CS4414 Fall 2020 Homework 1 (revised Sept 6)

Assigned Sept 8, 2020. Due Sept 18, 2020 via CMS upload. Late submissions accepted until Sept 22 (but we deduct 1pt per day late). Mandatory twice per week CMS "checkins"! See below!

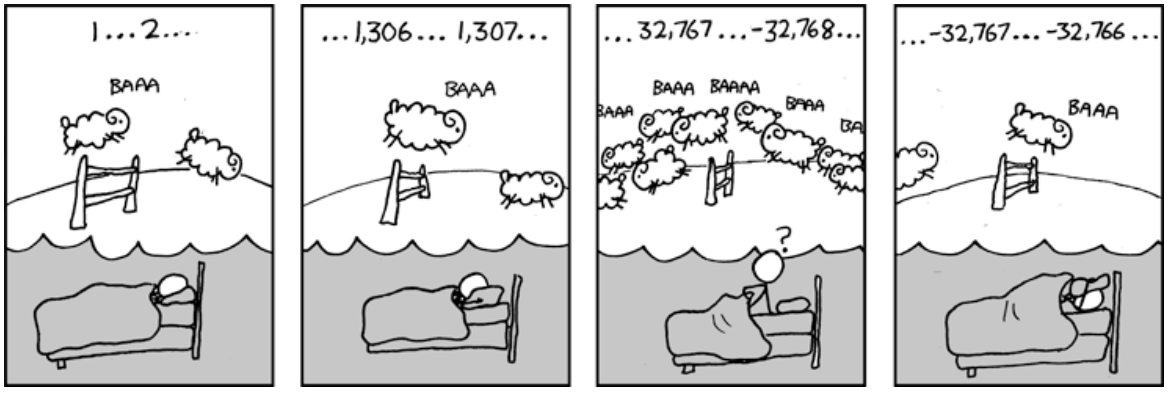
Goals: Homework 1 is a pure C++ assignment, to be done on a Linux system (it can be your own computer, or a CSUG Lab or MEng Lab machine, or even on Windows by telling the system to install the "Windows Subsystem for Linux" extension. If you do this, be sure to follow the Microsoft instructions – don't be fooled by "copy cat" web sites, because they might be trying to trick you into installing malware). Homework 1 is intended as a simple but real task, to help you gain hands-on experience using Linux commands and C++ as a programming language. It is very important to do this assignment on your own: With too much help you'll solve the problem, yet you won't build up C++ proficiency.

We plan to automate grading in CS4414, as much as possible (not every aspect can be done by an autograder, but many parts can be automated). Accordingly, please make sure your program takes input *exactly* as shown below, and that the output looks *exactly* like the output in our examples – don't change minor things, even aspects like formatting of the output lines. That way, when our computerized grading script runs your program with particular inputs, we'll know what the outputs should be, and can automatically check.

The task:

Students who have worked primarily in Python become so comfortable with its support for arbitrary-length integers that it is easy to forget that within the machine itself, every integer is represented by a field with some fixed number of bits: this could be 8, 16, 32, 64 or even 128. One of those bits (the "high order" bit) is reserved to represent the sign: 0 for a positive number, and 1 for a negative number, although most languages also support "unsigned int" data types in which there is no sign bit. A notorious problem, in fact, is that if an integer becomes too large, it spills over into the sign bit and the computer will begin to interpret the number as a negative one (hence the XKCD comic, below).

Homework 1 will create a "big numbers" arithmetic package for integers represented as vectors of individual digits, employing "elementary school" mathematics for operations such as + - * / % and gcd. **Bignum** versions of these and a few other operations are used when performing encryption.



CS4414 Fall 2020 Homework 1 (revised Sept 6)

We need a **Bignum** solution because keys can easily be larger than any of these limits. For example, many cryptographic tools use the "Rivest, Shamir and Adelman" (RSA) cryptographic tools. RSA keys are usually at least 256 bits in length, and some systems use keys that are even longer.

For homework 1, you'll create your own Bignum class, using the C++ vector class to hold the numbers (a std::vector is a variable length array). To keep things as simple as possible, each vector element will hold a single (non-negative) digit. We don't need to deal with negative numbers in this homework: RSA and similar packages are defined purely on non-negative integers.

Most Linux programs center on a main method that accepts Linux command-line arguments, performs the requested operation, prints the answer (a big number) and then terminates. This is how your homework 1 will work. Our autograder will work by compiling and running your program on our machine many times with various test arguments to see whether it works correctly according to our specification.

Below, we'll offer some examples you could test to confirm that your code is working (and you can easily create new test cases, beyond the ones we provide). As you will see, our test cases ask your solution to perform a series of simple operations, each time running your program separately for that particular test.

