AP® Computer Science A
2011 Free-Response Questions

About the College Board

The College Board is a mission-driven not-for-profit organization that connects students to college success and opportunity. Founded in 1900, the College Board was created to expand access to higher education. Today, the membership association is made up of more than 5,900 of the world's leading educational institutions and is dedicated to promoting excellence and equity in education. Each year, the College Board helps more than seven million students prepare for a successful transition to college through programs and services in college readiness and college success — including the SAT® and the Advanced Placement Program®. The organization also serves the education community through research and advocacy on behalf of students, educators and schools.

Visit the College Board on the Web: www.collegeboard.org.
AP Central is the official online home for the AP Program: apcentral.collegeboard.com.

COMPUTER SCIENCE A
SECTION II
Time—1 hour and 45 minutes
Number of questions—4
Percent of total score—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. Digital sounds can be represented as an array of integer values. For this question, you will write two unrelated methods of the `Sound` class.

   A partial declaration of the `Sound` class is shown below.

```
public class Sound
{
  /** the array of values in this sound; guaranteed not to be null */
  private int[] samples;

  /** Changes those values in this sound that have an amplitude greater than limit.
   *  Values greater than limit are changed to limit.
   *  Values less than -limit are changed to -limit.
   *  @param limit the amplitude limit
   *         Precondition: limit ≥ 0
   *  @return the number of values in this sound that this method changed
   */
  public int limitAmplitude(int limit)
  {  /* to be implemented in part (a) */  }

  /** Removes all silence from the beginning of this sound.
   *  Silence is represented by a value of 0.
   *  Precondition: samples contains at least one nonzero value
   *  Postcondition: the length of samples reflects the removal of starting silence
   */
  public void trimSilenceFromBeginning()
  {  /* to be implemented in part (b) */  }

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

(a) The volume of a sound depends on the amplitude of each value in the sound. The amplitude of a value is its absolute value. For example, the amplitude of -2300 is 2300 and the amplitude of 4000 is 4000.

Write the method `limitAmplitude` that will change any value that has an amplitude greater than the given limit. Values that are greater than `limit` are replaced with `limit`, and values that are less than `-limit` are replaced with `-limit`. The method returns the total number of values that were changed in the array. For example, assume that the array `samples` has been initialized with the following values.

40	2532	17	-2300	-17	-4000	2000	1048	-420	33	15	-32	2030	3223

When the statement

```
int numChanges = limitAmplitude(2000);
```

is executed, the value of `numChanges` will be 5, and the array `samples` will contain the following values.

40	2000	17	-2000	-17	-2000	2000	1048	-420	33	15	-32	2000	2000

Complete method `limitAmplitude` below.

```
/** Changes those values in this sound that have an amplitude greater than limit.
 *  Values greater than limit are changed to limit.
 *  Values less than -limit are changed to -limit.
 *  @param limit the amplitude limit
 *         Precondition: limit ≥ 0
 *  @return the number of values in this sound that this method changed
 */
public int limitAmplitude(int limit)
```

(b) Recorded sound often begins with silence. Silence in a sound is represented by a value of 0.

Write the method `trimSilenceFromBeginning` that removes the silence from the beginning of a sound. To remove starting silence, a new array of values is created that contains the same values as the original `samples` array in the same order but without the leading zeros. The instance variable `samples` is updated to refer to the new array. For example, suppose the instance variable `samples` refers to the following array.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	0	0	0	-14	0	-35	-39	0	-7	16	32	37	29	0	0

After `trimSilenceFromBeginning` has been called, the instance variable `samples` will refer to the following array.

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	-14	0	-35	-39	0	-7	16	32	37	29	0	0

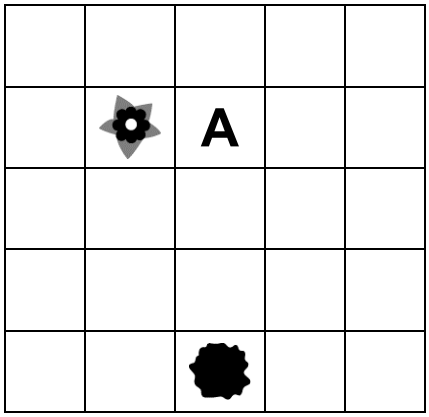
Complete method `trimSilenceFromBeginning` below.

