CS61B, Fall 2008 **Final Examination Solutions** **P. N. Hilfinger**

1. [4 points] Answer "true" or "false" for each of the following statements about Java, and give a short explanation (≤ 20 words) for each answer. *IMPORTANT:* you *must* give an explanation for each to get credit!

a. If `c` is a `char` variable, then the (legal Java) statements

```
c ^= (c >> 8);
c ^= (c >> 4);
c ^= (c >> 2);
c ^= (c >> 1);
c &= 1;
```

will set `c` to 1 if its original value had an odd number of 1 bits, and otherwise to 0.

True. This series of operations sets the low-order bit of `c` *to the xor of all 16 bits in* `c`*, and 0s the rest.*

b. Assuming that `inp` is a `Scanner`, the following (legal Java) fragment for reading commands and calling an appropriate method to deal with them won't work

```
String command = inp.next ();
if (command == "quit")
   doQuit ();
if (command == "add")
   doAdd ();
// etc.
```

True. `==` *on String is not the same as* `.equals`.

c. If `C` is a private static nested class defined inside another class `D`, then there is no way that a method defined in a different package can ever call any method of any object of type `C`.

False. Such a class can still implement a public interface and be returned from any public method of `D`.

d. The second method below is a non-destructive version of the first method.

```
double[][] normalize (double[][] A) {
    for (double[] row : A)
        for (int i = 1; i < row.length; i += 1)
            row[i] /= row[0];
    return A;
}

double[][] normalize (double[][] A0) {  // Non-destructive ?
    double[][] A = new double[A0.length][];
    System.arraycopy (A0, 0, A, 0, A.length);
    for (double[] row : A)
        for (int i = 1; i < row.length; i += 1)
            row[i] /= row[0];
    return A;
}
```

False. The rows of `A` *are shared with* `A0`*;* `A[i] == A0[i]`.

2. [4 points] Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be three Java statements that reference no local variables or parameters except for constant (final) ones, and let `x` be an integer variable with $0 \leq$ `x` ≤ 2. Write a statement that executes $\mathcal{A}$ if `x==0`, $\mathcal{B}$ if `x==1`, and $\mathcal{C}$ if `x==2`. Add any additional declarations you need to make it work. However, you may not use any Java library calls, nor any of the keywords **if**, **while**, **for**, **try**, or **switch**, nor the operators `?:` anywhere in your solution.

Use OOP: the statement is

```
actions[x].act ();
```

where

```
interface Actor {
   void act ();
}

Actor[] actions = new Actor[] {
  new Actor () { public void act () { 𝒜 },
  new Actor () { public void act () { ℬ },
  new Actor () { public void act () { 𝒞 }
};
```

Or, if you prefer not to use anonymous classes, make three classes implementing `Actor` *with the contents above.*

3. [6 points] The TEX program has an algorithm for optimally inserting line breaks into sequences of words so as to make the "best" paragraph out of these words. Its idea of "best" involves assigning penalties to each possible line of text, and finding the line breaks that lead to the smallest sum of penalties. At some point, we want to compute the best possible score (smallest sum of penalities) for a given sequence of words. Here's a simple method:

```
/** The cost of the best paragraph that can be formed by breaking up
 *  the words WORDS[START], WORDS[START+1], ... WORDS[END-1] into
 *  lines. */
static int cost (String[] words, int start, int end) {
    int bestScore;
    bestScore = penalty (words, start, end);
    for (int k = start; k < end; k += 1)
        bestScore = Math.min (bestScore,
                              cost (words, start, k) + cost (words, k, end));
    return bestScore;
}
```

where `penalty` is some function defined elsewhere that computes the penalty for forming a single (unbroken) line from the words between START and END.

- As written, this function is rather slow. Rewrite it to speed it up as much as you can.

  I'll assume that all penalties are non-negative.

```
static int cost (String[] words, int start, int end) {
    int[][] cache = new int[words.length+1][words.length+1];
    for (int[] r : cache)
       Arrays.fill (r, -1);
    cost (words, start, end, cache);
}

static int cost (String[] words, int start, int end, int[][] cache) {
    if (cache[start][end] == -1) {
       int bestScore;
       bestScore = penalty (words, start, end);
       for (int k = start; k < end; k += 1)
           bestScore = Math.min (bestScore,
                                 cost (words, start, k, cache)
                                 + cost (words, k, end,  cache));
       cache[start][end] = bestScore;
    }
    return cache[start][end];
}
```

- Assume that `penalty`(W, S, E) requires time $\Theta(E - S)$. What is the running time of your `cost`(W, S, E) as a function of $E - S$?

