

The Exam

AP[®] Computer Science A Exam

SECTION I: Multiple-Choice Questions

2009

DO NOT OPEN THIS BOOKLET UNTIL YOU ARE TOLD TO DO SO.**At a Glance****Total Time**

1 hour, 15 minutes

Number of Questions

40

Percent of Total Grade

50%

Writing Instrument

Pencil required

Electronic Device

None allowed

Instructions

The orange Appendix booklet is provided for use in both Section I and Section II.

Section I of this exam contains 40 multiple-choice questions. Fill in only the ovals for numbers 1 through 40 on your answer sheet.

Indicate all of your answers to the multiple-choice questions on the answer sheet. No credit will be given for anything written in this exam booklet, but you may use the booklet for notes or scratch work. After you have decided which of the suggested answers is best, completely fill in the corresponding oval on the answer sheet. Give only one answer to each question. If you change an answer, be sure that the previous mark is erased completely. Here is a sample question and answer.

Sample Question Sample Answer

Chicago is a

(A) ☒ (B) ☐ (C) ☐ (D) ☐ (E) ☐

(A) state

(B) city

(C) country

(D) continent

(E) village

Use your time effectively, working as quickly as you can without losing accuracy. Do not spend too much time on any one question. Go on to other questions and come back to the ones you have not answered if you have time. It is not expected that everyone will know the answers to all of the multiple-choice questions.

About Guessing

Many students wonder whether or not to guess the answers to questions about which they are not certain. In this section of the exam, as a correction for random guessing, one-fourth of the number of questions you answer incorrectly will be subtracted from the number of questions you answer correctly. If you are not sure of the best answer but have some knowledge of the question and are able to eliminate one or more of the answer choices, your chance of answering correctly is improved, and it may be to your advantage to answer such a question.

Section I

COMPUTER SCIENCE A SECTION I

Time—1 hour and 15 minutes

Number of questions—40

Percent of total grade—50

Directions: Determine the answer to each of the following questions or incomplete statements, using the available space for any necessary scratch work. Then decide which is the best of the choices given and fill in the corresponding oval on the answer sheet. No credit will be given for anything written in the examination booklet. Do not spend too much time on any one problem.

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- Assume that declarations of variables and methods appear within the context of an enclosing class.
- Assume that method calls that are not prefixed with an object or class name and are not shown within a complete class definition appear within the context of an enclosing class.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null`.

1. Consider the following code segment.

```
int value = 15;
while (value < 28)
{
    System.out.println(value);
    value++;
}
```

What are the first and last numbers output by the code segment?

	<u>First</u>	<u>Last</u>
(A)	15	27
(B)	15	28
(C)	16	27
(D)	16	28
(E)	16	29

Section I

2. A teacher put three bonus questions on a test and awarded 5 extra points to anyone who answered all three bonus questions correctly and no extra points otherwise. Assume that the `boolean` variables `bonusOne`, `bonusTwo`, and `bonusThree` indicate whether a student has answered the particular question correctly. Each variable was assigned `true` if the answer was correct and `false` if the answer was incorrect.

Which of the following code segments will properly update the variable `grade` based on a student's performance on the bonus questions?

- I.

```
if (bonusOne && bonusTwo && bonusThree)
    grade += 5;
```
- II.

```
if (bonusOne || bonusTwo || bonusThree)
    grade += 5;
```
- III.

```
if (bonusOne)
    grade += 5;
if (bonusTwo)
    grade += 5;
if (bonusThree)
    grade += 5;
```

- (A) I only
- (B) II only
- (C) III only
- (D) I and III
- (E) II and III

Assume that an array of integer values has been declared as follows and has been initialized.

```
int[] arr = new int[10];
```

Which of the following code segments correctly interchanges the value of `arr[0]` and `arr[5]` ?

- (A) `arr[0] = 5;`
`arr[5] = 0;`
- (B) `arr[0] = arr[5];`
`arr[5] = arr[0];`
- (C) `int k = arr[5];`
`arr[0] = arr[5];`
`arr[5] = k;`
- (D) `int k = arr[0];`
`arr[0] = arr[5];`
`arr[5] = k;`
- (E) `int k = arr[5];`
`arr[5] = arr[0];`
`arr[0] = arr[5];`

Section I

4. Consider the following code segment.

```
ArrayList<String> items = new ArrayList<String>();  
items.add("A");  
items.add("B");  
items.add("C");  
items.add(0, "D");  
items.remove(3);  
items.add(0, "E");  
System.out.println(items);
```

What is printed as a result of executing the code segment?

- (A) [A, B, C, E]
 - (B) [A, B, D, E]
 - (C) [E, D, A, B]
 - (D) [E, D, A, C]
 - (E) [E, D, C, B]
-

5. When designing a class hierarchy, which of the following should be true of a superclass?

- (A) A superclass should contain the data and functionality that are common to all subclasses that inherit from the superclass.
- (B) A superclass should be the largest, most complex class from which all other subclasses are derived.
- (C) A superclass should contain the data and functionality that are only required for the most complex class.
- (D) A superclass should have public data in order to provide access for the entire class hierarchy.
- (E) A superclass should contain the most specific details of the class hierarchy.

Questions 6-7 refer to the following code segment.

```
int k = a random number such that  $1 \leq k \leq n$  ;  
  
for (int p = 2; p <= k; p++)  
    for (int r = 1; r < k; r++)  
        System.out.println("Hello");
```

6. What is the minimum number of times that Hello will be printed?

- (A) 0
 - (B) 1
 - (C) 2
 - (D) $n - 1$
 - (E) $n - 2$
-

7. What is the maximum number of times that Hello will be printed?

- (A) 2
- (B) $n - 1$
- (C) $n - 2$
- (D) $(n - 1)^2$
- (E) n^2

Section I

8. Consider the following instance variable and incomplete method. The method `calcTotal` is intended to return the sum of all values in `vals`.

```
private int[] vals;  
  
public int calcTotal()  
{  
    int total = 0;  
  
    /* missing code */  
  
    return total;  
}
```

Which of the code segments shown below can be used to replace `/* missing code */` so that `calcTotal` will work as intended?

- I.

```
for (int pos = 0; pos < vals.length; pos++)  
{  
    total += vals[pos];  
}
```
- II.

```
for (int pos = vals.length; pos > 0; pos--)  
{  
    total += vals[pos];  
}
```
- III.

```
int pos = 0;  
while (pos < vals.length)  
{  
    total += vals[pos];  
    pos++;  
}
```

- (A) I only
- (B) II only
- (C) III only
- (D) I and III
- (E) II and III

9. Consider the following code segment.

```
String str = "abcdef";  
for (int rep = 0; rep < str.length() - 1; rep++)  
{  
    System.out.print(str.substring(rep, rep + 2));  
}
```

What is printed as a result of executing this code segment?

- (A) abcdef
- (B) aabbccddeeff
- (C) abbccddeef
- (D) abcbcdcdedef
- (E) Nothing is printed because an `IndexOutOfBoundsException` is thrown.

Section I

10. Consider the following method.

```
public void numberCheck(int maxNum)
{
    int typeA = 0;
    int typeB = 0;
    int typeC = 0;

    for (int k = 1; k <= maxNum; k++)
    {
        if (k % 2 == 0 && k % 5 == 0)
            typeA++;
        if (k % 2 == 0)
            typeB++;
        if (k % 5 == 0)
            typeC++;
    }

    System.out.println(typeA + " " + typeB + " " + typeC);
}
```

What is printed as a result of the call `numberCheck(50)` ?

- (A) 5 20 5
- (B) 5 20 10
- (C) 5 25 5
- (D) 5 25 10
- (E) 30 25 10

1. Consider the following method that is intended to modify its parameter `nameList` by replacing all occurrences of `name` with `newValue`.

```
public void replace(ArrayList<String> nameList,  
                    String name, String newValue)  
{  
    for (int j = 0; j < nameList.size(); j++)  
    {  
        if ( /* expression */ )  
        {  
            nameList.set(j, newValue);  
        }  
    }  
}
```

Which of the following can be used to replace `/* expression */` so that `replace` will work as intended?

- (A) `nameList.get(j).equals(name)`
- (B) `nameList.get(j) == name`
- (C) `nameList.remove(j)`
- (D) `nameList[j] == name`
- (E) `nameList[j].equals(name)`

Section I

12. Consider the following incomplete method.

```
public int someProcess(int n)
{
    /* body of someProcess */
}
```

The following table shows several examples of input values and the results that should be produced by calling `someProcess`.

Input Value n	Value Returned by <code>someProcess(n)</code>
3	30
6	60
7	7
8	80
9	90
11	11
12	120
14	14
16	160

Which of the following code segments could be used to replace `/* body of someProcess */` so that the method will produce the results shown in the table?

I.

```
if ((n % 3 == 0) && (n % 4 == 0))
    return n * 10;
else
    return n;
```

II.

```
if ((n % 3 == 0) || (n % 4 == 0))
    return n * 10;

return n;
```

```
II.  if (n % 3 == 0)
      if (n % 4 == 0)
          return n * 10;

      return n;
```

- A) I only
- B) II only
- C) III only
- D) I and III
- E) II and III

Section I

13. Consider the following method.

```
// precondition: x >= 0
public void mystery(int x)
{
    if ((x / 10) != 0)
    {
        mystery(x / 10);
    }

    System.out.print(x % 10);
}
```

Which of the following is printed as a result of the call `mystery(123456)` ?

- (A) 16
- (B) 56
- (C) 123456
- (D) 654321
- (E) Many digits are printed due to infinite recursion.

Consider the following instance variables and incomplete method that are part of a class that represents an item. The variables `years` and `months` are used to represent the age of the item, and the value for `months` is always between 0 and 11, inclusive. Method `updateAge` is used to update these variables based on the parameter `extraMonths` that represents the number of months to be added to the age.

```
private int years;
private int months; // 0 <= months <= 11

// precondition: extraMonths >= 0
public void updateAge(int extraMonths)
{
    /* body of updateAge */
}
```

Which of the following code segments could be used to replace `/* body of updateAge */` so that the method will work as intended?

- I. `int yrs = extraMonths % 12;`
`int mos = extraMonths / 12;`
`years = years + yrs;`
`months = months + mos;`
- II. `int totalMonths = years * 12 + months + extraMonths;`
`years = totalMonths / 12;`
`months = totalMonths % 12;`
- III. `int totalMonths = months + extraMonths;`
`years = years + totalMonths / 12;`
`months = totalMonths % 12;`

- (A) I only
- (B) II only
- (C) III only
- (D) II and III only
- (E) I, II, and III

Section I

15. Consider the following method.

```
public String inRangeMessage(int value)
{
    if (value < 0 || value > 100)
        return "Not in range";
    else
        return "In range";
}
```

Consider the following code segments that could be used to replace the body of `inRangeMessage`.

- I. if (value < 0)
 {
 if (value > 100)
 return "Not in range";
 else
 return "In range";
 }
 else
 return "In range";
- II. if (value < 0)
 return "Not in range";
 else if (value > 100)
 return "Not in range";
 else
 return "In range";
- III. if (value >= 0)
 return "In range";
 else if (value <= 100)
 return "In range";
 else
 return "Not in range";

Which of the replacements will have the same behavior as the original version of `inRangeMessage` ?

- (A) I only
(B) II only
(C) III only
(D) I and III
(E) II and III

Consider the following class declaration.

```
public class SomeClass
{
    private int num;

    public SomeClass(int n)
    {
        num = n;
    }

    public void increment(int more)
    {
        num = num + more;
    }

    public int getNum()
    {
        return num;
    }
}
```

The following code segment appears in another class.

```
SomeClass one = new SomeClass(100);
SomeClass two = new SomeClass(100);
SomeClass three = one;

one.increment(200);

System.out.println(one.getNum() + " " + two.getNum() + " " +
                    three.getNum());
```

What is printed as a result of executing the code segment?

- (A) 100 100 100
- (B) 300 100 100
- (C) 300 100 300
- (D) 300 300 100
- (E) 300 300 300

Section I

17. The following incomplete method is intended to sort its array parameter `arr` in increasing order.

```
// postcondition: arr is sorted in increasing order
public static void sortArray(int[] arr)
{
    int j, k;

    for (j = arr.length - 1; j > 0; j--)
    {
        int pos = j;

        for ( /* missing code */ )
        {
            if (arr[k] > arr[pos])
            {
                pos = k;
            }
        }
        swap(arr, j, pos);
    }
}
```

Assume that `swap(arr, j, pos)` exchanges the values of `arr[j]` and `arr[pos]`. Which of the following could be used to replace `/* missing code */` so that executing the code segment sorts the values in array `arr`?

- (A) `k = j - 1; k > 0; k--`
- (B) `k = j - 1; k >= 0; k--`
- (C) `k = 1; k < arr.length; k++`
- (D) `k = 1; k > arr.length; k++`
- (E) `k = 0; k <= arr.length; k++`

- i. Assume that x and y are boolean variables and have been properly initialized.

$(x \ \&\& \ y) \ || \ !(x \ \&\& \ y)$

The result of evaluating the expression above is best described as

- (A) always true
- (B) always false
- (C) true only when x is true and y is true
- (D) true only when x and y have the same value
- (E) true only when x and y have different values

Section I

19. Assume that the following variable declarations have been made.

```
double d = Math.random();  
double r;
```

Which of the following assigns a value to r from the uniform distribution over the range $0.5 \leq r < 5.5$?

- (A) $r = d + 0.5;$
- (B) $r = d + 0.5 * 5.0;$
- (C) $r = d * 5.0;$
- (D) $r = d * 5.0 + 0.5;$
- (E) $r = d * 5.5;$

. Consider the following instance variables and method that appear in a class representing student information.

```
private int assignmentsCompleted;
private double testAverage;

public boolean isPassing()
{ /* implementation not shown */ }
```

A student can pass a programming course if at least one of the following conditions is met.

