Question 2: Class — 9 points

Canonical solution

9 points

```
public class SignedText
{
   private String firstName;
   private String lastName;

   public SignedText(String first, String last)
   {
      firstName = first;
      lastName = last;
   }

   public String getSignature()
   {
      String sig = "";
      if (!firstName.equals(""))
      {
         sig += firstName.substring(0, 1) + "-";
      }
      sig += lastName;
      return sig;
   }

   public String addSignature(String textStr)
   {
      String sig = getSignature();
      int index = textStr.indexOf(sig);

      if (index == -1)
      {
         return textStr + sig;
      }
      else if (index == 0)
      {
         return textStr.substring(sig.length()) + sig;
      }
      else
      {
         return textStr;
      }
   }
}
```

`SignedText`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class SignedText`	Responses **will not** earn the point if they • declare the class as `private` • include extraneous code outside the class • include `()` with or without parameters in the class header	**1 point**
2	Declares all appropriate `private` instance variable(s) and constructor initializes instance variable(s) using appropriate parameters	Responses **can** still earn the point even if they • create and store the signature using only one `String` instance variable Responses **will not** earn the point if they • declare any instance variables `static` • omit `private` in instance variable declaration • declare variables outside the class • declare an instance variable inside the constructor • fail to use either parameter • assign values to instance variables from an unknown source • assign initial value to local variable instead of instance variable, even if the local variables are declared as `private` • assign parameter(s) to variable(s) with incompatible data type • return a value from the constructor • define the constructor inside a method or define a method inside the constructor	**1 point**

3	Declares constructor header:
	`SignedText(String ___,`
	`String ___)`
		Responses **will not** earn the point if they
		• declare the constructor as `private` or `static`
			1 point

4	Declares method headers:
	`public String getSignature()`
	`public String addSignature(String ___)`
		Responses **will not** earn the point if they
		• omit `public` in either method header or declare either method as something other than `public`
		• use incorrect method name
		• use incorrect return or parameter type
			1 point

5	Compares first name to the empty string	Responses **can** still earn the point even if they • perform the comparison implicitly (e.g., by comparing the string's length to 0)	**1 point**

6	Determines appropriate signature string in both cases (*algorithm*)	Responses **can** still earn the point even if they • fail to return a value in some cases (*return from* `getSignature` *not assessed*) Responses **will not** earn the point if they • construct the correct string but return something else • define another method inside the signature method or vice versa	**1 point**
7	Calls `String` methods using correct syntax throughout the class	Responses **can** still earn the point even if they • call `substring` only one time • call `indexOf`, `contains`, `charAt`, or other appropriate methods Responses **will not** earn the point if they • only call `length` and/or `equals` without using other `String` methods • fail to call at least one `String` method which accesses elements of a string	**1 point**
8	Identifies the three required cases for `addSignature` using appropriate conditions	Responses **can** still earn the point even if they • return an incorrect string in one or more cases	**1 point**
9	`addSignature` returns appropriate `String` in all three cases (*algorithm*)	Responses **can** still earn the point even if they • identify one or more of the three cases incorrectly, as long as they are unambiguously no-match, match-start, and match-end • implement `getSignature` incorrectly Responses **will not** earn the point if they • call `getSignature` incorrectly, or incorrectly implement its functionality in `addSignature` • fail to identify three distinct cases • build correct strings but assign them to cases incorrectly • print the string instead of or in addition to returning it • define another method inside `addSignature` or vice versa	**1 point**

	Question-specific penalties
	None

Alternate canonical:

```
public class SignedText
{
   private String signature;

   public SignedText(String first, String last)
   {
      signature = last;
      if (first.length() > 0)
      {
         signature = first.substring(0, 1) + "-" + last;
      }
   }

   public String getSignature()
   {
      return signature;
   }

   public String addSignature(String textStr)
   {
      String result = textStr;
      int index = textStr.indexOf(signature);
      if (index < 0)
      {
         result += signature;
      }
      else if (index == 0)
      {
         result = result.substring(signature.length()) + signature;
      }
      return result;
   }
}
```

Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares `"int G=99, g=0;"`, then uses `"while (G < 10)"` instead of `"while (g < 10)"`, the context does **not** allow for the reader to assume the use of the lower case variable.*

2025 Digital Decision Rules

- Some non-ASCII characters are not printing correctly in ONE, particularly extended punctuation. Certain kinds of double-quotes may display as â□□; a character that displays as ï¼□ may be a parenthesis, comma, or semicolon (or other punctuation). If the badly displayed character makes sense as one of those, evaluate the response accordingly.
- If there are missing closed-double-quotes or closed-parentheses, assume they are at the end of the line where they opened, immediately before the semicolon or curly bracket (if any).
- Assume an open/left curly bracket `{` immediately after any method header or class header that does not already have one.
- Assume an appropriate amount of closing brackets before any method header to close all open brackets from the previous method or constructor.
- If there are missing curly brackets, clear indentation can "convey intent". Evaluate the response accordingly.
- Inside a method with left-justified code, indentation cannot "convey intent", so missing curly brackets cannot be assumed. Without bracketing or indentation, only the first line of a `while / if / for` is controlled by the statement; with open curly bracket and no indentation cues, the entire remainder of the method is "inside" the statement.

