Problem Set 1: RMQ

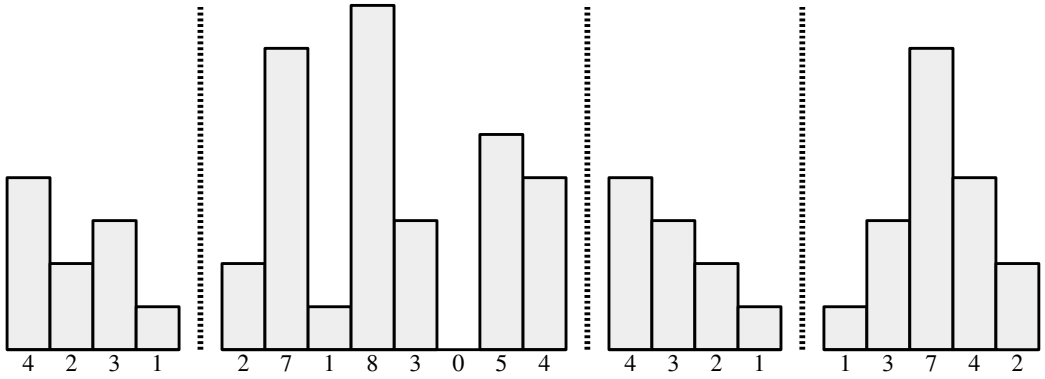
This problem set is all about range minimum queries and the techniques that power those data structures. In the course of working through it, you'll fill in some gaps from lecture and will get to see how to generalize these techniques to other settings. Plus, you'll get the chance to implement the techniques 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 21 at 2:30PM Pacific time.

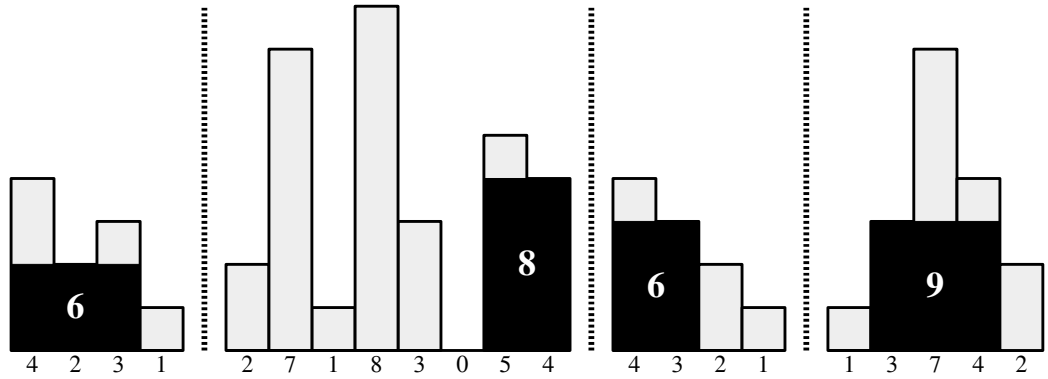
Problem One: Skylines

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.

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 $\min_{i \le s \le k,\, j \le t \le l} A[s, t]$. 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.

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 research project topic if you've liked our discussion of RMQ so far!

Problem Three: Hybrid RMQ Structures

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

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

For example:

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

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

i. Using the hybrid framework, show that 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 kth 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 for each fixed k the 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 section is just for fun.)

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

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

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

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

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 fast RMQ data structure!

Problem Four: On Constant Factors

The Fischer-Heun RMQ structure picks

$$b = \tfrac{1}{4} \log_2 n$$

as its block size. Explain what would happen if we instead picked $b = \log_2 n$. Specifically, address the impact, if any, on the correctness and runtime of the data structure.

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.