CS 166 NOTES: DATA STRUCTURES

ARUN DEBRAY
APRIL 2, 2014

Contents

1. Range Minimum Query: 3/31/14 1
2. Cartesian Trees and RMQ: 4/2/14 3

1. Range Minimum Query: 3/31/14

"They told me [my email address] would stick with me for the rest of my life, and I said, 'how long could that be?' "

The course website is `cs166.stanford.edu`. The textbook is CLRS, which is often the 161 textbook; the 3rd edition is ideal, but the 2nd also works. There will be seven problem sets, a midterm, and a final project; interestingly, the problem sets can be pair-solved, with two people submitting one problem set.

The first question: why study data structures? One interesting answer is that it lies at the intersection of theory and practice. A lot of data-structures research is motivated by problems in the real world, and a lot of it is motivated by theoretical drives to achieve better results and asymptotics. Practical results follow from theoretical ones, and vice versa.

Data structures also provide new ways to model and structure problems. A data structure is a way of modeling the world, so having more of them provides more ways to efficiently solve problems. This is similar to what you may have seen in CS 161, where the point was not to memorize algorithms, but to provide a bunch of frameworks for building algorithms. In some sense, data structures expand the scope of what one can do efficiently, replacing naïve approaches with more efficient ones, or easier ones.

Today, we will talk about the range minimum query problem and several different solutions for it. It's easy to state: given a fixed array A and two indices $i \le j$, what is the smallest element out of $A[i], A[i+1], \ldots, A[j]$? For example, in the array $[31, 41, 59, 26, 53, 58, 97, 93]$, the minimum between 59 and 97 is 26. Notationally, the query $\text{RMQ}_A(i, j)$ indicates the minimum of array A between indices i and j; but it also denotes the answer to that query. This somewhat simple problem ends up being useful in other places. Furthermore, for simplicity, assume 0-indexing.

The obvious solution is $O(n)$: just iterate across A from index i to index j. But sometimes, if it's known ahead of time that there will be k queries, then one can do better. In an array of length n, there are $\Theta(n^2)$ possible queries, so one could precompute all of them and store them in a table, allowing answer time in $O(1)$ per query.

So now, how to fill the table? One way to do this would be to take the approach above, and use the linear algorithm on each possible query. Thus, the total time required is $O(n^3)$. Though it's not *a priori* a tight bound, this will be $\Theta(n^3)$ in this case: if $a = j - i$, then this algorithm has to look at a elements to compute $\text{RMQ}_A(i, j)$. Thus, one could formally sum everything up; specifically, whenever $j - i \ge n/2$, it takes at least $\Theta(n)$ work, and there are $\Theta(n^2)$ such entries, so the total work required is in fact $\Theta(n^3)$.

But we can do better! By using dynamic programming, this table can be filled in with quadratic time. This uses the following algorithm:

- On the main diagonal, $i = j$, so $\text{RMQ}_A(i, i) = i$, so fill these in.
- In general, if $i, \ldots, j$ is broken into ranges $i, \ldots, k$ and $k, \ldots, j$, then the minimum overall is the lesser of the two minima: $\text{RMQ}_A(i, j) = \min\{\text{RMQ}_A(i, k), \text{RMQ}_A(k, j)\}$. Thus, starting from the next diagonal, compute the minimum of the element to the left and the element below (the minima of the two ranges, which sometimes overlap).

Thus, it's constant amount of work per range, so $\Theta(n^2)$ overall. It's better if you're making lots and lots of queries, but for fewer queries one might not want to do as much preprocessing. This is a common theme.

Some more notation: an RMQ data structure has time complexity $\langle p(n), q(n)\rangle$ if preprocessing takes time at most $p(n)$ and queries take time at most $q(n)$. Thus, we have seen two data structures so far, one of which is $\langle O(1), O(n)\rangle$, and the other of which is $\langle O(n^2), O(1)\rangle$. These are two extremes on a curve of tradeoffs; what's ideal for a given application?[1]

Another approach is block-based:

- First, split the block size into $O(n/b)$ blocks of size b.
- Then, compute the minimum value in each block.
- Now, for long queries, if a query extends over a whole block, then the precomputed minimum can be used, which saves time.

