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

CS61B **P. N. Hilfinger**
Fall 2017

Test #1 Solution

Reference Material.

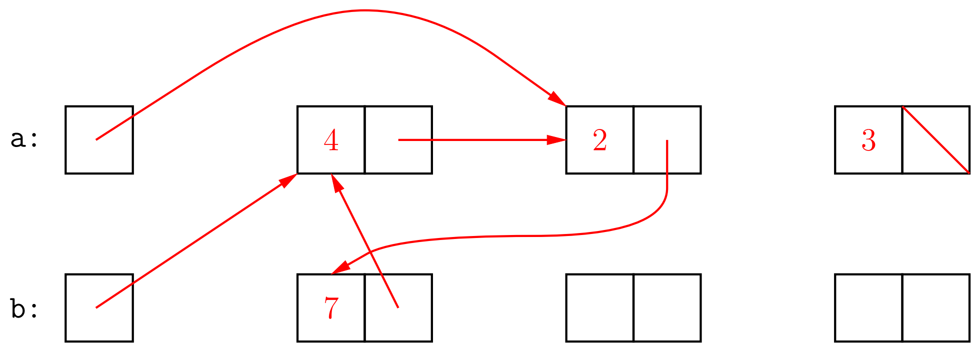
```
/* arraycopy(FROM_ARR, FROM_INDEX, TO_ARR, TO_INDEX, LENGTH) */
import static java.lang.System.arraycopy;

public class IntList {
    /** First element of list. */
    public int head;
    /** Remaining elements of list. */
    public IntList tail;

    /** A List with head HEAD0 and tail TAIL0. */
    public IntList(int head0, IntList tail0)
    { head = head0; tail = tail0; }

    /** A List with null tail, and head = 0. */
    public IntList() { this(0, null); }

    /** Returns a new IntList containing the ints in ARGS. */
    public static IntList list(Integer ... args) {
           // Implementation not shown
    }

    /** Returns true iff L is an IntList with the same items as this
     *  list in the same order. */
    @Override
    public boolean equals(Object L) {
           // Implementation not shown
    }

    /** Return the length of the non-circular list headed by L. */
    @Override
    public int size() {
           // Implementation not shown
    }

}
```

1. [3 points] For the following code snippets, fill in the box and pointer diagrams to show the variables and objects created and their contents after executing the snippets, using the empty boxes provided. After the code executes, some objects may be unreachable from any named pointer variable (may be "garbage" to use the technical term); show them anyway. The double boxes represent `IntList` objects (see definition on page 2), with the left box containing the `head` field. Show also any output produced. You may not need all the boxes or output lines provided.

(a) [1 point]

Write Output Here:

```
IntList a = IntList.list(1, 2, 3);
IntList b = a;
a.head = 4;

System.out.println(b.head);                        4
a = a.tail;
a.tail = new IntList(7, b);

System.out.println(b.tail.tail.tail.tail.head);    2
```



(b) [1 point] Here, the clusters of boxes in this problem are array objects. The array indices (0 and 1 or 0, 1, and 2) are not shown. Boolean values print as “true” or “false”.

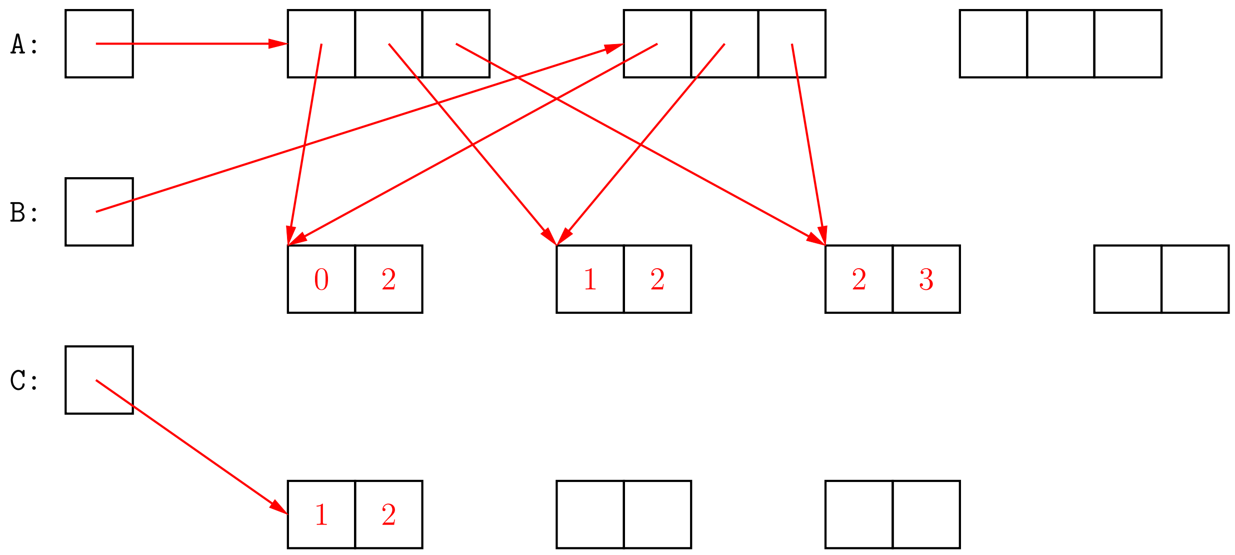
```
int[][] A = new int[3][2];
int[][] B = new int[3][];
int[] C = new int[] { 1, 2 };
for (int i = 0; i < 3; i += 1) {
    for (int j = 0; j < 2; j += 1) {
        A[i][j] = i + j;
    }
}
for (int i = 0; i < 3; i += 1) {
    B[i] = A[i];
}
A[0][1] += 1;
```

