



AP[®] Computer Science AB 2002 Scoring Guidelines

The materials included in these files are intended for use by AP teachers for course and exam preparation in the classroom; permission for any other use must be sought from the Advanced Placement Program[®]. Teachers may reproduce them, in whole or in part, in limited quantities, for face-to-face teaching purposes but may not mass distribute the materials, electronically or otherwise. These materials and any copies made of them may not be resold, and the copyright notices must be retained as they appear here. This permission does not apply to any third-party copyrights contained herein.

These materials were produced by Educational Testing Service[®] (ETS[®]), which develops and administers the examinations of the Advanced Placement Program for the College Board. The College Board and Educational Testing Service (ETS) are dedicated to the principle of equal opportunity, and their programs, services, and employment policies are guided by that principle.

The College Board is a national nonprofit membership association dedicated to preparing, inspiring, and connecting students to college and opportunity. Founded in 1900, the association is composed of more than 4,200 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 22,000 high schools, and 3,500 colleges, through major programs and services in college admission, 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 equity and excellence, and that commitment is embodied in all of its programs, services, activities, and concerns.

Copyright © 2002 by College Entrance Examination Board. All rights reserved. College Board, Advanced Placement Program, AP, SAT, and the acorn logo are registered trademarks of the College Entrance Examination Board. APIEL is a trademark owned by the College Entrance Examination Board. PSAT/NMSQT is a registered trademark jointly owned by the College Entrance Examination Board and the National Merit Scholarship Corporation. Educational Testing Service and ETS are registered trademarks of Educational Testing Service.

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 1

Part A: EmptySeatCount	3 points
-------------------------------	-----------------

- +1 Loop over matrix
 - +1/2 attempt (traverse and index some matrix in two dimensions: *anything[var1][var2]* shows access to multiple rows with multiple columns in each row)
 - +1/2 correct

- +1 1/2 Identify empty seats of correct type
 - +1/2 attempt (*EmptyTestAttempt** OR compares seatType against something)
 - +1/2 correct use of abstraction
 - +1/2 correctly identify all empty seats

- +1/2 Count – initialize counter, conditionally increment counter, return value

Note: If specific columns are used for seat types, cannot get any Identify points.

Part B: FindBlock	3 points
--------------------------	-----------------

Note: No deduction for missing check of parameters; bad check can lose Traverse correct ½ point.

- +1 Traverse row and test for empty seat
 - +1/2 attempt (traverse a row, compare something against empty string)
 - +1/2 correct (no out-of-bounds, correct use of abstraction)

- +1 1/2 Find a block of empty seats
 - +1/2 attempt (nested traversal with *EmptyTestAttempt** OR attempt to count adjacent empty seats)
 - +1/2 traverse potential blocks (checks range of block)
 - +1/2 correctly identify & maintain block location (block with enough empty seats found if it exists)

- +1/2 Return correct value (leftmost location of empty block or -1)

**EmptyTestAttempt* = compare mySeat(s) or GetPassenger(s) or GetName(s) against empty string
OR compare GetPassenger(s) with Passenger()

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 1 (cont'd.)

Part C: AssignGroup

3 points

Reminder: assume FindBlock returns -1 on parameters outside bounds

+1/2 Loop through rows correctly and terminate

+1 Find block of empty seats

+1/2 attempt (call FindBlock, use returned value, parameters optional
OR reimplement correctly)

+1/2 correct (correct call to FindBlock OR reimplemented correctly)

+1 Assign passengers to seats if found

+1/2 attempt (must attempt to assign a passenger from group in context of search for block,
index not required)

+1/2 correct

+1/2 Return correct boolean

Notes: If group is placed more than once, must lose Loop and Assign correct points.

No loop loses Loop, Assign correct, Return correct.

Constant loop loses Loop.

**EmptyTestAttempt* = compare mySeat(s) or GetPassenger(s) or GetName(s) against empty string
OR compare GetPassenger(s) with Passenger()

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 2

Part A: AppendQueue	3 points
----------------------------	-----------------

- +1 loop while source is not empty
(lose this point if read into another structure that has size or order problems)
- +1 dequeue card from source queue
- +1 enqueue card into destination (lose this point for destination.MakeEmpty())

Note: -1 usage for appending destination to source

Part B: OneRound	6 points
-------------------------	-----------------

- +1 loop until one player wins round or both tie
 - +1/2 attempt
 - +1/2 correct exit when both empty or when card values are unequal

Processing of Players' Cards: (#Note: Check for One Empty Pile must not interrupt this sequence.)