No Penalty

- `:` instead of `;` and vice versa
- `,` instead of `;` and vice versa

Question 1: Methods and Control Structures — 9 points

Canonical solution

(a) — **5 points**

```
public int scoreGuess(String guess)
{
   int count = 0;

   for (int i = 0; i <= secret.length() - guess.length(); i++)
   {
      if (secret.substring(i, i + guess.length()).equals(guess))
      {
         count++;
      }
   }

   return count * guess.length() * guess.length();
}
```

(b) — **4 points**

```
public String findBetterGuess(String guess1, String guess2)
{
   if (scoreGuess(guess1) > scoreGuess(guess2))
   {
      return guess1;
   }
   if (scoreGuess(guess2) > scoreGuess(guess1))
   {
      return guess2;
   }
   if (guess1.compareTo(guess2) > 0)
   {
      return guess1;
   }
   return guess2;
}
```

(a) `scoreGuess`

	Scoring Criteria	Decision Rules	
1	Compares `guess` to a substring of `secret`	Responses **can** still earn the point even if they only call `secret.indexOf(guess)` Responses **will not** earn the point if they use `==` instead of `equals`	**1 point**
2	Uses a substring of `secret` with correct length for comparison with `guess`	Responses **can** still earn the point even if they • only call `secret.indexOf(guess)` • use `==` instead of `equals`	**1 point**
3	Loops through all necessary substrings of `secret` (*no bounds errors*)	Responses **will not** earn the point if they skip overlapping occurrences	**1 point**
4	Counts number of identified occurrences of `guess` within `secret` (*in the context of a condition involving both* `secret` *and* `guess`)	Responses **can** still earn the point even if they • initialize count incorrectly or not at all • identify occurrences incorrectly	**1 point**
5	Calculates and returns correct final score (*algorithm*)	Responses **will not** earn the point if they • initialize count incorrectly or not at all • fail to use a loop • fail to compare `guess` to multiple substrings of `secret` • count the same matching substring more than once • use a changed or incorrect `guess` length when computing the score	**1 point**
		Total for part (a)	**5 points**

(b) `findBetterGuess`

	Scoring Criteria	Decision Rules	
6	Calls `scoreGuess` to get scores for `guess1` and `guess2`	Responses **will not** earn the point if they • fail to include parameters in the method calls • call the method on an object or class other than `this`	**1 point**
7	Compares the scores	Responses **will not** earn the point if they • only compare using `==` or `!=` • fail to use the result of the comparison in a conditional statement	**1 point**
8	Determines which of `guess1` and `guess2` is alphabetically greater	Responses **can** still earn the point even if they reverse the comparison Responses **will not** earn the point if they • reimplement `compareTo` incorrectly • use result of `compareTo` as if `boolean`	**1 point**
9	Returns the identified `guess1` or `guess2` (*algorithm*)	Responses **can** still earn the point even if they • call `scoreGuess` incorrectly • compare strings incorrectly Responses **will not** earn the point if they • reverse a comparison • omit either comparison • fail to return a guess in some case	**1 point**
		Total for part (b)	**4 points**
	Question-specific penalties		
	None		
		Total for question 1	**9 points**

Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares `"int G=99, g=0;"`, then uses `"while (G < 10)"` instead of `"while (g < 10)"`, the context does **not** allow for the reader to assume the use of the lower case variable.*

2018

AP CollegeBoard

AP Computer Science A

Scoring Guidelines

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

- v) Array/collection access confusion (`[] get`)
- w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
- x) Local variables used but none declared
- y) Destruction of persistent data (e.g., changing value referenced by parameter)
- z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length`/`size` confusion for array, String, List, or ArrayList; with or without ( )
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing ; where structure clearly conveys intent
- Missing { } where indentation clearly conveys intent
- Missing ( ) on parameter-less method or constructor invocations
- Missing ( ) around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example,* "ArayList" *instead of* "ArrayList". *As a counterexample, note that if the code declares* "`int G=99, g=0;`", *then uses* "`while (G < 10)`" *instead of* "`while (g < 10)`", *the context does **not** allow for the reader to assume the use of the lower case variable.*

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 1: Frog Simulation

Part (a)	`simulate`	**5 points**

Intent: *Simulate the distance traveled by a hopping frog*

+1 Calls `hopDistance` and uses returned distance to adjust (or represent) the frog's position

+1 Initializes and accumulates the frog's position at most `maxHops` times (*must be in context of a loop*)

+1 Determines if a distance representing multiple hops is at least `goalDistance`

+1 Determines if a distance representing multiple hops is less than starting position

+1 Returns `true` if goal ever reached, `false` if goal never reached or position ever less than starting position

Part (b)	`runSimulations`	**4 points**

Intent: *Determine the proportion of successful frog hopping simulations*

+1 Calls `simulate` the specified number of times (*no bounds errors*)

+1 Initializes and accumulates a count of `true` results

+1 Calculates proportion of successful simulations using `double` arithmetic

+1 Returns calculated value

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 1: Scoring Notes

Part (a) `simulate`			**5 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Calls `hopDistance` and uses returned distance to adjust (or represent) the frog's position	• use `hopDistance()` as a position, like `hopDistance() < 0`	• only use `hopDistance()` as a count, like `hopDistance() < maxHops`
+1	Initializes and accumulates the frog's position at most `maxHops` times (*must be in context of a loop*)		• do not use a loop
+1	Determines if a distance representing multiple hops is at least `goalDistance`	• use some number of hops `*` `hopDistance()` as the frog's final position	
+1	Determines if a distance representing multiple hops is less than starting position		
+1	Returns `true` if goal ever reached, `false` if goal never reached or position ever less than starting position	• have checks for all three conditions and correct return logic based on those checks, even if a check did not earn a point	• do not check all three conditions • only check for `goalDistance` after the loop • only check for starting position after the loop
Part (b) `runSimulations`			**4 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Calls `simulate` the specified number of times (*no bounds errors*)	• do not use the result of calling `simulate`	• do not use a loop
+1	Initializes and accumulates a count of `true` results		• initialize the count inside a loop • do not use a loop
+1	Calculates proportion of successful simulations using `double` arithmetic	• perform the correct calculation on an accumulated value, even if there was an error in the accumulation	• fail to divide by the parameter
+1	Returns calculated value		• calculate values using nonnumeric types • return a count of simulations