```
/** Removes all silence from the beginning of this sound.
 *  Silence is represented by a value of 0.
 *  Precondition: samples contains at least one nonzero value
 *  Postcondition: the length of samples reflects the removal of starting silence
 */
public void trimSilenceFromBeginning()
```

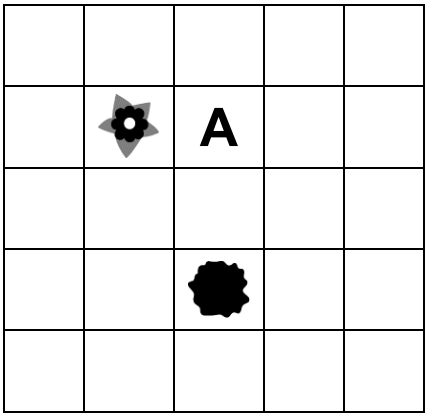
2. This question involves reasoning about the GridWorld case study. Reference materials are provided in the Appendix.

   An attractive critter is a critter that processes other actors by attempting to relocate all of the other actors in the grid, including other attractive critters. The attractive critter attempts to move each other actor one grid cell closer to itself in the direction specified by `getDirectionToward`. An actor is relocated only if the location into which it would be relocated is empty. If an actor cannot be moved, it is left in its original position. After trying to move all other actors, the attractive critter moves like a critter.

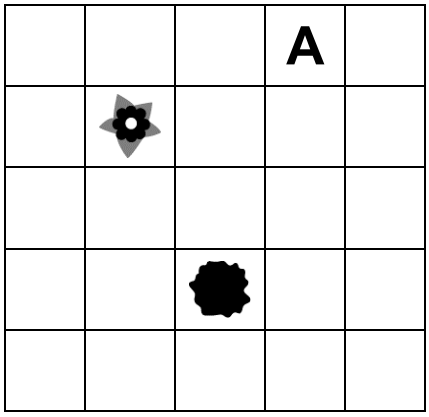
   The following series of figures represents an example of how the attractive critter affects the other actors in the grid.



An attractive critter (indicated by the letter A) is in location (1, 2), a flower is in location (1, 1), and a rock is in location (4, 2).



When the attractive critter acts, the rock will be relocated to location (3, 2) because that location is one grid cell closer to the attractive critter and is empty. The flower will not be relocated because the grid cell that is one location closer to the attractive critter is already occupied.



After attempting to relocate all the other actors in the grid, the attractive critter then moves like a critter. In this example, the attractive critter moves to location (0, 3).

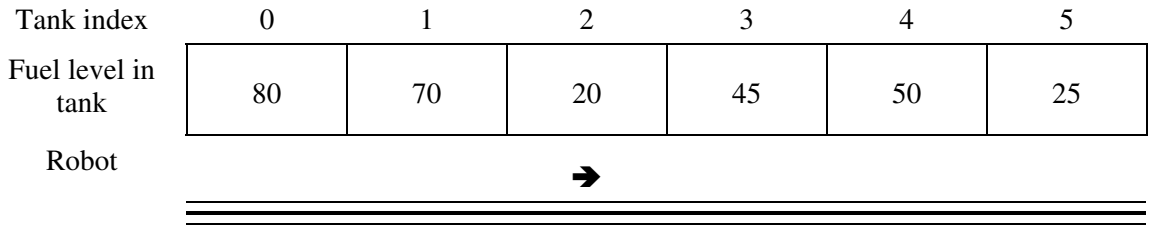
The order in which the actors in the grid are processed is not specified, making it possible to get different results from the same grid of actors.

Write the complete `AttractiveCritter` 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.

3. A fuel depot has a number of fuel tanks arranged in a line and a robot that moves a filling mechanism back and forth along the line so that the tanks can be filled. A fuel tank is specified by the `FuelTank` interface below.

```
public interface FuelTank
{
  /** @return an integer value that ranges from 0 (empty) to 100 (full) */
  int getFuelLevel();
}
```

A fuel depot keeps track of the fuel tanks and the robot. The following figure represents the tanks and the robot in a fuel depot. The robot, indicated by the arrow, is currently at index 2 and is facing to the right.



The state of the robot includes the index of its location and the direction in which it is facing (to the right or to the left). This information is specified in the `FuelRobot` interface as shown in the following declaration.