- | |
|---|
| <ul style="list-style-type: none">+1/2 dequeue both cards in all cases+1 compare card values<ul style="list-style-type: none">+1/2 attempt (must compare card-like items – cards, values)+1/2 correct (must cover < > =)+1/2 enqueue both cards into discard pile when values equal+1 call AppendQueue for player with higher card value<ul style="list-style-type: none">+1/2 attempt (must attempt to move all cards from discard pile to player's pile using AppendQueue or correct reimplementatation)+1/2 correct (must move both players' cards correctly) |
|---|

Check for One Empty Pile:

- | |
|--|
| <ul style="list-style-type: none">+1 check for one player with empty pile<ul style="list-style-type: none">+1/2 attempt (must attempt to check one pile empty, other not)+1/2 correct (must exit loop to get correct; see also #Note above.)+1 call AppendQueue for player w/non-empty pile<ul style="list-style-type: none">+1/2 attempt (must attempt to move all cards from discard pile to player's pile using AppendQueue or correct reimplementatation)+1/2 correct |
|--|

Additional scoring information on next page.

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 2 (cont'd.)

Usage:

- 1/2 dequeue returns value
- 1/2 missing or incorrect declaration of discard pile (part B)
- 1/2 type included with parameter in function call, e.g., source.dequeue(Card & card1)
- 1 Card/int declaration error, then treat as Card

Note: Recursive solution would need helper to carry discard pile (or return indication of winner). Recursive solution without this loses loop-correct ½ point and both call-AppendQueue-correct ½ points.

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 3

Part A: AllFish

3 points

- +1 loop through rows
 - +1/2 attempt (must access elements of myWorld)
 - +1/2 correct (myWorld.length() or NumRows() OK)

- +1 loop through linked list
 - +1/2 attempt
 - must use auxiliary ptr, lose full pt if myWorld[row] is changed
 - must have 2 of 3: init, increment, test
 - +1/2 correct

- +1 add **fish** to vector
 - +1/2 attempt
 - +1/2 correct (count must be incremented, any other counter must be initialized as well)
(fishList[count++] OK)

Part B: AddFish

6 points

- +1/2 increment myFishCreated (must come before call to Fish constructor)
- +1/2 increment myFishCount

- +1/2 assign auxiliary pointer to myWorld[pos.Row()]

- +1 create new ListNode correctly
 - +1/2 attempt (attempt to create new ListNode with a fish, an int, and pointer)
 - +1/2 correct (new ListNode with the Fish, column index and some reasonable pointer)

- +1 1/2 check for and insert at front – empty list or column smaller than first column in list
 - +1/2 attempt at least one case
(must be conditional and have a reference to myWorld[pos.Row()])
 - +1/2 correctly identify both empty list and front insertion
 - +1/2 assign new ListNode to front of list
(myWorld[pos.Row()] must be reset and next field properly set for cases checked)

- +2 check for and insert at middle or at end
 - +1/2 attempt at least one case
 - +1/2 correct location found in both cases
(must have comparison and use ptr->next or use trailer)
 - +1/2 assign new ListNode at correct point in middle and assign next correctly
 - +1/2 assign new ListNode at end and assign next correctly

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 4

Part A: BitsToWord

4 points

- +1 check all bits
 - +1/2 attempt (must check different bits of code)
 - +1/2 correct

- +1 pointer movement: initialize, left, right
 - +1/2 attempt (initialize and try to move left or right based on bit OR if no init, move correctly)
 - +1/2 correct (lose this if bit is used when at leaf)
(lose this if myRoot is used)

- +1 1/2 handle letter
 - +1/2 determine when at letter (one child NULL or check for letter is enough)
(can lose this if it misses the last letter because letter checked before move)
 - +1/2 letter added to result
 - +1/2 reset pointer to root

- +1/2 return built word (including declaration/initialization)

Note: A recursion after each letter is found can work well. If trouble with indexing, lose check all bits correct 1/2 point. If trouble with recursion, lose either reset pointer to root or pointer movement, as appropriate. Recursion on every bit needs helper function to be correct.

**AP[®] COMPUTER SCIENCE AB
2002 SCORING GUIDELINES**

Question 4 (cont'd.)