  $\Theta((E - S)^3)$. *We call* `penalty` $(E - S)^2$ *times and its cost is bounded by* $E - S$ *each time.*

4. [1 point] If I could see a certain object from the center of the earth at sunset this evening, it would appear to be moving at about 2300 mph and accelerating at roughly right angles to its path at about 0.009 ft/sec^2. What is it?

The moon.

5. [7 points] In this problem, you are to design and implement a scheduler for discrete event simulation system. Such systems allow programs to maintain a sequence of *events,* each scheduled to take place at a particular (simulated) time. The system repeatedly finds the next scheduled event, sets the "virtual clock" to the time at which that event is scheduled to take place, and executes whatever action the event calls for (which may include adding more events to the queue or removing or rescheduling events already there). So for example, to simulate traffic, you might seed the process with a bunch of events that consist of cars arriving at the next intersection in their route. As each event "fires," the virtual clock skips ahead to the time of arrival, and the car's action might do things such as stop the car, schedule an event of starting again at the time the traffic signal changes, etc.

Programmers writing such simulations must be able to do the following:

- *Read* the current value of the virtual clock.
- *Schedule* an event—that is, to indicate that some piece of code of their own choosing is to execute at a specified virtual time.
- Receive a *token* of this future event when they schedule it—an object of some kind that allows their program to manipulate that event if necessary in certain ways:
  - To *cancel* the event so that it does not occur.
  - To *reschedule* the event at a different (but still future) time.
  - To *query* the currently scheduled time for the event.

To run the simulation, the programmer schedules a few initial events, and then calls some method of the simulator (here, we call it `simulate`), which does the rest.

This problem is in two parts, starting on the next page. In part (a), you are to write a package that meets these requirements. In part (b), you are to use it to translate a piece of pseudocode into Java. We'd like you to keep things at a reasonably high level; use the Java library to supply a reasonable data structure for storing your events rather than supplying your own implementation of a priority queue. Your answer to part (a) must be general-purpose; it must know nothing about the particular application in part (b).

a. Complete the package below. Add any public or private classes, interfaces, methods, and constructors you please, and any private or package private fields. Again, don't roll your own data structures; feel free to use anything in the Java library.

```
package simulate;

public class Simulation {
   /** A Simulation initialized with no scheduled events and virtual
    *  time 0. */
   public Simulation () {
       time = 0;
   }

   /** Simulate events (starting with those currently scheduled)
    *  starting at TIME0 and ending when there are no more events or
    *  the virtual time of the next event reaches or passes TIME1.
    *  First, cancel any events scheduled before TIME0. */
   public void simulate (int time0, int time1) { // FILL IN
       queue.headMap (time0).clear ();
       time = time0;
       while (time < time1 &&  queue.size () != 0) {
           time = queue.firstKey ();
           ArrayList<Event> events = queue.first ();
           Event e = events.remove (0);
           if (events.isEmpty ())
               queue.remove (time);
           e.act ();
       }
   }

   private final TreeMap<Integer, ArrayList<Event>> queue =
      new TreeMap<Integer, ArrayList<Event>> ();
   private int time;
```

Continues on next page.

```
   public int getTime () {
      return time;
   }

   public schedule (Event e, int time) {
      e.time = time;
      if (!queue.containsKey (time)) {
          queue.put (time, new ArrayList<Event>());
      }
      queue.get (time).add (e);
   }

   public cancel (Event e) {
      ArrayList<Event> events = queue.get (e.time);
      if (events != null) {
          events.remove (e);
          if (events.size () == 0)
              queue.remove (e.time);
      }
   }

   public reschedule (Event e, int time) {
      if (time < this.time) throw new IllegalArgumentException ();
      cancel (e);
      schedule (e, time);
   }
}

public abstract class Event {
   int time;

   public int getTime () { return time; }

   abstract protected void act ();
}
```

b. Translate the following pseudocode into Java (in the anonymous package), using your package from part (a). Assume that any functions mentioned but not defined below are defined somewhere.

```
Schedule a call to arriveOffice () at time 140 + random().
Schedule a call to eatBreakfast () at time 90.
Schedule a call to arriveSchool () at time 120.
Simulate times 100–200.
```

