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.

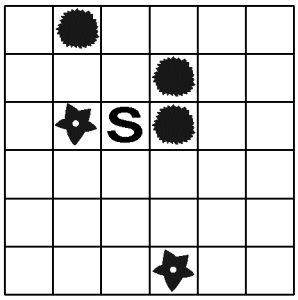
```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 *  in the array values.
 *  @param values an array of integer values representing a series of number cube tosses
 *         Precondition: values.length > 0
 *  @return the starting index of a run of maximum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

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

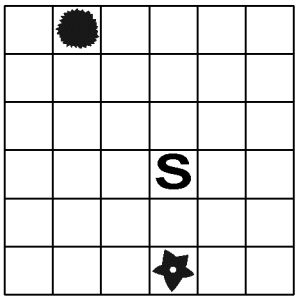
Consider the following scenario.

INITIAL WORLD



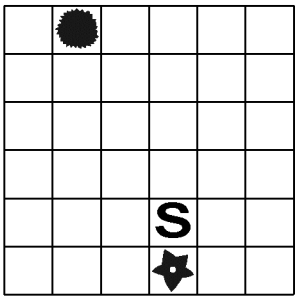
`StockpileCritter` is in location (2, 2), stockpile is empty

AFTER ONE ACT



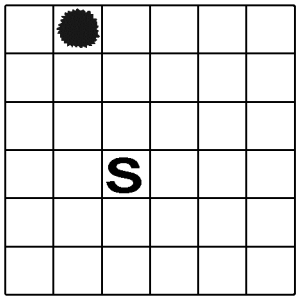
Gathered 3 actors, used 1 energy unit, 2 remaining in stockpile, moved to location (3, 3)

AFTER TWO ACTS



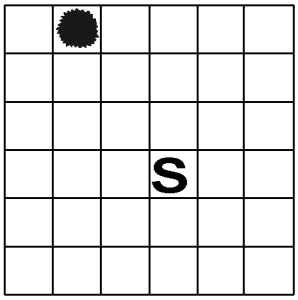
No actors gathered, used 1 energy unit, 1 remaining in stockpile, moved to location (4, 3)

AFTER THREE ACTS



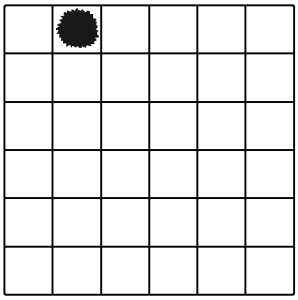
Gathered 1 actor, used 1 energy unit, 1 remaining in stockpile, moved to location (3, 2)

AFTER FOUR ACTS



No actors gathered, used 1 energy unit, 0 remaining in stockpile, moved to location (3, 3)

AFTER FIVE ACTS



Stockpile empty, removed self from grid

Write the complete `StockpileCritter` 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 `Critter` class and that updating an object's instance variable changes the state of that object.

AP® COMPUTER SCIENCE A
2009 SCORING GUIDELINES

Question 1: Number Cube

Part (a) `getCubeTosses` **4 points**

+1 constructs array
- **+1/2** constructs an array of type `int` **or** size `numTosses`
- **+1/2** constructs an array of type `int` **and** size `numTosses`

+2 1/2 processes tosses
- **+1** repeats execution of statements `numTosses` 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 `values`
- **+1/2** accesses element of `values` in context of iteration
- **+1/2** accesses all elements of `values`, no out-of-bounds access potential

+1 determines existence of run of consecutive elements
- **+1/2** comparison involving an element of `values`
- **+1/2** 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
- **+1/2** returns starting index of identified maximum length run
- **+1/2** returns -1 if no run identified