

3. The `CourseRecord` class is used to store information about a student enrolled in a course. A partial definition of the `CourseRecord` class is shown.

```
public class CourseRecord
{
    /**
     * Returns a unique ID for the student associated with this
     * CourseRecord
     */
    public String getStudentID()
    { /* implementation not shown */ }

    /**
     * Returns a nonnegative integer representing the number of times the
     * student associated with this CourseRecord has been absent in the
     * course
     */
    public int getAbsences()
    { /* implementation not shown */ }

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

The `Attendance` class maintains two `ArrayLists` of `CourseRecord` objects, named `historyList` and `mathList`. A partial definition of the `Attendance` class is shown.

```
public class Attendance
{
    /* The students enrolled in a history course */
    private ArrayList<CourseRecord> historyList;

    /* The students enrolled in a math course */
    private ArrayList<CourseRecord> mathList;

    /**
     * Returns the number of students who are enrolled in both
     * the history course and the math course but have more
     * absences in the history course than the math course
     * Preconditions:
     *     No student ID appears multiple times in historyList.
     *     No student ID appears multiple times in mathList.
     *     historyList and mathList do not contain any null elements.
     * Postcondition:
     *     historyList and mathList are unchanged.
     */
    public int moreHistoryThanMathAbsences()
    { /* to be implemented */ }

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

Write the `Attendance` method `moreHistoryThanMathAbsences`. The method should return the number of students who are enrolled in both the history course and the math course but have more absences in the history course than the math course.

For example, suppose `historyList` and `mathList` have the following contents.

`historyList`:

Student ID	"rc29"	"br98"	"dr03"	"ot32"	"sq98"	"ry00"
Number of Absences	1	1	2	2	3	1

`mathList`:

Student ID	"fr27"	"sq98"	"dr03"	"dk12"	"ot32"	"js33"	"ry00"
Number of Absences	2	1	2	1	1	0	3

In this example, there are four student IDs ("dr03", "ot32", "sq98", and "ry00") that appear in both `historyList` and `mathList`. Of these, only "ot32" and "sq98" have more absences in the history course than the math course, so the `moreHistoryThanMathAbsences` method should return 2.

Complete method `moreHistoryThanMathAbsences`.

```
/**
 * Returns the number of students who are enrolled in both
 * the history course and the math course but have more
 * absences in the history course than the math course
 * Preconditions:
 *     No student ID appears multiple times in historyList.
 *     No student ID appears multiple times in mathList.
 *     historyList and mathList do not contain any null elements.
 * Postcondition:
 *     historyList and mathList are unchanged.
 */
public int moreHistoryThanMathAbsences()
```

3. This question involves pairing competitors in a tournament into one-on-one matches for one round of the tournament. For example, in a chess tournament, the competitors are the individual chess players. A game of chess involving two players is a match. The winner of each match goes on to a match in the next round of the tournament. Since half of the players are eliminated in each round of the tournament, there is eventually a final round consisting of one match and two competitors. The winner of that match is considered the winner of the tournament.

Competitors, matches, and rounds of the tournament are represented by the `Competitor`, `Match`, and `Round` classes. You will write the constructor and one method of the `Round` class.

```
/** A single competitor in the tournament */
public class Competitor
{
    /** The competitor's name and rank */
    private String name;
    private int rank;

    /**
     * Assigns n to name and initialRank to rank
     * Precondition: initialRank >= 1
     */
    public Competitor(String n, int initialRank)
    { /* implementation not shown */ }

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

```
/** A match between two competitors */
public class Match
{
    public Match(Competitor one, Competitor two)
    { /* implementation not shown */ }

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

/** A single round of the tournament */
public class Round
{
    /** The list of competitors participating in this round */
    private ArrayList<Competitor> competitorList;

    /** Initializes competitorList, as described in part (a) */
    public Round(String[] names)
    { /* to be implemented in part (a) */ }

    /**
     * Creates an ArrayList of Match objects for the next round
     * of the tournament, as described in part (b)
     * Preconditions: competitorList contains at least one element.
     *                 competitorList is ordered from best to worst rank.
     * Postcondition: competitorList is unchanged.
     */
    public ArrayList<Match> buildMatches()
    { /* to be implemented in part (b) */ }

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

- A. Write the constructor for the `Round` class. The constructor should initialize `competitorList` to contain one `Competitor` object for each name in the `String[]` `names`. The order in which `Competitor` objects appear in `competitorList` should be the same as the order in which they appear in `names`, and the rank of each competitor is based on the competitor's position in `names`. Names are listed in `names` in order from the best-ranked competitor with rank 1 to the worst-ranked competitor with rank n , where n is the number of elements in `names`.

For example, assume the following code segment is executed.

```
String[] players = {"Alex", "Ben", "Cara"};
Round r = new Round(players);
```

The following shows the contents of `competitorList` in `r` after the constructor has finished executing.

0	1	2
name: "Alex" rank: 1	name: "Ben" rank: 2	name: "Cara" rank: 3

Complete the `Round` constructor.

```
/** Initializes competitorList, as described in part (a) */
public Round(String[] names)
```

B. Write the `Round` method `buildMatches`. This method should return a new `ArrayList<Match>` object by pairing competitors from `competitorList` according to the following rules.

- If the number of competitors in `competitorList` is even, the best-ranked competitor is paired with the worst-ranked competitor, the second-best-ranked competitor is paired with the second-worst-ranked competitor, etc.
- If the number of competitors in `competitorList` is odd, the competitor with the best rank is ignored and the remaining competitors are paired according to the rule for an even number of competitors.

Each pair of competitors is used to create a `Match` object that should be added to the `ArrayList` to return. Competitors may appear in either order in a `Match` object, and matches may appear in any order in the returned `ArrayList`.

The following example shows the contents of `competitorList` in a `Round` object `r1` containing an odd number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r1.buildMatches()`.