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 1: Frog Simulation

Part (a)

```
public boolean simulate()
{
   int position = 0;

   for (int count = 0; count < maxHops; count++)
   {
      position += hopDistance();
      if (position >= goalDistance)
      {
         return true;
      }
      else if (position < 0)
      {
         return false;
      }
   }
   return false;
}
```

Part (b)

```
public double runSimulations(int num)
{
   int countSuccess = 0;

   for (int count = 0; count < num; count++)
   {
      if(simulate())
      {
         countSuccess++;
      }
   }
   return (double)countSuccess / num;
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 2: Word Pair

Part (a)	`WordPairList`	**5 points**

Intent: *Form pairs of strings from an array and add to an* `ArrayList`

+1 Creates new `ArrayList` and assigns to `allPairs`

+1 Accesses all elements of `words` (*no bounds errors*)

+1 Constructs new `WordPair` using distinct elements of `words`

+1 Adds all necessary pairs of elements from word array to `allPairs`

+1 **On exit:** `allPairs` contains all necessary pairs and no unnecessary pairs

Part (b)	`numMatches`	**4 points**

Intent: *Count the number of pairs in an* `ArrayList` *that have the same value*

+1 Accesses all elements in `allPairs` (*no bounds errors*)

+1 Calls `getFirst` or `getSecond` on an element from list of pairs

+1 Compares first and second components of a pair in the list

+1 Counts number of matches of pair-like values

Question-Specific Penalties

-1 (z) Constructor returns a value

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 2: Scoring Notes

Part (a) `WordPairList`			**5 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Creates new `ArrayList` and assigns to `allPairs`	• `allPairs = new ArrayList();` • `allPairs = new ArrayList<>();` • `this.allPairs = ...`	• initialize a local variable that is never assigned to `allPairs`
+1	Accesses all elements of `words` (*no bounds errors*)		
+1	Constructs new `WordPair` using distinct elements of `words`		
+1	Adds all necessary pairs of elements from word array to `allPairs`	• have a loop bounds error • add unnecessary pairs	• improperly add to an `ArrayList`, e.g., `allPairs.get(i) = x;` • only add consecutive pairs `(words[i], words[i+1])`
+1	**On exit:** `allPairs` contains all necessary pairs and no unnecessary pairs	• improperly add to an `ArrayList`, e.g., `allPairs.get(i) = x;` • have a loop bounds error	• add pairs (i, i) or (i, j) where i > j
Part (b) `numMatches`			**4 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Accesses all elements in `allPairs` (*no bounds errors*)		• access elements of `allPairs` as array elements (e.g., `allPairs[i]`)
+1	Calls `getFirst` or `getSecond` on an element from list of pairs		
+1	Compares first and second components of a pair in the list		• compare using ==
+1	Counts number of matches of pair-like values		• fail to initialize the counter

Return is not assessed in part (b).

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 2: Word Pair

Part (a)

```
public WordPairList(String[] words)
{
   allPairs = new ArrayList<WordPair>();

   for (int i = 0; i < words.length-1; i++)
   {
      for (int j = i+1; j < words.length; j++)
      {
         allPairs.add(new WordPair(words[i], words[j]));
      }
   }
}
```

Part (b)

```
public int numMatches()
{
   int count = 0;

   for (WordPair pair: allPairs)
   {
      if (pair.getFirst().equals(pair.getSecond()))
      {
         count++;
      }
   }
   return count;
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

2018 SCORING GUIDELINES

Question 3: Code Word Checker

Class: `CodeWordChecker`	**9 points**

Intent: *Define implementation of a class to determine if a string meets a set of criteria*

+1 Declares header: `public class CodeWordChecker implements StringChecker`

+1 Declares all appropriate `private` instance variables

+3 Constructors

- **+1** Declares headers: `public CodeWordChecker(int __, int __, String __)` and `public CodeWordChecker(String __)`
- **+1** Uses all parameters to initialize instance variables in 3-parameter constructor
- **+1** Uses parameter and default values to initialize instance variables in 1-parameter constructor

+4 `isValid` method

- **+1** Declares header: `public boolean isValid(String __)`
- **+1** Checks for length between min and max inclusive
- **+1** Checks for unwanted string
- **+1** Returns `true` if length is between min and max and does not contain the unwanted string, `false` otherwise

