2000 Advanced Placement Program®
Free-Response Questions

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 3,800 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 5,000 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®, the Advanced Placement Program® (AP®), and Pacesetter®. 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.

The College Board

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 (e.g., `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. Unless otherwise noted, assume that all functions are called only when their preconditions are satisfied. A Quick Reference to the AP C++ classes is included in the case study insert.

1. A *mode* is a value in an array that is larger than both the value immediately before it in the array and the value immediately after it. In other words, a mode occurs at index `k` in the array `A` if `A[k] > A[k - 1]` and `A[k] > A[k + 1]`. The array is *unimodal* if the values increase until they reach a mode, then decrease, so that there is only one mode. For example, the array `A` shown below is unimodal with its mode occurring at index `4`. Assume that the mode does not occur at the first or last entry in the array.

Index k	A[k]	
0	3	
1	5	
2	9	
3	10	
4	12	← mode
5	11	
6	9	
7	4	

(a) Write function `IsMode`, as started below. `IsMode` returns `true` if `data[k]` is larger than `data[k - 1]` and larger than `data[k + 1]`; otherwise, it returns `false`. In the example above, the call `IsMode(A, 4)` returns `true` and the call `IsMode(A, 5)` returns `false`.

Complete function `IsMode` below.

```
bool IsMode(const apvector<int> & data, int k)
// precondition:  0 < k < data.length() - 1
```

(b) Write function `ModeIndex`, as started below. `ModeIndex` returns the index of the mode of `data`. You may assume that `data` is unimodal and the mode occurs at an index `k`, where `0 < k < data.length() - 1`. In the example above, the call `ModeIndex(A)` returns `4`.

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

Complete function `ModeIndex` below.

```
int ModeIndex(const apvector<int> & data)
// precondition:  data is unimodal and data.length() ≥ 3
```

(c) Write function `PrintHistogram`, as started below. `PrintHistogram` prints a character histogram of a unimodal array of nonnegative values, `data`, such that the longest bar of the histogram (the mode) has `longestBar` characters `barChar`, and all other bars have a number of `barChar` characters proportional to the corresponding value in the array `data` (rounding down).

For example, assume that `apvector data` contains the values shown below.

The call `PrintHistogram(data, 20, 'x')` will print the histogram shown in the Output column below.

Index `k`	`data[k]`	Length of bar	Output of call `PrintHistogram (data, 20, 'x')`
0	3	5	xxxxx
1	5	8	xxxxxxxx
2	9	15	xxxxxxxxxxxxxxx
3	10	16	xxxxxxxxxxxxxxxx
4	12	20	xxxxxxxxxxxxxxxxxxxx
5	11	18	xxxxxxxxxxxxxxxxxx
6	9	15	xxxxxxxxxxxxxxx
7	4	6	xxxxxx

In writing `PrintHistogram`, you may call functions `IsMode` and `ModeIndex` specified in parts (a) and (b). Assume that `IsMode` and `ModeIndex` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `PrintHistogram` below.

```
void PrintHistogram(const apvector<int> & data,
                    int longestBar, char barChar)
// precondition:  data is unimodal and data.length() ≥ 3;
//                data[k] ≥ 0 for 0 ≤ k < data.length()
```

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 exam.

   (a) Write the new `BigInt` public member function `IsOdd`, as started below. `IsOdd` should return `true` if the `BigInt` is odd; otherwise, it should return `false`.

   You may NOT assume that the `%` or `%=` operators have been defined for the `BigInt` class.

   Complete function `IsOdd` below.

```
bool BigInt::IsOdd() const
// postcondition: returns true if this BigInt is odd;
//                otherwise, returns false
```

   (b) Write the free function `Power`, as started below. `Power` returns the value of `base` to the `exp` power, that is $\texttt{base}^{\texttt{exp}}$, where `exp` $\geq$ `0`. For example, the call `Power(3, 5)` returns `243`, which is 3^5.

   You must use the following algorithm.

> Initialize a variable, `product`, to be `1`.
> While `exp` is not zero do the following:
> if `exp` is odd, `product` is set to `product` times the `base`
> square the `base`
> divide `exp` by two
> When done, `product` contains the result.