`competitorList`:

0	1	2
name: "Alex" rank: 1	name: "Ben" rank: 2	name: "Cara" rank: 3

The `ArrayList` to be returned contains a single match between Ben and Cara, the second and third-ranked competitors:

0
First Competitor: name: "Ben" rank: 2
Second Competitor: name: "Cara" rank: 3

The next example shows the contents of `competitorList` in a `Round` object `r2` containing an even number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r2.buildMatches()`.

`competitorList`:

0	1	2	3
name: "Rei" rank: 1	name: "Sam" rank: 2	name: "Vi" rank: 3	name: "Tim" rank: 4

The `ArrayList` to be returned contains two matches: one match between the first- and last-ranked competitors Rei and Tim, and a second match between the second- and third-ranked competitors Sam and Vi:

0	1
First Competitor: name: "Rei" rank: 1	First Competitor: name: "Sam" rank: 2
Second Competitor: name: "Tim" rank: 4	Second Competitor: name: "Vi" rank: 3

Complete the `buildMatches` method.

```
/**
 * Creates an ArrayList of Match objects for the next round
 * of the tournament, as described in part (b)
 * Preconditions: competitorList contains at least one element.
 *                competitorList is ordered from best to worst rank.
 * Postcondition: competitorList is unchanged.
 */
public ArrayList<Match> buildMatches()
```


3. This question involves the manipulation and analysis of a list of words. The following `WordChecker` class contains an `ArrayList<String>` to be analyzed and methods that are used to perform the analysis. You will write two methods of the `WordChecker` class.

```
public class WordChecker
{
    /** Initialized in the constructor and contains no null elements */
    private ArrayList<String> wordList;

    /**
     * Returns true if each element of wordList (except the first) contains the previous
     * element as a substring and returns false otherwise, as described in part (a)
     * Precondition: wordList contains at least two elements.
     * Postcondition: wordList is unchanged.
     */
    public boolean isWordChain()
    { /* to be implemented in part (a) */ }

    /**
     * Returns an ArrayList<String> based on strings from wordList that start
     * with target, as described in part (b). Each element of the returned ArrayList has had
     * the initial occurrence of target removed.
     * Postconditions: wordList is unchanged.
     * Items appear in the returned list in the same order as they appear in wordList.
     */
    public ArrayList<String> createList(String target)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `isWordChain` method, which determines whether each element of `wordList` (except the first) contains the previous element as a substring. The following table shows two sample `isWordChain` method calls.

<code>wordList</code>	<code>isWordChain</code> Return Value	Explanation
<code>["an", "band", "band", "abandon"]</code>	<code>true</code>	Each element contains the previous element as a substring.
<code>["to", "too", "stool", "tools"]</code>	<code>false</code>	"tools" does not contain the substring "stool".

Complete the `isWordChain` method.

```
/**
 * Returns true if each element of wordList (except the first) contains the previous
 * element as a substring and returns false otherwise, as described in part (a)
 * Precondition: wordList contains at least two elements.
 * Postcondition: wordList is unchanged.
 */
public boolean isWordChain()
```

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 `createList` method, which creates and returns an `ArrayList<String>`. The method identifies strings in `wordList` that start with `target` and returns a new `ArrayList` containing each identified string without the starting occurrence of `target`. Elements must appear in the returned list in the same order as they appear in `wordList`.

Consider an example where `wordList` contains the following strings.

```
["catch", "bobcat", "catchacat", "cat", "at"]
```

The following table shows the `ArrayList` returned by some calls to `createList`. In all cases, `wordList` is unchanged.

Method Call	ArrayList Returned by <code>createList</code>	Explanation
<code>createList("cat")</code>	<code>["ch", "chacat", ""]</code>	Only "catch", "catchacat", and "cat" begin with "cat".
<code>createList("catch")</code>	<code>["", "acat"]</code>	Only "catch" and "catchacat" begin with "catch".
<code>createList("dog")</code>	<code>[]</code>	None of the words in <code>wordList</code> begin with "dog".

Complete the `createList` method.

```
/**
 * Returns an ArrayList<String> based on strings from wordList that start
 * with target, as described in part (b). Each element of the returned ArrayList has had
 * the initial occurrence of target removed.
 * Postconditions: wordList is unchanged.
 * Items appear in the returned list in the same order as they appear in wordList.
 */
public ArrayList<String> createList(String target)
```

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 WordChecker
private ArrayList<String> wordList
public boolean isWordChain()
public ArrayList<String> createList(String target)
```

3. This question involves the analysis of weather data. The following `WeatherData` class has an instance variable, `temperatures`, which contains the daily high temperatures recorded on consecutive days at a particular location. The class also contains methods used to analyze that data. You will write two methods of the `WeatherData` class.

```
public class WeatherData
{
    /** Guaranteed not to be null and to contain only non-null entries */
    private ArrayList<Double> temperatures;

    /**
     * Cleans the data by removing from temperatures all values that are less than
     * lower and all values that are greater than upper, as described in part (a)
     */
    public void cleanData(double lower, double upper)
    { /* to be implemented in part (a) */ }

    /**
     * Returns the length of the longest heat wave found in temperatures, as described in
     * part (b)
     * Precondition: There is at least one heat wave in temperatures based on threshold.
     */
    public int longestHeatWave(double threshold)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `cleanData` method, which modifies the `temperatures` instance variable by removing all values that are less than the `lower` parameter and all values that are greater than the `upper` parameter. The order of the remaining values in `temperatures` must be maintained.

For example, consider a `WeatherData` object for which `temperatures` contains the following.

99.1	142.0	85.0	85.1	84.6	94.3	124.9	98.0	101.0	102.5
------	-------	------	------	------	------	-------	------	-------	-------

The three shaded values shown would be removed by the method call `cleanData(85.0, 120.0)`.

99.1	142.0	85.0	85.1	84.6	94.3	124.9	98.0	101.0	102.5
------	-------	------	------	------	------	-------	------	-------	-------

The following shows the contents of `temperatures` after the three shaded values are removed as a result of the method call `cleanData(85.0, 120.0)`.

99.1	85.0	85.1	94.3	98.0	101.0	102.5
------	------	------	------	------	-------	-------

Complete method `cleanData`.

