AP® Computer Science A
2006 Scoring Guidelines

The College Board: Connecting Students to College Success

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 association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). 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
2006 SCORING GUIDELINES

Question 1: Daily Schedule

Part A: `conflictsWith` **1 1/2 points**

+1/2 call `OBJ1.overlapsWith(OBJ2)`
+1/2 access `getTime` of `other` and `this`
+1/2 return correct value

Part B: `clearConflicts` **3 points**

+2 loop over `apptList`
- **+1/2** reference `apptList` in loop body
- **+1/2** access appointment *in context of loop* (`apptList.get(i)`)
- **+1** access all appointments (cannot skip entries after a removal)

+1 remove conflicts *in context of loop*
- **+1/2** determine when conflict exists (must call `conflictsWith`)
- **+1/2** remove all conflicting appointments (and no others)

Part C: `addAppt` **4 1/2 points**

+1/2 test if emergency (*may limit to when emergency AND conflict exists*)
+1/2 clear conflicts if and only if emergency
(must not reimplement `clearConflicts` code)
+1/2 add `appt` if emergency

+2 non-emergency case
- **+1/2** loop over `apptList` (must reference `apptList` in body)
- **+1/2** access `apptList` element and check for `appt` conflicts *in context of loop*
- **+1/2** exit loop with state (conflict / no conflict) correctly determined *(includes loop bound)*
- **+1/2** add `appt` if and only if no conflict

+1 return true if any appointment added, false otherwise (must return both)

Usage: -1 if loop structure results in failure to handle empty `apptList`

AP® COMPUTER SCIENCE A
2006 SCORING GUIDELINES

Question 2: Taxable Items (Design)

Part A: `purchasePrice` **2 1/2 points**

+1 call `getListPrice()`

+1 calculate correct purchase price (*no penalty if truncate/round to 2 decimal places*)

+1/2 return calculated price

Part B: `Vehicle` **6 1/2 points**

+1/2 `class Vehicle extends TaxableItem`
+1/2 `private double dealerCost`
+1/2 `private double dealerMarkup` (*no penalty if also store tax in field*)

+2 1/2 constructor
- **+1/2** `Vehicle(double ?, double ?, double ?)`
  `int/float` (*OK if match fields*)
- **+1** call parent constructor
  - **+1/2** attempt using `super`
  - **+1/2** correct call: `super(rate)` (*note: must be first line in method*)
- **+1** initialize dealer cost and markup fields
  - **+1/2** attempt (must use parameters on RHS or in mutator call)
  - **+1/2** correct

+1 `changeMarkup`
- **+1/2** `public void changeMarkup(double ?)`
  `int/float` (*OK if matches field; no penalty if returns reasonable value*)
- **+1/2** assign parameter to markup field

+1 1/2 `getListPrice`
- **+1** `public double getListPrice()`
- **+1/2** return sum of dealer cost and markup fields

Note: -1 usage if reimplement `purchasePrice` to do anything other than
`return super.purchasePrice();`

AP® COMPUTER SCIENCE A
2006 SCORING GUIDELINES

Question 3: Customer List

Part A: `compareCustomer` **3 points**

+1 1/2 perform comparison
- **+1/2** attempt (must call `OBJ1.compareTo(OBJ2)`)
- **+1/2** correctly access and compare names
- **+1/2** correctly access and compare IDs

+1/2 return 0 if and only if `this = other`
+1/2 return positive if and only if `this > other`
+1/2 return negative if and only if `this < other`

Part B: `prefixMerge` **6 points**

+1/2 initialize unique variables to index fronts of arrays

+1 1/2 loop over arrays to fill result
- **+1/2** attempt (must reference `list1` and `list2` inside loop)
- **+1** correct (lose this if add too few or too many `Customer` elements)

+1 1/2 compare array fronts (in context of loop)
- **+1/2** attempt (must call `compareCustomer` on array elements)
- **+1** correctly compare front `Customer` elements

+1 1/2 duplicate entries
- **+1/2** check if duplicate entries found
- **+1/2** if duplicates, copy only one to `result` (without use of additional structure)
- **+1/2** update indices into both arrays (`list1` and `list2`)

+1 nonduplicate entries
- **+1/2** copy only smallest entry to `result` (without use of additional structure)
- **+1/2** update index into that array only (`list1` or `list2`)

Note: Solution may use constants as returned from part A.

Usage: -1/2 `compareTo` instead of `compareCustomer` for `Customer` objects

Question 4: Drop Game (MBS)

Part A: `dropLocationForColumn` **3 1/2 points**

+1 1/2 loop over `Locations` in column
- **+1/2** correct loop (traverse entire column or until empty location found)
- **+1** construct `Location` object *in context of loop*
  - **+1/2** attempt using column
  - **+1/2** correct

+1 1/2 find drop `Location`
- **+1/2** check if constructed `Location` is empty
- **+1** if exists, return empty `Location` with largest row # (*no loop, no point*)

+1/2 return `null` if column is full

Part B: `dropMatchesNeighbors` **5 1/2 points**

+1 get drop `Location`
- **+1/2** attempt (must call `dropLocationForColumn`)
- **+1/2** correct (must use result)

+1/2 return `false` if drop location is `null`

+1 1/2 get neighboring pieces
- **+1/2** attempt to access adj. neighbors (`getNeighbor` or `neighborsOf` or row/column access)
- **+1/2** correctly access 3 E/W/S neighbor `Location` objects
- **+1/2** correctly access 3 neighbor `Piece` objects