Where we define:

```
arriveOffice ():
    Print the current simulation time.

arriveSchool ():
    If the value of discussWithTeacher () > 5 minutes, reschedule
        the arriveOffice to be later by discussWithTeacher () - 5 minutes.
    Schedule a call to pickUpChild () at getDismissalTime ().
```

```
final Simulation s = new Simulation ();

Event arriveSchool = new Event () {
   protected void act () {
     int d = discussWithTeacher ();
     if (d > 5)
         s.reschedule (arriveOffice, d-5 + arriveOffice.getTime ());
     s.schedule (new Event () { protected void act () { pickUpChild (); } },
                 getDismissalTime ());
  }
};
Event arriveOffice = new Event () {
    protected void act () {
        System.out.println (s.getTime ());
    }
}

s.schedule (arriveOffice, 140 + random ());
s.schedule (arriveSchool, 120);
s.schedule (new Entry { protected void act () { eatBreakfast (); } }, 90);

s.simulate (100, 200);
```

6. [10 points] The following questions involve sorting. Warning: do not assume that the algorithms illustrated always conform exactly to those presented in the reader and lecture notes. We are interested in whether you understand the major ideas behind the algorithms. Where the question asks for a reason, you *must* provide an explanation to get credit.

a. T. C. Pits[1] has recently acquired a Sorting Adversary Machine (which of course we'll call SAM). Given a sorting algorithm, this machine will figure out an arbitrarily large sequence of test data sets of increasing size that make the algorithm look as bad or good as possible (depending on its dial settings) and determine how the algorithm scales with increasing input size for these test data sets. "This algorithm you gave me can't be quicksort!" he exclaims, "It scales as the square of the input size instead of $O(N \lg N)$." Is he right? If not, what kind of test data sets could SAM have come up with to get this scaling from quicksort?

He's wrong. If the pivot that quicksort selects is always the smallest data item, it becomes like a straight selection sort, $O(N)$ time for each of $O(N)$ iterations.

b. When he applied SAM to a second sorting algorithm and set the dial for "best cases." T. C. Pits found out that it ran in time $O(N)$, where N is the number of records to be sorted. Could this algorithm have been heapsort?

No. Heapsort requires $\Theta(\lg N)$ time to move each item, even for sorted data.

c. What sorting algorithm does the following illustrate? The first line is the input.

```
cfej gihi jfef a hiha ecaa egeb giei bji jjb gji bbdi ibbi ehf bhhd
a bji jjb gji ehf hiha ecaa egeb bhhd jfef gihi giei bbdi ibbi cfej
a ecaa jjb ibbi bbdi egeb jfef giei cfej ehf hiha bhhd gihi bji gji
a ibbi bbdi ecaa jfef cfej egeb ehf bhhd giei hiha gihi jjb bji gji
a bbdi bhhd bji cfej ecaa egeb ehf giei gihi gji hiha ibbi jfef jjb
```

LSD radix sort.

d. What sorting algorithm does the following illustrate? The first line is the input.

```
cfej gihi jfef a hiha ecaa egeb giei bji jjb gji bbdi ibbi ehf bhhd
a bji bbdi bhhd cfej ecaa egeb ehf gihi giei gji hiha ibbi jfef jjb
a bbdi bhhd bji cfej ecaa egeb ehf gihi giei gji hiha ibbi jfef jjb
a bbdi bhhd bji cfej ecaa egeb ehf giei gihi gji hiha ibbi jfef jjb
```

MSD radix sort.

e. What sorting algorithm does the following illustrate? The first line is the input.

```
cfej gihi jfef a hiha ecaa egeb giei bji jjb gji bbdi ibbi ehf bhhd dab
cfej gihi a jfef ecaa hiha egeb giei bji jjb bbdi gji ehf ibbi bhhd dab
a cfej gihi jfef ecaa egeb giei hiha bbdi bji gji jjb bhhd dab ehf ibbi
a cfej ecaa egeb giei gihi hiha jfef bbdi bhhd bji dab ehf gji ibbi jjb
a bbdi bhhd bji cfej dab ecaa egeb ehf giei gihi gji hiha ibbi jfef jjb
```

Merge sort.

