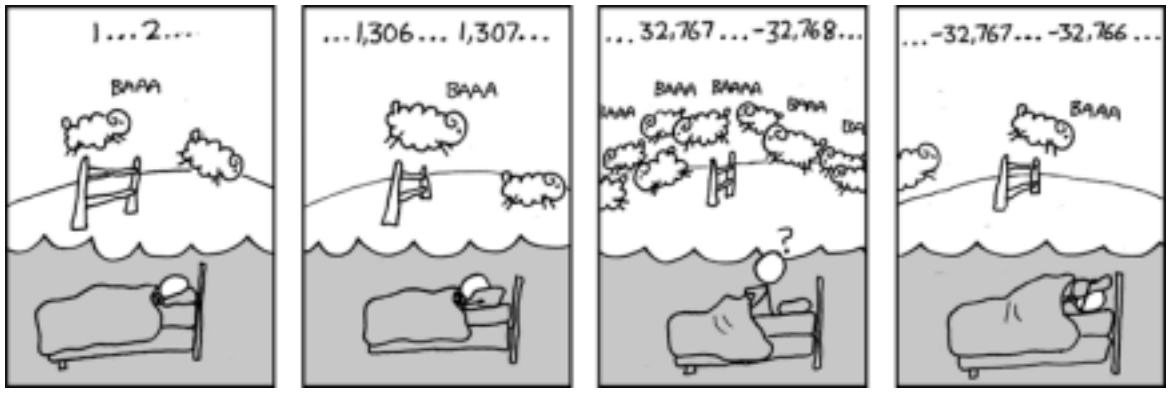
CS4414 Fall 2024 Homework 4 Part 1
Due November 10

Goals: One limitation of C++ is imposed by the CPU: an integer is represented by a field with some fixed number of bits. On Intel, this could be 8, 16, 32, or even 64: a "long long". In the case of a signed integer, one bit (the "high order" bit) is reserved to represent the sign: 0 for a positive number, and 1 for a negative number. Unsigneddata types use that bit for data because they don't need a 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 4 part1 will create a "big numbers" arithmetic package for integers represented as vectors of individual digits, employing "elementary school" mathematics for operations such as + - * / % and a tricky operation we'll need for homework 4 part2: a^b % c.

The most common use of a **Bignum** solution arises because encryption keys can easily be larger than any of these limits. The case we will consider in hw4 part1 and part2 involves the "Rivest, Shamir and Adelman" (RSA) cryptographic tools. RSA keys are usually at least 2096 bits in length, and some systems use keys of size
4192 bits or longer. A key of that size would be 31 (or 61) decimal digits in length: definitely a very big number, and much too large to hold in any C++ integer type.

For homework 4, you'll create your own Bignum class, using the C++ vector class to hold the digits. We won't need to deal with negative numbers in this homework: RSA and similar packages are defined purely on unsigned, hence non-negative, integers.

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.

Beyond basic arithmetic operations

In homework 4 part2 we will use **a^b % c** frequently. So, beyond the operators listed above, add modular exponentiation as an extra operator (this form of ^ needs 3 bignum arguments, not 2).

Here's how it works. Start by looking closely at the best way to compute a^b. Suppose that b is a number like 23. Write b in binary: 23 = 16+7, hence 010111. Now, consider the sequence a, a^2, a^4, …. If we want to compute a^{23} we could consider this to be the same as a^1*a^2*a^4*a^{16}, directly from that binary

encoding (as you can see, we do not multiply by a^8 because that bit was 0: we multiply by a^k if the k'th bit is 1, counting from the low-end bits on the right). This makes a^b is very easy to compute. We simply take 1, then start with a, square it repeatedly, and multiply our result by this squared value if the corresponding bit is equal to a 1.

The problem, though, is that with bignums, a^b will become enormous. For example, if a is a 30-digit number in base 10, a^2 would generally be a 60-digit number. Since b itself could be a 30 digit number you can see where this is going: a^b would probably be 3600 base-10 digits long.

But a^b % c is a different expression: the result can't become larger than c unless, while computing a^b we neglect the "mod c" part. Thus if c is 30 digits long, we can repeatedly truncate to 30 digits. Moreover, the "mod c" operator commutes! When we compute $a^1*a^2*a^4*a^{16}$ we could do this as (((((a^1 % x) * (a^2 % c)) % c) * (a^4 % c)) % *(a^{16} % c)) % c) In effect, we apply mod c at every step to prevent our numbers from ever getting really huge. Each time we square our working exponential of a, we take the result mod c. And each time we multiply this against our working result, we compute the result mod c as well. The squared numbers get somewhat big, but those are the only ones that do, and our result never becomes really huge.

Detailed assignment.

Let's break the task down into steps.

