CSCI 145 Week 8
Assignment 2

Assignment Description

For this assignment you will create a class `LDecimal`. The `LDecimal` class will be a linked list class that provides arbitrary precision decimal arithmetic.. You must submit your program by 11:59 pm on Wednesday, March 4, 2015. This assignment will be worth 15% of your grade for the course.

Arbitrary precision arithmetic

In Java the largest `int` (found at `Integer.MAX_VALUE`) is 2,147,483,647 (= 2^{31}-1), and the largest `long` (found at `Long.MAX_VALUE`) is 9,223,372,036,854,775,807 (= 2^{63}-1). There are computations where these maximums are too small. For example, your Internet browser must do computations with large prime numbers (300 or more digits) when using HTTPS to protect your communications with banks and stores.

In order to do these computations you need a program that can do arbitrary precision arithmetic. (Arbitrary precision means that the maximum size of numbers is limited by the time and memory available for the computation and not the size of a machine word in your computer.) Ordinarily, these packages handle numbers as arrays of `int`s or `long`s. However, for this assignment, you are going to build a simple arbitrary precision arithmetic class that does computations using a linked list of decimal digits, much like you learned to do arithmetic in elementary school.

Assignment Details

Outline

- Requirements for the `LDecimal` class
- Suggested steps for building the `LDecimal` class
- Testing your `LDecimal` class
- Additional notes
- Description of `LDecimal` methods
- Algorithms for arbitrary precision decimal arithmetic

Requirements for the LDecimal class

1. Your class must be named `LDecimal`.
2. Your class must be public.
3. Your class must provide the methods listed under *Description of `LDecimal` methods* for construction, accessing, and manipulating `LDecimal` objects.

4. `LDecimal` objects will be immutable. There should be no public methods that can change the value of an `LDecimal` once it has been created.
5. Other than for your own internal testing purposes, your `LDecimal` class should do no input or output. Your submitted version of the `LDecimal` class cannot do any input or output.
6. Your `LDecimal` class must allow the provided test program, `LDecimalTest.java`, to compile and execute. Use of this program is described in more detail under *Testing your `LDecimal` class*.
7. **Important!** The purpose of this assignment is for you to gain experience with implementing and manipulating linked lists. For this reason, you are not allowed to use the Java `BigInteger` or `BigDecimal` classes or any of the Java Collections classes, such as `LinkedList` or `ArrayList`. You are free to look at the documentation and implementation of those classes to gain insight into the problem.

   If you are unsure about the classes you can use, please consult with your TA or me. However, as a point of reference, the only class from the Java library that I used to do the solution was the `StringBuilder` class. My use of this class is described under *Additional Notes*.

Suggested steps for building the LDecimal class

When doing this assignment, you should approach things in the following order:

1. Create an `LDecimal` class that will allow `LDecimalTest` to compile. To do that you will need to create stubs for all the public `LDecimal` methods.
2. Now work through the methods in the following order:
   a) The `LDecimal` constructors, `digits`, `signum`, and `toString`.
   b) `compareTo` and `equals`
   c) `add`
   d) `subtract`
   e) `multiply`
   f) `divide`

   The tests in `LDecimalTest` are organized in five phases. The first four phases correspond (a), (b), (c) and (d), and (e) and (f) in that order. The fifth ("torture") is a set of special tests that use the already built methods and do arithmetic with very large numbers.

Testing your LDecimal class

I have provided a program, `LDecimalTest.java`, that tests your implementation of `LDecimal`. The `LDecimalTest` class uses some additional packages that are not part of the Java standard library. The file `junit.jar` contains the additional classes needed by `LdecimalTest`.

To compile and run your program in JGrasp you need to identify the location of `junit.jar` to jGrasp. You can do this by using the Settings/"PATH / CLASSPATH" menu item. Click on the CLASSPATHS tab. In that tab, click the New button. Now Browse to the `junit.jar` file. The file path and name should appear in the top of the two text boxes. Click OK (twice). You can test that this works by compiling `LDecimalTest.java`.

If you would like to compile using the command line, here is how you do it:

```
$ javac -cp junit.jar LDecimalTest.java
```

Here is how to run the program from the command line:

```
(Mac OS X and Linux)$ java -cp .:junit.jar LDecimalTest
(Windows)> java -cp .;junit.jar LDecimalTest
```

The difference between the two commands is the character separating the `.` and `junit.jar` in the the classpath (`-cp`). (On Mac OS X and Linux it's ':'. On Windows it's ';'.)