Part B: CharToBitsHelper

5 points

Base Case (3 points)

+1 1/2 T == NULL or check for leaf
+1/2 attempt
+1 correct return value

+1 1/2 Find letter
+1/2 attempt
+1 correct return value

Recursive Case (2 points)

+1 recursive calls, left and right
+1/2 attempt (must attempt two calls containing at least T->left and T->right)
+1/2 correct parameters for both cases

+1 return correct string after recursive calls
+1/2 attempt (must return a string based on recursive calls)
+1/2 correct
(note: if recursive calls are made as if void functions, lose entire return point)

Note: While loop(s) with recursive calls score as follows:

Can only get points for "Find letter" (1 ½) and "Recursive calls left and right" (1)

Usage: 0, '0', and "0" are considered equivalent – no point deducted

AP[®] COMPUTER SCIENCE AB 2002 SCORING GUIDELINES

Grading Guidelines for AP Computer Science Free-Response Questions

The grading for each question is based on a rubric that has been developed by the question and exam leaders. The rubric allocates points to different elements of a solution.

A common pattern for a rubric is to indicate a point (or half point) for an attempt at an element of a solution, with another point if that element is correct. In general, the attempt point is given if there is clear evidence that the student understands that element of the problem. For example, a loop that is clearly an attempt to iterate over the correct range would get the attempt point but might lose the correct point if there was an off-by-one error or an incorrect increment. Similarly, an assignment or conditional that involved an expression including an array access would get an attempt point if the expression was substantially correct, but had an error with the array index or a minor error in the expression. The rubric describes what constitutes enough to warrant giving an attempt point. A correct point means correct, with the exceptions noted below.

Some elements of a problem may simply be all-or-nothing for a particular point.

The “General Usage” sheet specifies how errors not incorporated into the rubric should be handled. Some items on the “General Usage” sheet are also addressed in a rubric; in such a case, the rubric takes precedence. (Check with question leader for application of General Usage.)

Sometimes a rubric does not cover an error appropriately. For example, there might be $\frac{1}{2}$ point allocated for a function to have a correct return statement. However, a missing return combined with output to the screen of the return value indicates a fundamental misconception about functions that warrants a larger penalty. On the other hand, a solution might be correct except for a minor error or an error that is repeated several times. Some minor errors should not be deducted at all and some should not be deducted repeatedly if the code is otherwise correct. The “General Usage” sheet covers these situations.

General Usage specifies certain minor errors that should not be deducted, such as missing semicolons or the use of "[r, c]" instead of "[r][c]" for accessing an apmatrix. An element of a solution with such an error can get credit for being correct.

General Usage also indicates minor errors worth $\frac{1}{2}$ point and major errors worth 1 point. These errors should be taken only once on a particular problem, and not repeatedly. If a usage deduction is made, then an element of the problem can get credit for being correct on that part if it has no other errors.

Usage points cannot be deducted for a part of a problem that would thereby receive a negative score.

AP[®] COMPUTER SCIENCE AB 2002 SCORING GUIDELINES

2002 General Usage

Some usage errors may be 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 on a part 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. If it occurs on different parts of a problem, it should be deducted only once.

Non-penalized errors

case discrepancies, unless
confuses identifiers

missing ;'s

missing { }'s where indentation
clearly conveys intent

default constructor called with
parens, e.g., `BigInt b()`

`obj.Func` instead of `obj.Func()`

loop variables used outside loop

`[r, c]` instead of `[r][c]`

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

missing ()'s around if/while tests

`<<` instead of `>>` (and vice-versa)

`*foo.data` instead of `(*foo).data`

Minor errors (1/2 point)

misspelled/confused identifier
(e.g., `link/next`)

no variables declared

void function returns a value

modifying a `const` parameter

unnecessary `cout << "done"`

unnecessary `cin` (to pause)

no `*` in pointer declaration

use of `L->item` when `L.item` is
correct

Major errors (1 point)

reads new values for parameters
(write prompts are part of this point)

function result written to output

uses type or class name instead
of variable identifier, for example
`Fish.move()` instead of `f.move()`

`MemberFunction(obj)` instead
of `obj.MemberFunction()`

`param.FreeFunction()` instead
of `FreeFunction(param)`

Use of object reference that is
incorrect or not needed, for
example, use of `f.move()` inside
member function of `Fish` class

Use of private data when it is not
accessible, instead of the
appropriate accessor function

Note: Case discrepancies for identifiers fall under the “not penalized” category. However, if they result in another error, they must be penalized. Sometimes students bring this on themselves with their definition of variables. For example, if a student declares `"Fish fish;"`, then uses `Fish.move()` instead of `fish.move()`, the one point usage deduction applies.