CS61B, Spring 2006 **Final Examination** **P. N. Hilfinger**

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 10 problems on 18 pages. Officially, it is worth 50 points.

This is an open-book test. You have three hours to complete it. You may consult any books, notes, or other inanimate objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and lab section in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the building once or twice.

Your name: _____ Login: _____

Page 2. _____/7

Page 3. _____/2

Page 4. _____/2

Page 5. _____/3

Page 6. _____/3

Page 9. _____/2

Page 10. _____/2

Page 11. _____/2

Page 12. _____/5

Page 13. _____/4

Page 14. _____/6

Page 15. _____/3

Page 16. _____/3

Page 17–18. _____/6

TOT _____/50

Login of person to your Left: _____ Right: _____

Discussion section number or time: _____

Discussion TA: _____

Lab section number or time: _____

Lab TA: _____

1. [7 points] Answer "true" or "false" for each of the following statements about Java, and give a short explanation ($\leq$ 20 words) for each answer. *IMPORTANT:* you *must* give an explanation for each to get credit!

a. If all the constructors of class `Foo` are private, no `Foo`'s can ever be created.

b. Suppose that variable `L` is an `ArrayList<int[]>`, and `f` is a destructive operation. Then to insure that we don't lose the contents of the original `int[]` arrays in `L` after calling `f`, it suffices to write something like

```
/* Original value of L */
ArrayList<String> L0 = new ArrayList<String> (L);
f (L);
```

c. The legal Java fragment

```
byte b = 127;
b += 1;
```

causes `b` to overflow to 0.

d. If `x` and `y` are local variables of type `int`, then no possible definition of `swap` will cause `swap(x,y)` to exchange their values.

e. If `Foo.x` is a private static variable defined in class `Foo`, then there is no way that a method call `f(Foo.x)` outside the class can modify the value of `x` (that is, the value of `Foo.x` after the call will be `==` to its value before).

f. If `Foo.x` is a static variable then the assignment `Foo.x = null`, if legal, will always modify that same static variable, regardless of where (in what class) the assignment appears.

g. If `Foo.x` is a static field whose type is `AClass`, then the call `x.f()` must call a method whose body is either written in the class `AClass` or else is inherited from its parent class.

2. [4 points] For the two questions that follow, give the tightest bounds you can. In general, a smaller $O(\cdot)$ bound is better than a larger, and a $\Theta(\cdot)$ bound is better than a $O(\cdot)$ bound.

Both problems involve computing the cost of a sequence of *matrix multiplications.* To multiply an $m_1 \times m_2$ matrix by a $m_2 \times m_3$ matrix takes $m_1 \cdot m_2 \cdot m_3$ ordinary multiplications (if done in the usual way). If you multiply together an $m_1 \times m_2$ matrix, an $m_2 \times m_3$ matrix and an $m_3 \times m_4$ matrix, the time required depends on how you group the matrices—whether you multiply the first two and then multiply the result by the last ($m_1 \cdot m_2 \cdot m_3^2 \cdot m_4$ multiplications) or multiply the last two and then multiply that result by the first ($m_1 \cdot m_2^2 \cdot m_3 \cdot m_4$). With more than three matrices, life gets still more complicated. The questions are about two ways to calculate the smallest number of multiplications you'll need to multiply k matrices. We are *not* actually doing the multiplications, nor are we determining the best way to group the matrices for multiplication; we're just calculating how long it *would* take to do the computation optimally.

a.
```
/** The minimum cost of multiplying together k matrices of dimensions
 *  M[0]xM[1], M[1]xM[2], ..., and  M[k-1]xM[k], where all M[i] are
 *  positive, and 1 <= k = M.length-1. */
static long multCost (int[] M) {
  return multCost (M, 0, M.length-1);
}

private static long multCost (int[] M, int L, int U) {
  if (L+1 >= U)
    return 0;
  long min;
  min = Long.MAX_VALUE;
  for (int i = L+1; i < U; i += 1)
    min =
      Math.min (min,
                multCost (M, L, i) + multCost (M, i, U) + M[L]*M[i]*M[U]);
  return min;
}
```

In the worst case, as a function of `M.length`, how many times do we compute the subexpression `M[L]*M[i]*M[U]`?