- The student has a test average that is greater than or equal to 90.
- The student has a test average that is greater than or equal to 75 and has at least 4 completed assignments.

Consider the following proposed implementations of the `isPassing` method.

- I.

```
if (testAverage >= 90)
    return true;
if (testAverage >= 75 && assignmentsCompleted >= 4)
    return true;
return false;
```
- II.

```
boolean pass = false;
if (testAverage >= 90)
    pass = true;
if (testAverage >= 75 && assignmentsCompleted >= 4)
    pass = true;
return pass;
```
- III.

```
return (testAverage >= 90) ||
        (testAverage >= 75 && assignmentsCompleted >= 4);
```

Which of the implementations will correctly implement method `isPassing`?

- (A) I only
- (B) II only
- (C) I and III only
- (D) II and III only
- (E) I, II, and III

Section I

Questions 21-25 refer to the code from the GridWorld case study. A copy of the code is provided in the Appendix.

21. Consider the following code segment.

```
Location loc1 = new Location(3, 3);
Location loc2 = new Location(3, 2);

if (loc1.equals(loc2.getAdjacentLocation(Location.EAST)))
    System.out.print("aaa");

if (loc1.getRow() == loc2.getRow())
    System.out.print("XXX");

if (loc1.getDirectionToward(loc2) == Location.EAST)
    System.out.print("555");
```

What will be printed as a result of executing the code segment?

- (A) aaaXXX555
- (B) aaaXXX
- (C) XXX555
- (D) 555
- (E) aaa

1. A `RightTurningBug` behaves like a `Bug`, except that when it turns, it turns 90 degrees to the right. The declaration for the `RightTurningBug` class is as follows.

```
public class RightTurningBug extends Bug
{
    public void turn()
    {
        /* missing implementation */
    }
}
```

Consider the following suggested replacements for `/* missing implementation */`.

- I. `int desiredDirection = (getDirection() + Location.RIGHT)`
`% Location.FULL_CIRCLE;`

`while (getDirection() != desiredDirection)`
`{`
 `super.turn();`
`}`
- II. `super.turn();`
`super.turn();`
- III. `setDirection(getDirection() + Location.RIGHT);`

Which of the replacements will produce the desired behavior?

- (A) I only
- (B) II only
- (C) I and II only
- (D) I and III only
- (E) I, II, and III

Section I

23. Consider the following declarations.

```
Actor a = new Actor();  
Bug b = new Bug();  
Rock r = new Rock();  
Critter c = new Critter();
```

Consider the following lines of code.

```
Line 1:  int dir1 = c.getDirection();  
Line 2:  int dir2 = a.getDirection();  
Line 3:  int dir3 = b.getDirection();  
Line 4:  ArrayList<Location> rLoc = r.getMoveLocations();  
Line 5:  ArrayList<Location> cLoc = c.getMoveLocations();
```

Which of the lines of code above will cause a compile time error?

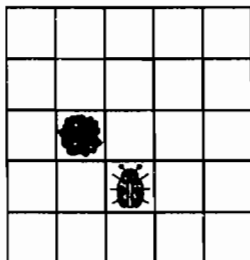
- (A) Line 1 only
- (B) Lines 2 and 3 only
- (C) Line 4 only
- (D) Line 5 only
- (E) Lines 4 and 5 only

. Consider the following `TestBug` class declaration.

```
public class TestBug extends Bug
{
    public void act()
    {
        if (canMove())
        {
            move();
            if (canMove())
                move();
        }
        else
        {
            setDirection(getDirection() + Location.HALF_CIRCLE);
        }
    }
}
```

The following code segment will produce a grid that has a `Rock` object and a `TestBug` object placed as shown.

```
Grid<Actor> g = new BoundedGrid<Actor>(5, 5);
Rock r = new Rock();
r.putSelfInGrid(g, new Location(2, 1));
Bug t = new TestBug();
t.putSelfInGrid(g, new Location(3, 2));
```



Which of the following best describes what the `TestBug` object `t` does as a result of calling `t.act()` ?

- (A) Moves forward two locations and remains facing current direction
- (B) Moves forward two locations and turns 180 degrees
- (C) Moves forward one location and remains facing current direction
- (D) Moves forward one location and turns 180 degrees
- (E) Stays in the same location and turns 180 degrees

Section I

25. A DancingCriticter is a Critter that moves in the following manner. The DancingCriticter makes a left turn if at least one of its neighbors is another DancingCriticter. It then moves like a Critter. If none of its neighbors are DancingCriticter objects, it moves like a Critter without making a left turn. In all other respects, a DancingCriticter acts like a Critter by eating neighbors that are not rocks or critters. Consider the following implementations.

```
I. public class DancingCriticter extends Critter
{
    public ArrayList<Actor> getActors()
    {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        for (Actor a : getGrid().getNeighbors(getLocation()))
        {
            if (a instanceof DancingCriticter)
                actors.add(a);
        }
        return actors;
    }

    public void processActors(ArrayList<Actor> actors)
    {
        if (actors.size() > 0)
        {
            setDirection(getDirection() + Location.LEFT);
        }
        super.processActors(actors);
    }
}
```

```
II. public class DancingCriticter extends Critter
{
    public void processActors(ArrayList<Actor> actors)
    {
        boolean turning = false;
        for (Actor a : actors)
        {
            if (a instanceof DancingCriticter)
                turning = true;
        }
        if (turning)
        {
            setDirection(getDirection() + Location.LEFT);
        }
    }
}
```

```

I. public class DancingCritic extends Critter
{
    public void makeMove(Location loc)
    {
        boolean turning = false;
        for (Actor a : getGrid().getNeighbors(getLocation()))
        {
            if (a instanceof DancingCritic)
                turning = true;
        }
        if (turning)
        {
            setDirection(getDirection() + Location.LEFT);
        }
        super.makeMove(loc);
    }
}

```

Which of the proposed implementations will correctly implement the DancingCritic class?

- A) I only
- B) II only
- C) III only
- D) I and II only
- E) I, II, and III

Section I

26. Consider the following code segment.

```
int k = 0;
while (k < 10)
{
    System.out.print((k % 3) + " ");
    if ((k % 3) == 0)
        k = k + 2;
    else
        k++;
}
```

What is printed as a result of executing the code segment?

- (A) 0 2 1 0 2
- (B) 0 2 0 2 0 2
- (C) 0 2 1 0 2 1 0
- (D) 0 2 0 2 0 2 0
- (E) 0 1 2 1 2 1 2

Consider the following method. Method `allEven` is intended to return `true` if all elements in array `arr` are even numbers; otherwise, it should return `false`.

```
public boolean allEven(int[] arr)
{
    boolean isEven = /* expression */ ;

    for (int k = 0; k < arr.length; k++)
    {
        /* loop body */
    }

    return isEven;
}
```

Which of the following replacements for `/* expression */` and `/* loop body */` should be used so that method `allEven` will work as intended?

- | | <u>/* expression */</u> | <u>/* loop body */</u> |
|-----|-------------------------|---|
| (A) | false | if ((arr[k] % 2) == 0)
isEven = true; |
| (B) | false | if ((arr[k] % 2) != 0)
isEven = false;
else
isEven = true; |
| (C) | true | if ((arr[k] % 2) != 0)
isEven = false; |
| (D) | true | if ((arr[k] % 2) != 0)
isEven = false;
else
isEven = true; |
| (E) | true | if ((arr[k] % 2) == 0)
isEven = false;
else
isEven = true; |

Section I

28. Consider the following code segment.

```
int x = /* some integer value */ ;
int y = /* some integer value */ ;

boolean result = (x < y);

result = ( (x >= y) && !result );
```

Which of the following best describes the conditions under which the value of result will be true after the code segment is executed?

- (A) Only when $x < y$
 - (B) Only when $x \geq y$
 - (C) Only when x and y are equal
 - (D) The value will always be true.
 - (E) The value will never be true.
-

29. Consider the following code segment.

```
for (int outer = 0; outer < n; outer++)
{
    for (int inner = 0; inner <= outer; inner++)
    {
        System.out.print(outer + " ");
    }
}
```

If n has been declared as an integer with the value 4, what is printed as a result of executing the code segment?

- (A) 0 1 2 3
- (B) 0 0 1 0 1 2
- (C) 0 1 2 2 3 3 3
- (D) 0 1 1 2 2 2 3 3 3 3
- (E) 0 0 1 0 1 2 0 1 2 3

30. Consider the following class declarations.

```
public class Base
{
    private int myVal;

    public Base()
    {    myVal = 0;    }

    public Base(int x)
    {    myVal = x;    }
}

public class Sub extends Base
{
    public Sub()
    {    super(0);    }
}
```

Which of the following statements will NOT compile?

- (A) Base b1 = new Base();
 - (B) Base b2 = new Base(5);
 - (C) Base s1 = new Sub();
 - (D) Sub s2 = new Sub();
 - (E) Sub s3 = new Sub(5);
-

31. Assume that a and b are variables of type int. The expression

`!(a < b) && !(a > b)`

is equivalent to which of the following?

- (A) true
- (B) false
- (C) a == b
- (D) a != b
- (E) !(a < b) && (a > b)

Section I

32. Consider the following code segment.

```
int a = 24;
int b = 30;
while (b != 0)
{
    int r = a % b;
    a = b;
    b = r;
}

System.out.println(a);
```

What is printed as a result of executing the code segment?

- (A) 0
- (B) 6
- (C) 12
- (D) 24
- (E) 30

3. Consider the following method.

```
public int sol(int lim)
{
    int s = 0;

    for (int outer = 1; outer <= lim; outer++)
    {
        for (int inner = outer; inner <= lim; inner++)
        {
            s++;
        }
    }

    return s;
}
```

What value is returned as a result of the call `sol(10)` ?

- (A) 20
- (B) 45
- (C) 55
- (D) 100
- (E) 385

Section I

34. Consider the following incomplete method. Method `findNext` is intended to return the index of the first occurrence of the value `val` beyond the position `start` in array `arr`.

```
// returns index of first occurrence of val in arr
// after position start;
// returns arr.length if val is not found
public int findNext(int[] arr, int val, int start)
{
    int pos = start + 1;

    while ( /* condition */ )
        pos++;

    return pos;
}
```

For example, consider the following code segment.

```
int[] arr = {11, 22, 100, 33, 100, 11, 44, 100};

System.out.println(findNext(arr, 100, 2));
```

The execution of the code segment should result in the value 4 being printed.

Which of the following expressions could be used to replace `/* condition */` so that `findNext` will work as intended?

- (A) `(pos < arr.length) && (arr[pos] != val)`
- (B) `(arr[pos] != val) && (pos < arr.length)`
- (C) `(pos < arr.length) || (arr[pos] != val)`
- (D) `(arr[pos] == val) && (pos < arr.length)`
- (E) `(pos < arr.length) || (arr[pos] == val)`

5. Consider the following code segments.

I.

```
int k = 1;
while (k < 20)
{
    if (k % 3 == 1)
        System.out.print( k + " ");

    k = k + 3;
}
```

II.

```
for (int k = 1; k < 20; k++)
{
    if (k % 3 == 1)
        System.out.print( k + " ");
}
```

III.

```
for (int k = 1; k < 20; k = k + 3)
{
    System.out.print( k + " ");
}
```

Which of the code segments above will produce the following output?

1 4 7 10 13 16 19

- (A) I only
- (B) II only
- (C) I and II only
- (D) II and III only
- (E) I, II, and III

Section I

36. Consider the following two methods that appear within a single class.

```
public void changeIt(int[] list, int num)
{
    list = new int[5];
    num = 0;

    for (int x = 0; x < list.length; x++)
        list[x] = 0;
}
```

```
public void start()
{
    int[] nums = {1, 2, 3, 4, 5};
    int value = 6;

    changeIt(nums, value);

    for (int k = 0; k < nums.length; k++)
        System.out.print(nums[k] + " ");

    System.out.print(value);
}
```

What is printed as a result of the call `start()` ?

- (A) 0 0 0 0 0 0
- (B) 0 0 0 0 0 6
- (C) 1 2 3 4 5 6
- (D) 1 2 3 4 5 0
- (E) `changeIt` will throw an exception.

7. Consider the following declaration of the class `NumSequence`, which has a constructor that is intended to initialize the instance variable `seq` to an `ArrayList` of `numberOfValues` random floating-point values in the range `[0.0, 1.0)`.

```
public class NumSequence
{
    private ArrayList<Double> seq;

    // precondition: numberOfValues > 0
    // postcondition: seq has been initialized to an ArrayList of
    //                length numberOfValues; each element of seq
    //                contains a random Double in the range [0.0, 1.0)
    public NumSequence(int numberOfValues)
    {
        /* missing code */
    }
}
```

Which of the following code segments could be used to replace `/* missing code */` so that the constructor will work as intended?

- I.

```
ArrayList<Double> seq = new ArrayList<Double>();
for (int k = 0; k < numberOfValues; k++)
    seq.add(new Double(Math.random()));
```
- II.

```
seq = new ArrayList<Double>();
for (int k = 0; k < numberOfValues; k++)
    seq.add(new Double(Math.random()));
```
- III.

```
ArrayList<Double> temp = new ArrayList<Double>();
for (int k = 0; k < numberOfValues; k++)
    temp.add(new Double(Math.random()));

seq = temp;
```

- (A) II only
- (B) III only
- (C) I and II
- (D) I and III
- (E) II and III

Section I

38. Consider the following code segment.

```
double a = 1.1;
double b = 1.2;

if ((a + b) * (a - b) != (a * a) - (b * b))
{
    System.out.println("Mathematical error!");
}
```

Which of the following best describes why the phrase "Mathematical error! " would be printed?
(Remember that mathematically $(a + b) * (a - b) = a^2 - b^2$.)

