



## AP Computer Science AB 1999 Free-Response Questions

**The materials included in these files are intended for non-commercial use by AP teachers for course and exam preparation; 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 3,900 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<sup>®</sup>, the PSAT/NMSQT<sup>™</sup>, the Advanced Placement Program<sup>®</sup> (AP<sup>®</sup>), and Pacesetter<sup>®</sup>. 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 © 2001 by College Entrance Examination Board. All rights reserved. College Board, Advanced Placement Program, AP, and the acorn logo are registered trademarks of the College Entrance Examination Board.

**COMPUTER SCIENCE AB**  
**PROGRAMMING METHODOLOGY, PROGRAMMING IN C++**  
**Suggested time — 45 minutes**  
**20 Questions**

**Directions:** For each of the following questions, decide which is the best of the choices given. Then fill in the corresponding oval on the answer sheet.

1. A patchwork quilt can be made by sewing together many blocks, all of the same size. Each individual block is made up of a number of small squares cut from fabric. A block can be represented as a two-dimensional array of nonblank characters, each of which stands for one small square of fabric. The entire quilt can also be represented as a two-dimensional array of completed blocks. The example below shows an array that represents a quilt made of 9 blocks (in 3 rows and 3 columns). Each block contains 20 small squares (of 4 rows by 5 columns). The quilt uses 2 different fabric squares, represented by the characters 'x' and '.'. We consider only quilts where the main block alternates with the same block flipped upside down (i.e., reflected about a horizontal line through the block's center), as in the example below.

x...x	..x..	x...x
.x.x.	..x..	.x.x.
..x..	.x.x.	..x..
..x..	x...x	..x..
..x..	x...x	..x..
..x..	.x.x.	..x..
.x.x.	..x..	.x.x.
x...x	..x..	x...x
x...x	..x..	x...x
.x.x.	..x..	.x.x.
..x..	.x.x.	..x..
..x..	x...x	..x..

Consider the problem of storing and displaying information about a quilt.

The class `Quilt`, whose declaration is shown below, is used to keep track of the blocks for an entire quilt. Since the pattern is based on one block, we only store that block and the number of rows and columns of blocks. For the example shown above, we would store the upper left  $4 \times 5$  block, 3 for the number of rows of blocks in the quilt and 3 for the number of columns of blocks in the quilt.

```
class Quilt
{
public:
    Quilt(istream & inFile, int rowsOfBlocks, int colsOfBlocks);
    // constructor, given number of blocks in each row and column

    apmatrix<char> QuiltToMat();
    // returns a matrix with the entire quilt stored in it

private:
    apmatrix<char> myBlock; // stores pattern for one block
    int myRowsOfBlocks;    // number of rows of blocks in the quilt
    int myColsOfBlocks;    // number of columns of blocks in the quilt

    void PlaceBlock(int startRow, int startCol,
                    apmatrix<char> & qmat);
    void PlaceFlipped(int startRow, int startCol,
                      apmatrix<char> & qmat);
};
```

**GO ON TO THE NEXT PAGE**

- (a) Write the code for the constructor that initializes a quilt, as started below. The constructor reads the block pattern for the main block from a file represented by the parameter `inFile`. You may assume the file is open and that the file contains the number of rows followed by the number of columns for the block, followed by the characters representing the pattern. For example, the file `pattern`, which contains the pattern for the first block in the quilt shown above, would look like this:

```
4 5
x...x
.x.x.
..x..
..x..
```

The constructor also sets the number of rows and columns of blocks which make up the entire quilt in the initializer list.

Complete the constructor below. Assume that the constructor is called only with parameters that satisfy its precondition.

```
Quilt::Quilt(istream & inFile, int rowsOfBlocks, int colsOfBlocks)
    : myBlock(0, 0), myRowsOfBlocks(rowsOfBlocks),
      myColsOfBlocks(colsOfBlocks)
// precondition:  inFile is open, rowsOfBlocks > 0, colsOfBlocks > 0
// postcondition: myRowsOfBlocks and myColsOfBlocks are initialized to
//                the number of rows and columns of blocks that make up
//                the quilt; myBlock has been resized and
//                initialized to the block pattern from the
//                stream inFile.
```

- (b) Write the private member function `PlaceFlipped`, as started below. `PlaceFlipped` is intended to place a flipped (upside-down) version of the block into the matrix `qmat` with the flipped block's upper left corner located at the `startRow`, `startCol` position in `qmat`.

