AP® Computer Science A
2010 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.com.
AP Central is the official online home for the AP Program: apcentral.collegeboard.com.

AP® COMPUTER SCIENCE A
2010 GENERAL SCORING GUIDELINES

Apply the question-specific rubric first. To maintain scoring intent, a single error is generally accounted for only once per question thereby mitigating multiple penalties for the same error. The error categorization below is for cases not adequately covered by the question-specific rubric. Note that points can only be deducted if the error occurs in a part that has earned credit via the question-specific rubric. Any particular error is **penalized only once** in a question, even if it occurs on different parts of that question.

Nonpenalized Errors

spelling/case discrepancies if no ambiguity*

local variable not declared if others are declared in some part

use 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 Fish;`

missing `( )` on parameterless method call

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

missing `;` when majority are present

missing `public` on class or constructor header

extraneous `[]` when referencing entire array

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

Minor Errors (1/2 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 `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 Errors** (½ point)*: A minor error that occurs **exactly once** when the same concept is **correct two or more times** is regarded as an oversight and **not penalized**. A minor error **must be penalized** if it 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 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, "*`ArrayList`*" instead of "*`ArrayList`*". As a counter example, 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
2010 SCORING GUIDELINES

Question 1: Master Order

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

Intent: *Compute and return the sum of the number of boxes of all cookie orders in* `this.orders`

+1 Considers all `CookieOrder` objects in `this.orders`
- **+1/2** Accesses any element of `this.orders`
- **+1/2** Accesses all elements of `this.orders` with no out-of-bounds access potential

+1 1/2 Computes total number of boxes
- **+1/2** Creates an accumulator (declare and initialize)
- **+1/2** Invokes `getNumBoxes` on object of type `CookieOrder`
- **+1/2** Correctly accumulates total number of boxes

+1/2 Returns computed total

Part (b) `removeVariety` **6 points**

Intent: *Remove all* `CookieOrder` *objects from* `this.orders` *whose variety matches* `cookieVar`; *return total number of boxes removed*

+4 Identifies and removes matching `CookieOrder` objects
- **+1/2** Accesses an element of `this.orders`
- **+1/2** Compares parameter `cookieVar` with `getVariety()` of a `CookieOrder` object (must use `.equals` or `.compareTo`)
- **+1** Compares parameter `cookieVar` with `getVariety()` of all `CookieOrder` objects in `this.orders`, no out-of-bounds access potential
- **+1/2** Removes an element from `this.orders`
- **+1/2** Removes only matching `CookieOrder` objects
- **+1** Removes all matching `CookieOrder` objects, no elements skipped

+1 1/2 Computes total number of boxes in removed `CookieOrder` objects
- **+1/2** Creates an accumulator (declare and initialize)
- **+1/2** Invokes `getNumBoxes` on object of type `CookieOrder`
- **+1/2** Correctly accumulates total number of boxes (must be in context of loop and match with `cookieVar`)

+1/2 Returns computed total

Usage:

–1 consistently references incorrect name instead of `orders`, of potentially correct type

–1 1/2 consistently references incorrect name instead of `orders`, incorrect type (e.g., `this`, `MasterOrder`)

AP® COMPUTER SCIENCE A
2010 SCORING GUIDELINES

Question 2: APLine

Intent: *Design complete* `APLine` *class including constructor,* `getSlope` *and* `isOnLine` *methods*

+1 Complete, correct header for `APLine` [`class APLine`]
Note: Accept any visibility except private

+1 1/2 State maintenance
- **+1/2** Declares at least one instance variable capable of maintaining numeric value
- **+1/2** Declares at least three instance variables capable of maintaining numeric values
- **+1/2** All state variables have private visibility

Note: Accept any numeric type (primitive or object)
Note: Accept any distinct Java-valid variable names

+1 1/2 `APLine` Constructor
Method header
- **+1/2** Correctly formed header (visibility not private; name `APLine`)
- **+1/2** Specifies exactly three numeric parameters

Method body
- **+1/2** Sets appropriate state variables based on parameters (no shadowing errors)

Note: Interpret instance fields by usage not by name

+2 1/2 `getSlope`
Method header
- **+1/2** Correct method header
(visibility not private; type `double` or `Double`; name `getSlope`; parameterless)

Method body
- **+1/2** Computation uses correct formula for slope
- **+1** Computation uses double precision (no integer division)
- **+1/2** Returns computed value

+2 1/2 `isOnLine`
Method header
- **+1/2** Correct formed header (visibility not private; type `boolean` or `Boolean`, name `isOnLine`)
- **+1/2** Specifies exactly two numeric parameters

