AP Computer Science

THE LARGE INTEGER CASE STUDY IN C++

Advanced Placement Program®
THE COLLEGE BOARD

COLLEGE BOARD REGIONAL OFFICES

Middle States

Mary Alice Gilligan
Suite 410, 3440 Market Street
Philadelphia, PA 19104-3338
(215) 387-7600

Midwest

Bob McDonough/Paula Herron
Suite 401, 1800 Sherman Avenue
Evanston, IL 60201-3715
(847) 866-1700

New England

Fred Wetzel
470 Totten Pond Road
Waltham, MA 02154-1982
(617) 890-9150

South

Geoffrey Freer/Tom New
Suite 250, 2970 Clairmont Road
Atlanta, GA 30329-1639
(404) 636-9465

Southwest

Paul Williamson/Frances Brown/Mondy Raibon
Suite 1050, 98 San Jacinto Boulevard
Austin, TX 78701-4039
(512) 472-0231

West

Lindy Daters/Claire Pelton
Suite 480, 2099 Gateway Place
San Jose, CA 95110-1017
(408) 452-1400

Canada (AP Program Only)

George Ewonus
212-1755 Springfield Road
Kelowna, B.C., Canada V1Y 5V5
(250) 861-9050

NATIONAL OFFICE

Wade Curry • Philip Arbolino • Charlotte Gill • Frederick Wright
45 Columbus Avenue • New York, NY 10023-6992 • (212) 713-8000

This booklet was 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.

Founded in 1900, the College Board is a not-for-profit educational association that supports academic preparation and transition to higher education for students around the world through the ongoing collaboration of its member schools, colleges, universities, educational systems, and organizations. In all of its activities, the Board promotes equity through universal access to high standards of teaching and learning and sufficient financial resources so that every student has the opportunity to succeed in college and work. The College Board champions — by means of superior research; curricular development; assessment; guidance, placement, and admission information; professional development; forums; policy analysis; and public outreach — educational excellence for all students.

THE COLLEGE BOARD: EDUCATIONAL EXCELLENCE FOR ALL STUDENTS

Advanced Placement
Computer Science

The Large Integer Case Study in C++

A Manual for Students

The AP Program wishes to acknowledge and to thank
Owen Astrachan of Duke University for developing this case study
and the accompanying documentation.

This is the premiere posting of the Advanced Placement Computer Science Large Integer Case Study in C++. Comments and/or suggestions regarding this material should be sent to Gail Chapman at gchapman@ets.org.

For more information about AP Computer Science, see the AP Computer Science section of College Board Online (CBO):

http://www.collegeboard.org/ap/computer-science/html/indx001.html

College Board Online also has a publications store where you can place orders for College Board and AP publications. The AP Aisle of the College Board Online store can be found at:

http://cbweb2.collegeboard.org/shopping/

CONTENTS

LARGE INTEGER CASE STUDY IN C++

IMPLEMENTATION OF THE LARGE INTEGER PACKAGE

APPENDICES

Large Integer Case Study in C++
for
AP Computer Science

Introduction

This document describes the process of producing a solution to a programming problem. It is intended to provide an opportunity for "apprenticeship learning." The narrative is written as instruction from an expert to an apprentice, and study questions represent places where the expert would say, "Now you go try this." Ideally, you will get involved in the solution of the problem and be able to compare your own design and development decisions with those of the authors.

This case study includes programs that solve the problem, together with a narrative of how the programs are designed and implemented. It describes and justifies choices made during the problem solution. It also contains exercises to guide your study of the problem and its solution. The program code described here is available via the Internet (see the URL below).

As you read the case study and the code, you may have questions regarding C++ and how it is used. In some cases the questions may not be answered in this document. You should consult a teacher, a book, or the website with supporting materials and explanations at:

http://www.collegeboard.org/ap/computer-science/html/indx001.html.

Problem Statement

Arithmetic operations are fundamental in computing and programming. In many programming environments the largest value of an integer variable is much too small to represent large quantities such as the population of the world, the U.S. national debt, the number of occurrences of the letter 'e' in Melville's *Moby Dick*, and many other quantities. Encryption and verification methods are other applications that use extremely large integers. Although a 16 bit integer is limited (normally) to positive values less than 32,768, even a 32 bit integer has a maximum value of 2,147,483,647 which is still too small to represent some of the quantities mentioned above. Using floating point (real) numbers isn't always a reasonable option since computer arithmetic with such numbers is inexact.

We will construct a software package that permits storage and manipulation of large (i.e., greater than INT_MAX[1]) integer values. We'll try to produce a tool that might be used in a variety of environments where such values are needed. Once this package has been built, it should be possible to use the code, unchanged, in any programming context. Some examples of applications include:

- A calculator that handles very large integer values.
- A program that investigates properties of numbers including Fibonacci sequences, large factorial values, and prime numbers.
- A program used for encrypting messages and transactions that take place over the Internet.
- A banking program that uses large account balances and transactions.
- A spreadsheet program that uses large values.

We'll implement the software package as a C++ class named BigInt. To test the class we'll use a program that works as a simple, line-oriented calculator. The first version of the class will use int values rather than large integer values. As we design and implement the class BigInt we'll be able to use the same calculator program to test the implementation since all interaction with the class is through public member functions which will not change although the private implementation will change.

Description of the Calculator

The calculator will implement addition, subtraction, and multiplication. An expression like 25 ∗ 32 + 5 is evaluated using the calculator program by entering numbers and operators one per line. After each value is entered, the current value of the expression is displayed and used as the left operand for the next operator. Entry of an "=" terminates the program. A sample run is shown below. Notice that in its current form it accepts C++ int values rather than BigInt values. The code for this calculator program is given in Appendix A.

```
Enter value: 5000
——> 5000
Enter operator (+ - * (= to quit)): *
```

[1] The constant INT_MAX is defined in the header file <limits.h>.

```
Enter value: 3
——> 15000
Enter operator (+ - * (= to quit)): +
Enter value: 17
——> 15017
Enter operator (+ - * (= to quit)): *
Enter value: 4
——> -5468
Enter operator (+ - * (= to quit)): =
```

The limitations of arithmetic in C++ are shown in the sample run where the value of $15017 * 4$ is given as -5468 rather than the correct value of 60,068[2]. This kind of error is called *overflow*; the maximum integer value has been exceeded and the result is incorrect. This calculator program performs only three arithmetic operations. A fully functional calculator would include additional operations including division, exponentiation, etc., and a method for evaluating expressions other than the line-oriented approach.

Study Questions

1. Suppose you want to add unary operations (i.e., ones requiring only one operand) to the calculator. For example, we might add "M" to indicate unary minus (i.e., change the sign of the old value) and "A" to indicate absolute value. What changes would need to be made to the program?
2. Suppose that words rather than single character symbols are to be used for the operators (e.g., the user types "times" instead of "$*$"). Where in the program would the necessary changes occur?
3. Suppose that the calculator is to be converted to use C++ double values rather than integers. Where would changes need to be made?
4. Most modern systems support graphical displays and mouse (or other pointer) input. Rewrite the Calculator program to take advantage of these features to produce an on-screen push-button calculator (this is a large programming project).

[2] Not all computers will generate incorrect results for $15,017 * 4$. However, all computers will generate an overflow for some int values.

5. Show how a new command, "C", which acts like the "Clear" command on a calculator could be implemented. The clear command sets the current value to zero.
6. (AB only) How could the calculator be modified to allow the use of parentheses to control the order of operations?

Specification of BigInt

We'll implement a new class BigInt that performs arithmetic with values that exceed those of the type int. Before proceeding, we'll develop a detailed specification for the class BigInt. Since the specification describes the new class abstractly, with no reference to how it will be implemented, it is referred to as an abstract data type, or ADT. The specification includes the kinds of values that can be represented by the class BigInt and the operations that can be performed on the values.

Size limits: We want no limit on the size of BigInt values other than those limits caused by the finite memory of computers. In other words, we do not want to predetermine a limit on the number of digits in a BigInt value. This requirement will help determine how we store the digits in a BigInt object.

Operations: We'll categorize BigInt operations as shown below. These aren't all the operations that can be performed, but are more than enough to implement the calculator program. The operations will support other programs that use BigInt values, too.

Fundamental

- Constructors

I/O

- operator <<
- operator >>
- Print

Arithmetic

- operator +=
- operator –=
- operator *=

Comparison

- operator <
- operator >
- operator ==
- operator !=

Conversion

- ToDouble
- ToString
- ToInt

Full specifications for each of these functions will be provided later. Most of the functions do exactly as the name implies (i.e., operator += adds BigInt values). We'll provide a brief commentary here for those functions and operators that are different from int functions.

We will implement several constructors. For example, we will want a default constructor so that we can define vectors of BigInt values. We'll want to construct a BigInt from an int value. It will also be useful to construct a BigInt from a string value since a string can contain as many characters as needed to represent any BigInt value. In some sense the int and string constructors allow us to convert int and string values to corresponding BigInt values. The "To" functions: ToDouble(), ToString(), and ToInt() permit conversion in the other direction, e.g., ToDouble() converts a BigInt to the corresponding double value.

Initially we will implement neither a destructor, nor a copy constructor, nor an assignment operator for the class BigInt. These functions are necessary only in certain situations so we may need to revisit this decision later when we decide how to implement BigInt values.

We show three arithmetic assignment operators: +=, −=, and *=. As we will see, once these operators are implemented, the implementation of the corresponding arithmetic operators: +, −, * is straightforward.

Study Questions

1. What are the largest and smallest integer values in the programming environment you use?

2. Each BigInt object will need to store the digits that represent the BigInt value. The decision to allow arbitrarily large BigInt values affects the choices for storing digits. Name one method for storing digits that will permit an arbitrary number of digits to be stored. What effect would a limit on the number of digits in a BigInt have in the design of the BigInt class?

3. Based on your knowledge of pencil-and-paper methods for doing arithmetic, what do you think will be the most difficult arithmetic operation (+, ∗, −) to implement for the BigInt class? Why?

4. Experiment with the calculator. If you enter *abcd1234* when a number is expected, what happens? If you enter *1234abcd* is the behavior different? What happens if you enter an operator that's not one of the three that are implemented?

5. List as many operations as you can that are performed on integers, but that are not included in the list of BigInt functions and operators above.

6. (AB only) What implementation decisions would require providing a destructor, a copy constructor, and an assignment operator?

7. Consider the headers for operator− and operator+ given below.

```
BigInt operator - (const BigInt & big, int small);
// postcondition: returns big - small
BigInt operator + (const BigInt & big, int small);
// postcondition: returns big + small
```

   Write the body of operator− assuming that operator+ has been written.

Solution Narrative

We'll begin by considering and resolving some of the missing specifications of the problem. We'll move next to the design of the data structure for storing a BigInt. Different design possibilities are explored and a representation of the final design evolves as part of this exploration. Error handling will be discussed at this stage.

After discussing the design, we'll move to the BigInt implementation. Several BigInt operations are designed, coded, and tested. The tests reveal some weaknesses in the original design, and these are repaired. During this process the calculator program is modified to make use of BigInt values. Opportunities for the use of BigInt values in different contexts are examined, too.

Design Goals

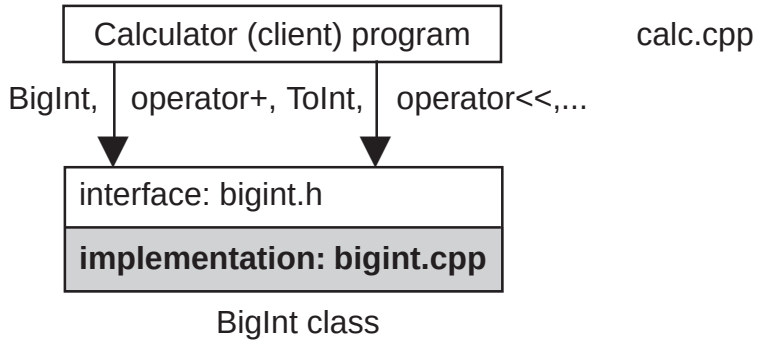
The class BigInt will be implemented as a header file declaring the class and a corresponding implementation file consisting of the member functions of the class and other non-member functions used to manipulate BigInt objects. Throughout our discussion, we will refer to the declarations, operators and functions as the "class" and the calculator (or other) program using the class as the "client" program. A well-designed class can be used with any client program without changing the class.

The class should:

- be convenient and intuitive for programmers to use;
- be useful in a wide variety of applications;
- mimic operations on C++ integers so that a program written and debugged using int values can be quickly and reliably converted to use BigInt values; and
- utilize information hiding; it should be possible to use BigInt variables without knowledge of the underlying data structures or algorithms used.

Overall Structure

The calculator program will be organized as shown in the diagram below:



The calculator, or other client program, creates and manipulates BigInt values by making calls to the functions and operators whose prototypes are declared in bigint.h, the interface part of the class. The client program does not refer to how the class BigInt is implemented. The implementation is hidden from the client as shown by the shaded box in the diagram. For example, suppose an array of digits (0..9) is used to represent a BigInt. This choice should only be reflected in the shaded rectangle. The client program does not contain any code that depends upon this choice. If the implementation is changed to use an array of characters ('0'..'9') rather than digits, the client code will not need to be changed.

A C++ class can be used to enforce this separation of implementation from interface. Some implementation decisions are visible in the private section of a class declaration, but client programs cannot access private data or private functions. Client programs manipulate BigInt values only through public member functions and other functions declared in bigint.h as helping, non-member functions. All these functions are implemented in the file BigInt.cpp in code that client programs access by function calls.

Error Handling

As the sample run of the calculator program earlier showed, arithmetic can result in errors because of the limited range of values of the type integer. Errors can result from bad I/O, too. For example, if you enter "apple" when a program expects a numeric value an error usually occurs. These same kinds of errors will occur when BigInt operations are used. As part of the design of the class, we'll need to decide how to handle such errors. For example, I/O errors could be handled by reading all input using strings so that bad inputs (e.g., a letter instead of a digit) are trapped internally. Functions for I/O, arithmetic, and other BigInt operations will need to indicate that an error has occurred or handle the error. We'll consider several options for handling errors. It's possible, of course, to avoid error handling entirely. Client programs would then be responsible for not calling any BigInt functions and operators that could cause errors. However, making error-checking the responsibility of the BigInt class makes developing client programs simpler. We'll consider four methods for handling errors.

1. Errors can be ignored, operations are performed as well as possible. For example, illegal values such as 0139abcd3 can be converted to zero.
2. Errors can be trapped, an error message printed, and the client program halted.
3. An exception could be thrown when errors occur. Client programs are responsible for catching exceptions and taking appropriate action.
4. A BigInt could include a special "error" value in addition to an arithmetic value. The client program is responsible for checking a BigInt error status and taking appropriate action.

Ignoring errors is undesirable since it will lead to incorrect output with no indication that an error has occurred. For example, if the class is used in a banking program and a division by zero error results in improper bookkeeping, customers won't be happy that errors have been ignored to make writing the class simpler.