```
public interface FuelRobot
{
  /** @return the index of the current location of the robot */
  int getCurrentIndex();


  /** Determine whether the robot is currently facing to the right
   *  @return true  if the robot is facing to the right (toward tanks with larger indexes)
   *          false if the robot is facing to the left (toward tanks with smaller indexes)
   */
  boolean isFacingRight();


  /** Changes the current direction of the robot */
  void changeDirection();


  /** Moves the robot in its current direction by the number of locations specified.
   *  @param numLocs the number of locations to move. A value of 1 moves
   *                 the robot to the next location in the current direction.
   *          Precondition: numLocs > 0
   */
  void moveForward(int numLocs);
}
```

A fuel depot is represented by the `FuelDepot` class as shown in the following class declaration.

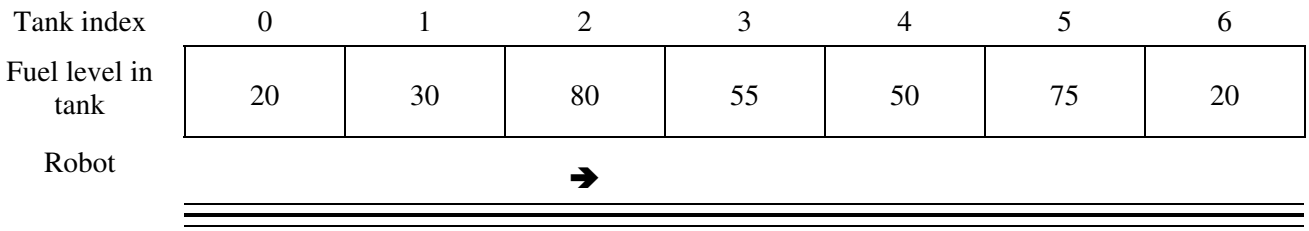
```
public class FuelDepot
{
  /** The robot used to move the filling mechanism */
  private FuelRobot filler;

  /** The list of fuel tanks */
  private List<FuelTank> tanks;

  /** Determines and returns the index of the next tank to be filled.
   *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
   *  @return index of the location of the next tank to be filled
   *  Postcondition: the state of the robot has not changed
   */
  public int nextTankToFill(int threshold)
  {  /* to be implemented in part (a) */  }


  /** Moves the robot to location locIndex.
   *  @param locIndex the index of the location of the tank to move to
   *         Precondition: 0 ≤ locIndex < tanks.size()
   *  Postcondition: the current location of the robot is locIndex
   */
  public void moveToLocation(int locIndex)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `FuelDepot` method `nextTankToFill` that returns the index of the next tank to be filled.

The index for the next tank to be filled is determined according to the following rules:

- Return the index of a tank with the lowest fuel level that is less than or equal to a given threshold. If there is more than one fuel tank with the same lowest fuel level, any of their indexes can be returned.
- If there are no tanks with a fuel level less than or equal to the threshold, return the robot's current index.

For example, suppose the tanks contain the fuel levels shown in the following figure.



The following table shows the results of several independent calls to `nextTankToFill`.

`threshold`	Return Value	Rationale
50	0 or 6	20 is the lowest fuel level, so either 0 or 6 can be returned.
15	2	There are no tanks with a fuel level ≤ `threshold`, so the robot's current index is returned.

Complete method `nextTankToFill` below.

```
/** Determines and returns the index of the next tank to be filled.
 *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
 *  @return index of the location of the next tank to be filled
 *  Postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
```

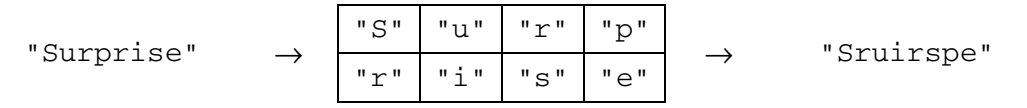
(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```
/** Moves the robot to location locIndex.
 *  @param locIndex the index of the location of the tank to move to
 *         Precondition: 0 ≤ locIndex < tanks.size()
 *  Postcondition: the current location of the robot is locIndex
 */
public void moveToLocation(int locIndex)
```

4. In this question you will write two methods for a class `RouteCipher` that encrypts (puts into a coded form) a message by changing the order of the characters in the message. The route cipher fills a two-dimensional array with single-character substrings of the original message in row-major order, encrypting the message by retrieving the single-character substrings in column-major order.

   For example, the word `"Surprise"` can be encrypted using a 2-row, 4-column array as follows.



   An incomplete implementation of the `RouteCipher` class is shown below.