- (A) Precedence rules make the if condition true.
 - (B) Associativity rules make the if condition true.
 - (C) Roundoff error makes the if condition true.
 - (D) Overflow makes the if condition true.
 - (E) A compiler bug or hardware error has occurred.
-

39. Consider the following recursive method.

```
public static String recur(int val)
{
    String dig = "" + (val % 3);

    if (val / 3 > 0)
        return dig + recur(val / 3);

    return dig;
}
```

What is printed as a result of executing the following statement?

```
System.out.println(recur(32));
```

- (A) 20
- (B) 102
- (C) 210
- (D) 1020
- (E) 2101

0. Consider the following method.

```
public String goAgain(String str, int index)
{
    if (index >= str.length())
        return str;

    return str + goAgain(str.substring(index), index + 1);
}
```

What is printed as a result of executing the following statement?

```
System.out.println(goAgain("today", 1));
```

- (A) today
- (B) todayto
- (C) todayoday
- (D) todayodayay
- (E) todayodaydayayy

END OF SECTION I

**IF YOU FINISH BEFORE TIME IS CALLED,
YOU MAY CHECK YOUR WORK ON THIS SECTION.**

DO NOT GO ON TO SECTION II UNTIL YOU ARE TOLD TO DO SO.

NO TEST MATERIAL ON THIS PAGE

Content of Appendixes

Appendix A	A Exam Java Quick Reference
Appendix B	Testable API
Appendix C	Testable Code for APCS A/AB
Appendix E	Quick Reference A/AB
Appendix G	Index for Source Code

Appendix A — A Exam Java Quick Reference

Accessible Methods from the Java Library That May Be Included on the Exam

class java.lang.Object

- boolean equals(Object other)
- String toString()

class java.lang.Integer

- Integer(int value)
- int intValue()

class java.lang.Double

- Double(double value)
- double doubleValue()

class java.lang.String

- int length()
- String substring(int from, int to) // returns the substring beginning at from
// and ending at to-1
- String substring(int from) // returns substring(from, length())
- int indexOf(String str) // returns the index of the first occurrence of str;
// returns -1 if not found
- int compareTo(String other) // returns a value < 0 if this is less than other
// returns a value = 0 if this is equal to other
// returns a value > 0 if this is greater than other

class java.lang.Math

- static int abs(int x)
- static double abs(double x)
- static double pow(double base, double exponent)
- static double sqrt(double x)
- static double random() // returns a double in the range [0.0, 1.0)

class java.util.ArrayList<E>

- int size()
- boolean add(E obj) // appends obj to end of list; returns true
- void add(int index, E obj) // inserts obj at position index ($0 \leq \text{index} \leq \text{size}$),
// moving elements at position index and higher
// to the right (adds 1 to their indices) and adjusts size
- E get(int index)
- E set(int index, E obj) // replaces the element at position index with obj
// returns the element formerly at the specified position
- E remove(int index) // removes element from position index, moving elements
// at position index + 1 and higher to the left
// (subtracts 1 from their indices) and adjusts size
// returns the element formerly at the specified position

Appendix B — Testable API

info.gridworld.grid.Location class (implements Comparable)

```
public Location(int r, int c)
    constructs a location with given row and column coordinates

public int getRow()
    returns the row of this location

public int getCol()
    returns the column of this location

public Location getAdjacentLocation(int direction)
    returns the adjacent location in the direction that is closest to direction

public int getDirectionToward(Location target)
    returns the closest compass direction from this location toward target

public boolean equals(Object other)
    returns true if other is a Location with the same row and column as this location; false otherwise

public int hashCode()
    returns a hash code for this location

public int compareTo(Object other)
    returns a negative integer if this location is less than other, zero if the two locations are equal, or a positive
    integer if this location is greater than other. Locations are ordered in row-major order.
    Precondition: other is a Location object.

public String toString()
    returns a string with the row and column of this location, in the format (row, col)
```

Compass directions:

```
public static final int NORTH = 0;
public static final int EAST = 90;
public static final int SOUTH = 180;
public static final int WEST = 270;
public static final int NORTHEAST = 45;
public static final int SOUTHEAST = 135;
public static final int SOUTHWEST = 225;
public static final int NORTHWEST = 315;
```

Turn angles:

```
public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;
```

info.gridworld.grid.Grid<E> interface

`int getNumRows()`
returns the number of rows, or -1 if this grid is unbounded

`int getNumCols()`
returns the number of columns, or -1 if this grid is unbounded

`boolean isValid(Location loc)`
returns true if `loc` is valid in this grid, false otherwise
Precondition: `loc` is not null

`E put(Location loc, E obj)`
puts `obj` at location `loc` in this grid and returns the object previously at that location (or null if the location was previously unoccupied).
Precondition: (1) `loc` is valid in this grid (2) `obj` is not null

`E remove(Location loc)`
removes the object at location `loc` from this grid and returns the object that was removed (or null if the location is unoccupied)
Precondition: `loc` is valid in this grid

`E get(Location loc)`
returns the object at location `loc` (or null if the location is unoccupied)
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedLocations()`
returns an array list of all occupied locations in this grid

`ArrayList<Location> getValidAdjacentLocations(Location loc)`
returns an array list of the valid locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getEmptyAdjacentLocations(Location loc)`
returns an array list of the valid empty locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedAdjacentLocations(Location loc)`
returns an array list of the valid occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<E> getNeighbors(Location loc)`
returns an array list of the objects in the occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

info.gridworld.actor.Actor class

```
public Actor()  
    constructs a blue actor that is facing north  
  
public Color getColor()  
    returns the color of this actor  
  
public void setColor(Color newColor)  
    sets the color of this actor to newColor  
  
public int getDirection()  
    returns the direction of this actor, an angle between 0 and 359 degrees  
  
public void setDirection(int newDirection)  
    sets the direction of this actor to the angle between 0 and 359 degrees that is equivalent to newDirection  
  
public Grid<Actor> getGrid()  
    returns the grid of this actor, or null if this actor is not contained in a grid  
  
public Location getLocation()  
    returns the location of this actor, or null if this actor is not contained in a grid  
  
public void putSelfInGrid(Grid<Actor> gr, Location loc)  
    puts this actor into location loc of grid gr. If there is another actor at loc, it is removed.  
    Precondition: (1) This actor is not contained in a grid (2) loc is valid in gr  
  
public void removeSelfFromGrid()  
    removes this actor from its grid.  
    Precondition: this actor is contained in a grid  
  
public void moveTo(Location newLocation)  
    moves this actor to newLocation. If there is another actor at newLocation, it is removed.  
    Precondition: (1) This actor is contained in a grid (2) newLocation is valid in the grid of this actor  
  
public void act()  
    reverses the direction of this actor. Override this method in subclasses of Actor to define types of actors with  
    different behavior  
  
public String toString()  
    returns a string with the location, direction, and color of this actor
```

info.gridworld.actor.Rock class (extends Actor)

```
public Rock()  
    constructs a black rock  
  
public Rock(Color rockColor)  
    constructs a rock with color rockColor  
  
public void act()  
    overrides the act method in the Actor class to do nothing
```

info.gridworld.actor.Flower class (extends Actor)

```
public Flower()  
    constructs a pink flower  
  
public Flower(Color initialColor)  
    constructs a flower with color initialColor  
  
public void act()  
    causes the color of this flower to darken
```

Appendix C — Testable Code for APCS A/AB

Bug.java

```

package info.gridworld.actor;

import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A Bug is an actor that can move and turn. It drops flowers as it moves.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Bug extends Actor

    /**
     * Constructs a red bug.
     */
    public Bug()
    {
        setColor(Color.RED);
    }

    /**
     * Constructs a bug of a given color.
     * @param bugColor the color for this bug
     */
    public Bug(Color bugColor)
    {
        setColor(bugColor);
    }

    /**
     * Moves if it can move, turns otherwise.
     */
    public void act()
    {
        if (canMove())
            move();
        else
            turn();
    }

    /**
     * Turns the bug 45 degrees to the right without changing its location.
     */
    public void turn()
    {
        setDirection(getDirection() + Location.HALF_RIGHT);
    }

```

```
/**
 * Moves the bug forward, putting a flower into the location it previously occupied.
 */
public void move()
{
    Grid<Actor> gr = getGrid();
    if (gr == null)
        return;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (gr.isValid(next))
        moveTo(next);
    else
        removeSelfFromGrid();
    Flower flower = new Flower(getColor());
    flower.putSelfInGrid(gr, loc);
}

/**
 * Tests whether this bug can move forward into a location that is empty or contains a flower.
 * @return true if this bug can move.
 */
public boolean canMove()
{
    Grid<Actor> gr = getGrid();
    if (gr == null)
        return false;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (!gr.isValid(next))
        return false;
    Actor neighbor = gr.get(next);
    return (neighbor == null) || (neighbor instanceof Flower);
    // ok to move into empty location or onto flower
    // not ok to move onto any other actor
}
}
```

BoxBug.java

```

import info.gridworld.actor.Bug;

/**
 * A BoxBug traces out a square "box" of a given size.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class BoxBug extends Bug {

    private int steps;
    private int sideLength;

    /**
     * Constructs a box bug that traces a square of a given side length
     * @param length the side length
     */
    public BoxBug(int length) {
        steps = 0;
        sideLength = length;
    }

    /**
     * Moves to the next location of the square.
     */
    public void act() {
        if (steps < sideLength && canMove())
        {
            move();
            steps++;
        }
        else
        {
            turn();
            turn();
            steps = 0;
        }
    }
}

```

Criticr.java

```

package info.gridworld.actor;

import info.gridworld.grid.Location;
import java.util.ArrayList;

/**
 * A Critter is an actor that moves through its world, processing
 * other actors in some way and then moving to a new location.
 * Define your own critters by extending this class and overriding any methods of this class except for act.
 * When you override these methods, be sure to preserve the postconditions.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Critter extends Actor
{
    /**
     * A critter acts by getting a list of other actors, processing that list, getting locations to move to,
     * selecting one of them, and moving to the selected location.
     */
    public void act()
    {
        if (getGrid() == null)
            return;
        ArrayList<Actor> actors = getActors();
        processActors(actors);
        ArrayList<Location> moveLocs = getMoveLocations();
        Location loc = selectMoveLocation(moveLocs);
        makeMove(loc);
    }

    /**
     * Gets the actors for processing. Implemented to return the actors that occupy neighboring grid locations.
     * Override this method in subclasses to look elsewhere for actors to process.
     * Postcondition: The state of all actors is unchanged.
     * @return a list of actors that this critter wishes to process.
     */
    public ArrayList<Actor> getActors()
    {
        return getGrid().getNeighbors(getLocation());
    }
}

```

```

/**
 * Processes the elements of actors. New actors may be added to empty locations.
 * Implemented to "eat" (i.e., remove) selected actors that are not rocks or critters.
 * Override this method in subclasses to process actors in a different way.
 * Postcondition: (1) The state of all actors in the grid other than this critter and the
 * elements of actors is unchanged. (2) The location of this critter is unchanged.
 * @param actors the actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
    for (Actor a : actors)
    {
        if (!(a instanceof Rock) && !(a instanceof Critter))
            a.removeSelfFromGrid();
    }
}

/**
 * Gets a list of possible locations for the next move. These locations must be valid in the grid of this critter.
 * Implemented to return the empty neighboring locations. Override this method in subclasses to look
 * elsewhere for move locations.
 * Postcondition: The state of all actors is unchanged.
 * @return a list of possible locations for the next move
 */
public ArrayList<Location> getMoveLocations()
{
    return getGrid().getEmptyAdjacentLocations(getLocation());
}

/**
 * Selects the location for the next move. Implemented to randomly pick one of the possible locations,
 * or to return the current location if locs has size 0. Override this method in subclasses that
 * have another mechanism for selecting the next move location.
 * Postcondition: (1) The returned location is an element of locs, this critter's current location, or null.
 * (2) The state of all actors is unchanged.
 * @param locs the possible locations for the next move
 * @return the location that was selected for the next move.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
{
    int n = locs.size();
    if (n == 0)
        return getLocation();
    int r = (int) (Math.random() * n);
    return locs.get(r);
}

```

```

/**
 * Moves this critter to the given location loc, or removes this critter from its grid if loc is null.
 * An actor may be added to the old location. If there is a different actor at location loc, that actor is
 * removed from the grid. Override this method in subclasses that want to carry out other actions
 * (for example, turning this critter or adding an occupant in its previous location).
 * Postcondition: (1) getLocation() == loc.
 * (2) The state of all actors other than those at the old and new locations is unchanged.
 * @param loc the location to move to
 */
public void makeMove(Location loc)
{
    if (loc == null)
        removeSelfFromGrid();
    else
        moveTo(loc);
}

```

ChameleonCriticter.java

```

import info.gridworld.actor.Actor;
import info.gridworld.actor.Criticter;
import info.gridworld.grid.Location;

import java.util.ArrayList;

/**
 * A ChameleonCriticter takes on the color of neighboring actors as it moves through the grid.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class ChameleonCriticter extends Criticter
{
    /**
     * Randomly selects a neighbor and changes this critter's color to be the same as that neighbor's.
     * If there are no neighbors, no action is taken.
     */
    public void processActors(ArrayList<Actor> actors)
    {
        int n = actors.size();
        if (n == 0)
            return;
        int r = (int) (Math.random() * n);

        Actor other = actors.get(r);
        setColor(other.getColor());
    }

    /**
     * Turns towards the new location as it moves.
     */
    public void makeMove(Location loc)
    {
        setDirection(getLocation().getDirectionToward(loc));
        super.makeMove(loc);
    }
}

```

Appendix E — Quick Reference A/AB

Location Class (implements Comparable)

```
public Location(int r, int c)
public int getRow()
public int getCol()
public Location getAdjacentLocation(int direction)
public int getDirectionToward(Location target)
public boolean equals(Object other)
public int hashCode()
public int compareTo(Object other)
public String toString()
```

NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST
LEFT, RIGHT, HALF_LEFT, HALF_RIGHT, FULL_CIRCLE, HALF_CIRCLE, AHEAD

Grid<E> Interface

```
int getNumRows()
int getNumCols()
boolean isValid(Location loc)
void put(Location loc, E obj)
void remove(Location loc)
Location get(Location loc)
ArrayList<Location> getOccupiedLocations()
ArrayList<Location> getValidAdjacentLocations(Location loc)
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
ArrayList<E> getNeighbors(Location loc)
```

Actor Class

```
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
```

Rock Class (extends Actor)

```
public Rock()  
public Rock(Color rockColor)  
public void act()
```

Flower Class (extends Actor)

```
public Flower()  
public Flower(Color initialColor)  
public void act()
```

Bug Class (extends Actor)

```
public Bug()  
public Bug(Color bugColor)  
public void act()  
public void turn()  
public void move()  
public boolean canMove()
```

BoxBug Class (extends Bug)

```
public BoxBug(int n)  
public void act()
```

Critter Class (extends Actor)

```
public void act()  
public ArrayList<Actor> getActors()  
public void processActors(ArrayList<Actor> actors)  
public ArrayList<Location> getMoveLocations()  
public Location selectMoveLocation(ArrayList<Location> locs)  
public void makeMove(Location loc)
```

ChameleonCritter Class (extends Critter)

```
public void processActors(ArrayList<Actor> actors)  
public void makeMove(Location loc)
```

Appendix G — Index for Source Code

This appendix provides an index for the Java source code found in Appendix C.

