This is an individual assessment, and, as the name suggests, must be completed individually. Specifically, you're not allowed to work with a partner, and you should not discuss these problems with other students in CS166. However, the course staff are happy to answer clarifying questions on Pi-azza (if you do, please post the question privately) or in our office hours.

**Due Tuesday, April 21 at 2:30PM Pacific time.**
Your task is to implement several RMQ structures in C++. In doing so, we hope that you’ll better understand how those structures work, get some experience translating ideas from Theoryland into actual code, and discover some nuances of how these structures work.

We've provided starter files at /usr/class/cs166/assignments/a1 and a Makefile that will build the project. Check the README for information about the starter files and how to run the provided test drivers. Consult the “Assignment Policies” handout for information on how to submit your work.

i. Implement the PrecomputedRMQ type using the \(O(n^2), O(1)\) RMQ data structure that precomputes the answers to all possible range minimum queries. This is mostly a warmup to make sure you're able to get our test harness running and your code compiling.

ii. Implement the SparseTableRMQ type using a sparse table RMQ data structure. Watch your memory usage here. Don't allocate \(\Theta(n^2)\) memory to hold a table of size \(\Theta(n \log n)\). Space, like time, is a scarce resource.

You will need to do some preprocessing to be able to find the largest \(k\) where \(2^k \leq j - i + 1\) in time \(O(1)\). You should not assume that the number of bits in a \texttt{size\_t} is a constant, so doing a \texttt{for} loop over the bits of a \texttt{size\_t} does not take time \(O(1)\). (We're mentioning this because you'll need to key your table on a power of two that depends on the size of the query range, and you shouldn't determine this range size using loops that depend on the number of bits in a \texttt{size\_t}.) However, you can assume that each arithmetic instruction (add, subtract, bitwise XOR, etc.) takes time \(O(1)\); even though these operations work on multiple bits in parallel, it's customary to count them as taking time \(O(1)\). More on this later in the quarter.

Your solution here should be deterministic, meaning that you should not use hash containers like \texttt{std::unordered\_map} or \texttt{std::unordered\_set}.

iii. Implement the HybridRMQ type using the \(O(n), O(\log n)\) hybrid structure we described in the first lecture, which combines a sparse table with the \(O(1), O(n)\) linear-scan solution.

iv. Implement the FischerHeunRMQ type. You're welcome to implement either the slightly simplified version of the Fischer-Heun structure described in lecture (which uses Cartesian tree numbers and is a bit simpler to implement) or the version from the original paper (which uses ballot numbers). You may want to base your code for this part on the code you wrote in part (iii).

Encode your Cartesian tree numbers or ballot numbers as actual integers rather than, say, as a \texttt{std::vector<bool>} or \texttt{std::string}, as these latter approaches are significantly slower.

Big-O notation is not a perfect measure of efficiency. The hidden constants can be significant, and some effects not captured by big-O notation (for example, caching and locality of reference) can have a bigger impact than just the number of instructions executed.

vi. Implement the FastestRMQ type with the fastest implementation of RMQ that you can come up with. Here, “fastest” is measured purely as the wall-clock runtime on a variety of different work-flows. You're free to implement this type however you'd like, as long as you didn't just copy and paste one of the implementations from earlier in this problem. Describe your design decisions in the header file.

We will use whatever Makefile you provide to run your code and will test on\texttt{myth}. We recommend that you, at a bare minimum, use the \texttt{-Ofast} and \texttt{-march=native} options to \texttt{g++}.

A little incentive for part (v) of this problem: we’ll take some time out of a later class meeting to celebrate the people who came up with the fastest solutions. We’re excited to see what you come up with!

We recommend using the handy \texttt{gprof} tool to profile your implementation and find where the bottlenecks are. To use \texttt{gprof}, add the \texttt{-pg} (profile-guided) command-line option to \texttt{g++}, run the generated executable, then run \texttt{gprof} to analyze where the program was spending its time.