

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 <code>clearPair</code>	Method Call	puzzle after call to <code>clearPair</code>	Return Value	Explanation
0 7 9 0 7 4 1 6 8 4 1 8	<code>clearPair(0, 1)</code>	0 0 9 0 0 4 1 6 8 4 1 8	<code>true</code>	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	<code>clearPair(2, 2)</code>	1 2 3 4 5 6 7 8 5 4 1 2	<code>false</code>	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	<code>clearPair(1, 1)</code>	8 1 0 5 0 0 3 0 3 4 5 8	<code>true</code>	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	<code>clearPair(0, 2)</code>	0 7 0 2 6 5 4 4 4	<code>true</code>	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
<code>getNextLoc(0, 0)</code>	Returns the neighbor to the right (the <code>Location</code> representing the element at row 0 and column 1), since $3 < 11$
<code>getNextLoc(1, 3)</code>	Returns the neighbor below (the <code>Location</code> representing the element at row 2 and column 3), since $15 < 16$
<code>getNextLoc(2, 4)</code>	Returns the neighbor below (the <code>Location</code> representing the element at row 3 and column 4), since the given element has no neighbor to the right
<code>getNextLoc(4, 3)</code>	Returns the neighbor to the right (the <code>Location</code> 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. 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 <code>evaluateLight</code>	Value Returned	Explanation
<code>sim.evaluateLight(0, 3);</code>	<code>false</code>	The light is on, and the number of lights that are on in its column is even.
<code>sim.evaluateLight(6, 0);</code>	<code>true</code>	The light is off, and the number of lights that are on in its column is divisible by 3.
<code>sim.evaluateLight(4, 1);</code>	<code>false</code>	Returns the light's current status.
<code>sim.evaluateLight(5, 4);</code>	<code>true</code>	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

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.

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

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  $\geq 0$ 
 * @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)
```