Cornell net id _______ Name (last name in ALL CAPS) ____

CS 1110 Prelim 3 *Grades expected to be on the CMS late tonight* 14 April 2009

This 90-minute exam has 6 questions (numbered 0..5) worth a total of 100 points. Spend a few minutes looking at all the questions before beginning. Use the back of the pages if you need more space.

Question 0 (2 points). Fill in your netID and name, legibly, at the top of *each* page. (Hint: do it now.)

The questions in this exam deal with class `BigInt`, which implements integers of any size. Type **`int`** has the range $-2^{31}..2^{31}-1$, and type **`long`** has the range $-2^{63}..2^{63}-1$, but some applications require larger numbers. For example, consider looking for large prime numbers. In 1752, Euler used clever reasoning to show that $2^{31}-1$ was prime. In 1876, Lucas showed that $2^{127}-1$, which has 39 digits, is prime. But since the advent of computers, much larger primes have been found. The longest one so far, 12,978,189 digits long(!), was found in 2008. How does one calculate such things when the basic types **`int`** and **`long`** are so restricted? One way is to write a class each instance of which contains an arbitrarily large integer.

We give below the specification of class `BigInt`, showing many but not all of its methods. For example, we show a method to add two `BigInt`s together but no methods for subtracting, multiplying, etc. You will be asked to write some of these methods. Later in the course, we'll show how this can be made far more efficient by allowing larger numbers in each array element `numb[i]`.

Note that there is a constructor for making a `BigInt` out of an **`int`**, but if you add or multiply many such `BigInt`s together, you may get an integer that is far too big to be in the range of **`int`** or even **`long`**. So, we really need methods for adding, subtracting, and multiplying `BigInt`s together.

```
/** An instance maintains an integer,
    which can be any size. */
public class BigInt {

  /* -1 or 1, depending on whether this
     integer is negative or non-negative. */
  private int sign;

  /* the digits of the absolute value of the integer are

        numb[0..numb.length-1]

     with the least significant digit first.
     E.g. the integer 3124 is stored as {4, 2, 1, 3}.
     numb.length = number of digits in the integer
     with no leading 0's (0 for integer 0). */
  private int[] numb;

  /** Constructor: an instance with value 0. */
  public BigInt()

  /** Constructor: an instance for integer n. */
  public BigInt(int n)

  /** = number of digits for n, with no leading 0's. */
  public static int length(int n)

  /** = this integer, as a string. */
  public String toString()

  /** = "b is a BigInt and contains the same
        integer as this one." */
  public boolean equals(Object b)

  /** = 1, 0, or -1 depending on whether this
      BigInt is greater than, equal to, or less than b. */
  public int compareTo(BigInt b)

  /** = 1, 0, or -1 depending on whether the
        absolute value of this BigInt is
        >, =, or < b. */
  private int compareAbs(BigInt b)

  /** = the sum of x and y.
      Precondition: x and y are non-negative. */
  public static BigInt add(BigInt x, BigInt y)

  /** = – x. */
  public static BigInt negate(BigInt x)

  /** = index of minimum of b[h..k].
        Precondition: h <= k.*/
  public static int min(BigInt[] b, int h, int k)

  /** Sort array b, using selection sort. */
  public static void sort(BigInt[] b)

}
```

Cornell net id _______ Name (last name in ALL CAPS) ____

Question 1 (15 points). Recursion

Function `length` in `BigInt` is supposed to calculate the number of digits needed to store an **`int`** `n` without any leading 0's. For example, the integer 3712 has four (4) digits. Read carefully the specification of the fields in class `BigInt` to understand how many digits are needed to store the integer 0.

Below, write function `length` recursively —do not use a loop. Remember, it should work for negative as well as positive integers `n`.

```
/** = number of digits for n, with no leading 0's */
public static int length(int n)  {




}
```

Question 2 (23 points). Constructors

Below, write the constructor of class `BigInt` that has an **`int`** parameter. It will need a loop; you don't have to write a loop invariant, but doing so may help you get it right. Remember, it should work for negative as well as positive integers n.