Now, we can analyze this in terms of n and b. The preprocessing time is $O(b)$ work per block, and there are $O(n/b)$ blocks, so the preprocessing takes time $O(n)$. It's independent of block size, which makes sense. Then, to compute $\text{RMQ}_A(i, j)$, there are two partial blocks and at most n/b full blocks to compute the minimum from, so the tile is $O(b + n/b)$.

Thus, if b increases, then n/b drops, and if b decreases, then n/b rises. So to minimize this, we use calculus!

$$\frac{\mathrm{d}}{\mathrm{d}b}\left(b + \frac{n}{b}\right) = 1 - \frac{n}{b^2}.$$

Thus, setting this to zero, the optimal runtime seems to be when $b = n^{1/2}$, which makes the runtime $O(n^{1/2})$. Thus, we have a data structure with runtime $\langle O(n), O(n^{1/2})\rangle$: with modest preprocessing, we do a lot better.

The reason our $\langle O(n^2), O(1)\rangle$ solution was so slow is because every minimum had to be computed. Thus, one could only partially compute the table and then use what's computed to fill in the rest. And it doesn't have to just make jumps of one step; instead, as long as there are two ranges whose union is the whole range we are querying, then the minimum can be efficiently calculated. Alternatively, even if fewer entries in the diagonal are computed, then more than two have to be taken for the minimum. Thus, it's possible for the query time to be $O(1)$ even if only some of the table is precomputed. But can we do it in $o(n^2)$ preprocessing time, i.e. strictly better than the first solution?

Here's where it gets a bit magical; for each index i, compute the RMQ for ranges starting at i of size $1, 2, 4, \ldots, 2^k$ as long as they fit in the array. This gives both large and small ranges starting at any point in the array, which is good. Thus, there are only $O(n \log n)$ ranges.

Claim. Every range $i, \ldots, j$ can be composed of at most two of these precomputed ranges.

The intuition here is to pick powers of 2 starting at i and ending at j; if they don't overlap, you picked too small powers of 2.

Now, to answer $\text{RMQ}_A(i, j)$, find the largest k such that $2^k \le j - i + 1$, which can be done in $O(1)$ time with a little thought. Then, compute look up $\text{RMQ}_A(i, i + 2^k)$ and $\text{RMQ}_A(j - 2^k, j)$, and take their minimum (which corresponds to the solution for the range we care about). Thus, we end up using a slightly different table: starting position versus length (as a power of two), and each range is split down the middle to compiute its RMQ from the two subranges. This is called a sparse table, and gives an $\langle O(n \log n), O(1)\rangle$ solution to RMQ, which is asymptitically better than precomputing the ranges, which might be surprising.

Finally, one can hybridize this approach by combining it with the blocking approach; in some sense, once the block array is constructed, the goal is to do three RMQs: one on the block array, and two inside the edge blocks. Basically, constructing RMQ structures on each of the summary block array and on each block allows for one to get the answers. This sort of approach is generally called a macro/micro decomposition.

Claim. Suppose that one uses a $\langle p_1(n), q_1(n)\rangle$-time RMQ solution for the block minimums and a $\langle p_2(n), q_2(n)\rangle$-time RMQ solution within each block. Then, if b is the block size, the preprocessing time structure for the hybrid structure is $O(n + p_1(n, b) + (n/b)p_2(b))$, and the query time is $O(q_1(n/b) + q_2(b))$.

Proof. For the query time, there's one query on the top, i.e. $q_1(n/b)$ (since there are n/b blocks), and two of the bottom (so $O(q_2(b))$), and then constant time to compute the three-way minimum.[2] ⊠

This can be sanity-checked by looking at the first block structure we used, which just used the linear algorithm $p_1(n) = p_2(n) = 1$ and $q_1(n) = q_2(n) = 1$. When these are substituted into the equations from the claim, then the correct numbers fall out. But of course, we can do better!

