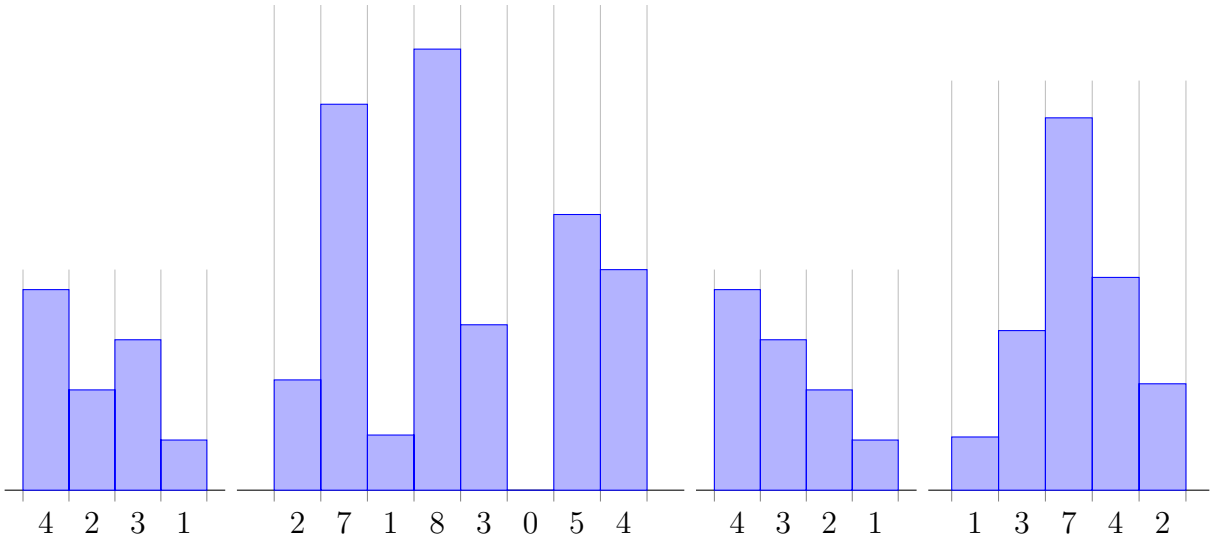
CS166 Problem Set 1: Range Minimum Queries

Luis A. Perez

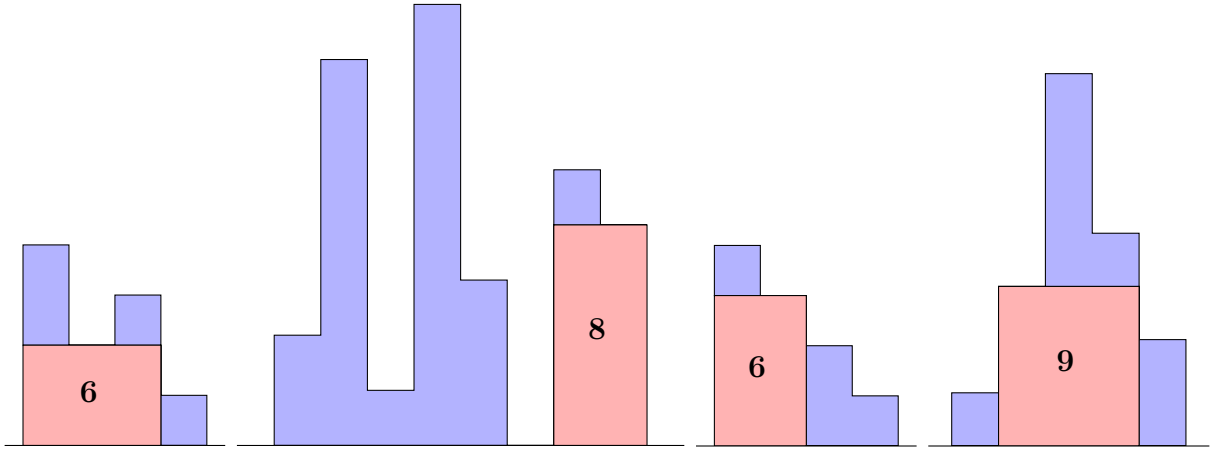
Problem One: Skylines (3 Points)

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 $O(n)$.

