

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

1. This question involves the `AppointmentBook` class, which provides methods for students to schedule appointments with their teacher. Appointments can be scheduled during one of eight class periods during the school day, numbered 1 through 8. A requested appointment has a duration, which is the number of minutes the appointment will last. The 60 minutes within a period are numbered 0 through 59. In order for an appointment to be scheduled, the teacher must have a block of consecutive, available minutes that contains at least the requested number of minutes in a requested period. Scheduled appointments must start and end within the same period.

The `AppointmentBook` class contains two helper methods, `isMinuteFree` and `reserveBlock`. You will write two additional methods of the `AppointmentBook` class.

```
public class AppointmentBook
{
    /**
     * Returns true if minute in period is available for an appointment and returns
     * false otherwise
     * Preconditions: 1 <= period <= 8; 0 <= minute <= 59
     */
    private boolean isMinuteFree(int period, int minute)
    { /* implementation not shown */ }

    /**
     * Marks the block of minutes that starts at startMinute in period and
     * is duration minutes long as reserved for an appointment
     * Preconditions: 1 <= period <= 8; 0 <= startMinute <= 59;
     * 1 <= duration <= 60
     */
    private void reserveBlock(int period, int startMinute, int duration)
    { /* implementation not shown */ }

    /**
     * Searches for the first block of duration free minutes during period, as described in
     * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
     * such block is found.
     * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
     */
    public int findFreeBlock(int period, int duration)
    { /* to be implemented in part (a) */ }

    /**
     * Searches periods from startPeriod to endPeriod, inclusive, for a block
     * of duration free minutes, as described in part (b). If such a block is found,
     * calls reserveBlock to reserve the block of minutes and returns true; otherwise
     * returns false.
     * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
     */
    public boolean makeAppointment(int startPeriod, int endPeriod,
                                   int duration)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `findFreeBlock` method, which searches `period` for the first block of free minutes that is `duration` minutes long. If such a block is found, `findFreeBlock` returns the first minute in the block. Otherwise, `findFreeBlock` returns `-1`. The `findFreeBlock` method uses the helper method `isMinuteFree`, which returns `true` if a particular minute is available to be included in a new appointment and returns `false` if the minute is unavailable.

Consider the following list of unavailable and available minutes in period 2.

Minutes in Period 2	Available?
0–9 (10 minutes)	No
10–14 (5 minutes)	Yes
15–29 (15 minutes)	No
30–44 (15 minutes)	Yes
45–49 (5 minutes)	No
50–59 (10 minutes)	Yes

The method call `findFreeBlock(2, 15)` would return `30` to indicate that a 15-minute block starting with minute 30 is available. No steps should be taken as a result of the call to `findFreeBlock` to mark those 15 minutes as unavailable.

The method call `findFreeBlock(2, 9)` would also return `30`. Whenever there are multiple blocks that satisfy the requirement, the earliest starting minute is returned.

The method call `findFreeBlock(2, 20)` would return `-1`, since no 20-minute block of available minutes exists in period 2.

Complete method `findFreeBlock`. You must use `isMinuteFree` appropriately in order to receive full credit.

```
/**
 * Searches for the first block of duration free minutes during period, as described in
 * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
 * such block is found.
 * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
 */
public int findFreeBlock(int period, int duration)
```

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 `makeAppointment` method, which searches the periods from `startPeriod` to `endPeriod`, inclusive, for the earliest block of `duration` available minutes in the lowest-numbered period. If such a block is found, the `makeAppointment` method calls the helper method `reserveBlock` to mark the minutes in the block as unavailable and returns `true`. If no such block is found, the `makeAppointment` method returns `false`.

Consider the following list of unavailable and available minutes in periods 2, 3, and 4 and three successive calls to `makeAppointment`.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–14 (15 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–4 (5 minutes)	No
4	5–29 (25 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

The method call `makeAppointment(2, 4, 22)` returns `true` and results in the minutes 5 through 26, inclusive, in period 4 being marked as unavailable.

The method call `makeAppointment(3, 4, 3)` returns `true` and results in the minutes 0 through 2, inclusive, in period 3 being marked as unavailable.

The method call `makeAppointment(2, 4, 30)` returns `false`, since there is no block of 30 available minutes in periods 2, 3, or 4.

The following shows the updated list of unavailable and available minutes in periods 2, 3, and 4 after the three example method calls are complete.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–2 (3 minutes)	No
3	3–14 (12 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–26 (27 minutes)	No
4	27–29 (3 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

Complete method `makeAppointment`. Assume that `findFreeBlock` works as intended, regardless of what you wrote in part (a). You must use `findFreeBlock` and `reserveBlock` appropriately in order to receive full credit.

```
/**
 * Searches periods from startPeriod to endPeriod, inclusive, for a block
 * of duration free minutes, as described in part (b). If such a block is found,
 * calls reserveBlock to reserve the block of minutes and returns true; otherwise
 * returns false.
 * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
 */
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```

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 AppointmentBook
private boolean isMinuteFree(int period, int minute)
private void reserveBlock(int period, int startMinute,
                           int duration)
public int findFreeBlock(int period, int duration)
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```


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.

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

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.

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

or

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

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`.

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

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