Sometimes halting a program is the only course of action. For example, out-of-bounds indexing of apstring and apvector variables stops a client program after printing an error message. On the other hand, if your program is halted you might lose data.

Using exceptions is probably the best approach, but exceptions are not supported at this time by all C++ compilers. Furthermore, properly designed exceptions rely on inheritance, a topic not currently covered in the AP Computer Science course description.

Making client programs check for errors offers flexibility since programmers can decide when to check for errors. However, associating an error value with each BigInt may complicate the coding of the arithmetic operations since we'll need to handle error values appropriately.

We'll choose the second method for handling errors: the errors will be trapped in the implementation of the BigInt member functions, an error message printed, and the program halted. This has drawbacks, but mirrors the approach taken by the apmatrix and apvector classes.

Study Questions

1. Consider the error handling provided by your C++ system. What does the system do if a file is not present in a call to open? What happens on integer overflow or divide by zero? Determine which method(s) are used and discuss the relative desirability of other options.
2. List several errors that might be generated by BigInt operations and develop a declaration for an enumerated type (enum) to hold the errors.
3. Some systems allow error checking to be "turned off" entirely for greater speed. Under what circumstances is this approach preferred?

4. Consider another method for handling errors:

   Use an interactive error-handling approach. An error message is displayed to the user who then has the option of (a) correcting the value that caused the error, (b) halting the program, or (c) ignoring the error.

   Describe the strengths and weaknesses of this approach.

5. Consider another method for handling errors:

   Error results are stored in a single global variable. This is set initially to indicate a "no error" condition. Whenever an error is detected, the global variable is set to an appropriate value, and the client program is responsible for examining the value of the global variable.

   Describe the strengths and weaknesses of this approach.

Formal Specifications for BigInt Functions

We cannot write code until we have specified exactly what each function is to accomplish. These specifications will include detailed pre- and post-conditions for each BigInt function. Function prototypes with pre- and post-conditions are shown below for the principle BigInt functions. In several of the prototypes the parameters are named lhs and rhs for left-hand side and right-hand side, respectively. For example, in the expression if (big == 3) the argument big is passed to the parameter lhs and the argument 3 is passed to the parameter rhs.[3]

[3] In many of the BigInt prototypes, parameters are declared as const reference parameters, e.g., in the BigInt constructor from an apstring, the prototype is BigInt(const apstring & s). Reference parameters are used to save time and memory since no local copy is made for a reference parameter whereas a local copy is made when parameters are passed by value. The const modifier makes it impossible to modify the parameter, so the efficiency of pass-by-reference is obtained without sacrificing the safety of pass-by-value. Unless there is a reason to have a local copy, class types (e.g., apstring and BigInt) parameters should be const reference rather than value.

Constructors

```
BigInt::BigInt()
// postcondition: bigint initialized to 0

BigInt::BigInt(int num)
// postcondition: bigint initialized to num

BigInt::BigInt(const apstring & s)
// precondition: s consists of digits only, optionally preceded
//               by + or -
// postcondition: bigint initialized to integer represented
//                by s if s is not a well-formed BigInt (e.g.,
//                contains non-digit characters) then an error
//                message is printed and abort called
```

I/O

```
ostream & BigInt::Print(ostream & os) const
// postcondition: BigInt inserted onto stream os

ostream & operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out

ostream & operator >>(istream & in, BigInt & big)
// postcondition: big extracted from in, must be whitespace
//                delimited
```

Arithmetic

```
const BigInt & BigInt::operator -=(const BigInt & rhs)
// postcondition: returns value of bigint - rhs after
//                subtraction

const BigInt & BigInt::operator +=(const BigInt & rhs)
// postcondition: returns value of bigint + rhs after
//                addition

const BigInt & BigInt::operator *=(const BigInt & rhs)
// postcondition: returns value of bigint * rhs after
//                multiplication
```

Comparison

```
bool operator == (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs == rhs, else returns false

bool operator != (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs != rhs, else returns false

bool operator > (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs > rhs, else returns false

bool operator < (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs < rhs, else returns false
```

Assignment and Conversion

```
 apstring BigInt::ToString() const
// postcondition: returns apstring equivalent of BigInt

int BigInt::ToInt() const
// precondition: INT_MIN <= self <= INT_MAX
// postcondition: returns int equivalent of self

double BigInt::ToDouble() const
// precondition: DBL_MIN <= self <= DBL_MAX
// postcondition: returns double equivalent of self
```

Data Structure Design

The function headers in the previous section provide the interface to the BigInt class. We now consider how to implement the type BigInt.

Choosing a data representation

We can represent a BigInt as a sequence or list of decimal digits 0-9. Each BigInt will also have a flag indicating if it is positive or negative. In C++, sequences can be stored using arrays[4], files, and linked lists (linked lists are part of the AB syllabus). Although it's possible to use files, arrays provide a much simpler mechanism for accessing the individual digits that make up a BigInt. There are several ways to store individual digits in an array. Determining the best method requires thinking about how the

[4] In this document when we use array we mean the apvector class rather than the built-in array type available in C++.

BigInt operations access individual digits. For example, three possible methods for storing the digits in the number 1,234,567 in an array are diagrammed below.

1	2	3	4	5	6	7		. . .		
7	6	5	4	3	2	1		. . .		
		. . .		1	2	3	4	5	6	7

Each BigInt has a rightmost or *least significant digit* (LSD) and a leftmost or *most significant digit* (MSD). In 1,234,567 the LSD is 7 and the MSD is 1. Printing requires accessing digits from the MSD to the LSD. Numbers are also entered by typing the MSD first, so the LSD of an entered BigInt is stored last. Arithmetic operations, however, usually require accessing the LSD first. For example, the typical method for adding integers, using the add-and-carry approach diagrammed below, accesses the LSD first and stores the MSD of the result last.

$$\begin{array}{cccc} & 1^{1} & 6^{1} & 3 \\ + & & 5 & 8 \\ \hline & 2 & 2 & 1 \end{array}$$

The choice of how digits are stored in an array has a big effect on how difficult it will be to implement the BigInt operations. A poor design decision can be hard to undo, so careful thought is needed at this point. No matter how we store the digits we will try to separate the implementations of BigInt functions from the precise way in which digits are stored. We will strive, for example, to design a class so that switching the order in which digits are stored in an array affects only a few BigInt functions, and none of the functions outlined above as the core BigInt functions.

We do this by separating the physical layout of the digits from the logical way the digits are used. We'll number each digit in a BigInt starting with zero as the number of the least significant digit. For the four digit number 5,679 the 9 is the 0th digit. We can also say that the 9 has index 0. The 7 has index 1, the 6 index 2, and the 5 has index 3. In general, the

most significant digit of a BigInt with n digits has index (n – 1). To achieve the separation of logical and physical layout we'll use three private member functions to access digits. This means that BigInt functions that access digits will not access an array directly, but will access digits only through the private digit-manipulating member functions. With this approach, only the private functions will need to be reimplemented if the physical layout of digits is changed. For example, if we change the implementation to use built-in arrays rather than the apvector class only the private functions will need to be rewritten. Similarly (AB only) if we use a linked list the changes are isolated in the private functions. Specifications for the private functions follow.

```
int BigInt::NumDigits() const
// postcondition: returns # digits in BigInt

int BigInt::GetDigit(int k) const
// precondition: 0 <= k < NumDigits()
// postcondition: returns k-th digit
//                (0 if precondition is false)
//                Note: 0th digit is the least significant digit

void BigInt::ChangeDigit(int k, int value)
// precondition: 0 <= k < NumDigits()
// postcondition: k-th digit changed to value
//                Note: 0th digit is the least significant digit

void BigInt::AddSigDigit(int value)
// postcondition:  value added to BigInt as most significant digit
//                Note: 0th digit is the least significant digit
```

As an example of how these functions might be used, the code below shows how the digits of a BigInt can be printed from most to least significant. This might be a first start at the implementation of the function BigInt::Print().

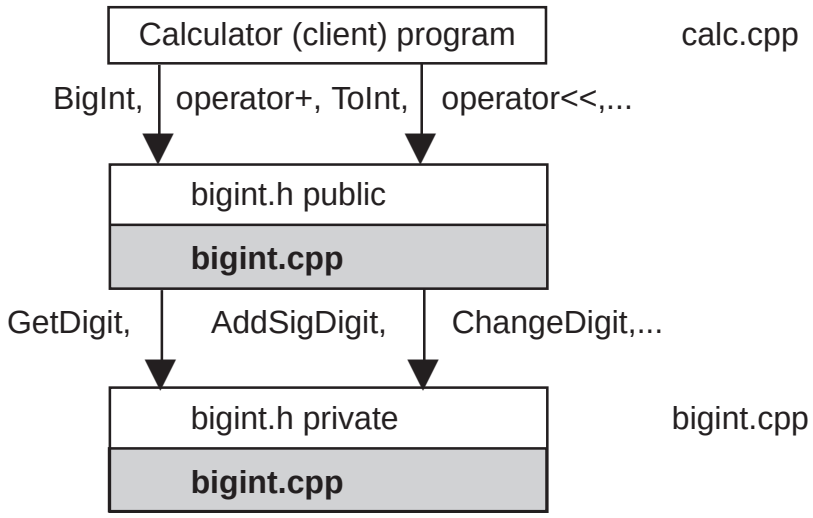
```
void BigInt::Print(ostream & out)
{
     int len = NumDigits();
     int k;
     for(k=len-1; k >= 0; k--)
     {
          cout << GetDigit(k);
     }
}
```

The final BigInt class will use a char vector to store individual digits, and digits will be stored as shown in the second of the diagrams above, i.e., with the least significant digit stored in the first array element (index 0). Note that this makes the physical index correspond to the logical index. For example, the least significant digit has physical index zero in the vector and logical index zero according to the scheme discussed above. However, it's not hard to change the order in which digits are stored. Since the changes are isolated in the private member functions GetDigit, ChangeDigit, and AddSigDigit most of the BigInt code won't need to be changed if the physical layout of digits is changed. The relevant parts of the private section of the header file bigint.h are reproduced here. Additional private helper functions are shown in Appendix B where the complete header file is shown.

```
private:
// private state/instance variables

enum Sign {positive,negative};
Sign mySign;                    // is number positive or negative
apvector<char> myDigits;        // stores all digits in number
int myNumDigits;                // stores # of digits in number

// helper functions

int GetDigit(int k) const;          // return digit (or 0)
void AddSigDigit(int value);        // add new most sig digit
void ChangeDigit(int k, int value); // change digit to value
```

A new structure for the program

The use of three private member functions changes the structure of the BigInt class. The new structure is shown below. A client program can only access BigInt variables using the functions in the public section of bigint.h. The decision on how digits are physically stored in the vector myDigits is isolated in the private member functions.



Study Questions

1. Why is a char vector used to store digits rather than an int vector? How will a change in the kind of element stored in the vector affect the implementation of all BigInt member functions.
2. We have chosen an enum for storing the sign of a BigInt. Describe two alternatives or types other than an enum that can be used to represent the sign of a BigInt.[5]
3. Write the function GetDigit based on the description and declarations in this section. What kinds of error should you worry about?
4. Why will it be difficult to write the non-member functions operator == and operator < given the current method for accessing digits using GetDigit? Write the function operator == for positive BigInt values assuming that NumDigits and GetDigit are public member functions.
5. (optional) Why is the apvector class a much better choice than the built-in array type given the specification of the BigInt class?

[5] The AP Computer Science C++ subset does not require the study of enums.

Implementation of the Large Integer Package

Building the Scaffolding: Fundamental and I/O Functions

We now turn to the implementation of the type BigInt. Rather than code all of the subprograms at once, we will implement them a few at a time, thoroughly test them as we go, and then move on to others. This way, when an error is discovered, the small amount of untested code is the most likely source of the error.

We begin with functions for constructing, reading, and writing BigInt values. When these operations have been implemented, we will have a reliable way of entering and printing values so that we can test other functions. We'll also be attentive to potential problems in our initial design of BigInt. Coding at this early stage often reveals ambiguities and problems in the specification and design decisions. We'll implement constructors first, then the output functions Print() and operator <<, then the function to read BigInt values: operator >>.

The constructors must initialize all private data fields of a BigInt. The postcondition of the default or parameterless constructor states that a BigInt is initialized to zero. This leads to the following code.

```
BigInt::BigInt()
// postcondition: bigint initialized to 0
   : mySign(positive),
     myDigits(1,'0'),
     myNumDigits(1)
{
          // all fields initialized in initializer list
}
```

The apvector field myDigits is initialized to contain one digit so that the BigInt represents zero. Initializer lists are the preferred method for constructing private data and setting initial values, although it is possible to initialize all fields in the body of the constructor as shown below.

```
BigInt::BigInt()
// postcondition: bigint initialized to 0
{
     mySign = positive;
     myDigits.resize(1);
     myDigits[0] = '0';
     myNumDigits = 1;
}
```

Initializer lists are preferred when private data must be constructed as well as being given initial values. In some cases, there is no alternative to initializing class data in an initializer list. For example, if the apvector class had no default (parameterless) constructor, the private data field myDigits would need to be constructed with an initial size in an initializer list. In the example above when an initializer list is not used, myDigits is first constructed to contain zero elements, then resized to contain one element. In some situations this is very inefficient since data will be constructed, then given an initial value in a separate statement, unless initializer lists are used. In the example above, more code is required when initializer lists are not used.

There are two other constructors that serve as conversion functions from integers and strings to BigInts. These are not shown here but are shown in Appendix C. Both constructors peel digits off one at a time and store them in a BigInt using the private member function AddSigDigit(). The AddSigDigit() function is responsible for resizing the vector myDigits when necessary.

Reading BigInt values will be facilitated by the constructor that converts a string[6] to a BigInt. Since strings can be read from any input stream, we can read a string and convert the string to a BigInt value. This approach is shown below for operator >>. The input stream must be returned from the function to facilitate chaining input operations as follows.

```
cin >> x >> y >> z;
```

Using string input for BigInt input means that it will not be possible to read a BigInt value unless it is followed by whitespace. This is a small price to pay for the ease with which we've implemented input for BigInt values.

```
istream & operator >> (istream & in, BigInt & big)
// postcondition: big extracted from in, must be whitespace
//                delimited
{
  apstring s;
  in >> s;
  big = BigInt(s);
  return in;
}
```

Input comes from the istream parameter in so that it will be possible to read from any stream rather than being limited to reading from the keyboard.

It's difficult to test the code for the constructors and for input without the function Print to display BigInt values. Nevertheless, we should think about testing the code by considering potential problems. For example, we must deal with the possibility of a leading '+' or '−' sign. We must also decide how to deal with non-digit characters. For example, what action should we take if the user enters 123af45? Thinking about these things may point out conceptual errors in the code. However, until we have a way of printing BigInt values we will not be able to test the functions thoroughly. The code in Appendix C shows that malformed BigInt values will cause a program to abort execution.

[6] Strings are implemented using the class apstring.