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 3: Scoring Notes

Class `CodeWordChecker`			**9 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Declares header: `public class CodeWordChecker implements StringChecker`	• omit keyword `public`	• declare class `private` • declare class `static`
+1	Declares all appropriate `private` instance variables		• declare variables as `static` • omit keyword `private` • declare variables outside the class
+3	Constructors		
+1	Declares headers: `public CodeWordChecker (int __, int __, String __)` and `public CodeWordChecker (String __)`	• omit keyword `public`	• declare method `static` • declare method `private`
+1	Uses all parameters to initialize instance variables in 3-parameter constructor		• fail to declare instance variables • initialize local variables instead of instance variables • assign variables to parameters
+1	Uses parameter and default values to initialize instance variables in 1-parameter constructor	• initialize instance variables to default values when declared	• fail to declare instance variables • initialize local variables instead of instance variables • assign variables to parameters
+4	`isValid` method		
+1	Declares header: `public boolean isValid (String __)`		• fail to declare method `public` • declare method `static`
+1	Checks for length between min and max inclusive		• fail to use instance variables • fail to declare the method header
+1	Checks for unwanted string		• fail to use instance variables • fail to declare the method header
+1	Returns `true` if length is between min and max and does not contain the unwanted string, `false` otherwise	• have incorrect checks for length and/or containment, but return the correct value based on those checks	• fail to declare the method header • fail to return in all cases • only check one substring location for containment

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 3: Code Word Checker

```
public class CodeWordChecker implements StringChecker
{
   private int minLength;
   private int maxLength;
   private String notAllowed;

   public CodeWordChecker(int minLen, int maxLen, String symbol)
   {
      minLength = minLen;
      maxLength = maxLen;
      notAllowed = symbol;
   }

   public CodeWordChecker(String symbol)
   {
      minLength = 6;
      maxLength = 20;
      notAllowed = symbol;
   }

   public boolean isValid(String str)
   {
      return str.length() >= minLength && str.length() <= maxLength &&
             str.indexOf(notAllowed) == -1;
   }

}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 4: Latin Squares

Part (a)	`getColumn`	**4 points**

Intent: *Create a 1-D array that contains the values from one column of a 2-D array*

+1 Constructs a new `int` array of size `arr2D.length`

+1 Accesses all items in one column of `arr2D` (*no bounds errors*)

+1 Assigns one element from `arr2D` to the corresponding element in the new array

+1 **On exit:** The new array has all the elements from the specified column in `arr2D` in the correct order

Part (b)	`isLatin`	**5 points**

Intent: *Check conditions to determine if a square 2-D array is a Latin square*

+1 Calls `containsDuplicates` referencing a row or column of `square`

+1 Calls `hasAllValues` referencing two different rows, two different columns, or one row and one column

+1 Applies `hasAllValues` to all rows or all columns (*no bounds errors*)

+1 Calls `getColumn` to obtain a valid column from `square`

+1 Returns `true` if all three Latin square conditions are satisfied, `false` otherwise

Question-Specific Penalties

-1 (r) incorrect construction of a copy of a row

-1 (s) syntactically incorrect method call to any of `getColumn()`, `containsDuplicates()`, or `hasAllValues()`

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 4: Scoring Notes

Part (a) `getColumn`			**4 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Constructs a new `int` array of size `arr2D.length`		• only create an `ArrayList`
+1	Accesses all items in one column of `arr2D` (*no bounds errors*)	• declare the new array of an incorrect size and use that size as the number of loop iterations	• switch row and column indices
+1	Assigns one element from `arr2D` to the corresponding element in the new array		• use `ArrayList` methods to add to array
+1	**On exit:** The new array has all the elements from the specified column in `arr2D` in the correct order		• switch row and column indices • do not use an index when assigning values to the array
Part (b) `isLatin`			**5 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Calls `containsDuplicates` referencing a row or column of `square`	• reference any row or column of `square`, even if the syntax of the reference is incorrect	
+1	Calls `hasAllValues` referencing two different rows, two different columns, or one row and one column	• reference any two distinct rows, two distinct columns, or a row and column of `square`, even if the syntax of the reference is incorrect	
+1	Applies `hasAllValues` to all rows or all columns (*no bounds errors*)		• only reference one array in the call to `hasAllValues`
+1	Calls `getColumn` to obtain a valid column from `square`		• reverse parameters
+1	Returns `true` if all three Latin square conditions are satisfied, `false` otherwise	• test the three sets of conditions and return the correct value	

Return is not assessed in Part (a).

AP® COMPUTER SCIENCE A
2018 SCORING GUIDELINES

Question 4: Latin Squares

Part (a)

```
public static int[] getColumn(int[][] arr2D, int c)
{
   int[] result = new int[arr2D.length];

   for (int r = 0; r < arr2D.length; r++)
   {
      result[r] = arr2D[r][c];
   }
   return result;
}
```

Part (b)

```
public static boolean isLatin(int[][] square)
{
   if (containsDuplicates(square[0]))
   {
      return false;
   }

   for (int r = 1; r < square.length; r++)
   {
      if (!hasAllValues(square[0], square[r]))
      {
         return false;
      }
   }

   for (int c = 0; c < square[0].length; c++)
   {
      if (!hasAllValues(square[0], getColumn(square, c)))
      {
         return false;
      }
   }

   return true;
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

2017

AP CollegeBoard

AP Computer Science A

Scoring Guidelines

2017 GENERAL SCORING GUIDELINES

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)

w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

x) Local variables used but none declared

y) Destruction of persistent data (e.g., changing value referenced by parameter)

z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `()`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "*ArrayList*" instead of "*ArrayList.*" As a counterexample, note that if the code declares "`int G=99, g=0;`", then uses "`while (G < 10)`" instead of "`while (g < 10)`", the context does **not** allow for the reader to assume the use of the lower case variable.*

