penalty. It is your responsibility to submit correctly in the first place.

We will remove all package declarations and import statements and add back only the imports that are given in the starter code. Therefore if you use other imports, then your code will not compile when we test it.

You will receive 0 for a submission that does not compile.

If you have any issues that you wish the TAs (graders) to be aware of, include them in the comment field in mycourses along with your submission. *Otherwise leave the mycourses comment field blank.* Please do not write things like "Here is my submission for A1" with student name and ID number. We already know this info.

You may submit multiple times, e.g. if you realize you made an error after you submit. Unless we announce otherwise, your grade will be determined by your most recent submission.

Late assignment policy:

Late assignments will be accepted up to only 2 days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So make sure you are nowhere near that threshold when you submit.

Please see the Course Outline (updated on Oct. 8) regarding policy on late submissions.

Introduction

Computers represent integers as binary numbers (base 2), typically using a relatively small number of bits e.g. 16, 32, or 64. In Java, integer primitive types *short, int* and *long* use these fixed number of bits, respectively. For certain applications such as in cryptography, however, one needs to work with very large positive numbers and do arithmetic operations on them. For such applications, it is necessary to go beyond the primitive type representation.

How can one represent a very large positive integer? For any base, one can represent *any* positive integer *m* uniquely as the sum of powers of the base. This defines a polynomial:

$$m = \sum_{i=0}^{n-1} a_i \ base^i = a_0 + a_1 \ base + a_2 \ base^2 + \ldots + a_{n-1} base^{n-1}$$

where $base$ is some number (e.g. 2, 10), and the coefficients a_i satisfy $0 \leq a_i < base$ and $a_{n-1} > 0$. Note that the condition $a_{n-1} > 0$ is required for a unique representation. Also note that when a positive integer m is represented as a list of coefficients $(a_0, a_1, a_2, \ldots a_{n-1})$, the ordering of the coefficients is opposite to the usual ordering that we use to write out the number, namely $a_{n-1} \ldots a_2, a_1, a_0$. For example, the integer 35461 is represented as a list of coefficients (1,6,4,5,3).

In this assignment, we will work with arithmetic operations on large positive integers. Java has built-in class for doing so, called **BigInteger**. You are not allowed to use this class. Instead you will work with a partially implemented class **MyBigInteger**. Whereas the Java class **BigInteger** can be used for negative and positive, our class **MyBigInteger** only allows us to represent non-negative integers.

The **MyBigInteger** class has two fields: **base** and **coefficients.** The **base** field is an **int** with values in {2, 3, …, 10}. We could have allowed for larger bases but that would have required using special symbols for the numbers greater than 10, e.g. as in hexadecimal, and these extra symbols would just complicate things. The **coefficients** field is an **ArrayList<Integer>** which represents the a_i values above.

We provide you an implementation of three of the four arithmetic algorithms that you learned in grade school and that were discussed in lecture 1, namely **plus(), times(), minus().** We also include slow versions of **slowTimes()** and **slowdividedBy()** which implement the slow multiplication and slow division algorithms that were mentioned in class.

We include several helper methods as well.

- **timesBaseToThePower()**
- **mod()**
- **clone()**
- **compareTo()**
- **toString()**
- **primesToN()**.

For all of the methods, you should read the comments in the code to see what the methods do.

You are also given code stubs of the methods that you are required to implement, and a **Tester** class with a simple example. Feel free to modify this **Tester** class as you wish.

What will you learn by doing this assignment?

There are several learning goals for this assignment.

First, you will understand much better how grade school arithmetic algorithms work and hopefully understand their time complexity. You have been using arithmetic operations of +, - , *, / since you were a child and so you take them for granted. After doing this assignment, you should understand these algorithms much better than you did before, in particular, the division algorithm which you are asked to implement.

Second, the assignment will help understand how to represent numbers in different bases. The definition of this representation was given on the previous page. But there are some subtleties in that definition that arise. For example, converting a number from one base representation to another can be tricky since the base itself that is used for the conversion method is a number that needs to be represented in some base.

Third, one of your tasks is to work with prime numbers. Although prime numbers will arise only occasionally in this course, they are extremely important in some application areas of computer science such as in cryptography. Those of you taking MATH 240 Discrete Structures 1 or MATH 346 Number Theory will be working with prime numbers. Your experience here should complement what you will learn in those courses.

Fourth, you will get some experience working with lists. In particular, you will use methods from the Java `ArrayList` class.

Lastly, this assignment will also give you more practice programming in Java! Although COMP 250 is not a course about how to program, programming *is* a core part of computer science and the more practice you get, the better you will become at it.

Your Tasks

Implement the following methods. The signatures of the methods are given in the starter code. You are *not* allowed to change the signatures.

Question 1: dividedBy (30 points)

Implement the **dividedBy** method using long division. An example of long division is given in the lecture 1 notes PDF. This example gives the main idea of how the algorithm works.

Before attempting to write any code, study the implementations of the **plus(), times(), minus()** and the various helper methods that are given to you. Make sure you understand about how they work, since similar ideas can be used to implement **dividedBy()**.