Solution: Overview For simplicity, we assume that all of the rectangles are distinct heights. Therefore, given heights h_i for the n rectangles, we have that $h_i \neq h_j, i \neq j$. We also have that $h_i \geq 0$ for all $0 \leq i < n$. The problem them boils down to finding a fast way to compute the area of all boxes and returning the maximum. We do this my considering all boxes of a given height h, and finding the box with the largest width (as this box has the largest area). Then, we simply iterate over this set of values and return the maximum, which corresponds the area of the largest box.

Algorithm For an array of heights H, we first construct and RMQ structure on this array (by processing the array).

We then use this RMQ to calculate $k = \text{RMQ}(0, n-1)$. We store $A[k] \leftarrow H[k] * n$. We then proceed to repeat this process on the left side $H[0, \cdots k-1]$ and right side $H[k+1, \cdots n-1]$ arrays (if they exist). More precisely, for the subarray $H[i, \cdots, j]$, we compute $k = \text{RMQ}(i, j)$ and compute and store the value $A[k] =\leftarrow H[k] * (j - i + 1)$.

Finally, we compute the maximum of the $A[i]$ and return this value.

Proof of Correctness We can see that $k = RMQ(i, j)$ gives us the index of the minimum rectangle in a subarray. It is then clear that the maximum area of a box contained entirely withing this subarray and whose height is $H[k]$ is $H[k] * (i - j + 1)$, since the box can just span the entire range. Furthermore, (1) any box of smaller height will then have smaller area (since it can at most span the entire array), and (2) any box of larger height cannot contain the rectangle at index k, since if it did, it's height would have to be at most $H[k]$, a contradiction.

As such, for each distinct height, we've computed the area of the largest box *of that height*. Our algorithm then simply takes the maximum of these values, and as such, must in-fact return the area of the largest box *of any height*, which is what we wanted.

Runtime Analysis The run-time analysis is straight-forward. Using the Fischer-Heun RMQ data-structure, the RMQ structure can be constructed in $O(n)$ time. Next, we note that each height has a one-to-one correspondance with a single RMQ query on our data structure. There are $O(n)$ distinct height, and as such, our algorithm performs $O(n)$ RMQ queries, each of constant time. Lastly, compute the maximum of A takes $O(n)$ times since, again, there are $O(n)$ heights.

The total run-time of our algorithm is therefore $O(n)$.

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.

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

Solution: Overview

The key insight into this data structure is that we can decompose the AMQ problem into multiple RMQ problems (WLOG, we define RMQ and AMQ as both returning the value, not the index). To be more precise, an area minimum query consists of first finding the minimum along each row/column, and then computing the minimum of these values. We can do this by making use of the Fischer-Heun RMQ, which is a $\langle O(n), O(1)\rangle$-time.

Pre-Processing Given a two-dimensional array A of size $n \times m$, we will construct $\min\{n, m\}$ RMQs (1-dimensional) data structures, one for each row (or column, if there are less columns than rows). Constructing each of the $O(\min\{n, m\})$ $RMQs$ takes $O(\max\{n, m\})$, for a total pre-processing time of $O(nm)$. We store these RMQs in an array indexed by their row/column index as RMQ_k.

Querying WLOG, we consider the case where $n \geq m$ (there are more columns than rows). The logic is equivalent with the roles of rows/columns swapped in when $m > n$. Therefore, when computing $AMQ((i, j), (k, l))$, we lookup the RMQ_r corresponding

to the row range $[i, \cdots, k]$. For each of $i \leq r \leq k$, we find $v_r = \text{RMQ}_r(j, l)$ (the minimum of the row). This takes $O(1)$ per row, with a total of $O(n)$ rows. We then compute the minimum of all computed v_r, and return this value, which takes $O(n)$. More generally, the query time is $O(\min\{n, m\})$.