[1]The Celebrated Programmer In The Street.

7. [12 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. Suppose that $g(n) = f_n(n)$ for $n = 0, 1, \ldots$ (so $g(0) = f_0(0)$, $g(1) = f_1(1)$, etc.), and that $f_n(x) \in \Theta(x^n)$ for all the f_n. Show that these facts still allow $g(n) \in O(1)$.

   Just because f_n eventually grows as x^n doesn't prevent it from having a particular value at $x = n$, such as 1.

b. Assume that we have a heap with its largest element at the root. If this heap has at least 8 items in it (all different), must the second smallest item be at the bottom? Either explain why it must, or give a counterexample.

   No. The second-smallest item can be one layer up, too, as in

```
      8
   6     7
  2  3  4  5
1
```

c. Starting with a sorted array, we randomly swap K pairs of *adjacent* items (that is, given an N-element array, we select K random (not necessarily distinct) integers, $0 < i < N$, in succession and swap elements i and $i - 1$ for each of these random i.) After we're done, how many comparisons will insertion sort will have to make in the worst case to set things right (as a function of K and N)?

   There are at most K inversions, so insertion sort will take time $O(K + N)$.

More parts on the next page.

d. You are working a project with Andrea, your partner, and using Subversion to keep track of your work. You keep the "official" version, containing the sum of your efforts and Andrea's, in repository `svn+ssh://···/ourstuff/trunk`. You keep a copy of that directory with the changes you are working on in `svn+ssh//···/ourstuff/branches/me`, which is reserved to you, and Andrea keeps a copy in `svn+ssh//···/ourstuff/branches/andrea`. Every now and then, you want to incorporate your changes into the trunk version. Describe how you (and Andrea) do this so that nobody's changes get lost and any conflicting changes get worked out.

Commit your branch. In a working copy of the trunk, do

`svn merge -r`$N : M$ `svn+ssh//$\cdots$/ourstuff/branches/me`

where N and M are revision numbers of your branch. Where the merge reports a conflict, edit the offending file to resolve the matter, and tell Subversion `svn resolved` *FILE. Finally, commit the new trunk.*

e. The rules of Java make the access to `a.prot_x` in the code below legal, while `b.prot_x` is illegal. What is the rationale for these language rules?

```
package A;                  | package B;
                            |
public class P {            | class Q extends A.P {
   protected int prot_x;    |    f (R a) {
   ...                      |       a.prot_x = 3; ...
}                           |    }
                            |    g (A.P b) {
                            |       b.prot_x = 3;   // ERROR!
                            |    }
                            | }
                            | class R extends Q { ... }
```

Being protected means that `prot_x` *is available for* `Q` *and its descendents to use in their own way. Since the static type of* `b` *is* `A.P`*, its dyanmic type is not constrained to be one of Q's descendents.*

f. Give a weighted undirected graph containing three vertices for which the shortest-path tree from a particular vertex differs from the result of Prim's algorithm.

```
        S
       / \
     2/   \4
     /     \
    A-------B
        3
```

The shortest path tree for `S` *contains the 2 and 4 edges. The MST from Prim's algorithm would contain the 2 and 3 edges.*

8. [7 points] In the following syntactically correct set implementation using a balanced binary search tree, there are several errors at *some* of the points indicated by "`// ERROR` *N*`?`," for $1 \le N \le 7$. When "`// ERROR` *N*`?`" occurs alone on a line, it means that there *might* be missing statements. Error markers with the same value of *N* lines that all (may be) part of the same error, and (might) need to be corrected as a group. Correct true errors *as succinctly as possible.* Do *not* correct things that don't need to be corrected (not all indicated points are erroneous). HINT: The code is long, but the data structure should be familiar. You should only have to look at a little context and the comments on the methods and fields involved to fix line marked ERROR, and to ignore most of the rest of the code.

```
/** A set of comparable objects.  The implementation guarantees that
 *  the add and contains methods operate in lg(size()) time. */
public class BalTreeSet<T extends Comparable<? super T>> {

    private Node<T> root;
    private int size;
    /** Work area used to return whether an item is found. */
    private final boolean[] found = new boolean[1];

    /** Initially empty set. */
    public BalTreeSet () { root = null; size = 0; }

    /** Number of (distinct) items I contain. */
    public int size () { return size; }

    /** True iff the value X has been added to me. */
    public boolean contains (T x) {
        if (x == null)
            return false;
                                        // OK
        root = root.findInsert (x, false, found);
        return found[0];
    }

    /** Insert X into me, if it is not already present.  Returns true
     *  if X was not previously present (so that my contents changed). */
    public boolean add (T x) {
        if (x == null)
            throw new IllegalArgumentException ("set may not contain null");
        if (root == null) {
            root = new Node<T> (x);
            size = 1;                   // ERROR 2 (missing)
            return true;
        }
        root = root.findInsert (x, true, found);
        if (!found[0])                  // ERROR 3 (missing !)
            size += 1;
        return !found[0];               // ERROR 3
    }
```

