Assignment 8. Implementing type integer

Due on Tuesday, 4 December, by midnight (submitted electronically). Download class List and a skeleton of class BigInt from the assignment subpage of the course website.

Introduction. As you know, types **int** and **long** deal only with a finite set of the integers. In this assignment, we ask you to implement type integer, which contains all the integers, in a class BigInt. This will provide you with a more thorough understanding of representing integers in different bases. You will use type BigInt to do a calculation or two.

Each instance of BigInt contains an instance of class List, which contains the unsigned integer. Use **our** class List. Also, we provide a skeleton of BigInt, with all the methods that you need defined in it and with many methods filled in. This simplifies your task a great deal.

Maintaining an integer in some base. Let base b be an integer, 2 <= b. Then a positive integer n can be written as

$$n = n_0*b^0 + n_1*b^1 + \ldots + n_{k-1}*b^{k-1} + n_k*b^k$$

where:

(0) Each n_i satisfies $0 <= n_i < b$

(1) The high-order digit n_k is not 0

The integer 0 is represented by $0*b^0$, or 0.

For example, the integer 341 in decimal is

$$341 = 1*10^0 + 4*10^1 + 3*b^2$$

Here, k is 2.

Comparing integers. Given the base b representations of nonnegative m and n, the expression n < m can be evaluated as follows. If one is longer than the other, the shorter one is smaller. Otherwise, compare n_0 and m_0 , then n_1 and m_1, then n_2 and m_2 , etc.; at each step, maintain a variable d that is -1, 0, or 1, depending on whether the part of n that has been compared thus far is less than, equal to, or greater than the part of m that has been compared.

Addition. Addition in base b is just like addition in base 10, e.g. we evaluate 5432+649 as shown below, where the top line represents the carry from the previous (lower-order) digit.

```
  0 1 0 1      (this is the carry)
    5 4 3 2
+     6 4 9    DONE IN BASE 10
    6 0 8 1
```

Thus, 2+9 = 11, which is treated as 1 with a carry of 1.

Then, 1+3+4 = 8, which is treated as 8 with a carry of 0.

Then, 0+4+6 = 10, which is treated as 0 with a carry of 1.

Then, 1+5=6, which is treated as 6 with a carry of 0.

Note that the digits are processed from low order to high order.

When carrying this out in base b, suppose the carry plus two digits sums to s. Then the value for that position is s%b and the carry is s/b.

Subtraction. Subtraction of nonegative integers is similar. Here, we always subtract the larger integer from the smaller, so that a nonnegative integer results. This requires comparing the integers before doing the subtraction. And, when subtracting, there may be a "carry" of -1, as shown below.

```
  -1 -1 -1      (this is the carry)
    5 4 3 2
-   6 2 9       DONE IN BASE 10
    4 7 0 3
```

For a particular position, suppose the carry plus the first digit minus the second digit is s. If s>=0, then the value for that position is s and the carry is 0. If s<0, then the value for that position is s+b (where b is the base) and the carry is -1.

The carry from the high-order position is always 0, because the smaller integer is subtracted from the larger one.

There is often a need to eliminate leading 0's. For example, using the scheme above, 5000-4999 = 0001, and this has to be changed to 1.

Class List. We use class List from the previous assignment. Use our version (from the course web site), because we have added some methods to it (e.g. size()). Before you begin, become familiar with the methods in class List and their specifications.

Class BigInt. Before you implement anything, read the rest of this handout completely! Then, when writing each method, CHECK IT THOROUGHLY, USING ENOUGH TEST CASES TO BE SURE THAT IT IS CORRECT! This is the only way to deal with such a programming task.

Note: a method that returns a BigInt should always produce a new instance of BigInt. It should NOT change the BigInts that are input to it. The user should be able to rely on inputs to functions not being changed!

Look at the beginning of class BigInt. BASE is the base that is being used, sign gives the sign of the integer, and bigint is the unsigned integer itself. Read the description of these variables carefully. Since each digit of a number is stored as an **int**, the BASE is restricted to Short values so that no overflow can occur --because we provide multiplication, BASE*BASE has to be an **int**.

When testing your work, use base 10 at first, so that you can see the output nicely. Then try it in smaller and larger bases.

Make use of method toStringBASE to print results for yourself. It does not rely on any of the methods that you are writing.

Method BigInt.toString may not work properly until you have implemented some of your methods.

WATCH OUT. We spent an hour looking for the mistake in

```
if (this.sign = b.sign)
```

This compiled, even though the = should have been ==. Evaluation of the if condition stores b.sign in this.sign and then yields the value of this.sign. Horrible!

Note: Although (most) integers are kept in base BASE, there will be a need in one place to keep an integer in a different base. That is why method encode has a parameter.