```
/**
 * Cleans the data by removing from temperatures all values that are less than
 * lower and all values that are greater than upper, as described in part (a)
 */
public void cleanData(double lower, double upper)
```

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 `longestHeatWave` method, which returns the length of the longest heat wave found in the `temperatures` instance variable. A heat wave is a sequence of two or more consecutive days with a daily high temperature greater than the parameter `threshold`. The `temperatures` instance variable is guaranteed to contain at least one heat wave based on the `threshold` parameter.

For example, consider the following contents of `temperatures`.

100.5	98.5	102.0	103.9	87.5	105.2	90.3	94.8	109.1	102.1	107.4	93.2
-------	------	-------	-------	------	-------	------	------	-------	-------	-------	------

In the following sample contents of `temperatures`, all heat waves based on the threshold temperature of 100.5 are shaded. The method call `longestHeatWave(100.5)` would return 3, which is the length of the longest heat wave.

100.5	98.5	102.0	103.9	87.5	105.2	90.3	94.8	109.1	102.1	107.4	93.2
-------	------	-------	-------	------	-------	------	------	-------	-------	-------	------

In the following sample contents of `temperatures`, all heat waves based on the threshold temperature of 95.2 are shaded. The method call `longestHeatWave(95.2)` would return 4, which is the length of the longest heat wave.

100.5	98.5	102.0	103.9	87.5	105.2	90.3	94.8	109.1	102.1	107.4	93.2
-------	------	-------	-------	------	-------	------	------	-------	-------	-------	------

Complete method `longestHeatWave`.

```
/**
 * Returns the length of the longest heat wave found in temperatures, as described in
 * part (b)
 * Precondition: There is at least one heat wave in temperatures based on threshold.
 */
public int longestHeatWave(double threshold)
```

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 WeatherData
private ArrayList<Double> temperatures
public void cleanData(double lower, double upper)
public int longestHeatWave(double threshold)
```

3. Users of a website are asked to provide a review of the website at the end of each visit. Each review, represented by an object of the `Review` class, consists of an integer indicating the user's rating of the website and an optional `String` comment field. The comment field in a `Review` object ends with a period (". "), exclamation point ("! "), or letter, or is a `String` of length 0 if the user did not enter a comment.

```
public class Review
{
    private int rating;
    private String comment;

    /** Precondition: r >= 0
     *      c is not null.
     */
    public Review(int r, String c)
    {
        rating = r;
        comment = c;
    }

    public int getRating()
    {
        return rating;
    }

    public String getComment()
    {
        return comment;
    }

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

GO ON TO THE NEXT PAGE.

The `ReviewAnalysis` class contains methods used to analyze the reviews provided by users. You will write two methods of the `ReviewAnalysis` class.

```
public class ReviewAnalysis
{
    /** All user reviews to be included in this analysis */
    private Review[] allReviews;

    /** Initializes allReviews to contain all the Review objects to be analyzed */
    public ReviewAnalysis()
    { /* implementation not shown */ }

    /** Returns a double representing the average rating of all the Review objects to be
     * analyzed, as described in part (a)
     * Precondition: allReviews contains at least one Review.
     * No element of allReviews is null.
     */
    public double getAverageRating()
    { /* to be implemented in part (a) */ }

    /** Returns an ArrayList of String objects containing formatted versions of
     * selected user comments, as described in part (b)
     * Precondition: allReviews contains at least one Review.
     * No element of allReviews is null.
     * Postcondition: allReviews is unchanged.
     */
    public ArrayList<String> collectComments()
    { /* to be implemented in part (b) */ }
}
```

GO ON TO THE NEXT PAGE.

- (a) Write the `ReviewAnalysis` method `getAverageRating`, which returns the average rating (arithmetic mean) of all elements of `allReviews`. For example, `getAverageRating` would return 3.4 if `allReviews` contained the following `Review` objects.

0	1	2	3	4
4 "Good! Thx"	3 "OK site"	5 "Great!"	2 "Poor! Bad."	3 ""

Complete method `getAverageRating`.

```
/** Returns a double representing the average rating of all the Review objects to be
 * analyzed, as described in part (a)
 * Precondition: allReviews contains at least one Review.
 * No element of allReviews is null.
 */
public double getAverageRating()
```

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 Review
private int rating
private String comment
public Review(int r, String c)
public int getRating()
public String getComment()
public class ReviewAnalysis
private Review[] allReviews
public ReviewAnalysis()
public double getAverageRating()
public ArrayList<String> collectComments()
```

GO ON TO THE NEXT PAGE.

(b) Write the `ReviewAnalysis` method `collectComments`, which collects and formats only comments that contain an exclamation point. The method returns an `ArrayList` of `String` objects containing copies of user comments from `allReviews` that contain an exclamation point, formatted as follows. An empty `ArrayList` is returned if no comment in `allReviews` contains an exclamation point.

- The `String` inserted into the `ArrayList` to be returned begins with the index of the `Review` in `allReviews`.
- The index is immediately followed by a hyphen (" - ").
- The hyphen is followed by a copy of the original comment.
- The `String` must end with either a period or an exclamation point. If the original comment from `allReviews` does not end in either a period or an exclamation point, a period is added.

The following example of `allReviews` is repeated from part (a).

0	1	2	3	4
4 "Good! Thx"	3 "OK site"	5 "Great!"	2 "Poor! Bad."	3 ""

The following `ArrayList` would be returned by a call to `collectComments` with the given contents of `allReviews`. The reviews at index 1 and index 4 in `allReviews` are not included in the `ArrayList` to return since neither review contains an exclamation point.

