Exam information

174 students took the exam. Scores ranged from 4 to 23.5, with a median of 18 and an average of 17.1. There were 76 scores between 18.5 and 24, 77 between 12.5 and 18, 19 between 6.5 and 12, and 2 between 0 and 6. (Were you to receive a grade of 18 out of 24 on your first exam, 27 out of 36 on your second, and 45 out of 60 on the final exam, plus good grades on homework and lab, you would receive an A--; similarly, a test grade of 12 may be projected to a B--.)

There were four versions of the exams, A, B, C, and D. (The version indicator appears at the bottom of the first page.) Versions A and C differed only in the sequence of the problems; versions B and D differed in the same way.

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

Problem 0

You lost $\frac{1}{2}$ point on this problem if you did any of the following:

- you earned some credit on a problem and did not put your login name on the page,
- you did not adequately identify your lab section, or
- you failed to put the names of your neighbors on the exam.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we would like to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to verify where students were sitting in the classroom while the exam was being administered.

Problem 1 (problem 2 on versions C and D)

This problem was based on lab activities of January 31 through February 6. In versions A and C, it involves copying an array of Point objects; the version B and D task was to copy an array of Measurement objects. Here’s a solution for the Point problem.

```java
path2 = new Point [path1.length];
for (int k=0; k<path2.length; k++) {
    if (path1[k] == null) {
        path2[k] = null;
    } else {
        path2[k] = new Point (path1[k].x, path1[k].y);
    }
}
```

The 3 points for this problem were split as follows:

- 1 point for calling new for the array variable;
- $1\frac{1}{2}$ points for calling new for all the individual Point or Measurement objects;
- $\frac{1}{2}$ point for everything else.
We did not deduct any points for forgetting the null check, or for using invented accessor methods (e.g. getFeet or getInches). Note that within a main method in the Measurement class, one is allowed access to private instance variables.

Many solutions failed to call new for the individual Point or Measurement objects, instead filling the array with assignment statements. This results in the situation diagrammed below, where a change to the the instance variables in one of the Point or Measurement objects affects the contents of both arrays.

Problem 2 (problem 1 on versions C and D)

This problem was based on lab activities of February 12 and 13. Part a involved determining whether assignment statements between two references in classes related by inheritance were legal; part b was to predict the result of two calls to equals. The sequence of statements differed in versions A and B, and in versions C and D.

Here are solutions to part a of version A.

<table>
<thead>
<tr>
<th>statement</th>
<th>error or OK?</th>
<th>explanation if error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card c1 = new GameCard (1,2);</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>GameCard c2 = new Card (15);</td>
<td>error</td>
<td>Not all Card objects are GameCard objects, so we have a type clash (at compile time).</td>
</tr>
<tr>
<td>GameCard c3 = c1;</td>
<td>error</td>
<td>Though c1 would contain a reference to a GameCard at run time, the compiler has no way of knowing that c1 will contain the correct type of Card. A compile-time error will result unless a cast is used. (Many of you missed this.)</td>
</tr>
</tbody>
</table>
Solutions to part b appear below.

<table>
<thead>
<tr>
<th>expression</th>
<th>true or false?</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1.equals (c2)</td>
<td>false</td>
<td>Card (the class of which c1 is an instance) has no equals method, so Object.equals is used. The latter compares the contents of the references c1 and c2, which don’t match.</td>
</tr>
<tr>
<td>c2.equals (c1)</td>
<td>false</td>
<td>The GameCard class does have an equals method defined, but its argument is a GameCard. The given call to equals has a Card argument. No equals with that signature is provided in the GameCard class, so Object.equals is used here as well. (Many of you missed this.)</td>
</tr>
</tbody>
</table>

Grading for each part was essentially $\frac{1}{2}$ point for the answer (error or OK, true or false) and $\frac{1}{2}$ point for the explanation. In part b, we deduced $\frac{1}{2}$ point for saying “Object.equals is used” without also also saying why.

