
# AP® Computer Science A
# 2011 Free-Response Questions

## About the College Board

The College Board is a mission-driven not-for-profit organization that connects students to college success and opportunity. Founded in 1900, the College Board was created to expand access to higher education. Today, the membership association is made up of more than 5,900 of the world's leading educational institutions and is dedicated to promoting excellence and equity in education. Each year, the College Board helps more than seven million students prepare for a successful transition to college through programs and services in college readiness and college success — including the SAT® and the Advanced Placement Program®. The organization also serves the education community through research and advocacy on behalf of students, educators and schools.



**Visit the College Board on the Web: www.collegeboard.org.**
**AP Central is the official online home for the AP Program: apcentral.collegeboard.com.**

## COMPUTER SCIENCE A
## SECTION II
**Time—1 hour and 45 minutes**
**Number of questions—4**
**Percent of total score—50**

**Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.**

Notes:
- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. Digital sounds can be represented as an array of integer values. For this question, you will write two unrelated methods of the `Sound` class.

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

```
public class Sound
{
  /** the array of values in this sound; guaranteed not to be null */
  private int[] samples;

  /** Changes those values in this sound that have an amplitude greater than limit.
   *  Values greater than limit are changed to limit.
   *  Values less than -limit are changed to -limit.
   *  @param limit the amplitude limit
   *         Precondition: limit ≥ 0
   *  @return the number of values in this sound that this method changed
   */
  public int limitAmplitude(int limit)
  {  /* to be implemented in part (a) */  }

  /** Removes all silence from the beginning of this sound.
   *  Silence is represented by a value of 0.
   *  Precondition: samples contains at least one nonzero value
   *  Postcondition: the length of samples reflects the removal of starting silence
   */
  public void trimSilenceFromBeginning()
  {  /* to be implemented in part (b) */  }

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

(a) The volume of a sound depends on the amplitude of each value in the sound. The amplitude of a value is its absolute value. For example, the amplitude of -2300 is 2300 and the amplitude of 4000 is 4000.

Write the method `limitAmplitude` that will change any value that has an amplitude greater than the given limit. Values that are greater than `limit` are replaced with `limit`, and values that are less than `-limit` are replaced with `-limit`. The method returns the total number of values that were changed in the array. For example, assume that the array `samples` has been initialized with the following values.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 2532 | 17 | -2300 | -17 | -4000 | 2000 | 1048 | -420 | 33 | 15 | -32 | 2030 | 3223 |

When the statement

```
int numChanges = limitAmplitude(2000);
```

is executed, the value of `numChanges` will be 5, and the array `samples` will contain the following values.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 2000 | 17 | -2000 | -17 | -2000 | 2000 | 1048 | -420 | 33 | 15 | -32 | 2000 | 2000 |

Complete method `limitAmplitude` below.

```
/** Changes those values in this sound that have an amplitude greater than limit.
 *  Values greater than limit are changed to limit.
 *  Values less than -limit are changed to -limit.
 *  @param limit the amplitude limit
 *         Precondition: limit ≥ 0
 *  @return the number of values in this sound that this method changed
 */
public int limitAmplitude(int limit)
```

(b) Recorded sound often begins with silence. Silence in a sound is represented by a value of 0.

Write the method `trimSilenceFromBeginning` that removes the silence from the beginning of a sound. To remove starting silence, a new array of values is created that contains the same values as the original `samples` array in the same order but without the leading zeros. The instance variable `samples` is updated to refer to the new array. For example, suppose the instance variable `samples` refers to the following array.

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 0 | 0 | 0 | 0 | -14 | 0 | -35 | -39 | 0 | -7 | 16 | 32 | 37 | 29 | 0 | 0 |

After `trimSilenceFromBeginning` has been called, the instance variable `samples` will refer to the following array.

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | -14 | 0 | -35 | -39 | 0 | -7 | 16 | 32 | 37 | 29 | 0 | 0 |

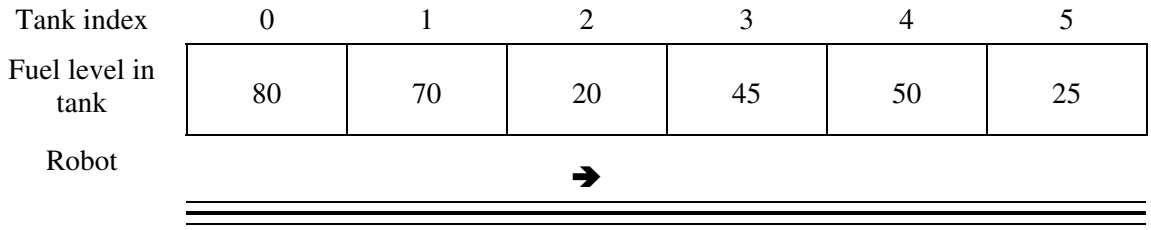
Complete method `trimSilenceFromBeginning` below.