Problem continues on the next page.

b. Here is an alternative algorithm for the same problem:

```
/** The minimum cost of multiplying together k matrices of dimensions
 *  M[0]xM[1], M[1]xM[2], ..., and  M[k-1]xM[k], where all M[i] are
 *  positive, and 1 <= k = M.length-1. */
static long multCost (int[] M) {
  return multCost (M, 0, M.length-1, new long[M.length][M.length]);
}

private static long multCost (int[] M, int L, int U, long[][] work) {
  if (L+1 >= U)
    return 0;
  if (work[L][U] == 0) {
     long min;
     min = Long.MAX_VALUE;
     for (int i = L+1; i < U; i += 1)
       min =
         Math.min (min,
                   multCost (M, L, i, work) + multCost (M, i, U, work)
                       + M[L]*M[i]*M[U]);
     work[L][U] = min;
  }
  return work[L][U];
}
```

Again, in the worst case, as a function of `M.length`, how many times do we compute the subexpression `M[L]*M[i]*M[U]`?

3. [6 points] For each task below, list one or more of the following search-related data structures that might reasonably be used as part of an efficient solution of the problem for large inputs. In each case, say concisely what the data structure would be used for (that is, what part it would play in the program).

1. a trie
2. hash table
3. priority queue (or heap)
4. stack (array implementation)
5. queue (circular buffer implementation)
6. sorted array
7. red-black tree (or B-tree).

"Appropriate" means that it will lead to an algorithm that behaves well (doesn't slow down inordinately) as input becomes large.

a. Implementing the Lisp (or Scheme) reader. This module takes identifiers (strings) and maps them to "symbols" (some kind of object) so that the same string always stands for the same symbol object and differing strings stand for different symbol objects. It operates throughout the execution of a Lisp program, creating new symbols as new identifiers are encountered during the reading of Lisp expressions from input.

b. Computing how best to intercept a fleeing car, knowing a map of the city and the presumed route of the car. We are interested here in the data structure used in doing this computation, assuming the data are already present.

c. Storing information about a set of points along a line that can be modified (that is, new points may be added or deleted from the set), and one can find points given an approximate position.

Problem continues on next page

d. Managing a discrete-event simulation. In this kind of simulation, the program simulates the occurrence and interaction of events that occur at particular (simulated) times in the order in which they would occur in "real life." Any event, when simulated, may cause new events that will happen later (e.g., the event of a machine tool starting will cause the later event of a finished piece of metal leaving that work station for the next). We say the times are simulated because the program processes events as fast as possible, essentially skipping over periods when nothing happens.

e. Keeping track of a set of tasks to be completed in the order they are received.

f. Storing a sequence of records that is often accessed in sorted order and often added to, but in which almost all additions sort before all previous entries.

4. [6 points] [DON'T PANIC: There's a lot of reading here, but not all that much work.] You've been asked to help implement part of a library for creating contraptions. A contraption consists of a network of operational *components* that receive values from each other, perform some operations on them (possibly with side effects), and may send results on to other components. A component receives input through some number of named *ports* (how many ports and what their names are depends on the type of component). It has one output.

The interfaces `Component` and `Port` describe all this:

```
interface Component {
   /** The input port named INPUTNAME. */
   Port getPort (String inputName);
   /** Connect my output to TOPORT. */
   void connect (Port toPort);
}

interface Port {
   /** Send input VAL to THIS. */
   void sendInput (Object val);
}
```

At most one output may be connected to any given port, but any given output may be connected to any number of ports (and must send each of its values to all of them).

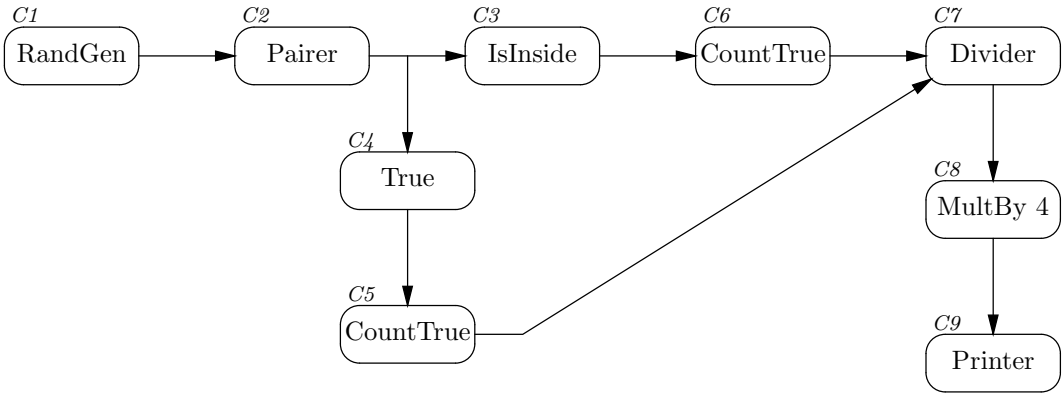
The class `Contraption` represents a set of named components

```
public class Contraption {
   /** Add COMPONENT to THIS, labeling it with NAME, which must be
    *  unique within THIS. */
   void add (String name, Component component) { ... }
   /** The Component named NAME, or null if none is present. */
   Component getComp (String name) { ... }

   /** Connect the output of the component named SOURCENAME
    *  in THIS to the input port named SINKPORTNAME of the component
    *  named SINKNAME in THIS.  (Convenience method). */
   void connect (String sourceName, String sinkName, String sinkPortName) {
     getComp (sourceName).connect (getComp (sinkName).getPort (sinkPortName));
   }

}
```