We now proceed to the output functions Print() and operator <<. Since the specifications for BigInt functions include ToString() which converts a BigInt to a string, we'll use this to generate BigInt output. This makes the output functions straightforward to implement. As shown below, digits are converted to characters and concatenated to a string which is returned; each digit of a BigInt value is accessed using the function GetDigit().

```
apstring BigInt::ToString() const
// postcondition: returns apstring equivalent of BigInt
{
   int k;
   int len = NumDigits();
   apstring s = "";

   if (IsNegative())
   {
     s = '-';
   }
   for(k=len-1; k >= 0; k--)
   {
     s += char('0' + GetDigit(k));
   }
   return s;
}
void BigInt::Print(ostream & os) const
// postcondition: BigInt inserted onto stream os
{
  os << ToString();
}

ostream & operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out
{
  big.Print(out);
  return out;
}
```

Testing the Class

To test the functions we've written we'll write a program that includes the header file bigint.h and which links in the implementation file bigint.cpp. On many systems, including Symantec C++, Metrowerks Codewarrior, Visual C++, and Borland C++, testing a multifile program requires using a project[7]. On some systems it is possible to include implementations (i.e., .cpp files) in a library that is linked automatically with client test programs. We'll assume, however, that a project is used. To test the BigInt implementation we include bigint.cpp and apstring.cpp in the project. On some systems apvector.cpp must also be in the project. Programming environments are smart enough not to recompile the string and vector implementations when only the client test program changes. Therefore, we don't expect our test program to take long to compile, although it is common for many system files (e.g., iostream.h) to require lengthy compilation.

We must also provide documentation in the header file bigint.h so that a programmer can use our class effectively without access to the implementation. The documentation in bigint.h should contain enough information for the class to be used, *and no more*. We don't want to mention implementation details that might change. Unfortunately, the private section of classes is shown in C++ header files so users can see some information about the implementation, although client programs cannot access private data or call private functions. Although there are methods for concealing the private section from client programs, these require using either pointers or inheritance. Therefore, we will be content with revealing some details of the BigInt implementation in the private section of the class declaration. Appendix B shows the header file bigint.h.

The program testio.cpp simply reads and prints BigInt values entered by the user. Since input and output use the string constructor and conversion function ToString(), these functions are tested using testio.cpp. The test program is in Appendix E.

[7] Information on how to use projects is accessible via the case study web page.

We must carefully choose test data for even a small program like testio.cpp. In testing any program, we wish to test a few typical values, all edge case or boundary values, and a variety of error values to ensure that the code is robust. For testing this program, we tested various lengths of integers, including lengths of one digit, a few digits, and hundreds of digits. We also tested numbers with leading '−' and '+' characters, and embedded '−' and '+' characters. Finally, we entered sequences containing letters at the beginning, end, and middle to see how our code handles error values.

One series of tests generated incorrect results. If a number was entered with leading zeroes (e.g., 00345) the program stored and printed the leading zeroes. Entering the sequence 0000 generated the sequence 0000. We did not anticipate this case in specifying and developing the code. We will deal with this problem in the next section.

Study Questions

1. List all the values you would use to test the I/O functions and the string conversion functions thoroughly. Give an explanation for each data set describing its importance. How could you modify testio.cpp to test the constructor that has an int parameter?
2. The stream member function width() can be used to specify a fieldwidth when generating output. Do you expect it to work correctly with BigInt values? Modify testio.cpp to see if width() works as intended.
3. A reference to an uninitialized variable can generate an error in some C++ environments. The default BigInt constructors could be modified to detect that a variable has not been given a value. Instead of initializing a BigInt variable to zero, the constructor could leave the BigInt in an uninitialized state. With this knowledge, we can generate an error status if an attempt is made to examine the value of such a variable. Describe in some detail how this might be implemented.
4. What modifications would need to be made for these functions to represent and manipulate numbers in base 3? In base 2? Show how a constant could be used to vary the base of the number system in use.

Refining Our Implementation

The same integer can be represented in many different ways by adding leading zeroes. For example, 012 and 00012 both represent the number twelve. This poses a problem since we would prefer that variables holding the same value have the same representation. Two approaches to resolving this seem reasonable:

1. Allow this ambiguity, but let the various functions resolve it as necessary. For example, Print() can avoid printing leading zeroes.
2. Represent all BigInt values without leading zeroes. Each function that might produce leading zeroes (e.g., operator >>) will need to trim them off.

We chose the second method since it more closely resembles how we think of integers (i.e., leading zeroes are not written or stored).

The input function (actually, the constructor that converts a string to a BigInt) will need to trim leading zeroes before returning. Since this is likely to be an operation needed by other functions, but not by client programs, we'll implement a private function to trim leading zeroes. Because we're not sure what other conditions might arise that require conversion to a common form we'll call the function Normalize rather than something seemingly more appropriate like TrimZeroes. A value is normalized when it is in a standard format. For example, in a class for representing fractions (rational numbers) we might convert all fractions to lowest terms, e.g., Fraction a(6,9) would represent two-thirds when normalized.

```
void BigInt::Normalize()
// postcondition: all leading zeroes removed
```

Finally, one special case remains. Suppose that a BigInt consists entirely of zeroes, i.e., it represents the number zero. Should the final zero be trimmed or should it remain? We'll store one digit '0' since it more closely matches our intuitive idea of a representation for zero, i.e., a single digit 0. We must also consider the idea of a "negative zero." If you enter −00000, should this be treated differently than +0000? Since one of the design decisions we made is to have only one way of representing zero, we'll use a positive zero only.

The header for Normalize() does not appear in the public section of the class declaration since client programs will not need to call it. The Normalize function is shown in Appendix C; it is called from the BigInt string constructor to trim zeroes that may be part of the string being converted to a BigInt. As we develop other functions, Normalize will be called whenever it is possible that a number contains leading zeroes.

Study Questions

1. Suppose that we decide not to eliminate leading zeroes until necessary. Which of the BigInt functions might generate leading zeroes? Which might need to eliminate them from parameters if they were present?
2. What happens to the size of the private variable myDigits if the user enters 100 zeroes? Is this desirable?
3. What arithmetic operations can introduce leading zeroes?
4. Describe how a Normalize function for rational numbers (fractions) might convert to lowest terms (so that numerator and denominator have no common factors).

Comparison Operations

After verifying that our code can reliably read and write BigInt values, we may proceed to develop the other BigInt subprograms. We choose to implement the comparison operations first because they are reasonably simple and may be useful in creating more complex functions (e.g., subtraction, multiplication). We'll check for equality first, then develop code for BigInt inequalities.

As we noted earlier, checking for equality requires accessing individual digits of BigInt values. Since digits are accessed using the private member function GetDigit(), the function that checks for equality will need to be able to call GetDigit(). Ideally we would like to overload operator == so that we can compare BigInt values using code like if (a == b). However, we cannot make operator == a member function.[8] We could make operator == a

[8] If we implement operator == as a member function it will have one parameter since the statement if(a.operator == (3)) is another way to write (a == 3) when == is a member function. It is not possible to write if(3.operator == (b)) instead of if(3 == b) since 3 is an int and has no member functions. Since we would like to write both if(3 == b) and if(b== 3) we make operator == a non-member (or free) function. Then if(3== b) is simply a nicer way of writing if(operator == (3,b)) which works because the 3 can be converted to a BigInt since there is a BigInt constructor with an int parameter.

friend function. Friend functions and classes can access private data. In general, it's a good idea to avoid using friend functions and classes since granting access to private information doesn't conform to the goals of information hiding. Instead of using friend functions, we'll write a public helper member function BigInt::Equal() to make it possible to overload operator == just as the member function BigInt::Print() made it possible to overload operator << for BigInt values.

Two integers are equal if and only if they have the same sign, the same number of digits, and the same sequence of digits. The code in the function Equal below illustrates this.

```
bool BigInt::Equal(const BigInt & rhs) const
// postcondition: returns true if self == rhs, else returns false
{

   if (NumDigits() != rhs.NumDigits() || IsNegative() !=
       rhs.IsNegative())
   {
     return false;
   }
   // assert: same sign, same number of digits;

   int k;
   int len = NumDigits();
   for(k=0; k < len; k++)
   {
     if (GetDigit(k) != rhs.GetDigit(k)) return false;
   }
   return true;
}

bool operator == (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs == rhs, else returns false
{
  return lhs.Equal(rhs);
}
```

The relational operators < and > require more thought. In particular, we must take more care with signs. For example, while 34 > 23, the code must recognize that $-34 < -23$. In writing the code, we take the same general approach as we used with BigInt::Equal(): we deal with the signs and lengths first, and then with the sequence of digits. To implement operator <

we'll implement a helper function BigInt::LessThan(). We'll see that both operator < and operator > can then be implemented easily. In the body of LessThan we first check to see if the sign of BigInt rhs is different than the sign of the BigInt for which the member function is called (self in the comments below). If the signs are different, self is less than rhs only if it is negative and rhs is positive.

```
bool BigInt::LessThan(const BigInt & rhs) const
// postcondition: return true if self < rhs, else returns false
{
  // if signs aren't equal, self < rhs only if self is negative

  if (IsNegative() != rhs.IsNegative())
  {
    return IsNegative();
  }
```

Next we take care of values with different numbers of digits. Since both numbers have the same sign, individual digits don't need to be examined to determine which is larger. If both are positive, then self is smaller if it has fewer digits than rhs (223 < 1234). Similarly, if both are negative, then self is less than rhs if self has more digits ($-1234 < -123$).

```
// if # digits aren't the same must check # digits and sign

  if (NumDigits() != rhs.NumDigits())
  {
     return (NumDigits() < rhs.NumDigits() && IsPositive()) ||
            (NumDigits() > rhs.NumDigits() && IsNegative());
  }
```

Finally, when both the signs and the lengths are the same, we must traverse the digits from left to right, searching for a digit that differs. The signs of self and rhs and the value of the differing digits can be used to determine if self is smaller than rhs. The final code appears in Appendix C. Implementing operator < is straightforward.

```
bool operator < (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs < rhs, else returns false
{
     return lhs.LessThan(rhs);
}
```

We now turn to implementing operator >; there are two alternatives. We can write code similar to the code developed for BigInt::LessThan(), but reflecting the difference between < and >. Alternatively, we can reuse the code for operator < and the mathematical equivalence that lhs < rhs if and only if rhs > lhs. This second alternative requires much less code.

```
bool operator > (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs > rhs, else returns false
{
     return rhs < lhs;
}
```

We developed a small program called testcomp.cpp to test ==, < and >; it is included in Appendix E. We also implemented operators <= and >= since these can be implemented easily in terms of the operators already implemented. Our test data includes:

1. pairs of values containing all possible combinations of signs:

   ```
   123, 123
   −123, −123
   −123, 123
   123, −123
   ```

2. combinations involving zero:

   ```
   −123, 0
   0, −123
   123, 0
   0, 123
   0, 0
   ```

3. comparing values with different numbers of digits.

Study Questions

1. Write operator != and operator <= Should these functions be implemented in terms of existing comparison functions?
2. Can BigInt::Equal() be written by traversing the digits from left-to-right rather than from right-to-left? What about the other comparison functions?
3. Some programmers prefer to write a single comparison function that returns −1 if the first parameter is less than the second, 0 if they are equal, and +1 if the first is greater than the second. Write a function Compare that implements this idea.
4. Discuss the advantages and disadvantages of the method described in the previous question as opposed to the approach taken in this case study.

Implementing Addition

With the implementation of comparison functions and I/O functions complete, we are now ready to attack a more difficult function. We choose to implement addition first since it is the simplest of the arithmetic algorithms and may be useful in writing subtraction and multiplication. For example, we can think of 3 x 5 as 3 + 3 + 3 + 3 + 3. It's possible to write multiplication in terms of addition.

We'll start the development of the addition algorithm for numbers with the same sign by examining the standard pencil-and-paper algorithm diagrammed below and writing pseudocode based on this algorithm.

	4	1^1	6^1	3
+			5	8
	4	2	2	1

Each step in computing the sum of two digits in a digit sequence may generate a carry for the next step and may use a carry from the previous step. This leads to the pseudocode below.

```
carry = 0;
for each position from right to left of the numbers do
   sum = sum of digits from numbers plus carry
   place sum % 10 as new most significant digit of answer
   carry = sum / 10
if carry != 0
   place carry as new most significant digit of answer
```

We then translate each portion of this pseudocode into C++. The loop begins at the rightmost digit of each number and proceeds to the leftmost digit of the larger number. We will use this idea to implement operator +=. In general, it is easier and preferable to implement such operators first and to then implement corresponding operators like operator + in terms of operator +=. For example:

```
BigInt operator + (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns a bigint whose value is lhs + rhs
{
     BigInt result(lhs);
     result += rhs;
     return result;
}
```

The other arithmetic operators are implemented as easily using the corresponding arithmetic assignment operator. We now turn to implementing addition using operator +=. To access all digits of both operands starting from the least significant digit (the 0-th digit) we'll use the code below.

```
const BigInt & BigInt::operator +=(const BigInt & rhs)
// postcondition: returns value of bigint + rhs after addition
{
     int sum;
     int carry = 0;
```

```
int k;
int len = NumDigits();       // length of larger addend
if (len < rhs.NumDigits())
{
     len = rhs.NumDigits();
}
for(k=0; k < len; k++)
// process digits to form sum
```

In accessing the digits of each number, we must handle the case where one of the numbers has fewer digits than the other. For example, consider adding 1,234,567 and 999. At first, it might seem necessary to loop only three times — as many times as there are digits in the smaller number 999. Looping over digits seven times — as many digits as there are in 1,234,567 — might cause problems because there are not seven digits in both numbers. Although it is possible to write the code this way, the code will be complicated by special cases to differentiate which of the addends is smaller: the first or second. Fortunately we can hide this situation in the private function GetDigit() so that if we request a digit that doesn't exist (such as the fifth digit of 123) the function GetDigit() will return zero. This greatly simplifies the code and leads to the loop below. We use a constant BASE that is assigned the value 10 to facilitate using other number bases. It may be difficult to implement arithmetic in base 2 or base 16, but using a constant BASE, initially set to 10, will help if we decide to implement arithmetic in other bases.

```
for(k=0; k < len; k++)
{
     sum = GetDigit(k) + rhs.GetDigit(k) + carry;
     carry = sum / BASE;
     sum = sum % BASE;

     if (k < myNumDigits)
     {
          ChangeDigit(k,sum);
     }
     else
     {
          AddSigDigit(sum);
     }
}
```

It is also possible for the carry to be 1 after adding all the digits as when adding 357 + 662. We must check for this after the loop.

```
if (carry != 0)
{
     AddSigDigit(carry);
}
```

At this point in the implementation, it makes sense to test operator += and operator + even though they only handle numbers with the same sign. If there are any problems, we can fix them before continuing with addition of numbers whose signs are different. We then consider the signs of the numbers being added. If the signs of two numbers are both positive or both negative, then the usual pencil-and-paper addition algorithm will be sufficient to obtain the sum. On the other hand, if the numbers have different signs, then we need a subtraction rather than an addition. Since we know that we will also be developing a subtraction function, we can use it to solve the problem of adding two numbers whose signs are different. This leads to the initial code segment below for adding rhs[9]. (This code does not work.)

```
if (IsPositive() != rhs.IsPositive()) // signs not the same,
                                      // subtract
{
     if (rhs.IsPositive())   // change sign of rhs
         rhs.mySign = negative;
     else
         rhs.mySign = positive;

     *this -= rhs;       // x + y == x - (-y)
     return *this;
}
// signs are the same
```

[9] The expression *this refers to the object to which the BigInt rhs is added. We've referred to this as self in the code and in the case study. For example, in evaluating the statement x +=y; y is passed as the parameter rhs and x is referred to within the member function operator += by *this. Students in the A course are not responsible for understanding *this.