Correctness Correctness follows directly from the correctness of RMQ. The minimum of an area is minimum of the minimum of each row/column.

Runtime As argued above, the pre-processing time is $O(nm)$ and the query time is $O(\min n, m)$. This means this is a $\langle O(mn), O(\min\{m, n\})\rangle$-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.

Solution: Overview

This solution is very similar to the sparse-table for the traditional RMQ problem. The key insight is that we can answer $AMQ((i, j), (k, l))$ using the answers to four AMQs each consisting of the four quadrants making up the box $(i, j) - > (k, l)$. To be more precise, letting $r = k - i + 1$ and $c = l - j + 1$, if we know the answers to $AMQ((i, j), (i + r/2, j + c/2))$, $AMQ((i + r/2 + 1, j), (i + r, j + c/2))$, $AMQ((i, j + c/2 + 1), (i + r/2, j + c))$, and $AMQ((i + r/2 + 1, j + c/2 + 1), (i + r, j + c))$, the answer to the original problem is simply the minimum of these four values.

With the above intuition, we can pre-compute a sparse-table containing answers to AMQs with top-left corner (i, j) of dimension $2^p, 2^q$ for all powers of 2. This means that we can directly look up the answers to $AMQ((i, j), (i + 2^p, j + 2^q))$.

Pre-Processing Similar to the sparse-table RMQ, we can pre-process our sparse-table using dynamic programming. We begin by computing all AMQs of dimension $(2^0, 2^0)$ (this is just $A[i, j]$ for each (i, j)). For the iteration, given that we've pre-computed all AMQs of dimension $(2^p, 2^q)$, we can compute all AMQs for dimensions $(2^{p+1}, 2^q), (2^p, 2^{q+1})$, and $(2^{p+1}, 2^{q+1})$ using the following relationships:

$$AMQ((i, j), (i + 2^{p+1}, j + 2^q)) = \min \begin{cases} AMQ((i, j), (i + 2^p, j + 2^q)) \\ AMQ((i + 2^p + 1, j), (i + 2^{p+1}, j + 2^q)) \end{cases}$$

$$AMQ((i, j), (i + 2^p, j + 2^{q+1})) = \min \begin{cases} AMQ((i, j), (i + 2^p, j + 2^q)) \\ AMQ((i, j + 2^q + 1), (i + 2^p, j + 2^{q+1})) \end{cases}$$

$$AMQ((i, j), (i + 2^{p+1}, j + 2^{q+1})) = \min \begin{cases} AMQ((i, j), (i + 2^p, j + 2^q)) \\ AMQ((i + 2^p + 1, j), (i + 2^{p+1}, j + 2^q)) \\ AMQ((i, j + 2^q + 1), (i + 2^p, j + 2^{q+1})) \\ AMQ((i + 2^p + 1, j + 2^q + 1), (i + 2^p, j + 2^q)) \end{cases}$$

There are $O(nm)$ top-left corners, and for each of these corners, we compute all AMQs of dimensions $(2^p, 2^q)$ for $0 \leq p < n, 0 \leq q < m$, for a total of $O(\log n \log m)$

dimensions. With the above formulation using dynamic programming, this will take time $O(mn \log m \log n)$.

Querying Given $AMQ((i,j),(k,l))$, we first find the largest p and largest q such that $2^p \leq i-k+1$ and $2^q \leq l-j+1$. We can do this in time $O(1)$ by pre-computing these values in the pre-processing step (takes $O(\log n + \log m)$ time). Then we note that the square defined by $[(i,j),(k,l)]$ can be formed as the overlap of four squares (1) top-left: $[(i,j),(i+2^p, j+2^q)]$, (2) bottom-right:, $[(k-2^p+1, l-2^q+1),(k,l)]$, (3) top-right $[(i, l-2^q+1),(k-2^p, l)]$ and, (4) bottom-left $[(k-2^p+1, j),(k, l-2^q)]$. Since the ranges have dimensions which are powers of 2, they can be looked up in $O(1)$-time. We then return the minimum value.

