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.

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.

(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)
```

(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**

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)
```

2009 A Question 1: Number Cube — Assessment Rubric

Part A: `getCubeTosses` 4 pts

+1 constructs array
- **+½** constructs an array of type `int` **or** size `numTosses`
- **+½** constructs an array of type `int` **and** size `numTosses`

+2½ processes tosses
- **+1** repeats execution of statements `numTosses` times
- **+1** tosses `cube` in context of iteration
- **+½** collects results of tosses

+½ returns array of generated results

Part B: `getLongestRun` 5 pts

+1 iterates over `values`
- **+½** accesses element of `values` in context of iteration
- **+½** accesses all elements of `values`, no out of bounds access potential

+1 determines existence of run of consecutive elements
- **+½** comparison involving an element of `values`
- **+½** comparison of consecutive elements of `values`

+1 always determines length of at least one run of consecutive elements

+1 identifies maximum length run based on all runs

+1 return value
- **+½** returns starting index of identified maximum length run
- **+½** returns `-1` if no run identified

2009 A Question 1: Number Cube — Canonical Solution

PART A:

```
/** 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 int[numTosses];
    for (int i = 0; i < numTosses; i++) {
        cubeTosses[i] = cube.toss();
    }
    return cubeTosses;
}
```

PART B:

```
/** 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 currentLen = 0;
  int maxLen=0;
  int maxStart=-1;
  for (int i = 0; i < values.length-1; i++) {
    if (values[i]==values[i+1]) {
      currentLen++;
      if (currentLen > maxLen) {
        maxLen = currentLen;
        maxStart = i - currentLen + 1;
      }
    } else {
      currentLen = 0;
    }
  }
  return maxStart;
}
```

2009 A Question 1: Number Cube — Canonical Solution

Alternative solution:

```
public static int getLongestRun(int[] values){
    int maxStart=-1;
    int maxLen=-1;
    int currentLen = 0;
    int currVal = -1;
    int currStart = 0;
    for (int i = 0; i < values.length; i++) {
        if (values[i]==currVal) currentLen++;
        else {
            if (currentLen > maxLen) {
                maxLen = currentLen;
                maxStart = currStart;
            }
            currStart = i;
            currentLen = 1;
            currVal=values[i];
        }
    }
    if (currentLen > maxLen) {
        maxLen = currentLen;
        maxStart = currStart;
    }
    if (maxLen == 1) return -1;
    else return maxStart;
}
```

Alternative solution:

```
public static int getLongestRun(int[] values) {
    int maxLen  = 0;
    int currLen = 0;
    int index = -1;
    int currVal = -1;
    for (int i = values.length - 1; i >= 0; i--) {
        if (values[i] == currVal) currLen++;
        else {
            if (maxLen < currLen) {
                maxLen = currLen;
                index = i+1;
            }
            currVal = values[i];
            currLen = 1;
        }
    }
    if (maxLen < currLen) {
        maxLen = currLen;
        index = 0;
    }
    if (maxLen == 1) return -1;
    return index;
}
```

2009 A Question 1 (Number Cube) Folder # Reader #

Part A (4 pts) getCubeTosses						Part B (5 pts) getLongestRun										
construct arr		processes tosses				iterates		run exists				return				
or	and	numTosses	cube.toss()	coll	ret	acc one	acc all	comp one	comp cons	one run length	max run length	index	-1	**Usage Max -3**	**Booklet Number**	**Score**
½	½	**1**	**1**	½	½	½	½	½	½	**1**	**1**	½	½	part/reason		