"0-Good! Thx."	"2-Great!"	"3-Poor! Bad."
----------------	------------	----------------

Complete method `collectComments`.

```
/** Returns an ArrayList of String objects containing formatted versions of
 * selected user comments, as described in part (b)
 * Precondition: allReviews contains at least one Review.
 * No element of allReviews is null.
 * Postcondition: allReviews is unchanged.
 */
public ArrayList<String> collectComments()
```

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.

3. A high school club maintains information about its members in a `MemberInfo` object. A `MemberInfo` object stores a club member's name, year of graduation, and whether or not the club member is in *good standing*. A member who is in good standing has fulfilled all the responsibilities of club membership.

A partial declaration of the `MemberInfo` class is shown below.

```
public class MemberInfo
{
    /** Constructs a MemberInfo object for the club member with name name,
     * graduation year gradYear, and standing hasGoodStanding.
     */
    public MemberInfo(String name, int gradYear, boolean hasGoodStanding)
    { /* implementation not shown */ }

    /** Returns the graduation year of the club member. */
    public int getGradYear()
    { /* implementation not shown */ }

    /** Returns true if the member is in good standing and false otherwise. */
    public boolean inGoodStanding()
    { /* implementation not shown */ }

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

The `ClubMembers` class maintains a list of current club members. The declaration of the `ClubMembers` class is shown below.

```
public class ClubMembers
{
    private ArrayList<MemberInfo> memberList;

    /** Adds new club members to memberList, as described in part (a).
     * Precondition: names is a non-empty array.
     */
    public void addMembers(String[] names, int gradYear)
    { /* to be implemented in part (a) */ }

    /** Removes members who have graduated and returns a list of members who have graduated
     * and are in good standing, as described in part (b).
     */
    public ArrayList<MemberInfo> removeMembers(int year)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `ClubMembers` method `addMembers`, which takes two parameters. The first parameter is a `String` array containing the names of new club members to be added. The second parameter is the graduation year of all the new club members. The method adds the new members to the `memberList` instance variable. The names can be added in any order. All members added are initially in good standing and share the same graduation year, `gradYear`.

Complete the `addMembers` method.

```
/** Adds new club members to memberList, as described in part (a).
 * Precondition: names is a non-empty array.
 */
public void addMembers(String[] names, int gradYear)
```

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 `ClubMembers` method `removeMembers`, which takes the following actions.

- Returns a list of all students who have graduated and are in good standing. A member has graduated if the member's graduation year is less than or equal to the method's `year` parameter. If no members meet these criteria, an empty list is returned.
- Removes from `memberList` all members who have graduated, regardless of whether or not they are in good standing.

The following example illustrates the results of a call to `removeMembers`.

The `ArrayList memberList` before the method call `removeMembers(2018)`:

"SMITH, JANE"	"FOX, STEVE"	"XIN, MICHAEL"	"GARCIA, MARIA"
2019	2018	2017	2020
false	true	false	true

The `ArrayList memberList` after the method call `removeMembers(2018)`:

"SMITH, JANE"	"GARCIA, MARIA"
2019	2020
false	true

The `ArrayList` returned by the method call `removeMembers(2018)`:

"FOX, STEVE"
2018
true

Complete the `removeMembers` method.

```
/** Removes members who have graduated and returns a list of members who have graduated and are
 * in good standing, as described in part (b).
 */
public ArrayList<MemberInfo> removeMembers(int year)
```

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 MemberInfo
```

```
public MemberInfo(String name, int gradYear, boolean hasGoodStanding)
public int getGradYear()
public boolean inGoodStanding()
```

```
public class ClubMembers
```

```
private ArrayList<MemberInfo> memberList
```

```
public void addMembers(String[] names, int gradYear)
public ArrayList<MemberInfo> removeMembers(int year)
```

3. Many encoded strings contain *delimiters*. A delimiter is a non-empty string that acts as a boundary between different parts of a larger string. The delimiters involved in this question occur in pairs that must be *balanced*, with each pair having an open delimiter and a close delimiter. There will be only one type of delimiter for each string. The following are examples of delimiters.

Example 1

Expressions in mathematics use open parentheses " (" and close parentheses ") " as delimiters. For each open parenthesis, there must be a matching close parenthesis.

$(x + y) * 5$ is a valid mathematical expression.

$(x + (y)$ is NOT a valid mathematical expression because there are more open delimiters than close delimiters.

Example 2

HTML uses `<B>` and `</B>` as delimiters. For each open delimiter `<B>`, there must be a matching close delimiter `</B>`.

`<B> Make this text bold </B>` is valid HTML.

`<B> Make this text bold </UB>` is NOT valid HTML because there is one open delimiter and no matching close delimiter.

In this question, you will write two methods in the following `Delimiters` class.

```
public class Delimiters
{
    /** The open and close delimiters. */
    private String openDel;
    private String closeDel;

    /** Constructs a Delimiters object where open is the open delimiter and close is the
     * close delimiter.
     * Precondition: open and close are non-empty strings.
     */
    public Delimiters(String open, String close)
    {
        openDel = open;
        closeDel = close;
    }

    /** Returns an ArrayList of delimiters from the array tokens, as described in part (a). */
    public ArrayList<String> getDelimitersList(String[] tokens)
    {
        /* to be implemented in part (a) */
    }

    /** Returns true if the delimiters are balanced and false otherwise, as described in part (b).
     * Precondition: delimiters contains only valid open and close delimiters.
     */
    public boolean isBalanced(ArrayList<String> delimiters)
    {
        /* to be implemented in part (b) */
    }

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


- (a) A string containing text and possibly delimiters has been split into *tokens* and stored in `String[] tokens`. Each token is either an open delimiter, a close delimiter, or a substring that is not a delimiter. You will write the method `getDelimitersList`, which returns an `ArrayList` containing all the open and close delimiters found in `tokens` in their original order.