2017 SCORING GUIDELINES

Question 1: Digits

Part (a)	`Digits` constructor	**5 points**

Intent: *Initialize instance variable using passed parameter*

+1 Constructs `digitList`

+1 Identifies a digit in `num`

+1 Adds at least one identified digit to a list

+1 Adds all identified digits to a list (*must be in context of a loop)*

+1 **On exit:** `digitList` contains all and only digits of `num` in the correct order

Part (b)	`isStrictlyIncreasing`	**4 points**

Intent: *Determine whether or not elements in `digitList` are in increasing order*

+1 Compares at least one identified consecutive pair of `digitList` elements

+1 Determines if a consecutive pair of `digitList` is out of order (*must be in context of a `digitList` traversal*)

+1 Compares all necessary consecutive pairs of elements (*no bounds errors*)

+1 Returns `true` iff all consecutive pairs of elements are in order; returns `false` otherwise

Question-Specific Penalties

-2 (q) Uses confused identifier instead of `digitList`

2017 SCORING GUIDELINES

Question 1: Scoring Notes

Part (a) `Digits` constructor			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Constructs `digitList`		• initialize a local variable instead of `digitList` • create an `ArrayList<int>`
+1	Identifies a digit in `num`	• identify one digit of `num` or a length one substring/character of the `String` representation of `num`	• treat `num` itself as a `String` • convert `num` to a `String` incorrectly
+1	Adds at least one identified digit to a list	• call `add` for some `ArrayList` using the previously identified digit, even if that digit was identified incorrectly	• add `String` or `char` to `digitList` without proper conversion to the correct type
+1	Adds all identified digits to a list (*must be in the context of a loop*)	• call `add` for some `ArrayList` using previously identified digits, even if those digits were identified incorrectly	• identify only 1 digit
+1	**On exit:** `digitList` contains all and only digits of `num` in the correct order	• add to `digitList` even if it is not instantiated properly	• obtain a list with the digits in reverse order • omit one or more digits • add extra digits • mishandle edge case, e.g., 0 or 10 • make a bounds error processing the `String` representation of `num`
Part (b) `isStrictlyIncreasing`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Compares at least one identified consecutive pair of `digitList` elements	• compare two consecutive `Integers` using `compareTo` • explicitly convert two consecutive `Integers` to `ints` and compare those with >=, <= etc. • use auto-unboxing to convert two consecutive `Integers` to `ints` and compare those with >=, <= etc.	• access `digitList` as an array or string • fail to call `.get()` • compare using `!>`
+1	Determines if a consecutive pair of `digitList` is out of order (*must be in context of a `digitList` traversal*)	• determine the correct relationship between the two compared consecutive elements, even if the syntax of the comparison is incorrect	• fail to consider the case where the two elements are equal for the false case
+1	Compares all necessary consecutive pairs of elements (*no bounds errors*)		• return early
+1	Returns `true` iff all consecutive pairs of elements are in order; returns `false` otherwise	• compare consecutive pairs for inequality, but fail to consider the case when two elements are equal	• return prematurely via `if (...) return false; else return true;`

2017 SCORING GUIDELINES

Question 1: Digits

Part (a)

```
public Digits(int num)
{
      digitList = new ArrayList<Integer>();

      if (num == 0)
      {
            digitList.add(new Integer(0));
      }

      while (num > 0)
      {
            digitList.add(0, new Integer(num % 10));
            num /= 10;
      }
}
```

Part (b)

```
public boolean isStrictlyIncreasing()
{
      for (int i = 0; i < digitList.size()-1; i++)
      {
            if (digitList.get(i).intValue() >= digitList.get(i+1).intValue())
            {
                  return false;
            }
      }
      return true;
}
```

Note: The solutions shown above were written in compliance with the AP Java subset methods listed for `Integer` objects. Students were allowed to use the automatic "boxing" and "unboxing" of `Integer` objects in their solutions, which eliminates the need to use `"new Integer(...)"` in part (a) and `"intValue()"` in part (b).

2017 SCORING GUIDELINES

Question 2: MultPractice

Class: `MultPractice`	**9 points**

Intent: *Define implementation of class to produce multiplication practice problems*

+1 Declares header: `public class MultPractice implements StudyPractice`

+1 Declares all necessary `private` instance variables

+2 Constructor

- **+1** Declares header: `public MultPractice(int __, int __)`
- **+1** Initializes all instance variables using parameters

+3 `getProblem` method

- **+1** Declares header: `public String getProblem()`
- **+1** Builds string with current values of instance variables
- **+1** Returns constructed string

+2 `nextProblem` method

- **+1** Declares header: `public void nextProblem()`
- **+1** Updates instance variable(s) to reflect incremented second number

2017 SCORING GUIDELINES

Question 2: Scoring Notes

Class `MultPractice`			**9 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Declares header: `public class MultPractice implements StudyPractice`	• omit keyword `public`	• declare class `private`
+1	Declares all necessary `private` instance variables	• declare the unchanging instance variable as `final`	• declare variables as `static` • omit keyword `private`
+2	Constructor		
+1	Declares header: `public MultPractice (int ___, int ___)`	• omit keyword `public`	
+1	Initializes all instance variables using parameters		• fail to declare nonlocal variables • initialize local variables instead of instance variables • assign variables to parameters
+3	`getProblem` method		
+1	Declares header: `public String getProblem()`		• fail to declare method `public`
+1	Builds string with current values of instance variables	• write appropriate code in a method other than `getProblem` • make capitalization or spacing errors	• fail to declare nonlocal variables • fail to use instance variables • miscast `(String) intVar` • call `intVar.toString()`
+1	Returns constructed string		• return a literal string
+2	`nextProblem` method		
+1	Declares header: `public void nextProblem()`		• fail to declare method `public`
+1	Updates instance variable(s) to reflect incremented second number		• fail to declare non-local variables