Correctness Correctness follows from the fact that a larger square can be decomposed into four overlapping squares whose width is at least half the original and whose height is at least half the original. This is precisely when our algorithm does.

Runtime As argued above, the pre-processing time is $O(mn \log m \log n)$ and the query time is $O(1)$. This means this is a $\langle O(mn \log m \log n), O(1)\rangle$-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 $\mathbf{\log^{(k)} n}$ to be the function:

$$\overbrace{\log \log \log \ldots \log}^{k \text{ times}} n$$

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 $\left\langle O\left(n \log^{(k)} n\right), O(1)\right\rangle$. For notational simplicity, we'll refer to the kth of these structures as D_k.

Solution: We use the hybrid framework from class. We recall than in the hybrid framework, the time complexity is given by $\langle O(n + p_1(n/b) + (n/b)p_2(b)), O(q_1(n/b) + q_2(b))\rangle$ where the time complexity of our root RMQ is $\langle O(p_1(n)), O(q_1(n))\rangle$ and the time complexity of our leaf RMQ is $\langle O(p_2(n)), O(q_2(n))\rangle$.

We show that for any fixed $k \geq 1$, there is an RMQ data structure, D_k, with time complexity $\left\langle O\left(n \log^{(k)} n\right), O(1)\right\rangle$. We do this by induction on k using a constructive proof (we show what this data structure looks like).

For the base case, we being with $k = 1$. In this case, we simple use the sparse table RMQ, which we've shown in class has time-complexity $\langle O(n \log n), O(1)\rangle$.

Let us now begin the induction. Suppose we have access to D_k (and RMQ data structure with time complexity $\left\langle O\left(n \log^{(k)} n\right), O(1)\right\rangle$). Then to construct D_{k+1}, simply use the hybrid-framework with D_k for the summary and leaf RMQs and with block size $b = \log n$.

The pre-processing time is then given by:

$$\begin{aligned}
O(n + p_1(n/b) + (n/b)p_2(b)) &= O\left(n + (n/b)\log^{(k)}(n/b) + (n/b)b\log^{(k)} b\right) \\
&\qquad \text{(By our inductive hypothesis about } D_k\text{)} \\
&= O\left(n + n/b \log n + n \log^{(k)} b\right) \\
&\qquad \text{(Simplifying using } \log^{(k)}(n/b) = O(\log n)\text{)} \\
&= O\left(n + n/(\log n) \log n + n \log^{(k)} \log n\right) \\
&\qquad \text{(Subtituting our choise of } b = \log n\text{)} \\
&= O\left(n + n + n \log^{(k+1)} n\right) \text{ (Using definition of } \log^{(k)}\text{)} \\
&= O\left(n \log^{(k+1)} n\right)
\end{aligned}$$

The query-time is given by:

$$O\left(q_1(n/b) + q_2(b)\right) = O\left(1\right)$$

As such, we have just constructed D_{k+1} using D_k which has time complexity $\left\langle O\left(n \log^{(k+1)} n\right), O\left(1\right)\right.$ Therefore, we have now show that for any $k \geq 1$, D_k exists and has the desired time-complexity.

(Yes, we know that the Fischer-Heun structure is a $\langle O\left(n\right), O\left(1\right)\rangle$ 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 $O\left(1\right)$, 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).

Solution: Let us normalize such that the query-time for D_1 is $t_D(1) = 1$ time unit. Then with our hybrid framework, D_2 requires three queries on D_1 for a single D_2 query. That is to say, the $t_D(2) = 3t(1)$. In fact, D_k is constructed such that $t_D(k) = 3t_D(k-1)$. This means that the query-time is in fact $O\left(3^k\right)$ (it grows exponentially with k). However, this does not contradict our $O\left(1\right)$ query time from above since k is not a function of n, and as such, does not grow with our input array. For the purposes of this problem, k is a constant, so $O\left(3^k\right) = O\left(1\right)$,

Problem Four: Implementing RMQ Structures (10^+ Points)

This one is all coding, so you don't need to write anything here. Make sure to submit your final implementations on myth.