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

CS61B **P. N. Hilfinger**
Fall 2007

Supplementary Quiz Solutions

1. [10 points] One step in the quicksort algorithm is that of *partitioning* a list around some *pivot* element. Using the following class definition:

```
class IntList {
    public final int head;  // May not be assigned to after construction
    public IntList tail;
    public IntList (int head, IntList tail) {
        this.head = head; this.tail = tail;
    }
    /** The result of appending L2 to the end of L1.  L1 is modified;
     *  creates no new IntLists. */
    public static IntList dappend (IntList L1, IntList L2) {
        if (L1 == null) return L2;
        IntList result = L1;
        while (L1.tail != null) L1 = L1.tail;
        L1.tail = L2;
        return result;
    }
}
```

fill in the following two versions of partitioning:

a. [15 minutes]

```
/** A list consisting of all the values in L re-ordered such that:
 *     a. All elements that are less than PIVOT come first (in any order);
 *     b. All items equal to PIVOT (there may be 0 or more) come next; and
 *     c. All items greater than PIVOT come last (in any order).
 *   The operation is destructive: the original IntList items in list L
 *   may be modified, and no new IntList items may be created.
 */
IntList partition (IntList L, int pivot) {
    IntList left, mid, right;
    left = mid = right = null;
    while (L != null) {
        IntList next = L.tail;
        int h = L.head;
        if (h < pivot) {
            L.tail = left; left = L;
        } else if (h > pivot) {
            L.tail = right; right = L;
        } else {
            L.tail = mid; mid = L;
        }
        L = next;
    }
    return IntList.dappend (left, IntList.dappend (mid, right));
}
```

b. [15 minutes]

```
/** A list consisting of all the values in L re-ordered such that:
 *     a. All elements that are less than PIVOT come first (in any order);
 *     b. All items equal to PIVOT (there may be 0 or more) come next; and
 *     c. All items greater than PIVOT come last (in any order).
 *   The operation is nondestructive: the original IntList items in list L
 *   are unchanged.
 */
IntList partition (IntList L, int pivot) {
    // We could use essentially the same structure as the first
    // solution, but here's a different approach.
    return select (L, pivot, -1,
                   select (L, pivot, 0,
                           select (L, pivot, 1, null)));
}

/** The items, x,  of L such that x-pivot has the same sign as
 *  S (S in {-1,0,1}), followed by the list TAIL. */
IntList select (IntList L, int pivot, int s, IntList tail) {
    if (L == null)
       return tail;
    int h = L.head;
    if (s == ((h < pivot) ? -1 : (h == pivot) ? 0 : 1))
        return new IntList (h, select (L.tail, pivot, s, tail));
    else
        return select (L.tail, pivot, s, tail);
}
```

2. Who invented the word "gruntled"? **Ans:** *P. G. Wodehouse*

3. [10 points, 15 minutes] Consider the following definitions:

```
/** Represents a function from values of type T1 to values of type
 *  T0. */
interface UnaryFunction<T0,T1> {
   T0 eval (T1 x);
}

/** Represents a binary function that combines a value of type T0 with
 *  one of type T1 to produce a value of type T0. */
interface BinaryFunction<T0,T1> {
    T0 eval (T0 x, T1 y);
}

/** The list [F.eval(L0), F.eval(L1), ...], where L0, L1, ... are the
 *  values in L (non-destructive). */
static <T0,T1> List<T0> map (UnaryFunction<T0,T1> f, List<T1> L) {
   ArrayList<T0> V = new ArrayList<T0> ();
   for (T1 x : L)
      V.add (f.eval (x));
   return V;
}

/** The value INIT @ L0 @ L1 @ ..., where x @ y here means
 *  F.eval(x,y), and L0, L1, ... are the values in L. */
static <T0,T1> T0 reduce (BinaryFunction<T0,T1> f, T0 init, List<T1> L) {
   for (T1 x : L)
       init = f.eval (init, x);
   return init;
}
```

(Reminder: The `<T0,T1>` in front of each static function definition is just there to declare the type parameters.)

Problem continues on the next page.

Using the definitions on the preceding page, and any additional class declarations you need, but *no loops or new recursive functions,* implement the following (our solution has 11 lines besides those shown here):

```
/** The length of the longest string in L. */
static int longestLength (List<String> L) {
    return reduce (new max (), 0, map (new len (), L));
}

class max implements BinaryFunction<Integer, Integer> {
    public Integer eval (Integer x, Integer y) {
        return Math.max (x, y);
    }
}

class len implements UnaryFunction<Integer,String> {
    public Integer eval (String x) {
        return x.length ();
    }
}
```