COMPUTER SCIENCE 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.

Write your answers in pencil only in the booklet provided.

Notes:

- Assume that the classes in the Quick Reference have been imported where needed.
- Unless otherwise stated, 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. This question manipulates one-dimensional and two-dimensional arrays. In part (a) you will write a method to reverse elements of a one-dimensional array. In parts (b) and (c) you will write methods to reverse elements of a two-dimensional array.

   (a) Consider the following incomplete `ArrayUtil` class, which contains a static `reverseArray` method.

```
public class ArrayUtil
{
    /** Reverses elements of array arr.
     *  Precondition: arr.length > 0.
     *  Postcondition: The elements of arr have been reversed.
     *  @param arr the array to manipulate
     */
    public static void reverseArray(int[] arr)
    { /* to be implemented in part (a) */ }

    //Other methods are not shown.
}
```

Write the `ArrayUtil` method `reverseArray`. For example, if `arr` is the array `{2,7,5,1,0}`, the call to `reverseArray` changes `arr` to be `{0,1,5,7,2}`.

Complete method `reverseArray` below.

```
/** Reverses elements of array arr.
 *  Precondition: arr.length > 0.
 *  Postcondition: The elements of arr have been reversed.
 *  @param arr the array to manipulate
 */
public static void reverseArray(int[] arr)
```

(b) Consider the following incomplete `Matrix` class, which represents a two-dimensional matrix of integers. Assume that the matrix contains at least one integer.

```
public class Matrix
{
    private int[][] mat;

    /** Constructs a matrix of integers. */
    public Matrix (int[][] m)
    {  mat = m; }

    /** Reverses the elements in each row of mat.
     *  Postcondition: The elements in each row have been reversed.
     */
    public void reverseAllRows()
    { /* to be implemented in part (b) */ }

    /** Reverses the elements of mat.
     *  Postcondition:
     *    - The final elements of mat, when read in row-major order,
     *      are the same as the original elements of mat when read
     *      from the bottom corner, right to left, going upward.
     *    - mat[0][0] contains what was originally the last element.
     *    - mat[mat.length-1][mat[0].length-1] contains what was
     *      originally the first element.
     */
    public void reverseMatrix()
    { /* to be implemented in part (c) */ }

    //Other instance variables, constructors and methods are not sh
}
```

Write the `Matrix` method `reverseAllRows`. This method reverses the elements of each row. For example, if `mat1` refers to a `Matrix` object, then the call `mat1.reverseAllRows()` will change the matrix as shown below.

Before call

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

After call

	0	1	2	3
0	4	3	2	1
1	8	7	6	5
2	12	11	10	9

In writing `reverseAllRows`, you *must* call the `reverseArray` method in part (a). Assume that `reverseArray` works correctly regardless of what you wrote in part (a).

Complete method `reverseAllRows` below.

```
/** Reverses the elements in each row of mat.
 *  Postcondition: The elements in each row have been reversed.
 */
public void reverseAllRows()
```

(c) Write the `Matrix` method `reverseMatrix`. This method reverses the elements of a matrix such that the final elements of the matrix, when read in row-major order, are the same as the original elements when read from the bottom corner, right to left, going upward. Again let `mat1` be a reference to a `Matrix` object. The the call `mat1.reverseMatrix()` will change the matrix as shown below.

Before call

	0	1
0	1	2
1	3	4
2	5	6

After call

	0	1
0	6	5
1	4	3
2	2	1

In writing `reverseMatrix`, you *must* call the `reverseAllRows` method in part (b). Assume that `reverseAllRows` works correctly regardless of what you wrote in part (b).

Complete method `reverseMatrix` below.

```
/** Reverses the elements of mat.
 *  Postcondition:
 *    - The final elements of mat, when read in row-major order,
 *      are the same as the original elements of mat when read
 *      from the bottom corner, right to left, going upward.
 *    - mat[0][0] contains what was originally the last element.
 *    - mat[mat.length-1][mat[0].length-1] contains what was
 *      originally the first element.
 */
public void reverseMatrix()
```

2. A text-editing program uses a Sentence class that manipulates a single sentence. A sentence contains letters, blanks, and punctuation. The first character in a sentence is a letter, and the last character is a punctuation mark. Any two words in the sentence are separated by a single blank. A partial implementation of the Sentence class is as follows.

```
public class Sentence
{
    /** The sentence to manipulate */
    private String sentence;

    /**  @return an ArrayList of integer positions containing a
     *   blank in this sentence. If there are no blanks in the
     *   sentence, returns an empty list.
     */
    public List<Integer> getBlankPositions()
    { /* to be implemented in part (a) */ }

    /**  @return the number of words in this sentence
     *   Precondition: Sentence contains at least one word.
     */
    public int countWords()
    { /* to be implemented in part (b) */ }

    /** @return the array of words in this sentence
     *  Precondition:
     *   - Any two words in the sentence are separated by one blank.
     *   - The sentence contains at least one word.
     *  Postcondition: String[] returned containing the words in
     *                 this sentence.
     */
    public String[] getWords()
    { /* to be implemented in part (c) */ }

    //Constructor and other methods are not shown.
}
```