Long ago, Euclid devised what we now call the Euclidian algorithm for finding the greatest common divisor: given two integers, this is the largest integer that exactly divides both of them (for example, the GCD of 21 and 30 is 3; the GCD of 17 and 18 is 1). The algorithm is very simple and elegant:

```
int gcd(int a, int b) {
  if (b == 0)
     return a;
  return gcd(b, a % b);
}
```

Notice that GCD uses recursion and involves a series of remainder operations. This could reveal issues that a single operation might not, such as data structures that were left in a damaged state.

Thinking ahead: Just to let you see where we plan to take this, for homework assignment 2 we will be working on understanding the performance of a small application Ken built on his solution to assignment 1. His homework 2 basis (which we will give you) includes one additional operation: modular exponentiation, which involves first exponentiating a number to a given degree, and then taking the remainder modulo a third parameter. Although your humble professor prides himself on building the world's fastest code, in this particular case, his homework 2 solution is not fully optimized, leaving some tasks that we'll ask you to complete. The task used to evaluate homework 2 will center on encrypting or decrypting files line by line using RSA, a situation in which any performance issues are amplified because the various operations are called again and again. But Ken also left in some other kinds of issues that aren't purely "computational" in nature. This will give you a chance to use various tools we plan to teach you in recitation to understand and optimize the solution we provide You are welcome to also try homework 2 with your own homework 1, for fun, even though what you hand in for homework 2 will need to be based on our given version, to ensure a level playing field.

CS4414 Fall 2020 Homework 1 (revised Sept 6)

Before getting to the details of what you should implement, we need to say a few words about academic integrity. We will be checking for many kinds of evidence of cheating, and this raises a practical issue of whether or not it is a violation of academic integrity to take a standard example of how some sort of data structure should be used, and then base your own solution on that.

In fact, using recommended solutions, from the C++ "standard" site, is a recommended and common way to learn to use built-in types, such as std::vector, which we will use in Homework 1. But this could be detected as a possible academic integrity issue by the tools we use. Accordingly, please be sure to include a comment telling us that "the following method was obtained from xxxx" if you include unmodified code from elsewhere, particularly if the example is at all lengthy. You do *not* need to worry about this for situations where the example is just one or two lines of code, but it course arise if you work with an entire method obtained from some web site.

In contrast to the C++ standards (which is a whole site dedicated to defining C++ 17 and correct use recommendations), or the textbook we recommended for learning C++, we might take issue with solutions obtained from StackOverflow.com, CourseHero, or other sites where you pose a question and people respond by solving your problem for you, and provide the code. Even if their advice was free, asking for that kind of help violates the rules.

Help! We offer many ways to get help. Just reach out to Ken, or to any of the TAs. Post a question on Piazza. If your concern centers on C++ complaining about a line of code, you can even post that line of code and the first few error messages it printed. You can even ask a friend to help you understand something, such as what a C++ error message means. But don't post your whole program, or share it with a friend, and if you get help, that doesn't include having your friend take the keyboard and edit your code for you.

Detailed assignment.

Since this is your first C++ experience, we'll break the assignment down into simple steps, which we recommend completing and testing one by one in the given order.

Main procedure.

You will be coding a program that takes arguments when it is run and complains if the arguments are not provided. Start by implementing the main procedure and testing that it can handle all of these cases, but printing 0 for the answers.

Your program will be used as in this example:

```
ken@kenM6800:  bignum + 1 2
3
```

Here, we see the bash command-line prompt for Ken's computer (ken@kenM6800:). That was printed by bash to tell Ken that bash was waiting to know what program to execute next. Then we see the program name (the name of the file holding the compiled code): "**bignum**". This example assumes that the **bash PATH** variable includes the current directory, ".". Ken did that in his .**bashrc** file. Had he *not* done this, **bash** wouldn't find **bignum** without being told where to look, as in "**./bignum**". Then we see the operation requested ("add", represented by +) and the two numbers to add. The **bignum** program printed the answer: 3.

CS4414 Fall 2020 Homework 1 (revised Sept 6)

We will use this same approach for +, -, *, /, % (remainder) and gcd (greatest common divisor). One important comment: In bash, the character * has a special meaning: it means "all files in the current directory." Thus if we try to pass a * to **bignum**, bash will instead do a substitution, replacing the * with a list of files. To prevent this behavior, you will need to put a backslash before the *, as in *. In fact, there are a number of special bash characters where this can arise (others include %, >, <, !, ^, &, |, $).

