COMPUTER SCIENCE A
SECTION II

Time—1 hour and 30 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 Java Quick Reference 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 will not receive full credit.

1. Consider the following partial declaration for a `WordScrambler` class. The constructor for the `WordScrambler` class takes an even-length array of `String` objects and initializes the instance variable `scrambledWords`.

```
public class WordScrambler
{
  private String[] scrambledWords;

  /** @param wordArr an array of String objects
   *          Precondition: wordArr.length is even
   */
  public WordScrambler(String[] wordArr)
  {
    scrambledWords = mixedWords(wordArr);
  }


  /** @param word1 a String of characters
   *  @param word2 a String of characters
   *  @return a String that contains the first half of word1 and the second half of word2
   */
  private String recombine(String word1, String word2)
  {  /* to be implemented in part (a) */  }


  /** @param words an array of String objects
   *          Precondition: words.length is even
   *  @return an array of String objects created by recombining pairs of strings in array words
   *  Postcondition: the length of the returned array is words.length
   */
  private String[] mixedWords(String[] words)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `WordScrambler` method `recombine`. This method returns a `String` created from its two `String` parameters as follows.

- take the first half of `word1`
- take the second half of `word2`
- concatenate the two halves and return the new string.

For example, the following table shows some results of calling `recombine`. Note that if a word has an odd number of letters, the second half of the word contains the extra letter.

`word1`	`word2`	`recombine(word1, word2)`
`"apple"`	`"pear"`	`"apar"`
`"pear"`	`"apple"`	`"peple"`

Complete method `recombine` below.

```
/** @param word1 a String of characters
 *  @param word2 a String of characters
 *  @return a String that contains the first half of word1 and the second half of word2
 */
private String recombine(String word1, String word2)
```

(b) Write the `WordScrambler` method `mixedWords`. This method creates and returns a new array of `String` objects as follows.

It takes the first pair of strings in `words` and combines them to produce a pair of strings to be included in the array returned by the method. If this pair of strings consists of `w1` and `w2`, the method should include the result of calling `recombine` with `w1` and `w2` as arguments and should also include the result of calling `recombine` with `w2` and `w1` as arguments. The next two strings, if they exist, would form the next pair to be processed by this method. The method should continue until all the strings in `words` have been processed in this way and the new array has been filled. For example, if the array `words` contains the following elements:

```
{"apple", "pear", "this", "cat"}
```

then the call `mixedWords(words)` should return the following array.

```
{"apar", "peple", "that", "cis"}
```

In writing `mixedWords`, you may call `recombine`. Assume that `recombine` works as specified, regardless of what you wrote in part (a).

Complete method `mixedWords` below.

```
/** @param words an array of String objects
 *          Precondition: words.length is even
 *  @return an array of String objects created by recombining pairs of strings in array words
 *  Postcondition: the length of the returned array is words.length
 */
private String[] mixedWords(String[] words)
```

GO ON TO THE NEXT PAGE.

2. An array of positive integer values has the *mountain* property if the elements are ordered such that successive values increase until a maximum value (the peak of the mountain) is reached and then the successive values decrease. The `Mountain` class declaration shown below contains methods that can be used to determine if an array has the mountain property. You will implement two methods in the `Mountain` class.

```
public class Mountain
{
  /** @param array an array of positive integer values
   *  @param stop the last index to check
   *         Precondition: 0 ≤ stop < array.length
   *  @return true if for each j such that 0 ≤ j < stop, array[j] < array[j + 1] ;
   *          false otherwise
   */
  public static boolean isIncreasing(int[] array, int stop)
  {  /*  implementation not shown  */  }


  /** @param array an array of positive integer values
   *  @param start the first index to check
   *        Precondition: 0 ≤ start < array.length - 1
   *  @return true if for each j such that start ≤ j < array.length - 1,
   *                array[j] > array[j + 1];
   *          false otherwise
   */
  public static boolean isDecreasing(int[] array, int start)
  {  /*  implementation not shown  */  }


  /** @param array an array of positive integer values
   *         Precondition: array.length > 0
   *  @return the index of the first peak (local maximum) in the array, if it exists;
   *          -1 otherwise
   */
  public static int getPeakIndex(int[] array)
  {  /*  to be implemented in part (a)  */  }


  /** @param array an array of positive integer values
   *         Precondition: array.length > 0
   *  @return true if array contains values ordered as a mountain;
   *          false otherwise
   */
  public static boolean isMountain(int[] array)
  {  /*  to be implemented in part (b)  */  }

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

}
```

GO ON TO THE NEXT PAGE.

(a) Write the `Mountain` method `getPeakIndex`. Method `getPeakIndex` returns the index of the first peak found in the parameter `array`, if one exists. A peak is defined as an element whose value is greater than the value of the element immediately before it and is also greater than the value of the element immediately after it. Method `getPeakIndex` starts at the beginning of the array and returns the index of the first peak that is found or -1 if no peak is found.

For example, the following table illustrates the results of several calls to `getPeakIndex`.

arr	getPeakIndex(arr)
{11, 22, 33, 22, 11}	2
{11, 22, 11, 22, 11}	1
{11, 22, 33, 55, 77}	-1
{99, 33, 55, 77, 120}	-1
{99, 33, 55, 77, 55}	3
{33, 22, 11}	-1

Complete method `getPeakIndex` below.