The following examples show the contents of an `ArrayList` returned by `getDelimitersList` for different open and close delimiters and different `tokens` arrays.

Example 1

openDel: "("

closeDel: ")"

tokens:

"("	"x + y"	")"	" * 5"
-----	---------	-----	--------

ArrayList
of delimiters:

"("	")"
-----	-----

Example 2

openDel: "<q>"

closeDel: "</q>"

tokens:

"<q>"	"yy"	"</q>"	"zz"	"</q>"
-------	------	--------	------	--------

ArrayList
of delimiters:

"<q>"	"</q>"	"</q>"
-------	--------	--------

Class information for this question

```
public class Delimiters
```

```
private String openDel
```

```
private String closeDel
```

```
public Delimiters(String open, String close)
```

```
public ArrayList<String> getDelimitersList(String[] tokens)
```

```
public boolean isBalanced(ArrayList<String> delimiters)
```

Complete method `getDelimitersList` below.

```
/** Returns an ArrayList of delimiters from the array tokens, as described in part (a). */  
public ArrayList<String> getDelimitersList(String[] tokens)
```

- (b) Write the method `isBalanced`, which returns `true` when the delimiters are balanced and returns `false` otherwise. The delimiters are balanced when both of the following conditions are satisfied; otherwise, they are not balanced.
1. When traversing the `ArrayList` from the first element to the last element, there is no point at which there are more close delimiters than open delimiters at or before that point.
 2. The total number of open delimiters is equal to the total number of close delimiters.

Consider a `Delimiters` object for which `openDel` is `"<sup>"` and `closeDel` is `"</sup>"`. The examples below show different `ArrayList` objects that could be returned by calls to `getDelimitersList` and the value that would be returned by a call to `isBalanced`.

Example 1

The following example shows an `ArrayList` for which `isBalanced` returns `true`. As tokens are examined from first to last, the number of open delimiters is always greater than or equal to the number of close delimiters. After examining all tokens, there are an equal number of open and close delimiters.

"^{"	"^{"	"}"	"^{"	"}"	"}"
---------	---------	----------	---------	----------	----------

Example 2

The following example shows an `ArrayList` for which `isBalanced` returns `false`.

"^{"	"}"	"</sup>"	"<sup>"
---------	----------	----------	---------



When starting from the left, at this point, condition 1 is violated.

Example 3

The following example shows an `ArrayList` for which `isBalanced` returns `false`.

"</sup>"



At this point, condition 1 is violated.

Example 4

The following example shows an `ArrayList` for which `isBalanced` returns `false` because the second condition is violated. After examining all tokens, there are not an equal number of open and close delimiters.

"<sup>"	"^{"	"}"
---------	---------	----------

Class information for this question

```
public class Delimiters
private String openDel
private String closeDel

public Delimiters(String open, String close)
public ArrayList<String> getDelimitersList(String[] tokens)
public boolean isBalanced(ArrayList<String> delimiters)
```

Complete method `isBalanced` below.

```
/** Returns true if the delimiters are balanced and false otherwise, as described in part (b).
 * Precondition: delimiters contains only valid open and close delimiters.
 */
public boolean isBalanced(ArrayList<String> delimiters)
```

2. This question involves reasoning about pairs of words that are represented by the following `WordPair` class.

```
public class WordPair
{
    /** Constructs a WordPair object. */
    public WordPair(String first, String second)
    { /* implementation not shown */ }

    /** Returns the first string of this WordPair object. */
    public String getFirst()
    { /* implementation not shown */ }

    /** Returns the second string of this WordPair object. */
    public String getSecond()
    { /* implementation not shown */ }
}
```

You will implement the constructor and another method for the following `WordPairList` class.

```
public class WordPairList
{
    /** The list of word pairs, initialized by the constructor. */
    private ArrayList<WordPair> allPairs;

    /** Constructs a WordPairList object as described in part (a).
     * Precondition: words.length >= 2
     */
    public WordPairList(String[] words)
    { /* to be implemented in part (a) */ }

    /** Returns the number of matches as described in part (b).
     */
    public int numMatches()
    { /* to be implemented in part (b) */ }
}
```

- (a) Write the constructor for the `WordPairList` class. The constructor takes an array of strings `words` as a parameter and initializes the instance variable `allPairs` to an `ArrayList` of `WordPair` objects.

A `WordPair` object consists of a word from the array paired with a word that appears later in the array. The `allPairs` list contains `WordPair` objects (`words[i]`, `words[j]`) for every `i` and `j`, where $0 \leq i < j < \text{words.length}$. Each `WordPair` object is added exactly once to the list.

The following examples illustrate two different `WordPairList` objects.

Example 1

```
String[] wordNums = {"one", "two", "three"};
WordPairList exampleOne = new WordPairList(wordNums);
```

After the code segment has executed, the `allPairs` instance variable of `exampleOne` will contain the following `WordPair` objects in some order.

```
("one", "two"), ("one", "three"), ("two", "three")
```

Example 2

```
String[] phrase = {"the", "more", "the", "merrier"};
WordPairList exampleTwo = new WordPairList(phrase);
```

After the code segment has executed, the `allPairs` instance variable of `exampleTwo` will contain the following `WordPair` objects in some order.

```
("the", "more"), ("the", "the"), ("the", "merrier"),
("more", "the"), ("more", "merrier"), ("the", "merrier")
```

Class information for this question

```
public class WordPair
```

```
public WordPair(String first, String second)
public String getFirst()
public String getSecond()
```

```
public class WordPairList
```

```
private ArrayList<WordPair> allPairs

public WordPairList(String[] words)
public int numMatches()
```

Complete the `WordPairList` constructor below.

```
/** Constructs a WordPairList object as described in part (a).
```

- (b) Write the `WordPairList` method `numMatches`. This method returns the number of `WordPair` objects in `allPairs` for which the two strings match.