Here we make use of the algebraic relationship that A + B = A − (−B) to reduce the addition of terms with different signs to a subtraction of values with the same sign. Unfortunately the method above will not work because the parameter rhs is passed as a const reference parameter and the constness prevents our code from modifying rhs. One alternative is to make a copy of rhs which can then be altered. Another alternative is to assume that multiplication is implemented and write the code below.

```
if (IsPositive() != rhs.IsPositive()) // signs not the same,
                                          subtract
{
     *this -= (-1 * rhs);       // x + y == x - (-y)
     return *this;
}
// signs are the same
```

An alternative approach is to give operator += the ability to handle addition of any two numbers, regardless of signs. This would make the code for operator += more complex, but would lead to extremely simple code for operator −=. We chose instead to separate the addition and subtraction algorithms into two different functions and have each call the other if the values to be added (or subtracted) have different signs. The code for operator += appears in Appendix C.

Study Questions

1. Write the code for operator –= on the assumption that operator += can handle any combination of signs.

2. Complete the code fragment below to use subtraction when signs are different by creating a copy of rhs whose sign can be changed.

```
if (IsPositive() != rhs.IsPositive()) // signs not the
                                      // same, subtract
 {
   BigInt copy(rhs);
   // change sign of copy, subtract, return result
```

3. (AB Only) In algorithm analysis, one usually uses the "unit cost" assumption with regard to arithmetic operations: each such operation is O(1). What is the complexity (using big-Oh) for adding BigInt values of N digits? Calculating the Nth Fibonacci number can be done in O(N) time when integers are used. What would the big-Oh complexity be if BigInt values are used?

Testing Addition

To test operator += , we'll first create a program that reads two values and prints their sum. The program testadd.cpp is given below and in Appendix E. We could also modify the calculator program so that it used BigInt values to test operator +=, but at this point we'll use testadd.cpp.

```
#include <iostream.h>
#include "bigint.h"

int main()
{
     BigInt a,b;
     while (true)
     {
          cout << "enter two bigint values: ";
          cin >> a >> b;
          cout << "terms:" << endl << a << endl;
          cout << b << endl;
          cout << "sum = " << a + b << endl;
     }

     return 0;
}
```

This program tests operator + directly and operator += indirectly since this latter operator is called from the first as we've seen. In order to run the program, we must implement operator −= since operator += calls it. Rather than implement it fully, however, we could create a stub function that does nothing but print a message. This would let us concentrate on operator += and worry about operator −= after we have found any bugs that might be in operator +=. Addition of all BigInt values with different signs cannot be tested until operator −= is fully implemented.

Again, we must choose test data carefully. Some categories of test data are given below.

1. Values for a and b that require carrying.
2. Values of 0 for either or both of a and b.
3. Values that require growing the vector that stores digits.

The Subtraction Algorithm

We now turn to replacing the stub implementation of operator −= with working code. We'll use the same approach we used in operator += to handle different signs, this time relying on the algebraic relationship below.

```
x - y = x + (-y)
```

If the signs of the numbers x and y differ, we can rewrite the problem using addition. For example, to perform the subtraction (−3) − (4), we perform the addition (−3) + (−4). You may notice that operator += calls operator −= to handle the case of different signs, and that operator −= calls operator += to handle different signs. Since we are implementing a class, prototypes for both operators appear in the file bigint.h.

The organization of the code described above allows us to proceed with the subtraction algorithm on the assumption that the signs of the parameters are the same. In the pencil-and-paper algorithm, we normally write the longer number "on top." In other words, to calculate 45 − 126, most people rewrite this as 126 − 45, perform the subtraction (obtaining

81), and then change the sign of the result. We can simplify our algorithm by performing the same steps[10]. Care must be taken to handle negatives properly (e.g., $(-45) - (-126)$). The following code takes care of these cases:

```
// signs are the same, check which number is larger
// and switch to get larger number "on top" if necessary
// since sign can change when subtracting
// examples: 7 - 3 no sign change,     3 - 7 sign changes
//          -7 - (-3) no sign change, -3 -(-7) sign changes
if (IsPositive() && (*this) < rhs ||
    IsNegative() && (*this) > rhs)
{
  *this = rhs - *this;
  if (IsPositive()) mySign = negative;
  else              mySign = positive;  // toggle sign
  return *this;
}
// same sign and larger number on top
```

We can now proceed to the actual algorithm using the pseudocode below.

```
for each digit from rightmost digit to leftmost digit of self
  if my digit is greater than the digit from rhs
    then subtract the digits and change my value
    else regroup from the next place, subtract rhs's digit,
      and change my value
```

In the pencil-and-paper algorithm for subtraction, regrouping or borrowing is immediately propagated as far left as necessary. For example, if 5 is subtracted from 1003, the subtraction of the units digit results in a regrouping operation carried through to the thousands place:

```
   0  9  9  13
   1  Ø  Ø  Ø
-           5
-------------
      9  9  8
```

[10] The statement *this = rhs − *this makes an indirect recursive call of operator −=. For example, consider the statement big −= 7 where BigInt b has the value 3. This can also be written as big.operator −=(7). In this case big is *this in the body of operator −=. Because (*this) < rhs since 3 < 7, the statement *this = rhs − *this will be executed. This results in execution of result −= 3 where result has the value 7 (see the code in the Appendix C for operator −). Infinite recursion isn't possible because if a < b then it is not the case that b < a.

A simpler algorithm results if we handle regrouping much like carrying in addition. If regrouping is required in subtraction of some digit, the process is passed on to the next digit. When that digit is subtracted, a further regrouping may be generated, and so on. We use this method in our code.

To ensure that there are no leading zeroes, we must also call Normalize().

The code for operator += and operator −= functions is given in Appendix C. Since we have already modified the calculator program for both addition and subtraction of BigInt values, we'll use that as a test program rather than designing a different testing program.

Study Questions

1. Describe test data for the calculator program for testing addition and subtraction. Give an explanation for each data set describing its importance.
2. Write functions (for prefix operators) operator ++ and operator −− similar to the corresponding integer functions. Call the addition and subtraction functions as needed.
3. Write functions operator ++ and operator −− directly (i.e., without using your addition and subtraction functions). What is gained or lost in terms of time efficiency?

Multiplication

The pencil-and-paper algorithm for multiplication is diagrammed below.

			2	6	3	4
		×		4	7	2
			5	2	6	8
	1	8	4	3	8	
1	0	5	3	6		
1	2	4	3	2	4	8

This method involves repeatedly multiplying a number by one digit of the other number. Depending upon the place of the digit, the resulting product is multiplied by a power of 10 and then added to the result. In pseudo-code:

```
for each digit from rightmost to leftmost digit of the bottom
  number do
    multiply that digit by the top number
    multiply this product by 10^(p-1) (where p is 1, 2, 3, ...)
    add this into the accumulated result
```

We can use operator += for the addition. To facilitate the multiplication by 10 and the multiplication of "that digit by the top number" we'll implement a separate operator for multiplying a BigInt by a single digit. We'll use this function in implementing the more general multiplication by a BigInt value. We'll also modify the pseudocode above to avoid the calculation of 10^{p-1} that would require a loop. Rather than calculate increasingly larger powers of 10, we'll simply multiply the top number by 10 each time through the loop over the digits of the bottom number. Thus to multiply 12345 by 678 (where the top number is 12345) we'll actually calculate as follows:

12345 x 8 + 123450 x 7 + 1234500 x 6 = 8369910

This method requires only one multiplication by 10 each time through the for loop whose pseudocode is given. Before completing operator *= we'll turn to the operator to multiply a BigInt by a single digit.

Multiplication by an int

To multiply by a single digit we'll use an overloaded version of operator *=. Note that the parameter is an int, not a BigInt.

```
const BigInt & BigInt::operator *=(int num)
// postcondition: returns num * value of BigInt after
//                multiplication
```

We must be careful with signs since multiplication can change the sign of a number as when a positive number is multiplied by a negative number. The implementation requires traversing digits from right to left while keeping track of any carry. Our first sketch of the algorithm looks like this:

```
len = NumDigits();
for(k=0; k < len; k++)   // once for each digit
{
     product = num * GetDigit(k) + carry;
     carry = product/BASE;
     ChangeDigit(k,product % BASE);
}
while (carry != 0)     // carry all digits
{
  AddSigDigit(carry % BASE);
  carry /= BASE;
}
```

This code is adapted from the addition algorithm; it differs in two ways.

1. Product is calculated differently than sum is calculated.
2. If carry is a multi-digit number, processing carry after the for loop requires another loop.

We are designing this function to handle multiplication of a BigInt by a single digit, but as part of a defensive programming strategy we should take steps to ensure that the function can handle multiplication of BigInt values by any int value. For example, if num is too large, the statement below could result in a value larger than INT_MAX.

```
product = num * GetDigit(k) + carry;
```

If num is near INT_MAX and the current digit is near 9, then the result will be larger than INT_MAX. On some systems this will generate an error, on others it may produce erroneous results. We could write a precondition specifying a limit on the values of parameter num that the function handles correctly. For example, the following precondition limits the values of num to single digits and the number 10 (when BASE is 10).

```
// precondition: 0 <= num <= BASE
```

Client programs would then be responsible for calling the function only with values that satisfy the precondition. Alternatively, we could make the single digit version of operator *= a private function so that it is not accessible to client programs but can be called only from member functions. Instead, we'll convert integer values that are not single digits to BigInt values.

Finally, we must deal with the signs of the numbers being multiplied. This is a much simpler problem for multiplication than it is for addition. If the signs of two numbers being multiplied are different, the result is negative. Otherwise, it is positive.

Code for operator *= for both BigInt and digits is given in Appendix C. To test that the functions work, the calculator program can be used. Care must be taken in testing all boundary or edge cases with operator *= for BigInt values. For example, we must verify that different combinations of signs lead to correct results and also that multiplication in which one or both of the numbers is 0 is handled correctly.

Study Questions

1. The multiplication algorithm multiplies the top number by 10 each time through the loop over the digits of the bottom number. An alternative method is to work from the leftmost digit down to the rightmost digit of the bottom number. For example, to multiply 456 by 123, we do the following:

Multiply 456 by the leftmost digit in 123:	1 * 456 = 456
Multiply the result by 10:	456 * 10 = 4560
Multiply 456 by the next digit of 123 and add to previous result:	4560 + 2 * 456 = 5472
Multiply the result by 10:	5472 * 10 = 54720
Multiply 456 by the next digit of 123 and add to previous result:	54720 + 3 * 456 = 56088

   Modify operator *= (const BigInt & big) to use this approach.

2. (AB Only) Find a big-Oh expression for the runtime of operator *= (const BigInt & big) when applied to two N-digit numbers. Find an expression for multiplying one N-digit BigInt by a digit using operator *= (int num).

3. It's possible to write functions to handle adding BigInt and int values specially just as we developed two operators for multiplication: one for BigInt values and one for int values. We can implement addition and subtraction operators in two ways:

   (a) by converting the integer to a BigInt and calling the appropriate BigInt operator; or

   (b) by writing special code to implement the operation via low-level reference to the internals of BigInt.

   Discuss the advantages and disadvantages of (a) vs (b).

4. Develop a precondition for BigInt::operator *= (int num) that describes the range of values for num that do not result in overflow.

Aliasing: A New Problem Arises

The test programs revealed no problems with the implementation of any of the BigInt functions, nor did the calculator program show any problems. Because we use overloaded arithmetic and I/O operators for BigInt values, modifying the calculator program was straightforward and required changing only two definitions for the variables current and accumulator. However, we then implemented the function below for raising BigInt values to a power and found some surprising results. This function is from a client program, it's not part of the class BigInt.

```
BigInt power(const BigInt & a, int n)
// precondition: 0 <= n
// postcondition: returns a^n, a raised to the n-th power
{
     if (n == 0) return 1;
     if (n == 1) return a;

     BigInt semi = power(a,n/2);
     semi *= semi;                // this is semi^2
     if (n % 2 == 0) return semi;
     else            return semi * a;
}
```

We tested this function with int values and it produced correct results. However, when tested with BigInt values as shown, some surprising results occurred. For example, the function produces the following results

for a BigInt raised to an int power (the correct result is shown in parentheses).

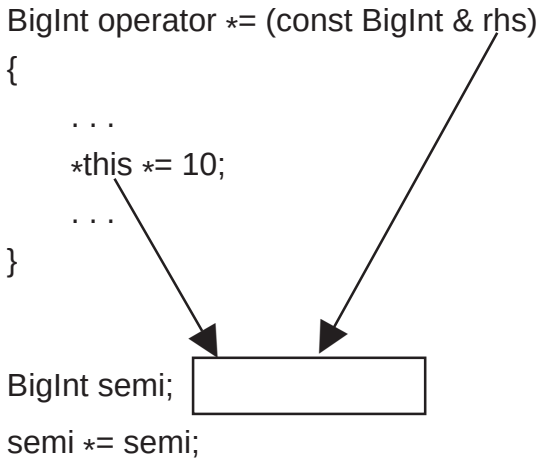
$7^2 = 49$ (49)
$12^2 = 164$ (144)
$17^2 = 1309$ (289)
$55^4 = 16803875$ (9150625)

On some computers different results may appear, but the results won't be correct. It's difficult to track down this bug, but since we know that the first test programs generated correct results, there is reason to think that the arithmetic operators are basically correct and concentrate on the different way the operators are called from the function power(). The multiplication of semi *= semi is the cause of the problem.

When a reference parameter is used, the computer passes only a reference to the location of the actual parameter (or argument)[11] rather than the value of the argument. This mechanism allows the value of the argument to be changed by the function. When a call is made to a function with the same argument passed to two or more reference parameters, errors can occur. In this case, the call

```
semi *= semi;
```

means that in the body of operator *= the values *this and rhs will refer to the same memory location, i.e., semi. The diagram below shows how this can happen.



[11]Argument is a synonym for actual parameter — what is passed to a function. The term *formal parameter* is used for the parameter within the function as opposed to what is passed. Rather than use the adjectives formal and actual to modifier parameter, many people use argument for what is passed and parameter for what is used within the function.

We will be unable to modify the value of *this (e.g., by multiplying by 10), without also modifying rhs since they share the same vector of digits. Different test programs may yield even more bizarre results.

