CS/CE 224/272 - Object Oriented Programming and Design Methodologies: Assignment #2

Fall Semester 2024

Due on October 12, 2024, 11.59pm

Instructions

1. This homework consists of one large programming exercise.
2. This exercise requires you to think in terms of classes and objects and apply accordingly.
3. This exercise also requires you to learn about how to read and write data from/to text files.
4. No extensions will be given for Assignment Submission.
5. While collaboration and discussion are encouraged, this assignment is strictly individual. You may discuss the logic on paper but you may not look into anyone else's code, nor share your code with anyone else. Plagiarism and copying, whether partial or complete will result in a *zero* grade for both students, whose assignments are found to be similar. Furthermore, the plagiarism may be reported to the University Conduct Committee as well.
6. **Submission: Submission will be via Canvas LMS. Go to the Assignment 02 submission module on LMS. Name your files according to this format:** $< section > \_ < studentID > .zip$**, for example, if your section is L8 and your student ID is "12345," your submission should be named** $L8\_12345.zip$**. Upload the zip file. Any extra files submitted may lead to a grade penalty.**

Problem 1

Introduction

Imagine you're working at a cryptocurrency company responsible for maintaining a secure and decentralized blockchain network. One of the key components of blockchain technology is the cryptographic algorithms that secure transactions and validate blocks. These algorithms often involve computations with very large numbers, particularly in the areas of public-key cryptography and hashing.

As part of the process to validate a new block on the blockchain, miners must solve a complex mathematical problem—finding a hash value that meets specific criteria. This process involves computing hashes that use extremely large numbers, sometimes with hundreds or even thousands of digits. Given the nature of cryptocurrency mining, the difference between success and failure could come down to handling these large numbers efficiently.

Your task is to create a class **BigNum** to represent and manipulate these large numbers, since built-in data types cannot handle them. You need to ensure that your BigNum class can perform arithmetic operations and compare large numbers which will help you in future to verify if a solution satisfies the required conditions for blockchain validation.

Thanks to your development team lead, the initial definition of class and its member function declarations have already been provided to you in **BigNum.h**. You need to decide the data structure to represent these large numbers and then write all member function definitions of the BigNum class in a file named **BigNum.cpp**.

To store the list of digits you may use a data structure of your choice - e.g. string, vector of characters, or a linked list of characters. Alternately, you might use a dynamic array of characters. Note that in order to store a single digit, a char requires only 1 byte whereas an integer would take 4 bytes. Therefore, using char is recommended and encouraged instead of int. Using some other data structure is also permissible but **static arrays are not allowed**. **You must also have a member attribute which represents whether the number is positive or negative.**

Problem 1 continued on next page...

Description of Functions

Here's the extended description for all the functions declared in the provided BigNum.h file:

Constructors and Destructor:

1. **BigNum()**: This is the default constructor. The initial value of the object should be '0'.
2. **BigNum(const BigNum& bigNum)**: This is the copy constructor. It should create a new `BigNum` object that is a deep copy of the given `BigNum` object.
3. **BigNum(const string& numStr)**: This constructor initializes a `BigNum` object with the value represented by the string `numStr`. The string may contain very large numbers, and you should convert this string into a format that your `BigNum` class can handle.
4. **BigNum(const int num)**: This constructor initializes a `BigNum` object with the value represented by the integer `num`.
5. **~BigNum()**: This is the destructor. If you're using a dynamic array to store the digits, ensure that memory is properly deallocated to prevent memory leaks.

Input/Output Operations:

1. **void input()**: This method sets the value of the current `BigNum` object based on the input from console. Ensure proper validation of the input string and conversion to your internal format.
2. **void print()**: This method prints the value of the current `BigNum` object to the standard output. Separate groups of 3 digits by a comma(,) e.g. 1,123,456,789
3. **void inputFromFile(const string& fileName)**: This method sets the value of the current `BigNum` object based on the input from the text file having the file name passed as a parameter. Ensure proper validation of the input string and conversion to your internal format.
4. **void printToFile(const string& fileName)**: This method prints the value of the current `BigNum` object to the file having the file name passed as a parameter.

