AP Computer Science A
1999 Free-Response Questions

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

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

Note: Assume that the standard libraries (`iostream.h`, `fstream.h`, `math.h`, etc.) and the AP C++ classes are included in any program that uses a program segment you write. If other classes are to be included, that information will be specified in individual questions. A Quick Reference to the AP C++ classes is included in the case study insert.

GO ON TO THE NEXT PAGE

1. Assume that student records are implemented using the following declaration.

```
struct StudentInfo
{
   apstring name;
   int creditHours;
   double gradePoints;
   double GPA;
};
```

(a) Write function `ComputeGPA`, as started below. `ComputeGPA` should fill in the `GPA` data member for the first `numStudents` records in its `apvector` parameter `roster`. A student's GPA (grade point average) is computed by dividing `gradePoints` by `creditHours`. The GPA for a student with 0 credit hours should be set to 0.

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

```
void ComputeGPA(apvector<StudentInfo> & roster, int numStudents)
// precondition:  roster contains numStudents records,
//                0 < numStudents ≤ roster.length(), in which the
//                name, creditHours and gradePoints data members
//                have been initialized.
// postcondition: The GPA data member for the first numStudents records
//                in roster has been calculated.
```

GO ON TO THE NEXT PAGE

(b) Write function `IsSenior`, as started below. `IsSenior` should return `true` if the given student has at least 125 credit hours and has a GPA of at least 2.0; otherwise, `IsSenior` should return `false`.

For example:

student				Result of the call `IsSenior(student)`
name	creditHours	gradePoints	GPA	
King	45	171	3.8	`false` (not enough credit hours)
Norton	128	448	3.5	`true`
Solo	125	350	2.8	`true`
Kramden	150	150	1.0	`false` (GPA too low)

Complete function `IsSenior` below.

```
bool IsSenior(const StudentInfo & student)
// postcondition: returns true if this student's credit hours ≥ 125
//                and GPA ≥ 2.0; otherwise, returns false
```

Part (c) begins on page 6.

GO ON TO THE NEXT PAGE

(c) Write function FillSeniorList, as started below. FillSeniorList determines which students in the array roster are seniors and copies those students' records to the array seniors. It should also set the value of parameter numSeniors to be the number of seniors in the array seniors.

In writing FillSeniorList, you may call function IsSenior specified in part (b). Assume that IsSenior works as specified, regardless of what you wrote in part (b).

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

```
void FillSeniorList(const apvector<StudentInfo> & roster,
                    int numStudents, apvector<StudentInfo> & seniors,
                    int & numSeniors)
// precondition: roster contains numStudents records,
//               0 < numStudents ≤ roster.length(),
//               and seniors is large enough to hold all of
//               the seniors' records
```

ADDITIONAL WORKSPACE

GO ON TO THE NEXT PAGE

2.

(a) Write function `WordIndex`, as started below. The array `wordList` contains `numWords` strings in alphabetical order. If `word` is already in `wordList`, then `WordIndex` should return the index of `word` in `wordList`. Otherwise, `WordIndex` should return the index of the first string in `wordList` that comes after `word` in alphabetical order; it should return `numWords` if `word` comes after all of the strings in `wordList` in alphabetical order.

For example, assume that array `wordList` is as follows:

0	1	2	3
"apple"	"berry"	"pear"	"quince"

Function Call	Value Returned
`WordIndex("air", wordList, 4)`	0
`WordIndex("apple", wordList, 4)`	0
`WordIndex("orange", wordList, 4)`	2
`WordIndex("raspberry", wordList, 4)`	4

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

```
int WordIndex(const apstring & word,
              const apvector<apstring> & wordList, int numWords)
// precondition: wordList contains numWords strings in alphabetical
//               order, 0 ≤ numWords < wordList.length()
```

GO ON TO THE NEXT PAGE

(b) Write function `InsertInOrder`, as started below. The array `wordList` contains `numWords` strings in alphabetical order. If the string `word` is already in `wordList`, `InsertInOrder` should not change any of its parameters. Otherwise, it should insert `word` into `wordList` in alphabetical order (i.e., all values greater than `word` should be moved one place to the right to make room for `word`), and it should also increment `numWords` by 1. Assume that `wordList.length()` is greater than `numWords`.