   Assume that a new member function, `DivBy2`, has been defined for the `BigInt` class, as specified below. `DivBy2` divides this `BigInt` by `2` (using integer division). (You do not need to write the body of `DivBy2`.)

```
void BigInt::DivBy2();  // this BigInt is divided by 2
```

   In writing `Power`, you may use the `BigInt` public member function `DivBy2` specified above and you may use the `BigInt` public member function `IsOdd` specified in part (a). Assume that `IsOdd` works as specified, regardless of what you wrote in part (a).

   Complete function `Power` below.

```
BigInt Power(const BigInt & base, const BigInt & exp)
// precondition:  base > 0 and exp ≥ 0
// postcondition: returns the value of base to the exp
```

3. A `WordCollection`, shown in the class declaration below, stores a group of words. The collection may store multiple instances of any word. In this question, you will not implement any of the member functions of class `WordCollection`.

```
class WordCollection
{
  public:
    int Size() const;
       // returns the total number of items stored in the collection

    void Insert(const apstring & word);
       // adds word to the collection (duplicates allowed)

    void Remove(const apstring & word);
       // removes one instance of word from the collection if word is
       // present; otherwise, does nothing

    apstring FindKth(int k) const;
       // returns kth word in alphabetical order, where
       // 1 ≤ k ≤ Size()

    // other public member functions not shown

  private:
    // private data members not shown
};
```

The public member function `FindKth` returns the *k*th word in alphabetical order from the collection (the word with rank *k*), even though the underlying implementation of `WordCollection` may not be sorted. The rank ranges from 1 (first in alphabetical order) to *N*, where *N* is the number of words in the collection. For example, assume that `WordCollection C` stores the following words.

```
{ "at", "bad", "all", "at" }
```

The following table illustrates the results of calling `C.FindKth(k)`.

k	C.FindKth(k)
1	"all"
2	"at"
3	"at"
4	"bad"

(a) Write free function `Occurrences`, as started below. `Occurrences` returns the number of times that `word` appears in `WordCollection C`. If `word` is not in `C`, `Occurrences` should return 0.

In writing `Occurrences`, you may call any of the member functions of the `WordCollection` class. Assume that the member functions work as specified.

Complete function `Occurrences` below.

```
int Occurrences(const WordCollection & C, const apstring & word)
// postcondition: returns the number of occurrences of word in C
```

(b) Write free function `RemoveDuplicates`, as started below. `RemoveDuplicates` removes all but one occurrence of `word` from `C`. If `word` is not in collection `C`, then `RemoveDuplicates` does nothing.

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

Complete function `RemoveDuplicates` below.

```
void RemoveDuplicates(WordCollection & C, const apstring & word)
// postcondition: if word is present in C, all but one occurrence
//                is removed; otherwise, C is unchanged
```

(c) Write free function `MostCommon`, as started below. `MostCommon` returns the word that appears most often in the collection. If there is more than one such word, return any one of them. You may assume that `C` is not empty.

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

Complete function `MostCommon` below.

```
apstring MostCommon(const WordCollection & C)
// precondition:  C is not empty
// postcondition: returns the word that appears most often in C;
//                if there is more than one such word,
//                returns any one of those words
```

4. One way of encrypting a word is to encrypt pairs of letters in the word together. A scheme to do this is to fill a 6 x 6 square with the 26 capital letters of the alphabet and the ten digits `'0'` through `'9'`. Each letter and digit appears exactly once in the square.

   To encrypt a letter pair, the rectangle formed by the two letters is used. Each letter of the original pair is replaced by the letter located on the same row and in the other corner of the rectangle. If both letters happen to be in the same row or column, the letters are swapped.