```
/** @param array an array of positive integer values
 *         Precondition: array.length > 0
 *  @return the index of the first peak (local maximum) in the array, if it exists;
 *          -1 otherwise
 */
public static int getPeakIndex(int[] array)
```

GO ON TO THE NEXT PAGE.

(b) Write the `Mountain` method `isMountain`. Method `isMountain` returns `true` if the values in the parameter `array` are ordered as a mountain; otherwise, it returns `false`. The values in `array` are ordered as a mountain if all three of the following conditions hold.

- There must be a peak.
- The array elements with an index smaller than the peak's index must appear in increasing order.
- The array elements with an index larger than the peak's index must appear in decreasing order.

For example, the following table illustrates the results of several calls to `isMountain`.

arr	isMountain(arr)
{1, 2, 3, 2, 1}	true
{1, 2, 1, 2, 1}	false
{1, 2, 3, 1, 5}	false
{1, 4, 2, 1, 0}	true
{9, 3, 5, 7, 5}	false
{3, 2, 1}	false

In writing `isMountain`, assume that `getPeakIndex` works as specified, regardless of what you wrote in part (a).

Complete method `isMountain` below.

```
/** @param array an array of positive integer values
 *         Precondition: array.length > 0
 *  @return true if array contains values ordered as a mountain;
 *          false otherwise
 */
public static boolean isMountain(int[] array)
```

GO ON TO THE NEXT PAGE.

3. A two-dimensional array of temperatures is represented in the following class.

```
public class TemperatureGrid
{
  /** A two-dimensional array of temperature values, initialized in the constructor (not shown)
   *  Guaranteed not to be null
   */
  private double[][] temps;

  /** Computes and returns a new temperature value for the given location.
   *  @param row a valid row index in temps
   *  @param col a valid column index in temps
   *  @return the new temperature for temps[row][col]
   *          - The new temperature for any element in the border of the array is the
   *            same as the old temperature.
   *          - Otherwise, the new temperature is the average of the four adjacent entries.
   *  Postcondition: temps is unchanged.
   */
  private double computeTemp(int row, int col)
  {  /* to be implemented in part (a) */  }


  /** Updates all values in temps and returns a boolean that indicates whether or not all the
   *  new values were within tolerance of the original values.
   *  @param tolerance a double value >= 0
   *  @return true if all updated temperatures are within tolerance of the original values;
   *          false otherwise.
   *  Postcondition: Each value in temps has been updated with a new value based on the
   *                 corresponding call to computeTemp.
   */
  public boolean updateAllTemps(double tolerance)
  {  /* to be implemented in part (b) */  }

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

GO ON TO THE NEXT PAGE.

(a) Write method `computeTemp`, which computes and returns the new temperature for a given element of `temps` according to the following rules.

1. If the element is in the border of the array (in the first row or last row or first column or last column), the new temperature is the same as the old temperature.

2. Otherwise, the new temperature is the average (arithmetic mean) of the temperatures of the four adjacent values in the table (located above, below, to the left, and to the right of the element).

If `temps` is the table shown below, `temps.length` is `5` and `temps[0].length` is `6`.

	0	1	2	3	4	5
0	95.5	100.0	100.0	100.0	100.0	110.3
1	0.0	50.0	50.0	50.0	50.0	0.0
2	0.0	40.0	40.0	40.0	40.0	0.0
3	0.0	20.0	20.0	20.0	20.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

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

Function Call	Result
`computeTemp(2, 3)`	`37.5` (the average of the values 50.0, 20.0, 40.0, and 40.0)
`computeTemp(1, 1)`	`47.5` (the average of the values 100.0, 40.0, 0.0, and 50.0)
`computeTemp(0, 2)`	`100.0` (the same as the old value)
`computeTemp(1, 3)`	`60.0` (the average of the values 100.0, 40.0, 50.0, and 50.0)

Complete method `computeTemp` below.

```
/** Computes and returns a new temperature value for the given location.
 *  @param row a valid row index in temps
 *  @param col a valid column index in temps
 *  @return the new temperature for temps[row][col]
 *          - The new temperature for any element in the border of the array is the
 *            same as the old temperature.
 *          - Otherwise, the new temperature is the average of the four adjacent entries.
 *  Postcondition: temps is unchanged.
 */
private double computeTemp(int row, int col)
```

(b) Write method `updateAllTemps`, which computes the new temperature for every element of `temps`. The new values should be based on the original values, so it will be necessary to create another two-dimensional array in which to store the new values. Once all the computations are complete, the new values should replace the corresponding positions of `temps`. Method `updateAllTemps` also determines whether every new temperature is within `tolerance` of the corresponding old temperature (i.e., the absolute value of the difference between the old temperature and the new temperature is less than or equal to `tolerance`). If so, it returns `true`; otherwise, it returns `false`.