For example, if quilt `Q` contains the block shown in part (a) and if `M` is a matrix large enough to hold the characters in the whole quilt, then the call

```
Q.PlaceFlipped(4, 10, M)
```

would place the flipped version of `Q`'s quilt block into matrix `M` as the third block in the second row of quilt blocks. This is the block whose upper-left corner is at position `M[4][10]`. In the diagram below, the upper-left corner of the flipped block being placed into `M` is circled.

x...x	..x..	x...x
.x.x.	..x..	.x.x.
..x..	.x.x.	..x..
..x..	x...x	..x..
..x..	x...x	⊙.x..
..x..	.x.x.	..x..
.x.x.	..x..	.x.x.
x...x	..x..	x...x
x...x	..x..	x...x
.x.x.	..x..	.x.x.
..x..	.x.x.	..x..
..x..	x...x	..x..

You may adapt the code of the private member function `PlaceBlock`, given below, which places the block (not inverted) into the matrix `qmat` with the block's upper left corner located at the `startRow`, `startCol` position.

```
void Quilt::PlaceBlock(int startRow, int startCol,
                       apmatrix<char> & qmat)
// precondition: startRow ≥ 0; startCol ≥ 0;
//               startRow + myBlock.numrows() ≤ qmat.numrows();
//               startCol + myBlock.numcols() ≤ qmat.numcols();
// postcondition: myBlock has been copied into the matrix
//               qmat with its upper-left corner at the position
//               startRow, startCol
{
    int r, c;
    for (r = 0; r < myBlock.numrows(); r++)
    {
        for (c = 0; c < myBlock.numcols(); c++)
        {
            qmat[startRow + r][startCol + c] = myBlock[r][c];
        }
    }
}
```

Complete the member function `PlaceFlipped` below. Assume that `PlaceFlipped` is called only with parameters that satisfy its precondition.

```
void Quilt::PlaceFlipped(int startRow, int startCol,
                        apmatrix<char> & qmat)
// precondition:  startRow ≥ 0; startCol ≥ 0;
//               startRow + myBlock.numrows() ≤ qmat.numrows();
//               startCol + myBlock.numcols() ≤ qmat.numcols();
// postcondition: a flipped version of myBlock has been copied into the
//               matrix qmat with its upper-left corner at the position
//               startRow, startCol
{
    int r, c;

    for (r = 0; r < myBlock.numrows(); r++)
    {
        for (c = 0; c < myBlock.numcols(); c++)
        {

        }
    }
}
```

**GO ON TO THE NEXT PAGE**

- (c) Write the member function `QuiltToMat`, as started below. `QuiltToMat` returns a matrix representing the whole quilt in such a way that the main block alternates with the flipped version of the main block, as shown in the original example. If `Q` represents the example quilt, then the call `Q.QuiltToMat()` would return a matrix of characters with the given block placed starting with the upper-left corner at position 0, 0; the flipped block placed with its upper-left corner at position 0, 5; the given block placed with its upper-left corner at position 0, 10; the flipped block placed with its upper-left corner at position 4, 0, and so on.

In writing `QuiltToMat`, you may call functions `PlaceBlock` and `PlaceFlipped` specified in part (b). Assume that `PlaceBlock` and `PlaceFlipped` work as specified, regardless of what you wrote in part (b).

Complete the member function `QuiltToMat` below.

```
apmatrix<char> Quilt::QuiltToMat()
```

2. This question involves reasoning about the code from the Large Integer case study. A copy of the code is provided as part of this examination.

- (a) Write a new `BigInt` member function `Div2`, as started below. `Div2` should change the value of the `BigInt` to be the original value divided by 2 (integer division). Assume the `BigInt` is greater than or equal to 0. One algorithm for implementing `Div2` is:

1. Initialize a variable `carryDown` to 0.
2. For each digit, `d`, starting with the most significant digit,
  - 2.1 replace that digit with  $(d / 2) + \text{carryDown}$
  - 2.2 let `carryDown` be  $(d \% 2) * 5$
3. Normalize the result.

Complete member function `Div2` below.

```
void BigInt::Div2()  
// precondition: BigInt ≥ 0
```

- (b) Write a definition to overload the `/` operator, as started below. Assume that `dividend` and `divisor` are both positive values of type `BigInt`.