```
/** Removes all silence from the beginning of this sound.
 *  Silence is represented by a value of 0.
 *  Precondition: samples contains at least one nonzero value
 *  Postcondition: the length of samples reflects the removal of starting silence
 */
public void trimSilenceFromBeginning()
```

3. A fuel depot has a number of fuel tanks arranged in a line and a robot that moves a filling mechanism back and forth along the line so that the tanks can be filled. A fuel tank is specified by the `FuelTank` interface below.

```
public interface FuelTank
{
  /** @return an integer value that ranges from 0 (empty) to 100 (full) */
  int getFuelLevel();
}
```

A fuel depot keeps track of the fuel tanks and the robot. The following figure represents the tanks and the robot in a fuel depot. The robot, indicated by the arrow, is currently at index 2 and is facing to the right.




The state of the robot includes the index of its location and the direction in which it is facing (to the right or to the left). This information is specified in the `FuelRobot` interface as shown in the following declaration.

```
public interface FuelRobot
{
  /** @return the index of the current location of the robot */
  int getCurrentIndex();


  /** Determine whether the robot is currently facing to the right
   *  @return true  if the robot is facing to the right (toward tanks with larger indexes)
   *          false if the robot is facing to the left (toward tanks with smaller indexes)
   */
  boolean isFacingRight();


  /** Changes the current direction of the robot */
  void changeDirection();


  /** Moves the robot in its current direction by the number of locations specified.
   *  @param numLocs the number of locations to move. A value of 1 moves
   *                 the robot to the next location in the current direction.
   *          Precondition: numLocs > 0
   */
  void moveForward(int numLocs);
}
```

A fuel depot is represented by the `FuelDepot` class as shown in the following class declaration.

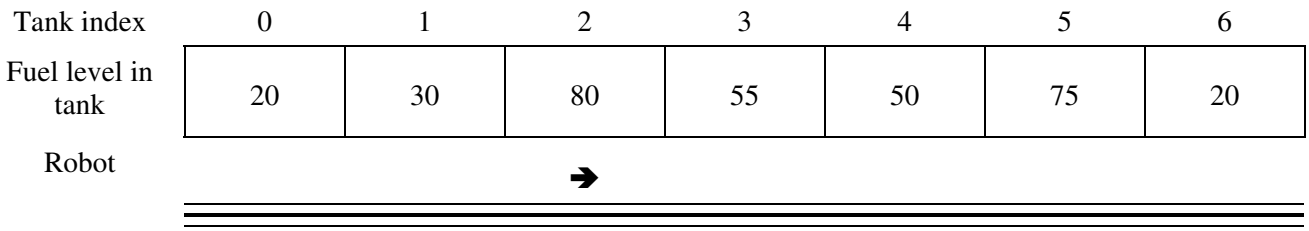
```
public class FuelDepot
{
  /** The robot used to move the filling mechanism */
  private FuelRobot filler;

  /** The list of fuel tanks */
  private List<FuelTank> tanks;

  /** Determines and returns the index of the next tank to be filled.
   *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
   *  @return index of the location of the next tank to be filled
   *  Postcondition: the state of the robot has not changed
   */
  public int nextTankToFill(int threshold)
  {  /* to be implemented in part (a) */  }


  /** Moves the robot to location locIndex.
   *  @param locIndex the index of the location of the tank to move to
   *         Precondition: 0 ≤ locIndex < tanks.size()
   *  Postcondition: the current location of the robot is locIndex
   */
  public void moveToLocation(int locIndex)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `FuelDepot` method `nextTankToFill` that returns the index of the next tank to be filled.

The index for the next tank to be filled is determined according to the following rules:

- Return the index of a tank with the lowest fuel level that is less than or equal to a given threshold. If there is more than one fuel tank with the same lowest fuel level, any of their indexes can be returned.
- If there are no tanks with a fuel level less than or equal to the threshold, return the robot's current index.