If you are unable to solve this question first and you wish to work on Questions 2 and 3 in the meantime, then you may simply call the **slowdividedBy()** method from within the **dividedBy()** instead. However, note that you will only be able to run your solution for small numbers.

We will test your **dividedBy()** method on moderate size numbers, with the intention of detecting relatively slow implementation. The time complexity of your **dividedBy()** method should be $\boldsymbol{O}(\boldsymbol{N^2})$ whereas the time complexity of **slowdividedBy()** is $\boldsymbol{O}(\boldsymbol{base^N})$, where $\boldsymbol{N}$ is the number of digits of the dividend. If your implementation runs extremely slowly, e.g. if you were to just copy the code from the **slowdividedBy()** method, then you will not receive any points for this question.

Question 2: Base conversion (40 points)

Implement a method **convert(int newBase)** that converts a **MyBigInteger** object from one base to another. The **convert** method is called by a **MyBigInteger** object that represents a positive integer in some base, and returns the same positive integer represented in the new base. The bases can be any of {2, 3, 4, …, 10}.

Begin by testing your methods on numbers that are written in base *10*. Once that is working, test it on combinations of different bases from 2 to 10. Use an online converter to verify your answers e.g. http://www.cleavebooks.co.uk/scol/calnumba.htm for small numbers. Your code will be tested on very big numbers.

Hint: You may wish to handle three cases separately, depending on whether the original base is smaller than, equal to, or larger than the new base.

Question 3: Prime Numbers (30 points)

A prime number is a positive integer $m > 1$ whose factors are only 1 and itself. So, a number m is prime if dividing it by any number in {2, …, $m - 1$} produces a non-zero remainder. By definition, $m = 1$ is not prime.

Any positive integer m can be written uniquely as a product of primes:

$$m = {p_1}^{\alpha_1} {p_2}^{\alpha_2} \ldots {p_k}^{\alpha_k}$$

where p_i are called *prime factors* and $\alpha_i > 0$ are the non-zero *orders* of the prime factors. For example, $24 = 2^3\ 3^1$.

One can determine if a number m is prime by brute force checking all the integers from 2 up to $m - 1$ to see if any of them divides evenly with no remainder. However, this method is very inefficient. It is enough to check potential factors less than or equal to $\sqrt{m}$. Think why.[1]

Your task is to implement a method **primeFactors().**

The method must return an **ArrayList<MyBigInteger>** whose elements are the prime factors of m, where m is 'this' **MyBigInteger** object. The number of copies of the prime factor in the returned list must be the order of that prime factor, and the prime factors in the list must go from smallest to largest. For example, if m = 24, the method must return a list (2, 2, 2, 3). If m is prime or if the method cannot find any prime factors, then the returned list will just contain one element, namely, the returned list will be (m). You do not need to handle the case $m = 0, 1$.

For testing purposes, you may wish to have a list of prime numbers. We provide you with efficient code for computing them. The code is an implementation of an ancient algorithm called the Sieve of Eratosthenes which computes all the primes up to some given limit, say n. This code is given to you as a helper method **primesToN()**. The maximum value of n that you can use is ultimately limited by the size of the JVM's memory. One way you can use this code is to generate large prime numbers, and then multiply these large prime numbers together to yield a very large non-prime number. (Any number that is a product of at least two primes is called a *composite* number.) This process is closely related to problems in cryptography where one person multiplies two large prime numbers together, and the other person is given the product and tries to compute what the two original (prime) numbers were.

We will run your **primeFactors()** method on some very big composite numbers. If your method does not complete within an allocated time, then it will deemed to fail the stress test. Note that such a failure may be only due to an inefficient **dividedBy()** method. In this case, you will be double penalized for having a slow **dividedBy()** method. So make sure your **dividedBy()** method passes its stress test.

---

[1] Since you are only looking for prime factors, in theory you only *need* to check potential factors that are themselves prime. But to only check prime factors, you would need to know in advance which potential factors are prime, which would require some work in itself. For this assignment, we don't require you to restrict your checks in this way.

Other Notes

The solution that you submit will be tested and graded automatically. However, it sometimes happens that the TA/grader needs to examine the code. In this case, it is helpful if you have added comments to describe what your solution is doing. If we need to check your code and we find it is unreadable, then we reserve the right to penalize you for this.

Eclipse does proper indentation automatically, as do other excellent IDEs. So please use it.

Be sure to use Java naming conventions for variable names. In particular, variables and method names should be mixed case with a lower case first letter.

Plagiarism

The solution that you submit must be your own work. You are expected to write the required methods by yourself, that is, without copying anyone else's coded solution. As stated in the Course Outline, we will run software that examines the similarity of pairs of submissions. If two student submissions are unusually similar, then the software will flag this. The TA's and the instructor will then compare the code by hand and if plagiarism is suspected then the case will be reported to a Disciplinary Officer. See the Course Outline for further details.

Get started early! Have fun! Good luck!