Problem Set 1: RMQ

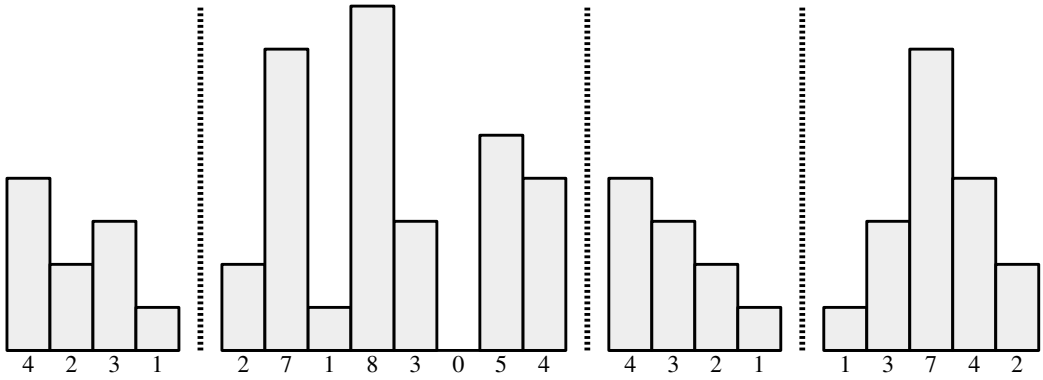
This problem set is all about range minimum queries and the techniques that power those data structures. In the course of working through it, you'll fill in some gaps from lecture and will get to see how to generalize these techniques to other settings. Plus, you'll get the chance to implement the techniques 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 18th at noon Pacific time.

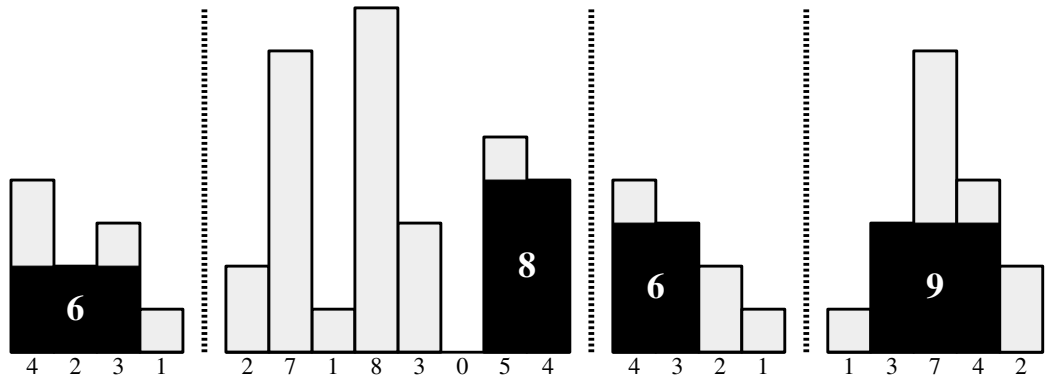
Problem One: Skylines

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: Area Minimum Queries

In what follows, if A is a 2D array, we'll denote by A[*i*, *j*] the entry at row *i*, column *j*, zero-indexed. When drawing out a grid, we'll assume that (0, 0) represents the upper-left corner of the grid, that (1, 0) represents the entry in the leftmost column of the row just beneath the top row, and that (0, 1) represents the entry in the top row, just to the right of the upper-left corner.

This problem concerns a two-dimensional variant of RMQ called the ***area minimum query*** problem, or ***AMQ***. In AMQ, you are given a fixed, two-dimensional array of values and will have some amount of time to preprocess that array. You'll then be asked to answer queries of the form "what is the smallest number contained in the rectangular region with upper-left corner (*i*, *j*) and lower-right corner (*k*, *l*), inclusive?" Mathematically, we'll define $AMQ_A((i, j), (k, l))$ to be $\min_{i \le s \le k,\, j \le t \le l} A[s, t]$. For example, consider the following array:

31	41	59	26	53	58	97
93	23	84	64	33	83	27
95	2	88	41	97	16	93
99	37	51	5	82	9	74
94	45	92	30	78	16	40
62	86	20	89	98	62	80

Here, A[0, 0] is the upper-left corner, and A[5, 6] is the lower-right corner. In this setting:

- $AMQ_A((0, 0), (5, 6)) = 2$
- $AMQ_A((0, 0), (0, 6)) = 26$
- $AMQ_A((2, 2), (3, 3)) = 5$

For the purposes of this problem, let *m* denote the number of rows in A and *n* the number of columns.

i. Design and describe an $\langle O(mn), O(\min\{m, n\})\rangle$-time data structure for AMQ.

ii. Design and describe an $\langle O(mn \log m \log n), O(1)\rangle$-time data structure for AMQ.

