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 17 at 2:30PM.

Problem One: Sparse Tables with O(1) Queries (2 Points)

To compute $\mathrm{RMQ}_A(i, j)$ with a sparse table in time O(1), it's necessary to compute in time O(1) the largest k for which $2^k \le j - i + 1$. Explain how to modify the preprocessing step of the sparse table by adding $O(n)$ additional work such that you can answer these queries in time O(1). Feel free to introduce as much additional memory as you think would be necessary.

For the purposes of this problem – and, more generally, throughout the world of data structures – we'll need to pin down what sorts of operations you can perform on an integer that fits into a machine word in time O(1). You can assume that any mathematical or logical operation for which there's a built-in C operator (addition, multiplication, bitwise AND, bitshifts, etc.) take time O(1). You can also assume that the size of the array you're working with fits into a machine word.

However, you should ***not*** assume that more complex mathematical operations (logarithms, radicals, etc.) can be computed in time O(1), nor should you assume access to processor-specific bitwise operations (e.g. `popcount`, `bsr`, etc.). These assumptions are typically made in the world of data structures.

Importantly, the runtime of your operations should ***not*** depend on the size of a machine word, and you should not assume that the word size is necessarily 32 or 64 bits.

Problem Two: Area Minimum Queries (3 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 $\mathrm{AMQ}_A((i, j), (k, l))$ 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:

- $\mathrm{AMQ}_A((0, 0), (5, 6)) = 2$
- $\mathrm{AMQ}_A((0, 0), (0, 6)) = 26$
- $\mathrm{AMQ}_A((2, 2), (3, 3)) = 5$

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.

Problem Three: Hybrid RMQ Structures (4 Points)

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

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

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 crazily fast RMQ data structure!

Problem Four: Implementing RMQ Structures (9 Points)

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

For the purposes of this problem, your structures should answer range minimum queries by returning the ***index*** at which the minimum value in the range resides, rather than the value at that index. If there are multiple values tied for the smallest, you can return the index of any one of them.

The data structures you'll be implementing will answer RMQ over arrays of `float`s. We've chosen arrays of `float`s because `int`s (representing indices) and `float`s (representing values) aren't implicitly convertible to one another in Java. In other words, if you try to assign an index to a value or vice-versa, you'll get a compiler error rather than a runtime error.

We've provided Java starter files at `/usr/class/cs166/assignments/ps1` and a Makefile that will build the project. The classes you need to implement are in the root directory. To run our driver program on a particular data structure, execute the command

```
./run your-rmq-structure random-seed_opt
```

Here, `your-rmq-structure` is the name of the class containing your RMQ structure. For example, you could run your sparse table code by running

```
./run SparseTableRMQ
```

The ***`random-seed`***$_{\text{opt}}$ parameter (a `long`) is optional and is used to force a specific random number seed when running the program. You might find this helpful during testing to guarantee that each run of the program tests your RMQ structure on identical inputs.

i. Implement 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 a sparse table RMQ data structure. Make sure that your data structure can answer queries in time $O(1)$; to do so, we recommend implementing the data structure you designed in Problem One.

    Watch your memory usage here. Don't allocate $\Theta(n^2)$ memory to hold a table of size $\Theta(n \log n)$. Java automatically zeroes out memory, so if you allocating $\Theta(n^2)$ memory requies $\Theta(n^2)$ time.

iii. Implement 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 Fischer-Heun data structure. 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).

    Your solution here should be deterministic, meaning that you should not use the `HashMap` or `HashSet` types. Encode your Cartesian tree numbers or ballot numbers as actual `int` values rather than, say, as an `ArrayList<Integer>`, `String`, etc.