4. The `Space` class is used to represent the spaces on a game board. A partial definition of the `Space` class is shown.

```
public class Space
{
   /**
    *  Returns the color of the space
    */
   public String getColor()
   { /* implementation not shown */ }

   /**
    *  Returns the point value of the space
    */
   public int getPoints()
   { /* implementation not shown */ }

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

The `GameBoard` class maintains a two-dimensional array of `Space` objects that represents a game board. A partial definition of the `GameBoard` class is shown.

```
public class GameBoard
{
   private Space[][] board;

   /**
    *  Returns the point value of the row in board specified by the
    *  parameter. The point value is the sum of the points in the row,
    *  or two times the sum of the points in the row if all spaces in
    *  the row are the same color.
    *  Preconditions: No elements of board are null.
    *                 board has at least two rows and
    *                 at least two columns.
    *                 targetRow is a valid row index.
    */
   public int getPointsForRow(int targetRow)
   { /* to be implemented */ }

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

When an element of the two-dimensional array `board` is accessed, the first index is used to specify the row and the second index is used to specify the column.

Write the `GameBoard` method `getPointsForRow`. The method should return the point value of the spaces in the row specified by the parameter `targetRow`. The point value of a row is determined as follows.

- If not all spaces in the row are the same color, the point value for the row is the sum of the points in the row.
- If all spaces in the row are the same color, the point value for the row is two times the sum of the points in the row.

For example, suppose `board` has the following contents. For each element, the first value is the color and the second value is the point value.

	0	1	2	3	4
0	"orange" 100	"red" 100	"blue" 500	"green" 500	"red" 100
1	"red" 300	"blue" 200	"blue" 400	"red" 100	"green" 100
2	"red" 200	"red" 300	"red" 100	"red" 200	"red" 200
3	"green" 100	"blue" 100	"blue" 200	"blue" 100	"blue" 200

For these contents of `board`, the expected behavior of `getPointsForRow` is as follows.

- The call `getPointsForRow(0)` should return `1300`. Not all spaces in this row are the same color, so the sum of the points in the row is returned.
- The call `getPointsForRow(2)` should return `2000`. The sum of the points in row 2 is `1000`, and all spaces in this row are the same color, so two times this sum is returned.

Complete method `getPointsForRow`.

```
/**
 *  Returns the point value of the row in board specified by the
 *  parameter. The point value is the sum of the points in the row,
 *  or two times the sum of the points in the row if all spaces in
 *  the row are the same color.
 *  Preconditions: No elements of board are null.
 *                 board has at least two rows and
 *                 at least two columns.
 *                 targetRow is a valid row index.
 */
public int getPointsForRow(int targetRow)
```

STOP

END OF EXAM

4. This question involves reasoning about a number puzzle that is represented as a two-dimensional array of integers. Each element of the array initially contains a value between 1 and 9, inclusive. Solving the puzzle involves clearing pairs of array elements by setting them to 0. Two elements can be cleared if their values sum to 10 or if they have the same value. The puzzle is considered solved if all elements of the array are cleared.

You will write the constructor and one method of the `SumOrSameGame` class, which contains the methods that manipulate elements of the puzzle.

```
public class SumOrSameGame
{
  private int[][] puzzle;

  /**
   * Creates a two-dimensional array and fills it with random integers,
   * as described in part (a)
   * Precondition: numRows > 0; numCols > 0
   */
  public SumOrSameGame(int numRows, int numCols)
  { /* to be implemented in part (a) */ }

  /**
   * Identifies and clears an element of puzzle that can be paired with
   * the element at the given row and column, as described in part (b)
   * Preconditions: row and col are valid row and column indices in puzzle.
   *  The element at the given row and column is between 1 and 9, inclusive.
   */
  public boolean clearPair(int row, int col)
  { /* to be implemented in part (b) */ }

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

A. Write the constructor for the `SumOrSameGame` class. The constructor initializes the instance variable `puzzle` to be a two-dimensional integer array with the number of rows and columns specified by the parameters `numRows` and `numCols`, respectively. Array elements are initialized with random integers between `1` and `9`, inclusive, each with an equal chance of being assigned to each element of `puzzle`.

When an element of the two-dimensional array is accessed, the first index is used to specify the row and the second index is used to specify the column.

Complete the `SumOrSameGame` constructor.

```
/**
 * Creates a two-dimensional array and fills it with random integers,
 * as described in part (a)
 * Precondition: numRows > 0; numCols > 0
 */
public SumOrSameGame(int numRows, int numCols)
```

B. Write the `clearPair` method, which takes a valid row index and valid column index as its parameters. The array element specified by those indices, which has a value between `1` and `9`, inclusive, is compared to other array elements in `puzzle` in an attempt to pair it with another array element that meets both of the following conditions.

- The row index of the second element is greater than or equal to the parameter `row`.
- The two elements have equal values or have values that sum to 10.

If such an array element is found, both array elements of the pair are cleared (set to `0`) and the method returns `true`. If more than one such array element is found, any one of those identified array elements can be used to complete the pair and can be cleared. If no such array element is found, no changes are made to `puzzle` and the method returns `false`.

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

`puzzle` before call to `clearPair`	Method Call	`puzzle` after call to `clearPair`	Return Value	Explanation

`0 7 9 0`
`7 4 1 6`
`8 4 1 8`
	`clearPair(0, 1)`	`0 0 9 0`
		`0 4 1 6`
		`8 4 1 8`
			`true`	The value `7` in row `0`, column `1` is matched with the value `7` in row `1`, column `0`.

`1 2 3 4`
`5 6 7 8`
`5 4 1 2`
	`clearPair(2, 2)`	`1 2 3 4`
		`5 6 7 8`
		`5 4 1 2`
			`false`	There is no element in row `2` or a later row to pair with the value `1`.

`8 1 0 5`
`0 4 3 6`
`3 4 5 8`
	`clearPair(1, 1)`	`8 1 0 5`
		`0 0 3 0`
		`3 4 5 8`
			`true`	The value `4` in row `1`, column `1` is matched with the value `6` in row `1`, column `3`. It could also have been matched with the value `4` in row `2`, column `1`.

`1 7 9`
`2 6 5`
`4 4 4`
	`clearPair(0, 2)`	`0 7 0`
		`2 6 5`
		`4 4 4`
			`true`	The value `9` in row `0`, column `2` is matched with the value `1` in row `0`, column `0`.

Complete method `clearPair`.

```
/**
 * Identifies and clears an element of puzzle that can be paired with
 * the element at the given row and column, as described in part (b)
 * Preconditions: row and col are valid row and column indices in puzzle.
 *  The element at the given row and column is between 1 and 9, inclusive.
 */
public boolean clearPair(int row, int col)
```

STOP

END OF EXAM

4. This question involves a path through a two-dimensional (2D) array of integers, where the path is based on the values of elements in the array. When an element of the 2D array is accessed, the first index is used to specify the row and the second index is used to specify the column. The following `Location` class represents a row and column position in the 2D array.