For example, the following code segment creates a `WordPairList` object.

```
String[] moreWords = {"the", "red", "fox", "the", "red"};
WordPairList exampleThree = new WordPairList(moreWords);
```

After the code segment has executed, the `allPairs` instance variable of `exampleThree` will contain the following `WordPair` objects in some order. The pairs in which the first string matches the second string are shaded for illustration.

```
("the", "red"), ("the", "fox"), ("the", "the"),
("the", "red"), ("red", "fox"), ("red", "the"),
("red", "red"), ("fox", "the"), ("fox", "red"),
("the", "red")
```

The call `exampleThree.numMatches()` should return 2.

Class information for this question

```
public class WordPair

public WordPair(String first, String second)
public String getFirst()
public String getSecond()

public class WordPairList

private ArrayList<WordPair> allPairs

public WordPairList(String[] words)
public int numMatches()
```

Complete method `numMatches` below.

```
/** Returns the number of matches as described in part (b).
 * /
```

1. This question involves identifying and processing the digits of a non-negative integer. The declaration of the `Digits` class is shown below. You will write the constructor and one method for the `Digits` class.

```
public class Digits
{
    /** The list of digits from the number used to construct this object.
     *   The digits appear in the list in the same order in which they appear in the original number.
     */
    private ArrayList<Integer> digitList;

    /** Constructs a Digits object that represents num.
     *   Precondition: num >= 0
     */
    public Digits(int num)
    { /* to be implemented in part (a) */ }

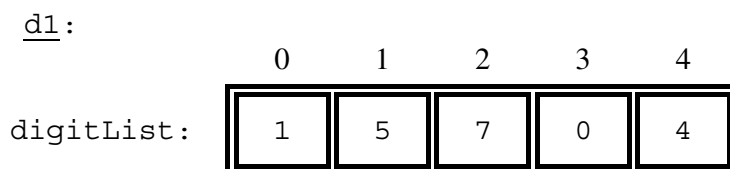
    /** Returns true if the digits in this Digits object are in strictly increasing order;
     *   false otherwise.
     */
    public boolean isStrictlyIncreasing()
    { /* to be implemented in part (b) */ }
}
```

Part (a) begins on page 4.

- (a) Write the constructor for the `Digits` class. The constructor initializes and fills `digitList` with the digits from the non-negative integer `num`. The elements in `digitList` must be `Integer` objects representing single digits, and appear in the same order as the digits in `num`. Each of the following examples shows the declaration of a `Digits` object and the contents of `digitList` as initialized by the constructor.

Example 1

```
Digits d1 = new Digits(15704);
```



Example 2

```
Digits d2 = new Digits(0);
```



WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete the `Digits` constructor below.

```
/** Constructs a Digits object that represents num.  
 *   Precondition: num >= 0  
 */  
public Digits(int num)
```

Part (b) begins on page 6.

- (b) Write the `Digits` method `isStrictlyIncreasing`. The method returns `true` if the elements of `digitList` appear in strictly increasing order; otherwise, it returns `false`. A list is considered strictly increasing if each element after the first is greater than (but not equal to) the preceding element.

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

Method call	Value returned
<code>new Digits(7).isStrictlyIncreasing()</code>	<code>true</code>
<code>new Digits(1356).isStrictlyIncreasing()</code>	<code>true</code>
<code>new Digits(1336).isStrictlyIncreasing()</code>	<code>false</code>
<code>new Digits(1536).isStrictlyIncreasing()</code>	<code>false</code>
<code>new Digits(65310).isStrictlyIncreasing()</code>	<code>false</code>

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `isStrictlyIncreasing` below.

```
/** Returns true if the digits in this Digits object are in strictly increasing order;  
 *      false otherwise.  
 */
```

2. This question involves two classes that are used to process log messages. A list of sample log messages is given below.

```
CLIENT3:security alert - repeated login failures
Webserver:disk offline
SERVER1:file not found
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
Webserver:error on /dev/disk
```

Log messages have the format *machineId:description*, where *machineId* identifies the computer and *description* describes the event being logged. Exactly one colon (":") appears in a log message. There are no blanks either immediately before or immediately after the colon.

The following `LogMessage` class is used to represent a log message.

```
public class LogMessage
{
    private String machineId;
    private String description;

    /** Precondition: message is a valid log message. */
    public LogMessage(String message)
    { /* to be implemented in part (a) */ }

    /** Returns true if the description in this log message properly contains keyword;
     *     false otherwise.
     */
    public boolean containsWord(String keyword)
    { /* to be implemented in part (b) */ }

    public String getMachineId()
    { return machineId; }

    public String getDescription()
    { return description; }

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

- (a) Write the constructor for the `LogMessage` class. It must initialize the private data of the object so that `getMachineId` returns the *machineId* part of the message and `getDescription` returns the *description* part of the message.

Complete the `LogMessage` constructor below.

```
/** Precondition: message is a valid log message. */  
public LogMessage(String message)
```

Part (b) begins on page 8.

(b) Write the `LogMessage` method `containsWord`, which returns `true` if the description in the log message *properly contains* a given keyword and returns `false` otherwise.

A description *properly contains* a keyword if all three of the following conditions are true.

- the keyword is a substring of the description;
- the keyword is either at the beginning of the description or it is immediately preceded by a space;
- the keyword is either at the end of the description or it is immediately followed by a space.

The following tables show several examples. The descriptions in the left table properly contain the keyword `"disk"`. The descriptions in the right table do not properly contain the keyword `"disk"`.

Descriptions that properly contain <code>"disk"</code>
<code>"disk"</code>
<code>"error on disk"</code>
<code>"error on /dev/disk disk"</code>
<code>"error on disk DSK1"</code>

Descriptions that do not properly contain <code>"disk"</code>
<code>"DISK"</code>
<code>"error on disk3"</code>
<code>"error on /dev/disk"</code>
<code>"diskette"</code>

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `containsWord` below.