```
public class RouteCipher
{
  /** A two-dimensional array of single-character strings, instantiated in the constructor */
  private String[][] letterBlock;

  /** The number of rows of letterBlock, set by the constructor */
  private int numRows;

  /** The number of columns of letterBlock, set by the constructor */
  private int numCols;

  /** Places a string into letterBlock in row-major order.
   *  @param str the string to be processed
   *  Postcondition:
   *    if str.length() < numRows * numCols, "A" is placed in each unfilled cell
   *    if str.length() > numRows * numCols, trailing characters are ignored
   */
  private void fillBlock(String str)
  {  /* to be implemented in part (a) */  }

  /** Extracts encrypted string from letterBlock in column-major order.
   *  Precondition: letterBlock has been filled
   *  @return the encrypted string from letterBlock
   */
  private String encryptBlock()
  {  /* implementation not shown */  }

  /** Encrypts a message.
   *  @param message the string to be encrypted
   *  @return the encrypted message;
   *          if message is the empty string, returns the empty string
   */
  public String encryptMessage(String message)
  {  /* to be implemented in part (b) */  }

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

(a) Write the method `fillBlock` that fills the two-dimensional array `letterBlock` with one-character strings from the string passed as parameter `str`.

The array must be filled in row-major order—the first row is filled from left to right, then the second row is filled from left to right, and so on, until all rows are filled.

If the length of the parameter `str` is smaller than the number of elements of the array, the string `"A"` is placed in each of the unfilled cells. If the length of `str` is larger than the number of elements in the array, the trailing characters are ignored.

For example, if `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at noon"`, the resulting contents of `letterBlock` would be as shown in the following table.

"M"	"e"	"e"	"t"	" "
"a"	"t"	" "	"n"	"o"
"o"	"n"	"A"	"A"	"A"

If `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at midnight"`, the resulting contents of `letterBlock` would be as shown in the following table.

"M"	"e"	"e"	"t"	" "
"a"	"t"	" "	"m"	"i"
"d"	"n"	"i"	"g"	"h"

The following expression may be used to obtain a single-character string at position `k` of the string `str`.

```
str.substring(k, k + 1)
```

Complete method `fillBlock` below.

```
/** Places a string into letterBlock in row-major order.
 *  @param str the string to be processed
 *  Postcondition:
 *    if str.length() < numRows * numCols, "A" is placed in each unfilled cell
 *    if str.length() > numRows * numCols, trailing characters are ignored
 */
private void fillBlock(String str)
```

(b) Write the method `encryptMessage` that encrypts its string parameter `message`. The method builds an encrypted version of `message` by repeatedly calling `fillBlock` with consecutive, nonoverlapping substrings of `message` and concatenating the results returned by a call to `encryptBlock` after each call to `fillBlock`. When all of `message` has been processed, the concatenated string is returned. Note that if `message` is the empty string, `encryptMessage` returns an empty string.

The following example shows the process carried out if `letterBlock` has 2 rows and 3 columns and `encryptMessage("Meet at midnight")` is executed.

Substring	`letterBlock` after Call to `fillBlock`	Value Returned by `encryptBlock`	Concatenated String
`"Meet a"`	`"M"` `"e"` `"e"` `"t"` `" "` `"a"`	`"Mte ea"`	`"Mte ea"`
`"t midn"`	`"t"` `" "` `"m"` `"i"` `"d"` `"n"`	`"ti dmn"`	`"Mte eati dmn"`
`"ight"`	`"i"` `"g"` `"h"` `"t"` `"A"` `"A"`	`"itgAhA"`	`"Mte eati dmnitgAhA"`

In this example, the method returns the string `"Mte eati dmnitgAhA"`.

Assume that `fillBlock` and `encryptBlock` methods work as specified. Solutions that reimplement the functionality of one or both of these methods will not receive full credit.

Complete method `encryptMessage` below.

```
/** Encrypts a message.
 *  @param message the string to be encrypted
 *  @return the encrypted message;
 *          if message is the empty string, returns the empty string
 */
public String encryptMessage(String message)
```

STOP

END OF EXAM