(a) Write the Sentence method getBlankPositions, which returns an ArrayList of integers that represent the positions in a sentence containing blanks. If there are no blanks in the sentence, getBlankPositions should return an empty list.
Some results of calling getBlankPositions are shown below.

Sentence	Result of call to getBlankPositions
I love you!	[1, 6]
The cat sat on the mat.	[3, 7, 11, 14, 18]
Why?	[ ]

Complete method getBlankPositions below.

```
/**  @return an ArrayList of integer positions containing a
 *   blank in this sentence. If there are no blanks in the
 *   sentence, returns an empty list.
 */
public List<Integer> getBlankPositions()
```

(b) Write the Sentence method countWords, which returns the number of words in a sentence. Words are sequences of letters or punctuation, separated by a single blank. You may assume that every sentence contains at least one word.
For example:

Sentence	Result returned by countWords
I love you!	3
The cat sat on the mat.	6
Why?	1

Complete method countWords below.

```
/**  @return the number of words in this sentence
 *   Precondition: Sentence contains at least one word.
 */
public int countWords()
```

(c) Write the Sentence method getWords, which returns an array of words in the sentence. A word is defined as a string of letters and punctuation, and does not contain any blanks. You may assume that a sentence contains at least one word.
Some examples of calling getWords are shown below.

Sentence	Result returned by getWords
The bird flew away.	{The, bird, flew, away.}
Wow!	{Wow!}
Hi! How are you?	{Hi!, How, are, you?}

In writing method getWords, you *must* use methods getBlankPositions and countWords, which were written in parts (a) and (b). You may assume that these methods work correctly, irrespective of what you wrote in parts (a) and (b).

Complete method getWords below.

```
/** @return the array of words in this sentence
 *  Precondition:
 *   - Any two words in the sentence are separated by one blank.
 *   - The sentence contains at least one word.
 *  Postcondition: String[] returned containing the words in
 *                 this sentence.
 */
public String[] getWords()
```

3. In this question you will implement two methods for a class `Tournament` that keeps track of the players who have registered for a tournament. The `Tournament` class uses the `Player` class shown below. A `Player` has a name and player number specified when a player is constructed.

```
public class Player
{
    public Player(String name, int playerNumber)
    { /* implementation not shown */ }

    public int getPlayerNumber()
    { /* implementation not shown */ }

    //Private instance variables and other methods are not shown.
}
```

An incomplete declaration for the `Tournament` class is shown below. There are 100 available slots for players in the tournament, and the players are numbered 0, 1, 2, ..., 99.

```
public class Tournament
{
    /** The list of slots in the tournament.
     *  Each element corresponds to a slot in the tournament.
     *  If slots[i] is null, the slot is not yet taken;
     *  otherwise it contains a reference to a Player.
     *  For example, slots[i].getPlayerNumber() returns i.
     */
    private Player[] slots;

    /** The list of names of players who wish to participate in
     *  the tournament, but cannot because all slots are taken.
     */
    private List<String> waitingList;
```

```
    /** If there are any empty slots (slots with no Player)
     *  assign the player with the specified playerName to an
     *  empty slot. Create and return the new Player.
     *  If there are no available slots, add the player's name
     *  to the end of the waiting list and return null.
     *  @playerName the name of the person requesting a slot
     *  @return the new Player
     */
    public Player requestSlot(String playerName)
    { /* to be implemented in part (a) */ }

    /** Release the slot for player p, thus removing that player
     *  from the tournament. If there are any names in waitingList,
     *  remove the first name and create a Player in the
     *  canceled slot for this person. Return the new Player.
     *  If waitingList is empty, mark the slot specified by p as
     *  empty and return null.
     *  Precondition: p is a valid Player for some slot in
     *                this tournament.
     *  @param p the player who will be removed from the tournament
     *  @return the new Player placed in the canceled slot
     */
    public Player cancelAndReassignSlot(Player p)
    { /* to be implemented in part (b) */ }

    //Constructor and other methods are not shown.
}
```

(a) Write the `Tournament` method `requestSlot`. Method `requestSlot` tries to reserve a slot in the tournament for a given player. If there are any available slots in the tournament, one of them is assigned to the named player, and the newly created `Player` is returned. If there are no available slots, the player's name is added to the end of the waiting list and `null` is returned.