This problem is called *aliasing*, a word used whenever there are two or more ways to refer to a single memory location. We need a middle road between using reference parameters and making no local copies which leads to the errors we've just encountered, and using value parameters which make local copies, but lead to wasted memory and time. We choose to use reference parameters, and to make local copies explicitly when necessary.

In operator *= (const BigInt & big), for example, we will need to make a local copy of *this (the number on the left of *= in a call such as x *= y) and use the copy. Since we'll use a copy, manipulations of the copy will not inadvertently change the value of rhs (and vice versa.) We'll also need to check the problem of subtracting a value from itself, i.e., x -= x. We modify addition and subtraction operators to handle this problem without making a copy, but by converting x += x into x *= 2 and converting x -= x into x = 0. For more general multiplication we will make a copy of *this and use the copy rather than *this.

Aliasing cannot occur in the I/O functions since they have only one BigInt parameter.

Fixing operator *=

To eliminate the aliasing problem in operator *= we make a copy as shown.

```
const BigInt & BigInt::operator *=(const BigInt & rhs)
// postcondition: returns value of bigint * rhs after
//                multiplication
{
   // uses standard "grade school method" for multiplying

   if (IsNegative() != rhs.IsNegative())
   {
       mySign = negative;
   }
   else
   {
       mySign = positive;
   }
```

```
    BigInt self(*this);             // copy of self
    BigInt sum(0);                  // to accumulate sum
    int k;
    int len = rhs.NumDigits();      // # digits in multiplier
    for(k=0; k < len; k++)
    {
    sum += self * rhs.GetDigit(k);  // k-th digit * self
    self *= 10;                     // add a zero
    }
    *this = sum;
    return *this;
}
```

We then manipulate the self local variable rather than the implicit parameter *this (recall that *this is the value on the left, e.g., x in x *= y.) We can construct a BigInt from another BigInt because the compiler implements this constructor for us. The constructor works correctly because each private data field is either a built-in type or has a constructor defined for it as with the apvector class. Complete code for all the arithmetic operators is given in Appendix C. Code for the BigInt class that does not fix aliasing is given in Appendix D.

Conversion Functions

The constructor with an int parameter and the constructor with a string parameter are effectively conversion functions from ints and strings, respectively, to a corresponding BigInt value. It would be useful to convert in the other direction: from a BigInt to an int, double, or string. The function ToString() which we've already discussed is such a function. We'll discuss conversion functions ToInt() and ToDouble() in this section.

ToInt(): This function converts a BigInt to an int, its prototype follows.

```
int BigInt::ToInt() const
// precondition: INT_MIN <= self <= INT_MAX
// postcondition: returns int equivalent of self
```

To implement this conversion function we'll use a variation of an algorithm called Horner's rule. The digits of the BigInt will be processed from left to right. At each step, the integer result will hold the value of all the BigInt digits that have been processed so far. For example, to convert

456 requires three steps: process the 4, process the 5, and process the 6. After each step the integer result has the values 4, 45, and 456 respectively. Before any BigInt digits have been processed, the int result should be zero. This leads to the code below.

```
int result = 0;
int k;

for(k=NumDigits()-1; k >= 0; k--)
{
  result = result * 10 + GetDigit(k);
}
```

We're now faced with a difficult design decision: what should happen if the BigInt being converted exceeds INT_MAX? Since the precondition for ToInt() states that the value being converted should be less than or equal to INT_MAX, we could decide not to deal with the problem at all. The postcondition would still be true whenever the precondition is satisfied. ToInt() might crash, return a value of 0, or simply never return if the precondition isn't satisfied. Instead we'll write code that checks for values that violate the precondition and convert them to the closest value. Alternatively, we could print an error message. We'll include the code below in ToInt().

```
if (INT_MAX < *this) return INT_MAX;
if (*this < INT_MIN) return INT_MIN;
```

ToDouble(): This code is identical to the ToInt() function except that it returns a double.

All of the BigInt functions, including the conversion functions, are shown in Appendix C. A brief testing program, testconv.cpp for the ToDouble() and ToInt() functions is included in Appendix E.

Study Questions

1. To avoid calling ToInt() with BigInt values that exceed INT_MAX, it's possible to convert the BigInt value to a double value and compare double values rather than comparing BigInt values. Write code implementing this approach.

2. Yet another approach to dealing with INT_MAX overflow requires changing the header of ToInt() to have a third parameter, bool overflow, which is set when overflow occurs. Write this version of ToInt() and discuss its advantages and disadvantages compared to the approach taken in this case study.

3. Write the body of a testing program for the constructor that converts an int to a BigInt and describe test data for it, giving an explanation for each data set and describing its importance.

4. Use the BigInt class to create a function that calculates values of large factorials.

```
BigInt Factorial(int n)
// Pre:  0 <= n
// Post: returns the value of N! = N * N-1 * N-2 *...*1
```

5. Modify the Factorial function so that n is a BigInt. Does it make sense to worry about finding the value of n! where INT_MAX < n?

6. Use the BigInt class to create a function that calculates large Fibonacci values.

```
BigInt Fibonacci(int n)
// pre:  1 <= n
// post: returns the Nth Fibonacci number, F_n,
//       where F_1 = 1, F_2 = 1, and F_n = F_n-1 + F_n-2
```

Appendix A: The Calculator

```
#include <iostream.h>
#include "apstring.h"

// This program implements a three function calculator
// as the first step to a test program for the BigInt class
//

enum  Optype { add, subtract, multiply, stop, illegal_op };

Optype StringToOp(const apstring & s);
Optype GetOperator();

int main()
{
   int accumulator = 0;        // accumulate in this object
   int current;                // value to be operated with
   Optype op = add;            // so first value is added to zero

   while (op != stop)
   {
      cout << "enter value: ";
      cin >> current;
      switch (op)
      {
        case add :
           accumulator += current;
           break;
        case subtract:
           accumulator -= current;
           break;
        case multiply:
           accumulator *= current;
           break;
        default:
           cerr << "error with operator" << endl;
      }
      cout << "——> " << accumulator << endl;
      op = GetOperator();
  }
  return 0;
}

Optype StringToOp(const apstring & s)
// precondition:  s is a valid operator
// postcondition: returns Optype equivalent, e.g., add for '+'
{
   // define array of strings to facilitate look up
   // built-in arrays are NOT part of AP C++ subset
   apstring OPSTRINGS[] = {'+', '-', '*', '=' };
   const int NUMOPS = 4;  // # valid operations
```

```
   int k;
   for(k=0;k < NUMOPS; k++)
   {
       if (s == OPSTRINGS[k]) return Optype(k);
   }
   return illegal_op;  // not found, return illegal_op
}

Optype GetOperator()
// postcondition: reads a whitespace delimited operator from cin
//                and returns the operator
{
   apstring s;
   Optype retval;
   do
   {
      cout << "enter + - * ( = to quit) ";
      cin >> s;
   } while ((retval = StringToOp(s)) == illegal_op);

   return retval;
}
```

Appendix B: The Header File bigint.h

```
#ifndef _BIGINT_H
#define _BIGINT_H

// author: Owen Astrachan
// 8/23/95, modified 7/5/96
//          modified 1/5/97
//
// implements an arbitrary precision integer class
//
// constructors:
//
// BigInt()         – default constructor, value of integers is 0
// BigInt(int n)    – initialize to value of n (C++ int)
// BigInt(const apstring & s) – initialize to value specified by s
//        it is an error if s is an invalid integer, e.g.,
//        "1234abc567". In this case the bigint value is garbage
//
//
// ***** arithmetic operators:
//
// all arithmetic operators +, -, * are overloaded both
// in form +=, -=, *= and as binary operators
//
// multiplication also overloaded for *= int
// e.g., BigInt a *= 3 (mostly to facilitate implementation)
//
// ***** logical operators:
//
// bool operator == (const BigInt & lhs, const BigInt & rhs)
// bool operator != (const BigInt & lhs, const BitInt & rhs)
// bool operator < (const BigInt & lhs, const BigInt & rhs)
// bool operator <= (const BigInt & lhs, const BigInt & rhs)
// bool operator > (const BigInt & lhs, const BigInt & rhs)
// bool operator >= (const BigInt & lhs, const BigInt & rhs)
//
// ***** I/O operators:
//
// void Print()
//      prints value of BigInt (member function)
// ostream & operator << (ostream & os, const BigInt & b)
//      stream operator to print value
//
// istream & operator >> (istream & in, const BigInt & b)
//      reads whitespace delimited BigInt from input stream in
//


#include <iostream.h>
#include "apstring.h"      // for strings
#include "apvector.h"      // for sequence of digits
```

Appendix B

```
class BigInt
{
   public:
     BigInt();                          // default constructor, value = 0
     BigInt(int);                       // assign an integer value
     BigInt(const apstring &);          // assign a string


     // may need these in alternative implementation


     // BigInt(const BigInt &);       // copy constructor
     // ~BigInt();                    // destructor
     // const BigInt & operator = (const BigInt &);
                                      // assignment operator


     // operators: arithmetic, relational

     const BigInt & operator += (const BigInt &);
     const BigInt & operator -= (const BigInt &);
     const BigInt & operator *= (const BigInt &);
     const BigInt & operator *= (int num);


     apstring ToString() const;           // convert to string
     int      ToInt()    const;           // convert to int
     double   ToDouble() const;           // convert to double

     // facilitate operators ==, <, << without friends


     bool Equal(const BigInt & rhs)      const;
     bool LessThan(const BigInt & rhs)   const;
     void Print(ostream & os)            const;

   private:

     // other helper functions


     bool IsNegative() const;  // return true iff number is negative
     bool IsPositive() const;  // return true iff number is positive
     int NumDigits()  const;   // return # digits in number


     int GetDigit(int k) const;
     void AddSigDigit(int value);
     void ChangeDigit(int k, int value);


     void Normalize();


     // private state/instance variables


     enum Sign{positive,negative};
     Sign mySign;                     // is number positive or negative
     apvector<char> myDigits;         // stores all digits of number
     int myNumDigits;                 // stores # of digits of number
};
```

```
// free functions

ostream & operator <<(ostream &, const BigInt &);
istream & operator >>(istream &, BigInt &);

BigInt operator +(const BigInt & lhs, const BigInt & rhs);
BigInt operator -(const BigInt & lhs, const BigInt & rhs);
BigInt operator *(const BigInt & lhs, const BigInt & rhs);
BigInt operator *(const BigInt & lhs, int num);
BigInt operator *(int num, const BigInt & rhs);

bool operator == (const BigInt & lhs, const BigInt & rhs);
bool operator < (const BigInt & lhs, const BigInt & rhs);
bool operator != (const BigInt & lhs, const BigInt & rhs);
bool operator > (const BigInt & lhs, const BigInt & rhs);
bool operator >= (const BigInt & lhs, const BigInt & rhs);
bool operator <= (const BigInt & lhs, const BigInt & rhs);

#endif  // _BIGINT_H not defined
```

Appendix C Contents

Appendix C: bigint.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <limits.h>
#include "bigint.h"
#include "apvector.h"


const int BASE = 10;


// author: Owen Astrachan
//
// BigInts are implemented using a Vector<char> to store
// the digits of a BigInt
// Currently a number like 5,879 is stored as the vector {9,7,8,5}
// i.e., the least significant digit is the first digit in the vector; for
// example GetDigit(0) returns 9 and GetDigit(3) returns 5.
// All operations on digits should be done using private
// helper functions:
//
// int GetDigit(k)         – return k-th digit
// void ChangeDigit(k,val) – set k-th digit to val
// void AddSigDigit(val)   – add new most significant digit val
//
// by performing all ops in terms of these private functions we
// make implementation changes simpler
//
// I/O operations are facilitated by the ToString() member function
// which converts a BigInt to its string (ASCII) representation


BigInt::BigInt()
// postcondition: bigint initialized to 0
   : mySign(positive),
     myDigits(1,'0'),
     myNumDigits(1)
{
      // all fields initialized in initializer list
}


BigInt::BigInt(int num)
// postcondition: bigint initialized to num
   : mySign(positive),
     myDigits(1,'0'),
     myNumDigits(0)
{
   // check if num is negative, change state and num accordingly


    if (num < 0)
  {
    mySign = negative;
    num = -1 * num;
  }
```

```
   // handle least-significant digit of num (handles num == 0)

   AddSigDigit(num % BASE);
   num = num / BASE;

   // handle remaining digits of num

   while (num != 0)
   {
      AddSigDigit(num % BASE);
      num = num / BASE;
  }
}

BigInt::BigInt(const apstring & s)
// precondition: s consists of digits only, optionally preceded by + or -
// postcondition: bigint initialized to integer represented by s
//                if s is not a well-formed BigInt (e.g., contains
//                non-digit characters) then an error message is
//                printed and abort called
   : mySign(positive),
     myDigits(s.length(),'0'),
     myNumDigits(0)
{
   int k;
   int limit = 0;

   if (s.length() == 0)
   {
      myDigits.resize(1);
      AddSigDigit(0);
      return;
   }
   if (s[0] == '-')
   {
      mySign = negative;
      limit = 1;
   }
   if (s[0] == '+')
   {
       limit = 1;
   }
   // start at least significant digit

   for(k=s.length() - 1; k >= limit; k—)
   {
      if (! isdigit(s[k]))
      {
         cerr << "badly formed BigInt value = " << s << endl;
         abort();
      }
         AddSigDigit(s[k]-'0');
   }
   Normalize();
}
```

```
const BigInt & BigInt::operator -=(const BigInt & rhs)
// postcondition: returns value of bigint - rhs after subtraction
{
   int diff;
   int borrow = 0;

   int k;
   int len = NumDigits();

   if (this == &rhs)     // subtracting self?
   {
      *this = 0;
      return *this;
   }

   // signs opposite? then lhs - (-rhs) = lhs + rhs

   if (IsNegative() != rhs.IsNegative())
   {
      *this +=(-1 * rhs);
      return *this;
   }
   // signs are the same, check which number is larger
   // and switch to get larger number "on top" if necessary
   // since sign can change when subtracting
   // examples: 7 - 3 no sign change,     3 - 7 sign changes
   //          -7 - (-3) no sign change, -3 -(-7) sign changes
   if (IsPositive() && (*this) < rhs ||
       IsNegative() && (*this) > rhs)
   {
      *this = rhs - *this;
      if (IsPositive()) mySign = negative;
      else              mySign = positive;  // toggle sign
      return *this;
   }
   // same sign and larger number on top

   for(k=0; k < len; k++)
   {
      diff = GetDigit(k) - rhs.GetDigit(k) - borrow;
      borrow = 0;
      if (diff < 0)
      {
          diff += 10;
          borrow = 1;
      }
      ChangeDigit(k,diff);
   }
   Normalize();
   return *this;
}
```

Appendix C

```
const BigInt & BigInt::operator +=(const BigInt & rhs)
// postcondition: returns value of bigint + rhs after addition
{

   int sum;
   int carry = 0;

   int k;
   int len = NumDigits();     // length of larger addend

   if (this == &rhs)         // to add self, multiply by 2
   {
      *this *= 2;
      return *this;
   }

   if (IsPositive() != rhs.IsPositive())  // signs not the same, subtract
   {
      *this -= (-1 * rhs);
      return *this;
   }