2009 A Question 1 (Number Cube) Folder # Reader # A1 - BENCHMARKS

Part A (4 pts) getCubeTosses						Part B (5 pts) getLongestRun										
construct arr		processes tosses				iterates		run exists				return				
or ½	and ½	numTosses 1	cube.toss() 1	coll ½	ret ½	acc one ½	acc all ½	comp one ½	comp cons ½	one run length 1	max run length 1	index ½	-1 ½	Usage Max -3 part/reason	Booklet Number	Score
✓	✓	1	1	✓	✓	✓	✓	✓	✓	1	1	✓	✓		A1a	9
✓	✓	1	1	✓	✓	✓	✓	✓	✓	1	—	✓	✓		A1b	8
✓	✓	1	1	✓	✓	✓	—	✓	✓	1	—	—	✓		A1c	7
—	—	1	1	✓	✓	✓	—	✓	✓	1	—	—	✓		A1d	6
✓	—	1	—	✓	✓	✓	—	✓	✓	1	—	—	—		A1e	5
✓	✓	—	1	✓	✓	—	—	✓	✓	—	—	—	—		A1f	4
—	—	1	—	—	✓	✓	—	✓	✓	—	—	—	—		A1g	3
—	—	—	—	—	—	✓	✓	✓	✓	—	—	—	—		A1h	2
—	—	—	—	—	—	✓	—	✓	—	—	—	—	—		A1i	1

Wed Thu Fri Sat Sun Mon Tue
am pm

6/7/2009 10:47

AP® SUMMER INSTITUTE SCORING NOTES
2009 AP Computer Science A

Question 1

Sample Identifier: A1a
Score: 9

- Part (a): The array construction has both `new` and `int[numTosses]` .
- Part (a): It iterates `numTosses` times by using `arr.length` .
- Part (b): The iteration over `values` begins at `1` (`runIndex` is initialized to 0 for the first element).
- Part (b): `values[i] == values[runIndex]` compares consecutive elements because `i` and `runIndex` are initially `1` and `0` .
- Part (b): The check for "maximum length run" immediately follows `runLength++;` so this check is always executed when a new longer run is processed.
- Part (b): It returns `-1` if there is no run because `maxRunIndex` is initialized to `-1` and is not changed when there is no run.

Sample Identifier: A1b
Score: 8

- Part (a): This is a canonical solution.
- Part (b): 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.
- Part (b): `temp` and `longest` are one less than the actual run lengths, but they are consistent so it does not cause a problem.
- Part (b): The check for 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 found if it occurs at the end of `values` .
- Part (b): `pos` is set to `i` (the next position that could begin a run) when `values[] != values[i+1]`. This position is then assigned to `maxPos` when a new longest run is identified and is subsequently returned.
- Part (b): It returns `-1` if there is no run because `maxPos` is initialized to `-1` and is not changed when there is no run.

Sample Identifier: A1c
Score: 7

- Part (a): `new int(numTosses)` received full credit because `[]` vs. `()` is an un-penalized error.
- Part (b): It does not access all elements of `values` because the `i < values.length` loop test allows `values[i+1]` to be out of bounds.
- Part (b): It calculates one run length correctly because `run` is originally initialized, properly incremented, and reset after a run.
- Part (b): The check for 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 found if it occurs at the end of `values` .
- Part (b): The second `for` loop is used to locate the starting index of the maximum length run. This loop also fails to find the maximum run length because `run` isn't re-initialized before the loop or in the loop at the end of a run.

- Part (b): The value `j-run` is used to calculate the starting index of the maximum length run. This would be incorrect even if `run` was initialized and re-initialized to `0` because it `j-run` would be one less than the correct value.
- Part (b): It returns `-1` if there is no run because `startRun` is initialized to `-1` and is not changed when there is no run.

Sample Identifier: A1d
Score: 6

- Part (a): "constructs array" was not earned because `new` is missing.
- Part (a): "collects results of tosses" was earned because the `values` declaration attempt indicates that `values` is of type `int[]` .
- Part (b): It does not access all elements of `values` because the `a < values.length` loop test allows `values[a+1]` to be out of bounds.
- Part (b): It does not correctly determine the length of the maximum run because the inner `while` loop goes out of bounds when there is a run at the end of `values` .
- Part (b): It returns a run length, not a starting index.
- Part (b): The test for which value to return (`run == 1`) is incorrect. However, since this is considered part of "returns starting index" ½ point, the "return `-1`" ½ point was earned.

