258 students took the exam. The average was 26.4 out of 36; the median was 27.5; scores ranged from 3 to 35.5. 133 students scored between 27.5 and 36, 99 between 18.5 and 27, 24 between 9.5 and 18, and 2 scored 8.5 or less. (Were you to receive grades of 75% on each in-class exam—*i.e.* 36 on this exam—and on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 18 may be projected to a B–.

There were two versions of the test. (The version indicator appears at the bottom of the first page.) They differed in superficial ways.

Incorrect answers were coded with letters (e.g. A, B, C). The codes are explained in each problem below.

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 your entire exam.

Problem 0 (2 points)

Each of the following cost you a point on this problem: you earned some credit on a problem and did not put your five-digit 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 want 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 (5 points)

This problem had three parts.

Part a asked why there was only one check for null in a Point.equals method. The answer is that this is the missing reference, and calling a nonstatic method with this == null crashes the program.

This was worth 1 point. You had to mention this to get any credit. Some vague answers received a ½ point deduction. A common mistake was to refer to this by some other name.

Part b involved converting equals to a static method. Here's a solution:

```
public static boolean equals (Point p1, Point p2) {
   if (p1 == null) {
      return p2 == null;
   } else if (p2 == null) {
      return false;
   } else {
      return p1.myX == p2.myX && p1.myY == p2.myY;
   }
}
```

Note that if both p1 and p2 are null, equals returns true.

The 3 points for this problem were split 1 for each of the two null checks and 1 for the rest. A special-case deduction of 1 was given for the code

```
if (p1 == null || p2 == null) {
   return false;
} ...
```

Codes for errors are as follows.

code	*deduction*	*reason*
A	–2	there's no check for null at all
B	–1	the code doesn't check null correctly
C	–1	equality testing is messed up
D	–3	the static equals method doesn't take two arguments
E	–1.5	only one null check is made
F	–0.5	the keyword static isn't used
G	0	parameters are both of type Object
I	–3	only == is used

You were allowed to have either Object or Point references as the arguments.

Finally, part c was to design an experiment to see if Object.equals has been overridden. This can be done by creating two objects with the same contents but different-valued references, and then comparing them.

```
Point p1 = new Point (3, 6);
Point p2 = new Point (3, 6);
if (p1.equals (p2)) {
   System.out.println ("equals has been overridden");
} else {
   System.out.println ("equals has not been overridden");
}
```

This was worth 1 point. You needed two different Point references pointing to Point objects with the same contents for any credit. A special case was something like the following:

```
Point a = new Point (1, 2);
Point b = new Point (1, 2);
Point c = (Point) obj;
if (a.equals (c)) ...
```

which lost ½ point.

Code "H" referred to miscellaneous errors such as including only one print statement or using try … catch.

Problem 2 (5 points)

In part a, you were to implement a subclass of Point named URQuadPoint (LLQuadPoint in version B) that only allowed points whose coordinates were nonnegative (negative in version B). A new constructor is all that's needed. Here's a solution:

```
public class URQuadPoint extends Point {

   public URQuadPoint (int x, int y) {
      super (x, y);
      if (x < 0 || y < 0) {
         throw new IllegalArgumentException ( );
      }
   }
}
```

The call to super has to be the first line in the method. A common error was to do the test for negative values first, and then call super after the test is passed. You didn't need to say "throws IllegalArgumentException" in the URQuadHeader since it inherits from RuntimeException. You were allowed to have either Object or Point references as the arguments.

Codes are as follows. Maximum score is 3.

code	*deduction*	*reason*
A	–1.5	no call to super
B	–0.5	bad or missing comparison, e.g. "or" or "&&" instead of "ll"
C	–1	doesn't throw an exception
D	–0.5	throws exception but doesn't call new
E	–0.5	"super (x, y)" isn't the first line (see above)
F	–0.5	"throws IllegalArgumentException" appears in the wrong place in the header (i.e. before the formal parameters)
G	–3	defining new instance variables rather than using those of Point
H	–0.5	omits parentheses in "new IllegalArgumentException ( )"
I	–0.5	uses "super" incorrectly, e.g. "superclass" or "super.Point"
J	–1	doesn't define a constructor
K	–1	calls super but access private instance variables
L	–0.5	constructor takes wrong arguments
M	–0.5	wrong name for constructor
N	–0.5	wrong argument when calling new IllegalArgumentException
P	–2	provides extra methods

Part b, worth 2 points, asks for the effect of including the two given assignment statements:

```
Point p1 = new URQuadPoint (-7, 3);
URQuadPoint p3 = new Point (7, 3);
```

The assignment to p1 compiles without error, and obeys requirements involving inheritance. However, calling the URQuadPoint constructor throws IllegalArgumentException because of the negative parameter value. The assignment to p3 without casting is flagged by the compiler, since there is no way to guarantee that p3 is a URQuadPoint.