If one has block size b, then it takes $O((n/b)\log(n/b))$ time to construct a sparse table over the block minimums. But since $\log(n.b) = O(\log n)$, then this is at most $O((n/b)\log n)\ldots$ thus, if $b = \Theta(\log n)$, then the time needed to construct

[1] This analysis doesn't really talk about memory; you could be more precise, but in this class time complexity will be more important, and in many cases the time complexity of preprocessing is equal to the overall space complexity. In other words, we are not going to space today.

[2] These are easy arguments to follow, but there's a lot of minding one's ps and qs.

a sparse table over the minima is $O((n/b)\log n) = O(n)$. The intuition is that $O(n \log n)$ is only slightly larger than $O(n)$, so we don't have to compute all that many things. But it's still pretty magical.

Another possible hybrid is to set the block size to $\log n$ again, but use the structure which doesn't use preprocessing. Thus, the overall preprocessing time is

$$O\left(n + p_1\left(\frac{n}{b}\right) + \left(\frac{n}{b}\right)p_2(b)\right) = O\left(n + n + \frac{n}{\log n}\right) = O(n).$$

Then, the query time is $O(q_1(n.b) + q_2(b)) = O(\log n)$. This $\langle O(n), O(\log n)\rangle$ solution is quite impressive, and is actually hard to beat on reasonable arrays (because of constant factors on asymptotically better solutions).

So instead let's use the $\langle O(n \log n), O(1)\rangle$ sparse table for both the top and bottom RMQ structures, and a block structure of size $\log n$. Thus, the preprocessing time is

$$\begin{aligned} p(n) &= O\left(n + p_1\frac{n}{b} + \frac{n}{b}p_2(b)\right) \\ &= O\left(n + \frac{n}{\log n}\log n \log\log n\right) \\ &= O(n \log\log n). \end{aligned}$$

Then, the query time is $O(1)$! So we have $\langle O(n \log\log n), O(1)\rangle$. $\log\log n$ is an interesting function, very small for any number you care about.

If instead you use the sparse table for the top and the $\langle O(n), O(\log n)\rangle$ structure for the bottom, then with $b = \log n$, one gets the complexity $\langle O(n), O(\log\log n)\rangle$.

One function that pops up on the homework is the log-star function. $\log^* n$ is the number of times you have to compute $\log(\log \ldots \log n)$ until it drops below 1. This turns huge numbers into very small ones (Keith almost called them astronomical, except they are much larger than the universe, e.g. $\log^*\left(2^{2^{2^2}}\right) = 4$).

We see these solutions trending to $\langle O(n), O(1)\rangle$, and it turns out there is such a solution, but it looks like dark magic.

2. Cartesian Trees and RMQ: 4/2/14

"Well, we're theoreticians, so we say, 'actually, the universe is pretty small. . . ' "

We saw on Monday lots of different structures and/or algorithms for the RMQ problem; in particular, one can break an array into a block structure and then compute the RMQ using hybridized algorithms, one for the large blocks and one for the individual entries within the blocks. If we use a $\langle p_1(n), q_1(n)\rangle$-time solution on the top and a $\langle p_2(n), q_2(n)\rangle$-time solution within each block, and the block size is b, then the hybrid structure has runtime $\langle O(n + p_1(n/b) + n/bp_2(b)), Q(q_1(n.b) + q_2(b))\rangle$.

Another trick we'll leverage today is that if one uses a sparse table as a top structure, the construction time is $O((n/b)\log n)$, so if $b = \Theta(\log n)$, then the construction time is $O(n)$!

We were trying to find an $\langle O(n), O(1)\rangle$-time structure. This means we need $p_2(n) = O(n)$ and $q_2(n) = O(1)$. So that's kind of recursive, and suggests that it's not possible. But fortunately, there's a little detail we've been ignoring.

Reframe the problem as solving RMQ on a large number of small arrays with $O(1)$ query time, but *average* preprocessing time $O(n)$. This isn't exactly the same problem, which allows us to find the solution we want.