(a)

/** Constructor: an instance for integer n. */
public BigInt(**int** n) {

}

(b) In the constructor of part (a), you either put `super();` as the first statement or you didn't. Is it legal to do the opposite of what you chose? If not, explain why not; if yes, explain how or whether doing the opposite would change the result of the call on the constructor.

(c) Below, write the constructor of class `BigInt` that has no parameters. The body must have only one statement: a call on the constructor that has one parameter.

/** Constructor: an instance with value 0 */
public BigInt() {

}

Question 3. (20 points). Selection sort

(a) Write procedure `BigInt.sort`, which is specified below. You must write a selection-sort algorithm. (1) First, write the pre- and post-conditions, as pictures, in mathematical notation, or in English. (2) Second, develop the loop invariant from the pre- and post-conditions. (3) Third, develop the repetend, using the four loopy questions. When writing the repetend, in maintaining the invariant, state what has to be done in one or two statements, written in English. Do not say *how* anything is done; say *what* is to be done. Don't forget part (b), below.

```
/** Sort array b, using selection sort */
public static void sort(BigInt[] b) {









}
```

(b) In part (a), you wrote a loop with a repetend, and the repetend was written mostly in English. Below, show how to implement those English statements in Java. You can use other methods in class `BigInt`, so have a look around at that class on page 1 of this exam.

Cornell net id ______ **Name (last name in ALL CAPS)** ____

Question 4 (20 points). Using loop invariants

Function `add` has to add two BigInts (represented in arrays) together, using the standard addition algorithm. To remind you how this works, we give an example in the box on the right. For example, 6+4 = 10, so one writes the 0 below the line and writes the "carry", *1*, at the column to the left of the 6+4 column. Thus, the second-column addition is 1+3+8 = 12, which also produces a carry.

$$\boxed{\begin{array}{ccc} \mathit{1} & \mathit{1} & \\ 1 & 3 & 6 \\ 2 & 8 & 4 \\ \hline 4 & 2 & 0 \end{array}}$$

You have to complete the code below. It assumes that `x`, `y`, and `z` are `BigInt` variables and that **`int`** arrays `x.numb` and `y.numb` contain integers. The code adds `x.numb` and `y.numb` together and stores the answer (except for the final carry) in `z.numb`. Do this in two steps, (a) and (b).

(a) The invariant is given in English. Write the first part of the loop invariant using our pictorial representation of arrays.

(b) Complete the assignment to variable `carry`, and, below the invariant, write a *single* loop (**for** or **while**, whichever you prefer) that performs the addition. Be careful. Arrays `x.numb` and `y.numb` may have different lengths, so don't try, for example, to reference `x.numb[k]` if it doesn't exist.

```
z.numb= new int[Math.max(x.numb.length, y.numb.length);
carry=                    ;
// invariant: z.numb[0..k-1] is the sum of x.numb[0..k-1] and y.numb[0..k-1],
//            except that the carry into position k is in variable carry (carry is 0 if k is 0).
```

```
// postcondition: z.numb contains the sum of x.numb and y.numb,
//                except that the carry for the final position is in variable carry.
```

Cornell net id _______ **Name (last name in ALL CAPS)** ____

0	___________	out of 02
1	___________	out of 15
2	___________	out of 23
3	___________	out of 20
4	___________	out of 20
5	___________	out of 20
Total	________	out of 100

Question 5 (20 points). Function equals

(a) Write function `equals`. Don't write a loop; instead, look for another function in `BigInt` that can do much (but not all) of what is required.

```
/** = "b is a BigInt and contains the same integer as this one". */
   public boolean equals(Object b) {









}
```

(b) Write function `equals` again, this time using no calls on other methods in class `BigInt`.

```
/** = "b is a BigInt and contains the same integer as this one". */
   public boolean equals(Object b) {









}
```