In the examples below, `numWords = 3` before the following call is made.

```
InsertInOrder("pear", wordList, numWords)
```

Before the call	After the call	
wordList	wordList	numWords
"apple" "berry" "quince"	"apple" "berry" "pear" "quince"	4
"apple" "berry" "pear"	"apple" "berry" "pear"	3
"apple" "fig" "peach"	"apple" "fig" "peach" "pear"	4
"quince" "raisin" "tart"	"pear" "quince" "raisin" "tart"	4

In writing `InsertInOrder`, you may include calls to function `WordIndex` specified in part (a). Assume that `WordIndex` works as specified, regardless of what you wrote in part (a).

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

```
void InsertInOrder(const apstring & word,
                   apvector <apstring> & wordList, int & numWords)
// precondition:  wordList contains numWords strings in alphabetical
//                order, 0 ≤ numWords < wordList.length()
// postcondition: if word was already in wordList, then wordList and
//                numWords are unchanged;
//                otherwise, word has been inserted into wordList in
//                sorted order, and numWords has been incremented by 1
```

GO ON TO THE NEXT PAGE

3. 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) + carryDown

  2.2 let carryDown be (d % 2) * 5

3. Normalize the result
```

   Complete member function `Div2` below.

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

GO ON TO THE NEXT PAGE

(b) Write function `DivPos`, as started below. `DivPos` returns the quotient of the integer division of `dividend` by `divisor`. 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`:

bigNum1	bigNum2	DivPos(bigNum1, bigNum2)
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 algorithm for implementing division using binary search is as follows:

1. Initialize `low` to `0` and `high` to `dividend`.
2. For each iteration,

   2.1 compute `mid = (low + high + 1)`

   2.2 divide `mid` by `2`

   2.3 if `mid * divisor` is larger than `dividend` (`mid` is too large to be the quotient) then set `high` equal to `mid - 1` else set `low` equal to `mid`.

3. When `low == high` the search terminates, and you should return `low`.

In writing function `DivPos`, 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 function `DivPos` below. Assume that `DivPos` is called only with parameters that satisfy its precondition.

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

4. 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 4×5 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.

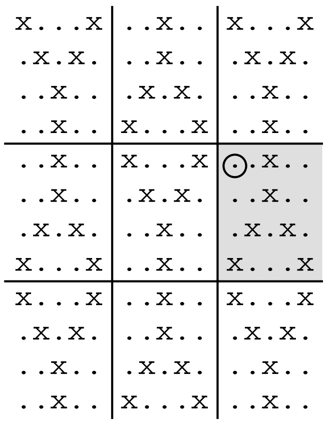
```
Quilt::Quilt(istream & inFile, int rowsOfBlocks, int colsOfBlocks)
  : myBlock(0, 0), myRowsOfBlocks(rowsOfBlocks),
    myColsOfBlocks(colsOfBlocks)
// precondition:  inFile is open, rowsOfBlocks > 0, colsOfBlocks > 0
// postcondition: myRowsOfBlocks and myColsOfBlocks are initialized to
//                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];
    }
  }
}
```

GO ON TO THE NEXT PAGE

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()
```

GO ON TO THE NEXT PAGE

ADDITIONAL WORKSPACE

END OF EXAMINATION

- MAKE SURE THAT YOU HAVE COMPLETED THE IDENTIFICATION INFORMATION AS REQUESTED ON THE BACK COVER OF THIS BOOKLET.
- CHECK TO SEE THAT YOUR AP NUMBER APPEARS IN THE BOX ON THE BACK COVER.
- MAKE SURE THAT YOU HAVE USED THE SAME SET OF AP NUMBER LABELS ON ALL AP EXAMINATIONS YOU HAVE TAKEN THIS YEAR.