Sample Identifier: A1e
Score: 5

- Part (a): `new [numTosses]` earned the first ½ point of "constructs array".
- Part (a): The cube is not tossed.
- Part (b): It does not access all elements of `values` because `values[j+1]` is out of bounds.
- Part (b): The length of a run is correctly accumulated in `r`.
- Part (b): The "determine the maximum length run" point was not earned because the `while` loop goes out of bounds when there is a run at the end of `values`. Also `rounds` is declared in the `if` condition and is not initialized.
- Part (b): It returns a run length, not a starting index.
- Part (b): There is no attempt to "return `-1`".

Sample Identifier: A1f
Score: 4

- Part (a): It "processes tosses" `numTosses+1` times.
- Part (b): Instead of using `values`, this solution creates and uses a new array named `results`. So the "iterates over `values`" point was not earned, but the solution is eligible for other points.
- Part (b): There was no attempt to "determine the length of a run", or to "determine the maximum length run", so these points and the "returns starting index of identified maximum length run" ½ point were not earned.

AP® SUMMER INSTITUTE SCORING NOTES
2009 AP Computer Science A

Sample Identifier: A1g
Score: 3

- Part (a): `new Array` contains `new`, but none of the other "constructs array" elements are present.
- Part (a): It does not correctly toss `cube` or collect the results of the attempted toss.
- Part (a): The "returns generated results" ½ point is earned because the `[]` on `return cubeTosses[];` is a non-penalized error.
- Part (b): `values[i+1]` is out of bounds.
- Part (b): The loop counts all equal pairs, not just equal consecutive pairs because `consecutiveNums` is not reset at the end of each run. So the "length of one run" point was not earned.
- Part (b): 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.
- Part (b): The statements after the `return consecutiveNums – i;` are dead code, so the "return `-1`" ½ point was not earned.

Sample Identifier: A1h
Score: 2

- Part (a): No declaration or creation of an array or collection is attempted, so "constructs array", "collects results", and "returns generated results" are not earned.
- Part (a): The "processes tosses" loop iterates `numTosses+1` times.
- Part (a): There is no `cube.toss()`
- Part (b): The loop bounds are appropriate for `values == values[x-1]`, so the "iterates over `values`" and "determines existence of a run" points are earned.
- Part (b): `return longRun;` is inside the loop, and `longRun` is not reset at the end of the run, so "determines length of one run" was not earned.
- Part (b): 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.
- Part (b): The `return -1` is not based on the non-existence of a run, so it was not earned.

Sample Identifier: A1i
Score: 1

- Part (a): `new arrayList` contains `new`, but none of the other "constructs array" elements are present.
- Part (a): The "processes tosses" loop does not initialize `i` .
- Part (a): It returns `values[i]`, not `values` .
- Part (b): The "iterates over `values`" loop does not access the last element of `values`.
- Part (b): Consecutive elements of `values` are not compared.
- Part (b): There is no attempt to "determine the existence of a run" or to "determine the maximum length run", so these points and the "returns starting index of identified maximum length run" ½ point were not earned.
- Part (b): The `return -1` is not based on the non-existence of a run, so it was not earned.

A1a

(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 < arr.length; i++)
        arr[i] = cube.toss();
    return arr;
}
```

Part (b) begins on page 6.

A/a

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 maxRunIndex = -1;
    int maxRunLength = 1;
    int runIndex = 0, runLength = 1;
    for (int i = 1; i < values.length; i++) {
        if (values[i] == values[runIndex]) {
            runLength ++;
            if (runLength > maxRunLength) {
                maxRunLength = runLength;
                maxRunIndex = runIndex;
            }
        }
        else {
            runIndex = i;
            runLength = 1;
        }
    }
    return maxRunIndex;
}
```

GO ON TO THE NEXT PAGE.

