LABS 1&2 - Operations with Huge Integers

Assessment: 10% of the total course mark.

Instructions:

- This assignment will span Labs 1 and 2.
- By the end of the Lab 1 session you must demonstrate to your TA the following parts: (i) the constructors of `HugeInteger` class and (ii) the addition operation for positive integers.
- The entire solution must be demonstrated by the end of the Lab 2 session.
- The electronic submission on Avenue to Learn of all `C++` source code must to be done by 5:20 pm the day of your lab session. Specifically, you have to submit the `C++` code for your `HugeInteger` class, for the test class and for any other classes you may need in your implementation for both labs 1 and 2. The code for each class must be submitted in a separate TEXT file (e.g., "HugeInteger.h.txt" and "HugeInteger.cpp.txt").
- The report (in PDF) must be submitted by **11:59 pm on March 6, 2021**.
- A **bonus of** 2% **of the course mark** will be awarded if the asymptotic time complexity of your implementation of the multiplication of two integers of at most n digits is $o(n^2)$ (i.e., subquadratic). A description of the algorithm and an evaluation of the running time are additionally needed for the bonus to be awarded.

Description: The range of integers that can be represented in `C++` using a primitive data type is only from -2^{63} to $2^{63}-1$. What if we need to manipulate integer values beyond this range?

In this assignment you will write a `HugeInteger` class which is able to represent arbitrarily large integer numbers. This class must implement arithmetic operations on integers such as addition, subtraction, multiplication and comparison. You have to implement this class without using `C++` predefined classes, unless specified otherwise.

Additionally, you have to measure experimentally the running times of the operations implemented in your `HugeInteger` class and compare them with the measured running times of the corresponding operations provided by `boost::multiprecision::cpp_int` class.

Specifications:

The class `HugeInteger` must contain at least the following **public** methods:

1) `HugeInteger add(const HugeInteger& h)`: Returns a new `HugeInteger` representing the sum of `this HugeInteger` and `h`.

2) `HugeInteger subtract(const HugeInteger& h)`: Returns a new `HugeInteger` representing the difference between `this HugeInteger` and `h`.
3) `HugeInteger multiply(const HugeInteger& h)`: Returns a new `HugeInteger` representing the product of `this HugeInteger` and `h`. This method **should not** be implemented as repeated addition (calculate m*n by adding m, n time) since it's too slow.
4) `int compareTo(const HugeInteger& h)`: Returns -1 if `this HugeInteger` is less than `h`, 1 if `this HugeInteger` is larger than `h`, and 0 if `this HugeInteger` is equal to `h`.
5) `std::string toString()`: Returns a string representing the sequence of digits corresponding to the decimal representation of `this HugeInteger`. Please make sure this method **works in both Lab 1&2**.

The class `HugeInteger` must contain at least the following **public** constructors:

1) `HugeInteger(const std::string& val)` creates a `HugeInteger` from the decimal `string` representation `val`. The string contains an optional minus sign at the beginning followed by one or more decimal digits. No other characters are allowed in the string.
2) `HugeInteger(int n)` creates a random `HugeInteger` of n digits, the first digit being different from 0; n must be larger or equal to 1.

Each constructor must **throw an exception** if the argument passed to the constructor does not comply to the specifications. In your solution, you may use `C++` API methods for string manipulation and for pseudo-random number generation. You can also use **std::vector** if necessary.

Theory and Experiment:

Theory

You have to compute the amount of memory (in bytes) required to store an integer of n decimal digits by using your `HugeInteger` class.

For each of the operations on huge integers (i.e., comparison, addition, subtraction, multiplication) you are required to compute a big-Theta estimate of the worst-case and of the average-case running times and of the amount of extra memory allocated during the execution, as a function of the number of digits n in the decimal representation. Extra memory refers to the memory needed during the method execution apart from the memory used to store the input numbers and the result.

Experiment

You are also required to measure the running time of each of the operations (i.e., comparison, addition, subtraction and multiplication) using your implementation of the `HugeInteger` class as well as using the implementation provided by the `Boost` libraries as the `boost::multiprecision::cpp_int` class. Write a function `HugeIntTiming()` to estimate the running time of each operation. For fixed n this program should instantiate

random integers of size n (i.e., of n decimal digits) and should then run each of the operations on the integers measuring the amount of time required to perform each operation. To measure the running time the `std::chrono::system_clock::now()` (since `C++11`) method can be used, which returns the current time (can be converted to milliseconds) since GMT midnight January 1, 1970. In order to have reasonable accuracy in the measurement, you should run each operation many times on a given pair of integers. Also you should run the measurements on at least 100 different pairs of random integers. For example, to measure the running time of the addition operation in class `HugeInteger` you may use the following code:

```
...
#include <chrono>
using namespace std::chrono;
...
system_clock::time_point startTime, endTime;
double runTime=0.0;
double durationMs=0.0;
int n = 500;
for (int numInts=0; numInts < MAXNUMINTS; numInts++){
    HugeInteger huge1(n);        //creates a random integer of n digits
    HugeInteger huge2(n);        //creates a random integer of n digits
    startTime = system_clock::now();
    for(int numRun=0; numRun < MAXRUN; numRun++){
       HugeInteger huge3 = huge1.add(huge2);
    }
    endTime = system_clock::now();
    durationMs = (duration<double, std::milli>(endTime - startTime)).count();
    runTime += durationMs / ((double)MAXRUN);
}
runTime = runTime/((double)MAXNUMINTS);
...
```

You should set `MAXRUN` such that for each run the value of `durationMs` is at least 500, and set `MAXNUMINTS` to be at least 100. If you notice that the data is noisy, you may want to increase these limits.

