UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2013

Test #1 Solutions

1. [4 points] The class `DecimalNumeral` represents arbitrarily large non-negative integers, written in decimal. It differs from the usual sort of large-integer type in that it makes its digits available by means of an iterator. Fill in the `toString` method (part a) on this page, and the supporting classes (parts b and c) on subsequent pages.

```
/** Represents a non-negative decimal numeral. MODIFY ONLY toString IN THIS CLASS. */
public abstract class DecimalNumeral {
    /** The decimal digits of this numeral, presented right to left
     *  (units digit first). */
    abstract public Iterator<Integer> digits();

    @Override
    public String toString() { // see part (a)
    }

    /** Return the decimal numeral whose digits are DIGS (an array of
     *  Integer).  DIGS must have at least one element and each item
     *  must be in the range 0-9. */
    public static DecimalNumeral num(Integer... digs) {
        IntList L; L = null;
        for (int d : digs) {
            L = new IntList(d, L);
        }
        return new Literal(L);
    }

    /** The numeral that denotes the sum of this numeral and Y. */
    public DecimalNumeral add(DecimalNumeral y) {
        return new Sum(this, y);
    }
}
```

a. Fill in the definition of the `toString` method in `DecimalNumeral`. You may *not* make any assumptions about how the numerals you print were created.

```
@Override
public String toString() { // Solution
    ArrayList<Integer> digs = new ArrayList<>();
    for (Iterator<Integer> it = digits(); it.hasNext(); ) {
        digs.add(0, it.next());  // Add to the front
    }
    String result;
    result = "";
    for (Integer d : digs) {
        result += d.toString();
    }
    return result;
}
```

Continuation of problem #1

b. Complete the definition of `Literal` class. Its constructor creates a `Literal` from an `IntList` containing its digits from right to left. You may only add private methods, instance variables, and classes if needed.

```
class Literal extends DecimalNumeral {
    /** My digits, units digit first. */
    private IntList _digits;

    Literal(IntList L) {
        _digits = L;
    }

    @Override
    public Iterator<Integer> digits() { // Solution
        return new DigitIterator();
    }

    // Solution
    private class DigitIterator implements Iterator<Integer> {
        private IntList _next;
        DigitIterator() {
            _next = _digits;  // or Literal.this._digits;
        }
        @Override
        public boolean hasNext() {
            return _next != null;
        }
        @Override
        public Integer next() {
            if (_next == null) {
                throw new NoSuchElementException();
            }
            Integer result = _next.head;
            _next = _next.tail;
            return result;
        }
        @Override
        public void remove() { // Did not need this
            throw new UnsupportedOperationException();
        }
    }
}
```

Continuation of problem #1

c. Fill in the definition of the `Sum` class.

```
class Sum extends DecimalNumeral {
    private DecimalNumeral _x, _y;

    Sum(DecimalNumeral x, DecimalNumeral y) {
        _x = x; _y = y;
    }

    public Iterator<Integer> digits() { // Solution
        return new SumIterator();
    }

    // Solution:
    private class SumIterator implements Iterator<Integer> {
        private Iterator<Integer> _xi, _yi;
        private int _carry;
        SumIterator() {
            _xi = _x.digits();  // or xi = Sum.this._x;
            _yi = _y.digits();
            _carry = 0;
        }
        @Override
        public boolean hasNext() {
            return _carry != 0 || _xi.hasNext() || _yi.hasNext();
        }
        @Override
        public Integer next() {
            int result;
            result = _carry;
            if (_xi.hasNext()) {
                result += _xi.next();
            }
            if (_yi.hasNext()) {
                result += _yi.next();
            }
            _carry = result / 10;
            return result % 10;
        }
        @Override
        public void remove() { // Did not need this
            throw new UnsupportedOperationException();
        }
    }
}
```

2. [3 points] Fill in the following method to fulfill its comment. For example, (using Java's binary literals),

```
    select(0b111000, 0b110101, 0b001010) == 0b001101.
```

The answer consists of a single return statement (no loops).

```
    /** Returns the value whose kth bit (for all 0<=k<32) is the kth bit
     *  of LEFT if bit k of WHICH is 0, and the kth bit of RIGHT
     *  otherwise.  */
    static int select(int which, int left, int right) { // Solution
        return (left & ~which) | (right & which);
    }
```

3. [2 points] The definition of `Rational` from Lab #3 contained this method:

```
    /** Returns my representation as a String.  Returns a String of the form
     *  N/D or -N/D, where N/D is a fraction in lowest terms, leaving off /D
     *  when D is 1. */
    public String toString() {
        if (_den == 1) {
            return String.format("%d", _num);
        } else {
            return String.format("%d/%d", _num, _den);
        }
    }
```