```
public class Location
{
   private int theRow;
   private int theCol;

   public Location(int r, int c)
   {
      theRow = r;
      theCol = c;
   }

   public int getRow()
   {  return theRow;  }

   public int getCol()
   {  return theCol;  }
}
```

The following `GridPath` class contains the 2D array and methods to use to determine a path through the array. You will write two methods of the `GridPath` class.

```
public class GridPath
{
    /** Initialized in the constructor with distinct values that never change */
    private int[][] grid;

    /**
     *  Returns the Location representing a neighbor of the grid element at row and col,
     *  as described in part (a)
     *  Preconditions: row is a valid row index and col is a valid column index in grid.
     *       row and col do not specify the element in the last row and last column of grid.
     */
    public Location getNextLoc(int row, int col)
    {  /* to be implemented in part (a) */  }

    /**
     *  Computes and returns the sum of all values on a path through grid, as described in
     *  part (b)
     *  Preconditions: row is a valid row index and col is a valid column index in grid.
     *       row and col do not specify the element in the last row and last column of grid.
     */
    public int sumPath(int row, int col)
    {  /* to be implemented in part (b) */  }

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

(a) Write the `getNextLoc` method, which returns a `Location` object that represents the smaller of two neighbors of the `grid` element at `row` and `col`, according to the following rules.

- The two neighbors that are considered are the element below the given element and the element to the right of the given element, if they exist.
- If both neighbors exist, the `Location` of the neighbor with the smaller value is returned. Two neighbors will always have different values.
- If only one neighbor exists, the `Location` of the existing neighbor is returned.

For example, assume that `grid` contains the following values.

	0	1	2	3	4
0	12	3	4	13	5
1	11	21	2	14	16
2	7	8	9	15	0
3	10	17	20	19	1
4	18	22	30	25	6

The following table shows some sample calls to `getNextLoc`.

Method Call	Explanation
`getNextLoc(0, 0)`	Returns the neighbor to the right (the `Location` representing the element at row 0 and column 1), since `3 < 11`
`getNextLoc(1, 3)`	Returns the neighbor below (the `Location` representing the element at row 2 and column 3), since `15 < 16`
`getNextLoc(2, 4)`	Returns the neighbor below (the `Location` representing the element at row 3 and column 4), since the given element has no neighbor to the right
`getNextLoc(4, 3)`	Returns the neighbor to the right (the `Location` representing the element at row 4 and column 4), since the given element has no neighbor below

In the example, the `getNextLoc` method will never be called with row `4` and column `4`, as those values would violate the precondition of the method.

Complete the `getNextLoc` method.

```
/**
 *  Returns the Location representing a neighbor of the grid element at row and col,
 *  as described in part (a)
 *  Preconditions: row is a valid row index and col is a valid column index in grid.
 *       row and col do not specify the element in the last row and last column of grid.
 */
public Location getNextLoc(int row, int col)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Location

private int theRow
private int theCol

public Location(int r, int c)
public int getRow()
public int getCol()

public class GridPath

private int[][] grid

public Location getNextLoc(int row, int col)
public int sumPath(int row, int col)
```

(b) Write the `sumPath` method, which returns the sum of all values on a path in `grid`. The path begins with the element at `row` and `col` and is determined by successive calls to `getNextLoc`. The path ends when the element in the last row and the last column of `grid` is reached.

For example, consider the following contents of `grid`. The shaded elements of `grid` represent the values on the path that results from the method call `sumPath(1, 1)`. The method call returns `19` because 3 + 2 + 9 + 4 + 0 + 1 = 19.

	0	1	2	3	4
0	12	30	40	25	5
1	11	3	22	15	43
2	7	2	9	4	0
3	8	33	18	6	1

Write the `sumPath` method. Assume `getNextLoc` works as intended, regardless of what you wrote in part (a). You must use `getNextLoc` appropriately in order to receive full credit.

```
/**
 * Computes and returns the sum of all values on a path through grid, as described in
 * part (b)
 * Preconditions: row is a valid row index and col is a valid column index in grid.
 *      row and col do not specify the element in the last row and last column of grid.
 */
public int sumPath(int row, int col)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Location

private int theRow
private int theCol

public Location(int r, int c)
public int getRow()
public int getCol()

public class GridPath

private int[][] grid

public Location getNextLoc(int row, int col)
public int sumPath(int row, int col)
```

STOP

END OF EXAM

4. This question involves pieces of candy in a box. The `Candy` class represents a single piece of candy.

```
public class Candy
{
    /** Returns a String representing the flavor of this piece of candy */
    public String getFlavor()
    {  /* implementation not shown */  }

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

The `BoxOfCandy` class represents a candy box where the candy is arranged in a rectangular grid. The instance variable of the class, `box`, is a rectangular two-dimensional array of `Candy` objects. A location in the candy box may contain a piece of candy or may be empty. A piece of candy is represented by a `Candy` object. An empty location is represented by `null`.

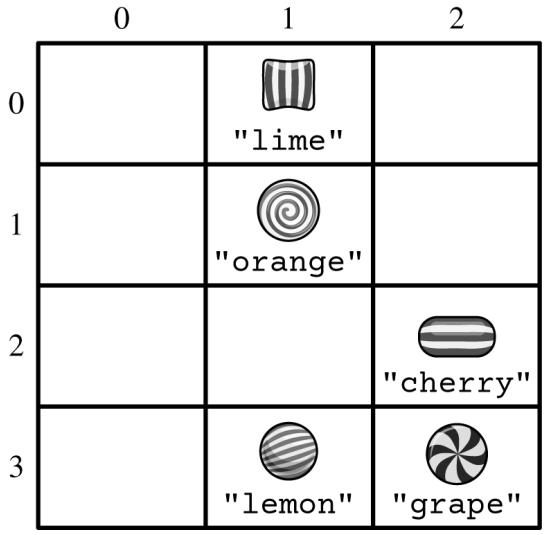
You will write two methods of the `BoxOfCandy` class.

```
public class BoxOfCandy
{
    /** box contains at least one row and is initialized in the constructor. */
    private Candy[][] box;

    /**
     *  Moves one piece of candy in column col, if necessary and possible, so that the box
     *  element in row 0 of column col contains a piece of candy, as described in part (a).
     *  Returns false if there is no piece of candy in column col and returns true otherwise.
     *  Precondition: col is a valid column index in box.
     */
    public boolean moveCandyToFirstRow(int col)
    {  /* to be implemented in part (a) */  }

    /**
     *  Removes from box and returns a piece of candy with flavor specified by the parameter, or
     *  returns null if no such piece is found, as described in part (b)
     */
    public Candy removeNextByFlavor(String flavor)
    {  /* to be implemented in part (b) */  }

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

(a) Write the `moveCandyToFirstRow` method, which attempts to ensure that the `box` element at row `0` and column `col` contains a piece of candy, using the following steps.