For each of the operations (comparison, addition, subtraction, multiplication) measure the running time for various values of n, for instance, $n = 10, 100, 500, 1000, 5000, 10000$.

Report Description: The report must contain the following sections:

Description of Data Structures and Algorithms

- Describe the data structure used to implement the `HugeInteger` class and how you implemented each operation (comparison, addition, subtraction, multiplication). In other words, describe how the integer is stored, what additional variables you need to maintain, etc. Describe in detail the algorithm used to implement each

operation. Include figures to illustrate your algorithms (for example, show the data structure before, in the middle and after the execution of each method).

Theoretical Analysis of Running Time and Memory Requirement

- You also have to compute the amount of memory (in bytes) required to store an integer of n decimal digits using your `HugeInteger` class. You should be able to derive a formula for the number of bytes required to store your integer, as a function of the number of decimal digits n and plot it in a figure.
- Using big-Theta notation, compute the running time (in the worst case and in the average case) and the amount of extra memory requirement for each of the operations (addition, subtraction, comparison, multiplication). Include a detailed description of how you arrived at these results.

Test Procedure

- Design a test procedure and justify that it demonstrates the complete functionality of your `HugeInteger` class.
- What are the possible cases that can arise? What inputs will you use to test that all the special cases work correctly? What inputs will you use to ensure each of the required operations are functioning properly?
- Do your outputs meet the specifications?
- Did you have any difficulties in debugging the code?
- Are there any input conditions that you could not check?

Experimental Measurement, Comparison and Discussion

- Describe how you measured the running time for each operation. Provide the values of the parameters used in your program.
- For each of the four operations (comparison, addition, subtraction, multiplication) include a table with the measured running times using your implementation. In the same table include the running time of the corresponding operation using the `boost::multiprecision::cpp_int` class, and a scaled version of the theoretical running times. You must have measurements for various values of n, where n is the number of digits in each of the two integers.
- For each of the four operations include a plot of all three sets of running time measurement data on the same axes.
- Document any problems you had in obtaining the experimental data.

Discussion of Results and Comparison

- How well do your theoretical calculations correspond to your measured running times? Do your experimental results make sense? Explain why or why not.
- How does your implementation compare with the `boost::multiprecision::cpp_int` class in terms of running time for each operation?

- Describe any improvements you would make given extra time, which would improve the running time/memory complexity of your program.

NOTE: If you get inspiration from other sources you *must* acknowledge them in the report. If you discuss the solution with your friends should also acknowledge this and provide their names. You are not allowed to copy portions of the code or portions of the report from anywhere. The source code and the report should be your own original work.

USING BOOST LIBRARY: Use `boost::multiprecision::cpp_int` in your `C++` project (only in the test code).

- Download the source code from `https://www.boost.org/users/download/`
- Unpack it to a local path, e.g., the source code path is C:\boost_1_78_0
- Add the path to your compiler's include paths, e.g., as in Figure 1 for Eclipse IDE.
- Use `boost::multiprecision::cpp_int` in your code, e.g.,

```
...
#include <boost/multiprecision/cpp_int.hpp>
using namespace boost::multiprecision;
...
cpp_int i("123");
cpp_int j,k;
j.assign("456");
k = i + j;
std::cout << "k = " << k << std::endl;
...
```

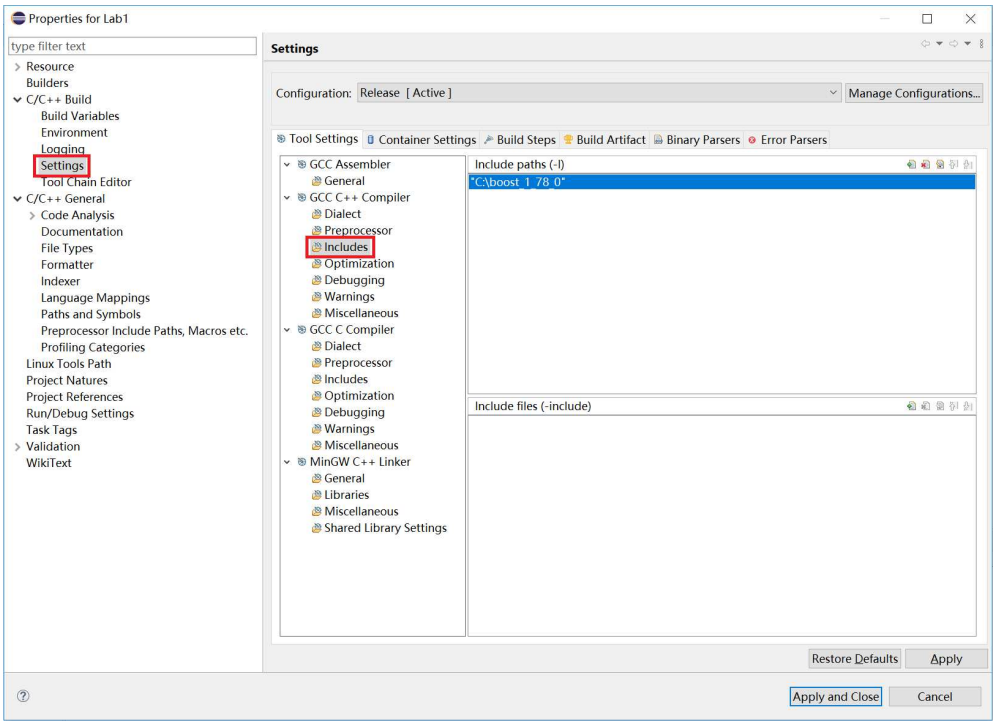

Figure 1: Adding boost path in Eclipse