For example, assume that `bigNum1` and `bigNum2` are positive values of type `BigInt`:

<u>bigNum1</u>	<u>bigNum2</u>	<u>bigNum1 / bigNum2</u>
18	9	2
17	2	8
8714	2178	4
9990	999	10

There are many ways to implement division; however, you must use a binary search algorithm to find the quotient of `dividend` divided by `divisor` in this problem. You will receive no credit on this part if you do not use a binary search algorithm.

One possible algorithm for implementing division using binary search is as follows:

Let `low` and `high` represent a range in which the quotient is found.

Initialize `low` to 0 and `high` to `dividend`.

For each iteration, compute `mid = (low + high + 1)`, divide `mid` by 2, and compare

`mid * divisor` with `dividend` to maintain the invariant that `low ≤ quotient` and

`high ≥ quotient`.

When `low == high`, the quotient has been found.

In writing function `operator/` you may call function `Div2` specified in part (a). Assume that `Div2` works as specified, regardless of what you wrote in part (a). You will receive NO credit on this part if you do not use a binary search algorithm.

Complete `operator/` below. Assume that `operator/` is called only with parameters that satisfy its precondition.

```
BigInt operator/ (const BigInt & dividend, const BigInt & divisor)
// precondition: dividend > 0, divisor > 0
```

3. Consider designing a data structure to represent a high school club of students with a common interest. The information to be stored in the data structure is as follows:

1. The name of the club.
2. A linked list of club members. For each member, the student's name and grade level in high school is stored.

One way to do this is to use the declarations given below. The first is for a node in the linked list of club members and the second is for the club itself.

```

struct Member
{
    apstring name;      // name of club member
    int level;         // grade 9, 10, 11, or 12
    Member * next;     // next member on the list

    Member(const apstring & nm, int lv, Member * nx);
    // constructor
};

Member::Member(const apstring & nm, int lv, Member * nx)
: name(nm), level(lv), next(nx)
{}

struct Club
{
    apstring clubName; // name of club
    Member * memberList; // list of members in the club

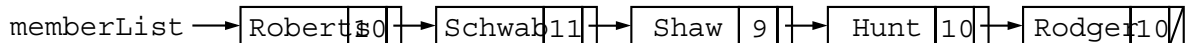
    Club(); // default constructor
    Club(const apstring & clubnm); // constructor
};

```

For example, shown below are two variables of type Club; the first represents the German club and the second represents the Computer club.

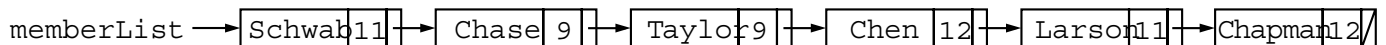
Club1

clubName: German



Club2

clubName: Computer

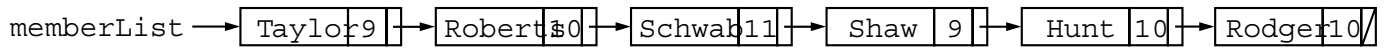


- (a) Write function `InsertMember`, as started below. `InsertMember` adds a student with the given name and respective level in high school to the given club.

For example, after the call `InsertMember("Taylor", 9, Club1)`, variable `Club1` might be as shown below. The diagram shows that the new member "Taylor" was inserted at the beginning of the list of members, but the student could have been inserted anywhere in the list.

Club1

clubName: German



Complete function `InsertMember` below. Assume that `InsertMember` is called only with parameters that satisfy its precondition.

```
void InsertMember(const apstring & name, int level, Club & anyClub)
// precondition: anyClub contains zero or more members, name does not
//               appear in anyClub, and level is 9, 10, 11, or 12
// postcondition: a new member with the given name and respective level
//               has been added to anyClub
```

- (b) Write function `CountLevel`, as started below. `CountLevel` counts and returns the number of club members of the specified level in `anyClub`.

For example, the call `CountLevel(Club1, 10)` returns 3, since there are 3 tenth graders in the German club. The call `CountLevel(Club2, 10)` returns 0 since there are no tenth graders in the Computer club.

Complete function `CountLevel` below. Assume that `CountLevel` is called only with parameters that satisfy its precondition.

```
int CountLevel(const Club & anyClub, int level)
// precondition: level is 9, 10, 11, or 12
// postcondition: returns the number of members in anyClub
//                of that level
```

(c) Write function `PrintClubsWithMostInLevel`, as started below.

`PrintClubsWithMostInLevel` is given an array of clubs and a grade level; it determines which of the clubs in the array has the most members in the given level in high school, and prints the name of that club. If there is a tie, multiple clubs are printed, one club per line.