Each answer is worth 1 point. No partial credit was given.

Problem 3 (5 points)

This problem asked you to supply the code that produces given output. The two versions differed in the output produced; we will work with version A.

version A	version B
First step (5,6) (5,6) (14,15) Second step (7,8) (7,8) (15,16)	First step (7,8) (1,2) (7,8) Second step (9,10) (2,3) (9,10)

The first requirement is to initialize the array:

```
Point [ ] myPoints = new Point [3];
```

Next, we try to figure out how the output is produced. The program segments after the initialization of the myPoints entries essentially set each (x,y) to be (x+1,y+1). That suggests initial values for the three points:

```
myPoints[0] = new Point (4, 5);    // incrementing gives (5, 6).
myPoints[1] = new Point (4, 5);    // ditto
myPoints[2] = new Point (13, 14); // incrementing gives (14, 15)
```

But now, examine the result of the second increment. myPoints[2] has the answer we expect, but neither of the other elements do. Each of myPoints[0] and myPoints[1] appears to have been incremented *twice*.

The "aha" is to realize that if *two* of the myPoint elements refer to *the same* Point, that Point will get incremented once for *each* reference. The correct sequence of assignments is thus

```
myPoints[0] = new Point (3, 4);
myPoints[1] = myPoints[0];
myPoints[2] = new Point (13, 14);
```

The problem was worth 5 points. Error codes are as follows.

code	*deduction*	*reason*
A	–1	Numeric arguments are incorrect (if the index values are also incorrect, an additional 0.5 was deducted)
B	–1	Incorrect or no initialization of myPoints

C	–3	`myPoints[0] = new Point (4, 5);`
		`myPoints[1] = new Point (4, 5);`
		`myPoints[2] = new Point (13, 14);`

D	–1	Switched results, e.g.
		`myPoints[0] = myPoints[1];`
		`myPoints[1] = new Point (3, 4);`
		`myPoints[2] = new Point (13, 14)`

E	–0.5	Not using new, or other syntactic errors

Problem 4 (7 points)

This was a two-part problem. Part a, worth 3 points, was to implement a removeAdjacentDuplicates method for the IntSequence class. Here are some solutions:

```
public void removeAdjacentDuplicates ( ) {
   int k = 0;
   while (k < myCount) {
      if (k < myCount-1 && myValues[k] == myValues[k+1]) {
         remove (k+1);  // it updates myCount
      } else {
         k = k+1;
      }
   }
   return this;
}
```

```
public void removeAdjacentDuplicates ( ) {
   int k = 0;
   while (k+1 < myCount) {
      if (myValues[k] == myValues[k+1]) {
         remove (k+1);  // it updates myCount
      } else {
         k = k+1;
      }
   }
   return this;
}
```

```
public void removeAdjacentDuplicates ( ) {
   for (int i = 0; i+1 < myCount; i++){
      if (myValues[i] == myValues[i+1]) {
         remove (i+1);  // it updates myCount
         i--;
      }
   }
   return this;
}
```

Variations on the above code included moving backward through the sequence.

Here are error codes.

code	*deduction*	*reason*
A	–1	Off-by-one error if myCount == myValues.length (a common error)
B	–1	Index incremented every iteration even if removing an element (a common error)
C	–1	Omitting a return
D	–1	Throwing an exception if removeAdjacentDuplicates is called with an empty IntSequence
E	–1	Modifying the value of myCount even though remove also changes it (a common error)
F	–1	Attempting to modify size, e.g. size ( )++
G	–1	Calling elementAt(0) on a potentially empty IntSequence (a common error)
H	–1	Uses myValues.length as the bound rather than size ( ) or myCount
I	–1	Passing a value rather than an index to remove
J	–1	Removing duplicates rather than adjacent duplicates, or using contains in other ways

You were allowed either to remove adjacent duplicates in this or to do so in a copy. (Your choice may have affected your score for problem 5.)

Part b was worth 4 points. It involved devising test cases for removeAdjacentDuplicates that the given example did not test. If you mentioned six or seven appropriate tests, you earned all 4 points; four or five earned 3 points; three earned 2 points; and one or two earned 1 point.

The given example on version A took {2, 3, 55, 3, 3, 16, 16, 16} as argument and returned {2, 3, 55, 3, 16}. It has duplicates at the end and in the middle. Other candidate tests included the following:

1. ending with a nonduplicate
2. starting with a duplicate
3. duplicates only in the middle
4. a sequence of one element
5. a longer sequence with all duplicates
6. a longer sequence with no duplicates
7. an empty sequence (note that a null sequence isn't possible)
8. a single element between sequences of duplicates
9. duplicates only in the middle

The example on version B took {16, 16, 16, 3, 55, 3, 2} as argument and returned {16, 3, 55, 3, 2}. Good test categories here include starting with a nonduplicate and ending with

a duplicate, multiple duplicates, and duplicates only at the end, along with tests 3-9 above.

