Goals

The goals of this lab are

- Design and implement a polynomial library to support various operations such as addition, multiplication and modulus.
- Create a fast polynomial multiplication algorithm (hint: threads may help) **and ensure efficient implementation of other operations.**

The way that you approach the problem is up to you, but the design of your solution is a key part of the lab experience.

Submission

Upload poly.h and poly.cpp to Gradescope.

Part 1

All functions must be implemented as described, but timing doesn't matter for this submission. We strongly suggest getting the following to work before working on anything else

- Addition. In particular you must support
  - addition of two polynomials
  - an expression of the form polynomial + int
  - an expression of the form int + polynomial
- Multiplication. In particular you must support
  - multiplication of two polynomials
  - an expression of the form polynomial * int
  - an expression of the form int * polynomial
- Modulo
  - polynomial % polynomial
- Constructor
- Canonical form

Note: We use canonical form to compare your solutions to ours. This means that if your canonical form doesn't work we have no way to verify if your solution is correct. So it's imperative that your canonical form is correct.

Modulo

You will need to implement the polynomial modulus operator; i.e. the remainder after polynomial long division. In other words: Let $p(x)$ and $d(x)$ be polynomials such that

$$p(x) = d(x) * q(x) + r(x)$$

where the degree of $r(x)$ is strictly less than the degree of $p(x)$. Then the modulus $p(x)$ % $d(x) = r(x)$. Your implementation need only account for cases where all the coefficients (including those of $q(x)$ and $r(x)$) are integers. Note that you are only required to compute $r(x)$, you do not need to calculate $q(x)$.

Some links you might find helpful: * Wikipedia page with long division algorithm * Polynomial long division blog post

Part 2

All functions must be implemented as described. Timing matters for this submission, and we'll provide more timing information closer to this deadline.

Restrictions

- You must implement all functions/operators described in the header file
- You ARE allowed to (and should) add extra fields and functions to the header file, but you're not allowed to remove any fields/methods that we give you. This means you can define whatever helper functions/extra fields that you want to help with the project.
- The auto grader will only link against pthreads, do not include any other external library that will require the auto grader to link against it.
- You must turn in a poly.cpp file and a poly.h file. Each function/operator defined in poly.h must have a definition in poly.cpp.

Polynomial files

We have provided two sample polynomial files (`result.txt` and `simple_poly.txt`) to help with testing.

The format of the polynomial file is

```
<coefficient>x^<power>
<coefficient>x^<power>
....
<coefficient>x^<power>;
```

note: the semicolon is there to make parsing easier, since it marks the end of the polynomial. Do not assume the powers are in any particular order (ascending, descending etc.).

It's also important to notice that there are no addition signs between the polynomials, the addition is implied. So if you wanted to write

$$3x^{10} - 7x^2 + 1$$

This would be represented in the polynomial file as

```
3x^10
-7x^2
1x^0
;
```

The `simple_poly.txt` file contains two polynomials, and `result.txt` contains their expected product.

Thread examples

This is a short example of how to use threads, and also has documentation for what functions threads have.

Here is a more in depth explanation of how to use threads.

Running your code

We've given you a main.cpp file that has some code in it that you can use to time your own code. That will help you figure out how long your polynomial multiplication takes to run.

Notes for Gradescope auto-grader

- Each suite of test cases includes tests such as addition, modulo in addition to the multiplication test. Even though we are not timing individual operation on Part 1, it is still important the full suite of test

cases completes within a timeout period. If it takes much longer, you may get a "timeout" message. In this case, you should do some local testing to figure which of the operations is expensive.
- Gradescope will not display the score for Part 2. Instead, it will show the execution time. The final score will be given after the deadline. For both parts, standard project policy applies (additional test cases may be added after the deadline).

Hints and Advice

- Start immediately - you are on your own for testing, so budget plenty of time.
- First get a correct sequential implementation. This will ensure you at least get credit for this portion, and will give you a starting point for your optimizations for running time.
- What is a good data structure to use to represent a polynomial? You are free to use any STL containers of your choice, but you should carefully think through what a good choice is, and how to use it.
- How would you handle sparse polynomials? E.g., consider a polynomial with just two terms, $x^{100000} + 4$. What is a good way to represent it? How would the representation impact addition and multiplication?
- To do multiplication, you can potential speedup with multiple threads. What is a good design to obtain the parallelism?
- a * b and b * a should have similar speeds. Does your design do this? Only worry about this once you've gotten a parallel implementation working
- The print() function may be used for debugging and it is not graded. Feel free to overload operator« instead of using the print() function.

How to use TA time

You should be using TA time to get feedback on your design and ideas, and questions related to threads. Don't expect straight answers on what a good design is, but expect TAs to make you think of the pros and cons of a design you propose.