2017 SCORING GUIDELINES

Question 2: MultPractice

```
public class MultPractice implements StudyPractice
{
      private int first;
      private int second;

      public MultPractice(int num1, int num2)
      {
            first = num1;
            second = num2;
      }

      public String getProblem()
      {
            return first + " TIMES " + second;
      }

      public void nextProblem()
      {
            second++;
      }
}
```

2017 SCORING GUIDELINES

Question 3: PhraseEditor

Part (a)	`replaceNthOccurrence`	**5 points**

Intent: *Replace the nth occurrence of a given string with a given replacement*

+1 Calls `findNthOccurrence` to find the index of the `nth` occurrence

+1 Preserves `currentPhrase` only if `nth` occurrence does not exist

+1 Identifies components of `currentPhrase` to retain (uses `substring` to extract before/after)

+1 Creates replacement string using identified components and `repl`

+1 Assigns replacement string to instance variable (`currentPhrase`)

Part (b)	`findLastOccurrence`	**4 points**

Intent: *Return the index of the last occurrence of a given string*

+1 Calls `findNthOccurrence` to find the index of the *n*th occurrence

+1 Increments (or decrements) the value used as *n* when finding *n*th occurrence

+1 Returns the index of the last occurrence, if it exists

+1 Returns -1 only when no occurrences exist

Question-Specific Penalties

-1 (q) Uses `currentPhrase.findNthOccurrence`

-2 (r) Confused identifier instead of `currentPhrase`

2017 SCORING GUIDELINES

Question 3: Scoring Notes

Part (a) `replaceNthOccurrence`			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Calls `findNthOccurrence` to find the index of the `nth` occurrence	• do not use the result of calling `findNthOccurrence`	
+1	Preserves `currentPhrase` only if `nth` occurrence does not exist		• fail to use a conditional
+1	Identifies components of `currentPhrase` to retain (uses `substring` to extract before/after)	• identify start and end of substring to be replaced	
+1	Creates replacement string using identified components and `repl`		• create a replacement string that is out of order
+1	Assigns replacement string to instance variable (`currentPhrase`)		
Part (b) `findLastOccurrence`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Calls `findNthOccurrence` to find the index of the *n*th occurrence	• do not use the result of calling `findNthOccurrence`	• `return currentPhrase.lastIndexOf(str);` • call `findNthOccurrence` with an integer parameter of 0
+1	Increments (or decrements) the value used as *n* when finding *n*th occurrence	• `return currentPhrase.lastIndexOf(str);` • advance through `currentPhrase` searching for *n*th occurrence of `str`	
+1	Returns the index of the last occurrence, if it exists	• `return currentPhrase.lastIndexOf(str);` • compute the correct value to be returned in all cases, but no return statement exists for any case	• shorten string being searched • always return in first iteration of the loop
+1	Returns -1 only when no occurrences exist	• `return currentPhrase.lastIndexOf(str);`	• compute the correct value to be returned in all cases, but no return statement exists for any case • always return in first iteration of the loop

2017 SCORING GUIDELINES

Question 3: PhraseEditor

Part (a)

```
public void replaceNthOccurrence(String str, int n, String repl)
{
      int loc = findNthOccurrence(str, n);

      if (loc != -1)
      {
            currentPhrase = currentPhrase.substring(0, loc) + repl +
                            currentPhrase.substring(loc + str.length());
      }
}
```

Part (b)

```
public int findLastOccurrence(String str)
{
      int n = 1;
      while (findNthOccurrence(str, n+1) != -1)
      {
            n++;
      }
      return findNthOccurrence(str, n);
}
```

2017 SCORING GUIDELINES

Question 4: Successor Array

Part (a)	`findPosition`	**5 points**

Intent: *Find the position of a given integer in a 2D integer array*

+1 Accesses all necessary elements of `intArr` (*no bounds errors*)

+1 Identifies `intArr` element equal to `num` (*in context of an* `intArr` *traversal*)

+1 Constructs `Position` object with same row and column as identified `intArr` element

+1 Selects constructed object when `intArr` element identified; `null` when not

+1 Returns selected value

Part (b)	`getSuccessorArray`	**4 points**

Intent: *Create a successor array based on a 2D integer array*

+1 Creates 2D array of `Position` objects with same dimensions as `intArr`

+1 Assigns a value to a location in 2D successor array using a valid call to `findPosition`

+1 Determines the successor `Position` of an `intArr` element accessed by row and column (*in context of* `intArr` *traversal*)

+1 Assigns all necessary locations in successor array with corresponding position object or `null` (*no bounds errors*)