Bug.java

Bug()	C1
Bug (Color bugColor)	C1
act()	C1
turn()	C1
move()	C2
canMove()	C2

BoxBug.java

BoxBug (int length)	C3
act()	C3

Critter.java

act()	C4
getActors()	C4
processActors (ArrayList<Actor> actors)	C5
getMoveLocations()	C5
selectMoveLocation (ArrayList<Location> locs)	C5
makeMove (Location loc)	C6

ChameleonCritter.java

processActors (ArrayList<Actor> actors)	C6
makeMove (Location loc)	C6

AP[®] Computer Science A Exam

SECTION II: Free-Response Questions

2009

DO NOT OPEN THIS BOOKLET UNTIL YOU ARE TOLD TO DO SO.

At a Glance**Total Time**

1 hour, 45 minutes

Number of Questions

4

Percent of Total Grade

50%

Writing Instrument

Pencil

Electronic Device

None allowed

Weight

The questions are weighted equally.

IMPORTANT Identification Information

PLEASE PRINT WITH PEN:

1. First two letters of your last name
-
-

First letter of your first name

2. Date of birth

Month

Day

Year

3. Six-digit school code

4. Unless I check the box below, I grant the College Board the unlimited right to use, reproduce, and publish my free-response materials, both written and oral, for educational research and instructional purposes. My name and the name of my school will not be used in any way in connection with my free-response materials. I understand that I am free to mark "No" with no effect on my grade or its reporting.

No, I do not grant the College Board these rights. ☐**Instructions**

The questions for Section II are printed in the green insert and in this booklet. You may use the insert to organize your answers and for scratch work, but you must write your answers in this pink Section II booklet. No credit will be given for any work written in the insert. The orange Appendix booklet is provided for use in Section II.

Write your answer to each question in the space provided in the Section II booklet. Some questions require you to write program segments, and these must be written in Java. Show all your work. Credit for partial solutions will be given. Write clearly and legibly. Cross out any errors you make. Erased or crossed-out work will not be graded.

Assume classes listed in the Quick Reference found in the Appendix are included in any program that uses a program segment you write. If other classes are to be included, that information will be specified in individual questions. Unless otherwise noted, assume that all methods are invoked only when their preconditions are satisfied. A Quick Reference to required Java classes is included in the Appendix.

Manage your time carefully. Do not spend too much time on any one question. You may proceed freely from one question to the next. You may review your responses if you finish before the end of the exam is announced.

**COMPUTER SCIENCE A
SECTION II**

Time—1 hour and 45 minutes

Number of questions—4

Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.

Unless otherwise noted in the question, assume that parameters in method calls are not null and that methods are called only when their preconditions are satisfied.

In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

Section II

1. A statistician is studying sequences of numbers obtained by repeatedly tossing a six-sided number cube. On each side of the number cube is a single number in the range of 1 to 6, inclusive, and no number is repeated on the cube. The statistician is particularly interested in runs of numbers. A run occurs when two or more consecutive tosses of the cube produce the same value. For example, in the following sequence of cube tosses, there are runs starting at positions 1, 6, 12, and 14.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Result	1	5	5	4	3	1	2	2	2	2	6	1	3	3	5	5	5	5

The number cube is represented by the following class.

```
public class NumberCube
{
    /** @return an integer value between 1 and 6, inclusive
     */
    public int toss()
    { /* implementation not shown */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

You will implement a method that collects the results of several tosses of a number cube and another method that calculates the longest run found in a sequence of tosses.

Write the method `getCubeTosses` that takes a number cube and a number of tosses as parameters. The method should return an array of the values produced by tossing the number cube the given number of times.

Complete method `getCubeTosses` below.

```
/** Returns an array of the values obtained by tossing a number cube numTosses times.
 * @param cube a NumberCube
 * @param numTosses the number of tosses to be recorded
 *      Precondition: numTosses > 0
 * @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
```

Section II

- (b) Write the method `getLongestRun` that takes as its parameter an array of integer values representing a series of number cube tosses. The method returns the starting index in the array of a run of maximum size. A run is defined as the repeated occurrence of the same value in two or more consecutive positions in the array.

For example, the following array contains two runs of length 4, one starting at index 6 and another starting at index 14. The method may return either of those starting indexes.

If there are no runs of any value, the method returns `-1`.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Result	1	5	5	4	3	1	2	2	2	2	6	1	3	3	5	5	5	5

WRITE YOUR SOLUTION ON THE NEXT PAGE.

complete method `getLongestRun` below.

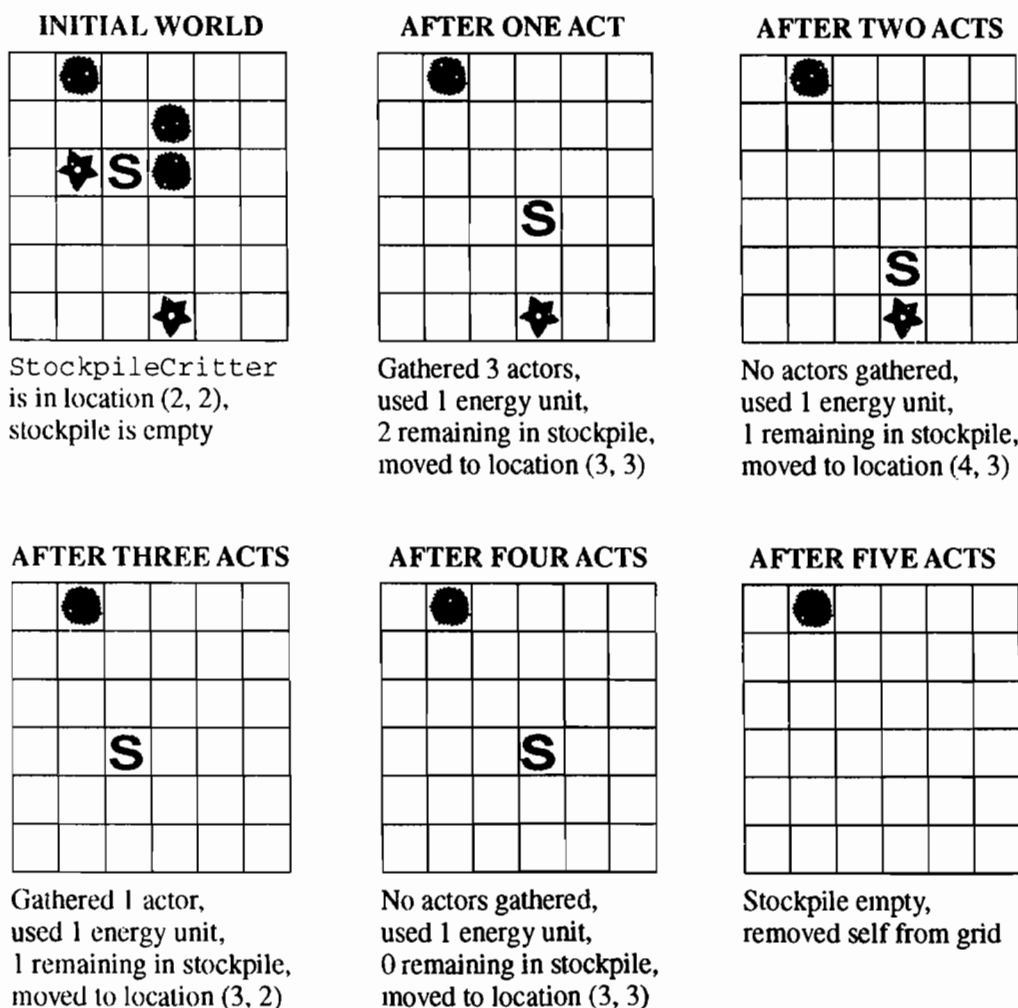
```
* Returns the starting index of a longest run of two or more consecutive repeated values
  in the array values.
  @param values an array of integer values representing a series of number cube tosses
           Precondition: values.length > 0
  @return the starting index of a run of maximum size;
           -1 if there is no run
/
public static int getLongestRun(int[] values)
```

Section II

2. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

A `StockpileCriticter` is a `Criticter` that uses other actors as a source of energy. Each actor represents one unit of energy. The `StockpileCriticter` behaves like a `Criticter` except in the way that it interacts with other actors. Each time the `StockpileCriticter` acts, it gathers all neighboring actors by removing them from the grid and keeps track of them in a stockpile. The `StockpileCriticter` then attempts to reduce its stockpile by one unit of energy. If the stockpile is empty, the `StockpileCriticter` runs out of energy and removes itself from the grid.

Consider the following scenario.



Write the complete `StockpileCriticter` class, including all instance variables and required methods. Do NOT override the `act` method. Remember that your design must not violate the postconditions of the methods of the `Criticter` class and that updating an object's instance variable changes the state of that object.

ADDITIONAL WORK SPACE

Section II

3. An electric car that runs on batteries must be periodically recharged for a certain number of hours. The battery technology in the car requires that the charge time not be interrupted.

The cost for charging is based on the hour(s) during which the charging occurs. A rate table lists the 24 one-hour periods, numbered from 0 to 23, and the corresponding hourly cost for each period. The same rate table is used for each day. Each hourly cost is a positive integer. A sample rate table is given below.

Hour	Cost	Hour	Cost	Hour	Cost
0	50	8	150	16	200
1	60	9	150	17	200
2	160	10	150	18	180
3	60	11	200	19	180
4	80	12	40	20	140
5	100	13	240	21	100
6	100	14	220	22	80
7	120	15	220	23	60

The class `BatteryCharger` below uses a rate table to determine the most economic time to charge the battery. You will write two of the methods for the `BatteryCharger` class.

```
public class BatteryCharger
{
    /** rateTable has 24 entries representing the charging costs for hours 0 through 23. */
    private int[] rateTable;

    /** Determines the total cost to charge the battery starting at the beginning of startHour.
     * @param startHour the hour at which the charge period begins
     *      Precondition:  $0 \leq \text{startHour} \leq 23$ 
     * @param chargeTime the number of hours the battery needs to be charged
     *      Precondition:  $\text{chargeTime} > 0$ 
     * @return the total cost to charge the battery
     */
    private int getChargingCost(int startHour, int chargeTime)
    {
        /* to be implemented in part (a) */
    }

    /** Determines start time to charge the battery at the lowest cost for the given charge time.
     * @param chargeTime the number of hours the battery needs to be charged
     *      Precondition:  $\text{chargeTime} > 0$ 
     * @return an optimal start time, with  $0 \leq \text{returned value} \leq 23$ 
     */
    public int getChargeStartTime(int chargeTime)
    {
        /* to be implemented in part (b) */
    }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

- a) Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```
/** Determines the total cost to charge the battery starting at the beginning of startHour.
 * @param startHour the hour at which the charge period begins
 *      Precondition:  $0 \leq \text{startHour} \leq 23$ 
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition:  $\text{chargeTime} > 0$ 
 * @return the total cost to charge the battery
 */
private int getChargingCost(int startHour, int chargeTime)
```

Section II

- (b) Write the `BatteryCharger` method `getChargeStartTime` that returns the start time that will allow the battery to be charged at minimal cost. If there is more than one possible start time that produces the minimal cost, any of those start times can be returned.

For example, using the rate table given at the beginning of the question, the following table shows the resulting minimal costs and optimal starting hour of several possible charges.

Hours of Charge Time	Minimum Cost	Start Hour of Charge	Last Hour of Charge
1	40	12	12
2	110	0 or 23	1 0 (the next day)
7	550	22	4 (the next day)
30	3,710	22	3 (two days later)

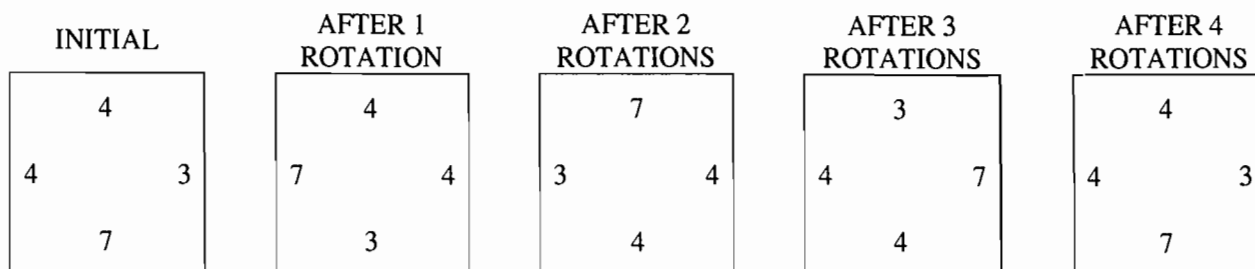
WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).
Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return an optimal start time, with  $0 \leq \text{returned value} \leq 23$ 
 */
public int getChargeStartTime(int chargeTime)
```

Section II

4. A game uses square tiles that have numbers on their sides. Each tile is labeled with a number on each of its four sides and may be rotated clockwise, as illustrated below.