   For example, in the following arrangement `AP` is encrypted as `DM`.

```
S   T   U   V   W   X

Y   Z   0   1   2   3

4   5   6   7   8   9

A---B---C---D   E   F
|           |
G   H   I   J   K   L
|           |
M---N---O---P   Q   R
```

   Consider the following declaration for a class that uses this scheme to encrypt a word.

```
struct Point
{
  int row;
  int col;

  Point();                          // default constructor
  Point(int newRow, int newCol); // sets row to newRow, col to newCol
};

class Encryptor
{
  public:
    Encryptor();
       // fills the matrix with the 26 letters of the alphabet
       // and the 10 digits '0' through '9'

    apstring EncryptWord(const apstring & word) const;
       // returns an encrypted form of the word

  private:
    apmatrix<char> myMat;

    apstring EncryptTwo(const apstring & pair) const;
       // returns an encrypted form of the pair

    Point GetCoordinates(char ch) const;
       // returns the coordinates of ch in the 2-dimensional array
};
```

(a) Write member function `GetCoordinates`, as started below. `GetCoordinates` takes a given letter or digit and returns its row and column in the 2-dimensional array. Assume that the parameter `ch` is a capital letter in the range `'A'` through `'Z'` or a digit in the range `'0'` through `'9'`.

The following example shows the point locations of character `ch` in the given matrix.

Encryptor.myMat						ch	Point coordinates	
S	T	U	V	W	X	P	row = 5	col = 3
Y	Z	0	1	2	3	8	row = 2	col = 4
4	5	6	7	8	9	M	row = 5	col = 0
A	B	C	D	E	F			
G	H	I	J	K	L			
M	N	O	P	Q	R			

Complete function `GetCoordinates` below.

```
Point Encryptor::GetCoordinates(char ch) const
// precondition:  'A' ≤ ch ≤ 'Z' or '0' ≤ ch ≤ '9'
// postcondition: returns the row and column number of the
//                location of ch in myMat
```

(b) Write member function `EncryptTwo`, as started below. `EncryptTwo` is passed a two-character string and returns an encoded two-character string.

The encoding of a letter pair is formed as follows.

1. If both letters are in the same row or column, swap the two letters.
2. Otherwise, find the other two corners of the rectangle formed by the two letters. Each letter of the original pair is replaced by the letter located on the same row and in the other corner of the rectangle.

For example, to encrypt a letter pair, say `NE`, look at the rectangle with corners `N` and `E`. The encrypted letter pair is `QB` because `Q` is the letter at the other corner on the same row as `N`, and `B` is the letter at the other corner on the same row as `E`.

```
        S   T   U   V   W   X

        Y   Z   0   1   2   3

        4   5   6   7   8   9

        A   B---C---D---E   F
            |           |
        G   H   I   J   K   L
            |           |
        M   N---O---P---Q   R

  Letters: BR  NE  ET  RE  TH  PR  GG
Encrypted: FN  QB  BW  QF  HT  RP  GG
```

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

Complete function `EncryptTwo` below.

```
apstring Encryptor::EncryptTwo(const apstring & pair) const
// precondition:  pair.length() is 2
// postcondition: returns an encoded two-character string
```

(c) Write member function `EncryptWord`, as started below. `EncryptWord` takes a word parameter and returns a string that contains the encryption of that word. Every two letters of the word are examined and encrypted by replacing the original letters with those located in the opposite corners of the rectangle formed by the two letters. If the original word contains an odd number of letters the last letter is unchanged.

The following are examples of encrypted words using the 2-dimensional array shown below.

```
S   T   U   V   W   X

Y   Z   0   1   2   3

4   5   6   7   8   9

A   B   C   D   E   F

G   H   I   J   K   L

M   N   O   P   Q   R
```

```
     Word:    COMPUTER    SCIENCE    STUDENTS
Encrypted:    OCPMTUFQ    UAKCOBE    TSVCBQST
```

In writing `EncryptWord`, you may call `EncryptTwo` specified in part (b). Assume that `EncryptTwo` works as specified, regardless of what you wrote in part (b).

Complete function `EncryptWord` below.

```
apstring Encryptor::EncryptWord(const apstring & word) const
// precondition:  word contains only capital letters 'A' through 'Z'
//                and digits '0' through '9'.
// postcondition: returns an encrypted version of word, in which every
//                two letters have been examined and encrypted by
//                replacing the original letters with those located
//                in the opposite corners of the rectangle formed by
//                the two letters. If the original word contains an odd
//                number of letters, the last letter is left unchanged.
```

END OF EXAMINATION