Question-Specific Penalties

-1 (s) Uses confused identifier `Arr`

-1 (t) Uses `intArr[].length` as the number of columns

-1 (u) Uses non-existent accessor methods from `Position`

2017 SCORING GUIDELINES

Question 4: Scoring Notes

Part (a) `findPosition`			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Accesses all necessary elements of `intArr` (*no bounds errors*)		• use `if (...) return; else return null;` inside loop • confuse row and column bounds • fail to traverse `intArr`
+1	Identifies `intArr` element equal to `num` (*in context of an `intArr` traversal*)		• use `.equals` instead of `==`
+1	Constructs `Position` object with same row and column as identified `intArr` element		• omit keyword `new` • use `(r,c)` instead of `Position(r,c)`
+1	Selects constructed object when `intArr` element identified; `null` when not	• use `"null"` instead of `null` • construct a `String` object using row and column indices	• use `if (...) return; else return null;` inside loop • use `(r,c)` instead of `Position(r,c)`
+1	Returns selected value		
Part (b) `getSuccessorArray`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Creates 2D array of `Position` objects with same dimensions as `intArr`		• omit keyword `new`
+1	Assigns a value to a location in 2D successor array using a valid call to `findPosition`	• call `Successors.findPosition(…)`	• reimplement the code from `findPosition` • call `findPosition` with a single argument • call `this.findPosition(…)`
+1	Determines the successor `Position` of an `intArr` element accessed by row and column *(in context of `intArr` traversal)*	• reimplement the code from `findPosition`	• call `findPosition` using an integer that is not identified with a location in `intArr` • call `findPosition` with a single argument
+1	Assigns all necessary locations in successor array with corresponding position object or `null` (*no bounds errors*)	• use `SuccessorArray` dimensions correctly, even if `SuccessorArray` was not initialized properly • only assign non-null entries to `SuccessorArray`	• reimplement the code from `findPosition` but mishandle the null case. • fail to traverse `intArr`

Return is not assessed in Part (b).

2017 SCORING GUIDELINES

Question 4: Successor Array

Part (a)

```
public static Position findPosition(int num, int[][] intArr)
{
      for (int row=0; row < intArr.length; row++)
      {
            for (int col=0; col < intArr[0].length; col++)
            {
                  if (intArr[row][col] == num)
                  {
                        return new Position(row, col);
                  }
            }
      }
      return null;
}
```

Part (b)

```
public static Position[][] getSuccessorArray(int[][] intArr)
{
      Position[][] newArr = new Position[intArr.length][intArr[0].length];

      for (int row=0; row < intArr.length; row++)
      {
            for (int col=0; col < intArr[0].length; col++)
            {
                  newArr[row][col] = findPosition(intArr[row][col]+1, intArr);
            }
      }
      return newArr;
}
```

AP® Computer Science A 2016 Scoring Guidelines

2016 GENERAL SCORING GUIDELINES

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)

w) Extraneous code that causes side-effect (e.g., writing to output, failure to compile)

x) Local variables used but none declared

y) Destruction of persistent data (e.g., changing value referenced by parameter)

z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- = instead of == and vice versa
- `length`/`size` confusion for array, String, List, or ArrayList; with or without ( )
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing ; where structure clearly conveys intent
- Missing { } where indentation clearly conveys intent
- Missing ( ) on parameter-less method or constructor invocations
- Missing ( ) around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context. For example,* "ArayList" *instead of* "ArrayList". *As a counter example, note that if the code declares* "Bug bug;"*, then uses* "Bug.move()" *instead of* "bug.move()"*, the context does **not** allow for the reader to assume the object instead of the class.*

2016 SCORING GUIDELINES

Question 1: Random String Chooser

Part (a)	`RandomStringChooser`	**7 points**

Intent: *Define implementation of class to choose a random string*

+1 Uses correct class, constructor, and method headers

+1 Declares appropriate `private` instance variable(s)

+1 Initializes all instance variable(s) (*point lost if parameter not used in any initialization*)

+4 Implements `getNext`

- **+1** Generates a random number in the proper range (*point lost for improper or missing cast*)
- **+1** Chooses a string from instance variable using generated random number
- **+1** Updates state appropriately (*point lost if constructor parameter is altered*)
- **+1** Returns chosen string or `"NONE"` as appropriate

Part (b)	`RandomLetterChooser`	**2 points**

Intent: *Define implementation of a constructor of a class that extends* `RandomStringChooser`

+1 `getSingleLetters(str)`

+1 `super(getSingleLetters(str));` (*point lost if not first statement in constructor*)

2016 CANONICAL SOLUTIONS

Question 1: Random String Chooser

Part (a):

```
public class RandomStringChooser
{
    private List<String> words;

    public RandomStringChooser(String[] wordArray)
    {
       words = new ArrayList<String>();

       for (String singleWord : wordArray)
       {
            words.add(singleWord);
       }
    }

    public String getNext()
    {
       if (words.size() > 0)
       {
            return words.remove((int)(Math.random() * words.size()));
       }
       return "NONE";
    }
}
```

Part (b):

```
    public RandomLetterChooser(String str)
    {
         super(getSingleLetters(str));
    }
```

2016 SCORING GUIDELINES

Question 2: Log Messages

Part (a) `LogMessage` constructor **2 points**

Intent: *Initialize instance variables using passed parameter*

+1 Locates colon

+1 Initializes instance variables with correct parts of the parameter

Part (b) `containsWord` **2 points**

Intent: *Determine whether description properly contains a keyword*

+1 Identifies at least one properly-contained occurrence of `keyword` in `description`

+1 Returns `true` if and only if `description` properly contains `keyword`
Returns `false` otherwise (*no bounds errors*)

Part (c) `removeMessages` **5 points**

Intent: *Remove log messages containing keyword from system log list and return these messages in a new list*

+1 Accesses all items in `messageList` (*no bounds errors; point lost if no removal attempted*)

+1 Identifies keyword-containing entry using `containsWord`

+1 Adds all and only identified entries to new list (*point lost if original order not maintained*)

+1 Removes all identified entries from `messageList` (*point lost if* `messageList` *reordered)*