For example, suppose the tanks contain the fuel levels shown in the following figure.




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

| `threshold` | Return Value | Rationale |
|---|---|---|
| 50 | 0 or 6 | 20 is the lowest fuel level, so either 0 or 6 can be returned. |
| 15 | 2 | There are no tanks with a fuel level ≤ `threshold`, so the robot's current index is returned. |

Complete method `nextTankToFill` below.

```
/** Determines and returns the index of the next tank to be filled.
 *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
 *  @return index of the location of the next tank to be filled
 *  Postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
```

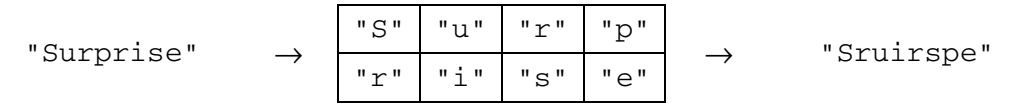
(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```
/** Moves the robot to location locIndex.
 *  @param locIndex the index of the location of the tank to move to
 *         Precondition: 0 ≤ locIndex < tanks.size()
 *  Postcondition: the current location of the robot is locIndex
 */
public void moveToLocation(int locIndex)
```

4. In this question you will write two methods for a class `RouteCipher` that encrypts (puts into a coded form) a message by changing the order of the characters in the message. The route cipher fills a two-dimensional array with single-character substrings of the original message in row-major order, encrypting the message by retrieving the single-character substrings in column-major order.

   For example, the word `"Surprise"` can be encrypted using a 2-row, 4-column array as follows.




   An incomplete implementation of the `RouteCipher` class is shown below.

```
public class RouteCipher
{
  /** A two-dimensional array of single-character strings, instantiated in the constructor */
  private String[][] letterBlock;

  /** The number of rows of letterBlock, set by the constructor */
  private int numRows;

  /** The number of columns of letterBlock, set by the constructor */
  private int numCols;

  /** Places a string into letterBlock in row-major order.
   *  @param str the string to be processed
   *  Postcondition:
   *    if str.length() < numRows * numCols, "A" is placed in each unfilled cell
   *    if str.length() > numRows * numCols, trailing characters are ignored
   */
  private void fillBlock(String str)
  {  /* to be implemented in part (a) */  }

  /** Extracts encrypted string from letterBlock in column-major order.
   *  Precondition: letterBlock has been filled
   *  @return the encrypted string from letterBlock
   */
  private String encryptBlock()
  {  /* implementation not shown */  }

  /** Encrypts a message.
   *  @param message the string to be encrypted
   *  @return the encrypted message;
   *          if message is the empty string, returns the empty string
   */
  public String encryptMessage(String message)
  {  /* to be implemented in part (b) */  }

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

(a) Write the method `fillBlock` that fills the two-dimensional array `letterBlock` with one-character strings from the string passed as parameter `str`.

The array must be filled in row-major order—the first row is filled from left to right, then the second row is filled from left to right, and so on, until all rows are filled.

If the length of the parameter `str` is smaller than the number of elements of the array, the string `"A"` is placed in each of the unfilled cells. If the length of `str` is larger than the number of elements in the array, the trailing characters are ignored.

For example, if `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at noon"`, the resulting contents of `letterBlock` would be as shown in the following table.

| | | | | |
|---|---|---|---|---|
| "M" | "e" | "e" | "t" | " " |
| "a" | "t" | " " | "n" | "o" |
| "o" | "n" | "A" | "A" | "A" |

If `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at midnight"`, the resulting contents of `letterBlock` would be as shown in the following table.

| | | | | |
|---|---|---|---|---|
| "M" | "e" | "e" | "t" | " " |
| "a" | "t" | " " | "m" | "i" |
| "d" | "n" | "i" | "g" | "h" |

The following expression may be used to obtain a single-character string at position `k` of the string `str`.

```
str.substring(k, k + 1)
```

Complete method `fillBlock` below.

```
/** Places a string into letterBlock in row-major order.
 *  @param str the string to be processed
 *  Postcondition:
 *    if str.length() < numRows * numCols, "A" is placed in each unfilled cell
 *    if str.length() > numRows * numCols, trailing characters are ignored
 */
private void fillBlock(String str)
```

(b) Write the method `encryptMessage` that encrypts its string parameter `message`. The method builds an encrypted version of `message` by repeatedly calling `fillBlock` with consecutive, nonoverlapping substrings of `message` and concatenating the results returned by a call to `encryptBlock` after each call to `fillBlock`. When all of `message` has been processed, the concatenated string is returned. Note that if `message` is the empty string, `encryptMessage` returns an empty string.

