

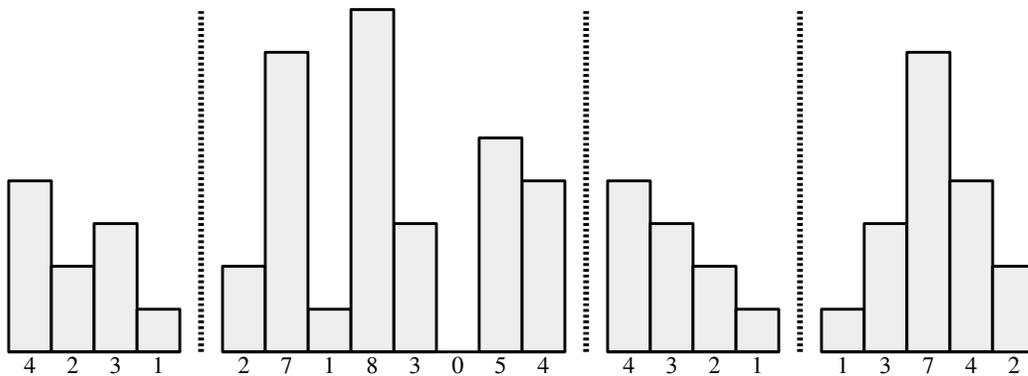
Problem Set 1: Range Minimum Queries

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.

Due Tuesday, April 16 at 2:30PM.

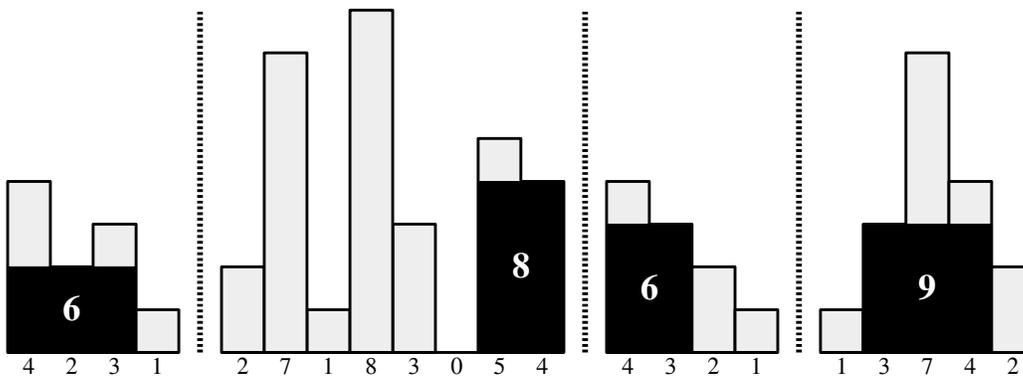
Problem One: Skylines (3 Points)

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 Problem Set 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 (4 Points)

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.

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) ?” Mathematically, we'll define $AMQ_A((i, j), (k, l))$ to be $\min_{i \leq s \leq k, j \leq t \leq l} A[s, t]$. 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.

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

It turns out that you can improve these bounds all the way down to $\langle O(mn), O(1) \rangle$ using some very clever techniques. This might make for a fun final project topic if you've liked our discussion of RMQ so far!

Problem Three: Hybrid RMQ Structures (4 Points)

Let's begin with some new notation. For any $k \geq 0$, let's define the function $\log^{(k)} n$ to be the function

$$\log \log \log \dots \log n \text{ (} k \text{ times)}$$

For example:

$$\log^{(0)} n = n \quad \log^{(1)} n = \log n \quad \log^{(2)} n = \log \log n \quad \log^{(3)} n = \log \log \log n$$

This question explores these sorts of repeated logarithms in the context of range minimum queries.

- i. Using the hybrid framework, show that for any fixed $k \geq 1$, there is an RMQ data structure with time complexity $\langle O(n \log^{(k)} n), O(1) \rangle$. For notational simplicity, we'll refer to the k th of these structures as D_k .

(Yes, we know that the Fischer-Heun structure is a $\langle O(n), O(1) \rangle$ solution to RMQ and therefore technically meets these requirements. But for the purposes of this question, let's imagine that you didn't know that such a structure existed and were instead curious to see how fast an RMQ structure you could make without resorting to the Method of Four Russians. 😊)

- ii. Although every D_k data structure has query time $O(1)$, the query times on the D_k structures will increase as k increases. Explain why this is the case and why this doesn't contradict your result from part (i).

(The rest of this page is just for fun.)

The *iterated logarithm function*, denoted $\log^* n$, is defined as follows:

$$\log^* n \text{ is the smallest value of } k \text{ for which } \log^{(k)} n \leq 1$$

Intuitively, $\log^* n$ measures the number of times that you have to take the logarithm of n before n drops to one. For example:

$$\log^* 1 = 0 \quad \log^* 2 = 1 \quad \log^* 4 = 2 \quad \log^* 16 = 3 \quad \log^* 65,536 = 4 \quad \log^* 2^{65,536} = 5$$

This function grows *extremely* slowly. For reference, the number of atoms in the universe is estimated to be about $10^{80} \approx 2^{240}$, and from the values above you can see that $\log^* 10^{80}$ is 5.

For arrays of length n , the data structure $D_{\log^* n}$ is an $\langle O(n \log^* n), O(\log^* n) \rangle$ solution to RMQ. Given that $\log^* n$ is, practically speaking, a constant, that makes for a crazily fast RMQ data structure!

Problem Four: Implementing RMQ Structures (10* Points)

In this problem, you'll implement several RMQ structures in C++. In doing so, we hope that you'll get a better feeling for some of the issues involved in translating data structures into code.

We've provided starter files at `/usr/class/cs166/assignments/ps1` and a `Makefile` that will build the project. Check out the `README` file for information about the organization of the starter files and how to run the provided test drivers.

- 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.
- 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 should not assume that the number of bits in a `size_t` is a constant. Each arithmetic instruction (add, subtract, bitwise XOR, etc.) takes time $O(1)$, so doing a for loop over the bits of a `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 `size_t`.)

Your solution here should be deterministic, meaning that you should not use the `std::unordered_map` or `std::unordered_set` types.

- 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.

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.

- v. 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 workflows that we'll use during grading. You're free to implement this type however you'd like, subject to the restriction that you describe your design decisions in the header file and you've done something more than just copying and pasting one of the four implementations from above.

A little incentive for part (v) of this problem: if your implementation is in the top 75% in terms of speed, we'll give you one bonus point. If your implementation is in the top 25%, we'll give you a second bonus point. If your implementation is the fastest across all submissions, we'll give you a third.

These bonus points won't be factored into your grade until after we've computed a grading curve, so you are not at any disadvantage if your implementation doesn't earn any bonus points. They're purely there to motivate you to think about different trade-offs in data structure design and implementation.

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