```
/** Returns true if the description in this log message properly contains keyword;  
 *      false otherwise.  
 */  
public boolean containsWord(String keyword)
```

Part (c) begins on page 10.

- (c) The `SystemLog` class represents a list of `LogMessage` objects and provides a method that removes and returns a list of all log messages (if any) that properly contain a given keyword. The messages in the returned list appear in the same order in which they originally appeared in the system log. If no message properly contains the keyword, an empty list is returned. The declaration of the `SystemLog` class is shown below.

```
public class SystemLog
{
    /** Contains all the entries in this system log.
     *  Guaranteed not to be null and to contain only non-null entries.
     */
    private List<LogMessage> messageList;

    /** Removes from the system log all entries whose descriptions properly contain keyword,
     *  and returns a list (possibly empty) containing the removed entries.
     *  Postcondition:
     *  - Entries in the returned list properly contain keyword and
     *    are in the order in which they appeared in the system log.
     *  - The remaining entries in the system log do not properly contain keyword and
     *    are in their original order.
     *  - The returned list is empty if no messages properly contain keyword.
     */
    public List<LogMessage> removeMessages(String keyword)
    { /* to be implemented in part (c) */ }

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

Write the `SystemLog` method `removeMessages`, which removes from the system log all entries whose descriptions properly contain `keyword` and returns a list of the removed entries in their original order. For example, assume that `theLog` is a `SystemLog` object initially containing six `LogMessage` objects representing the following list of log messages.

```
CLIENT3:security alert - repeated login failures
Webserver:disk offline
SERVER1:file not found
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
Webserver:error on /dev/disk
```

The call `theLog.removeMessages("disk")` would return a list containing the `LogMessage` objects representing the following log messages.

```
Webserver:disk offline
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
```

After the call, `theLog` would contain the following log messages.

```
CLIENT3:security alert - repeated login failures
SERVER1:file not found
Webserver:error on /dev/disk
```


Assume that the `LogMessage` class works as specified, regardless of what you wrote in parts (a) and (b). You must use `containsWord` appropriately to receive full credit.

Complete method `removeMessages` below.

```
/** Removes from the system log all entries whose descriptions properly contain keyword,
 * and returns a list (possibly empty) containing the removed entries.
 * Postcondition:
 *   - Entries in the returned list properly contain keyword and
 *     are in the order in which they appeared in the system log.
 *   - The remaining entries in the system log do not properly contain keyword and
 *     are in their original order.
 *   - The returned list is empty if no messages properly contain keyword.
 */
public List<LogMessage> removeMessages(String keyword)
```

3. A two-dimensional array of integers in which most elements are zero is called a *sparse array*. Because most elements have a value of zero, memory can be saved by storing only the non-zero values along with their row and column indexes. The following complete `SparseArrayEntry` class is used to represent non-zero elements in a sparse array. A `SparseArrayEntry` object cannot be modified after it has been constructed.

```
public class SparseArrayEntry
{
    /** The row index and column index for this entry in the sparse array */
    private int row;
    private int col;

    /** The value of this entry in the sparse array */
    private int value;

    /** Constructs a SparseArrayEntry object that represents a sparse array element
     * with row index r and column index c, containing value v.
     */
    public SparseArrayEntry(int r, int c, int v)
    {
        row = r;
        col = c;
        value = v;
    }

    /** Returns the row index of this sparse array element. */
    public int getRow()
    { return row; }

    /** Returns the column index of this sparse array element. */
    public int getCol()
    { return col; }

    /** Returns the value of this sparse array element. */
    public int getValue()
    { return value; }
}
```

The `SparseArray` class represents a sparse array. It contains a list of `SparseArrayEntry` objects, each of which represents one of the non-zero elements in the array. The entries representing the non-zero elements are stored in the list in no particular order. Each non-zero element is represented by exactly one entry in the list.

```
public class SparseArray
{
    /** The number of rows and columns in the sparse array. */
    private int numRows;
    private int numCols;

    /** The list of entries representing the non-zero elements of the sparse array. Entries are stored in the
     * list in no particular order. Each non-zero element is represented by exactly one entry in the list.
     */
    private List<SparseArrayEntry> entries;

    /** Constructs an empty SparseArray. */
    public SparseArray()
    { entries = new ArrayList<SparseArrayEntry>(); }

    /** Returns the number of rows in the sparse array. */
    public int getNumRows()
    { return numRows; }

    /** Returns the number of columns in the sparse array. */
    public int getNumCols()
    { return numCols; }

    /** Returns the value of the element at row index row and column index col in the sparse array.
     * Precondition:  $0 \leq \text{row} < \text{getNumRows}()$ 
     *  $0 \leq \text{col} < \text{getNumCols}()$ 
     */
    public int getValueAt(int row, int col)
    { /* to be implemented in part (a) */ }

    /** Removes the column col from the sparse array.
     * Precondition:  $0 \leq \text{col} < \text{getNumCols}()$ 
     */
    public void removeColumn(int col)
    { /* to be implemented in part (b) */ }

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

The following table shows an example of a two-dimensional sparse array. Empty cells in the table indicate zero values.

	0	1	2	3	4
0					
1		5			4
2	1				
3		-9			
4					
5					

The sample array can be represented by a `SparseArray` object, `sparse`, with the following instance variable values. The items in `entries` are in no particular order; one possible ordering is shown below.

`numRows: 6`

`numCols: 5`

`entries:`

<table><tr><td>row:</td><td>1</td></tr><tr><td>col:</td><td>4</td></tr><tr><td>value:</td><td>4</td></tr></table>	row:	1	col:	4	value:	4	<table><tr><td>row:</td><td>2</td></tr><tr><td>col:</td><td>0</td></tr><tr><td>value:</td><td>1</td></tr></table>	row:	2	col:	0	value:	1	<table><tr><td>row:</td><td>3</td></tr><tr><td>col:</td><td>1</td></tr><tr><td>value:</td><td>-9</td></tr></table>	row:	3	col:	1	value:	-9	<table><tr><td>row:</td><td>1</td></tr><tr><td>col:</td><td>1</td></tr><tr><td>value:</td><td>5</td></tr></table>	row:	1	col:	1	value:	5
row:	1																										
col:	4																										
value:	4																										
row:	2																										
col:	0																										
value:	1																										
row:	3																										
col:	1																										
value:	-9																										
row:	1																										
col:	1																										
value:	5																										

- (a) Write the `SparseArray` method `getValueAt`. The method returns the value of the sparse array element at a given row and column in the sparse array. If the list `entries` contains an entry with the specified row and column, the value associated with the entry is returned. If there is no entry in `entries` corresponding to the specified row and column, 0 is returned.

In the example above, the call `sparse.getValueAt(3, 1)` would return -9, and `sparse.getValueAt(3, 3)` would return 0.

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Part (a) continues on page 12.

Complete method `getValueAt` below.