**Problem 3 (problem 4 on versions C and D)**

This problem was based on lab activities of January 29 and 30. The two versions of code to be analyzed appear below.

**Versions A and C (edited slightly for space)**

```java
public boolean contains1MoreThan(String s) {
    if (s.length()==0) {
        return true;
    } else if (myString.length() != 1+s.length()) {
        return false;
    } else {
        Debug rest = new Debug (myString.substring(1));
        if (myString.charAt(0) == s.charAt(0)) {
            return rest.contains1MoreThan(s.substring(1));
        } else {
            return rest.contains1MoreThan(s);
        }
    }
}
```

**Versions B and D**

```java
public boolean contains1MoreThan(String s) {
    return helper(s, 0, 0) == myString.length()-1;
}
private int helper (String s, int myPos, int sPos) {
    if (sPos >= s.length()) {
        return 0;
    } else if (myString.charAt(myPos) == s.charAt(sPos)) {
        return 1 + helper(s, myPos+1, sPos+1);
    } else {
        return helper(s, myPos+1, sPos);
    }
}
```

The version A/C code may return true prematurely before myString has been checked. It won’t ever crash, however; the checks for an empty s and a myString that’s longer than s together guarantee that there is at least one character in each of s and myString. The version B/D code crashes in helper when myString runs out before the argument. When contains1MoreThan returns a result, it’s the correct one.

Part a was to find *any* myString and s for which contains1MoreThan performed incorrectly. In versions A and C, this was any pair where myString is the result of adding a single character somewhere other than at the end of s (returns false instead of true), or any myString whose length isn’t 1 and an s that’s empty (returns true instead of false). The counterpart in versions B and D was any pair where myString doesn’t contain all the characters of s in the order in which they appear in s.
Part b was to analyze individual cases. Answers appear below.

**Versions A and C**

Pairs of nonnull myString and s for which contains1MoreThan correctly returns true: any pair for which myString is the result of adding a single character to the end of s.

Pairs of nonnull myString and s for which contains1MoreThan correctly returns false: any pair for which s is nonempty and myString is not the result of adding a single character to s.

Pairs of nonnull myString and s for which contains1MoreThan should return true, but instead crashes or returns false: any pair for which myString is the result of adding a single character somewhere other than at the end of s.

Pairs of nonnull myString and s for which contains1MoreThan should return false, but instead crashes or returns true: any myString with length ≠ 1 and an empty s.

**Versions B and D**

Pairs of nonnull myString and s for which contains1MoreThan correctly returns true: any pair for which myString is the result of adding a single character to s.

Pairs of nonnull myString and s for which contains1MoreThan correctly returns false: any pair for which myString is the result of inserting 0 characters or more than 1 character into s.

Pairs of nonnull myString and s for which contains1MoreThan should return false, but instead crashes or returns false: none.

Pairs of nonnull myString and s for which contains1MoreThan should return true, but instead crashes or returns true: any pair for which myString is not the result of inserting 0 or more characters into s, i.e. myString doesn’t contain all the characters in s.

Part a was worth 2 points, 1 for the example and one for an explanation of how it causes contains1MoreThan to misbehave. Part b was worth 5 points, up to 2 for one of your answers and up to 1 each for the others. (This awarded a premium for getting at least one of the choices correct.) Answers that described a significant subset of correct (myString, s) pairs, or that identified all the correct pairs plus a few incorrect pairs, lost \( \frac{1}{2} \) point. Especially common errors in versions A and C were to omit the requirement for a nonempty s in describing pairs where contains1MoreThan correctly returns false, and to say that no pairs cause contains1MoreThan to incorrectly return false.

**Problem 4 (problem 3 on versions C and D)**

This problem was based on lab activities of January 31 and February 1 (testing) and February 7 and 8 (iterators). All versions involved the implementation of an iterator that enumerates the starting positions of groups in the underlying IntSequence. (In versions A and C, a group was a maximal sequence of negative values or a maximal sequence of nonnegative values. Groups in versions B and D involves even numbers and odd numbers.) Here are two solutions to part a.