1. We recommend that you create a folder called **bignum** and use **cd bignum**. **cd ~** will take you back to your home directory.
2. We will implement our code in three files: main.cpp, bignum.cpp and bignum.hpp. Notice that these all have lower-case names, even though when we define the **Bignum** class, it will have an upper-case name (this is standard for class names in C++). The reason is that some operating systems don't distinguish between upper-case and lower-case file names. Code is more portable if all files simply use lower-case, so this is the approach we are adopting in CS4414.
3. In what follows, we assume that the name of the compiled program is **bignum**. C++ has a -o argument let you name the output; otherwise, it would be named **a.out**.
4. If executed with the wrong number of arguments, bignum should complain, as follows (in these examples we won't show the bash prompt):

   bignum
   Usage: bignum op number1 number2.

   bignum + 1 2 3 4 5
   Usage: bignum op number1 number2.

5. If executed with an argument that should be an integer but that has other characters in it (or if given a negative integer) print an error message like this. Notice that the error message includes quote characters! Much like for the * operator, C++ understands quote characters and you will need to put a backslash in front of a quote if you want it to just be treated as part of a string for these error messages.

   bignum + -28221110 carwash
   Error: "-28221110" is not an unsigned integer.
   Error: "carwash" is not an unsigned integer.
   bignum + 1.8221110 2000
   Error: "1.8221110" is not an unsigned integer.
   bignum + 18221110 carwash
   Error: "carwash" is not an unsigned integer.
   bignum + 1.8221110 carwash
   Error: "1.8221110" is not an unsigned integer.
   Error: "carwash" is not an unsigned integer.

   Notice that for step 5, you need to scan the integer arguments to check that they only contain digits 0-9. This involves writing a loop in C++ to look at the corresponding string, character by character. You'll find it easiest to turn the argument passed by Linux into a std::string, which will let you use the rich collection of operations available on strings in C++.
6. If executed with an invalid operator, it should print this:

bignum \# 18221110 55771
Error: # is not a supported operator.
If the numbers and the operator are incorrect, we won't reach this point because we would already have printed error messages about the numbers, and then terminated (the operator check should occur after the numbers are checked). The backslash is needed to tell bash that the argument should be treated as text (you can also put the # in quotes)

7. When implementing your main procedure, please be careful to read the documentation about C++ main procedures in Linux. You will see that main is called by bash with an argument count and an argument vector. The count is just what it sounds like: the number of arguments. However, and this can be a surprise for some people, there is an additional "first" argument: the name by which the program was launched. Thus if you run **bignum** with 3 arguments the count will actually be 4, and the first of the character strings will just be a char* object containing the string **bignum**, null-terminated. Ignore this argument.

   We mentioned that in C++, it is easiest to convert char* to std::string. You'll do that this way:

```
int main(int argc, char** argv)
{;
        std::string args[argc] ;
        for(int n = 0 ; n < argc; n++)
                args[n] = std::string(argv[n]);
}
```

   If you do this, argc will be the same as before, and args will be an array of strings containing the arguments.

   We recommended that you first implement main, then tackle the **Bignum** class. We won't actually test this case, because it will "vanish" when you tackle the next steps, but it may be a good idea to test that you've correctly obtained the three arguments by printing them:

   bignum + 18221110 2000
   The bignum program was run with operator '+' and arguments 18221110 and 2000.

   Notice that at this stage, the numbers are still in std::string objects, and are not yet "**Bignum** objects." The operator is also a string, but we have shown it in single quotes. C++ treats the single-quote in a special way too, so you would need a backslash to print it in a message with this exact format, just like for the double quote in step 6.

Bignum class specification.

Once you have completed your implementation of the main procedure, you will edit **bignum.hpp** and create a specification for the **Bignum** class. This specification will not have implementations of any methods, but it will give the types of the methods and their arguments.

CS4414 Fall 2020 Homework 1 (revised Sept 6)

The class name will be **Bignum** (note the capital letter: by convention, type names associated with user-defined classes start with a capital letter, whereas methods and variables start with lower case letters). A **Bignum** will be a C++ object containing a value field, which will be a std::vector of "int" objects, holding one int per digit (thus, we will only store values in the range 0 to 9 into these digits).

When we created our sample solution, we adopted the standard rule for base-10 numbers, namely that a **Bignum** would either be 0, or would have a first digit that is non-zero. Oddly, this turns out to be harder to implement than we expected, because it means that when constructing a **Bignum** (for example, the result of a subtraction operation), you must be careful with 0s. To see this, consider 1234-1234. We want this to be equal to 0, not 0000. Yet digit by digit, you do get 0000 as you subtract.