Complete method `requestSlot` below.

```
/** If there are any empty slots (slots with no Player)
 *  assign the player with the specified playerName to an
 *  empty slot. Create and return the new Player.
 *  If there are no available slots, add the player's name
 *  to the end of the waiting list and return null.
 *  @playerName the name of the person requesting a slot
 *  @return the new Player
 */
public Player requestSlot(String playerName)
```

(b) Write the `Tournament` method `cancelAndReassignSlot`. This method releases a previous player's slot. If the waiting list for the tournament contains any names, the newly available slot is reassigned to the person at the front of the list. That person's name is removed from the waiting list, and the newly created `Player` is returned. If the waiting list is empty, the newly released slot is marked as empty, and null is returned.

In writing `cancelAndReassignSlot`, you may use any accessible methods in the `Player` and `Tournament` classes. Assume that these methods work as specified.

Information repeated from the beginning of the question

```
public class Player

public Player(String name, int playerNumber)
public int getPlayerNumber()

public class Tournament

private Player[] slots
private List<String> waitingList
public Player requestSlot(String playerName)
public Player cancelAndReassignSlot(Player p)
```

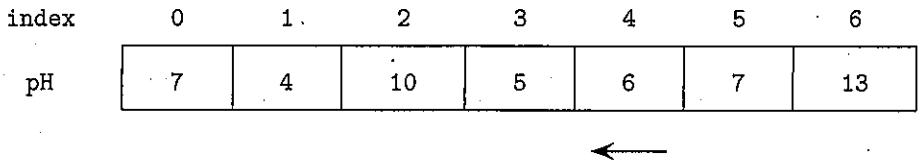
Complete method `cancelAndReassignSlot` below.

```
/** Release the slot for player p, thus removing that player
 *  from the tournament. If there are any names in waitingList,
 *  remove the first name and create a Player in the
 *  canceled slot for this person. Return the new Player.
 *  If waitingList is empty, mark the slot specified by p as
 *  empty and return null.
 *  Precondition: p is a valid Player for some slot in
 *                this tournament.
 *  @param p the player who will be removed from the tournament
 *  @return the new Player placed in the canceled slot
 */
public Player cancelAndReassignSlot(Player p)
```

4. A chemical solution is said to be *acidic* if it has a pH integer value from 1 to 6, inclusive. The lower the pH, the more acidic the solution.
An experiment has a large number of chemical solutions arranged in a line and a mechanical arm that moves back and forth along the line, so that the acidity of each solution can be altered by adding various chemicals. A chemical solution is specified by the `Solution` interface below.

```
public interface Solution
{
    /** @return an integer value that ranges from 1 (very acidic)
     *  to 14 */
    int getPH();

    /** Set PH to newValue.
     *  @param newValue the new PH value */
    void setPH(int newValue);
}
```

The experiment keeps track of the solutions and the mechanical arm. The figure below represents the solutions and mechanical arm in an experiment. The arm, indicated by the arrow, is currently at index 4 and is facing left. The second row of integers represents the pH values of the solutions.



In this experiment, the most acidic solution is at index 1, since its pH value is the lowest.

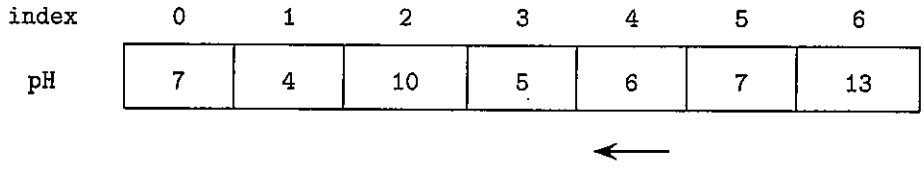
The state of the mechanical arm includes the index of its location and direction it is facing (to the right or to the left). A mechanical arm is specified by the `MechanicalArm` interface below.

```
public interface MechanicalArm
{
    /** @return the index of the current location of the
     *  mechanical arm */
    int getCurrentIndex();

    /** @return true if the mechanical arm is facing right
     *  (toward solutions with larger indexes),
     *  false if the mechanical arm is facing left
     *  (toward solutions with smaller indexes)
     */
    boolean isFacingRight();

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

    /** Moves the mechanical arm forward in its current direction
     *  by the number of locations specified.
     *  @param numLocs the number of locations to move
     *  Precondition: numLocs >= 0.
     */
    void moveForward(int numLocs);
}
```