Problem 5 (6 points)

For this problem, you were to implement an iterator for the unduplicated elements of an IntSequence, where "duplicate" was the same as in problem 4. Your implementation was also to take constant time to initialize. The best solutions used an index variable—we'll call it iterIndex—and maintained the invariant that iterIndex is the position in the sequence of the next value to return, or is equal to myCount if there are no more values left to return. Here's the code.

```
private int iterIndex;

public void initIterator ( ) {
   iterIndex = 0;
}

public boolean hasNext ( ) {
   return iterIndex < myCount;
}

public int next ( ) {
   int returnVal = myValues[iterIndex];
   while (true) {
      iterIndex++;
      if (iterIndex == myCount) {
         return returnVal;
      }
      if (myValues[iterIndex] != returnVal) {
         return returnVal;
      }
   }
}
```

You could earn a maximum of 6 points. There were many possible errors.

code	*deduction*	*reason*
A	–4	No loop or recursive call
B	–2	Change of state in hasNext
C	–3	Modifying of the underlying sequence, i.e. by changing myValues or myCount (but see below)
D	–1	Crash at the start or end of the sequence
E	–1	Skipping an element
F	–1	Misordered statements or method calls (e.g. by accessing myValues[k] before checking if k < myCount
G	–1	Off-by-one error
H	–1	Bad loop structure in next or hasNext
I	–1	Unreachable code after a return

J	–1	Missing return statement (missing only the word "return" lost ½ point)
K	–1	Irrelevant invariant
L	–0.5	Initializing the index variable at declaration rather than in the constructor (thereby causing a rerun of the iteration not to work)
M	–2	No loop in next where one is necessary
N	–0.5	Returning an index rather than a sequence element from next
P	–0.5	Calling a method with seq.method instead of this.method
Q	–1	Returns the item immediately following the appropriate item
R	–0.5	Violating the requirement for constant-time initialization
S	–0.5	No parentheses after a method call
T	–2	No instance variables
U	–0.5	Inconsistent variable names
V	–1	Calls remove that would modify this (but see below)
AA	–1	Using an uninitialized variable
BB	–0.5	Sets an int variable to null
CC	–0.5	Uses myValues.length rather than size ( ) or myCount
DD	–1	Type confusion, e.g. [ ] for IntSequence
EE	–1	Omitted update to a variable
FF	–2	Using initIterator to print values
GG	–0.5	Incorrect operator, e.g. "&&" or "or" for "ll"
HH	–0.5	Invariant doesn't match code
JJ	–0.5	Undeclared variable

You received no penalty for modifying this if you (a) called removeAdjacentDuplicates *and* (b) returned from removeAdjacentDuplicates a copy of the IntSequence.

Solutions with errors accumulating a deduction of 4 or more points were graded on a positive scale:

- ½ point for an initialized variable and an invariant relating it to the next value to return or the previously returned value (code W);
- ½ point for a loop or recursion in next or hasNext involving the index variable (code X);
- ½ point for a comparison in next of one sequence value with another sequence value (code Y);
- ½ point for a loop or recursion in next (code Z).

Problem 6 (6 points)

You were to complete a reimplementation of the adding machine program from an earlier homework assignment, using only statements from the list below.

`subtotal = 0;`	`justStarting = false;`
`subtotal = subtotal + k;`	`justStarting = true;`
`total = total + k;`	`k = inputValues.nextInt ( );`
`while (k != 0) {`	`while (k == 0) {`
`System.out.println (“subtotal “ + subtotal);`	`System.out.println (“total “ + total);`

Each iteration of the outer loop processes a subtotalable group. We know at the start of the code we're to supply that a number has been read. Either this value is not 0 and it starts a subtotalable group, or it is 0 but we're "just starting". In the latter case, we want to report a subtotal without any more input. The code we add reads all the values in the subtotal group, then reports the subtotal.

Here is a solution.

```
while (k != 0) {
   subtotal = subtotal + k;
   total = total + k;
   k = inputValues.nextInt ( );
}
justStarting = false;    // this can go inside or above the loop
System.out.println ("subtotal " + subtotal);
subtotal = 0;
```

This problem was worth 6 points. Codes and deductions appear below.

code	*deduction*	*reason*
A	–2	An extra loop "while (k == 0) " whose body includes subtotal printing and reset
B	–0.5	Omitted or misplaced line, or use of a statement not in the above table (assuming it did not change the difficulty of the problem)
C	–1	Incorrect assignment to justStarting
D	–1	Infinite loop