The following example shows the process carried out if `letterBlock` has 2 rows and 3 columns and `encryptMessage("Meet at midnight")` is executed.

| Substring | `letterBlock` after Call to `fillBlock` | Value Returned by `encryptBlock` | Concatenated String |
|---|---|---|---|
| `"Meet a"` | `"M"` `"e"` `"e"`<br>`"t"` `" "` `"a"` | `"Mte ea"` | `"Mte ea"` |
| `"t midn"` | `"t"` `" "` `"m"`<br>`"i"` `"d"` `"n"` | `"ti dmn"` | `"Mte eati dmn"` |
| `"ight"` | `"i"` `"g"` `"h"`<br>`"t"` `"A"` `"A"` | `"itgAhA"` | `"Mte eati dmnitgAhA"` |

In this example, the method returns the string `"Mte eati dmnitgAhA"`.

Assume that `fillBlock` and `encryptBlock` methods work as specified. Solutions that reimplement the functionality of one or both of these methods will not receive full credit.

Complete method `encryptMessage` below.

```
/** Encrypts a message.
 *  @param message the string to be encrypted
 *  @return the encrypted message;
 *          if message is the empty string, returns the empty string
 */
public String encryptMessage(String message)
```

**STOP**

**END OF EXAM**

# AP® Computer Science A
# 2011 Scoring Guidelines

## The College Board

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the College Board is composed of more than 5,700 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,800 colleges through major programs and services in college readiness, college admission, guidance, assessment, financial aid and enrollment. Among its widely recognized programs are the SAT®, the PSAT/NMSQT®, the Advanced Placement Program® (AP®), SpringBoard® and ACCUPLACER®. The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities and concerns.



**Visit the College Board on the Web: www.collegeboard.org.**
**AP Central is the official online home for the AP Program: apcentral.collegeboard.com.**

# AP® COMPUTER SCIENCE A
# 2011 GENERAL SCORING GUIDELINES

**Apply the question-specific rubric first; the question-specific rubric *always* takes precedence.**
**Penalties:** The penalty categorization below is for cases not covered by the question-specific rubric. Points can only be deducted in a part of the question that has earned credit via the question-specific rubric, and no section may have a negative point total. A given penalty can be assessed **only once** in a question, even if it occurs on different parts of that question. A maximum of 3 penalty points may be assessed over the entire question.

## Nonpenalized Errors

spelling/case discrepancies if no ambiguity*

local variable not declared if other variables are declared in some part

use of keyword as identifier

`[]` vs. `()` vs. `<>`

`=` instead of `==` (and vice versa)

`length`/`size` confusion for array, `String`, and `ArrayList`, with or without `()`

`private` qualifier on local variable

extraneous code with no side effect; *e.g., precondition check*

common mathematical symbols for operators (x • ÷ ≤ ≥ < > ≠)

missing { } where indentation clearly conveys intent and { } used elsewhere

default constructor called without parens;
*e.g.,* `new Critter;`

missing `( )` on parameter-less method call

missing `( )` around `if/while` conditions

missing `;` when majority are present

missing `public` on class or constructor header

extraneous `[]` when referencing entire array

`[i,j]` instead of `[i][j]`

extraneous size in array declaration,
*e.g.,* `int[size] nums = new int[size];`

## Minor Errors (½ point)

confused identifier *(e.g.,* `len` *for* `length` *or* `left()` *for* `getLeft()` *)*

local variables used but none declared

missing `new` in constructor call

modifying a constant (`final`)

use of `equals` or `compareTo` method on primitives, *e.g.,* `int x; …x.equals(val)`

array/collection access confusion (`[]` `get`)

assignment dyslexia,
*e.g.,* `x + 3 = y;` *for* `y = x + 3;`

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, *e.g.,* `a = method(int x)`

missing `public` from method header when required

`"false"`/`"true"` or 0/1 for boolean values

`"null"` for `null`

*Applying Minor Penalties (½ point)*:
A minor infraction that occurs **exactly once** when the same concept is **correct two or more times** is regarded as an oversight and **not penalized**. A minor penalty **must be assessed** if the item is the **only instance, one of two**, or occurs **two or more times**.