- If the element at row `0` and column `col` already contains a piece of candy, then `box` is unchanged and the method returns `true`.
- If the element at row `0` and column `col` does not contain a piece of candy, then the method searches the remaining rows of column `col` for a piece of candy. If a piece of candy can be found in column `col`, it is moved to row `0`, its previous location is set to `null`, and the method returns `true`; otherwise, the method returns `false`.

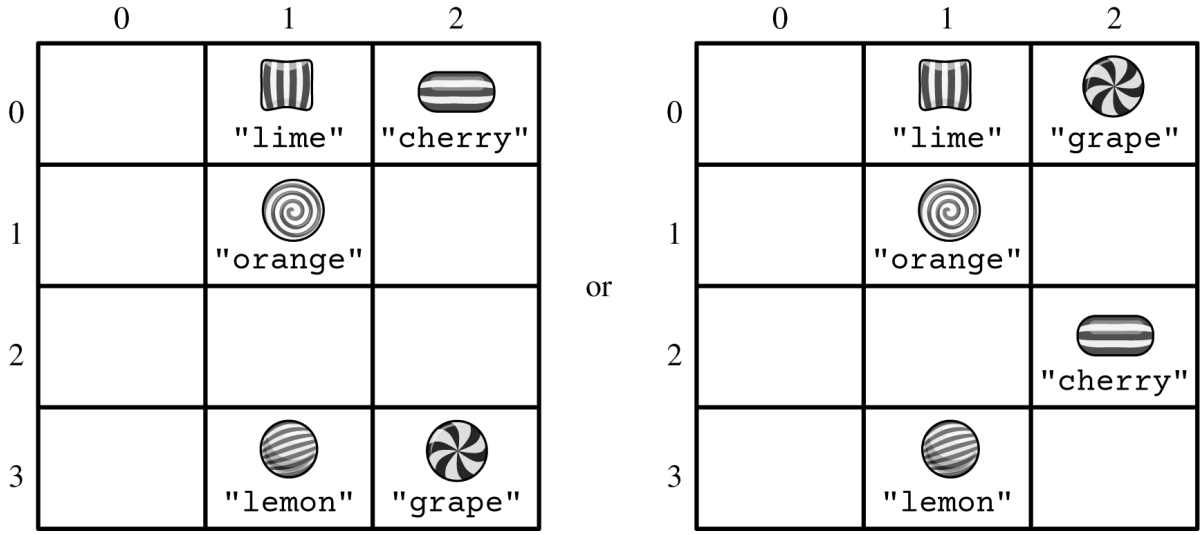
In the following example, the grid represents the contents of `box`. An empty square in the grid is `null` in `box`. A non-empty square in the grid represents a `box` element that contains a `Candy` object. The string in the square of the grid indicates the flavor of the piece of candy.



The method call `moveCandyToFirstRow(0)` returns `false` because the `box` element at row `0` and column `0` does not contain a piece of candy and there are no pieces of candy in column `0` that can be moved to row `0`. The contents of `box` are unchanged.

The method call `moveCandyToFirstRow(1)` returns `true` because the `box` element at row `0` and column `1` already contains a piece of candy. The contents of `box` are unchanged.

The method call `moveCandyToFirstRow(2)` moves one of the two pieces of candy in column `2` to row `0` of column `2`, sets the previous location of the piece of candy that was moved to `null`, and returns `true`. The new contents of `box` could be either of the following.



Complete the `moveCandyToFirstRow` method.

```
/**
 * Moves one piece of candy in column col, if necessary and possible, so that the box
 * element in row 0 of column col contains a piece of candy, as described in part (a).
 * Returns false if there is no piece of candy in column col and returns true otherwise.
 * Precondition: col is a valid column index in box.
 */
public boolean moveCandyToFirstRow(int col)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Candy
public String getFlavor()
public class BoxOfCandy
private Candy[][] box
public boolean moveCandyToFirstRow(int col)
public Candy removeNextByFlavor(String flavor)
```

(b) Write the `removeNextByFlavor` method, which attempts to remove and return one piece of candy from the box. The piece of candy to be removed is the first piece of candy with a flavor equal to the parameter `flavor` that is encountered while traversing the candy box in the following order: the last row of the box is traversed from left to right, then the next-to-last row of the box is traversed from left to right, etc., until either a piece of candy with the desired flavor is found or until the entire candy box has been searched.

If the `removeNextByFlavor` method finds a `Candy` object with the desired flavor, the corresponding `box` element is assigned `null`, all other `box` elements are unchanged, and the removed `Candy` object is returned. Otherwise, `box` is unchanged and the method returns `null`.

The following examples show three consecutive calls to the `removeNextByFlavor` method. The traversal of the candy box always begins in the last row and first column of the box.

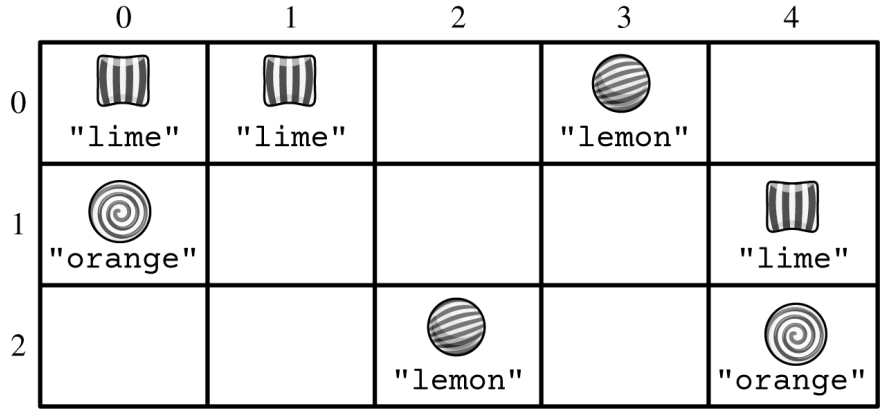
The following grid shows the contents of `box` before any of the `removeNextByFlavor` method calls.

	0	1	2	3	4
0	 "lime"	 "lime"		 "lemon"	
1	 "orange"			 "lime"	 "lime"
2	 "cherry"		 "lemon"		 "orange"

The method call `removeNextByFlavor("cherry")` removes and returns the `Candy` object located in row `2` and column `0`. The following grid shows the updated contents of `box`.