The tiles are represented by the `NumberTile` class, as given below.

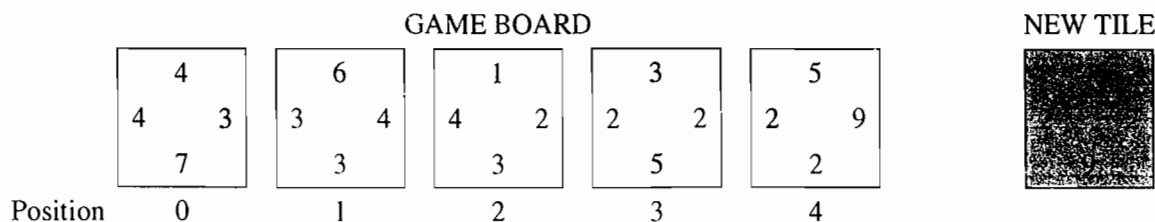
```
public class NumberTile
{
    /** Rotates the tile 90 degrees clockwise
     */
    public void rotate()
    { /* implementation not shown */ }

    /** @return value at left edge of tile
     */
    public int getLeft()
    { /* implementation not shown */ }

    /** @return value at right edge of tile
     */
    public int getRight()
    { /* implementation not shown */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

Tiles are placed on a game board so that the adjoining sides of adjacent tiles have the same number. The following figure illustrates an arrangement of tiles and shows a new tile that is to be placed on the game board.



its original orientation, the new tile can be inserted between the tiles at positions 2 and 3 or between the tiles at positions 3 and 4. If the new tile is rotated once, it can be inserted before the tile at position 0 (the first tile) or after the tile at position 4 (the last tile). Assume that the new tile, in its original orientation, is inserted between the tiles at positions 2 and 3. As a result of the insertion, the tiles at positions 3 and 4 are moved one location to the right, and the new tile is inserted at position 3, as shown below.

GAME BOARD AFTER INSERTING TILE					
<div>4 4 3 7</div>	<div>6 3 4 3</div>	<div>1 4 2 3</div>	<div>4 2 2 9</div>	<div>3 2 2 5</div>	<div>5 2 9 2</div>
Position 0	1	2	3	4	5

partial definition of the `TileGame` class is given below.

```
public class TileGame
{
    /** represents the game board; guaranteed never to be null */
    private ArrayList<NumberTile> board;

    public TileGame()
    { board = new ArrayList<NumberTile>(); }

    /** Determines where to insert tile, in its current orientation, into game board
     *  @param tile the tile to be placed on the game board
     *  @return the position of tile where tile is to be inserted:
     *          0 if the board is empty;
     *          -1 if tile does not fit in front, at end, or between any existing tiles;
     *          otherwise, 0 ≤ position returned ≤ board.size()
     */
    private int getIndexForFit(NumberTile tile)
    { /* to be implemented in part (a) */ }

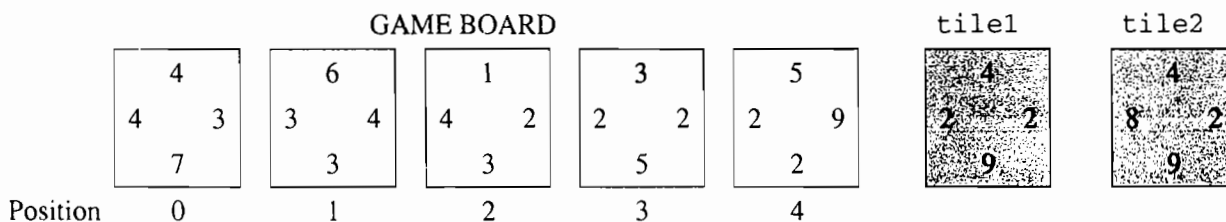
    /** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
     *  * If there are no tiles on the game board, the tile is placed at position 0.
     *  * The tile should be placed at most 1 time.
     *  * Precondition: board is not null
     *  * @param tile the tile to be placed on the game board
     *  * @return true if tile is placed successfully; false otherwise
     *  * Postcondition: the orientations of the other tiles on the board are not changed
     *  * Postcondition: the order of the other tiles on the board relative to each other is not changed
     */
    public boolean insertTile(NumberTile tile)
    { /* to be implemented in part (b) */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

Section II

- (a) Write the `TileGame` method `getIndexForFit` that determines where a given tile, in its current orientation, fits on the game board. A tile can be inserted at either end of a game board or between two existing tiles if the side(s) of the new tile match the adjacent side(s) of the tile(s) currently on the game board. If there are no tiles on the game board, the position for the insert is 0. The method returns the position that the new tile will occupy on the game board after it has been inserted. If there are multiple possible positions for the tile, the method will return any one of them. If the given tile does not fit anywhere on the game board, the method returns `-1`.

For example, the following diagram shows a game board and two potential tiles to be placed. The call `getIndexForFit(tile1)` can return either 3 or 4 because `tile1` can be inserted between the tiles at positions 2 and 3, or between the tiles at positions 3 and 4. The call `getIndexForFit(tile2)` returns `-1` because `tile2`, in its current orientation, does not fit anywhere on the game board.



WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `getIndexForFit` below.

```
/** Determines where to insert tile, in its current orientation, into game board
 * @param tile the tile to be placed on the game board
 * @return the position of tile where tile is to be inserted:
 *         0 if the board is empty;
 *         -1 if tile does not fit in front, at end, or between any existing tiles;
 *         otherwise,  $0 \leq \text{position returned} \leq \text{board.size}()$ 
 */
private int getIndexForFit(NumberTile tile)
```

Section II

- (b) Write the `TileGame` method `insertTile` that attempts to insert the given tile on the game board. The method returns true if the tile is inserted successfully and false only if the tile cannot be placed on the board in any orientation.

Assume that `getIndexForFit` works as specified, regardless of what you wrote in part (a).

Complete method `insertTile` below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 * If there are no tiles on the game board, the tile is placed at position.
 * The tile should be placed at most 1 time.
 * Precondition: board is not null
 * @param tile the tile to be placed on the game board
 * @return true if tile is placed successfully; false otherwise
 * Postcondition: the orientations of the other tiles on the board are not changed
 * Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
public boolean insertTile(NumberTile tile)
```

STOP

END OF EXAM

AP Computer Science A

2009

APPENDIX - ANSWERS

Matching multiple-choice answers and free-response scoring follow.

Java/GridWorld edition; original case-study references retained.

Chapter III: Answers to the 2009 AP Computer Science A Exam

Section I: Multiple Choice

- Section I Answer Key and Percent Answering Correctly
- Analyzing Your Students' Performance on the Multiple-Choice Section
- Diagnostic Guide for the 2009 AP Computer Science A Exam

Section II: Free Response

- Comments from the Chief Reader
- Scoring Guidelines, Sample Student Responses, and Commentary

Section I: Multiple Choice

Listed below are the correct answers to the multiple-choice questions, the percent of AP students who answered each question correctly by AP score, and the total percent answering correctly.

Section I Answer Key and Percent Answering Correctly

Item No.	Correct Answer	Percent Correct by Score					Total Percent Correct
		5	4	3	2	1	
1	A	98	97	96	96	79	92
2	A	98	95	93	89	73	89
3	D	99	96	90	84	50	81
4	C	97	93	88	81	56	81
5	A	95	86	76	73	53	76
6	A	93	80	69	60	30	65
7	D	88	65	42	28	16	50
8	D	99	96	91	85	63	85
9	C	69	47	39	31	20	42
10	D	92	85	77	74	49	74
11	A	96	83	65	47	27	64
12	B	96	93	91	88	57	82
13	C	73	35	17	8	13	33
14	D	86	64	44	37	23	52
15	B	83	69	63	58	44	63
16	C	72	49	40	36	36	48
17	B	77	48	32	19	10	39
18	A	92	78	67	61	42	68
19	D	95	74	55	41	19	57
20	E	96	85	69	58	30	67

Item No.	Correct Answer	Percent Correct by Score					Total Percent Correct
		5	4	3	2	1	
21	B	83	66	56	47	31	56
22	E	88	71	56	44	20	56
23	C	93	83	75	66	50	73
24	A	96	88	78	71	44	74
25	C	47	23	19	20	20	27
26	D	91	75	62	54	29	61
27	C	90	61	34	23	15	47
28	B	88	68	52	44	25	56
29	D	84	71	52	37	18	53
30	E	93	79	61	45	21	60
31	C	88	68	61	57	46	64
32	B	86	62	47	37	25	52
33	C	80	58	37	27	15	45
34	A	67	41	28	22	14	36
35	E	92	76	61	54	31	63
36	C	37	20	13	14	19	22
37	E	63	29	12	7	6	26
38	C	44	25	19	18	16	25
39	E	41	19	12	8	5	18
40	D	52	25	15	11	8	24

Analyzing Your Students' Performance on the Multiple-Choice Section

If you give your students the 2009 exam for practice, you may want to analyze the results to find overall strengths and weaknesses in their understanding of AP Computer Science

A. The following diagnostic worksheet will help you do this. You are permitted to photocopy and distribute it to your students for completion.

1. In each section, students should insert a check mark for each correct answer.
2. Add together the total number of correct answers for each section.

3. To compare the student's number of correct answers for each section with the average number correct for that section, copy the number of correct answers to the "Number Correct" table at the end of the Diagnostic Guide.

In addition, under each item the percent of AP students who answered correctly is shown, so students can analyze their performance on individual items. This information will be helpful in deciding how students should plan their study time. Please note that one item may appear in several different categories, as questions can cross over different topics.

Diagnostic Guide for the 2009 AP Computer Science A Exam

Programming Fundamentals (Average number correct = 17.43)

Question #	1	2	3	4	6	7	8	9	10	11	12	13	14	15	17
Correct/Incorrect															
Percent of Students Answering Correctly	92	89	81	81	65	50	85	42	74	64	82	33	52	63	39

Question #	18	19	20	21	23	26	27	29	32	33	34	35	36	37	40
Correct/Incorrect															
Percent of Students Answering Correctly	68	57	67	56	73	61	47	53	52	45	36	63	22	26	24

Data Structures (Average number correct = 3.97)

Question #	3	4	11	17	27	34	36	37
Correct/Incorrect								
Percent of Students Answering Correctly	81	81	64	39	47	36	22	26

Logic (Average number correct = 2.54)

Question #	18	20	28	31
Correct/Incorrect				
Percent of Students Answering Correctly	68	67	56	64

Algorithms/Problem Solving (Average number correct = 8.83)

Question #	1	2	3	6	7	8	12	14	15	17	22	25	27	28
Correct/Incorrect														
Percent of Students Answering Correctly	92	89	81	65	50	85	82	52	63	39	56	27	47	56

Object-Oriented Programming (Average number correct = 4.14)

Question #	5	16	22	23	24	25	30
Correct/Incorrect							
Percent of Students Answering Correctly	76	48	56	73	74	27	60

Diagnostic Guide for the 2009 AP Computer Science A Exam (continued)

Recursion (Average number correct = 0.74)

Question #	13	39	40
Correct/Incorrect			
Percent of Students Answering Correctly	33	18	24

Software Engineering (Average number correct = 0.25)

Question #	38
Correct/Incorrect	
Percent of Students Answering Correctly	25

Case Study (Average number correct = 2.86)

Question #	21	22	23	24	25
Correct/Incorrect					
Percent of Students Answering Correctly	56	56	73	74	27

Number Correct

	Programming Fundamentals	Data Structures	Logic	Algorithms/ Problem Solving	Object-Oriented Programming	Recursion	Software Engineering	Case Study
Number of Questions	30	8	4	14	7	3	1	5
Average Number Correct	17.43 (58.1%)	3.97 (49.6%)	2.54 (63.5%)	8.83 (63.1%)	4.14 (59.1%)	0.74 (24.7%)	0.25 (25.0%)	2.86 (57.2%)
My Number Correct								

Section II: Free Response

Comments from the Chief Reader

Jody Paul

Department of Computer Science

Metropolitan State College of Denver

Denver, Colorado

The free-response section of the 2009 AP Computer Science A Exam is comparable in content, structure, and difficulty to exams from the preceding several years. It consists of 4 multi-part questions that afforded performance-based assessment of fundamental computer science skills and concepts from the curriculum defined in the *AP Computer Science A Course Description*. The free-response questions were scored according to rigorous standards and procedures, and the resultant scores comprised 50 percent of the composite score for the exam.

All questions required students to read and analyze problem descriptions and to design and generate programs using the Java programming language. Question 2 required students to design and implement a complete Java class derived from one defined in the *GridWorld AP Computer Science Case Study*. All others required writing the bodies of Java methods for which signatures were provided.

Question 1 focused on the array data structure, its construction and traversal, the application of basic algorithms, and method invocation for a specified object. Question 2 involved reasoning about code from the *GridWorld* case study, emphasizing object-oriented concepts. Question 3 focused on array traversal, abstraction, and algorithms for accumulation and finding a minimum. Question 4 focused on object-oriented programming, algorithm development and list processing, and required significant problem analysis and algorithm design before writing code. The dominant concepts assessed included the following:

- interpretation of textual problem statements (a problem-solving skill)
- object-oriented design and programming
- adhering to specified interfaces and constraints
- Java-based array construction, traversal and element access
- manipulation and element access of Java's `ArrayList` objects
- translation of algorithms into Java programming language code

- implementation of common algorithmic tasks (such as finding maximum or minimum values)
- numerical iteration using Java's `for` or `while` loop constructs
- correct use of the Java programming language

The most prevalent errors noticed during scoring included:

- failure to meet required solution specifications (such as violations of well-specified constraints or failure to return the specified result)
- incorrect use of nested `if/else` constructs
- incorrect array construction, traversal and element access
- difficulty working with boundary conditions of array and `ArrayList` data structures
- confusion of array and `ArrayList` indices with their elements
- loop boundary errors

A cautionary reminder: this set of questions, as any set of such questions, represents only a subset of the testable curriculum that may be assessed in the future. For example, although Question 2 involved extending the `Critter` class from the *GridWorld* case study, that case study includes a rich set of other classes and concepts upon which future questions may be based.

Scoring Guidelines, Sample Student Responses, and Commentary

The answers presented on the following pages are actual student responses to the free-response questions on the 2009 AP Computer Science A Exam. The students gave permission to have their work reproduced at the time they took the exam. These responses were read and scored by the Table Leaders and Readers assigned to each particular question during the AP Reading in June 2009. The actual scores that these student responses earned, as well as a brief explanation of why, are included.

2009 General Scoring Guidelines

Apply the question-specific rubric first. The question-specific rubric always takes precedence.

Refer to the general error categorization below only for cases not already covered by the question-specific rubric.

Points can only be deducted if the error occurs in a part which has earned credit via the question-specific rubric.

Any error included below is penalized only once in a question, even if it occurs on different parts of that question.

Special handling of minor errors (½ point): If a specific minor error (½ point) is the **only instance, one of two**, or occurs two or more times, then it must be penalized as shown. If a specific minor error occurs exactly once when the same concept is correct two or more times, then it is not penalized (regarded as an oversight).

Nonpenalized Errors	Minor Errors (1/2 point)	Major Errors (1 point)
spelling/case discrepancies*	confused identifier (e.g., <code>len</code> for <code>length</code> or <code>left()</code> for <code>getLeft()</code>)	extraneous code which causes side-effect; e.g., <i>information written to output</i>
local variable not declared if other variables are declared in some part	no local variables declared	use interface or class name instead of variable identifier; e.g., <code>Simulation.step()</code> instead of <code>sim.step()</code>
use keyword as identifier	missing <code>new</code> in constructor call	<code>aMethod(obj)</code> instead of <code>obj.aMethod()</code>
<code>[]</code> vs. <code>()</code> vs. <code><></code>	modifying a constant (<code>final</code>)	use private data or method when not accessible
<code>=</code> instead of <code>==</code> (and vice versa)	use <code>equals</code> or <code>compareTo</code> method on primitives, e.g., <code>int x; ...x.equals(val)</code>	destruction of data structure (e.g., by using root reference to a <code>TreeNode</code> for traversal of the tree)
length/size confusion for array, <code>String</code> , and <code>ArrayList</code> , with or without <code>()</code>	array/collection access confusion (<code>[]</code> <code>get</code>)	use class name in place of <code>super</code> either in constructor or in method call
<code>private</code> qualifier on local variable	assignment dyslexia, e.g., <code>x + 3 = y;</code> for <code>y = x + 3;</code>	<code>void</code> method (or constructor) returns a value
extraneous code with no side-effect; e.g., <i>precondition check</i>	<code>super(method())</code> instead of <code>super.method()</code>	
common mathematical symbols for operators (<code>×</code> <code>÷</code> <code>≤</code> <code>≥</code> <code><</code> <code>></code> <code>≠</code>)	formal parameter syntax (with type) in method call, e.g., <code>a = method(int x)</code>	
missing <code>{ }</code> where indentation clearly conveys intent and <code>{ }</code> used elsewhere	missing <code>{ }</code> where indentation clearly conveys intent and <code>{ }</code> not used elsewhere	
missing <code>;</code> where indentation clearly conveys intent and <code>;</code> used elsewhere	missing <code>;</code> where indentation clearly conveys intent and <code>;</code> not used elsewhere	
default constructor called without parens; e.g., <code>new Fish;</code>	missing <code>public</code> from method header when required	
missing <code>()</code> on parameter-less method call	"false"/"true" or 0/1 for boolean values	
missing <code>()</code> around <code>if/while</code> conditions	"null" for null	
missing <code>public</code> on class or constructor header		
extraneous <code>[]</code> when referencing entire array		
extraneous size in array declaration, e.g., <code>int[size] nums = new int[size];</code>		

*Note: Spelling and case discrepancies for identifiers fall under the "non-penalized" category only if the correction can be **unambiguously** inferred from context; for example, "Queu" instead of "Queue." Note, however, that if a solution declares "Actor actor," then uses "Actor.move()" instead of "actor.move()", the context does not allow assuming the object versus the class.

Question 1—Overview

This question focused on the array data structure, its construction and traversal, the application of basic algorithms, and method invocation for a specified object. Students were provided with the framework of a helper class, `NumberCube`, that represented a conventional six-sided die (a cube with the numbers 1 to 6 on its sides). They were asked to implement two static methods of unspecified classes. In part (a) students were required to implement the `getCubeTosses` method that returns an array of values obtained by invoking the `toss` method of a `NumberCube` object. This could be accomplished by creating an integer array of the specified length, then filling it with values obtained by invoking `toss` on the supplied `NumberCube` object. In part (b) students were required to implement the `getLongestRun` method that identifies and returns the starting index of the longest sequence of two or more consecutively repeated values in an array. This involved traversing a supplied array of integer values to locate such sequences.

Most of the student errors on this question involved incorrect array construction and access, flawed loop-based iteration, and failures to address the question's specifications. A significant number of students were unable to construct an array properly, and there appeared to be confusion of array indices with array elements. There was a proliferation of typical loop boundary errors as well. Solutions included both single and nested loop algorithms, with the nested loop attempts usually implemented incorrectly. Many students also failed to address the question's specifics and did unnecessary work, for example, by generating a random number instead of invoking the required `toss` method, or returning the length of the maximum-length run rather than the required index. Identifying the maximum-length run appeared to be the most difficult task, and solutions often failed when the maximum-length run included the end of the array.

This question appears to have been of average difficulty and was comparable in difficulty to similar questions on previous exams. The mean score was 4.70 out of a possible 9 points. Scores were generally in the 5 to 9 range, but 0 percent of students received scores of 0 or submitted blank papers. Disregarding scores of 0 and blank papers, the mean was 5.88.

Scoring Guidelines for Question 1

Number Cube

Part (a)	getCubeTosses	4 points
+1	constructs array	
+1/2	constructs an array of type <code>int</code> or size <code>numTosses</code>	
+1/2	constructs an array of type <code>int</code> and size <code>numTosses</code>	
+2 1/2	processes tosses	
+1	repeats execution of statements <code>numTosses</code> times	
+1	tosses cube in context of iteration	
+1/2	collects results of tosses	
+1/2	returns array of generated results	
Part (b)	getLongestRun	5 points
+1	iterates over <code>values</code>	
+1/2	accesses element of <code>values</code> in context of iteration	
+1/2	accesses all elements of <code>values</code> , no out-of-bounds access potential	
+1	determines existence of run of consecutive elements	
+1/2	comparison involving an element of <code>values</code>	
+1/2	comparison of consecutive elements of <code>values</code>	
+1	always determines length of at least one run of consecutive elements	
+1	identifies maximum length run based on all runs	
+1	return value	
+1/2	returns starting index of identified maximum length run	
+1/2	returns -1 if no run identified	

Sample Student Responses for Question 1

Student Response 1 (Score: 8)

- (a) Write the method `getCubeTosses` that takes a number cube and a number of tosses as parameters. The method should return an array of the values produced by tossing the number cube the given number of times.

Complete method `getCubeTosses` below.