## Major Errors (1 point)

extraneous code that causes side effect; *e.g., information written to output*

interface or class name instead of variable identifier; *e.g.,* `Bug.move()` *instead of* `aBug.move()`

`aMethod(obj)` instead of `obj.aMethod()`

attempt to use private data or method when not accessible

destruction of persistent data *(e.g., changing value referenced by parameter)*

use of class name in place of `super` in constructor or method call

`void` method (or constructor) returns a value

* *Spelling and case discrepancies for identifiers fall under the "nonpenalized" category only if the correction can be* ***unambiguously*** *inferred from context; for example, "*`ArayList`*" instead of "*`ArrayList`*". As a counterexample, note that if a student 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.*

# AP® COMPUTER SCIENCE A
# 2011 SCORING GUIDELINES

## Question 1: Sound

**Part (a)** `limitAmplitude` **4½ points**

*Intent: Change elements of* `samples` *that exceed* ±`limit`; *return number of changes made*

**+3** Identify elements of `samples` to be modified and modify as required
- **+1** Consider elements of `samples`
  - **+½** Accesses more than one element of `samples`
  - **+½** Accesses every element of `samples` (*no bounds errors*)
- **+2** Identify and change elements of `samples`
  - **+½** Compares an element of `samples` with `limit`
  - **+½** Changes at least one element to `limit` or `-limit`
  - **+1** Changes all and only elements that exceed ±`limit` to `limit` or `-limit` appropriately

**+1½** Calculate and return number of changed elements of `samples`
- **+1** Initializes and updates a counter to achieve correct number of changed samples
- **+½** Returns value of an updated counter (*requires array access*)

**Part (b)** `trimSilenceFromBeginning` **4½ points**

*Intent: Remove leading elements of* `samples` *that have value of* 0, *potentially resulting in array of different length*

**+1½** Identify leading-zero-valued elements of `samples`
- **+½** Accesses every leading-zero element of `samples`
- **+½** Compares 0 and an element of `samples`
- **+½** Compares 0 and multiple elements of `samples`

**+1** Create array of proper length
- **+½** Determines correct number of elements to be in resulting array
- **+½** Creates new array of determined length

**+2** Remove silence values from `samples`
- **+½** Copies some values other than leading-zero values
- **+1** Copies all and only values other than leading-zero values, preserving original order
- **+½** Modifies instance variable `samples` to reference newly created array

**Question-Specific Penalties**

**–1** Array identifier confusion (e.g., `value` instead of `samples`)
**–½** Array/collection modifier confusion (e.g., using `set`)

# AP® COMPUTER SCIENCE A
# 2011 SCORING GUIDELINES

## Question 3: Fuel Depot

| **Part (a)** | `nextTankToFill` | **5 points** |
|---|---|---|

*Intent:* *Return index of tank with minimum level* (`<= threshold`)

**+4** Determine minimum element of `tanks` that is `<= threshold`, if any
- **+1½** Consider fuel levels of elements of `tanks`
  - **+½** Accesses fuel level of an element of `tanks`
  - **+½** Accesses at least one element of `tanks` in context of repetition (iteration/recursion)
  - **+½** Accesses every element of `tanks` at least once
- **+2½** Identify minimum element of `tanks` that is `<= threshold`
  - **+½** Compares fuel levels from at least two elements of `tanks`
  - **+½** Implements algorithm to find minimum
  - **+½** Identifies tank (*object or index*) holding identified minimum
  - **+½** Compares `threshold` with fuel level from at least one element of `tanks`
  - **+½** Determines element identified as minimum fuel level that is also `<= threshold`

**+1** Return the index of the element satisfying the conditions, or the current index if no element does so
- **+½** Returns index of element identified as satisfying threshold & minimum conditions*
- **+½** Returns `filler.getCurrentIndex()` when no element satisfies conditions*

**Note: Point is not awarded if wrong data type is returned.*

| **Part (b)** | `moveToLocation` | **4 points** |
|---|---|---|

*Intent:* *Move robot to given tank location*

**+2** Ensure robot is pointing in direction of tank to be filled
- **+½** Determines direction `filler` is currently facing
- **+½** Changes `filler`'s direction for some condition
- **+1** Establishes `filler`'s direction as appropriate for all conditions

**+2** Place robot at specified location
- **+½** Invokes `moveForward` method with a parameter
- **+½** Invokes `moveForward` method with a verified non-zero parameter
- **+1** Invokes `filler.moveForward` method with a correctly computed parameter