The constructors and function encode. We give you these, so you can have an idea of what the methods that deal with integers look like. Method encode implements an algorithm that we have seen on an assignment. STUDY IT.

1. Method equals. Write (and test) this method.

2. Methods less. Write static method less.We give you instance method less, so that you have an idea how signs of integers are treated.

3. Method negate. Write the body of method negate. Remember, don't change the BigInt that it (implicitly) works with; create a new one and fill in its sign and bigint fields. Also, remember that a BigInt that is 0 must have **true** for its sign.

4. Static method add. Write static method add, which adds two unsigned integers (represented as Lists) and produces the result in a third List. Make use of the conventional way of adding, as described earlier.

Don't go on to the next step until you are positive that add is correct. You can test this method by making it public (temporarily) and calling it from your method main with suitable arguments. Be sure to make it private when you are finished.

To help you out, we have included the invariant for the first loop of our algorithm, which processes the two lists that represent the integers until one of them becomes empty.. You don't have to use the idea given by this invariant if you don't want. But, IF YOU DON'T USE THIS INVARIANT, DELETE IT!

5. Instance method add. Write instance method add. Depending on whether the signs of **this** and b are the same or different and on which is larger, it will have to call add(this.bigint, b.bigint), subtract(this.bigint, b.bigint), or subtract(b.bigint, this.bigint). The calls on subtract will not work properly because you haven't written subtract. So at this point, you can test this method only on integers with the same sign.

6. Static method subtract. Write static method subtract. As with add, we give an invariant, which you can use OR DELETE. Note also the last statement; knowing that you will generally forget to remove leading zeros from the result, we have done it for you.

7. Instance method subtract. Write instance method subtract. As with add, whether you call static methods add or subtract depends on the sign of the operands.

Using BigInt We have implemented multiplication and a simplified version of division:

```
/** Store in q the quotient when this is divided by y
      and return the remainder.
    Precondition (1): 0 < y < BASE.
    Precondition (2): this > 0.
    Precondition (3): q is nonnull and a different
                      folder from this and y. */
public int division(int y, BigInt q)
```

This method calculates the quotient and remainder when **this** integer is divided by y, but only if y is less than BASE. Use this method only when you are using a big enough base. The biggest base you can use is Short.MAX_VALUE. The above is for your information only; you don't need to use it.

(a) Using BigInt. Below is a conventional method for calculating factorial n. Change it so that it returns factorial n as a BigInt, instead of an **int**, and put it in class BigInt. Its name should be *fact*, and variables n and int should be **int**s; only x should be a BigInt.

```
// = factorial n (for 0 <= n)
public static int fact(int n) {
  if (n <= 1)
     {  return n;  }
  int x= 2; int i= 2;
  // invariant: 2 <= i <= n  and  x = factorial i
  while (i != n) {
       i= i+1;
       x= x*i;
  }
  return x;
}
```

After you have tested method fact, use it to find out the first integer n, call it N, for which fact(n) can not be represented as a value of type **long**.This will give you a feel for when type **long** is no longer adequate.

How many digits in a BASE does it takes to represent fact(N)? To find out, run the program to print:

```
fact(N).numberDigits()
```

Run it 3 times, once using BASE 2, once using BASE 10, and once using BASE Short.MAX_VALUE, each time printing the value of the expression show above.

Include the value N and the three lengths in a comment in the specification of method fact, so we can see that you have done this task.

(b) Using BigInt. Below is a method to calculate Fibonnaci(n). Change it so that it returns its result as a BigInt, instead of an **int**, and put it in class BigInt. Its name should be *Fib*, and variables n and k should be **int**s; only b and c should be BigInts.

```
// = Fibonacci number n (for n>= 0).
//   Note:Fib(0) = 0, Fib(1) = 1,
//    Fib(n) = Fib(n-1) + Fib(n-1) for n>=2
public static int Fib(int n) {
  if (n<=1)
     return n;
  int b= 0
  int c= 1;
  int k= 1;
  //inv: 1 <= k <= n and c = Fib(k) and b = Fib(k-1)
  while (k != n) {
        temp= b+c;
        b= c;
        c= temp;
        k= k+1;
     }
  return c;
}
```

After you have tested it. use your program to calculate fib(200).

Include the value fib(200) and the number of digits it takes to represent it in base Short.MAX_VALUE as a comment in the specification of method Fib, so we can see that you have done this task.

What to hand in. Please submit your assignment electronically, Submit only method BigInt; that's all we need. Remember: the comments on methods fact and Fib have to contain some information. (See a description of tasks (a) and (b) on this page).

IMPORTANT NOTE. Recognizing that you may have difficulty with time, only 10 points will be given for the two subtract methods. Therefore, you can leave these undone and still get 90/100. Tasks (a) and (b), presented on this page, do not require subtraction.