For example, here is how we might build a contraption that approximates π by generating a stream of random points in the unit square, and counting the number that fall inside the circle of radius 0.5 whose center is in the center of the unit square. This circle has area $\pi/4$, so we expect that to be the fraction of random points that fall in the circle. Multiplying that fraction by 4 should therefore give us successively better approximations of π. In this example, we assume that components that have a single input port name it just `"In"`.

```
Contraption C = new Contraption();
C.add ("C1", new RandGen ());
C.add ("C2", new Pairer ());
C.add ("C3", new IsInside ());
C.add ("C4", new True ());
C.add ("C5", new CountTrue ());
C.add ("C6", new CountTrue ());
C.add ("C7", new Divider ());
C.add ("C8", new MultBy (4.0));
C.add ("C9", new Printer ());
```

```
// Continued
C.connect ("C1", "C2", "In");
C.connect ("C2", "C3", "In");
C.connect ("C2", "C4", "In");
C.connect ("C4", "C5", "In");
C.connect ("C3", "C6", "In");
C.connect ("C5","C7","Divisor");
C.connect ("C6","C7","Dividend");
C.connect ("C7", "C8", "In");
C.connect ("C8", "C9", "In");
```



Here we have a RandGen sending random numbers to a Pairer, which spits out one pair of numbers for each two it receives. The pairs go to an insideness tester, which spits out a true value (the object `Boolean.TRUE`) for each pair it receives that is inside the circle, and a false for each that isn't. The true values are counted by one of two CountTrues, which take in a stream of trues and falses, and output the cumulative number of trues received each time. By routing the output of the pair generator through a component that generates a value of true for every input it receives, and then counting the number of these true values, we get a count of the total number of pairs. The two counts then get divided, multiplied by 4 and printed. The Printer component never sends anything to its output, but simply prints each input it receives. The result is that the contraption prints a stream of (presumably) better and better approximations of π.

To make this all do something, we assume that somewhere there is a loop

```
while (...)
    C.getComp ("C1").getPort ("In").sendInput ("");
```

and that the RandGen component spits out a random number each time it receives an input (any input) on its 'In' port.

a. Fill in the class `Contraption` (use extra sheets as needed):

```
public class Contraption {
   /** Add COMPONENT to THIS, labeling it with NAME, which must be
    *  unique within THIS. */
   void add (String name, Component component) {
      // FILL IN









   }

   /** The Component named NAME, or null if none is present. */
   Component getComp (String name) {
      // FILL IN









   }

   // OTHER METHODS, FIELDS, CONSTRUCTORS HERE, IF NEEDED. */









}
```

Continued on next page

b. Fill in the `Pairer` class, which takes a stream of `Objects` and outputs a pair (type Object[] with two members) for every two inputs it receives. That is, after receiving one input, say 0.75 (as a `Double`), it outputs nothing, and then after receiving a second input, say 0.3, it outputs the `Double[]` value `{0.75,0.3}`.

```
public class Pairer implements Component                    {
   public Pairer () {
      // FILL IN



   }

   // FILL IN OTHER METHODS, FIELDS, CLASSES HERE */
```

```
}
```

Continued on next page

c. Fill in the `Divider` class, which accepts a `Double` input named `"Dividend"` and another named `"Divisor"` (in either order) and outputs their quotient as a `Double` when it has received both inputs. (So, if it first receives 25 on its Dividend port, it does nothing; when it next receives 50 on its Divisor port, it outputs 0.5 as a `Double`.) Do *not* worry about the situation in which you receive (for example) two Dividends before receiving a Divisor.

```
public class Divider implements Component                    {
    public Divider () {
       // FILL IN




    }

   // FILL IN OTHER METHODS, FIELDS, CLASSES HERE */
```

```
}
```

5. [5 points] Each of the parts below shows the states of an array at major intermediate points in a sorting algorithm. In each case, the array numbered '0)' is the original input array. For each part, indicate (in the space on the right) what sorting algorithm is being illustrated. Be as specific as possible.