# AP® COMPUTER SCIENCE A
# 2011 SCORING GUIDELINES

## Question 4: Cipher

**Part (a)** `fillBlock` **3½ points**

*Intent:* *Fill* `letterBlock` *in row-major order from parameter; pad block or truncate string as needed*

- **+½** Copies at least one substring from parameter to `letterBlock`
- **+½** Completely fills `letterBlock` from parameter if possible (*no bounds errors in* `letterBlock` *or parameter*)
- **+1** Results in a distribution of all consecutive one-character substrings from parameter to `letterBlock` (*ignores surplus characters*)
- **+½** Copies these one-character substrings from parameter to `letterBlock` in such a way that the result is in row-major order
- **+1** Pads `letterBlock` with `"A"` if and only if parameter is shorter than block size

**Part (b)** `encryptMessage` **5½ points**

*Intent:* *Return encrypted string created by repeatedly invoking* `fillBlock` *and* `encryptBlock` *on substrings of parameter and concatenating the results*

- **+2** Partition parameter
  - **+½** Returns the empty string if the parameter is the empty string
  - **+½** Creates substrings of parameter that progress through the parameter string (*can overlap or skip*)
  - **+1** Processes every character in parameter exactly once (*no bounds errors*)
- **+3** Fill and encrypt a block, concatenate results
  - **+½** Invokes `fillBlock` with parameter or substring of parameter
  - **+½** Invokes `fillBlock` on more than one substring of parameter
  - **+½** Invokes `encryptBlock` after each invocation of `fillBlock`
  - **+½** Concatenates encrypted substrings of parameter
  - **+1** Builds complete, encrypted message
- **+½** Return resulting built string

**Question-Specific Penalties**

- **–1½** Use of identifier with no apparent resemblance to `letterBlock` for two-dimensional array


# AP® Computer Science A
# 2010 Free-Response Questions

## The College Board

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the College Board is composed of more than 5,700 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,800 colleges through major programs and services in college readiness, college admission, guidance, assessment, financial aid and enrollment. Among its widely recognized programs are the SAT®, the PSAT/NMSQT®, the Advanced Placement Program® (AP®), SpringBoard® and ACCUPLACER®. The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities and concerns.



**Visit the College Board on the Web: www.collegeboard.com.**
**AP Central is the official online home for the AP Program: apcentral.collegeboard.com.**

## COMPUTER SCIENCE A
## SECTION II

**Time—1 hour and 45 minutes**

**Number of questions—4**

**Percent of total score—50**

**Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.**

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. An organization raises money by selling boxes of cookies. A cookie order specifies the variety of cookie and the number of boxes ordered. The declaration of the `CookieOrder` class is shown below.

```
public class CookieOrder
{
  /** Constructs a new CookieOrder object. */
  public CookieOrder(String variety, int numBoxes)
  {  /* implementation not shown */  }

  /** @return the variety of cookie being ordered
   */
  public String getVariety()
  {  /* implementation not shown */  }

  /** @return the number of boxes being ordered
   */
  public int getNumBoxes()
  {  /* implementation not shown */  }

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

The `MasterOrder` class maintains a list of the cookies to be purchased. The declaration of the `MasterOrder` class is shown below.

```
public class MasterOrder
{
  /** The list of all cookie orders */
  private List<CookieOrder> orders;

  /** Constructs a new MasterOrder object. */
  public MasterOrder()
  {  orders = new ArrayList<CookieOrder>();  }

  /** Adds theOrder to the master order.
   *  @param theOrder the cookie order to add to the master order
   */
  public void addOrder(CookieOrder theOrder)
  {  orders.add(theOrder);  }

  /** @return the sum of the number of boxes of all of the cookie orders
   */
  public int getTotalBoxes()
  {  /* to be implemented in part (a) */  }

  /** Removes all cookie orders from the master order that have the same variety of
   *  cookie as cookieVar and returns the total number of boxes that were removed.
   *  @param cookieVar the variety of cookies to remove from the master order
   *  @return the total number of boxes of cookieVar in the cookie orders removed
   */
  public int removeVariety(String cookieVar)
  {  /* to be implemented in part (b) */  }

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

(a) The `getTotalBoxes` method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns 0.