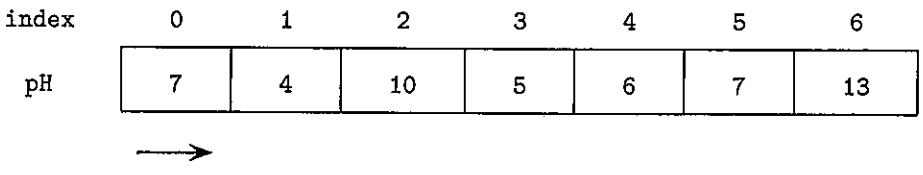
An experiment is represented by the Experiment class shown below.

```
public class Experiment
{
    /** The mechanical arm used to process the solutions */
    private MechanicalArm arm;

    /** The list of solutions */
    private List<Solution> solutions;

    /** Resets the experiment.
     *  Postcondition:
     *   - The mechanical arm has a current index of 0.
     *   - It is facing right.
     */
    public void reset()
    { /* to be implemented in part (a) */ }

    /** Finds and returns the index of the most acidic solution.
     *  @return index the location of the most acidic solution
     *  or -1 if there are no acidic solutions
     *  Postcondition:
     *   - The mechanical arm is facing right.
     *   - Its current index is at the most acidic solution, or at
     *     0 if there are no acidic solutions.
     */
    public int mostAcidic()
    { /* to be implemented in part (b) */ }
}
```

(a) Write the Experiment method reset that places the mechanical arm facing right, at index 0.
For example, suppose the experiment contains the solutions with pH values shown. The arrow represents the mechanical arm.



A call to reset will result in



Information repeated from the beginning of the question

```
public interface Solution

int getPH()
void setPH(int newValue)

public interface MechanicalArm

int getCurrentIndex()
boolean isFacingRight()
void changeDirection()
void moveForward(int numLocs)

public class Experiment

private MechanicalArm arm
private List<Solution> solutions
public void reset()
public int mostAcidic()
```

Complete method reset below.

```
/** Resets the experiment.
 *  Postcondition:
 *    - The mechanical arm has a current index of 0.
 *    - It is facing right.
 */
public void reset()
```

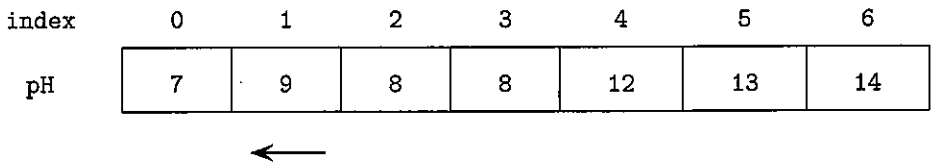
(b) Write the Experiment method mostAcidic that returns the index of the most acidic solution and places the mechanical arm facing right at the location of the most acidic solution. A solution is acidic if its pH is less than 7. The lower the pH, the more acidic the solution. If there are no acidic solutions in the experiment, the mostAcidic method should return -1 and place the mechanical arm at index 0, facing right.
For example, suppose the experiment has this state:

index	0	1	2	3	4	5	6
pH	7	4	10	5	6	7	13

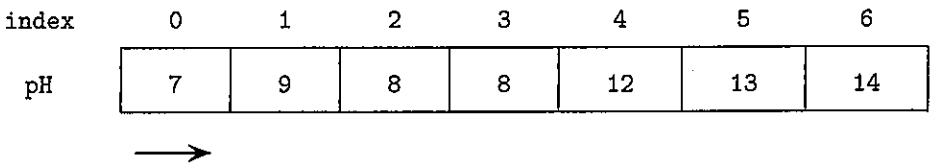
A call to mostAcidic should return the value 1 and result in the following state for the experiment:

index	0	1	2	3	4	5	6
pH	7	4	10	5	6	7	13

If the experiment has this state,



a call to `mostAcidic` should return the value `-1` and result in the following state for the experiment:



Information repeated from the beginning of the question

```
public interface Solution

int getPH()
void setPH(int newValue)

public interface MechanicalArm

int getCurrentIndex()
boolean isFacingRight()
void changeDirection()
void moveForward(int numLocs)

public class Experiment

private MechanicalArm arm
private List<Solution> solutions
public void reset()
public int mostAcidic()
```

Complete method `mostAcidic` below.

```
/** Finds and returns the index of the most acidic solution.
 *  @return index the location of the most acidic solution
 *  or -1 if there are no acidic solutions
 *  Postcondition:
 *   - The mechanical arm is facing right.
 *   - Its current index is at the most acidic solution, or at
 *     0 if there are no acidic solutions.
 */
public int mostAcidic()
```

END OF EXAMINATION