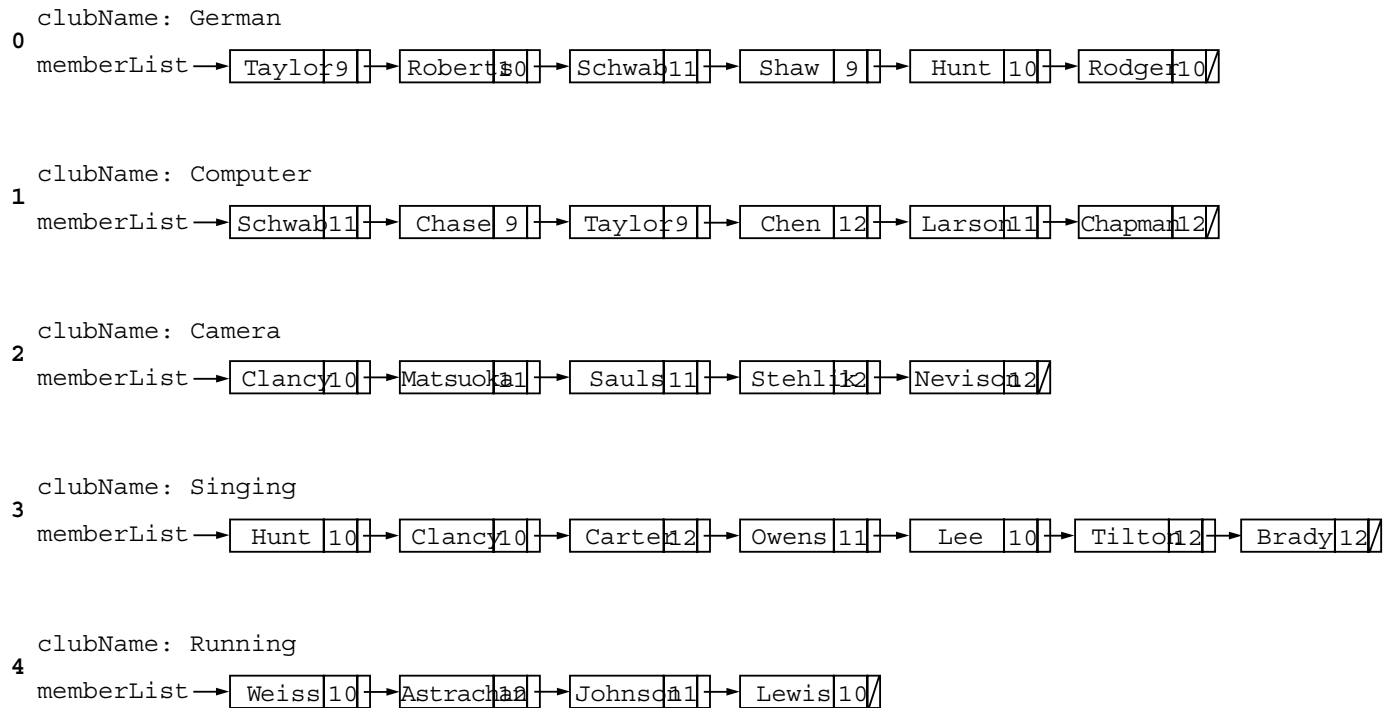
For example if `clubs` is the array shown below,

`PrintClubsWithMostInLevel (clubs, 10)` would print

German

Singing

as both of these clubs contain the largest number of tenth graders (3).