As a hint, think about what these constraints say if you're looking at, say, a $1 \times n$ array, which should essentially reduce to a regular RMQ structure.

When writing up your answers, please follow the general advice from our "Assignment Policies" handout. Begin with a high-level overview of the approach, then drop down into lower levels of detail and explain how each operation works, concluding with a runtime analysis.

Fun fact: You can improve these bounds all the way down to $\langle O(mn), O(1)\rangle$ using a Four Russians speedup and some very clever auxiliary data structures. Check out the paper *Data Structures for Range Minimum Queries in Multidimensional Arrays* by Yuan and Atallah for details.

Problem Three: On Constant Factors

As a refresher, here's how the preprocessing step for the Fischer-Heun RMQ structure works:

- Split the input array into blocks of size $b = \frac{1}{2} \log_4 n$.
- Construct an array of the block minima and build a sparse table RMQ on the minima.
- Construct an array of size 4^b storing pointers to RMQ structures, initially all null.
- For each block in the input array, compute its Cartesian tree number. If the array entry for that number is null, create a new $\langle O(n^2), O(1) \rangle$ precompute-all RMQ structure and store it in the array at that point. At this point, the array slot at that index is definitely not null, so store a pointer from the current block to that RMQ structure.

You may have noticed that we specifically picked $b = \frac{1}{2} \log_4 n$ as its block size. Why is that ½ there? Why is this a base-4 logarithm? This question explores the answer to that question.

i. Redo the analysis of the Fischer-Heun preprocessing time we did in class (the one starting on Slide 245 of the slide deck) under the assumption that $b = \log_2 n$. What big-O runtime bound do you get?

   Just to make sure we're clear here, the only change you're making to the Fischer-Heun data structure is dropping the leading coefficient from b. The rest of the approach remains the same.

The bound you came up with in part (i) is not "tight," in the sense that if $p(n)$ is the true worst-case preprocessing cost for the modified Fischer-Heun structure, then $p(n) = o(f(n))$, where $f(n)$ is the bound you came-up with in part (i) of this problem and o denotes little-o notation.

ii. Give a tight bound on the worst-case preprocessing cost of the modified Fischer-Heun structure. By "a tight bound," we mean that we'd like you to find a function $p(n)$ such that the worst-case runtime of the Fischer-Heun structure is $\Theta(p(n))$.

   Some hints on this problem:

   - Your analysis will have to deviate significantly from the one we did in class, since as mentioned above, that analysis will overestimate the total work. Go back to basics, accounting for the cost of each step of the preprocessing algorithm in the worst case.
   - Feel free to use the fact that the number of distinct block types of size b is given by C_b, where C_b denotes the bth Catalan number.
   - Consider the relationship between how many blocks an input array of length n consists of and the number of possible blocks of size b. Which is bigger? By how much?
   - The analysis you did in part (i) not only overestimates the amount of work done, but *significantly* overestimates the amount of work done. You should see a major reduction in the bound, not just, say, shaving off a factor of log n or log log n or something like that.

Generally speaking, it's good to interrogate constant factors and other seemingly arbitrary design decisions in a data structure. Sometimes, those choices really are arbitrary and can be tuned for performance reasons. Other times, those choices are there for a specific reason, and seeing why helps you better understand why things work the way they do.

Problem Four: 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 structure work, get some experience translating ideas from Theoryland into actual code, and discover some nuances of how these structure 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. For full credit, your code should run cleanly under `valgrind` and should compile will no warnings (e.g. it should compile file with `-Wall -Werror` enabled).

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.

   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 Θ(n^2) memory to hold a table of size Θ(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 ⟨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 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?