CSC 211 Assignment 6

Recursion, templates, and memoization

Due Wednesday, April 11th by 11PM

Background

In class, we have explored several problems that lend themselves to recursive solutions. Two of these, factorial and the Fibonacci numbers, are what you will approach in this assignment.

There are many numeric data types in C++: the `int` family, the `float` family, and now your `bigint` class. We could imagine implementing factorial or Fibonacci with any of them. Indeed, you will implement factorial for **all** of them, by writing just one function! And, you will implement Fibonacci for `bigint`s, taking advantage of a nifty algorithmic trick called **memoization** (discussed in class).

Submission limits

For this assignment, you only get 5 submissions to Mimir, so make sure to **test locally** first. Only submit to Mimir when you have successfully run your program locally on several different inputs (including on files that don't exist).

Academic honesty

Remember, this is a **solo assignment. You may not show your code to any classmate, or look at any classmate's code.**

Getting Started

- Get into your Docker development environment (or appropriate alternative, such as CodeAnywhere.com)
- Download the assignment framework with `git clone https://github.com/csc211/a6`
- You will see this `Readme.md` along with some C++ source files (`fib.cpp`, `fact.cpp`), a `bigint` library, and a compile script `compile`

- The compile script will build you two executable binaries, `fib` and `fact`, which (once you complete the assignment) will compute Fibonacci numbers and factorials, respectively.
- This time, some of the code you write will be in an `.h` file, because you are going to write *templated* functions.

Requirements

- Write a **templated, recursive** function `factorial()` in the file `fact.h`.
  - This function should work for any numeric type that supports multiplication: `int`, `double`, `bigint`, etc.
  - It should return the same type given as its argument: `int` for `int`, `bigint` for `bigint`, and so on.
  - It must compute the factorial of the argument supplied.
  - Your `factorial()` function must be recursive; you will lose points for an iterative solution.
- Write a **recursive** function `r_fib()` which can be called by the `fib()` function I have supplied, to compute the n^{th} Fibonacci number.
  - The key here is that `r_fib()` has an additional argument, `memo`, a memoization table implemented as an `unordered_map` as discussed in class.
  - Computing Fibonacci numbers recursively can be **really** computationally expensive, but by taking advantage of memoization, it can be quite fast (linear time).
  - Your Fibonacci calculator isn't templated; it need only work on `bigint`s.
  - Your `r_fib()` function must be recursive; you will lose points for an iterative solution.

Comments

Your comments should **explain the contract of any function you write**. That is what arguments does a function expect, and what value does it return, and **what is the relationship between the two**.

Comments should not focus on explaining the C++ language to the reader, though at this point, if doing so helps you understand your own code, it's fine. Any code you write that you think is particularly "clever" should be explained (for example, non-obvious corrections for off-by-one errors, or a for loop that starts at an index other than 0).

Hints

For the `unordered_map` type, it is essential to query whether an entry is in the map before trying to access it. Otherwise, the entry is inserted when you try to access it.

Given an `unordered_map m`, you can check whether an entry is in it by seeing whether the `find()` method returns a value equal to that returned by the `end()` method. If `m.find(k) == m.end()` then there was no entry in the map `m` for key `k`. Otherwise, you can then access the value associated with key `k` just as you would in a vector: `m[k]` returns the value associated with key `k`.

To insert into an unordered_map, it's also just like vector assignment: `m[k] = v` would insert a new entry into `m`, with key `k` and value `v`.

Once you have written code that compiles, the provided `compile` script will produce *two* executables: `fib` and `fact`. You can run them as follows:

`./fib 10` will compute and print the 10th Fibonacci number.

`./fact 10` will compute and print 10 factorial, using (at present) three data types.

The reason you are writing `factorial()` in a `.h` file is that you are really not writing a function. You are writing a template for a function, which the compiler can use to **generate** as many functions as it determines are needed at compile time. Since the `main()` I provided in `fibmain.cpp` uses `int`, `bigint`, and `double`, the compiler will generate functions for each of those three types. If you wish, you can add additional functionality; try computing factorial on `unsigned char` or `unsigned long long`. **I encourage you to play with the factorial program once you get it written** – notice how for different data types, large factorials (try `./fact 100`) overflows differently?

Testing your own code

The provided `main()` functions in `fib.cpp` and `fact.cpp` should let you determine if your code is working. You are welcome to add more testing functionality.

Grading Rubric

For this assignment, correctly passing all tests on Mimir is worth 80% of your grade. The remaining 20% is based on reasonable commenting habits, and the structure and organization of your code.

Submitting

You will submit `fib.cpp` **and** `fact.h` via Mimir, where its functional correctness will be graded automatically. For this assignment, you only get 5 submissions to Mimir, so make sure to **test locally** first.