We recommend that you include a **Bignum::check** method that checks a **Bignum** and complains if something is wrong with it, like extra 0's, or digits other than 0-9 in the vector. Use this while debugging.

Please read about the std::vector data type. In order to make use of it, you will need to add an "include" to the top of your bignum.hpp file. You need to figure out what this include should look like. Then you can use std::vector.

Any object needs to be initialized, and in C++ this is done by a method called a "constructor". Your **Bignum** class should define two constructors. One would take a std::string as its argument, and create a **Bignum** holding the corresponding data. The second constructor will have no argument, and initialize the **Bignum** to hold a single-digit number, 0.

The **Bignum** class will define a set of methods for the operators. Some students might find it easier to work with methods that have names, like **add, sub, mul, div, rem** and **gcd**. But the more modern approach in C++ would be to "overload" the binary operators: +, -, *, / and %. In this case you still write a method, but rather than a name, it gets invoked by the compiler when it sees two **Bignum** objects being added, subtracted, etc. In writing our version of this, we found it useful to also implement operators <, > and ==, and we added a helper method that multiples a Bignum by a single digit.

Finally, **Bignum** should support a method **to_string** that will convert the **Bignum** to a std::string. It would have no arguments (meaning, to call it you would just write **xxx. to_string()**), and it would convert the integer represented by xxx to an object of type std::string, for use in printing. Hint: convert the first digit to a string, then append digit by digit to build up the entire number.

Hint: be careful not to end up with a number that includes a bunch of leading 0's. We want the result of

bignum * 123 0

to be 0, not 000. This is actually trickier than it sounds… You will need to learn about the std::vector methods size(), push_back(), begin() and emplace(): xxx.push_back(item) is used to append an extra item to a vector, making it longer. xxx.emplace(xxx.begin(), item) adds an item at the front. You can also access a particular item as you would with a fixed length array: xxx[idx] is the item at offset idx.

Bignum class implementation, Main Procedure

Next, edit bignum.cpp and implement the **Bignum** class. You will need to write the code corresponding to each of the methods defined in bignum.hpp. vSubtraction can result in negative numbers, but we will

not be working with negative numbers at this time. For this reason, if sub is called with A and B, but B is larger than A, print "Unsupported: Negative number" on a line by itself, and then return **Bignum** 0.

You will find it relatively easy to implement + and -, so tackle those first, and test them, including the case where a – operation would generate a negative number. Be sure to test carries for +, borrows for -, and check cases that could leave a leading 0 to make sure your code to inhibit leading 0's works. Subtract some numbers x from themselves and make sure this is always 0. To test your solution, you will have to revisit your main.cpp file to add the code to connect the various arguments to your **Bignum** package.

In our sample solution, we implemented a bunch of comparison operators like <, >, ==, <=, >=. Perhaps you won't need them, but if you plan to do as we did, implement these next.

The next thing we did was a bit tricky but made * much easier: we implemented Bignum * int for any small int, like 7 or 10. Even this case has carries, but once you have this helper method, it is much easier to build the "real" multiplication for Bignum * Bignum.

And guess what? Now you are ready to tackle that case!. Use the standard grade-school approach for multiplication, but leverage your Bignum * int and Bignum + Bignum code: it works, so why not use it? Test Bignum multiplication carefully before you move on to the next operations. When learning C++, it really is best to write a little code, test it, write a little more code, test again. If you write a lot of code, you maybe overwhelmed by too many compiler errors (until you learn the syntax, tiny syntax errors cause almost comical numbers of complaints from C++), and worse, huge numbers of bugs that could just be confusing and depressing. By taking one small step, successfully, you will gain skills to use in the next step.

You'll find that / is surprisingly similar to * if you use a grade-school approach: to compute A/B, align B under a prefix of A that is at least as large as B. Now you can use repeated subtraction until the remainder is smaller than B. Then, shift to the right and consume additional digits of A. At the end of this procedure you will have computed both the dividend and the remainder. When testing, use lots of print statements, and really look closely at the output. Fix the "first problem" you see in the sequence before tackling others. Often, in C++, once something goes wrong, the issues snowball. Fixing that first issue might fix many others that show up in later print statements.

Next it will be time to implement %. One option is to implement the / algorithm as a helper method, in which case you can use it to compute A/B and A%B depending on how it is called. A second is to compute A%B by computing (A– (A/B)*B)). Notice that we are not really focused on performance in Homework 1. We care much more about correctness and clarity of your code.