Consider the arrays $[10, 30, 20, 40]$ and $[166, 361, 261, 464]$. These arrays look very different to us, but are identical with respect to an RMQ: if one uses the same indices for an RMQ for these two arrays, then the minimum is at the same index. This suggests that we should be able to recycle the structures we've set up already. Thus, *from this point forward, let* $\text{RMQ}_A(i, j)$ *denote the* index *of the minimum value in the range, rather than the value itself.*

Definition. With this redefinition, given two blocks B_1 and B_2 of length b, say that B_1 and B_2 have the same block type, denoted $B_1 \sim B_2$, if for all $0 \le i \le j < b$, $\text{RMQ}_{B_1}(i, j) = \text{RMQ}_{B_2}(i, j)$.

This is an equivalence relation. Thus, if we can somehow detect that two blocks in an array have the same RMQ structure, then we can share the structure, saving preprocessing time, so the algorithm will do something like compute the block type, and reuse an old type if it already exists; if not, then store the new structure.

Here are some nuances: the block size must be chosen such that there aren't too many possible block types, so that we do in fact reuse RMQ structures, but the blocks aren't too small such that we can't efficiently build the summary structure on top.

Furthermore, we need to be able to check if two blocks have the same block type, but right now, it's only defined in terms of RMQ, so it's not *a priori* useful: we need to have RMQ to compute RMQ, at least right now. It seems reasonable to define the permutation type of a block to be the what order the maximum, second maximum,

etc., in, but this is too fine: the following three blocks have the same block type, but different permutation types: [261, 268, 161, 167, 166], [167, 261, 161, 268, 166], and [166, 268, 161, 261, 167]. It's also a problem that the number of possible permutations of a block is $b!$ — which is way too large, introducing a constraint on the block size.

Claim. If $B_1 \sim B_2$, then the minimum elements of B_1 and B_2 must be at the same position. (Assume that we're talking about the first occurrence of the minimum element.)

Proof. Compute RMQ$(1, n)$, which must be the same for B_1 and B_2. ⊠

Claim. Thus, this property must also hold recursively on the subarrays to the left and right of the minimum.

So it turns out there's a data structure (of course) that can be used to keep track of these minima called a Cartesian tree. These pop up in places you'd never expect within CS. It's a binary tree derived from an array as follows:

- The empty array has ab empty Cartesian tree.
- For a nonempty array, the root stores the index of the minimum value, and its left and right children are the Cartesian trees for the subarrays to the left and the right of the minimum, respectively.

Theorem 2.1. *Let B_1 and B_2 be blocks of the same length b. Then, $B_1 \sim B_2$ iff B_1 and B_2 have equal Cartesian trees.*

Proof sketch. In the forward direction, proceed by induction. Since B_1 and B_2 have the same RMQs, then their corresponding ranges have the same minima. But then, this holds true for the left and right subarrays, so by induction, the chidren of the root are also the same.

Conversely, we will show that it's possible to answer RMQ using a recursive walk on a Cartesian tree. This can also be shown inductively: if the interval contains the minimum, then return the minimum; if not, it is entirely contained in one of the left or right subarrays, so one can throw away the other half of the tree and recurse. Thus, B_1 and B_2 have the same RMQs, so they have the same block type. ⊠

One thing I didn't expect to see today is that we can answer the lowest common ancestor problem on a tree (when did two things diverge from a lowest common ancestor) with an Euler tour! (Pre- and post-order traversal, in some sense.) Basically, do an Euler tour of a tree ad store the depth at the index in question. Then, using RMQ, one obtains the lowest common ancestor (and using a related structure called a suffix tree, answer a whole bunch of problems about string processing.) But it's also possible to solve RMQ using lowest common answer...

Now, back to the show. We want to construct Cartesian trees: do a linear scan over the array, and identify the minimum; then, recursively process the left and right halves. This is just Quicksort... so it will take time $O(n \log n)$ to $O(n^2)$, depending on where the minima are. We can do better.

It turns out it's possible to build a Cartesian tree in linear time, by going from left to right: if one already has a tree for the first k elements, adding one more isn't too bad. Firstly, it cannot be a left child of anything, because then its parent would come after it. And, similarly, none of its ancestors can be left children. Thus, it must be somewhere on the right spine. Finally, notice that Cartesian trees are min-heaps for the original array (and indices). So just look at the right spine, and just ask, where does $A[k+1]$ belong in the sequence? It belongs between two elements, so splice it in between them: the right child of the thing before it, and the thing after it as a left child.