```
/** Returns an array of the values obtained by tossing a number cube numTosses times.
 * @param cube a NumberCube
 * @param numTosses the number of tosses to be recorded
 *      Precondition: numTosses > 0
 * @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
{
    int[] count = new int[numTosses];
    for (int i=0; i<numTosses; i++)
    {
        count[i] = cube.toss();
    }
    return count;
}
```

Student Response 1 (continued)

Complete method `getLongestRun` below.

```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 * in the array values.
 * @param values an array of integer values representing a series of number cube tosses
 * Precondition: values.length > 0
 * @return the starting index of a run of maximum size;
 *         -1 if there is no run
 */
```

```
public static int getLongestRun(int[] values)
{
    boolean inLong = false;
    int longest = 1, temp = 1;
    int pos = -1, maxPos = -1;
    for (int i = 0; i < values.length - 1; i++)
    {
        if (values[i] == values[i + 1])
        {
            if (inLong)
            {
                temp++;
            }
            else
            {
                pos = i;
                inLong = true;
            }
        }
        else
        {
            inLong = false;
            if (temp > longest)
            {
                longest = temp;
                maxPos = pos;
            }
            temp = 1;
            pos = -1;
        }
    }
    return maxPos;
}
```

Ident Response 1 (continued)

Commentary

Part (a) is a canonical solution that earned all 4 points.

Part (b) is a good solution that earned 4 out of 5 possible points. It iterates over all elements of `values` because the loop test of `i < values.length-1` keeps `values[i+1]` from going out-of-bounds. It also compares consecutive elements of `values`. The variables `temp` and `longest` are one less than the actual run lengths, but they are consistent, so it does not cause a problem.

The check for the maximum-length run is in the `else` clause, so the maximum run isn't identified until `i` advances beyond the current run and `values[i] != values[i+1]`. Therefore, the longest run isn't identified if it occurs at the end of `values`. The variable `pos` is set to `i` (the next position that could begin a run) when `values[i] != values[i+1]`. This position is then assigned to `maxPos` when a new longest run is identified and `maxPos` is subsequently returned. Finally, the solution code returns -1 if there is no run because `maxPos` is initialized to -1 and is not changed when there is no run.

Student Response 2 (Score: 6)

- (a) Write the method `getCubeTosses` that takes a number cube and a number of tosses as parameters. The method should return an array of the values produced by tossing the number cube the given number of times.

Complete method `getCubeTosses` below.

```
/** Returns an array of the values obtained by tossing a number cube numTosses times.
 * @param cube a NumberCube
 * @param numTosses the number of tosses to be recorded
 *      Precondition: numTosses > 0
 * @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
```

```
{
    int[] arr = new int[numTosses];
    for (int i = 0; i < numTosses; i++)
    {
        arr[i] = toss();
    }
    return arr;
}
```

ident Response 2 (continued)

Complete method getLongestRun below.

```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 * in the array values.
 * @param values an array of integer values representing a series of number cube tosses
 *      Precondition: values.length > 0
 * @return the starting index of a run of maximum size;
 *      -1 if there is no run
 */
```

```
public static int getLongestRun(int[] values)
```

```
    int count = 0;
    int count2 = 1;
    for (int i = 0; i < values.length - 1; i++)
    {
        if (values[i] == values[i+1])
        {
            count2++;
            for (int j = i+2; j < values.length; j++)
            {
                if (values[j] == values[i])
                    count2++;
            }
            if (count2 > count)
            {
                count = count2;
                count2 = 0;
            }
        }
    }
```

```
    if (count > 0)
        return count;
    return -1;
```

Student Response 2 (continued)

Commentary

Part (a) is a good solution that earned 3 out of 4 possible points. The `toss` method was not invoked properly on the `cube` parameter, but the remainder of the solution is canonical.

Part (b) earned $2\frac{1}{2}$ points out of 5 possible points. It iterates over all elements of `values` because the outer loop test of `i < values.length-1` keeps `values[i+1]` from going out-of-bounds. It does not correctly find one run length because the inner `for` loop counts all pairs where `values[j] == values[i]`, not just the ones in a single consecutive run. Also, `count2`, the run length counter, is reset prematurely when a new maximum length run is identified. This can result in the longest run being missed. The method returns a count instead of an array index but correctly returns -1 when there are no runs.

This solution scored $5\frac{1}{2}$ points, which was rounded to 6 points.

Ident Response 3 (Score: 3)

Write the method `getCubeTosses` that takes a number cube and a number of tosses as parameters. The method should return an array of the values produced by tossing the number cube the given number of times.

Complete method `getCubeTosses` below.

```
/** Returns an array of the values obtained by tossing a number cube numTosses times.
 * @param cube a NumberCube
 * @param numTosses the number of tosses to be recorded
 *      Precondition: numTosses > 0
 * @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
{
    int[] cubeTosses = new Array;
    for (int i = 0; i < numTosses; i++)
    {
        cubeTosses[i] += cube.toss(i)
    }
    return cubeTosses;
}
```

Student Response 3 (continued)

Complete method `getLongestRun` below.