```
/** Returns the value of the element at row index row and column index col in the sparse array.  
 * Precondition:  $0 \leq \text{row} < \text{getNumRows}()$   
 *  $0 \leq \text{col} < \text{getNumCols}()$   
 */  
public int getValueAt(int row, int col)
```

Part (b) begins on page 13.

(b) Write the `SparseArray` method `removeColumn`. After removing a specified column from a sparse array:

- All entries in the list `entries` with column indexes matching `col` are removed from the list.
- All entries in the list `entries` with column indexes greater than `col` are replaced by entries with column indexes that are decremented by one (moved one column to the left).
- The number of columns in the sparse array is adjusted to reflect the column removed.

The sample object `sparse` from the beginning of the question is repeated for your convenience.

	0	1	2	3	4
0					
1		5			4
2	1				
3		-9			
4					
5					

The shaded entries in `entries`, below, correspond to the shaded column above.

`numRows`: 6

`numCols`: 5

<code>entries</code> :	row: 1 col: 4 value: 4	row: 2 col: 0 value: 1	row: 3 col: 1 value: -9	row: 1 col: 1 value: 5
------------------------	---	---	--	---

When `sparse` has the state shown above, the call `sparse.removeColumn(1)` could result in `sparse` having the following values in its instance variables (since `entries` is in no particular order, it would be equally valid to reverse the order of its two items). The shaded areas below show the changes.

`numRows`: 6

`numCols`: 4

<code>entries</code> :	row: 1 col: 3 value: 4	row: 2 col: 0 value: 1
------------------------	---	---

Class information repeated from the beginning of the question

```
public class SparseArrayEntry

public SparseArrayEntry(int r, int c, int v)
public int getRow()
public int getCol()
public int getValue()

public class SparseArray

private int numRows
private int numCols
private List<SparseArrayEntry> entries
public int getNumRows()
public int getNumCols()
public int getValueAt(int row, int col)
public void removeColumn(int col)
```

Complete method `removeColumn` below.

```
/** Removes the column col from the sparse array.
 * Precondition:  $0 \leq \text{col} < \text{getNumCols}()$ 
 */
public void removeColumn(int col)
```

4. This question involves the design of an interface, writing a class that implements the interface, and writing a method that uses the interface.
- (a) A *number group* represents a group of integers defined in some way. It could be empty, or it could contain one or more integers.

Write an interface named `NumberGroup` that represents a group of integers. The interface should have a single `contains` method that determines if a given integer is in the group. For example, if `group1` is of type `NumberGroup`, and it contains only the two numbers `-5` and `3`, then `group1.contains(-5)` would return `true`, and `group1.contains(2)` would return `false`.

Write the complete `NumberGroup` interface. It must have exactly one method.

Part (b) begins on page 17.

- (b) A *range* represents a number group that contains all (and only) the integers between a minimum value and a maximum value, inclusive.

Write the `Range` class, which is a `NumberGroup`. The `Range` class represents the group of `int` values that range from a given minimum value up through a given maximum value, inclusive. For example, the declaration

```
NumberGroup range1 = new Range(-3, 2);
```

represents the group of integer values -3, -2, -1, 0, 1, 2.

Write the complete `Range` class. Include all necessary instance variables and methods as well as a constructor that takes two `int` parameters. The first parameter represents the minimum value, and the second parameter represents the maximum value of the range. You may assume that the minimum is less than or equal to the maximum.

Part (c) begins on page 18.

- (c) The `MultipleGroups` class (not shown) represents a collection of `NumberGroup` objects and is a `NumberGroup`. The `MultipleGroups` class stores the number groups in the instance variable `groupList` (shown below), which is initialized in the constructor.

```
private List<NumberGroup> groupList;
```

Write the `MultipleGroups` method `contains`. The method takes an integer and returns `true` if and only if the integer is contained in one or more of the number groups in `groupList`.

For example, suppose `multiple1` has been declared as an instance of `MultipleGroups` and consists of the three ranges created by the calls `new Range(5, 8)`, `new Range(10, 12)`, and `new Range(1, 6)`. The following table shows the results of several calls to `contains`.

Call	Result
<code>multiple1.contains(2)</code>	<code>true</code>
<code>multiple1.contains(9)</code>	<code>false</code>
<code>multiple1.contains(6)</code>	<code>true</code>

Complete method `contains` below.

```
/** Returns true if at least one of the number groups in this multiple group contains num;  
 *      false otherwise.  
 */  
public boolean contains(int num)
```

STOP

END OF EXAM