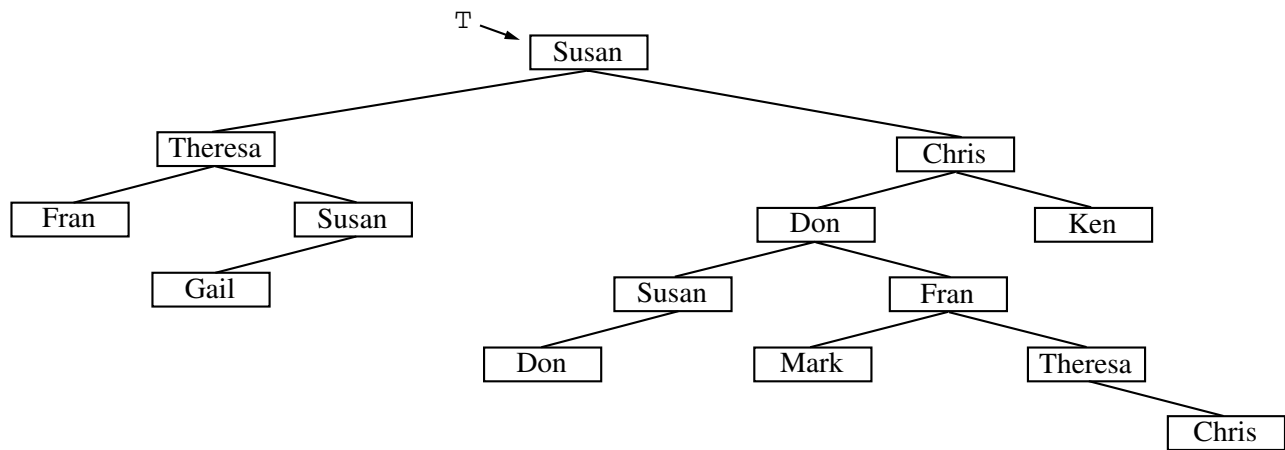


In writing `PrintClubsWithMostInLevel`, you may call function `CountLevel` specified in part (b). Assume `CountLevel` works as specified, regardless of what you wrote in part (b).

Complete function `PrintClubsWithMostInLevel` below. Assume that `PrintClubsWithMostInLevel` is called only with parameters that satisfy its precondition.

```
void PrintClubsWithMostInLevel(const apvector<Club> & clubsArray,
                               int level)
// precondition: clubsArray contains clubsArray.length() clubs
// postcondition: prints the name of the club or clubs in clubsArray
//                that contain the largest number of members in a given
//                level in high school (9 - 12), one club per line.
```

4. Consider a binary tree of names. Names may appear more than once in the tree as shown in the example below.



Assume that a binary tree of names is implemented using the following declaration.

```
struct TreeNode
{
    apstring name;
    TreeNode * left;
    TreeNode * right;
};
```

Assume that the integer function `Max` has been defined. `Max` returns the greater of its two integer parameters, as specified below.

```
int Max(int x, int y)
// postcondition: returns the maximum of x and y
```

A path in a tree is a sequence of nodes

$$\text{node}_1, \text{node}_2, \dots, \text{node}_k$$

such that for any  $j$ ,  $\text{node}_{j+1}$  is either the left child or right child of  $\text{node}_j$ . The length of a path is the number of nodes in the path ( $k$  in the example just given).

- (a) Write function `PathLength`, as started below. If person  $P$  is in tree  $T$ , then `PathLength(T, P, 1)` should return the length of the longest path from the root of  $T$  to a node containing  $P$ ; if person  $P$  does not appear in tree  $T$ , then `PathLength(T, P, 1)` should return 0. Note that parameter `level` can be used to keep track of the current level of the tree.

**GO ON TO THE NEXT PAGE**

For the tree given above, the following are examples of calls to `PathLength`.

<u>Function Call</u>	<u>Value Returned</u>
<code>PathLength(T, "Susan", 1)</code>	4
<code>PathLength(T, "Ken", 1)</code>	3
<code>PathLength(T, "Chris", 1)</code>	6
<code>PathLength(T, "David", 1)</code>	0
<code>PathLength(T-&gt;left, "Theresa", 1)</code>	1
<code>PathLength(T-&gt;right-&gt;left, "Don", 1)</code>	3

In writing `PathLength`, you may call function `Max` as specified in the beginning of this question. Assume that `Max` works as specified.

Complete function `PathLength` below.

```
int PathLength(TreeNode * T, const apstring & someName, int level)
```

- (b) Write function `RootPath`, as started below. `RootPath` should return the length of the longest path from the root of the tree to a node containing the same name as the root; if no node other than the root contains that name, then `RootPath` returns 1; if the tree is empty, `RootPath` should return 0. For the tree given above, the following are examples of calls to `RootPath`.

<u>Function Call</u>	<u>Value Returned</u>
<code>RootPath(T)</code>	4
<code>RootPath(T-&gt;left)</code>	1
<code>RootPath(T-&gt;right)</code>	5
<code>RootPath(T-&gt;left-&gt;left-&gt;left)</code>	0

In writing `RootPath`, you may call function `PathLength` specified in part (a). Assume that `PathLength` works as specified, regardless of what you wrote in part (a).

Complete function `RootPath` below.

```
int RootPath(TreeNode * T)
```