a.
0) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
1) 2261 4411 5103 1163 9914 3715 6035 9797 0608 7188
2) 5103 0608 4411 9914 3715 6035 2261 1163 7188 9797
3) 6035 5103 1163 7188 2261 4411 0608 3715 9797 9914
4) 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914

b.
0) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
1) 0608 1163 2261 3715 4411 5103 6035 7188 9914 9797
2) 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914

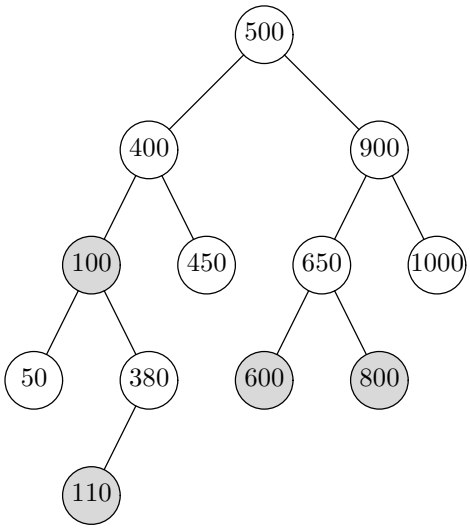
c.
0) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
1) 5103 9914 0608 3715 2261 6035 7188 9797 1163 4411
2) 0608 3715 5103 9914 2261 7188 6035 9797 1163 4411
3) 0608 2261 3715 5103 6035 7188 9797 9914 1163 4411
4) 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914

d.
0) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
1) 0608 5103 9914 3715 6035 2261 9797 7188 1163 4411
2) 0608 1163 5103 9914 3715 6035 2261 9797 7188 4411
3) 0608 1163 2261 5103 9914 3715 6035 9797 7188 4411
4) 0608 1163 2261 3715 5103 9914 6035 9797 7188 4411
5) 0608 1163 2261 3715 4411 5103 9914 6035 9797 7188
6) 0608 1163 2261 3715 4411 5103 6035 9914 9797 7188
7) 0608 1163 2261 3715 4411 5103 6035 7188 9914 9797
8) 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914

e.
0) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
1) 9914 9797 5103 7188 6035 2261 0608 3715 1163 4411
2) 9797 7188 5103 4411 6035 2261 0608 3715 1163 9914
3) 7188 6035 5103 4411 1163 2261 0608 3715 9797 9914
4) 6035 4411 5103 3715 1163 2261 0608 7188 9797 9914
5) 5103 4411 2261 3715 1163 0608 6035 7188 9797 9914
6) 4411 3715 2261 0608 1163 5103 6035 7188 9797 9914
7) 3715 1163 2261 0608 4411 5103 6035 7188 9797 9914
8) 2261 1163 0608 3715 4411 5103 6035 7188 9797 9914
9) 1163 0608 2261 3715 4411 5103 6035 7188 9797 9914
10) 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914

6. [4 points]

a. Given the following red-black tree (shaded nodes are red):



Draw the corresponding (2,4) tree.

b. If a (2,4) tree has depth h (that is, the (empty) leaves are at distance h from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?

7. [6 points] The call `riffle (L1, L2, PATN` takes three `IntLists` and uses `PATN` to specify how to riffle-shuffle the items of `L1` and `L2` together (destructively). As usual, an `IntList` consists of a `head` field of type `int` and a `tail` field of type `IntList`. The `PATN` argument must be a non-empty sequence of positive integers.. If it contains the integers k_1, k_2, ..., k_n, then the result of `riffle (L1, L2, PATN)` consists of k_1 items from `L1`, followed by k_2 from `L2`, followed by k_3 from `L1`, etc. If the `PATN` list is exhausted, it repeats (i.e., k_{n+1} is, in effect, k_1). Whenever one of the lists `L1` or `L2` is exhausted, the remaining list is simply appended as the end of the result, ignoring `PATN`. For example, if `L1` contains (3,1,5,9,6,5,8,9,7,9,3), `L2` contains contains (1,4,2,3,5), and `PATN` contains (1,2,3), then the result is (3,1,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3). Fill in the blanks below to produce this effect.

```
static IntList riffle (IntList L1, IntList L2, IntList pattern) {

  if (_____________)
    return L2;

  if (_____________)
    return L1;
  IntList result, last, p;
  result = last = p = null;

  while (_____________________________________) {
    if (p == null)
      p = pattern;
    if (result == null)

      _____________________________ = L1;
    else

      _____________________________ = L1;

    for (int i = 0; i < p.head && _____________________; i += 1) {

      last = _____________;
      L1 = L1.tail;
    }
    IntList t = L1; L1 = L2; L2 = t;
    p = p.tail;
  }
  if (L1 == null)

    ___________________________________________;
  else

    ___________________________________________;
  return result;
}
```