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:

```
% ./bignum + 1 2

bignum + 1 2
 3
```

We will use this same approach for +, -, *, /, % (remainder). For the a^b % c case, aka modular exponentiation, we just add an extra operator, ^, and it requires a third argument to bignum. 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 %, >, <, !, ^, &, |, $). Putting quotes around a special character can work, too.

1. We will implement our code in three files: main.cpp, bignum.cpp and bignum.hpp.

2. If executed with the wrong number of arguments, bignum should complain, as follows (in these examples we won't show the bash prompt):

% ./bignum + 1 2 3 4 5
Error: +-*/% requires two numbers

3. 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 3, 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++.
4. If executed with an invalid operator, it should print this:

   % ./bignum \# 18221110 55771
   Error: "#" is not a supported operator
   The backslash is needed to tell bash that the argument should be treated as text (you can also put the # in quotes)
   You can assume only one type error will occur at a command. We are not testing the case where the numbers and the operator are incorrect.
5. 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 "zero'th" 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 zero'th element in the array of char* c-string pointers will point to the c-string **bignum**, null-terminated. Ignore this argument.

   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
bignum + 18221110 2000
18223110

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.

The class name will be **Bignum**. 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.

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 **modexp**. 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. 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. Examples:

- 00000 should be 0
- 07654321 should be 754321
- But 351800061 remains unchanged

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.

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.

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)). We care about correctness and clarity of your code.

Last, implement **^** using the method shown earlier.

Test your code!

Test that your package is really working, for example (these all need some form of quote around the operator – omitted to keep the line uncluttered)

% ./bignum + 18221110 2000
bignum + 18221110 2000
18223110

% ./bignum - 18221110 2000
bignum - 18221110 2000
18219110

% ./bignum – 2000 18221110

bignum – 2000 18221110
Unsupported: Negative number
0

% ./bignum * 18221110 2000
bignum / 18221110 2000
36442220000

% ./bignum / 18221110 2000
bignum / 18221110 2000
9110

% ./bignum / 18221110 0
bignum / 18221110 0
Error: Divide by zero
0

% ./bignum % 18221110 2000
bignum % 18221110 2000
1110

% ./bignum * 1771919 779351
bignum * 1771919 779351
1380946844569

% ./bignum * 1771919 329351
bignum * 1771919 329351
583583294569

% ./bignum ^ 17 35 19
bignum ^ 17 35 19
9

% ./bignum ^ 23 11 37
bignum ^ 23 11 37
29

% ./bignum ^ 2345 6789 10201
bignum ^ 2345 6789 10201
4829

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).

Submission

You will be submitting a zip file containing (only) your 3 source-code files (main.cpp, bignum.cpp, bignum.hpp) and CMakeLists.txt; our testing script will recompile the program on our machine and then run it. Just like in HW3 part 2, we will be using your CMakeLists.txt file to create an executable **called bignum**, and will run the executable with our test cases like below:

% ./bignum + 1 1
bignum + 1 1

2

Note that for this assignment, we will take points off if the program is highly inefficient, i.e. exceeds the time limit of 20 seconds per one command.

Formatting Requirements:

In order for us to grade your assignment, you must follow the following formatting rules:

1. Make sure your program checks for the following potential errors: the arguments are integers, the operator argument is valid, and that you have the correct number of arguments. If one of these errors occur, you must print out the following:
    a. For the integer argument case:
        i. **Error: "<argument that is invalid>" is not an unsigned integer**
        ii. Example:
            1. **% ./bignum + 1.42 32**
               Error: "1.42" is not an unsigned integer
    b. For the invalid operator argument case:
        i. **Error: "<operator argument that is invalid>" is not a supported operator**
        ii. Example:
            1. **% ./bignum $ 2 5**
               Error: "$" is not a supported operator
    c. For the incorrect number of arguments case:
        i. For +, -, /, *, and %:
            1. **Error: +-*/% requires two numbers**
        ii. For ^:
            1. **Error: Exponent requires three numbers**
        iii. Example:
            1. **% ./bignum + 3 4 5**
               Error: +-*/% requires two numbers
            2. **% ./bignum ^ 1 2**
               Error: Exponent requires three numbers
2. For the valid commands, we expect you to print two things:
    a. The command (This can help you to debug the program with the autograder we provided)
    b. Output result

c. Example:
    i. **% ./bignum + 3 4**
    bignum + 3 4
    7
    ii. **% ./bignum / 10 2**
    bignum / 10 2
    5

d. In the case of a divide by zero, we expect you to print out the command, followed by **Error: Divide by zero** and then an output of zero
    i. Example:
        1. **% ./bignum / 10 0**
        bignum / 10 0
        Error: Divide by zero
        0

e. In the case of a negative number, we expect you to print out the command, followed by **Unsupported: Negative number** and then an output of zero
    i. Example:
        1. **% ./bignum - 143 155**
        Unsupported: Negative number
        0