```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 * in the array values.
 * @param values an array of integer values representing a series of number cube tosses
 *      Precondition: values.length > 0
 * @return the starting index of a run of maximum size;
 *      -1 if there is no run
 */
public static int getLongestRun(int[] values)
{
    int consecutiveNums = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i+1] == values[i])
            consecutiveNums++;
    }
    return consecutiveNums - i;

    if (consecutiveNums == 0)
        return -1;
}
```

Commentary

Part (a) earned 1½ points out of 4 possible points. The declaration of the new array contains the keyword `new` but none of the other “constructs array” elements are present. It does repeat the execution of statements `numTosses` times, but it does not correctly toss the cube or properly collect the results of the attempted tosses. The “returns generated results” ½ point was earned because the `[]` on `return cubeTosses [];` is a non-penalized error under the general scoring guidelines.

Part (b) earned 1½ of the possible 5 points. It does not iterate over all elements of `values` because `values[i+1]` is out-of-bounds when `i` is `values.length-1`. It does compare consecutive elements of `values`. However, the loop counts all equal pairs, not just equal pairs in a single run, because `consecutiveNums` is not reset at the end of each run. So the “length of one run” point was not earned. There is no attempt to determine the maximum length run, so this point and the “returns starting index of identified maximum length run” ½ point were not earned. The statements after the `return consecutiveNums - i;` are dead code, so the “return -1” ½ point was not earned.

Question 2—Overview

This question involved reasoning about the code from the GridWorld case study, emphasizing object-oriented concepts. Students demonstrated their understanding of the case study and its interacting classes by extending the `Critter` class to derive a `StockpileCritter` class with partially modified behavior. This question tested numerous concepts: creating a class, inheriting from an existing class, overriding appropriate methods, and maintaining the overridden methods' postconditions. Students were specifically instructed not to override the `act` method, and they were explicitly cautioned to abide by the postconditions of all methods.

Students were asked to change the behavior of the `StockpileCritter` in the following ways: during each `act`, `StockpileCritter` (1) gathers up all neighboring actors, removes them from the grid, and adds them to its stockpile, and (2) reduces the stockpile by one, if possible, or, if not possible, removes itself from the grid. In all other ways, `StockpileCritter` should behave as a `Critter` would behave. Students were cautioned to abide by the postconditions of all methods and additionally were told that they could not override the `act` method.

The postconditions for `getActors`, `getMoveLocations`, and `selectMoveLocation` do not allow any state changes. The postconditions for `processActors` only allow state changes for this critter and the actors in the parameter `actors`. Additionally, the postconditions for `processActors` state that the location of this critter is unchanged. The `makeMove` method has two postconditions: at the end of `makeMove`, `getLocation` must equal parameter `loc` and only this critter and the actor in location `loc` can change state.

Most students chose to override the `processActors` method, since this method is responsible for the way in which a critter processes its actors. The requirements for this method were to stockpile the actors in some way, removing them from the grid in the process. Most students chose to count the number of actors in the parameter `actors` and then remove each of the actors from the grid, although some students stored the actors in a second array list and then removed one actor from the grid. Additionally, some students chose to decrement the stockpile after gathering the actors.

The `makeMove` method was chosen by the vast majority of students as the second (and last) method to override. Using only `processActors` and `makeMove` to override resulted in a loss of 1 point for violating postconditions by not overriding the required methods to correctly code the `StockpileCritter`. If students chose to call `removeSelfFromGrid()` in `processActors`, they violated the postcondition that states that the location of this critter is unchanged. If they removed this `StockpileCritter` based on the status of the stockpile in `makeMove`, they violated the postcondition `getLocation() == loc`.

The `selectMoveLocation` method was rarely chosen by the students to override. To correctly code the `StockpileCritter` class, the `selectMoveLocation` method had to be overridden. This method is responsible for choosing the next location and then returning it to the `act` method. The `act` method then passes that information to `makeMove` by way of the parameter `loc`. If `loc == null`, then the `makeMove` method will remove the `StockpileCritter` from the grid and at the conclusion of the method, the postconditions will be satisfied.

The most common student errors were ignoring required solution specifications through either overriding the `act` method or violating well-specified postconditions. For example, a common erroneous approach involved decrementing the stockpile in the `makeMove` method, which violated one of that method's postconditions. There was also apparent confusion over the responsibilities of the various case-study methods. Other common errors included declaration of local variables instead of instance variables, incomplete overriding of methods, and calculations that were off by one when determining stockpile depletion.

Very few solutions were awarded 9 points. Nearly every solution violated a postcondition of `makeMove` or `processActors`. Once that point was not awarded, the rest of the students' code was scored to see if they created a `StockpileCritter` that would behave as outlined in the problem.

This question had the fewest scores of 0 and blank papers by far (less than 13 percent combined), which is extraordinary performance, especially for a case-study question. The mean score was 4.64 out of a possible 9 points. Students generally did well, though very few solutions earned a perfect score of 9 because nearly every solution violated a necessary postcondition. Disregarding scores of 0 and blank papers, the mean was 5.32.

Scoring Guidelines for Question 2

Stockpile Critter (GridWorld)

- +1 class header
 - +1/2 properly formed class header for `StockpileCritter`
 - +1/2 extends `Critter` class
- +1 1/2 stockpile state
 - +1/2 declares instance variable capable of maintaining state
 - +1/2 private visibility
 - +1/2 initialization of state appropriate to usage of variable
- +1 overrides methods and maintains all necessary post-conditions
(No points awarded if overrides `act` method)
- +1 `processActors` overridden (No points awarded if overrides `act` method)
- +1 stockpile state maintenance
 - +1/2 accumulates based on number of actors passed to `processActors`
 - +1/2 decrements appropriately each `act`
- +1 1/2 removes neighboring actors from grid
 - +1/2 removes **at least one** neighboring actor from grid
 - +1 removes **all** neighboring actors from grid
- +2 self-removal
 - +1/2 checks status of stockpile by using state variable in a relational expression
 - +1/2 ever removes self from grid
 - +1 removes self from grid when and only when stockpile state indicates empty

Sample Student Responses for Question 2

Student Response 1 (Score: 8)

```
public class StockpileCritic extends Critter
{
    private int energy;
    public StockpileCritic()
    {
        energy = 0;
    }
    public void processActors (ArrayList<Actor> actors)
    {
        for (Actor a : actors)
        {
            energy ++;
            a.removeSelfFromGrid();
        }
    }
    public void makeMove (Location loc)
    {
        if (energy == 0)
            removeSelfFromGrid();
        else
            super.makeMove(loc);
        energy --;
    }
}
```

Commentary

The student correctly declares a class `StockpileCritic` that extends `Critic`. There is a private instance variable, `energy`, that is correctly initialized. The student fails to override the method `selectMoveLocation` but does override the method `processActors` correctly. Within `processActors`, the student correctly adds the number of neighboring actors to the stockpile and correctly removes all these neighbors from the grid. When overriding `makeMove`, the student checks the status of the stockpile and correctly calls `removeSelfFromGrid` in all appropriate cases. There is also a correct decrement to the stockpile.

Student Response 2 (Score: 7)

```
public class StockpileGritter extends Gritter {
    int stockPile = 0;
    public void processActors (ArrayList<Actor> actors) {
        for (Actor a: actors)
        {
            if (!(a instanceof Rock) && !(a instanceof Gritter))
                a.removeSelfFromGrid();
            stockPile++;
        }
    }

    public void makeMove (Location loc)
    {
        if (loc == null || stockPile <= 0)
            removeSelfFromGrid();
        else
            stockPile--;
        super.makeMove(loc);
    }
}
```

dent Response 2 (continued)

ommentary

The student correctly declares a class `StockpileCritic` that extends `Critic`. There is an instance variable, `stockpile`, that is correctly initialized; however, this variable does not have `private` scope. The student fails to override the method `selectMoveLocation` but does override the method `processActors` correctly. Within `processActors`, the student correctly adds the number of neighboring actors to the stockpile and correctly removes one of these neighbors from the grid. However, because of the conditional guard, all of the neighbors might not be removed from the grid. When overriding `makeMove`, the student checks the status of the stockpile and correctly calls `removeSelfFromGrid` in all appropriate cases. There is also a correct decrement to the stockpile. This solution scored $6\frac{1}{2}$ points, which was rounded to 7 points.

Student Response 3 (Score: 3)

```
public class StockpileCritic extends Critter
{ public int stockpile=0;
  public StockpileCritic ( )
  {
    if ( getOccupiedAdjacentLocations() = null )
    { return removeSelfFromGrid; }
    else
    { getOccupiedAdjacentLocations().removeSelfFromGrid();
      stockpile++; }
    if ( stockpile == 0 )
    { removeSelfFromGrid; }
  }
}
```

Commentary

The student correctly declares a class `StockpileCritic` that extends `Critic`. There is an instance variable `stockpile`, that is correctly initialized, but it does not have `private` scope. The student does not override either of the methods `processActors` or `selectMoveLocation`. There is no accumulation to the state variable based on the parameter passed to `processActors`, and the student never correctly removes any neighboring actors from the grid. The student checks the status of the stockpile and correctly calls `removeSelfFromGrid` in one case. However, there is never a decrement to the stockpile variable. Because there is no decrement, there is no way to ensure that `removeSelfFromGrid` is called correctly in all cases.

Question 3—Overview

This question focused on array traversal, abstraction, and algorithms for accumulation and finding a minimum. Students were provided with the framework of the `BatteryCharger` class that included a private array instance variable with exactly 24 `int` elements, and they were asked to implement two instance methods. The first method, `getChargingCost`, required calculation of a total charging cost given a start time (`startHour`) and a number of hours (`chargeTime`). This could be accomplished by accessing elements of the instance array, beginning with the element at index `startHour`, and traversing in a circular manner (for example, by using the modulus operator), accumulating the values from the array, and returning the sum. The second method, `getChargeStartTime`, required students to return the start time that would allow the battery to be charged at minimal cost. This was best accomplished by invoking the `getChargingCost` method from part (a) for each of the 24 potential start times, comparing the results to determine which achieve the minimum charging cost, and returning that start time.

Although most students appeared to understand the concept of accumulating, many had difficulty with properly understanding the accumulation process. Most students did not take advantage of the modulus operator (`%`), instead using alternative error-prone approaches. There was also some apparent confusion with 0-based array indexing. A significant number of solutions to part (b) did not use the most straightforward approach of invoking `getChargingCost`. Other common errors included improperly initialized variables, incorrect approaches to finding the minimum, and returning the minimum charging cost rather than the start hour of that minimum charging cost.

Students performed best on this question out of all the questions on the exam, with more than 40 percent earning scores of 7, 8, or 9. The mean score was 4.75 out of a possible 9 points, even though 25 percent earned scores of 0 or submitted blank papers. Disregarding scores of 0 and blank papers, the mean was a very strong 6.35. A somewhat bimodal distribution indicates that this question was straightforward for those who were well prepared, yet quite difficult for those who were not.

Scoring Guidelines for Question 3

Battery Charger

Part (a)	getChargingCost	5 points
	<ul style="list-style-type: none"> +1 1/2 accesses array elements <ul style="list-style-type: none"> +1/2 accesses any element of <code>rateTable</code> +1/2 accesses an element of <code>rateTable</code> using an index derived from <code>startHour</code> +1/2 accesses multiple elements of <code>rateTable</code> with no out-of-bounds access potential +2 1/2 accumulates values <ul style="list-style-type: none"> +1/2 declares and initializes an accumulator +1/2 accumulates values from elements of <code>rateTable</code> +1/2 selects values from <code>rateTable</code> using an index derived from <code>startHour</code> and <code>chargeTime</code> +1 determines correct sum of values from <code>rateTable</code> based on <code>startHour</code> and <code>chargeTime</code> +1 value returned <ul style="list-style-type: none"> +1/2 returns any non-constant (derived) value +1/2 returns accumulated value 	
Part (b)	getChargeStartTime	4 points
	<ul style="list-style-type: none"> +1/2 invokes <code>getChargingCost</code> or replicates functionality with no errors +1 determines charging cost <ul style="list-style-type: none"> +1/2 considers all potential start times; must include at least 0 ... 23 +1/2 determines charging cost for potential start times <p><i>Note: No penalty here for parameter passed to <code>getChargingCost</code> that violates its preconditions (e.g., 24)</i></p> +1 compares charging costs for two different start times +1 determines minimum charging cost based on potential start times <p><i>Note: Penalty here for using result of call to <code>getChargingCost</code> that violates its preconditions (e.g., 24)</i></p> +1/2 returns start time for minimum charging cost 	

Sample Student Responses for Question 3

Student Response 1 (Score: 9)

- Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```
/** Determines the total cost to charge the battery starting at the beginning of startHour.
 * @param startHour the hour at which the charge period begins
 *      Precondition: 0 ≤ startHour ≤ 23
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return the total cost to charge the battery
 */
```

```
private int getChargingCost(int startHour, int chargeTime)
```

```
{
    int cost = 0;
    for (int i = 0; i < chargeTime; i++)
    {
        int index = (startHour + i) % 24;
        cost += rateTable[index];
    }
    return cost;
}
```

Student Response 1 (continued)

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return an optimal start time, with  $0 \leq \text{returned value} \leq 23$ 
 */
public int getChargeStartTime(int chargeTime)
{
    int startTime = 0;
    int startCost = getChargingCost(0, chargeTime);
    for (int i = 1; i < 24; i++)
    {
        int currentCost = getChargingCost(i, chargeTime);
        if (currentCost < startCost)
        {
            startTime = i;
            startCost = currentCost;
        }
    }
    return startTime;
}
```

Commentary

In part (a) the student correctly accesses an element from `rateTable` using an index derived from `startHour`. The loop body demonstrates accessing multiple elements of `rateTable` with no out-of-bounds potential. The student declares and initializes an accumulator and accumulates values from `rateTable` using an index derived from `startHour` and `chargeTime`. A correct sum of the values from `rateTable` is determined and this accumulated value is returned. Part (a) earned all 5 points.

In part (b) the student correctly invokes `getChargingCost`. For each potential start time (0 through 23), the charging cost is determined. The student compares two charging costs and correctly determines the minimum charging cost. The student correctly initializes and returns the start time for the minimum charging cost. Part (b) earned all 4 points.

Student Response 2 (Score: 6)

- (a) Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```

/** Determines the total cost to charge the battery starting at the beginning of startHour.
 * @param startHour the hour at which the charge period begins
 *      Precondition: 0 ≤ startHour ≤ 23
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return the total cost to charge the battery
 */
private int getChargingCost(int startHour, int chargeTime)
{
    int total = 0;
    for (int i = startHour; i < startHour + chargeTime; i++)
    {
        if (i ≤ 23)
            total += rateTable[i];
        else
            total += rateTable[i - 24];
    }
    return total;
}

```

Student Response 2 (continued)