But that's not obviously linear, so let's make a stack of the right spine. Thus, to insert a new node, pop things off until something less than the new value is there, and then set the new node's right child to be the last value popped (or null, if nothing was popped), and its parent to be the top node on the stack (or null, if the stack is empty); then, push the new node onto the stack. This naïvely has time $O(n^2)$, but it's actually $\Theta(n)$, because the work done per node is proportional to the number of stack operations performed when that node was processed... but each thing is only popped off the stack once and pushed onto the stack once. Thus, the total number of operations is $\Theta(n)$, and this is the fastest possible, since you need to look at every node in the tree.

Theorem 2.2. *The number of Cartesian trees for an array of length b is at most 4^b, and is actually related to the Catalan numbers*

$$\frac{1}{b+1}\binom{2b}{b} \sim \frac{4^b}{b^{3/2}\sqrt{\pi}}.$$

Now, we want to explicitly enumerate Cartesian trees in time 4^b.

Claim. The Cartesian tree produced by the stack-based algorithm is uniquely determined by the sequence of pushes and pops made on the stack.

Thus, we can write $2b$-bit numbers, where 0 corresponds to a push and 1 to a pop. We can pad the end with 0s. This number will be called the Cartesian tree number of a particular array (or tree, or block). So now, for determining whether two things have the same type, we don't need to build the Cartesian tree, and can instead just use the stack algorithm to compute the Cartesian tree number in linear time.

Now, combining all of these things together, one obtains the overall RNQ structure. In 2005, Fischer and Heun introduced a slight variation on this structure; but we will use a hybrid approach of block size b (which we'll specify later), with a sparse table as the top RMQ structure, and the full precomputation data structure within each block. However, modify this as follows:

- Make a table of length 4^b holding RMQ structures, with index corresponding to Cartesian tree number, and initially empty.
- When computing the RMQ for a particular block, store its structure at the corrsponding place in the table, or, if that entry already exists, just return a pointer to that location.

Now, we already have $O(1)$-time top-level queries, and the preprocessing time is as follows: the sparse table takes time $O((n/b)\log n)$, and the Cartesian trees take time $O(4^b b^2)$, so the total time is $O(n + (n/b)\log n + 4^b b^2)$ (since $O(n)$ time to split the blocks).

Exponentials, such as the 4^b above, are bad. So let's set $b = \Theta(\log n)$, so that $(n/b)\log n = O(n)$, as shown before, and $4^b b^2 = O(n)$. So the total preprocessing time is $O(n)$. (In fact, if $b = (1/2)\log_4 n$, then $4^b b^2 = o(n)$!)

Though this looks really contrived and ugly, it really makes sense in the context of reusing the RMQ structures, especially in the asymptotic case. In the real world, the $\langle O(n), O(\log n)\rangle$ hybrid usually has better constants, unless you're dealing with really large inputs; see the Fischer-Heun paper for more details.

One last little nuance is that the asymptotics assume that the Cartesian tree numbers fit into individual machine words... but if $b = (1/4)\log_2 n$, then each Cartesian tree number will have $(1/2)\log_2 n$ bits, so the Cartesian tree numbers will fit into a machine if n fits into a machine word. But in the *transdichotomous machine model*, we assume that the problem size always fits into a machine word, which is reasonable (in practice, for example, we're moving to 64-bit machines).

The technique used here is an example of the Method of Four Russians (at most one of whom was Russian... but I digress).

Here are some reasons we looked at RMQ first:

- There are many different approches to the same problem; different intuitions lead to structures with different runtimes.
- Data structures are built on data structures, much in the same way that algorithms are built on top of algorithms. This is common of modern data structures.
- The Method of Four Russians, which enumerates data structures to get a speedup, looks like magic, but still shows up a lot in practice.
- This is an example of a modern data structure; 2005 wasn't that long ago.