Initialization/Assignment Operations:

1. **void copy(const BigNum& bigNum)**: This function sets the object's value to the value of the object passed as a parameter. Make sure to properly handle self-assignment and memory management if using dynamic memory.
   e.g. a.copy(b); where both a and b are BigNum objects.
2. **void operator=(const BigNum& bigNum)**: This is the assignment operator. It should assign the value of one `BigNum` object to another. Make sure to properly handle self-assignment (i.e. a=a) and memory management if using dynamic memory.
   e.g. a = b; where both a and b are BigNum objects.
3. **void zerofy()**: This function initializes the current value of BigNum object to 0.
4. **void clear()**: This will be called in order to delete the current dynamic array if you are using it. Must be called inside destructor, input methods and as required. Note that this is a private function and not part of the interface of the BigNum class. Note that this just deletes any dynamic memory allocation and does not create a new one. One of the uses of this would be if let's say we call input() and then inputFromFile(), so you must call clear() inside inputFromFile() so that you delete the current memory and create a new one in order to save the new BigNum read from the file.

Problem 1 continued on next page...

Arithmetic Operations:

Addition:

1. **void increment()**: This function should implement the increment operation of a `BigNum` object. Increment the value of the object by 1 and then return the result as a new `BigNum` object. In other words, this function should implement a++;
   e.g. a.increment(); where a is a BigNum object.

2. **BigNum add(const BigNum& other)**: This function should implement the addition of two `BigNum` objects. Add the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a + b;
   e.g. BigNum c = a.add(b); where both a and b are BigNum objects.

3. **BigNum add(const int other)**: This function should implement the addition of a `BigNum` object and an integer. Add the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a + b;
   e.g. BigNum c = a.add(b); where a is a BigNum object and b is an int.

4. **void compoundAdd(const BigNum& bigNum)**: This function should implement the addition of two `BigNum` objects. Add the 2 values and save the result in the object calling the function. In other words, this function should implement a = a + b; or a += b;
   e.g. a.compoundAdd(b); where both a and b are BigNum objects.

5. **void compoundAdd(const int num)**: This function should implement the addition of a `BigNum` object and an integer. Subtract the 2 values and save the result in the object calling the function. In other words, this function should implement a = a + b; or a += b;
   e.g. a.compoundAdd(b); where a is a BigNum object and b is an int.

Subtraction:

1. **void decrement()**: This function should implement the decrement operation of a `BigNum` object. Decrement the value of the object by 1 and then return the result as a new `BigNum` object. In other words, this function should implement a–;
   e.g. a.decrement(); where a is a BigNum object.

2. **BigNum subtract(const BigNum& other)**: This function should implement the subtraction of two `BigNum` objects. Subtract the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a - b;
   e.g. BigNum c = a.subtract(b); where both a and b are BigNum objects.

3. **BigNum subtract(const int other)**: This function should implement the subtraction of a `BigNum` object and an integer. Subtract the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a - b;
   e.g. BigNum c = a.subtract(b); where a is a BigNum object and b is an int.

4. **void compoundSubtract(const BigNum& bigNum)**: This function should implement the subtraction of two `BigNum` objects. Subtract the 2 values and save the result in the object calling the function. In other words, this function should implement a = a - b; or a -= b;
   e.g. a.compoundSubtract(b); where both a and b are BigNum objects.

5. **void compoundSubtract(const int num)**: This function should implement the subtraction of a `BigNum` object and an integer. Subtract the 2 values and save the result in the object calling the function. In other words, this function should implement a = a - b; or a -= b;
   e.g. a.compoundSubtract(b); where a is a BigNum object and b is an int.

Multiplication and Division:

1. **BigNum multiply(const BigNum& other)**: This function should implement the multiplication of two `BigNum` objects. Multiply the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a * b;
   e.g. BigNum c = a.multiply(b); where both a and b are BigNum objects.