Continued.

```
/** A BST node that obeys the Balance Invariant: the heights of the
 *  left and right subtrees of any node differ by no more than 1.
 *  Each subtree keeps track of its height.  Null represents an
 *  empty tree.  */
private static class Node<T extends Comparable<? super T>> {
    T val;
    Node<T> left, right;
    int height;

    /** A new Node with empty left and right children and label VAL. */
    Node (T val) {
        this.val = val;
        this.left = this.right = null;
        this.height = 1;
    }

    /** Set FOUND[0] to true iff my subtree initially contains X,
     *  If INSERT, then add X if it is not already there.   Returns
     *  the resulting (balanced) tree in either case. */
    Node<T> findInsert (T x, boolean insert, boolean[] found) {
        int c = x.compareTo (val);
        if (c == 0) {
            found[0] = true;
            return this;
        }
        if (c < 0) {
            if (left == null) {
                found[0] = false;
                if (insert)
                    left = new Node<T> (x);
            } else
                left = left.findInsert (x, insert, found);  // ERROR 4
                                                   // (left. was missing)
        }
        else if (right == null) {
            found[0] = false;
            if (insert)
                right = new Node<T> (x);
        } else
            right = right.findInsert (x, insert, found);    // ERROR 4
                                                  // (right. was missing)
        if (insert && ! found[0]) {
            computeHeight ();
            return rebalance ();
        } else
            return this;
    }
```

```
/** The height of N, where a null tree has height 0. */
private static int height (Node<?> n) {
    if (n == null)
        return 0;
    else
        return n.height;
}

/** Adjust my height as necessary, assuming that my children
 *  have correct heights. */
private void computeHeight () {
    height = 1+Math.max (height (left), height (right)); // ERROR 5 (1+)
}

/* Assuming that my left and right subtrees individually obey
 * the Balance Invariant (above), and these subtrees' heights are
 * within 2 of each other, reestablish the Invariant for me.
 * When my subtrees are not in balance, there are two cases
 * (plus mirror images):
 *
 *       this                this             this, r: nodes.
 *      /   \                /  \             T(M): subtree T of height M
 *     A(N)  r             A(N) r
 *          / \                 / \
 *        B(N) C(N+1)      B(N+1) C(N)
 *
 *  In the left case, rotate left around this.  In the right case,
 *  rotate right around r and then left around this.  (Likewise for
 *  the two cases where the left subtree is the deeper. */
Node<T> rebalance () {
    if (height (right) > height (left) + 1) {
        if (height (right.left) > height (right.right))  // OK
            right = right.rotateRight ();
        return rotateLeft ();
    } else if (height (left) > height (right) + 1) {
        if (height (left.right) > height (left.left))    // OK
            left = left.rotateLeft ();
        return rotateRight ();
    } else
        return this;
}
```

Continued.

```
        /** Rotate the tree right around me, returning new root of subtree. */
        Node<T> rotateRight () {
            Node<T> n = left;
            left = left.right;
            n.right = this;
            computeHeight ();                   // ERROR 7 (statements switched)
            n.computeHeight ();                 // ERROR 7
            return n;
        }

        /** Rotate the tree left around me, returning new root of subtree. */
        Node<T> rotateLeft () {
            Node<T> n = right;
            right = right.left;
            n.left = this;
            computeHeight ();                   // ERROR 7
            n.computeHeight ();                 // ERROR 7
            return n;
        }
    }
}
```