	0	1	2	3	4
0	 "lime"	 "lime"		 "lemon"	
1	 "orange"			 "lime"	 "lime"
2			 "lemon"		 "orange"

The method call `removeNextByFlavor("lime")` removes and returns the `Candy` object located in row `1` and column `3`. The following grid shows the updated contents of `box`.



The method call `removeNextByFlavor("grape")` returns `null` because no grape-flavored candy is found. The contents of `box` are unchanged.

Complete the `removeNextByFlavor` method.

```
/**
 *  Removes from box and returns a piece of candy with flavor specified by the parameter, or
 *  returns null if no such piece is found, as described in part (b)
 */
public Candy removeNextByFlavor(String flavor)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Candy

public String getFlavor()

public class BoxOfCandy

private Candy[][] box

public boolean moveCandyToFirstRow(int col)
public Candy removeNextByFlavor(String flavor)
```

STOP

END OF EXAM

4. This question involves a two-dimensional array of integers that represents a collection of randomly generated data. A partial declaration of the `Data` class is shown. You will write two methods of the `Data` class.

```
public class Data
{
   public static final int MAX = /* value not shown */;
   private int[][] grid;

   /** Fills all elements of grid with randomly generated values, as described in part (a)
    *  Precondition: grid is not null.
    *       grid has at least one element.
    */
   public void repopulate()
   {  /* to be implemented in part (a) */  }

   /** Returns the number of columns in grid that are in increasing order, as described
    *  in part (b)
    *  Precondition: grid is not null.
    *       grid has at least one element.
    */
   public int countIncreasingCols()
   {  /* 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 `repopulate` method, which assigns a newly generated random value to each element of `grid`. Each value is computed to meet all of the following criteria, and all valid values must have an equal chance of being generated.

- The value is between 1 and `MAX`, inclusive.
- The value is divisible by 10.
- The value is not divisible by 100.

Complete the `repopulate` method.

```
/** Fills all elements of grid with randomly generated values, as described in part (a)
 *  Precondition: grid is not null.
 *      grid has at least one element.
 */
public void repopulate()
```

Begin your response at the top of a new page in the Free Response booklet and fill in the appropriate circle indicating the question number.
If there are multiple parts to this question, write the part letter with your response.

GO ON TO THE NEXT PAGE.

(b) Write the `countIncreasingCols` method, which returns the number of columns in `grid` that are in increasing order. A column is considered to be in increasing order if the element in each row after the first row is greater than or equal to the element in the previous row. A column with only one row is considered to be in increasing order.

The following examples show the `countIncreasingCols` return values for possible contents of `grid`.

The return value for the following contents of `grid` is `1`, since the first column is in increasing order but the second and third columns are not.

10	50	40
20	40	20
30	50	30

The return value for the following contents of `grid` is `2`, since the first and third columns are in increasing order but the second and fourth columns are not.

10	540	440	440
220	450	440	190

Complete the `countIncreasingCols` method.

```
/** Returns the number of columns in grid that are in increasing order, as described
 *  in part (b)
 *  Precondition: grid is not null.
 *       grid has at least one element.
 */
public int countIncreasingCols()
```

Begin your response at the top of a new page in the Free Response booklet and fill in the appropriate circle indicating the question number.
If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Data

public static final int MAX = /* value not shown */
private int[][] grid

public void repopulate()
public int countIncreasingCols()
```

GO ON TO THE NEXT PAGE.

STOP

END OF EXAM

4. This question involves manipulating a two-dimensional array of integers. You will write two `static` methods of the `ArrayResizer` class, which is shown below.

```
public class ArrayResizer
{
    /** Returns true if and only if every value in row r of array2D is non-zero.
     *  Precondition: r is a valid row index in array2D.
     *  Postcondition: array2D is unchanged.
     */
    public static boolean isNonZeroRow(int[][] array2D, int r)
    {  /* to be implemented in part (a) */  }

    /** Returns the number of rows in array2D that contain all non-zero values.
     *  Postcondition: array2D is unchanged.
     */
    public static int numNonZeroRows(int[][] array2D)
    {  /* implementation not shown */  }

    /** Returns a new, possibly smaller, two-dimensional array that contains only rows
     *  from array2D with no zeros, as described in part (b).
     *  Precondition: array2D contains at least one column and at least one row with no zeros.
     *  Postcondition: array2D is unchanged.
     */
    public static int[][] resize(int[][] array2D)
    {  /* to be implemented in part (b) */  }
}
```

(a) Write the method `isNonZeroRow`, which returns `true` if and only if all elements in row `r` of a two-dimensional array `array2D` are not equal to zero.

For example, consider the following statement, which initializes a two-dimensional array.

```
int[][] arr = {{2, 1, 0},
               {1, 3, 2},
               {0, 0, 0},
               {4, 5, 6}};
```

Sample calls to `isNonZeroRow` are shown below.

Call to `isNonZeroRow`	**Value Returned**	**Explanation**
`ArrayResizer.isNonZeroRow(arr, 0)`	`false`	At least one value in row `0` is zero.
`ArrayResizer.isNonZeroRow(arr, 1)`	`true`	All values in row `1` are non-zero.
`ArrayResizer.isNonZeroRow(arr, 2)`	`false`	At least one value in row `2` is zero.
`ArrayResizer.isNonZeroRow(arr, 3)`	`true`	All values in row `3` are non-zero.

Complete the `isNonZeroRow` method.

```
/** Returns true if and only if every value in row r of array2D is non-zero.
 *  Precondition: r is a valid row index in array2D.
 *  Postcondition: array2D is unchanged.
 */
public static boolean isNonZeroRow(int[][] array2D, int r)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

(b) Write the method `resize`, which returns a new two-dimensional array containing only rows from `array2D` with all non-zero values. The elements in the new array should appear in the same order as the order in which they appeared in the original array.

The following code segment initializes a two-dimensional array and calls the `resize` method.

```
int[][] arr = {{2, 1, 0},
               {1, 3, 2},
               {0, 0, 0},
               {4, 5, 6}};
int[][] smaller = ArrayResizer.resize(arr);
```

When the code segment completes, the following will be the contents of `smaller`.

```
{{1, 3, 2}, {4, 5, 6}}
```

A helper method, `numNonZeroRows`, has been provided for you. The method returns the number of rows in its two-dimensional array parameter that contain no zero values.

Complete the `resize` method. Assume that `isNonZeroRow` works as specified, regardless of what you wrote in part (a). You must use `numNonZeroRows` and `isNonZeroRow` appropriately to receive full credit.