If `temps` contains the values shown in the first table below, then the call `updateAllTemps(0.01)` should update `temps` as shown in the second table.

`temps` before the call `updateAllTemps(0.01)`

	0	1	2	3	4	5
0	95.5	100.0	100.0	100.0	100.0	110.3
1	0.0	50.0	50.0	50.0	50.0	0.0
2	0.0	40.0	40.0	40.0	40.0	0.0
3	0.0	20.0	20.0	20.0	20.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

`temps` after the call `updateAllTemps(0.01)`

	0	1	2	3	4	5
0	95.5	100.0	100.0	100.0	100.0	110.3
1	0.0	47.5	60.0	60.0	47.5	0.0
2	0.0	27.5	37.5	37.5	27.5	0.0
3	0.0	15.0	20.0	20.0	15.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

In the example shown, the call `updateAllTemps(0.01)` should return `false` because there are several new temperatures that are not within the given tolerance of the corresponding old temperature. For example, the updated value in `temps[2][3]` is `37.5`, the original value in `temps[2][3]` was `40.0`, and the absolute value of `(37.5 - 40.0)` is greater than the value of `tolerance` `(0.01)`.

Assume that `computeTemp` works as specified, regardless of what you wrote in part (a).

Complete method `updateAllTemps` below.

```
/** Updates all values in temps and returns a boolean that indicates whether or not all the
 *  new values were within tolerance of the original values.
 *  @param tolerance a double value >= 0
 *  @return true if all updated temperatures are within tolerance of the original values;
 *          false otherwise.
 *  Postcondition: Each value in temps has been updated with a new value based on the
 *                 corresponding call to computeTemp.
 */
public boolean updateAllTemps(double tolerance)
```

GO ON TO THE NEXT PAGE.

4. A school district would like to get some statistics on its students' standardized test scores. Scores will be represented as objects of the following `ScoreInfo` class. Each `ScoreInfo` object contains a score value and the number of students who earned that score.

```
public class ScoreInfo
{
  private int score;
  private int numStudents;

  public ScoreInfo(int aScore)
  {
    score = aScore;
    numStudents = 1;
  }

  /** adds 1 to the number of students who earned this score
   */
  public void increment()
  {  numStudents++;  }

  /** @return this score
   */
  public int getScore()
  {  return score;  }

  /** @return the number of students who earned this score
   */
  public int getFrequency()
  {  return numStudents;  }
}
```

GO ON TO THE NEXT PAGE.

The following `Stats` class creates and maintains a database of student score information. The scores are stored in sorted order in the database.

```
public class Stats
{
  private ArrayList<ScoreInfo> scoreList;
    // listed in increasing score order; no two ScoreInfo objects contain the same score

  /** Records a score in the database, keeping the database in increasing score order. If no other
   *  ScoreInfo object represents score, a new ScoreInfo object representing score
   *  is added to the database; otherwise, the frequency in the ScoreInfo object representing
   *  score is incremented.
   *  @param score a score to be recorded in the list
   *  @return true if a new ScoreInfo object representing score was added to the list;
   *          false otherwise
   */
  public boolean record(int score)
  { /*  to be implemented in part (a)  */ }

  /** Records all scores in stuScores in the database, keeping the database in increasing score order
   *  @param stuScores an array of student test scores
   */
  public void recordScores(int[] stuScores)
  {  /*  to be implemented in part (b)  */  }

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

(a) Write the `Stats` method `record` that takes a test score and records that score in the database. If the score already exists in the database, the frequency of that score is updated. If the score does not exist in the database, a new `ScoreInfo` object is created and inserted in the appropriate position so that the database is maintained in increasing score order. The method returns `true` if a new `ScoreInfo` object was added to the database; otherwise, it returns `false`.

Complete method `record` below.

```
/** Records a score in the database, keeping the database in increasing score order. If no other
 *  ScoreInfo object represents score, a new ScoreInfo object representing score
 *  is added to the database; otherwise, the frequency in the ScoreInfo object representing
 *  score is incremented.
 *  @param score a score to be recorded in the list
 *  @return true if a new ScoreInfo object representing score was added to the list;
 *          false otherwise
 */
public boolean record(int score)
```

(b) Write the `Stats` method `recordScores` that takes an array of test scores and records them in the database. The database contains at most one `ScoreInfo` object per unique score value. Each `ScoreInfo` object contains a score and an associated frequency. The database is maintained in increasing order based on the score.

In writing `recordScores`, assume that `record` works as specified, regardless of what you wrote in part (a).

Complete method `recordScores` below.

```
/** Records all scores in stuScores in the database, keeping the database in increasing score order
 *  @param stuScores an array of student test scores
 */
public void recordScores(int[] stuScores)
```

STOP

END OF EXAM