Complete method `getTotalBoxes` below.

```
/** @return the sum of the number of boxes of all of the cookie orders
 */
public int getTotalBoxes()
```

(b) The `removeVariety` method updates the master order by removing all of the cookie orders in which the variety of cookie matches the parameter `cookieVar`. The master order may contain zero or more cookie orders with the same variety as `cookieVar`. The method returns the total number of boxes removed from the master order.

For example, consider the following code segment.

```
MasterOrder goodies = new MasterOrder();
goodies.addOrder(new CookieOrder("Chocolate Chip", 1));
goodies.addOrder(new CookieOrder("Shortbread", 5));
goodies.addOrder(new CookieOrder("Macaroon", 2));
goodies.addOrder(new CookieOrder("Chocolate Chip", 3));
```

After the code segment has executed, the contents of the master order are as shown in the following table.

| "Chocolate Chip"<br>1 | "Shortbread"<br>5 | "Macaroon"<br>2 | "Chocolate Chip"<br>3 |
|---|---|---|---|

The method call `goodies.removeVariety("Chocolate Chip")` returns 4 because there were two Chocolate Chip cookie orders totaling 4 boxes. The master order is modified as shown below.

| "Shortbread"<br>5 | "Macaroon"<br>2 |
|---|---|

The method call `goodies.removeVariety("Brownie")` returns 0 and does <u>not</u> change the master order.

Complete method `removeVariety` below.

```
/** Removes all cookie orders from the master order that have the same variety of
 *  cookie as cookieVar and returns the total number of boxes that were removed.
 *  @param cookieVar the variety of cookies to remove from the master order
 *  @return the total number of boxes of cookieVar in the cookie orders removed
 */
public int removeVariety(String cookieVar)
```

2. An `APLine` is a line defined by the equation $ax + by + c = 0$, where $a$ is not equal to zero, $b$ is not equal to zero, and $a$, $b$, and $c$ are all integers. The slope of an `APLine` is defined to be the `double` value $-a/b$. A point (represented by integers $x$ and $y$) is on an `APLine` if the equation of the `APLine` is satisfied when those $x$ and $y$ values are substituted into the equation. That is, a point represented by $x$ and $y$ is on the line if $ax + by + c$ is equal to 0. Examples of two `APLine` equations are shown in the following table.

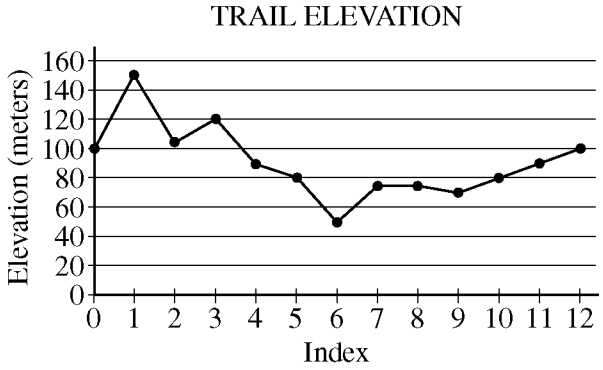
| **Equation** | **Slope (–*a* / *b*)** | **Is point (5, -2) on the line?** |
|---|---|---|
| $5x + 4y - 17 = 0$ | -5 / 4 = -1.25 | Yes, because 5(5) + 4(-2) + (-17) = 0 |
| $-25x + 40y + 30 = 0$ | 25 / 40 = 0.625 | No, because -25(5) + 40(-2) + 30 ≠ 0 |

Assume that the following code segment appears in a class other than `APLine`. The code segment shows an example of using the `APLine` class to represent the two equations shown in the table.

```
APLine line1 = new APLine(5, 4, -17);
double slope1 = line1.getSlope();         // slope1 is assigned -1.25
boolean onLine1 = line1.isOnLine(5, -2); // true because 5(5) + 4(-2) + (-17) = 0


APLine line2 = new APLine(-25, 40, 30);
double slope2 = line2.getSlope();         // slope2 is assigned 0.625
boolean onLine2 = line2.isOnLine(5, -2); // false because -25(5) + 40(-2) + 30 ≠ 0
```

Write the `APLine` class. Your implementation must include a constructor that has three integer parameters that represent $a$, $b$, and $c$, in that order. You may assume that the values of the parameters representing $a$ and $b$ are not zero. It must also include a method `getSlope` that calculates and returns the slope of the line, and a method `isOnLine` that returns `true` if the point represented by its two parameters (`x` and `y`, in that order) is on the `APLine` and returns `false` otherwise. Your class must produce the indicated results when invoked by the code segment given above. You may ignore any issues related to integer overflow.