```
/** Returns a new, possibly smaller, two-dimensional array that contains only rows from array2D
 *  with no zeros, as described in part (b).
 *  Precondition: array2D contains at least one column and at least one row with no zeros.
 *  Postcondition: array2D is unchanged.
 */
public static int[][] resize(int[][] array2D)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class ArrayResizer

public static boolean isNonZeroRow(int[][] array2D, int r)
public static int numNonZeroRows(int[][] array2D)
public static int[][] resize(int[][] array2D)
```

STOP

END OF EXAM

4. The `LightBoard` class models a two-dimensional display of lights, where each light is either on or off, as represented by a Boolean value. You will implement a constructor to initialize the display and a method to evaluate a light.

```
public class LightBoard
{
    /** The lights on the board, where true represents on and false represents off.
     */
    private boolean[][] lights;

    /** Constructs a LightBoard object having numRows rows and numCols columns.
     *  Precondition: numRows > 0, numCols > 0
     *  Postcondition: each light has a 40% probability of being set to on.
     */
    public LightBoard(int numRows, int numCols)
    {  /* to be implemented in part (a) */  }

    /** Evaluates a light in row index row and column index col and returns a status
     *  as described in part (b).
     *  Precondition: row and col are valid indexes in lights.
     */
    public boolean evaluateLight(int row, int col)
    {  /* to be implemented in part (b) */  }

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

}
```

(a) Write the constructor for the `LightBoard` class, which initializes `lights` so that each light is set to on with a 40% probability. The notation `lights[r][c]` represents the array element at row `r` and column `c`.

Complete the `LightBoard` constructor below.

```
/** Constructs a LightBoard object having numRows rows and numCols columns.
 *  Precondition: numRows > 0, numCols > 0
 *  Postcondition: each light has a 40% probability of being set to on.
 */
public LightBoard(int numRows, int numCols)
```

(b) Write the method `evaluateLight`, which computes and returns the status of a light at a given row and column based on the following rules.

1. If the light is on, return `false` if the number of lights in its column that are on is even, including the current light.
2. If the light is off, return `true` if the number of lights in its column that are on is divisible by three.
3. Otherwise, return the light's current status.

For example, suppose that `LightBoard sim = new LightBoard(7, 5)` creates a light board with the initial state shown below, where `true` represents a light that is on and `false` represents a light that is off. Lights that are off are shaded.

`lights`

	0	1	2	3	4
0	`true`	`true`	`false`	`true`	`true`
1	`true`	`false`	`false`	`true`	`false`
2	`true`	`false`	`false`	`true`	`true`
3	`true`	`false`	`false`	`false`	`true`
4	`true`	`false`	`false`	`false`	`true`
5	`true`	`true`	`false`	`true`	`true`
6	`false`	`false`	`false`	`false`	`false`

Sample calls to `evaluateLight` are shown below.

Call to `evaluateLight`	**Value Returned**	**Explanation**
`sim.evaluateLight(0, 3);`	`false`	The light is on, and the number of lights that are on in its column is even.
`sim.evaluateLight(6, 0);`	`true`	The light is off, and the number of lights that are on in its column is divisible by 3.
`sim.evaluateLight(4, 1);`	`false`	Returns the light's current status.
`sim.evaluateLight(5, 4);`	`true`	Returns the light's current status.

Class information for this question

```
public class LightBoard
private boolean[][] lights
public LightBoard(int numRows, int numCols)
public boolean evaluateLight(int row, int col)
```

Complete the `evaluateLight` method below.

```
/** Evaluates a light in row index row and column index col and returns a status
 *  as described in part (b).
 *  Precondition: row and col are valid indexes in lights.
 */
public boolean evaluateLight(int row, int col)
```

STOP

END OF EXAM

4. This question involves reasoning about arrays of integers. You will write two static methods, both of which are in a class named `ArrayTester`.

```
public class ArrayTester
{

    /** Returns an array containing the elements of column c of arr2D in the same order as
     *  they appear in arr2D.
     *  Precondition: c is a valid column index in arr2D.
     *  Postcondition: arr2D is unchanged.
     */
    public static int[] getColumn(int[][] arr2D, int c)
    {  /* to be implemented in part (a) */  }


    /** Returns true if and only if every value in arr1 appears in arr2.
     *  Precondition: arr1 and arr2 have the same length.
     *  Postcondition: arr1 and arr2 are unchanged.
     */
    public static boolean hasAllValues(int[] arr1, int[] arr2)
    {  /* implementation not shown */  }


    /** Returns true if arr contains any duplicate values;
     *          false otherwise.
     */
    public static boolean containsDuplicates(int[] arr)
    {  /* implementation not shown */  }


    /** Returns true if square is a Latin square as described in part (b);
     *          false otherwise.
     *  Precondition: square has an equal number of rows and columns.
     *                square has at least one row.
     */
    public static boolean isLatin(int[][] square)
    {  /* to be implemented in part (b) */  }
```

(a) Write a static method `getColumn`, which returns a one-dimensional array containing the elements of a single column in a two-dimensional array. The elements in the returned array should be in the same order as they appear in the given column. The notation `arr2D[r][c]` represents the array element at row `r` and column `c`.

The following code segment initializes an array and calls the `getColumn` method.

```
int[][] arr2D = { { 0, 1, 2 },
                  { 3, 4, 5 },
                  { 6, 7, 8 },
                  { 9, 5, 3 } };

int[] result = ArrayTester.getColumn(arr2D, 1);
```

When the code segment has completed execution, the variable `result` will have the following contents.

```
result: {1, 4, 7, 5}
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `getColumn` below.

```
/** Returns an array containing the elements of column c of arr2D in the same order as they
 *  appear in arr2D
```

(b) Write the static method `isLatin`, which returns `true` if a given two-dimensional square array is a *Latin square*, and otherwise, returns `false`.

A two-dimensional square array of integers is a Latin square if the following conditions are true.

- The first row has no duplicate values.
- All values in the first row of the square appear in each row of the square.
- All values in the first row of the square appear in each column of the square.

Examples of Latin Squares

1	2	3
2	3	1
3	1	2

10	30	20	0
0	20	30	10
30	0	10	20
20	10	0	30

Examples that are NOT Latin Squares

1	2	1
2	1	1
1	1	2

Not a Latin square because the first row contains duplicate values

1	2	3
3	1	2
7	8	9

Not a Latin square because the elements of the first row do not all appear in the third row

1	2
1	2

Not a Latin square because the elements of the first row do not all appear in either column

The `ArrayTester` class provides two helper methods: `containsDuplicates` and `hasAllValues`. The method `containsDuplicates` returns `true` if the given one-dimensional array `arr` contains any duplicate values and `false` otherwise. The method `hasAllValues` returns `true` if and only if every value in `arr1` appears in `arr2`. You do not need to write the code for these methods.