The `toString` method of implementations of `List<Foo>` in the Java library produces `"[`V_1`, `V_2`, ..., `V_n`]"` where the V_i come from calling `.toString()` on the elements of the list. Write a Java Pattern that matches the output of `.toString` applied to a `List<Rational>`. Feel free to define String variables that you then concatenate into the final pattern, if you find this helps.

```
// Solution.  To make things easier to read, we use the trick of
// composing our pattern strings out of smaller ones, using
// .replace to make the resulting (sub)patterns clearer (String.format
// would also work).

/** Integer numeral */
String Int = "(0|[1-9][0-9]*)";
    // Plain "[0-9]+" was also OK, although it allows octal numerals.
/** Rational numeral. */
String Rat = ("-?<I>(/<I>)?").replace("<I>", Int);

Pattern rationalsList =
    Pattern.compile(("\<R>(,␣<R>)*)?\\").replace("<R>", Rat));
```

4. [1 point] You are standing on land at night (the sun is well below the horizon) and can see the Pacific Ocean. There are no nearby sources of light and the moon is new. However, the landscape around you is illuminated brightly enough to read by. Where are you?

> *The intended solution was "the Moon," since under these circumstances, earthlight would be quite bright. Some people suggested some arctic region under the aurora. We gave half credit if the aurora was specifically mentioned—only half because they are usually pretty dim (the exception being during major solar storms). Some mentioned bioluminescent phenomona, but these would be local light sources.*

5. [4 points] For each of the following programs, indicate which of the following choices best describes its worst-case running time, as measured by the number of calls it makes to `f` (some constant-time function).

I. $\Theta(\lg N)$

II. $\Theta(N)$

III. $\Theta(N\sqrt{N})$

IV. $\Theta(N^2)$

V. $\Theta(2^N)$

a. For this function, pretend that integers have an unlimited range.

```
void doStuff(int N) {                   // Time:  I.
     for (int i = 2; i < N; i = 2*i)
         f(i);
}
```

b. Here, assume that if `L` is an `IntList`, then `nth(L, k)` fetches its k$^{\text{th}}$ item, or 0 if there is none. You don't have to implement it, but must make a reasonable hypothesis about how it is implemented (and thus how fast it is). Take N to be the length of `L`.

Note: Actually, this question has a problem. Assuming that we are counting only calls to 'f', the answer is II. If one does count the cost of 'nth', then the answer becomes IV.

```
void doMoreStuff(IntList L) {           // Time: IV  (or II)
    int i, j;
    i = 0;
    j = nth(L, i);
    while (j != 0) {
        f(j);
        i += 1;
        j = nth(L, i);
    }
```

c. Take N to be `A.length`.

```
void addThings(int A[]) {              // Time: II.
    int S = (int) Math.sqrt(A.length);
    for (int i = 0; i < A.length; i += S) {
        for (int j = 0; j < S; j += 1) {
            f(i+j);
        }
    }
}
```

More parts on the next page.

Continuation of problem 5.

d. Take $N = U - L$.

```
void checkThings(int A[], int L, int U) {   // Time: V
    if (L > U)
        return;
    if (f(A[U]))
        checkThings(A, L, U-1);
    if (f(A[L]))
        checkThings(A, L+1, N);
}
```

e. Take $N = U - L$.

```
void checkThings(int A[], int L, int U) {   // Time:  II
    if (L > U)
        return;
   if (f(A[U]))
        checkThings(A, L, U-1);
    else if (f(A[L]))
        checkThings(A, L+1, N);
}
```

6. [4 points] Given two `IntLists` that are in non-decreasing order, the following function is supposed to non-destructively return the result of removing all numbers that appear on the second list from the first. For example, if `L1` contains `[1, 7, 13, 15, 20, 25]` and `L2` contains `[6, 9, 13, 14, 17, 20, 21]`, then the result of `removeAll(L1,L2)` should be `[1, 7, 15, 25]`; and the lists referenced by `L1` and `L2` should not change. The method does not meet this specification. Show how to fix the problems. Make as little change as possible.

```
public static IntList removeAll(IntList fromList, IntList remove) {
    IntList theList, result, last;

    theList = fromList;
    result = last = null;     // ERROR: initialization was missing
    while (theList != null && remove != null)  {
                  // ERROR: didn't test for empty remove list.
        if (remove.head < theList.head) {  // ERROR: was ">"
            remove = remove.tail;
        } else {
            if (theList.head < remove.head) { // ERROR: was ">"
                if (result == null) { // Was misplaced
                    result = last = new IntList(theList.head, null);
                        // ERROR: was destructive.  Need new items.
                } else {
                    last = last.tail = new IntList(theList.head, null);
                    // ERROR: was destructive.  Need new items.
                }
            }
            theList = theList.tail;  // ERROR: List did not advance.
        }
    }

    if (result == null) {
        result = theList;
    } else {
        last.tail = theList;   // ERROR: Must finish list.
    }

    return result;
}
```