+2 1/2 determine matches
- **+1/2** correct `null` neighbor test
- **+1** compare colors of pieces
  - **+1/2** attempt (must reference `pieceColor`)
  - **+1/2** correct
- **+1** return correct Boolean value1

Usage: -1 `environment` or missing `theEnv`

AP® COMPUTER SCIENCE A/AB
2006 GENERAL USAGE

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

default constructor called without parens; for example, `new Fish;`

use keyword as identifier

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= 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, for example a check for precondition

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

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing "new" for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

Minor Errors (1/2 point)

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

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example
`int x; …x.equals(val)`

`[]` – `get` confusion if access not tested in rubric

assignment dyslexia, for example, `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`

Major Errors (1 point)

extraneous code which causes side-effect, for example, information written to output

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

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

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g., by using root reference to a `TreeNode` for traversal of the tree)

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

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 1: Daily Schedule

PART A:

```
public boolean conflictsWith(Appointment other)
{
  return getTime().overlapsWith(other.getTime());
}
```

PART B:

```
public void clearConflicts(Appointment appt)
{
  int i = 0;
  while (i < apptList.size())
  {
    if (appt.conflictsWith((Appointment)(apptList.get(i))))
    {
      apptList.remove(i);
    }
    else
    {
      i++;
    }
  }
}
```

ALTERNATE SOLUTION

```
public void clearConflicts(Appointment appt)
{
  for (int i = apptList.size()-1; i >= 0; i--)
  {
    if (appt.conflictsWith((Appointment)apptList.get(i)))
    {
      apptList.remove(i);
    }
  }
}
```

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 1: Daily Schedule (continued)

PART C:

```
public boolean addAppt(Appointment appt, boolean emergency)
{
  if (emergency)
  {
    clearConflicts(appt);
  }
  else
  {
    for (int i = 0; i < apptList.size(); i++)
    {
      if (appt.conflictsWith((Appointment)apptList.get(i)))
      {

        return false;
      }
    }
  }
  return apptList.add(appt);
```

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 2: Taxable Items (Design)

PART A:

```
public double purchasePrice()
{
  return (1 + taxRate) * getListPrice();
}
```

PART B:

```
public class Vehicle extends TaxableItem
{
  private double dealerCost;
  private double dealerMarkup;

  public Vehicle(double cost, double markup, double rate)
  {
    super(rate);
    dealerCost = cost;
    dealerMarkup = markup;
  }

  public void changeMarkup(double newMarkup)
  {
    dealerMarkup = newMarkup;
  }

  public double getListPrice()
  {
    return dealerCost + dealerMarkup;
  }

}
```

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 3: Customer List

PART A:

```
public int compareCustomer(Customer other)
{
  int nameCompare = getName().compareTo(other.getName());
  if (nameCompare != 0)
  {
    return nameCompare;
  }
  else
  {
    return getID() - other.getID();
  }
}
```

PART B:

```
public static void prefixMerge(Customer[] list1, Customer[] list2, Customer[] result)
{
  int front1 = 0;
  int front2 = 0;

  for (int i = 0; i < result.length; i++)
  {
    int comparison = list1[front1].compareCustomer(list2[front2]);
    if (comparison < 0)
    {
      result[i] = list1[front1];
      front1++;
    }
    else if (comparison > 0)
    {
      result[i] = list2[front2];
      front2++;
    }
    else
    {
      result[i] = list1[front1];
      front1++;
      front2++;
    }
  }
}
```

AP® COMPUTER SCIENCE A
2006 CANONICAL SOLUTIONS

Question 4: Drop Game (MBS)

PART A:

```
public Location dropLocationForColumn(int column)
{
  for (int r = theEnv.numRows()-1; r >= 0; r--)
  {
    Location nextLoc = new Location(r, column);
    if (theEnv.isEmpty(nextLoc))
    {
      return nextLoc;
    }
  }
  return null;
}
```

ALTERNATE SOLUTION

```
public Location dropLocationForColumn(int column)
{
  int maxRow = -1;
  for (int r = 0; r < theEnv.numRows(); r++)
  {
    if (theEnv.isEmpty(new Location(r, column)))
    {
      maxRow = r;
    }
  }

  if (maxRow < 0)
  {
    return null;
  }
  return new Location(maxRow, column);
}
```

Question 4: Drop Game (MBS) (continued)

PART B:

```
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
  Location loc = dropLocationForColumn(column);
  if (loc == null)
  {
    return false;
  }
  Piece n1 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.WEST)));
  Piece n2 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.EAST)));
  Piece n3 = (Piece)(theEnv.objectAt(theEnv.getNeighbor(loc, Direction.SOUTH)));
  return (n1 != null && n1.color().equals(pieceColor) &&
          n2 != null && n2.color().equals(pieceColor) &&
          n3 != null && n3.color().equals(pieceColor));
}
```

ALTERNATE SOLUTION

```
public boolean dropMatchesNeighbors(int column, Color pieceColor)
{
  Location loc = dropLocationForColumn(column);
  if (loc == null)
  {
    return false;
  }
  ArrayList neighbors = theEnv.neighborsOf(loc);
  int colorCount = 0;
  for (int i = 0; i < neighbors.size(); i++)
  {
    Piece nextNbr = (Piece)(theEnv.objectAt((Location)neighbors.get(i)));
    if (nextNbr != null && nextNbr.color().equals(pieceColor))
    {
      colorCount++;
    }
  }

  return (colorCount == 3);
}
```