+1 Constructs and returns `new ArrayList<LogMessage>`

2016 CANONICAL SOLUTIONS

Question 2: Log Messages

Part (a):

```
public LogMessage(String message)
{
      int colon = message.indexOf(":");
      machineId = message.substring(0, colon);
      description = message.substring(colon + 1);
}
```

Part (b):

```
public boolean containsWord(String keyword)
{
   if (description.equals(keyword))
   {   return true;   }
   if (description.indexOf(keyword + " ") == 0)
   {   return true;   }
   if (description.indexOf(" " + keyword + " ") != -1)
   {   return true;   }
   if (description.length() > keyword.length())
   {
       if ((description.substring(description.length() -
                                  keyword.length() - 1).equals(
                                  " " + keyword)))
       {
          return true;
       }
   }
   return false;
}
```

Part (c):

```
public List<LogMessage> removeMessages(String keyword)
{
      List<LogMessage> removals = new ArrayList<LogMessage>();

      for (int i = 0; i < messageList.size(); i++)
      {
            if (messageList.get(i).containsWord(keyword))
            {
                  removals.add(messageList.remove(i));
                  i--;
            }
      }
      return removals;
}
```

2016 SCORING GUIDELINES

Question 3: Crossword

Part (a)	`toBeLabeled`	**3 points**

Intent: *Return a `boolean` value indicating whether a crossword grid square should be labeled with a positive number*

+1 Checks `blackSquares[r][c]`

+1 Checks for black square/border above and black square/border to the left (*no bounds errors*)

+1 Returns `true` if square should be labeled with positive number; returns `false` otherwise

Part (b)	`Crossword` constructor	**6 points**

Intent: *Initialize each square in a crossword puzzle grid to have a color `(boolean)` and an integer label*

+1 `puzzle = new Square[blackSquares.length][blackSquares[0].length];` (*or equivalent*)

+1 Accesses all locations in `puzzle` (*no bounds errors*)

+1 Calls `toBeLabeled` with appropriate parameters

+1 Creates and assigns new `Square` to location in `puzzle`

+1 Numbers identified squares consecutively, in row-major order, starting at 1

+1 On exit: All squares in `puzzle` have correct color and number (*minor errors covered in previous points ok*)

Question-Specific Penalties

-2 (p) Consistently uses incorrect name instead of `puzzle`

-1 (q) Uses *array*`[].length` instead of *array*`[`*num*`].length`

2016 CANONICAL SOLUTIONS

Question 3: Crossword

Part (a):

```
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
{
      return (!(blackSquares[r][c]) &&
              (r == 0 || c == 0 || blackSquares[r - 1][c] ||
               blackSquares[r][c - 1]));
}
```

Part (b):

```
public Crossword(boolean[][] blackSquares)
{
      puzzle = new Square[blackSquares.length][blackSquares[0].length];
      int num = 1;

      for (int r = 0; r < blackSquares.length; r++)
      {
            for (int c = 0; c < blackSquares[0].length; c++)
            {
                  if (blackSquares[r][c])
                  {
                        puzzle[r][c] = new Square(true, 0);
                  }
                  else
                  {
                        if (toBeLabeled(r, c, blackSquares))
                        {
                              puzzle[r][c] = new Square(false, num);
                              num++;
                        }
                        else
                        {
                              puzzle[r][c] = new Square(false, 0);
                        }
                  }
            }
      }
}
```

2016 SCORING GUIDELINES

Question 4: String Formatter

Part (a)	`totalLetters`	**2 points**

Intent: *Calculate the total number of letters in a list of words*

+1 Accesses all strings in `wordList` and adds length of each to accumulated count (*no bounds errors*)

+1 Initializes and returns accumulated count

Part (b)	`basicGapWidth`	**2 points**

Intent: *Calculate the minimum number of spaces (basic gap width) to be placed between each word in the formatted string*

+1 Calls `totalLetters` correctly and uses result

+1 Returns correct calculated value

Part (c)	`format`	**5 points**

Intent: *Return a formatted string consisting of words from `wordList` separated by one or more spaces*

+1 Calls `basicGapWidth` and `leftoverSpaces` correctly and uses results

+1 Adds all strings in `wordList` to formatted string in original order (*no bounds errors*)

+1 Inserts *basicGapWidth* spaces between each pair of words in formatted string

+1 Inserts one space between first *leftoverSpaces* pairs of words in formatted string

+1 Initializes and returns formatted string (*no extra or deleted characters*)

2016 CANONICAL SOLUTIONS

Question 4: String Formatter

Part (a):

```
public static int totalLetters(List<String> wordList)
{
      int total = 0;

      for (String word : wordList)
      {
            total += word.length();
      }
      return total;
}
```

Part (b):

```
public static int basicGapWidth(List<String> wordList, int formattedLen)
{
      return (formattedLen – totalLetters(wordList)) / (wordList.size()-1);
}
```

Part (c):

```
public static String format(List<String> wordList, int formattedLen)
{
      String formatted = "";
      int gapWidth = basicGapWidth(wordList, formattedLen);
      int leftovers = leftoverSpaces(wordList, formattedLen);

      for (int w = 0; w < wordList.size() - 1; w++)
      {
            formatted = formatted + wordList.get(w);
            for (int i = 0; i < gapWidth; i++)
            {
                  formatted = formatted + " ";
            }
            if (leftovers > 0)
            {
                  formatted = formatted + " ";
                  leftovers--;
            }
      }
      formatted = formatted + wordList.get(wordList.size() - 1);

      return formatted;
}
```