Comments

In many introductory programming courses, you are taught to be obsessive about providing comments on everything. CS4414 is not an introductory course, and we favor a more professional approach. Each class should have a brief self-explanatory comment indicating what the class is for, but methods and other logic shouldn't have comments unless the logic is complicated or subtle. Think of a comment as a reminder to yourself: include a comment if you would find that comment helpful next time you look at this code (or a year from now, if you revisit it), but don't view comments as a kind of busy-work that has

no real value. We are only interested in "substantive" comments! **But do include a comment in each source file giving your name and your net-id.**

Test your code!

Test that your package is really working, for example:

```
% bignum + 18221110 2000
18223110
% bignum - 18221110 2000
18219110
% bignum – 2000 18221110
Unsupported: Negative number.
0
% bignum * 18221110 2000
36442220000
% bignum / 18221110 2000
9110
% bignum / 18221110 0
Error: Divide by zero.
0
% bignum % 18221110 2000
1110
% bignum * 1771919 779351
1380946844569
% bignum * 1771919 329351
583583294569
```

These are not enough cases to really evaluate your solution! You definitely need to add more test cases of your own, focusing on things like carry (in addition) and borrow (in subtraction). Because GCD depends on %, don't test GCD until % is definitely working. But then:

```
% bignum gcd 18221110 2000
10
% bignum gcd 1879817719191 61616718113
1
% bignum gcd 1380946844569 1771919
1771919
% bignum gcd 1380946844569 779351
779351
% bignum gcd 1771919 1380946844569
1771919
% bignum gcd 779351 1380946844569
779351
% bignum gcd 779351 1771919
1
```

CS4414 Fall 2020 Homework 1 (revised Sept 6)

% bignum gcd 1380946844569 583583294569
1771919

This homework will take a while!

We want you to work through the C++ startup jitters and become proficient, which is why we are giving you as much as 3 weeks for this assignment. But it will use up a few days in that period! So start early. As a special incentive, we are requiring progress reports twice each week. There is a way to skip a report… but not two in a row. In these reports you will (1) upload your code. So no excuses: you need to have code to upload! (2) answer questions about it, such as whether it has the main method, whether it compiles, whether it has some of the Bignum methods, whether it has all of them, etc. The role of these mid-way checkins is for us to reach out to you if you seem stuck. They are required, and we will deduct points if you skip the checkins and yet have not uploaded a working solution.

Performance

We said we won't test performance of your solution, but we were a tiny bit dishonest! In fact, we have a sample implementation of **Bignum**, created exactly as described above. We timed our solution and found that for the examples shown above, no run required more than 0.1s. Accordingly, our grading script has a 1s timer (10x larger than our program requires). Any operation that takes longer than 1s will be deemed to have gone into an infinite loop. If your program is too slow, talk to a TA for help understanding the bug (anything 10x slower than our version certainly is buggy in some strange way! After all, a computer can perform 1B operations per second, so 0.1s would be time for it to run 100M instructions. In fact a lot of time is just spent starting up, but even so… if your program runs for this long, it is really doing something very peculiar!)

Submission

You will be submitting a tar file containing (only) your 3 source-code files (main.cpp, bignum.cpp, bignum.hpp); our testing script will recompile the program on our machine and then run it. We compile with the following command, and it should not give any warnings or errors:

% g++ -std=c++17 -Wall -Wpedantic -o bignum main.cpp bignum.cpp

The use of g++ 17 is a slight shift from our course web pages, which said we would be using g++ 11. Now that g++ 20 is out in an early release form, most companies are switching to the stable g++ 17 libraries and compilers, and we might as well go with that trend. Anyhow, Sagar uses a lot of g++ 17 functionality. You have permission to use -O3 if you like, but keep in mind that heavy optimization scrambles debugging information, hence if you use -g (for gdb symbols) don't optimize. If you optimize, particularly at the O3 level, don't expect gdb to be very helpful!

Grading

Our auto-grader will check that your program compiles and executes correctly. This automated grade will then be augmented by a human review of your code. If our grader needs to "fix" something in your program to get our autograder to compile and run it, we will deduct points. We deduct for endless comments that don't tell the reader anything important about a complex or subtle aspect of the code.

CS4414 Fall 2020 Homework 1 (revised Sept 6)

Late Assignments

Try to finish on time. We automatically allow up to one extra week, at a price of 1% of the maximum score per day.

Partial Credit

We plan to award generous partial credit even for people who struggle on this first assignment or who can't get part of it finished, so please don't panic if you find this hard! C++ and Linux are a lot to learn quickly, and we realize that. As you become more comfortable with C++ and Linux, things become easier and more natural.

Regrades

You can request a regrade via CMS. We are very responsive on factual issues, such as "my program was correct, but because there was an extra blank space at the start of each line, your auto-grader gave me zero on every part." Asking us to "please just take another look and give me more points" probably won't result in more points.