

Problem Set 1: RMQ

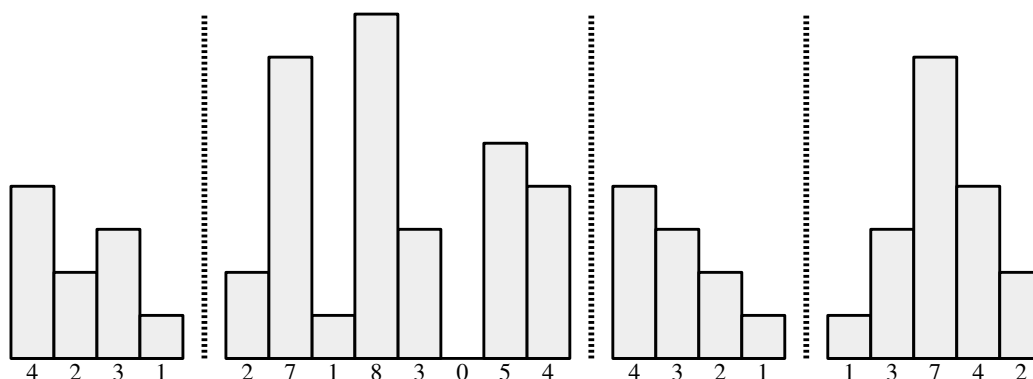
This problem set is all about range minimum queries and the techniques that power those data structures. In the course of working through it, you'll fill in some gaps from lecture and will get to see how to generalize these techniques to other settings. Plus, you'll get the chance to implement the techniques from lecture, which will help solidify your understanding.

You are welcome to work on this problem set either individually or in a pair. If you work with a partner, you should submit a single joint submission on Gradescope, rather than two separate submissions.

Due Tuesday, April 12th at the start of lecture

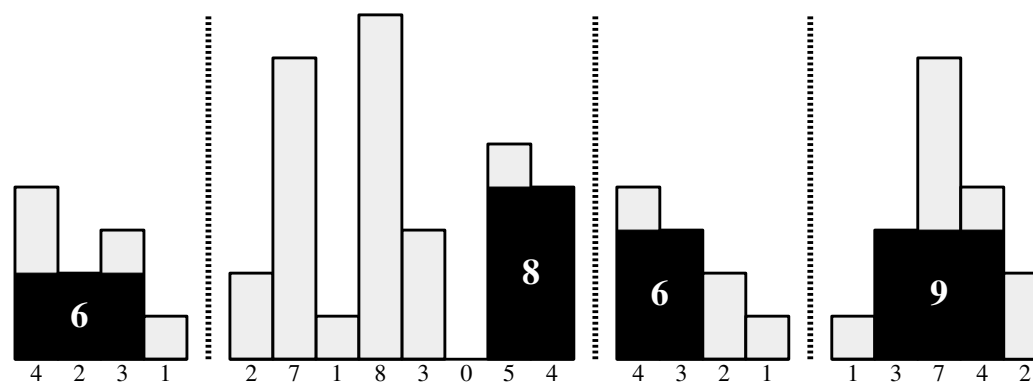
Problem One: Skyscrapers

A *skyline* is a geometric figure consisting of a number of variable-height boxes of width 1 placed next to one another that all share the same baseline. Here's some example skylines, which might give you a better sense of where the name comes from:



Notice that a skyline can contain boxes of height 0. However, skylines can't contain boxes of negative height.

You're interested in finding the area of the largest axis-aligned rectangle that fits into a given skyline. For example, here are the largest rectangles you can fit into the above skylines:



Design an $O(n)$ -time algorithm for this problem, where n is the number of constituent rectangles in the skyline. For simplicity, you can assume that no two boxes in the skyline have the same height. Follow the advice from our Assignment Policies handout when writing up your solution – give a brief overview of how your algorithm works, describe it as clearly as possible, formally prove correctness, and then argue why the runtime is $O(n)$.

Problem Two: Implementing RMQ Structures

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. As with the previous assignment, for full credit your code should run cleanly under `valgrind` and should compile with no warnings (e.g. it should compile with `-Wall -Werror` enabled).

- i. Implement the `PrecomputedRMQ` type using the $\langle O(n^2), O(1) \rangle$ 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.

Don't worry if you get out-of-memory errors for large values of n . Something to think through: if you're making a table of size n^2 when $n = 200,000$, how many bytes are you using?

Some notes on this problem:

- Your solution should be deterministic. This precludes the use of hash tables like `std::unordered_map` or `std::unordered_set`.
- Be mindful of the costs of individual operations on standard containers. For example, the cost of a lookup in a `std::map` is $O(\log n)$, where n is the number of items in the map, which will likely preclude using this approach.

- ii. Implement the `SparseTableRMQ` type using a sparse table. 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 `size_t` is a constant, so, for example, a for loop over the bits of a `size_t` does not take time $O(1)$. Think of it this way – if we were to port your code to a 2,048-bit computer, the time complexity of your code should be the same. 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 `size_t`.

Similarly, the convention in Theoryland is to *not* assume that floating-point operations take time $O(1)$. There are some surprising results about what you can do if you assume you have the ability to, say, divide and take floors of real numbers with high precision.

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 $O(1)$.

- iii. Implement the `HybridRMQ` type using the $\langle O(n), O(\log n) \rangle$ hybrid structure we described in the first lecture, which combines a sparse table with the $\langle O(1), O(n) \rangle$ 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 `std::vector<bool>` or `std::string`, as these latter approaches are significantly slower.

(Continued on the next page...)

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.

As an optional last step, but one we nonetheless think is very informative: update the `Makefile` to add the optimization flags `-Ofast` and `-march=native` (the first of these turns on optimizations that might technically break standards compatibility, the second says to generate code designed purely for the target machine even if that code isn't portable across other processors), then rebuild everything. Run some time trials with the different RMQ structures. Which one seems to be the fastest? Can you explain why?