   // process both numbers until one is exhausted

   if (len < rhs.NumDigits())
   {
       len = rhs.NumDigits();
   }
   for(k=0; k < len; k++)
   {
       sum = GetDigit(k) + rhs.GetDigit(k) + carry;
       carry = sum / BASE;
       sum = sum % BASE;

       if (k < myNumDigits)
       {
           ChangeDigit(k,sum);
       }
       else
       {
           AddSigDigit(sum);
       }
   }
   if (carry != 0)
   {
      AddSigDigit(carry);
   }
   return *this;
}

BigInt operator +(const BigInt & lhs, const BigInt & rhs)
// postcondition: returns a bigint whose value is lhs + rhs
{
   BigInt result(lhs);
   result += rhs;
   return result;
}
```

```
BigInt operator -(const BigInt & lhs, const BigInt & rhs)
// postcondition: returns a bigint whose value is lhs - rhs
{
   BigInt result(lhs);
   result -= rhs;
   return result;
}

void BigInt::Print(ostream & os) const
// postcondition: BigInt inserted onto stream os
{
   os << ToString();
}

apstring BigInt::ToString() const
// postcondition: returns apstring equivalent of BigInt
{
   int k;
   int len = NumDigits();
   apstring s = "";

   if (IsNegative())
   {
       s = '-';
   }
   for(k=len-1; k >= 0; k--)
   {
       s += char('0'+GetDigit(k));
   }
   return s;
}

int BigInt::ToInt() const
// precondition: INT_MIN <= self <= INT_MAX
// postcondition: returns int equivalent of self
{
   int result = 0;
   int k;
   if (INT_MAX < *this) return INT_MAX;
   if (*this < INT_MIN) return INT_MIN;

   for(k=NumDigits()-1; k >= 0; k--)
   {
       result = result * 10 + GetDigit(k);
   }
   if (IsNegative())
   {
       result *= -1;
   }
   return result;
}
```

Appendix C

```
double BigInt::ToDouble() const
// precondition: DBL_MIN <= self <= DLB_MAX
// postcondition: returns double equivalent of self
{
   double result = 0.0;
   int k;
   for(k=NumDigits()-1; k >= 0; k--)
   {
      result = result * 10 + GetDigit(k);
   }
   if (IsNegative())
   {
      result *= -1;
   }
   return result;
}

ostream & operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out
{
   big.Print(out);
   return out;
}

istream & operator >> (istream & in, BigInt & big)
// postcondition: big extracted from in, must be whitespace delimited
{
   apstring s;
   in >> s;
   big = BigInt(s);
   return in;
}

bool operator == (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs == rhs, else returns false
{
   return lhs.Equal(rhs);
}

bool BigInt::Equal(const BigInt & rhs) const
// postcondition: returns true if self == rhs, else returns false
{

   if (NumDigits() != rhs.NumDigits() || IsNegative() != rhs.IsNegative())
   {
      return false;
   }
   // assert: same sign, same number of digits;

   int k;
   int len = NumDigits();
   for(k=0; k < len; k++)
   {
      if (GetDigit(k) != rhs.GetDigit(k)) return false;
   }

  return true;
}
```

```
bool operator != (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns true if lhs != rhs, else returns false
{
   return ! (lhs == rhs);
}

bool operator < (const BigInt & lhs, const BigInt & rhs)
// postcondition: return true if lhs < rhs, else returns false
{
   return lhs.LessThan(rhs);
}

bool BigInt::LessThan(const BigInt & rhs) const
// postcondition: return true if self < rhs, else returns false
{
   // if signs aren't equal, self < rhs only if self is negative

   if (IsNegative() != rhs.IsNegative())
   {
       return IsNegative();
   }

   // if # digits aren't the same must check # digits and sign

   if (NumDigits() != rhs.NumDigits())
   {
       return (NumDigits() < rhs.NumDigits() && IsPositive()) ||
              (NumDigits() > rhs.NumDigits() && IsNegative());
   }

   // assert: # digits same, signs the same

   int k;
   int len = NumDigits();

   for(k=len-1; k >= 0; k--)
   {
       if (GetDigit(k) < rhs.GetDigit(k)) return IsPositive();
       if (GetDigit(k) > rhs.GetDigit(k)) return IsNegative();
   }
   return false;   // self == rhs
}

bool operator > (const BigInt & lhs, const BigInt & rhs)
// postcondition: return true if lhs > rhs, else returns false
{
   return rhs < lhs;
}

bool operator <= (const BigInt & lhs, const BigInt & rhs)
// postcondition: return true if lhs <= rhs, else returns false
{
   return lhs == rhs || lhs < rhs;
}
bool operator >= (const BigInt & lhs, const BigInt & rhs)
// postcondition: return true if lhs >= rhs, else returns false
{
   return lhs == rhs || lhs > rhs;
}
```

```
void BigInt::Normalize()
// postcondition: all leading zeros removed
{
   int k;
   int len = NumDigits();
   for(k=len-1; k >= 0; k--)         // find a non-zero digit
   {
       if (GetDigit(k) != 0) break;
       myNumDigits—;                 // "chop" off zeros
   }
   if (k < 0)  // all zeros
   {
       myNumDigits = 1;
       mySign = positive;
   }
}

const BigInt & BigInt::operator *=(int num)
// postcondition: returns num * value of BigInt after multiplication
{
   int carry = 0;
   int product;        // product of num and one digit + carry
   int k;
   int len = NumDigits();

   if (0 == num)      // treat zero as special case and stop
   {
      *this = 0;
      return *this;
   }

   if (BASE < num || num < 0)    // handle pre-condition failure
   {
      *this *= BigInt(num);
      return *this;
   }

   if (1 == num)       // treat one as special case, no work
   {
      return *this;
   }

   for(k=0; k < len; k++)   // once for each digit
   {
      product = num * GetDigit(k) + carry;
      carry = product/BASE;
      ChangeDigit(k,product % BASE);
   }

   while (carry != 0)     // carry all digits
   {
      AddSigDigit(carry % BASE);
      carry /= BASE;
   }
   return *this;
}
```

```
BigInt operator *(const BigInt & a, int num)
// postcondition: returns a * num
{
   BigInt result(a);
   result *= num;
   return result;
}

BigInt operator *(int num, const BigInt & a)
// postcondition: returns num * a
{
   BigInt result(a);
   result *= num;
   return result;
}

const BigInt & BigInt::operator *=(const BigInt & rhs)
// postcondition: returns value of bigInt * rhs after multiplication
{
   // uses standard “grade school method” for multiplying

   if (IsNegative() != rhs.IsNegative())
   {
       mySign = negative;
   }
   else
   {
       mySign = positive;
   }

   BigInt self(*this);               // copy of self
   BigInt sum(0);                    // to accumulate sum
   int k;
   int len = rhs.NumDigits();        // # digits in multiplier

   for(k=0; k < len; k++)
   {
      sum += self * rhs.GetDigit(k);     // k-th digit * self
      self *= 10;                        // add a zero
   }
   *this = sum;
   return *this;
}

BigInt operator *(const BigInt & lhs, const BigInt & rhs)
// postcondition: returns a bigint whose value is lhs * rhs
{
   BigInt result(lhs);
   result *= rhs;
   return result;
}

int BigInt::NumDigits() const
// postcondition: returns # digits in BigInt
{
   return myNumDigits;
}
```

Appendix C

```
int BigInt::GetDigit(int k) const
// precondition: 0 <= k < NumDigits()
// postcondition:  returns k-th digit
//                 (0 if precondition is false)
//                 Note: 0th digit is least significant digit
{
   if (0 <= k && k < NumDigits())
   {
      return myDigits[k] - '0';
   }
   return 0;
}

void BigInt::ChangeDigit(int k, int value)
// precondition: 0 <= k < NumDigits()
// postcondition: k-th digit changed to value
//                Note: 0th digit is least significant digit

{
   if (0 <= k && k < NumDigits())
   {
      myDigits[k] = char('0' + value);
   }
   else
   {
      cerr << "error changeDigit " << k << " " << myNumDigits << endl;
   }
}

void BigInt::AddSigDigit(int value)
// postcondition: value added to BigInt as most significant digit
//                Note: 0th digit is least significant digit

{
   if (myNumDigits >= myDigits.length())
   {
       myDigits.resize(myDigits.length() * 2);
   }
       myDigits[myNumDigits] = char('0' + value);
       myNumDigits++;
}
```

```
bool BigInt::IsNegative() const
// postcondition: returns true iff BigInt is negative
{
   return mySign == negative;
}

bool BigInt::IsPositive() const
// postcondition: returns true iff BigInt is positive
{
   return mySign == positive;
}
```

Appendix D: bigint.cpp with Aliasing Problems

The arithmetic operators *=, +=, and –= that do not check for aliasing are shown here. Compare these implementations with those in Appendix C which do check for aliasing.

```
const BigInt & BigInt::operator +=(const BigInt & rhs)
// postcondition: returns value of bigint + rhs after addition
{

   int sum;
   int carry = 0;

   int k;
   int len = NumDigits();     // length of larger addend

   if (IsPositive() != rhs.IsPositive())  // signs not the same, subtract
   {
      *this -= (-1 * rhs);
      return *this;
   }

   // process both numbers until one is exhausted

   if (len < rhs.NumDigits())
   {
       len = rhs.NumDigits();
   }
   for(k=0; k < len; k++)
   {
       sum = GetDigit(k) + rhs.GetDigit(k) + carry;
       carry = sum / BASE;
       sum = sum % BASE;

       if (k < myNumDigits)
       {
           ChangeDigit(k,sum);
       }
       else
       {
           AddSigDigit(sum);
       }
   }
   if (carry != 0)
   {
       AddSigDigit(carry);
   }
   return *this;
}
```

```
const BigInt & BigInt::operator -=(const BigInt & rhs)
// postcondition: returns value of bigint - rhs after subtraction
{
   int diff;
   int borrow = 0;

   int k;
   int len = NumDigits();

   // signs opposite? then lhs - (-rhs) = lhs + rhs

   if (IsNegative() != rhs.IsNegative())
   {
      *this +=(-1 * rhs);
      return *this;
   }
   // signs are the same, check which number is larger
   // and switch to get larger number "on top" if necessary
   // since sign can change when subtracting
   // examples: 7 - 3 no sign change,     3 - 7 sign changes
   //          -7 - (-3) no sign change, -3 -(-7) sign changes
   if (IsPositive() && (*this) < rhs ||
       IsNegative() && (*this) > rhs)
   {
       *this = rhs - *this;
       if (IsPositive()) mySign = negative;
       else              mySign = positive;  // toggle sign
       return *this;
   }
   // same sign and larger number on top

   for(k=0; k < len; k++)
   {
       diff = GetDigit(k) - rhs.GetDigit(k) - borrow;
       borrow = 0;
       if (diff < 0)
       {
           diff += 10;
           borrow = 1;
       }
       ChangeDigit(k,diff);
   }
   Normalize();
   return *this;
}
```

Appendix D

```
const BigInt & BigInt::operator *=(const BigInt & rhs)
// postcondition: returns value of bigint * rhs after multiplication
{
   // uses standard "grade school method" for multiplying

   if (IsNegative() != rhs.IsNegative())
   {
       mySign = negative;
   }
   else
   {
       mySign = positive;
   }

   BigInt sum(0);               // to accumulate sum
   int k;
   int len = rhs.NumDigits();   // # digits in multiplier

   for(k=0; k < len; k++)
   {
   sum += (*this) * rhs.GetDigit(k);    // k-th digit * self
   (*this) *= 10;                       // add a zero
   }
   *this = sum;
   return *this;
}
```

Appendix E: Test programs

The program testadd.cpp to test addition of BigInt values.

```
#include <iostream.h>
#include "bigint.h"

int main()
{
   BigInt a,b;
   while (true)
   {
          cout << "enter two bigint values: ";
          cin >> a >> b;
          cout << "terms: " << endl << a << endl;
          cout << b << endl;
          cout << "sum = " << a + b << endl;
   }
   return 0;
}
```

The program testio.cpp to test I/O of BigInt values.

```
#include <iostream.h>
#include "bigint.h"

// program to test BigInt I/O (and string conversion)

int main()
{
   BigInt a;

   while (true)
   {
          cout << "enter a big integer: ";
          cin >> a;
          cout << "bigint = " << a << endl;
   }
   return 0;
}
```

Appendix E

The program testconv.cpp to test conversion of BigInt values.

```
#include <iostream.h>
#include "bigint.h"

int main()
{
   BigInt a;

   while (true)
  {
          cout << "enter big: ";
          cin >> a;

          cout << "a = " << a << " as int = " << a.ToInt() << endl;
          cout << " as double = " << a.ToDouble() << endl;
   }

   return 0;
}
```

The program testcomp.cpp to test comparison of BigInt values.

```
#include <iostream.h>
#include "bigint.h"

int main()
{
   BigInt a,b;

   while (true)
   {
          cout << "enter two numbers: ";
          cin >> a >> b;

          cout << "first is less than second: " << (a < b) << endl;
          cout << "first is greater than second: " << (a > b) << endl;
          cout << "first is equal to second: " << (a == b) << endl;

   }

   return 0;
}
```

Appendix F: Sample AP Examination Questions

Multiple-Choice

Questions 1 - 2 refer to the following information.

The current version of the BigInt class stores the digits in an array of characters, with the least significant digit as the first element of the array. For example, the integer 1,234 is stored as the array

'4'	'3'	'2'	'1'

Consider changing the implementation of a BigInt so that the digits are stored in the opposite order, with the most significant digit as the first element.

1. Which of the following functions would have to be modified to implement the change described above?

   I. GetDigit
   II. NumDigits
   III. Normalize

   (A) I only
   (B) II only
   (C) III only
   (D) I and II
   (E) I and III

2. Consider the following function. This function relies on the original implementation of a BigInt.

```
apstring BigInt::toString() const
{
    int k;
    int len = numDigits();
    apstring s = "";
    if (isNegative())
    {
       s = '-';
    }
    for (k = len - 1; k >= 0; k--)
    {
         s += char('0' + getDigit(k));
    }
    return s;
}
```

What changes to this function would be necessary to incorporate the new implementation of a BigInt described above?

(A) Replacing int len = numDigits() with int len = numDigits() + 1
(B) Replacing the current for loop with for (k = 0; k <= len; k++)
(C) Replacing the current for loop with for (k = 0; k < len; k++)
(D) Replacing s += char("0" + getDigit(k)) with s += char("1" + getDigit(k))
(E) No change is necessary

3. What will happen when a program that contains the declaration.

```
BigInt A("123x");
```

is executed?

(A) An error message will be printed and the program will be halted.
(B) A global variable error will be set to true.
(C) Variable A will be initialized to 0.
(D) Variable A will be initialized to 123.
(E) The special error field of variable A will be set to indicate that A is invalid.

4. What changes to the current version of function lessThan would be needed to convert it to function greaterThan?

I. Change every occurrence of IsNegative in the original version to IsPositive in the new version.
II. Change every occurrence of IsPositive in the original version to IsNegative in the new version.
III. Change every occurrence of < in the original version to > in the new version, and > in the original version to < in the new version.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

5. Consider the following declaration.