Method body
- **+1/2** Computation uses correct formula involving state and parameters (`a*x + b*y + c`)
- **+1/2** Computation uses correct comparison test (equal to zero)
- **+1/2** Returns `true` if is on this `APLine`; `false` otherwise

AP® COMPUTER SCIENCE A
2010 SCORING GUIDELINES

Question 3: Trail

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

Intent: *Return* `true` *if maximum difference ≤ 10 (segment is level);* `false` *otherwise*

+3 Determination of information needed to test level-trail condition
- **+1/2** Creates and maintains local state for determination of maximum (or minimum);
  alternate solution: tests difference in elevations
- **+1/2** Accesses the value of any element of `this.markers`
- **+1** All and only appropriate elements of `this.markers` participate in determination of information needed to test level-trail condition; no out-of-bounds access potential
- **+1** Compares element to state in context of updating maximum (or minimum);
  alternate solution: tests difference in elevations

+1 Correctly determines information needed to test level-trail condition for the elements examined; must address two or more pairs of elements

+1 Returns `true` if determined maximum difference is ≤ 10, `false` otherwise

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

Intent: *Return* `true` *if trail is difficult (based on number of changes of given magnitude);* `false` *otherwise*

+3 Determine number of changes, greater than or equal to 30, between consecutive values in `this.markers`

- **+1/2** Creates, initializes and accumulates a count of number of changes
- **+1/2** Accesses the value of any element of `this.markers` in context of iteration
- **+1/2** Accesses the value of all elements of `this.markers`, no out-of-bounds access potential
- **+1/2** Computes difference of all and only consecutive values in `this.markers`
- **+1** Updates accumulated count if and only if absolute value of difference is >= 30

+1 Returns `true` if accumulated count is >= 3; `false` otherwise

AP® COMPUTER SCIENCE A
2010 SCORING GUIDELINES

Question 4: GridChecker (GridWorld)

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

Intent: *Identify and return actor in* `this.gr` *with most neighbors; return* `null` *if no actors in grid*

+1 Consider all occupied locations or all actors in grid
- **+1/2** Iterates over all occupied locations in `this.gr`
- **+1/2** Performs action using actor or location from `this.gr` within iteration

+1 1/2 Determination of maximum number of neighbors
- **+1/2** Determines number of occupied neighboring locations* of a location
- **+1** Correctly determines maximum number of neighbors

+1 1/2 Return actor
- **+1/2** Returns reference to `Actor` (not `Location`)
- **+1** Returns reference to a correct actor; `null` if no actors in `this.gr`

**Note: This may be done using* `getOccupiedAdjacentLocations`, `getNeighbors`, *or an iterative* `get` *of surrounding locations*

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

Intent: *Return list of all occupied locations within 2 rows/columns of parameter, parameter excluded*

+1/2 Creates and initializes local variable to hold collection of locations

+2 Consider surrounding locations
- **+1/2** Considers at least two locations 1 row and/or 1 column away from parameter
- **+1/2** Considers at least two locations 2 rows and/or 2 columns away from parameter
- **+1** Correctly identifies all and only valid locations within 2 rows and 2 columns of parameter

+1 Collect occupied locations†
- **+1/2** Adds any location object to collection
- **+1/2** Adds location to collection only if occupied

+1 1/2 Return list of locations
- **+1/2** Returns reference to a list of locations
- **+1/2** List contains all and only identified locations†
- **+1/2** Parameter `loc` excluded from returned list

†*Note: Duplication of locations in returned list is not penalized*

Usage: **–**½ parameter dyslexia in new `Location` constructor invocation

AP® COMPUTER SCIENCE A
2010 CANONICAL SOLUTIONS

Question 1: Master Order

Part (a):

```
public int getTotalBoxes() {
  int sum = 0;
  for (CookieOrder co : this.orders) {
    sum += co.getNumBoxes();
  }
  return sum;
}
```

Part (b):

```
public int removeVariety(String cookieVar) {
  int numBoxesRemoved = 0;
  for (int i = this.orders.size() - 1; i >= 0; i--) {
    if (cookieVar.equals(this.orders.get(i).getVariety())) {
      numBoxesRemoved += this.orders.get(i).getNumBoxes();
      this.orders.remove(i);
    }
  }
  return numBoxesRemoved;
}

// Alternative solution (forward traversal direction):

public int removeVariety(String cookieVar) {
  int numBoxesRemoved = 0;
  int i = 0;
  while (i < this.orders.size()) {
    if (cookieVar.equals(this.orders.get(i).getVariety())) {
      numBoxesRemoved += this.orders.get(i).getNumBoxes();
      this.orders.remove(i);
    } else {
      i++;
    }
  }
  return numBoxesRemoved;
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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
2010 CANONICAL SOLUTIONS

Question 2: APLine