Class information for this question

```
public class ArrayTester

public static int[] getColumn(int[][] arr2D, int c)
public static boolean hasAllValues(int[] arr1, int[] arr2)
public static boolean containsDuplicates(int[] arr)
public static boolean isLatin(int[][] square)
```

Complete method `isLatin` below. Assume that `getColumn` works as specified, regardless of what you wrote in part (a). You must use `getColumn`, `hasAllValues`, and `containsDuplicates` appropriately to receive full credit.

```
/** Returns true if square is a Latin square as described in part (b);
 *          false otherwise.
 *  Precondition: square has an equal number of rows and columns.
 *                square has at least one row.
 */
public static boolean isLatin(int[][] square)
```

STOP

END OF EXAM

4. This question involves reasoning about a two-dimensional (2D) array of integers. You will write two static methods, both of which are in a single enclosing class named `Successors` (not shown). These methods process a 2D integer array that contains consecutive values. Each of these integers may be in any position in the 2D integer array. For example, the following 2D integer array with 3 rows and 4 columns contains the integers 5 through 16, inclusive.

2D Integer Array

	0	1	2	3
0	15	5	9	10
1	12	16	11	6
2	14	8	13	7

The following `Position` class is used to represent positions in the integer array. The notation `(r,c)` will be used to refer to a `Position` object with row `r` and column `c`.

```
public class Position
{
    /** Constructs a Position object with row r and column c. */
    public Position(int r, int c)
    {  /* implementation not shown */  }

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

(a) Write a `static` method `findPosition` that takes an integer value and a 2D integer array and returns the position of the integer in the given 2D integer array. If the integer is not an element of the 2D integer array, the method returns `null`.

For example, assume that array `arr` is the 2D integer array shown at the beginning of the question.

- The call `findPosition(8, arr)` would return the `Position` object `(2,1)` because the value `8` appears in `arr` at row 2 and column 1.
- The call `findPosition(17, arr)` would return `null` because the value `17` does not appear in `arr`.

Complete method `findPosition` below.

```
/** Returns the position of num in intArr;
 *  returns null if no such element exists in intArr.
 *  Precondition: intArr contains at least one row.
 */
public static Position findPosition(int num, int[][] intArr)
```

Part (b) begins on page 18.

(b) Write a `static` method `getSuccessorArray` that returns a 2D successor array of positions created from a given 2D integer array.

The *successor* of an integer value is the integer that is one greater than that value. For example, the successor of 8 is 9. A 2D *successor array* shows the position of the successor of each element in a given 2D integer array. The 2D successor array has the same dimensions as the given 2D integer array. Each element in the 2D successor array is the position (row, column) of the corresponding 2D integer array element's successor. The largest element in the 2D integer array does not have a successor in the 2D integer array, so its corresponding position in the 2D successor array is `null`.

The following diagram shows a 2D integer array and its corresponding 2D successor array. To illustrate the successor relationship, the values 8 and 9 in the 2D integer array are shaded. In the 2D successor array, the shaded element shows that the position of the successor of 8 is `(0,2)` in the 2D integer array. The largest value in the 2D integer array is 16, so its corresponding element in the 2D successor array is `null`.

2D Integer Array

	0	1	2	3
0	15	5	9	10
1	12	16	11	6
2	14	8	13	7

2D Successor Array

	0	1	2	3
0	(1,1)	(1,3)	(0,3)	(1,2)
1	(2,2)	null	(1,0)	(2,3)
2	(0,0)	(0,2)	(2,0)	(2,1)

Class information for this question

```
public class Position

public Position(int r, int c)

public class Successors

public static Position findPosition(int num, int[][] intArr)
public static Position[][] getSuccessorArray(int[][] intArr)
```

Assume that `findPosition` works as specified, regardless of what you wrote in part (a). You must use `findPosition` appropriately to receive full credit.

Complete method `getSuccessorArray` below.

```
/** Returns a 2D successor array as described in part (b) constructed from intArr.
 *  Precondition: intArr contains at least one row and contains consecutive values.
 *                Each of these integers may be in any position in the 2D array.
 */
public static Position[][] getSuccessorArray(int[][] intArr)
```

STOP

END OF EXAM

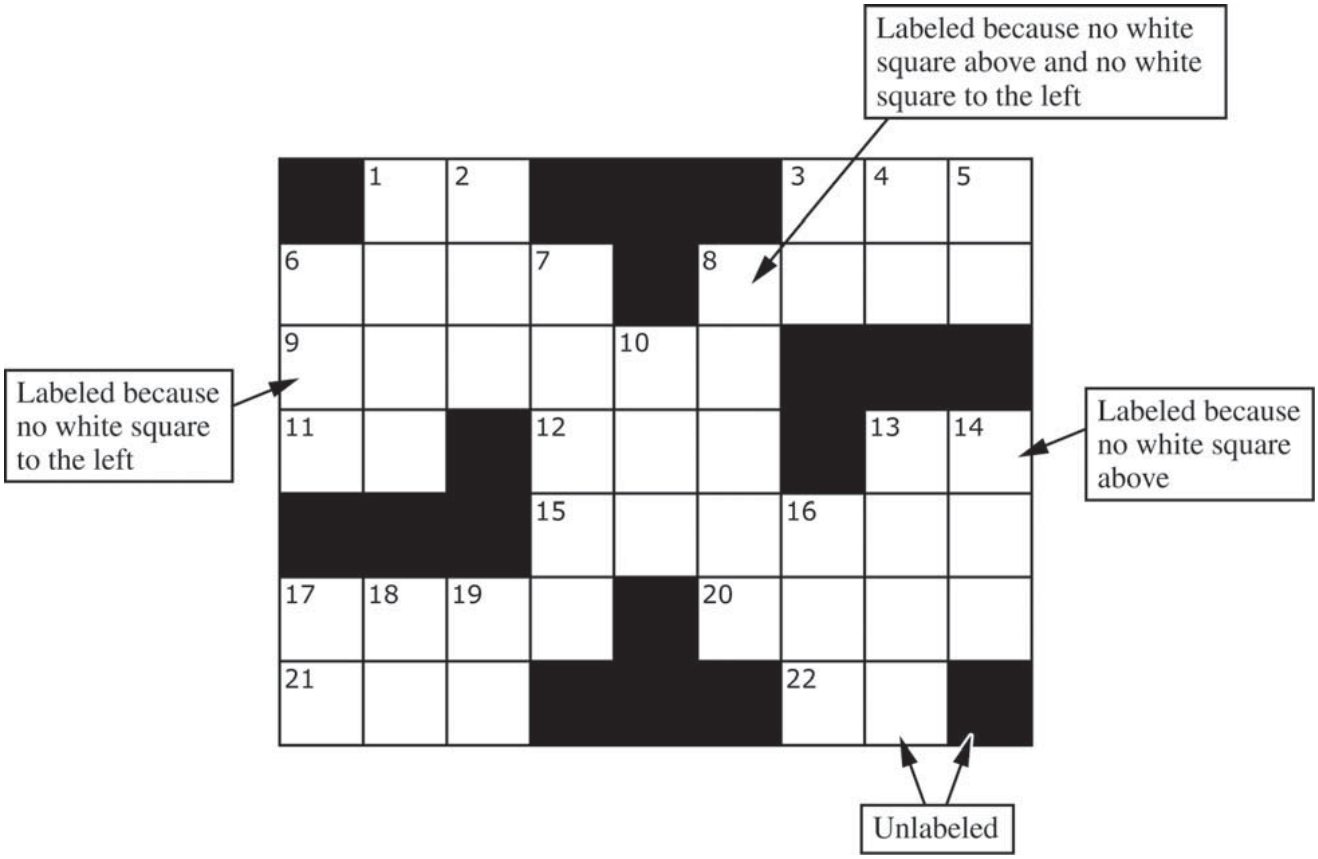
3. A crossword puzzle grid is a two-dimensional rectangular array of black and white squares. Some of the white squares are labeled with a positive number according to the *crossword labeling rule*.

   The crossword labeling rule identifies squares to be labeled with a positive number as follows.

   A square is labeled with a positive number if and only if
   - the square is white and
   - the square does not have a white square immediately above it, or it does not have a white square immediately to its left, or both.

   The squares identified by these criteria are labeled with consecutive numbers in row-major order, starting at 1.

   The following diagram shows a crossword puzzle grid and the labeling of the squares according to the crossword labeling rule.



This question uses two classes, a `Square` class that represents an individual square in the puzzle and a `Crossword` class that represents a crossword puzzle grid. A partial declaration of the `Square` class is shown below.