A16

(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;
}
```

Part (b) begins on page 6.

A16

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;
    }
}
```

GO ON TO THE NEXT PAGE.

AIc

(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 numVals[ ] = new int(numTosses);
    for (int i=0; i < numTosses; i++)
        numVals[i] = cube.toss( );
    return numVals;
}
```

Part (b) begins on page 6.

A1c

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 startRun = -1;
    int run = 0;
    int maxRun = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] == values[i+1])
            run ++;
        else
        {
            if (run > maxRun)
                maxRun = run;
            run = 0;
        }
    }

    for (int j = 0; j < values.length; j++)
    {
        if (values[j] == values[j+1])
            run ++;
        if ((run == maxRun) && (run > 1))
            startRun = j - run;
    }

    return startRun;
}
```

GO ON TO THE NEXT PAGE.

A1d

(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[numTosses] values;

for(int i = 0; i < numTosses; i++){
    values[i] = cube.toss();
}

return values;
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

A1d

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 run=0; int temp=1; int a=0
    while (a < values.length) {
        temp=0
        while (values[a] == values[a+1]) {
            a++;
            temp++;
        }
        if (temp > run){
            run = temp
        }
        a++;
    }
    if (run == 1)
        return -1
    else
        return run
}
```

GO ON TO THE NEXT PAGE.

A1e

(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[] nums = new [numTosses];
    for (int k=0; k < nums.length; k++)
    {
        int r = (int)(Math.random() * 5 + 1);
        nums[k] = r;
    }
    return nums;
}
```

Part (b) begins on page 6.

Ale

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)
    {
      for (int k=0 ; k < values.length ; k++)
        { int r=0;
          int j = k ;
          while (values[j] == values[j+1])
            {
              j++ ;
              r++;
            }
          if (int rounds < r)
            {
              rounds == r;
            }
        }
        return rounds;
```

GO ON TO THE NEXT PAGE.

A1f.

(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)
```

```
public static int[] getCubeTosses(NumberCube cube, int numTosse
{
int [] results = new int [numTosses];

for (int x = 0; x < numTosses; x++)
{
results[x] = cube.toss();
}

return results;

}
```

Part (b) begins on page 6.

Alf

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)
```

```
public static int getLongestRun(int [] values)
{
int [] results = values.getCubeTosses(cube, values.length
int startLongestSame = 0;
int tracker = 0;
for(int x =0; x < results.length ; x++)
{
for(int k=0, k < results.length; k++)
{
if (results[x] == results[x+1]) {
startLongestSame = x;
x++;
}
}
}
return startLongestSame;
}
```

GO ON TO THE NEXT PAGE.

A1g

(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[] cubeTosses = new Array;
   for(int i=0; i < numTosses; i++)
   {
      cubeTosses[] += cube.toss(i)
   }
   return      cubeTosses[];
}
```

Part (b) begins on page 6.

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(); ++)
  {
    if (values[i+1] == values[i])
      consecutiveNums++;
  }
  return consecutiveNums - i;

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

A1b

(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 x;
  for (x=0; x<= numTosses; x++)
  { int[] set(x, toss());
  }
    return int();
}
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

A1h

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 longRun = 0;
  int x;
  for (x = 1; x < values.length; x++)
    { if (values[x] == values[x-1])
      longRun++;
      if (longRun == 0)
      return -1; }
      return longRun;
}
```

GO ON TO THE NEXT PAGE.

Ali

(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)
{
    for (int = numTosses; i < numTosses; i++)
    {
     arrayList[] values = new arrayList;
     values[i];
    }
  return values[i];
}
```

Part (b) begins on page 6.

Complete method `getLongestRun` below.

Ali

```
/** 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)
{
   for(int i=0; i < values.length -1; i++);
    {
       if (values[i] == values[i] + 1)
          return values.substring(i, i + 1);
       else
       return -1;
    }

    return -1;
```

GO ON TO THE NEXT PAGE.