Write Output Here:

```
System.out.println(B[0][1]);
```
2

```
System.out.println(A[1] == B[1]);
```
true

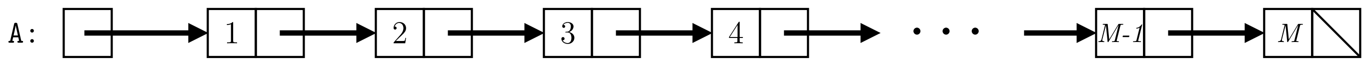
```
System.out.println(A[1] == C);
```
false



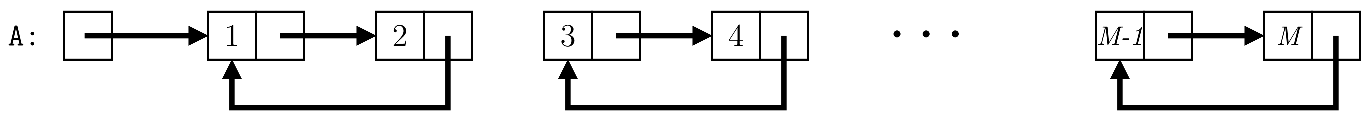
(c) [1 points] For this one, fill in the program so that it converts the *Before* diagram to the *After* diagram. All objects depicted are `IntList`s. **Do not** create any additional `IntList` objects, and **do not** change the `head` of any `IntList` object. The value of M is even, but should not be used in the program. Not all blank lines need to be used.

```
IntList A = IntList.list(1, 2, 3, 4, ..., M-1, M);


IntList p, n;

p = A;


while (p != null) {


    n = p.tail.tail;

    p.tail.tail = p

    p = n

}
```

Before:



After:



2. [3 points] Fill in the following non-destructive method to meet its specification. Feel free to use the `System.arraycopy` method from page 2.

```
/** Return an array containing subarrays of A concatenated
 *  together.  The kth such subarray starts at index STARTS[k]
 *  in A and has length LENGS[k].  Assume that STARTS and LENGS
 *  have the same length and that the indicated ranges are all
 *  within the bounds of A.
 *  For example, if A is
 *      { "A", "B", "C", "D", "E", "F", "G", "H" }
 *  STARTS is
 *      { 1,  5,  2 }
 *  and LENGS is
 *      { 2,  3,  4 }
 *  then the result will be
 *      { "B", "C", "F", "G", "H", "C", "D", "E", "F" }
 *        |<---->|  |<--------->|  |<-------------->|
 */
static String[] gather(String[] A, int[] starts, int[] lengs) {
    int N;
    N = 0;
    for (int L : lengs) {
        N += L;
    }

    String[] result = new String[N];
    int k;
    k = 0;
    for (int i = 0; i < starts.length; i += 1) {
        arraycopy(A, starts[i], result, k, lengs[i]);
        k += lengs[i];
    }

    return result;
}
```

3. [3 points] Indicate what the main program in class `C` below would print. The program executes without errors. You need not use all lines.

```
class A {
    int x = 3;

    void f() {
        B me = (B) this;
        System.out.println("(a)");
        g(this);
        System.out.println("(b)");
        System.out.println(x);
        System.out.println(me.x);

    }

    static void g(A x) {
        System.out.println("A.g");
        x.h();
    }

    void h() {
        System.out.println("A.h " + this.x);
    }
}

class B extends A {
    int x = 42;

    void h() {
        System.out.println("B.h " + this.x);
    }
}

class C {
    public static void main(String... args) {
        A p = new B();
        p.f();
    }
}
```

```
(a)
________________________
A.g
________________________
B.h 42
________________________
(b)
________________________
3
________________________
42
________________________

________________________

________________________

________________________
```

4. [4 points] The interface `IntListPred` defines boolean-function-like objects:

```
interface IntListPred {
    /** Return true iff X and L satisfy (are true according to)
     *  this predicate. */
    boolean apply(int x, IntList L);
}
```

That is, if `B` is an object of a class that implements the `IntListPred` interface, then `B.apply(x, L)` returns true or false for any `int` value `x` and any `IntList` value `L`.