```
BigInt  x(12497);
```

How many times is the x.myDigits vector resized when the BigInt constructor is called to initialize x to 12497?

(A) 0
(B) 1
(C) 2
(D) 3
(E) 5

6. The member function IsOdd is to be added to the BigInt class. IsOdd should return true when the number is odd and false when it is even. Consider the following incomplete implementation of IsOdd.

```
bool BigInt::IsOdd()const
{
  <missing code>
}
```

Which of the following statements can be substituted for <*missing code*> so that IsOdd will work as intended?

(A) return ((getDigit(numDigits()) % 2) != 0);
(B) return ((getDigit(numDigits()-1) % 2) != 0);
(C) return ((getDigit(1) % 2) != 0);
(D) return ((getDigit(0) % 2) != 0);
(D) return (getDigit(1) == 1);

7. The implementation of operator == for BigInts uses the private helper function Equal. Why is function Equal used, instead of simply implementing the equality test directly in the code for operator == ?

   (A) The operator == is not defined as a member function and therefore cannot call GetDigit.
   (B) The Equal function is required to prevent an aliasing problem.
   (C) The Equal function is required to prevent side effects.
   (D) The Equal function is required because the parameters to operator == are const reference.
   (E) Use of the Equal function makes operator == run more quickly.

Answers to Multiple-Choice Questions.

1. E
2. E
3. A
4. D
5. D
6. D
7. A

Free Response

1. 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.

   Consider implementing operator /= for BigInts. Integer division involves dividing one integer (the dividend) by another integer (the divisor) resulting in an answer (the quotient). For example, in the problem 13 / 4 = 3, the dividend is 13, the divisor is 4, and the quotient is 3.

   An attempt to divide by zero is an error, and should cause an error message to be printed, and abort to be called.

   Here are some examples that illustrate how operator /= should work.

Value of BigInt A	Value of BigInt B	Value of BigInt A after executing: A /= B;
13	4	3
4	13	0
100000	−2	−50000
−2	100000	0
4	0	divide by zero error

   (a) Write function operator /=, whose header is given below. In writing operator /= you may use the following simple algorithm for dividing two integers using repeated subtraction (although this may be less efficient than other algorithms).

   1. Copy self to tempDividend; make tempDividend positive.
   2. Copy the divisor to tempDivisor; make tempDivisor positive.
   3. Set quotient to zero.
   4. While tempDividend >= tempDivisor do
      (i) subtract tempDivisor from tempDividend.
      (ii) increment quotient by one.
   5. Fix the sign of quotient if necessary.

   Complete function operator /= below the following header.

```
const BigInt & BigInt::operator /= (const BigInt & divisor)
// precondition:  bigint = a
// postcondition: bigint = a / divisor, returns result
```

(b) Consider implementing operator/ for BigInts. Here are some examples of how operator / should work.

Value of BigInt A	Value of BigInt B	Value of Answer after executing: Answer = A / B;
13	4	3
4	13	0
100000	−2	−50000
−2	100000	0
4	0	divide by zero error

In writing operator/ you may call function operator /= of part (a). Assume that operator /= works as specified, regardless of what you wrote in part (a).

Complete function operator/ below the following header.

```
BigInt  operator / (const BigInt & lhs, const BigInt & rhs)
```

Solutions:

Part (a):

```
const BigInt & BigInt::operator /= (const BigInt & divisor)
// precondition:  self = a
// postcondition: if divisor != 0 then self = a/divisor, returns
//                reference to self
//                otherwise prints an error message and calls abort
{

   if (divisor == 0)
   {
       cerr << "Divide by 0 error" << endl;
       abort();
   }
   // uses repeated subtraction to solve division

   // Steps 1-3 (make copies of dividend, divisor, initialize quotient)

   BigInt dividend(*this);      // copy of self, handles aliasing
   BigInt tempDivisor(divisor);
   BigInt quotient(0);
   dividend.mySign = positive;
   tempDivisor.mySign = positive;
   while (dividend >= tempDivisor)
   {
       dividend -= tempDivisor;
       quotient += 1;
   }
   if (IsNegative() != divisor.IsNegative())
   {
       quotient.mySign = negative;
       quotient.Normalize();    // cleans up -0 answer to 0
   }
   *this = quotient;
   return *this;
}
```

Part (b):

```
BigInt operator / (const BigInt & lhs, const BigInt & rhs)
{
   BigInt result(lhs);
   result /= rhs;
   return result;
}
```

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.

   The three new member functions whose prototypes are given below are to be added to the class.

```
int     NumRightZeroes();
void    ShiftRight(int k);
void    SigDigs();
```

   (a) Write the member function NumRightZeroes whose header is given below. If x is a BigInt, the call x.NumRightZeroes() should return the number of zeroes to the right of the rightmost nonzero digit of x. If x is zero, then x.NumRightZeroes() should return zero. For example,

x	x.NumRightZeroes()
1230400560000	4
1234567	0
0	0

   Complete the member function NumRightZeroes below the following header.

```
int BigInt::NumRightZeroes()
```

(b) Write the member function ShiftRight whose header is given below. If x is a BigInt, the call x.ShiftRight(k) should change x so that its k rightmost digits are removed and all other digits are shifted to the right k places. The places on the left that are vacated by the shift should be filled with zeroes. If k is zero, x should be unchanged. For example,

Initial value of x	Call	x after the call
123456789	x.ShiftRight(3)	000123456
123456789	x.ShiftRight(7)	000000012
123456789	x.ShiftRight(9)	000000000
0	x.ShiftRight(1)	0

Complete the member function ShiftRight below the following header. Assume that ShiftRight is only called with parameters that satisfy its precondition.

```
void BigInt::ShiftRight(int k)
//  0 <= k <= NumDigits()
```

(c) Write the member function RemoveRightZeroes whose header is given below. If x is a BigInt, the call x.RemoveRightZeroes() should remove all zeroes to the right of the rightmost digit in x. If x is zero, then the call x.RemoveRightZeroes() should leave x unchanged. For example,

x	x after the call x.RemoveRight Zeroes()
1230400560000	123040056
1234567	1234567
0	0

In writing RemoveRightZeroes you may call functions NumRightZeroes and ShiftRight specified in parts (a) and (b). Assume that NumRightZeroes and ShiftRight work as specified, regardless of what you wrote in parts (a) and (b).

Complete the member function RemoveRightZeroes below the following header.

```
void  BigInt::RemoveRightZeroes()
```

Appendix F

Solutions

```
int BigInt::NumRightZeroes()
{
   int k = 0;
   while((k < NumDigits()) && (GetDigit(k) == 0))
   {
     k++;
   }

   if((k == 1) && (NumDigits() == 1))
     return 0;
   else
     return k;
}

void BigInt::ShiftRight(int k)
{
   int s, t = 0;

   for(s = k; s < NumDigits(); s++)
   {
     changeDigit(t, GetDigit(s));
     t++;
   }
   for(s = numDigits() - k; s < NumDigits(); s++)
   {
     changeDigit(s,0);
   }
}

void BigInt::RemoveRightZeroes()
{
   ShiftRight(NumRightZeroes());
   Normalize();
}
```

Alternate definition for NumRightZeroes using *this

```
int BigInt::NumRightZeroes()
{
   int k = 0;
   if (*this != 0)
   {
     while(GetDigit(k) == 0)
     {
       k++;
     }
   }
   return k;
}
```

Appendix G: Answers to Study Questions

Page 3

1. The enumerated type Optype would need to include the new operators. Functions StringToOp and GetOperator would need to be modified to allow for entry of the new operators and to return the appropriate OpType. In the main program the first two statements at the beginning of the while loop would need a conditional guard so that they would not be executed if a unary operator were specified, and two cases corresponding to the new operators must be added to the switch statement to set accumulator to the appropriate value.

2. The only change required would be in the function StringToOp: the names would replace the single character symbols in the definition of the apstring OPSTRING[ ].

3. The only changes required would be the types of the variables accumulator and current in the function main(): their type would be integer rather than double.

4. Consult the manuals for your version of C++ for assistance here.

5. See question 1 for general guidelines. The specific enumerated type to add to OpType could be an identifier such as Clear.

6. Expressions with parentheses must be evaluated using two steps: First, convert the expression to postfix form; second, evaluate the postfix expression. Most data structures textbooks include algorithms for these two steps.

1. This is system dependent. Most C++ systems use 32 bit integers, and in this case the largest value is +2,147,483,647 and the smallest value is −2,147,483,648. A few systems (including most Pascal systems) use 16 bit integers, and then the largest value is +32767 and the smallest value is −32768.

2. The digits could be stored as a linked list of integers or a linked list of characters. They could also be stored as an apvector of integers or characters, since apvectors, unlike standard arrays in most programming languages, can be resized as required to meet the needs of a running program. If the method chosen for storing the digits did not allow dynamic change as the program runs, it would be necessary to design the program to accommodate a worst case estimate of the number of digits expected in any BigInt value.

3. Multiplication should be the most difficult since it involves adding many BigInts, each of which is the result of multiplying one BigInt by a single digit.

4. *abcd*1234 causes the program to get stuck in an infinite loop within the GetOperator function: the failure to read an integer when a number is expected leaves the input stream in an illogical state, resulting in failure to pause for further input. 1234*abcd* results in no particular problem: the integer 1234 is accepted as input and the remaining characters are discarded. This result, however, can be system dependent. Entering an operator other than the legal operators simply results in being reprompted for an operator: the function StringToOp returns illegal_op in such cases, and GetOperator then reprompts for a new operator.

5. Possible answers might include division, %, exponentiation, less than or equal, greater than or equal, odd, even ....

6. If, for example, linked lists (pointers) were used to store the digits of a BigInt, a destructor would be required to deallocate the storage that is created (via the new operator) each time a BigInt is defined. Similarly, a copy constructor would be required to allocate new storage when making a copy of a BigInt. For similar reasons the assignment operator = would need to be defined so that an assignment x = a results in a complete copy being made of the value of a.

7.
```
BigInt operator - (const BigInt & big, int small)
{
   return big + (-small);
}
```

Page 10

1. Answers will vary according to C++ versions and error checking options that have been specified by the programmer.

2. Division by zero, overflow, integer too large, user inputs characters instead of digits.

```
ErrorType = (ZeroDivide, Overflow, BadInput);
```

3. If the code has been thoroughly tested and no errors exist, then error checking could be turned off.

4. Given some client programs, this method would be more desirable than halting the program or ignoring the error. However, giving a user the option of ignoring the error may produce unpredictable results in the program. The client program would need to offer these three options at any point at which an error could occur, making that program much more difficult to write. This approach is only appropriate for interactive programs. It would mean that the BigInt package could not be used for non-interactive programs.

5. This would be a good approach, but it requires considerable care to implement properly. It certainly leads to longer code since each function must consider and respond to the value of the global error variable. It also leaves uncertainty as to the source of an error if the client program has more than one BigInt to manipulate at a time. In such cases where was the error? What if the client program fails to check at some point? What if the client program resets the global variable to NoError. Strengths: The client program is given the option of dealing with the error, rather than ignoring the error or halting the client program. Weaknesses: the implementation of the BigInt package would be considerably more complex, and the client program would normally also be more complicated if the information in the global error variable is taken into account.

Page 17

1. A char vector uses less storage than an int vector: on most systems characters are stored in a single byte (8 bits) of memory whereas an integer is allocated 4 bytes of memory. The various BigInt functions will handle the conversions from chars to ints and conversely.

2. A Boolean value could be used with false meaning positive and true meaning negative. Alternatively, the integer values 0 and 1 or the characters + and − could be used. We could also choose to represent the sign of a BigInt as an additional character (+ or −) in the digit vector itself rather than in a separate variable.

3. Code for the GetDigit function is given on page 61. Note that it checks whether the parameter digit is within the bounds of the digit vector and takes appropriate action if it is not. The decision to return 0 in case the parameter is not within the vector bounds is a convenience that greatly simplifies code in definitions of several of the arithmetical functions.

4. Non-member functions operator == and operator < would not be able to call the private function GetDigit nor would they have direct access to the private variable myDigits. This could be remedied by making the non-member functions "friends" of the BigInt class, but use of the friend designation is discouraged since it violates the basic idea of data hiding. Instead, the functions NumDigits and GetDigit could be made public member functions. Then the following code would implement operator == for positive BigInt values.

```
bool operator == (const BigInt & lhs, const BigInt & rhs)
// precondition:   lhs and rhs are positive BigInt values.
// postcondition:  returns true if lhs == rhs, else returns false.
{
      if ( lhs.NumDigits() != rhs.NumDigits() return false;

      int k;
      int len = lhs.NumDigits();
      for ( k=0; k < len; k++ )
            if ( lhs.GetDigit(k) != rhs.GetDigit(k) return false;

      return true;
}
```

5. The built-in array type imposes limits on array lengths, and in particular an array's length cannot be changed at run time. This does not meet the specification that BigInt objects have no limit on their size (except for the limit on total memory available on a given computer). The apvector class imposes no limit on length and permits arrays to be resized at runtime.

Appendix G

Page 23

1. Answers should be specific instances of cases mentioned in the preceding paragraphs. For example, the following cases would be appropriate:

1	Typical good value
120	Typical good value ending with 0
0	Valid value
+123	Valid, uses +
−123	Valid, uses −
12+3	Invalid: contains an imbedded +
12−3	Invalid: contains an imbedded −
123d	Invalid: contains a bad character
d123	Invalid: contains a bad character
12d3	Invalid: contains a bad character
000	Valid value (fixed later)
00123	Valid value (fixed later)
some very long sequences of digits, with and without +, −, and bad characters	

   To test the constructor with an int parameter, modify testio.cpp by replacing the while loop body with

```
int n;
cout << "enter an integer: ";
cin >> n;
BigInt a(n);
cout << a;
```

2. The stream member function width() does work as expected. Recall that this function only controls the field width of the next stream insertion; thus it must immediately precede cout << a, as in the code fragment below. Recall also that strings are left justified in their field; thus in normal circumstances one wants to also make the call cout.setf(ios::right) to cause subsequent outputs to be right justified in their fields.