```
public class Square
{
    /** Constructs one square of a crossword puzzle grid.
     *  Postcondition:
     *    - The square is black if and only if isBlack is true.
     *    - The square has number num.
     */
    public Square(boolean isBlack, int num)
    {  /* implementation not shown */  }

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

A partial declaration of the `Crossword` class is shown below. You will implement one method and the constructor in the `Crossword` class.

```
public class Crossword
{
    /** Each element is a Square object with a color (black or white) and a number.
     *  puzzle[r][c] represents the square in row r, column c.
     *  There is at least one row in the puzzle.
     */
    private Square[][] puzzle;

    /** Constructs a crossword puzzle grid.
     *  Precondition: There is at least one row in blackSquares.
     *  Postcondition:
     *    - The crossword puzzle grid has the same dimensions as blackSquares.
     *    - The Square object at row r, column c in the crossword puzzle grid is black
     *      if and only if blackSquares[r][c] is true.
     *    - The squares in the puzzle are labeled according to the crossword labeling rule.
     */
    public Crossword(boolean[][] blackSquares)
    {  /* to be implemented in part (b) */  }

    /** Returns true if the square at row r, column c should be labeled with a positive number;
     *          false otherwise.
     *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
     *  Precondition: r and c are valid indexes in blackSquares.
     */
    private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
    {  /* to be implemented in part (a) */  }

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

Part (a) begins on page 14.

(a) Write the `Crossword` method `toBeLabeled`. The method returns `true` if the square indexed by row `r`, column `c` in a crossword puzzle grid should be labeled with a positive number according to the crossword labeling rule; otherwise it returns `false`. The parameter `blackSquares` indicates which squares in the crossword puzzle grid are black.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `toBeLabeled` below.

```
/** Returns true if the square at row r, column c should be labeled with a positive number;
 *          false otherwise.
 *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
 *  Precondition: r and c are valid indexes in blackSquares.
 */
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

Part (b) begins on page 16.

(b) Write the `Crossword` constructor. The constructor should initialize the crossword puzzle grid to have the same dimensions as the parameter `blackSquares`. Each element of the puzzle grid should be initialized with a reference to a `Square` object with the appropriate color and number. The number is positive if the square is labeled and 0 if the square is not labeled.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `toBeLabeled` works as specified, regardless of what you wrote in part (a). You must use `toBeLabeled` appropriately to receive full credit.

Complete the `Crossword` constructor below.

```
/** Constructs a crossword puzzle grid.
 *  Precondition: There is at least one row in blackSquares.
 *  Postcondition:
 *    – The crossword puzzle grid has the same dimensions as blackSquares.
 *    – The Square object at row r, column c in the crossword puzzle grid is black
 *      if and only if blackSquares[r][c] is true.
 *    – The squares in the puzzle are labeled according to the crossword labeling rule.
 */
public Crossword(boolean[][] blackSquares)
```

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 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. This question involves reasoning about one-dimensional and two-dimensional arrays of integers. You will write three static methods, all of which are in a single enclosing class, named `DiverseArray` (not shown). The first method returns the sum of the values of a one-dimensional array; the second method returns an array that represents the sums of the rows of a two-dimensional array; and the third method analyzes row sums.

   (a) Write a `static` method `arraySum` that calculates and returns the sum of the entries in a specified one-dimensional array. The following example shows an array `arr1` and the value returned by a call to `arraySum`.

	`arr1`				Value returned by `arraySum(arr1)`
0	1	2	3	4	
1	3	2	7	3	16

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `arraySum` below.

```
/** Returns the sum of the entries in the one-dimensional array arr.
 */
public static int arraySum(int[] arr)
```

Part (b) begins on page 4.

(b) Write a `static` method `rowSums` that calculates the sums of each of the rows in a given two-dimensional array and returns these sums in a one-dimensional array. The method has one parameter, a two-dimensional array `arr2D` of `int` values. The array is in row-major order: `arr2D[r][c]` is the entry at row `r` and column `c`. The method returns a one-dimensional array with one entry for each row of `arr2D` such that each entry is the sum of the corresponding row in `arr2D`. As a reminder, each row of a two-dimensional array is a one-dimensional array.

For example, if `mat1` is the array represented by the following table, the call `rowSums(mat1)` returns the array `{16, 32, 28, 20}`.

`mat1`

	0	1	2	3	4
0	1	3	2	7	3
1	10	10	4	6	2
2	5	3	5	9	6
3	7	6	4	2	1

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `arraySum` works as specified, regardless of what you wrote in part (a). You must use `arraySum` appropriately to receive full credit.

Complete method `rowSums` below.

```
/** Returns a one-dimensional array in which the entry at index k is the sum of
 *  the entries of row k of the two-dimensional array arr2D.
 */
public static int[] rowSums(int[][] arr2D)
```

Part (c) begins on page 6.