Here's a sample successful run of `LDecimalTest`.

```
$ java -cp .:junit.jar LDecimalTest
Running phase basic: constructor, digits, signum, toString (39
tests)
Starting tests: .......................................
Time: 0.016
OK! (39 tests passed.)

Running phase compare: compareTo and equals (18 tests)
Starting tests: ..................
Time: 0.016
OK! (18 tests passed.)

Running phase add: add and subtract (76 tests)
Starting tests: .....................................................
.......................
Time: 0.031
OK! (76 tests passed.)

Running phase multiply: multiply and divide (43 tests)
Starting tests: ...........................................
Time: 0.016
OK! (43 tests passed.)

Running phase torture: torture tests (2 tests)
Starting tests: ..
Time: 1.281
OK! (2 tests passed.)

Congratulations! All tests passed.
```

Each test is testing a single operation, for example adding 9 + 1 and checking that the answer is 10. Each dot indicates that one test passed.

The `LDecimalTest` class supports a number of options:

```
$ java -cp .:junit.jar LdecimalTest -h
usage: java LDecimalTest [options] [phases]
 -c,--continue   Continue testing after failures
 -d,--debug      Debugging mode (no timeouts)
 -h,--help       Print this help message
```

Some explanation:

- Ordinarily, if there are failures in one phase, the following phases will not be run. The continue option (-c or --continue) will cause later phases to execute anyway.
- There are timeouts specified for all tests: 10 milliseconds (0.010 seconds) for all tests except torture tests and 10 seconds for torture tests. These timeouts will interfere using the debugger with the program. The debug option (-d or --debug) will suppress the timeouts allowing use of the program with a debugger. If you use this option and your program has an infinite loop, you will have to manually stop the program.
- If you supply a list of phase names after the command, those phases will be run in that order. If no phases are specified, the five phases will be run in the standard order.

Here's one more example, a run with errors:

```
$ java -cp .:junit.jar LDecimalTest add multiply
Running phase add: add and subtract (76 tests)
Starting tests: ......................................................
......................
Time: 0.047
OK! (76 tests passed.)

Running phase multiply: multiply and divide (43 tests)
Starting tests: .......................EEE.EEEEEEEEEEEEEEE.
Time: 0.016

There were 18 failures:
1) t1TestDivide[1](LDecimalTest$LDecimalDivide)
java.lang.NullPointerException
        at LDecimal.doDivide(LDecimal.java:446)
        at LDecimal.divide(LDecimal.java:415)
        at
LDecimalTest$LDecimalDivide.t1TestDivide(LDecimalTest.java:622)
2) t1TestDivide[2](LDecimalTest$LDecimalDivide)
java.lang.NullPointerException
        at LDecimal.doDivide(LDecimal.java:446)
        at LDecimal.divide(LDecimal.java:415)
        at
LDecimalTest$LDecimalDivide.t1TestDivide(LDecimalTest.java:622)
```

Sixteen additional test failure reports have been deleted from this listing. They are all NullPointerExceptions at Ldecimal.java:446.

Note that this run only ran the two phases specified on the command line: add and multiply. Also, note the 'E' in the line of dots in the multiply phase. That shows which tests failed.

I will be demonstrating this program in class.

Additional notes

1. Watch out for arithmetic with `int`s. 9 * 1,000,000,000 will overflow the capacity of an `int`. Also, for `int`s, -(-2,147,483,648) == -2,147,483,648! Most of the intermediate arithmetic,

especially when converting an `int` to an `LDecimal` and multiplying and dividing should be done using `long`s.

2. Look for opportunities to (1) use one operation to perform another, or (2) write utility operations (make sure these are `private`) that support multiple operations. For example, (1) the `equals` method can use the `compareTo` method, (2) both `add` and `subtract` need to perform both addition and subtraction, depending on whether numbers have the same or different signs.
3. The `toString` method can perform poorly with very large numbers (ones with hundreds of digits or more). In order to deal with this do <u>not</u> do:

```
public String toString() {
    String result = "";
    for ( digit in digits of this number ) {
        result = result + digit;
    }
    return result;
}
```

   Instead do this:

```
public String toString() {
    StringBuilder result = new StringBuilder();
    for ( digit in digits of this number ) {
        result.append( digit );
    }
    return result.toString();
}
```

4. Implementation of the `equals` method is tricky. The parameter is of type `Object` which allows comparison with anything. Here's how you should write the `equals` method:

```
@Override
public boolean equals(Object other) {
   if (other == null || !(other instanceof LDecimal))
      return false;
   else {
      LDecimal otherLDecimal = (LDecimal)other;
      // Your logic to compare this and otherLDecimal
      // goes here.
      // Return true it they're equal, false otherwise
   }
}
```

   Some notes on equals:

   - The `@Override` tells the compiler that this method overrides the standard `equals` method.
   - The `if` condition tests to ensure that `other` is not `null` and that it's an `LString`. If either of these conditions fails, the method returns `false`, indicating inequality.

