COMP 250 Assignment 1

Prepared by Prof. Michael Langer

Posted: Tues, Sept 19, 2017
Due: Tues Oct 3 at 23:59 PM.

General Instructions

- The T.A.s handing this assignment are
  - Ram ramchalam.kinattinkararamakrishn@mail.mcgill.ca
  - Navin navin.mordani@mail.mcgill.ca

  Their office hours and location will be announced on mycourses.

- *Do not change any of the starter code that is given to you. Add code only where instructed, namely* in the "ADD CODE HERE" block. You may also add helper methods.

- You can use whatever package name you like. However, make sure the package name is the first line of the file. The tester 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. Try to identifiy tricky cases. Do *not* hand in your tester code.

  Your code will be tested on valid inputs only. For example, "12000101" and base 2 is not a valid since a base 2 representation only contains bits 0 and 1. Another example is that "0004753" is considered not valid (input or output) for any base because of the leading 0's.

- **Submission Instructions:**

  Submit a single zipped file **A1.zip** that contains the modified NaturalNumber.java file to the myCourses assignment A1 folder. Include your name and student ID number within the comment at the top of the NaturalNumber.java file.

  - If you submit a java file (rather than zip) or a rar file or if you accidently submit the starter code or a class file instead of a valid java solution file, you will be asked to resubmit and you will be penalized.
  - You will receive 0 for a submission that does not compile. (We will provide the package name.)

- 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.*
- You may submit multiple times, e.g. if you realize you made an error after you submit. We will automatically grade the most recent valid submission.

- **Late assignment policy:**

  Late assignments will be accepted up to only 3 days late and will be penalized by 20 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.

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. Using a fixed number of bits for integers limits the range of values that one can work with, however. This limitation is a significant problem in cryptography, for example, where one uses large integers to transform files so that they are encrypted.

For any base, one can represent any positive integer *p* uniquely as the sum of powers of the base. This defines a polynomial:

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

where the coefficients a_i satisfy $0 \leq a_i < base$ and $a_{n-1} > 0$. The last condition is important for uniqueness and comes up below in the **Tips** where we briefly discuss the case that two operands have a different number of digits.

The positive integer p can be 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 represent numbers, namely $a_{n-1} \ldots a_2, a_1, a_0$ e.g. integer 35461 is represented as a list of coefficients (1,6,4,5,3).

In this assignment, you will implement basic arithmetic operations on large positive integers. Java has class for doing so, called **BigInteger**. You will implement your own version of that class. In particular, you will implement basic arithmetic algorithms that you learned in grade school. Your representation will allow you to perform these operations in any base from 2 to 10. The methods could be extended to larger bases but we will not bother since it would require symbols for the numbers {10, 11, ..} and otherwise the solution would be the same.

You are given a partially implemented **NaturalNumber** class. The class has two private fields: **base** and **coefficients.** The coefficients a_i of the number are represented using the **LinkedList<Integer>** class. Coefficients order was described above. The starter code for the class also contains:

- code stubs of the methods that you are required to implement.

- helper methods that you are free to use, namely **clone(), timesBaseToThePower()**, **compareTo()**, **toString()**. You are not allowed to modify these helper methods.

- a **Tester** class with a simple example. Modify this example to test your code.

Your Task

Implement the following methods. We suggest you implement them in the order given. The signatures of the methods are given in the starter code. You are *not* allowed to change the signatures.

1) plus (30 points)

Implement the grade school addition operation.

We call it **plus** rather than **add** to avoid confusing it with the **add** method for lists. It is the easiest of the four methods to implement.

2) times (30 points)

Implement the grade school multiplication method. *Do not store the rows in a 2D table.* Instead, when you compute each row in the table, add it to an accumulated sum (using use the **plus()** method) which is initialized to zero. Once you have added a row, you can discard it. To compute each row, we suggest that you write a helper method **timesSingleDigit()** that multiplies the caller NaturalNumber object with a single digit.

The starter code provides a slow multiplication method which uses repeated addition. You should use this method to verify the correctness of your **times()** method for small operands. Note how slow the 'slow method' is relative to grade school multiplication for large operands.

3) minus (25 points)

Implement the grade school subtraction method. The starter code verifies **a.minus(b) > 0**, so you can assume this in your tests.

Although this question is worth fewer points, it is perhaps more challenging than the first two, because it can be tricky to handle the borrowing properly.

4) divide (15 points)

Implement the grade school long division algorithm which is a fast algorithm for performing integer division. This is the most challenging question since you will need to figure out for yourself what this algorithm does.

We have provided you with a slow division method which is based on repeated subtraction. This is mainly to verify the correctness of your code on small inputs.

Other Requirements

Use Java naming conventions for variable names e.g. variables and method names should be mixed case with a lower case first letter.

Although your solution will be tested and graded automatically, 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. We reserve the right to penalize you for poor style e.g. non-existing or unhelpful comments, or improper indentation. Eclipse does proper indentation automatically, as do other excellent IDEs.

Tips

We suggest that you begin by testing your code on numbers that are written in base *10*. Once that is working, test it on bases 2 to 9. Use an online converter to verify your answers e.g.

http://www.cleavebooks.co.uk/scol/calnumba.htm

You may write your own helper methods, but if you do then you must be sure to document them, so that the TA grader it can easily follow what you are doing.

Some methods in the starter code "clone" the operands and work with the cloned ones rather than the original ones. This is done because the methods change the operands. For example, some methods are just easier to implement if the two operands have the same number of digits. So, if in some instance the two operands don't have the same number of digits, then we clone and modify the smaller one by appending higher order digits with value 0. Second, a method might change the digits of one of the operands. For example, the subtraction operation **a.minus(b)** may require "borrowing" from higher to lower powers for the number **a**.

The code uses a **LinkedList** rather than an **ArrayList**. There is no significance to this, and we could have used an **ArrayList** instead. The advantages of using the **LinkedList** class is that it has methods **addFirst()** and **addLast()** which make code easier to read. Also, the method **addFirst(element)** is more efficient for **LinkedList** than the corresponding **add (0, element)** for an **ArrayList**. The disadvantage of using **LinkedList** is that the algorithms sometimes iterate over the coefficients, e.g. using a **for** loop over **coefficients.get(index)** with an **index** variable, and this is an inefficient way to iterate through linked lists, as I've argued in the lecture. This an important issue in practice and you should think about it when studying array lists versus linked lists, but we will ignore this particular inefficiency for the assignment.

Get started early! Have fun! Good luck!