(c) A two-dimensional array is *diverse* if no two of its rows have entries that sum to the same value. In the following examples, the array `mat1` is diverse because each row sum is different, but the array `mat2` is not diverse because the first and last rows have the same sum.

`mat1`

	0	1	2	3	4	Row sums
0	1	3	2	7	3	16
1	10	10	4	6	2	32
2	5	3	5	9	6	28
3	7	6	4	2	1	20

`mat2`

	0	1	2	3	4	Row sums
0	1	1	5	3	4	14
1	12	7	6	1	9	35
2	8	11	10	2	5	36
3	3	2	3	0	6	14

Write a `static` method `isDiverse` that determines whether or not a given two-dimensional array is diverse. The method has one parameter: a two-dimensional array `arr2D` of `int` values. The method should return `true` if all the row sums in the given array are unique; otherwise, it should return `false`. In the arrays shown above, the call `isDiverse(mat1)` returns `true` and the call `isDiverse(mat2)` returns `false`.

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

Assume that `arraySum` and `rowSums` work as specified, regardless of what you wrote in parts (a) and (b). You must use `rowSums` appropriately to receive full credit.

Complete method `isDiverse` below.

```
/** Returns true  if all rows in arr2D have different row sums;
 *         false otherwise.
 */
public static boolean isDiverse(int[][] arr2D)
```

3. A student in a school is represented by the following class.

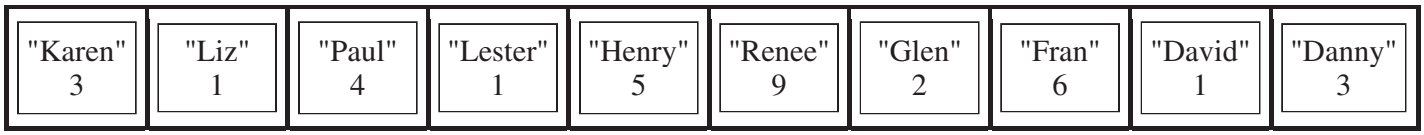
```
public class Student
{
  /** Returns the name of this Student. */
  public String getName()
  {  /* implementation not shown */  }

  /** Returns the number of times this Student has missed class. */
  public int getAbsenceCount()
  {  /* implementation not shown */  }

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

The class `SeatingChart`, shown below, uses a two-dimensional array to represent the seating arrangement of students in a classroom. The seats in the classroom are in a rectangular arrangement of rows and columns.

```
public class SeatingChart
{
  /** seats[r][c] represents the Student in row r and column c in the classroom. */
  private Student[][] seats;

  /** Creates a seating chart with the given number of rows and columns from the students in
   *  studentList. Empty seats in the seating chart are represented by null.
   *  @param rows the number of rows of seats in the classroom
   *  @param cols the number of columns of seats in the classroom
   *  Precondition: rows > 0; cols > 0;
   *                rows * cols >= studentList.size()
   *  Postcondition:
   *     – Students appear in the seating chart in the same order as they appear
   *       in studentList, starting at seats[0][0].
   *     – seats is filled column by column from studentList, followed by any
   *       empty seats (represented by null).
   *     – studentList is unchanged.
   */
  public SeatingChart(List<Student> studentList,
                      int rows, int cols)
  {  /* to be implemented in part (a) */  }

  /** Removes students who have more than a given number of absences from the
   *  seating chart, replacing those entries in the seating chart with null
   *  and returns the number of students removed.
   *  @param allowedAbsences an integer >= 0
   *  @return number of students removed from seats
   *  Postcondition:
   *    – All students with allowedAbsences or fewer are in their original positions in seats.
   *    – No student in seats has more than allowedAbsences absences.
   *    – Entries without students contain null.
   */
  public int removeAbsentStudents(int allowedAbsences)
  {  /* to be implemented in part (b) */  }

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

(a) Write the constructor for the `SeatingChart` class. The constructor initializes the `seats` instance variable to a two-dimensional array with the given number of rows and columns. The students in `studentList` are copied into the seating chart in the order in which they appear in `studentList`. The students are assigned to consecutive locations in the array `seats`, starting at `seats[0][0]` and filling the array column by column. Empty seats in the seating chart are represented by `null`.

For example, suppose a variable `List<Student> roster` contains references to `Student` objects in the following order.



A `SeatingChart` object created with the call `new SeatingChart(roster, 3, 4)` would have `seats` initialized with the following values.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	"Henry" 5	"Fran" 6	`null`
2	"Paul" 4	"Renee" 9	"David" 1	`null`

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Part (a) continues on page 9.

Complete the `SeatingChart` constructor below.

```
/** Creates a seating chart with the given number of rows and columns from the students in
 *  studentList. Empty seats in the seating chart are represented by null.
 *  @param rows the number of rows of seats in the classroom
 *  @param cols the number of columns of seats in the classroom
 *  Precondition: rows > 0; cols > 0;
 *                rows * cols >= studentList.size()
 *  Postcondition:
 *    – Students appear in the seating chart in the same order as they appear
 *      in studentList, starting at seats[0][0].
 *    – seats is filled column by column from studentList, followed by any
 *      empty seats (represented by null).
 *    – studentList is unchanged.
 */
public SeatingChart(List<Student> studentList,
                    int rows, int cols)
```

Part (b) begins on page 10.

(b) Write the `removeAbsentStudents` method, which removes students who have more than a given number of absences from the seating chart and returns the number of students that were removed. When a student is removed from the seating chart, a `null` is placed in the entry for that student in the array `seats`. For example, suppose the variable `SeatingChart introCS` has been created such that the array `seats` contains the following entries showing both students and their number of absences.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	"Henry" 5	"Fran" 6	`null`
2	"Paul" 4	"Renee" 9	"David" 1	`null`

After the call `introCS.removeAbsentStudents(4)` has executed, the array `seats` would contain the following values and the method would return the value 3.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	**`null`**	**`null`**	`null`
2	"Paul" 4	**`null`**	"David" 1	`null`

Class information repeated from the beginning of the question:

```
public class Student

public String getName()
public int getAbsenceCount()

public class SeatingChart

private Student[][] seats
public SeatingChart(List<Student> studentList,
                    int rows, int cols)
public int removeAbsentStudents(int allowedAbsences)
```

Complete method `removeAbsentStudents` below.

```
/** Removes students who have more than a given number of absences from the
 *  seating chart, replacing those entries in the seating chart with null
 *  and returns the number of students removed.
 *  @param allowedAbsences an integer >= 0
 *  @return number of students removed from seats
 *  Postcondition:
 *    - All students with allowedAbsences or fewer are in their original positions in seats.
 *    - No student in seats has more than allowedAbsences absences.
 *    - Entries without students contain null.
 */
public int removeAbsentStudents(int allowedAbsences)
```