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime)
{
    int position;
    for (int i = 1; int i <= 23; i++)
    {
        if (getChargingCost(i, chargeTime) < getChargingCost(i-1, chargeTime))
            position = i;
        else
            position = i-1;
    }
    return position;
}
```

Commentary

In part (a) the student correctly accesses an element from `rateTable` using an index derived from `startHour` and so earned the first two ½ points. Although the student attempts to handle the wrap-around case (`rateTable[23]` to `rateTable[0]`) by subtracting 23, when `startHour + chargeTime` is large (greater than 46), an index out-of-bounds error will occur. The student did not earn this ½ point. The student declares and initializes an accumulator and accumulates values from `rateTable` using an index derived from `startHour` and `chargeTime` and so earned the three ½ points. Since the correct sum will not be determined if a wrap-around is necessary, the student did not earn 1 point. A derived accumulated value is returned and so the student earned the last two ½ points. Part (a) earned 3 ½ points.

In part (b) the student correctly invokes `getChargingCost` and so earned the first ½ point. The student considers all times 0 through 23, and for each potential start time that is considered, the charging cost is determined. The student earned the next two ½ points. The student compares two charging costs and so earned 1 point. The minimum charging cost is not determined so the student did not earn 1 point. Since the start time for the minimum charging cost is not returned, the last ½ point was not earned. Part (b) earned 2 ½ points.

Ident Response 3 (Score: 3)

- Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```
/** Determines the total cost to charge the battery starting at the beginning of startHour.
 * @param startHour the hour at which the charge period begins
 *      Precondition: 0 ≤ startHour ≤ 23
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return the total cost to charge the battery
 */
```

```
private int getChargingCost(int startHour, int chargeTime) {
    int start = startHour;
    start = rateTable[start];
    int cTime = chargeTime;
    cTime = rateTable[cTime];
    charge = start - 24;
    return // values of rateTable[] inclusive
           // from cTime to charge, sum of all
           // values is charging cost.
}
```

Student Response 3 (continued)

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 * @param chargeTime the number of hours the battery needs to be charged
 *      Precondition: chargeTime > 0
 * @return an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime) {
    int cTime = chargeTime;
    int big;    int temp = 0;
    for (int k = 0; k < 23; k++) {
        temp = getChargingCost(k, cTime);
        if (temp < big) {
            temp = big;
        }
    }
    return temp;
}
```

Commentary

In part (a) the student accesses an element from `rateTable` using an index derived from `startHour` and so earned the first two ½ points. The access to `rateTable[cTime]` will cause an out-of-bounds error when `chargeTime` is greater than 23 so the student did not earn the next ½ point. Since there is no accumulator, the 2½ “accumulates values” points were not earned. Since no value is returned, the student did not earn either of the last two ½ points. Part (a) earned 1 point.

In part (b) the student correctly invokes `getChargingCost` and so earned the first ½ point. The student only considers start times 0 through 22 and did not earn the second ½ point. The student does determine a charging cost for each potential start time considered and so earned the next ½ point. The student does compare charging costs of two different start times and earned 1 point. Since the local variable `big` is not initialized, the minimum charge time will not be correctly determined so the student did not earn this 1 point. The start time for the minimum charging cost is not returned so the last ½ point was not earned. Part (b) earned 2 points.

Question 4—Overview

This question focused on object-oriented programming, algorithm development, and list processing. It required significant problem analysis and algorithm design. Students were expected to analyze the problem, design algorithms, and then implement them using the well-specified public interface of a `NumberTile` class. The question involved writing two related methods for a given `TileGame` class. The method `getIndexForFit` determines where a given `NumberTile` object can be added into the `ArrayList` instance variable `board`. The method `insertTile` modifies the state of a `TileGame` object by adding a given `NumberTile` object to `board`. In designing their solutions, students were required to consider the full set of potential cases and to work within the constraints of the public methods provided for `NumberTile`. This question assessed the ability to decompose a stated problem into computational constituents, use algorithmic thinking, rely on abstraction, and demonstrate facility with an `ArrayList` object.

Many students seemed unprepared to address this level of problem-solving and algorithm development in a free-response question. Incorrect program logic was pervasive, often evidenced by incorrect use of nested `if-else` constructs and behaviors that resulted in changes of an object's state that were prohibited by the specification. Incorrect use of object-oriented method invocation was common. There was also substantial confusion about working with lists, specifically accessing elements of an `ArrayList` and manipulating `ArrayList` objects.

This question had the highest number of scores of 0 and blank papers (almost 28 percent). This may indicate students' relative discomfort with more complex problem descriptions and greater emphasis on problem analysis, or it may be because the question required algorithm design rather than simply writing code for a given algorithm. (It was also the last question on the exam, so students could have been tired.) Otherwise, scores were evenly distributed, with a mean score of out of a possible 9 points. Disregarding scores of 0 and blank papers, the mean was 5.54.

Scoring Guidelines for Question 4

Tile Game

Part (a)	getIndexForFit	6 points
+1	empty board	
+1/2	checks for zero-sized board	
+1/2	returns 0 if empty board detected	
+1	accesses tiles from board	
+1/2	accesses any tile from board	
+1/2	accesses all tiles of board (as appropriate) with no out-of-bounds access potential	
+1	uses tile values	
+1/2	accesses left or right value of any tile	
+1/2	compares left (right) value of parameter with right (left) value of any tile from board	
+2	determines tile fit	
+1/2	only right value of parameter compared with left value of initial tile of board	
+1/2	only left value of parameter compared with right value of final tile of board	
+1	compares appropriate values of parameter and interior tiles of board	
+2	resultt	
+1/2	returns located index if tile fits in board	
+1/2	returns -1 if tile does not fit in board	
Part (b)	insertTile	3 points

- +1/2 invokes `getIndexForFit` or replicates functionality with no errors
- +1 1/2 tile orientation
 - +1/2 invokes `rotate` on parameter
 - +1/2 performs all necessary rotations
 - +1/2 invokes `getIndexForFit` for each necessary orientation
- +1/2 adds tile correctly and only if `getIndexForFit` returns value other than -1
- +1/2 returns `true` if `getIndexForFit` returns value other than -1; `false` otherwise

Sample Student Responses for Question 4
Student Response 1 (Score: 9)

Complete method `getIndexForFit` below.

```
/** Determines where to insert tile, in its current orientation, into game board
 * @param tile the tile to be placed on the game board
 * @return the position of tile where tile is to be inserted:
 *         0 if the board is empty;
 *        -1 if tile does not fit in front, at end, or between any existing tiles;
 *         otherwise,  $0 \leq \text{position returned} \leq \text{board.size()}$ 
 */
private int getIndexForFit(NumberTile tile)
{
    if (board.size() == 0)
        return 0;
    if (tile.getHeight() == board.get(0).getHeight())
        return 0;
    if (tile.getHeight() == board.get(board.size() - 1).getHeight())
        return board.size();

    if (board.size() > 1)
    {
        for (int i = 1; i < board.size(); i++)
        {
            if (board.get(i).getHeight() == tile.getHeight())
            {
                if (board.get(i - 1).getHeight() == tile.getHeight())
                    return i;
            }
        }
    }

    return -1;
}
```

Student Response 1 (continued)

- (b) Write the `TileGame` method `insertTile` that attempts to insert the given tile on the game board. The method returns `true` if the tile is inserted successfully and `false` only if the tile cannot be placed on the board in any orientation.

Assume that `getIndexForFit` works as specified, regardless of what you wrote in part (a).

Complete method `insertTile` below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 * If there are no tiles on the game board, the tile is placed at position 0.
 * The tile should be placed at most 1 time.
 * Precondition: board is not null
 * @param tile the tile to be placed on the game board
 * @return true if tile is placed successfully; false otherwise
 * Postcondition: the orientations of the other tiles on the board are not changed
 * Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
```

```
public boolean insertTile(NumberTile tile)
```

```
{
```

```
    int insertN = getIndexForFit(tile);
```

```
    if (insertN == -1)
```

```
    {
```

```
        tile.rotate();
```

```
        insertN = getIndexForFit(tile);
```

```
    }
```

```
    if (insertN == -1)
```

```
    {
```

```
        tile.rotate();
```

```
        insertN = getIndexForFit(tile);
```

```
    }
```

```
    if (insertN == -1)
```

```
    {
```

```
        tile.rotate();
```

```
        insertN = getIndexForFit(tile);
```

```
    }
```

```
    if (insertN == -1)
        return false;
```

```
    board.add(insertN, tile);
```

```
    return true;
```

```
}
```

Student Response 1 (continued)

Commentary

Part (a) earned all 6 points. The student determines if the board is empty and then proceeds to systematically check to see if the tile will fit into the board in its current orientation. Points were awarded for accessing the board and tile properly as well as checking to see if the tile can be placed at the beginning, middle, or end of the list. The expected return is the location of the tile to be placed in the board, 0 if the board is empty, or -1 if the tile cannot be placed.

Part (b) earned all 3 points. A proper call to the method `getIndexForFit(tile)` is made in conjunction with calls to `tile.rotate()`. This student chooses to perform the tests separately using `if` statements. If a placement for the tile is found, the tile is added to the board and the method will return `true`; otherwise the method will return `false`.

Student Response 2 (Score: 7)

Complete method `getIndexForFit` below.

```
/** Determines where to insert tile, in its current orientation, into game board
 * @param tile the tile to be placed on the game board
 * @return the position of tile where tile is to be inserted:
 *         0 if the board is empty;
 *        -1 if tile does not fit in front, at end, or between any existing tiles;
 *         otherwise,  $0 \leq \text{position returned} \leq \text{board.size}()$ 
 */
private int getIndexForFit(NumberTile tile)
{
    int r = tile.getRight();
    int left = tile.getLeft();
    if (r != left)
    {
        if (board.get(0).getLeft() == r)
        {
            return 0;
        }
        else if (board.get(board.size() - 1).getRight() == left)
        {
            return board.size();
        }
        else
        {
            return -1;
        }
    }
    else
    {
        for (int k = 0; k < board.size() - 2; k++)
        {
            if (board.get(k).getRight() == left)
            {
                return k + 1;
            }
        }
        return -1;
    }
}
```

Student Response 2 (continued)

- (b) Write the `TileGame` method `insertTile` that attempts to insert the given tile on the game board. The method returns `true` if the tile is inserted successfully and `false` only if the tile cannot be placed on the board in any orientation.

Assume that `getIndexForFit` works as specified, regardless of what you wrote in part (a).

Complete method `insertTile` below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 * If there are no tiles on the game board, the tile is placed at position 0.
 * The tile should be placed at most 1 time.
 * Precondition: board is not null
 * @param tile the tile to be placed on the game board
 * @return true if tile is placed successfully; false otherwise
 * Postcondition: the orientations of the other tiles on the board are not changed
 * Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
public boolean insertTile(NumberTile tile)
{
    for(int k = 0; k < 4 && tile.getIndexForFit() < 0; k++)
    {
        tile.rotate();
        if(k == 3 && tile.getIndexForFit() < 0)
            return false;
    }
    board.add(tile.getIndexForFit(), tile);
    return true;
}
```

Commentary

Part (a) lost the first two $\frac{1}{2}$ points for not checking to see if the board is empty. This student takes a different approach from the canonical solution. The student determines if the left and right sides of the tile are equal and then proceeds to compare the right side of the board tile to the inserted tile. The student earned the $\frac{1}{2}$ point for properly obtaining a tile from the board but lost the next $\frac{1}{2}$ point for not searching the entire board for possible tile placement as indicated by the upper loop boundary of `(board.size() - 2)`. The student earned the remaining points for this section by making the proper comparisons of the tile in its current orientation. Checks are made to see if the tile can be placed at the beginning, middle, or end of the list. The expected return is the index of the tile to be placed in the board, 0 if the board is empty, or -1 if the tile cannot be placed in the board. Part (a) earned $4\frac{1}{2}$ points.

Part (b) earned $2\frac{1}{2}$ points out of 3 possible points. The student lost one $\frac{1}{2}$ point due to an improper call to the method `getIndexForFit(tile)`.

Student Response 3 (Score: 2)

Complete method getIndexForFit below.

```
/** Determines where to insert tile, in its current orientation, into game board
 * @param tile the tile to be placed on the game board
 * @return the position of tile where tile is to be inserted:
 *         0 if the board is empty;
 *        -1 if tile does not fit in front, at end, or between any existing tiles;
 *        otherwise, 0 ≤ position returned ≤ board.size()
 */
private int getIndexForFit(NumberTile tile)
{
    if (board.size() == 0)
        return 0;
    for (int i = 0; i < board.size(); i++)
    {
        if (tile.right() == board.get(i).left() &&
            tile.left() == board.get(i+1).right())
            return i+1;
        else if (tile.right == board.get(board.size-1).left)
            return board.size-1;
        else if (tile.left == board.get(i).right)
            return 0;
        else if (tile != board.get(i))
            return -1;
    }
}
```

Student Response 3 (continued)

- (b) Write the `TileGame` method `insertTile` that attempts to insert the given tile on the game board. The method returns `true` if the tile is inserted successfully and `false` only if the tile cannot be placed on the board in any orientation.

Assume that `getIndexForFit` works as specified, regardless of what you wrote in part (a).

Complete method `insertTile` below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 * If there are no tiles on the game board, the tile is placed at position 0.
 * The tile should be placed at most 1 time.
 * Precondition: board is not null
 * @param tile the tile to be placed on the game board
 * @return true if tile is placed successfully; false otherwise
 * Postcondition: the orientations of the other tiles on the board are not changed
 * Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
```

```
public boolean insertTile(NumberTile tile)
```

```
{
    if (board != null)
    {
        int result = getIndexForFit();
        if (result == 0)
            return false;
        else if (result > 0)
        {
            return true;
            board.add(result, tile);
        }
        else if (result < 0)
            return false;
    }
}
```

Commentary

Part (a) earned the first two $\frac{1}{2}$ points for determining if the board is empty. The student earned two additional $\frac{1}{2}$ points by accessing the board properly and attempting to compare sides of the tile to the interior tiles of the board. Points were not awarded for the checks of the end tiles due to reversed logic. Part (a) earned 2 points.

Part (b) earned no points. The student has an improper call to `getIndexForFit(tile)` and the return of `true / false` is incorrect.