- The expression `(LDecimal)other`, converts the reference to an `Object` to a reference to an `LDecimal`. This will work since we've already determined that it's actually an `LDecimal`. (If, for some reason, it was not an `LDecimal`, an exception would be raised here.)

5. Make sure to check the *Algorithms for arbitrary precision decimal arithmetic* for ideas on how to do the arithmetic.

Description of LDecimal methods

The public constructors and methods required for the `LDecimal` class are listed here.

public LDecimal()	Construct an `LDecimal` object. The object will have value zero (0).
public LDecimal(int n)	Construct an `LDecimal` object whose value is the integer `n`. (See *Algorithms, Convert an integer to an LDecimal.*)
public int digits()	Return the number of digits in this `LDecimal`. The number of digits ignores the sign, so both 10 and -10 have two digits. The `LDecimal` zero has zero digits.
public int signum()	Return the signum of this `LDecimal`. The signum is 1 if the number is positive, -1 if the number is negative, and 0 if the number is zero.
public String toString()	Return a printable representation of this `LDecimal`. The representation of zero is “0”. The representation of a positive number is the digits of the number, for example “12”. The representation of a negative number is digits with a with a minus sign at the front, for example, “-12”. In all cases, there should be no space and no punctuation other than the minus sign at the beginning of a negative number. (See note 3, above.)
public int compareTo(LDecimal other)	Return the value `0` (zero) if this `LDecimal` is equal to (has the same digits and sign as) the parameter `other`; a value less than `0` if this `LDecimal` is less than `other`; and a value greater than `0` if this `LDecimal` is greater than `other`. Note: The returned values do not have to be -1, 0, and 1. Any integer with the correct sign will work.
public boolean equals(Object other)	Return `true` if this `LDecimal` is equal to `other`. Note that the parameter is of type Object and not the type `LDecimal`. See the relevant note for how to handle this. (See note 4, above.)
public LDecimal add(LDecimal val)	Return the sum of this `LDecimal` and the `LDecimal` other. If either this number or val is zero, same `LDecimal` object as the non-zero number must be returned. A new `LDecimal` is not to be constructed. If both are zero, either one can be returned. (See *Algorithms, Add two LDecimals with the same sign* and *Algorithms, Subtract two LDecimals with the same sign*.)

public LDecimal subtract(LDecimal val)	Return the difference of this `LDecimal` and the `LDecimal` other. If val is zero, this same `LDecimal` object must be returned. (See *Algorithms, Add two LDecimals with the same sign* and *Algorithms, Subtract two LDecimals with the same sign*.)
public LDecimal multiply(int val)	Return the product of this `LDecimal` and the `int` val. (See *Algorithms, Multiply an LDecimal by an integer*.)
public LDecimal divide(int val)	Return the result of dividing this `LDecimal` by the `int` val. This method must throw an ArithmeticException if val == 0. (See *Algorithms, Divide an LDecimal by an integer*.)

Algorithms for arbitrary precision decimal arithmetic

Here are algorithms for all the arithmetic needed for this assignment. For reference, the term *least significant digit* refers to the right-most digit of the number (the one that changes fastest when counting) and the term *most significant digit* refers to the left-most digit of the number.

These descriptions assume that the digits of an `LDecimal` are placed in a doubly linked list (allowing the list to be traversed in both directions) with the front of the list being the most significant digit and the tail of the list being the least significant digit.

You may have to modify these algorithms to work with your data structures. In particular, the `digits` method invites maintaining the number of digits as an attribute of an `LDecimal` object. In that case, these algorithms must be modified to compute that along with the sign and the digits of the new number.

You don't have to use these algorithms. They are provided here to show one approach to solving the problem.

The algorithms are given in Python-ish pseudo code.