3. A hiking trail has elevation markers posted at regular intervals along the trail. Elevation information about a trail can be stored in an array, where each element in the array represents the elevation at a marker. The elevation at the first marker will be stored at array index 0, the elevation at the second marker will be stored at array index 1, and so forth. Elevations between markers are ignored in this question. The graph below shows an example of trail elevations.




The table below contains the data represented in the graph.

**Trail Elevation (meters)**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Elevation | 100 | 150 | 105 | 120 | 90 | 80 | 50 | 75 | 75 | 70 | 80 | 90 | 100 |

The declaration of the `Trail` class is shown below. You will write two unrelated methods of the `Trail` class.

```
public class Trail
{
  /** Representation of the trail. The number of markers on the trail is markers.length. */
  private int[] markers;

  /** Determines if a trail segment is level. A trail segment is defined by a starting marker,
   *  an ending marker, and all markers between those two markers.
   *  A trail segment is level if it has a difference between the maximum elevation
   *  and minimum elevation that is less than or equal to 10 meters.
   *  @param start the index of the starting marker
   *  @param end the index of the ending marker
   *         Precondition: 0 <= start < end <= markers.length - 1
   *  @return true if the difference between the maximum and minimum
   *          elevation on this segment of the trail is less than or equal to 10 meters;
   *          false otherwise.
   */
  public boolean isLevelTrailSegment(int start, int end)
  {  /* to be implemented in part (a) */  }

  /** Determines if this trail is rated difficult. A trail is rated by counting the number of changes in
   *  elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
   *  with 3 or more such changes is rated difficult.
   *  @return true if the trail is rated difficult; false otherwise.
   */
  public boolean isDifficult()
  {  /* to be implemented in part (b) */  }

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

(a) Write the `Trail` method `isLevelTrailSegment`. A trail segment is defined by a starting marker, an ending marker, and all markers between those two markers. The parameters of the method are the index of the starting marker and the index of the ending marker. The method will return `true` if the difference between the maximum elevation and the minimum elevation in the trail segment is less than or equal to 10 meters.

For the trail shown at the beginning of the question, the trail segment starting at marker 7 and ending at marker 10 has elevations ranging between 70 and 80 meters. Because the difference between 80 and 70 is equal to 10, the trail segment is considered level.

The trail segment starting at marker 2 and ending at marker 12 has elevations ranging between 50 and 120 meters. Because the difference between 120 and 50 is greater than 10, this trail segment is not considered level.

Complete method `isLevelTrailSegment` below.

```
/** Determines if a trail segment is level. A trail segment is defined by a starting marker,
 *  an ending marker, and all markers between those two markers.
 *  A trail segment is level if it has a difference between the maximum elevation
 *  and minimum elevation that is less than or equal to 10 meters.
 *  @param start the index of the starting marker
 *  @param end the index of the ending marker
 *         Precondition: 0 <= start < end <= markers.length - 1
 *  @return true if the difference between the maximum and minimum
 *          elevation on this segment of the trail is less than or equal to 10 meters;
 *          false otherwise.
 */
public boolean isLevelTrailSegment(int start, int end)
```

(b) Write the `Trail` method `isDifficult`. A trail is rated by counting the number of changes in elevation that are at least 30 meters (up or down) between two consecutive markers. A trail with 3 or more such changes is rated difficult. The following table shows trail elevation data and the elevation changes between consecutive trail markers.

**Trail Elevation (meters)**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Elevation | 100 | 150 | 105 | 120 | 90 | 80 | 50 | 75 | 75 | 70 | 80 | 90 | 100 |
| | | \ / | \ / | \ / | \ / | \ / | \ / | \ / | \ / | \ / | \ / | \ / | \ / |
| Elevation change | | 50 | -45 | 15 | -30 | -10 | -30 | 25 | 0 | -5 | 10 | 10 | 10 |

This trail is rated difficult because it has 4 changes in elevation that are 30 meters or more (between markers 0 and 1, between markers 1 and 2, between markers 3 and 4, and between markers 5 and 6).

Complete method `isDifficult` below.

```
/** Determines if this trail is difficult. A trail is rated by counting the number of changes in
 *  elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
 *  with 3 or more such changes is rated difficult.
 *  @return true if the trail is rated difficult; false otherwise.
 */
public boolean isDifficult()
```