Question 1: Number Cube

<table>
<thead>
<tr>
<th>Part (a)</th>
<th>getCubeTosses</th>
<th>4 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>constructs array</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>constructs an array of type int or size numTosses</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>constructs an array of type int and size numTosses</td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td>processes tosses</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>repeats execution of statements numTosses times</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>tosses cube in context of iteration</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>collects results of tosses</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>returns array of generated results</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part (b)</th>
<th>getLongestRun</th>
<th>5 points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>iterates over values</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>accesses element of values in context of iteration</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>accesses all elements of values, no out-of-bounds access potential</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>determines existence of run of consecutive elements</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>comparison involving an element of values</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>comparison of consecutive elements of values</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>always determines length of at least one run of consecutive elements</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>identifies maximum length run based on all runs</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>return value</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>returns starting index of identified maximum length run</td>
<td></td>
</tr>
<tr>
<td>+1/2</td>
<td>returns -1 if no run identified</td>
<td></td>
</tr>
</tbody>
</table>
(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.

```java
/**
 * 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
 * @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.
Complete method `getLongestRun` below.

```java
/** 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
 * @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.
(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.

```java
/**
 * 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.
Complete method `getLongestRun` below.

```java
/**
 * Returns the starting index of a longest run of two or more consecutive repeated values
 * @param values an array of integer values representing a series of number cube tosses
 * @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.
(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.

```java
/** 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
 * @return an array of `numTosses` values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
{
    for (int i = 0; i < numTosses; i++)
    {
        //arrayList[] values = new ArrayList;
        values.add(i);
    }
    return values;
}
```

Part (b) begins on page 6.
Complete method `getLongestRun` below.

```java
/**
 * 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
 * @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.
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 assigning its values to those 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.

Sample: A1a
Score: 9

The solution presented for part (a) earned all 4 points. It is canonical except for the fact that the `for` loop iterates `numTosses` times by using `arr.length`.

The solution presented for part (b) earned all 5 points. The iteration over `values` begins at 1 (`runIndex` is initialized to 0 for the first element). The expression `values[i] == values[runIndex]` compares consecutive elements because `i` and `runIndex` are initially 1 and 0, respectively. The length of the current run is stored in `runLength`, which is appropriately initialized, incremented, and reset. The check for a maximum length run immediately follows `runLength++`. Consequently, this check is always executed when a new (possibly longer) run is processed.

The variable `runIndex` is used to keep track of the beginning index of the current run. It is initialized to 0 and reset to `i`, the beginning of the next potential run, when the current run ends. The value of `runIndex` is assigned to `maxRunIndex` when a new maximum length run is identified and is returned after the `for` loop exits. The solution returns −1 if there is no run because `maxRunIndex` is initialized to −1 and is unchanged when no run is identified.

Sample: A1b
Score: 7

The solution presented for part (a) earned all 4 points. The student chooses an alternate yet allowable form of the array declaration `int numVals[]` instead of the more common `int[] numVals`. Also, `new int (numTosses)` received full credit because the distinction between `[]` and `()` is not a penalized error.

The solution presented for part (b) earned 3 out of 5 possible points. It does not access all elements of `values` because the `i < values.length` loop test allows `values[i+1]` to be out-of-bounds. A run length of 1 is calculated correctly because `run` is initialized, properly incremented, and reset at the end of a run.
A check for the maximum length run is found in the `else` clause, and as a result the maximum length run is not identified until `i` advances beyond the current run and `values[i] != values[i+1]`. Consequently, the longest run fails to be identified when it occurs at the end of `values`. The second `for` loop is used to locate the starting index of the maximum length run. This loop also fails to find the maximum length run because `values[j+1]` can be out-of-bounds. Additionally, `run` is not reinitialized either before the loop or inside the loop at the end of a run.

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 reinitialized to 0 because `j-run` would be one less than the correct value. Also, the test `run > 1` should instead be `run > 0` because the value of `run` is always one less than the actual run length. The solution returns −1 if there is no run because `startRun` is initialized to −1 and is unchanged when no run is identified.

**Sample: A1c**

**Score: 1**

The solution presented for part (a) earned no points. There is an attempt to construct an `ArrayList` using the keyword `new`, but none of the other required elements is present to properly construct the array. The loop for processing tosses fails to initialize `i`, toss the cube, or collect results into `values`. Additionally, the code incorrectly returns `values[i]` instead of `values`.

The solution presented for part (b) earned 1 out of 5 points. The `for` loop test condition of `i < values.length −1` would be correct if `values[i+1]` were in the loop body. However, `values[i]` never accesses the last element of `values`. Also, consecutive elements of `values` are not compared.

There is no attempt to determine the existence of a run or the maximum length run, and so these points were not earned. The "returns starting index of identified maximum length run" ½ point was not earned. Finally, since `return −1` is not based on the nonexistence of a run, the corresponding ½ point was not earned.