8. [6 points] Answer each of the following *briefly*. Where a question asks for a yes/no answer, give a brief reason for the answer (or counter-example, if appropriate).

a. Assume that f and g are non-negative-valued functions. If $f(x) \in \Theta(g(x))$, is $2^{f(x)} \in \Theta\left(2^{g(x)}\right)$?

b. Assume that f and g are positive-valued functions. If $f(x) \in \Omega(g(x))$, is $\lg f(x) \in \Omega(\lg g(x))$?

c. Your program contains an `ArrayList<SomeType>`, initially empty. To it, you add N items. Next, you add an additional N items, all of which are larger than any of the initial N items. You add a third group of N items, all of which are larger than the preceding $2N$, and finally a fourth group, all larger than the preceding $3N$. Suppose you now sort this list of $4N$ items using insertion sort. About how long will this take, in the worst case, assuming that it takes time T to sort N items by insertion sort in the worst case?

More parts on the next page.

d. Your program isn't quite working properly. You discover that it contains a statement

```
L.head = y;
```

that sometimes inserts a null pointer into `L.head`, and in your program that is not supposed to happen. This line is executed millions of times in a typical run of the program. You'd like to be able to find out what is going on when this happens. What's the quickest way to do so, preferably with minimal modification to your program and without generating reams of debugging output?

e. You are using an array with N elements to hold a heap (maximal element on top) of in the usual fashion. The heap is initially empty, and you add N items to it one at a time in descending order (re-heapifying after each addition). How many comparisons between items on the heap are required to do all N insertions?

f. What does the following computation compute?

```
int whatisthis(int x) {
    x = (0x55555555 & x) + (0x55555555 & (x >>> 1));
    x = (0x33333333 & x) + (0x33333333 & (x >>> 2));
    x = (0x0f0f0f0f & x) + (0x0f0f0f0f & (x >>> 4));
    x = (0x00ff00ff & x) + (0x00ff00ff & (x >>> 8));
    x = (0x0000ffff & x) + (0x0000ffff & (x >>> 16));
    return x;
}
```

9. [1 point] Where can you expect to find the highest concentration of janjaweed?

10. [6 points] In the following (partial) implementation of a set (using a binary search tree), there are several errors at *some* of the points indicated. Correct these *as succinctly as possible.* Do *not* correct things that don't need to be corrected (not all indicated points are erroneous).

```
/** A set of objects of type ITEM. */
public class BST<Item extends Comparable<Item>> {

  /** An empty set */
  public BST () {
    root = null;
    size = 0;
  }

  /** Add item X to THIS. Has no effect if X is already present. */
  public void add (Item X) {
    size += 1;                  // ERROR?
    root = add (root, X);
  }

  /** The number of items in THIS. */
  public int size () {
    return size;                // ERROR?
  }

  /** Remove item X from THIS.  Has no effect if X is not present. */
  public void remove (Item X) {
    size -= 1;                  // ERROR?
    root = remove (root, X);
  }

  /** True iff X is present in THIS. */
  public boolean contains (Item X) {
    Implementation removed. Assume it works.
  }

  private Node add (Node T, Item X) {
    Implementation removed. Assume it works.
  }
```

Continues on next page.

```
  /* Remove item X from T, returning the modified T. */
  private Node remove (Node T, Item X) {
    if (T == null)
      return null;
    if (X > T.label)                // ERROR?
      T.left = remove (T.left, X);
    else if (X > T.label)           // ERROR?
      T.right = remove (T.right, X);
    else if (T.left.label == null)  // ERROR?
      return T.right;
    else if (T.right.label == null) // ERROR?
      return T.left;
    else
      T.right = removeSmallest (T, T.right);
    return T;
  }

  /** Return the result of removing the smallest node
   *  from T (destructively), placing the label of the
   *  removed node in NEWLOCATION. */
  private Node removeSmallest (Node newLocation, Node T) {
    if (T.left == null) {
      newLocation.label = T;    // ERROR?
      return T.right;           // ERROR?
    } else {
      T.right = removeSmallest (T, T.right); // ERROR?
      return T;                 // ERROR?
    }
  }

  private class Node {
    Item label;
    Node left, right;
    Node (Item x) {
      label = x;
      left = right = null;
    }
  }

  private Node root;
  private int size;
}
```