```
cout.setf(ios::right);
while (true)
{
      cout << "enter a big integer: ";
      cin >> a;
      cout << "bigint = ";
      cout.width(20);
      cout << a << endl;
}
```

Most systems provide manipulators that can be inserted directly into output streams to control output. This provides an alternative to calling the member functions of cout directly. For example in Metrowerks CodeWarrior the following code is equivalent to the above:

```
while (true)
  {
          cout << "enter a big integer: ";
                cin >> a;
          cout << "bigint = " << right << setw(20) << a << endl;
     }
```

3. There are several possibilities here. Perhaps the simplest is to initialize the number of digits myNumDigits to zero which is not a valid value. All the BigInt member functions would need to check myNumDigits (or use the helper function NumDigits) to determine if a BigInt is in an uninitialized state. Using a default value of zero makes the code much easier to develop since there's no need to include error checking code, or a call to an error checking function, in all the member functions.

4. The implementation on page 52 already uses a constant BASE to represent the base of the number system. All of the arithmetical functions use this constant rather then its default value 10, so if BigInts are initialized to numbers in base 2, or 3, or any one digit base, all arithmetic is carried out in this base. Only the conversion functions ToInt and ToDouble need to be changed to reflect a different value of BASE. It would be appropriate, also, to change the constructor that takes a string as input so that digits other than those in the range 0 <= digit <= BASE − 1 would be treated as bad input.

Page 25

1. The extraction operator >> would pass any leading zeros typed by the user to the constructor that takes a string parameter. Thus this constructor could create BigInts that have leading zeros. All of the arithmetic operators could produce leading zeros. Thus all of the arithmetic operators would need to check their input parameters for leading zeros and eliminate them, as would all of the comparison operators. Otherwise erroneous results may be obtained or overflow errors may be generated.

2. The size of the vector myDigits would be initialized to length 100, even though the value of the BigInt is zero. This is a waste of storage.

3. Only subtraction of numbers that have the same sign (or addition of numbers with opposite signs) can produce leading zeros. Multiplication does not produce leading zeros in any case. As implemented on page 54, therefore, only the −= operator can introduce leading zeros, and it removes them. All the other arithmetic operators ultimately refer the case of subtraction of numbers of the same sign back to −=.

4. A fraction will consist of two BigInts that represent the numerator and denominator of the fraction. Reducing the fraction to lowest terms, therefore, requires dividing the numerator and denominator by their greatest common divisor. A greatest common divisor function for BigInts can be defined by the usual (Euclidean) algorithm once the operators / (integer division) and % (mod) are defined for BigInts. Implementing / and % for BigInts is therefore a worthy exercise, best left until the standard operations +, =, and ∗ are understood.

Page 29

1.
```
bool operator != (const BigInt & lhs, const BigInt & rhs)
// postcondition:     returns true if lhs != rhs, else returns false
{
   return ( ! lhs == rhs );
}

bool operator <= (const BigInt & lhs, const BigInt & rhs)
// postcondition:     returns true if lhs <= rhs, else returns false
{
   return ( lhs < rhs || lhs == rhs );
}
```
   Operator <= might be implemented more efficiently using a procedure that only compares the list of digits once, rather than once for the less than check and once for the equal check. However, the implementation of <= given here (using existing functions) is very straightforward and so is less likely to contain errors.

2. BigInt::equal() can traverse from either direction since every digit must be the same. Greater or less-than comparisons need to go from the most to least significant digit.

3.
```
int Compare(const BigInt & A, const BigInt & B)
// postcondition: return -1 if A < B, 0 if A == B, 1 if A > B
{
 if ( A < B )
                return -1;
   else if ( A == B )
     return 0;
   else
                return 1;
}
```

4. Advantages: There is only one routine that is needed to solve any relational operation with BigInts.

   Disadvantages: The value returned (−1, 0, 1) would be more difficult to work with compared to boolean values, both because it could not be used directly in conditions and because it would require remembering what each value meant.

Page 34

1.
```
 const BigInt & BigInt::operator -= (const BigInt & rhs)
// precondition:      bigint = a
// postcondition:     bigint = a - rhs, returns result
{
   BigInt copy(rhs);
   // change sign of copy, add, return result
   if ( copy.mySign = positive )
        copy.mySign = negative;
   else
        copy.mySign = positive;
   // now copy == -rhs
   return ( *this += copy );
}
```

Appendix G

2.
```
if ( IsPositive() != rhs.IsPositive()
{
   BigInt copy(rhs);
   // change sign of copy, subtract, return result
   if ( copy.mySign = positive )
        copy.mySign = negative;
   else
        copy.mySign = positive;
   // now copy == -rhs
   return ( *this -= copy );
}
```

3. Since one pass is made down each digit list, the time would be $2N$ which is O(N). Calculating the Nth Fibonacci number iteratively would require time O(N^2).

Page 37

1. A and B, where A and B have the same number of digits.

abs(A) > abs(B)		abs(B) > abs(A)	
A positive	B positive	A positive	B positive
A negative	B positive	A negative	B positive
A positive	B negative	A positive	B negative
A negative	B negative	A negative	B negative

A and B, where A and B have different numbers of digits.

length of A > length of B		length of B > length of A	
A positive	B positive	A positive	B positive
A negative	B positive	A negative	B positive
A positive	B negative	A positive	B negative
A negative	B negative	A negative	B negative

A and B, where one or both is zero.

A = 0 and B positive	A positive and B = 0
A = 0 and B negative	A negative and B = 0
A = 0 and B = 0	

A and B, where A + B has exactly MaxDigits digits.
A and B, where A − B has exactly MaxDigits digits.
A and B, where A + B has more than MaxDigits digits.
A and B, where A − B has more than MaxDigits digits.

2. The prefix operators ++ and –– can be defined as member functions of BigInt. (The corresponding postfix operators cannot be defined in the same way.)

```
bigint & operator ++()
// precondition: bigint = a
// postcondition: bigint = a + 1, returns a + 1
{
   *this += 1;
   return *this;
}

bigint & operator --()
// precondition: bigint = a
// postcondition: bigint = a + 1, returns a + 1
{
   *this -= 1;
   return *this;
}
```

3. Program execution time might be faster (but not by more than a few comparisons), while programmer time increases significantly both in writing and especially in debugging. Dealing with already tested code saves programmer time.

```
BigInt & BigInt::operator ++()
// precondition: bigint = a
// postcondition: bigint = a + 1, returns a + 1
{
   if ( IsPositive() )                  // Add 1 to bigint
   {
         int currPos = 0;
         while ( getDigit(currPos) == BASE - 1 )
         {
               changeDigit(currPos, 0);
               currPos++;
         }
         if ( currPos < numDigits() )
               changeDigit(currPos, getDigit(currPos) + 1);
         else
               addSigDigit(1);
         }
         else if ( *this == -1 )  // In this case the sign changes
         {
```

```
            changeDigit(0, 0);
            mySign = positive;
      }
      else        // Subtract 1 from its abs val, keep same sign
      {
            int currPos = 0;
            while ( getDigit(currPos) == 0 )
            {
                  changeDigit(currPos, BASE - 1);
                  currPos++;
            }
            changeDigit(currPos, getDigit(currPos) - 1);
            (*this).Normalize(); // Leading zero could be added
      }
      return *this;
   }
```

The code for operator –– would be equally lengthy and similar in appearance.

Page 40

1.
```
const BigInt & BigInt::operator *=(const BigInt & rhs)
// precondition: bigint = a
// postcondition: bigint = a*rhs,returns result
{
   // uses "Horner's rule" algorithm for multiplying

   if (IsNegative() != rhs.IsNegative())
   {
       mySign = negative;
   }
   else
   {
       mySign = positive;
    }

   BigInt self(*this);            // copy of self
   BigInt sum(0);                 // to accumulate sum
   int k;
   int len = rhs.numDigits();     // # digits in multiplier

   for( k = len - 1; k >= 0; k-- ) // start at most sig. digit
   {
        sum *= 10;
        sum += self * rhs.getDigit(k); // k-th digit * self
   }
   *this = sum;
   return *this;
 }
```

2. We refer here to the algorithm for *= given in 1. The same analysis holds for the implementation of *= on page 59. Operator *= applied to two *N*-digit numbers is $O(N^2)$ since the int parameter version of *= is called once (in the for loop) for each digit in the BigInt represented by rhs. Multiplying one *N*-Digit BigInt by a single-digit integer using the int parameter version of *= is *O(N)* since only one pass is made through the digit list of the BigInt.

3. Method (a) is definitely easier for the programmer since all of the functions needed for this operation already exist. Method (b) would require code written from scratch. However, since operator + is designed to add two BigInts, extra operations might be performed unnecessarily by the use of Method (a).

4. Multiplication of an integer BASE by a one-digit integer will not cause overflow so long as BASE < MAX_INT / 10. Thus the precondition can be relaxed to allow BASE to be this large provided we modify the code for operator*=(int). In particular, we could define an auxiliary member function OneDigMultiply, and then the implementation of operator*=(int) could call this function to multiply each digit in the integer. The code for OneDigMultiply is very similar to that of operator*=(int).

Page 46

1.
```
int BigInt::ToInt() const
// precondition: INT_MIN <= self <= INT_MAX
// postcondition: returns int equivalent of self
{
   int result;
   double R;

   R = (*this).toDouble();

   if ( R > INT_MAX || R < -INT_MAX )
         cout << "error: INT_MAX exceeded in toInt()" << endl;
   else
         result = floor(R + 0.5); // Round to nearest integer

   return result;
}
```

2. Add the parameter bool & overflow to the header. Modify the conditional statement that tests whether the BigInt is out of integer range so that it sets the value of overflow (true if out of range, false otherwise).

   This will enable a client program to determine when integer overflow has occurred and take appropriate action. The implementation in the case study simply returns the value INT_MAX or INT_MIN in such cases, with no warning that the integer returned is in error. The disadvantage is that the use of the ToInt() function is more complicated since it must always be called with the third variable.

3.
```
{
   int num;
   cin >> num;
   BigInt big(num);
   cout << big;
}
```

   Test Data:

0	check zero
num < 0	to make sure negatives have a correct sign
num > 0	to make sure positives have a correct sign
num	with varied numbers of digits to make sure nothing extra appears or is lost
INT_MAX, and −INT_MAX − 1	

4.
```
BigInt Factorial(int n)
// precondition: 0 <= n
// postcondition: returns n! = n*(n-1)*(n-2)*...*3*2*1
{
   BigInt product = 1;
   int count;
   for ( count = 1; count <= n; count++ )
   {
     product *= count;
   }
   return product;
}
```

5.
```
BigInt Factorial(const BigInt & n)
// precondition: 0 <= num
// postcondition: returns n! = n*(n-1)*(n-2)*...*3*2*1
{
   BigInt product = 1;
   BigInt count;
   for ( count = 1; count <= n; count += 1 )
   {
      product *= count;
   }
   return product;
}
```

It makes little sense to compute factorials of such large numbers. 255! already has more than 500 digits. The factorial value of INT_MAX would have about 19 billion digits! This would require a book with more than 6,000,000 pages to print just one such BigInt, not to mention the 19 gigabytes of RAM that you would need just to process that one number!

6.
```
BigInt Fibonacci(int n)
// precondition:    1 <= n
// postcondition:   returns the nth Fibonacci number, Fn,
//                  where F1 = 1, F2 = 1, and Fn = Fn-1 + Fn-2.
{
   if ( n <= 2 ) return 1;

   BigInt low = 1, high = 1;
   BigInt next;
   for ( int i = 3; i <= n; i++ )
   {
         next = low + high;
         low = high;
         high = next;
   }
   return next;
}
```

AP Publications

There are a number of publications available from the AP Program; a few of them are described below. Publications can be ordered online through the AP Aisle of the College Board Online store at http://cbweb2.collegeboard.org/shopping/ or you can call AP Order Services at (609) 771-7243. American Express, VISA, and MasterCard are accepted for payment.

If you are mailing your order, send it to the Advanced Placement Program, Dept. E-22, P.O. Box 6670, Princeton, NJ 08541-6670. Payment must accompany all orders not on an institutional purchase order or credit card, and checks should be made payable to the College Board. The College Board pays fourth-class book rate (or its equivalent) postage on all prepaid orders; you should allow 4-6 weeks for delivery. Postage will be charged on all orders requiring billing and/or requesting a faster method of shipment.

Publications may be returned within 30 days of receipt if postage is prepaid and publications are in resalable condition and still in print. Unless otherwise specified, orders will be filled with the currently available edition. Prices are subject to change without notice.

- ***AP Bulletin for Students and Parents*** . . . Free. The bulletin provides a general description of the AP Program, including policies and procedures for preparing to take the exams, and registering for the AP courses. It describes each AP Exam, lists the advantages of taking the exams, describes the grade and award options available to students, and includes the upcoming exam schedule.

- ***College Explorer® PLUS*** . . . $195. This IBM-compatible software package allows users to research, review, and compare current AP policies at approximately 3,100 colleges.

- **Course Descriptions** . . . $12. Course Descriptions provide an outline of the course content, explain the kinds of skills students are expected to demonstrate in the corresponding introductory college-level course, and describe the AP Exam. They also provide sample multiple-choice questions with an answer key, as well as sample free-response questions.

  A complete set of Course Descriptions (one for each subject) is available for $100.

- **Five-year Set of Free-Response Questions** . . . $5. Each booklet contains copies of all the free-response questions from the last five exams in its subject. Collectively, the questions represent a comprehensive sampling of the concepts assessed on the exam in recent years and will give teachers plenty of materials to use for

essay-writing or problem-solving practice during the year. (If there have been any content changes to the exam in the past five years, it will be noted on the cover of the booklet.)

- **Free-Response Question booklets** . . . $12. These booklets contain one year's worth of free-response questions, taken directly from the AP Exam, along with the guidelines used to score the student responses, and samples of those responses. Teachers find the free-response questions useful for essay-writing and problem-solving practice or testing during the year. Subjects with an audio component include a cassette.

- **Guide to the Advanced Placement Program** . . . Free. Written for both administrators and AP Coordinators, this guide is divided into two sections. The first section provides general information about the AP Program, such as how to organize an AP Program, the kind of training and support that is available for AP teachers, and a look at the AP Exams and grades. The second section contains more specific details about testing procedures and policies and is intended for AP Coordinators.

- **Released Exams** . . . $20. About every four years, on a staggered schedule, the AP Program releases a complete copy (multiple-choice and free-response sections) of each exam. In addition to providing the multiple-choice questions and answers, the publication describes the process of scoring the free-response questions and includes examples of students' actual responses, the scoring standards, and commentary that explains why the responses received the scores they did.

  For each subject with a released exam, you can purchase a packet of 10 exams ($30) for use in your classroom (e.g., to simulate an AP Exam administration.)

- **Teacher's Guides** . . . $12. The guides contain syllabi developed by high school teachers currently teaching the AP course and college faculty who teach the equivalent course at their institution. Along with detailed lesson plans and innovative teaching tips, you'll find extensive lists of recommended teaching resources.

1996-97

Development Committee and Chief Faculty Consultant in Computer Science

Susan B. Horwitz, *Chair*	University of Wisconsin Madison
Theresa M. Cuprak	Carl Hayden High School Phoenix, Arizona
Donald L. Kreider	Dartmouth College Hanover, New Hampshire
Cary Matsuoka	Saratoga High School California
Christopher H. Nevison	Colgate University Hamilton, New York
Susan H. Rodger	Duke University Durham, North Carolina
Cathy L. Sauls	John Marshall High School San Antonio, Texas
Chief Faculty Consultant: Mark J. Stehlik	Carnegie Mellon University Pittsburgh, Pennsylvania

ETS Consultants:
Gail L. Chapman, Frances E. Hunt

I.N. 209123