Solution 1: invariant = “current is the index of the start of the next group to be processed by nextGroupStartPosition”.
public void initGroupStartPositions ( ) {  
    current = 0;
}

public int nextGroupStartPosition ( ) {  
    int rtn = current;
    for (current++;
    current < myCount
    && (myValues[current] >= 0) == (myValues[current-1] >= 0);
    current++) {
        return rtn;
    }

    public boolean moreGroupsRemain ( ) {  
        return current < myCount;
    }

Solution 2: invariant = “current is the index of the start of the group most recently processed by GroupStartPosition”.

public void initGroupStartPositions ( ) {  
    current = -1; // no group position returned yet
}

public int nextGroupStartPosition ( ) {  
    if (current == -1) {
        current++;
        return 0;
    }  
    for (current++;
    (myValues[current] >= 0) == (myValues[current-1] >= 0);
    current++) {
        return current;
    }

    public boolean moreGroupsRemain ( ) {  
        if (current == -1) {
            return myCount > 0;
        }
        int check;
        for (check = current+1;
        check < myCount
        && (myValues[check] >= 0) == (myValues[check-1] >= 0);
        check++) {
            return check < myCount;
        }

Some notes: The iterator version that maintains current as the index next to be returned is simpler and organized more conventionally than the other version. In the latter, search for the next group starting position must be done twice, once in nextGroupStartPosition and once—not using current—in moreGroupsRemain. (The moreGroupsRemain method is not allowed to maintain any state; calling it a million times should have no effect on the iteration through group start positions.)
By far, the most common error on this part was to maintain an “invariant” inconsistently. Some examples: if the nextGroupStartPosition in solution 2 is combined with the moreGroupsRemain in solution 1, moreGroupsRemain returns true even after the last group starting position has been returned; if current is initialized to 0 in solution 2 and no check is made in nextGroupStartPosition for current == 0, the first group start position never gets returned.

Grading of parts a and b (identifying the invariant) proceeded as follows. Part a was worth 5 points and part b 1 point.

In part a, you lost 2 points for inconsistent maintenance of the invariant. You lost 2 points for maintaining state in moreGrpsRemain, so that consecutive calls to moreGrpsRemain would have different effects. (A few students changed current in moreGrpsRemain, then reset it to its original value. Although this is a bad idea in general, it received no deduction.) You also lost 2 points for not having a loop in nextGroupStartPosition. Each occurrence of a minor or easy-to-fix Java or logic error lost $\frac{1}{2}$ point, except that neglecting to remember that $n%2$ is less than 0 when $n$ is negative received no deduction.

To earn anything in part b, you had to relate the value of current to the group start position returned by nextGroupStartPosition. Vagueness about how the two were related generally resulted in a $\frac{1}{2}$ point deduction. Your invariant description also had to agree with your code. However, if your code was inconsistent, maintaining one invariant in, say, nextGroupStartPosition and another in moreGroupsRemain, you were allowed to describe either invariant in part b.

In part c, worth 2 points, you were to identify a set of test cases that, together with the example given earlier in the problem, would provide as much evidence as possible about the correctness of your iterator methods. Here are some acceptable answers.

- an empty IntSequence (boundary case)
- a 1-element IntSequence (boundary case)
- a multi-element IntSequence with only one group (boundary case)
- an IntSequence that begins or ends with a multi-element group, depending on the version (typical case, chosen to minimize duplication with the given example)
- an IntSequence containing only groups, two or more, of size 1 (boundary case for group size, typical case for group count)

You received $\frac{1}{2}$ point for each test case and its justification (no points for a test case without some explanation). Some of you listed a group with negative numbers (in versions A and C) and a group with nonnegative numbers separately. You were only allowed to do this once. For example, if your four test cases were a one-element IntSequence containing only a 1, a one-element IntSequence containing only a –2, a two-element IntSequence containing 1 and 1, and a two-element IntSequence containing –2 and –2, you only received $1\frac{1}{2}$ points.