2. **BigNum div(const BigNum& other)**: This operator should implement integer division of two `BigNum` objects. Make sure to handle cases where division by zero might occur. In this case, just output the error message, "Invalid operation" to the console. Divide the 2 values and return the result as a new `BigNum` object. In other words, this function should implement c = a / b;
   e.g. BigNum c = a.div(b); where both a and b are BigNum objects.

3. **BigNum mod(const BigNum& other)**: This operator should implement the remainder operation between two `BigNum` objects. Make sure to handle cases where division by zero might occur. In this case, just output the error message, "Invalid operation" to the console. Find the remainder and return the result as a new `BigNum` object. In other words, this function should implement c = a % b;
   e.g. BigNum c = a.mod(b); where both a and b are BigNum objects.

Comparison Operations:

1. **bool equals(const BigNum& other)**: This operator should return `true` if the two `BigNum` objects are equal, otherwise return `false`.

2. **bool notEquals(const BigNum& other)**: This operator should return `true` if the two `BigNum` objects are not equal, otherwise return `false`.

3. **bool lessThan(const BigNum& other)**: This operator should compare two `BigNum` objects and return `true` if the current object is less than the other.

4. **bool greaterThan(const BigNum& other)**: This operator should compare two `BigNum` objects and return `true` if the current object is greater than the other.

Examples

Example 1

num1 = 111111111111111111111111
num2 = 111111111111111111111111
Addition: 2,222,222,222,222,222,222,222,222
Subtraction: 0
Multiplication: 1,234,567,901,234,567,901,234,567,654,320,987,654,320,987,654,321
Div: 1
Mod: 0

Example 2

num1 = 98765432198765432198765432198765432 1
num2 = 12345678912345678912345678912345678 9
Addition: 1,111,111,111,111,111,111,111,111,111,111,111,111,110
Subtraction: 864,197,532,864,197,532,864,197,532,864,197,532
Multiplication: 121,932,631,356,500,531,591,068,431,825,636,331,816,338,969,581,771,069,347,203,169,112,635,269

Div: 8
Mod: 9,000,000,009,000,000,009,000,000,009

More Examples

You can further test your code using the BigNum calculator here:
`https://www.calculator.net/big-number-calculator.html`. Set Precision to 1 to ensure integer operations.

Important Notes

1. The member **function declarations/prototypes** should be placed in **BigNum.h** (provided to you). You may add your own private functions if you want but do not add any public functions.
2. All member function definitions of BigNum class should be implemented in **BigNum.cpp**.
3. Use a separate driver file (**BigNumDriver.cpp**) for testing your implementation - do not include main() in BigNum.cpp. For grading, we will use our own driver files to test the functionality that you implement in BigNum.cpp.
4. In addition to standard input/output, you must input/output using plain text files. For reference, you may consult Chapter 12 (Pages 584–591) of the book "Object Oriented Programming in C++", 4th Edition by Robert Lafore.
5. It is recommended to use data validation at every step, since incorrect data entry might cause the program to crash e.g. you must validate that the entered characters are valid digits.
6. You should implement appropriate memory management, especially if you're using dynamic memory (e.g. dynamic arrays or linked lists). Make sure there are no memory leaks or dangling pointers in your code. Implement the Rule of 3 if you choose to implement data structures using dynamic memory.
7. Handle special corner cases such as division by zero; first operand being less than the right e.g. in division, subtraction, remainder operations; numbers with leading zeros; etc.
8. Use proper naming convention for all variable and function names - either use snake_case or use camelCase for function/variable names. Make sure to put appropriate code comments where you feel they are required. However, header comments are mandatory - include your name, ID and Section as comments at the top of your file.
9. Avoid use of global variables.

Points Distribution

Please make sure your code compiles. 50% grade penalty if your code does not compile. To test whether your code compiles or not, you may compile your code with the given **BigNumDriver.cpp** file. The following are the points distribution for each component of the assignment:

Component	**Points**
30 Functions (3 points each)	90
Comments/ Variable Naming/ Indentation	10
TOTAL	100