Convert an integer to an LDecimal

```
Ldecimal(n):
    t = n
    sign = 1
    if t < 0:
        t = -t
        sign = -1

    while t != 0:
        d = t % 10
        t = t / 10
        // d is the new most significant digit
        insert d at front of the digits
```

Add two LDecimals with the same sign

Note: To add two `LDecimals` with different signs you must adjust the signs appropriately and subtract.

```
add(other):
    // Handle special cases
    if other == 0:
        return this
    if this == 0:
        return other

    result = a new LDecimal object
    sign = this.sign
    currentThis = least significant digit of this
    currentOther = least significant digit of other
    carry = 0
    while there are digits left:
        // If the digits of either this or other run out before the other one does
        // act as if sufficient remaining zero digits are present
        d = currentThis + currentOther + carry
        carry = 0
        if d > 10:
            d = d – 10
            carry = 1
        // d is the new most significant digit
        insert d at front of the digits of result
        currentThis = currentThis.previous // move to the next most significant digit
        currentOther = currentOther.previous
    if carry > 0:
        insert 1 at the front of the digits of result
    return result
```

Subtract two LDecimals with the same sign

Note: To subtract two `LDecimals` with different signs you must adjust the signs appropriately and add.

```
subtract(other):
    // Handle special cases
    if other == 0:
        return this
    if this == 0:
        return -other

    result = a new LDecimal object
    sign = this.sign
    // Check to see which has the larger absolute value. (Absolute value ignores
    // the signs of the numbers)
```

```
        if |other| > |this|:
            swap this and other
            sign = -sign
        currentThis = least significant digit of this
        currentOther = least significant digit of other
        borrow = 0
        while there are digits left:
            // this will always have at least as many digits as other. That's a result of
            // making sure the absolute value of this is greater than the absolute value of
            // other. If the digits of other run out before the digits of this act as if
            // sufficient remaining zero digits are present in other
            d = currentThis - currentOther - borrow
            borrow = 0
            if d < 0:
                d = d + 10
                borrow = 1
            // d is the new most significant digit
            insert d at front of the digits of result
            currentThis = currentThis.previous // move to the next most significant digit
            currentOther = currentOther.previous

        // At this point borrow must be zero since |this| >= |other|
        // Take any zeros off the front of result
        current = most significant digit of result
        while current exists (is not null) and current == 0:
            remove most significant digit of result
            current = current.next
        if current does not exist:
            return LDecimal(0)
        return result
```

Multiply an LDecimal by an integer

```
    multiply(n):
        if this == 0 or n == 0:
            return LDecimal(0)
        result = a new LDecimal object

        if this.sign == sign of n:
            sign = 1
        else:
            sign = -1

        if n < 0:
            n = -n
        t = 0
        currentThis = least significant digit of this
        while there are still digits of this:
            t = t + currentThis * n
            insert t % 10 at front of the digits of result
            t = t / 10
            currentThis = currentThis.previous // move to the next most significant digit

        // Once the digits of this have exhausted we still have to handle any remaining
        // digits contained in t
        while t != 0:
            insert t % 10 at front of the digits of result
            t = t / 10
        return result
```

Divide an LDecimal by an integer

```
    divide(n):
        if other == 0:
            throw new ArithmeticException("Divide by zero")
        if this == 0:
            return LDecimal(0)

        result = a new LDecimal object
        if this.sign == sign of n:
            sign = 1
        else:
            sign = -1
        if n < 0:
            n = -n
        t = 0
        currentThis = most significant digit of this
        while there are still digits of this:
            t = t * 10  + currentThis
            insert t / n at front of the digits of result
            t = t % n
            currentThis = currentThis.next // move to the next least significant digit

        // It is possible that there are zero digits at the front of result. We want to eliminate
        // these.
        current = most significant digit of result
        while current exists and current == 0:
            remove most significant digit of result
            current = current.next
        if current does not exist:
            return LDecimal(0)
        return result
```

Coding Standards

Your program should follow the standards described as part of Lab 1.

Your program will be graded on conformance to the coding standards as well as correct functionality.

Turn in your program on Canvas

You should turn in your `LDecimal.java` class on Canvas. You should not turn in `LDecimalTest.java` or `junit.jar`. Your program will be tested with the standard version of `LDecimalTest`. You should remember not to make any modifications to this class.

Grading

Your program is due by 11:59pm, Wednesday, March 4. The grading will be:

- 10% – basic tests (Testing phase 1)
- 20% – compare tests (Testing phase 2)
- 20% – add and subtract tests (Testing phase 3)
- 20% – multiply and divide tests (Testing phase 4)
- 10% – torture tests collections (Testing phase 5)
- 20% – overall program organization and presentation
- There will be a 10% deduction for <u>not</u> turning your program in correctly.