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()
```

1. This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
public class WordMatch
{
    /** The secret string. */
    private String secret;

    /** Constructs a WordMatch object with the given secret string of lowercase letters. */
    public WordMatch(String word)
    {
        /* implementation not shown */
    }

    /** Returns a score for guess, as described in part (a).
     *  Precondition: 0 < guess.length() <= secret.length()
     */
    public int scoreGuess(String guess)
    {  /* to be implemented in part (a) */  }

    /** Returns the better of two guesses, as determined by scoreGuess and the rules for a
     *  tie-breaker that are described in part (b).
     *  Precondition: guess1 and guess2 contain all lowercase letters.
     *                guess1 is not the same as guess2.
     */
    public String findBetterGuess(String guess1, String guess2)
    {  /* to be implemented in part (b) */  }
}
```

(a) Write the `WordMatch` method `scoreGuess`. To determine the score to be returned, `scoreGuess` finds the number of times that `guess` occurs as a substring of `secret` and then multiplies that number by the square of the length of `guess`. Occurrences of `guess` may overlap within `secret`.

Assume that the length of `guess` is less than or equal to the length of `secret` and that `guess` is not an empty string.

The following examples show declarations of a `WordMatch` object. The tables show the outcomes of some possible calls to the `scoreGuess` method.

```
WordMatch game = new WordMatch("mississippi");
```

Value of `guess`	**Number of Substring Occurrences**	**Score Calculation**: (**Number of Substring Occurrences**) x (**Square of the Length of** `guess`)	**Return Value of** `game.scoreGuess(guess)`
`"i"`	4	`4 * 1 * 1 = 4`	4
`"iss"`	2	`2 * 3 * 3 = 18`	18
`"issipp"`	1	`1 * 6 * 6 = 36`	36
`"mississippi"`	1	`1 * 11 * 11 = 121`	121

```
WordMatch game = new WordMatch("aaaabb");
```

Value of `guess`	**Number of Substring Occurrences**	**Score Calculation**: (**Number of Substring Occurrences**) x (**Square of the Length of** `guess`)	**Return Value of** `game.scoreGuess(guess)`
`"a"`	4	`4 * 1 * 1 = 4`	4
`"aa"`	3	`3 * 2 * 2 = 12`	12
`"aaa"`	2	`2 * 3 * 3 = 18`	18
`"aabb"`	1	`1 * 4 * 4 = 16`	16
`"c"`	0	`0 * 1 * 1 = 0`	0

Complete the `scoreGuess` method.

```
/** Returns a score for guess, as described in part (a).
 *  Precondition: 0 < guess.length() <= secret.length()
 */
public int scoreGuess(String guess)
```

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 WordMatch

private String secret

public WordMatch(String word)
public int scoreGuess(String guess)
public String findBetterGuess(String guess1, String guess2)
```

(b) Write the `WordMatch` method `findBetterGuess`, which returns the better guess of its two `String` parameters, `guess1` and `guess2`. If the `scoreGuess` method returns different values for `guess1` and `guess2`, then the guess with the higher score is returned. If the `scoreGuess` method returns the same value for `guess1` and `guess2`, then the alphabetically greater guess is returned.

The following example shows a declaration of a `WordMatch` object and the outcomes of some possible calls to the `scoreGuess` and `findBetterGuess` methods.

```
WordMatch game = new WordMatch("concatenation");
```

Method Call	Return Value	Explanation
`game.scoreGuess("ten");`	`9`	`1 * 3 * 3`
`game.scoreGuess("nation");`	`36`	`1 * 6 * 6`
`game.findBetterGuess("ten", "nation");`	`"nation"`	Since `scoreGuess` returns `36` for `"nation"` and `9` for `"ten"`, the guess with the greater score, `"nation"`, is returned.
`game.scoreGuess("con");`	`9`	`1 * 3 * 3`
`game.scoreGuess("cat");`	`9`	`1 * 3 * 3`
`game.findBetterGuess("con", "cat");`	`"con"`	Since `scoreGuess` returns `9` for both `"con"` and `"cat"`, the alphabetically greater guess, `"con"`, is returned.

Complete method `findBetterGuess`.

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

```
/** Returns the better of two guesses, as determined by scoreGuess and the rules for a
 *  tie-breaker that are described in part (b).
 *  Precondition: guess1 and guess2 contain all lowercase letters.
 *                guess1 is not the same as guess2.
 */
public String findBetterGuess(String guess1, String guess2)
```

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 WordMatch

private String secret

public WordMatch(String word)
public int scoreGuess(String guess)
public String findBetterGuess(String guess1, String guess2)
```

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 $0 \le i < j <$ `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).
 * Precondition:
```

(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).
 */
```

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

```
public class ArrayTester
{

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


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


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


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

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

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

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

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

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

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

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `getColumn` below.

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

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

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

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

Examples of Latin Squares

1	2	3
2	3	1
3	1	2

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

Examples that are NOT Latin Squares

1	2	1
2	1	1
1	1	2

Not a Latin square because the first row contains duplicate values

1	2	3
3	1	2
7	8	9

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

1	2
1	2

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

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

Class information for this question

```
public class ArrayTester

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

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

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

STOP

END OF EXAM