```
public class APLine {
  /** State variables. Any numeric type; object or primitive. */
  private int a, b, c;

  /** Constructor with 3 int parameters. */
  public APLine(int a, int b, int c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  /** Determine the slope of this APLine. */
  public double getSlope() {
    return ( - (this.a / (double) this.b));
  }

  /** Determine if coordinates represent a point on this APLine. */
  public boolean isOnLine(int x, int y) {
    return (0 == (this.a * x) + (this.b * y) + this.c);
  }
}


// Alternative solution (state variables of type double):

public class APLine {
  private double a1, b1, c1;

  public APLine(int a, int b, int c) {
    this.a1 = a;
    this.b1 = b;
    this.c1 = c;
  }

  public double getSlope() {
    return -(this.a1 / this.b1);
  }

  public boolean isOnLine(int x, int y) {
    return (0 == (this.a1 * x) + (this.b1 * y) + this.c1);
  }
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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
2010 CANONICAL SOLUTIONS

Question 3: Trail

Part (a):

```
public boolean isLevelTrailSegment(int start, int end) {
  int min = this.markers[start];
  int max = this.markers[start];
  for (int i = start + 1; i <= end; i++) {
    if (min > this.markers[i]) {
      min = this.markers[i];
    }
    if (max < this.markers[i]) {
      max = this.markers[i];
    }
  }
  return ((max - min) <= 10);
}

// Alternative solution (compares differences; uses early return):

public boolean isLevelTrailSegment(int start, int end) {
  for (int i = start; i < end; i++) {
    for (int j = start + 1; j <= end; j++) {
      if (Math.abs(this.markers[i] - this.markers[j]) > 10) {
        return false;
      }
    }
  }
  return true;
}
```

Part (b):

```
public boolean isDifficult() {
  int numChanges = 0;
  for (int i = 0; i < this.markers.length - 1; i++) {
    if (Math.abs(this.markers[i] - this.markers[i + 1]) >= 30) {
      numChanges++;
    }
  }
  return (numChanges >= 3);
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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
2010 CANONICAL SOLUTIONS

Question 4: GridChecker (GridWorld)

Part (a):

```
public Actor actorWithMostNeighbors() {
  if (0 == this.gr.getOccupiedLocations().size()) {
    return null;
  }
  Location where = null;
  int most = -1;
  for (Location loc : this.gr.getOccupiedLocations()) {
    if (most < this.gr.getOccupiedAdjacentLocations(loc).size()) {
      most = this.gr.getOccupiedAdjacentLocations(loc).size();
      where = loc;
    }
  }
  return this.gr.get(where);
}

// Alternative solution (uses getNeighbors):

public Actor actorWithMostNeighbors() {
  if (0 == this.gr.getOccupiedLocations().size()) {
    return null;
  }
  Location where = this.gr.getOccupiedLocations().get(0);
  for (Location loc : this.gr.getOccupiedLocations()) {
    if (this.gr.getNeighbors(where).size() <
this.gr.getNeighbors(loc).size()) {
      where = loc;
    }
  }
  return this.gr.get(where);
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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
2010 CANONICAL SOLUTIONS

Question 4: GridChecker (GridWorld) (continued)

Part (b):

```
public List<Location> getOccupiedWithinTwo(Location loc) {
  List<Location> occupied = new ArrayList<Location>();
  for (int row = loc.getRow() - 2; row <= loc.getRow() + 2; row++) {
    for (int col = loc.getCol() - 2; col <= loc.getCol() + 2; col++) {
      Location loc1 = new Location(row, col);
      if (gr.isValid(loc1) && this.gr.get(loc1) != null &&
!loc1.equals(loc)) {
        occupied.add(loc1);
      }
    }
  }
  return occupied;
}

// Alternative solution (uses getOccupiedLocations):

public List<Location> getOccupiedWithinTwo(Location loc) {
  List<Location> occupied = new ArrayList<Location>();
  for (Location loc1 : this.gr.getOccupiedLocations()) {
    if ((Math.abs(loc.getRow() - loc1.getRow()) <= 2)
         && (Math.abs(loc.getCol() - loc1.getCol()) <= 2)
         && !loc1.equals(loc)) {
      occupied.add(loc1);
    }
  }
  return occupied;
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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.