If `L` and `insertions` are `IntList`s and `pred` is an `IntListPred`, then the call `insertVals(L, insertions, pred)` is intended to return a list constructed from `L` as if by the following process, but non-destructively:

- Insert the head of `insertions` (call it `x`) just in front of the first sublist of `L` (call it `sl`) such that `x` and `sl` satisfy `pred`. (A sublist of `L` is one that results from 0 or more `.tail` operations, where `null`, the end of the list, is always the last sublist.)
- If there is no such position in `L`, no more values are inserted
- Likewise, if `insertions` is empty, no more values are inserted.
- After an insertion at `sl`, the operation continues with `sl` and the rest of `insertions`.

For example, suppose that `LT` is an `IntListPred` such that `LT.apply(a, L)` returns true iff the `IntList L` is null or the integer `a` is less than the head of `L`. Then the call

```
insertVals(IntList.list(1, 4, 6, 20, 31),
           IntList.list(3, 10, 15, 25, 32, 33),
           LT);
```

should return an `IntList` equal to

```
IntList.list(1, 3, 4, 6, 10, 15, 20, 25, 31, 32, 33).
```

Fill in the `insertVals` method to perform as described. It must be non-destructive. You need not use all the lines.

```
public static IntList insertVals(IntList L, IntList insertions,
                                 IntListPred pred) {
    if (insertions == null) {
        return L;
    } else if (pred.apply(insertions.head, L)) {
        return new IntList(insertions.head,
                           insertVals(L, insertions.tail, pred));
    } else if (L == null) {
        return null;
    } else {
        return new IntList(L.head,
                           insertVals(L.tail, insertions, pred));
    }
}
```

5. [1 point] According to legend, who asked whom "What goes on four feet in the morning, two feet at noon, and three feet in the evening?"

It is what the Sphinx asked Oedipus.

6. [4 points] A *chooser* is a type of object with a `choose` method that takes two `Object` values and returns a two-element `Object` array containing the two values in some order, the first value in the array being the *chosen* value, and the second being the *other* value. We represent choosers with the `Chooser` interface:

```
public interface Chooser {
    /** Return either { X, Y } or { Y, X }. The first value of
     *  returned array is called the chosen value, and the second
     *  is the other value.  This method may throw an exception if
     *  X and Y do not have the proper dynamic types for this Chooser. */
    Object[] choose(Object x, Object y);

    /** Returns the chosen value from X and Y. */
    Object chosen(Object x, Object y);
    /** Returns the other (non-chosen) value from X and Y. */
    Object other(Object x, Object y);
}
```

(a) [1 point] Write the abstract class `Judge`, which is a subtype of `Chooser` that implements the methods `chosen` and `other` to conform to their comments, but does not implement the `choose` method.

```
public abstract class Judge implements Chooser {

    public abstract Object[] choose(Object x, Object y);  // Optional

    public Object chosen(Object x, Object y) {
        return choose(x, y)[0];
    }

    public Object other(Object x, Object y) {
        return choose(x, y)[1];
    }
}
```

(b) [1 point] Write a class `BadJudge` that implements `Chooser` and also takes a `Chooser`, `C`, as the argument to its constructor. The resulting `BadJudge` always chooses the opposite of `C`, so that its chosen value is the other value chosen by `C` and its other value is the chosen value of `C`. For example, if `chooseFirst` is a kind of `Chooser` such that `chooseFirst.choose(x, y)` is always `{x, y}`, then after

```
Chooser bad = new BadJudge(chooseFirst);
```

the value of `bad.choose(x, y)` is always `{y, x}`.

You may (but need not) use the class `Judge` in your solution and assume that it works, regardless of what you wrote in part (a).

```
class BadJudge extends Judge {

      private Chooser c;

      public BadJudge(Chooser c0) {
          c = c0;
      }

      public Object[] choose(Object x, Object y) {
          Object[] r = c.choose(x, y);
          return new Object[] { r[1], r[0] };
      }
}
```

(c) [1 point] Write a static method `best` that, given a non-empty array, `A`, of `Object`s and a `Chooser`, `C`, returns the "best" value from the sequence `A[0]`, `A[1]`, ... The best value of a one-element sequence is the single value of the sequence. The best value of a $k + 1$ element sequence is either the best value of first k elements or the last value of the sequence, whichever is chosen by `C`.

```
/** Return the best value in A according to C.  A must be non-empty. */
public static Object best(Object[] A, Chooser C) {

    Object r;
    r = A[0];

    for (int i = 1; i < A.length; i += 1) {
        r = C.chosen(r, A[i]);
    }

    return r;
}
```

(d) [1 point] Implement the method `maxString` to obey its comment, using the `best` method from part (c). You may implement any additional classes you want to make it work. (Yes, one can pass a `String[]` as the parameter `A`, but the compiler will consider the static type of `A[i]` to be `Object`.)

You may (but need not) use the class `Judge` in your solution and assume that it works, regardless of what you wrote in part (a).

```
/** Return the largest value in WORDS, as determined by the
 *  .compareTo method on Strings. */
static String maxString(String[] words) {

     return (String) best(words, new StringChooser());
}


// Other classes here

class StringChooser extends Judge {

    public choose(Object x, Object y) {

        if (((String) x).compareTo((String) y) > 0) {
            return new Object[] { x, y };
        } else {
            return new Object[] { y, x };
        }
    }
}
```