6.897: Advanced Data Structures Spring 2005

Lecture 15 — March 31, 2005

Prof. Erik Demaine *Scribe: Austin Clements*

1 Overview

In this lecture we introduced the topic of *range queries* with the static *range minimum query* (RMQ) and *lowest common ancestor* (LCA) problems.

In the static RMQ problem, we have a static array A of n numbers and we want to be able to efficiently find the minimum element of any contiguous subarray of A. In other words, we want to answer queries of the form $RMQ(i, j) = \min(A[i], \ldots, A[j])$. For example, 15 is the result of RMQ over the indicated range of the following array

$$17\ 0\ \underbrace{36\ 16\ 23\ \mathbf{15}\ 42}\ 18\ 20$$

To do this we will transform the RMQ problem into an LCA problem such that an efficient solution to LCA will yield an efficient solution to RMQ. In LCA we are given a tree and two nodes a and b and we want to find the lowest node of the tree that is a common ancestor of both a and b.

LCA was originally reduced to $O(1)$ time per query with $O(n)$ space by Harel and Tarjan [4]. Cole and Hariharan developed a dynamic version of LCA that achieves $O(1)$ time per operation and allows for insertion and deletion of keys as well as subdivision and merging of edges [2]. The simpler version of LCA discussed in this lecture was introduced later by Bender and Farach-Colton and achieves the same bounds as the original static result [1]. Using it, we can also solve RMQ with linear space and constant query time.

We can naïvely solve RMQ with $O(1)$ time queries by simply storing a table over i, j of minimum values for each range. This table requires $O(n^2)$ space and $O(n^2)$ time to compute (using a dynamic program). The rest of this lecture concerns reducing this to $O(n)$ space and $O(n)$ construction time.

2 Reducing RMQ to LCA

Cartesian trees are due to Gabow, Bentley, and Tarjan [3]. We used Cartesian trees in Lecture 13 for fast sorting and will now revisit them for the purposes of reducing RMQ to LCA. A Cartesian tree has the following structure

- *Root* – Minimum element of A. Suppose this is $A[i]$.
- *Left subtree* – Cartesian tree on $A[1], \ldots, A[i-1]$.
- *Right subtree* – Cartesian tree on $A[i+1], \ldots, A[n]$.

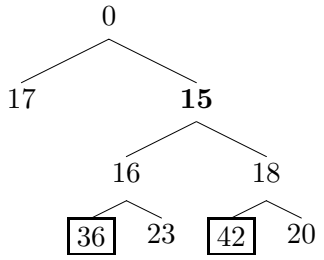

Figure 1: The Cartesian tree of $17, 0, 36, 16, 23, 15, 42, 18, 20$. The same RMQ as before is shown as the LCA of the boxed elements.

Cartesian trees can be constructed in $O(n)$ time, as we saw in Lecture 13. Figure 1 shows an example Cartesian tree.

To reduce RMQ to LCA, we put the elements of the array A into a Cartesian tree. The RMQ of elements i and j is simply the LCA of the nodes corresponding to i and j in the Cartesian tree. Thus, if we can solve LCA in $O(1)$ time, we can solve RMQ in $O(1)$ time.

3 Lowest Common Ancestor

Now we just need to know how to solve LCA quickly. Surprisingly, we will do this by reducing LCA back to RMQ! The trick is that we'll reduce it to a special case of RMQ, which we'll call "RMQ$\pm$1". RMQ$\pm$1 is identical to the usual RMQ, except that adjacent values in the list can only differ by ± 1. This distinguishing property lets us solve RMQ$\pm$1 easier.

3.1 Reduce LCA to RMQ$\pm$1

To reduce LCA to RMQ$\pm$1, we take the Euler tour of the tree we're solving LCA for. However, instead of storing the values of the nodes as we visit them, we store their depths as shown in the second row of Figure 2. Observe that depths change by ± 1 as we consider nodes in an Euler tour traversal, because adjacent nodes are in a parent-child relation.

0	17	0	15	16	36	16	23	16	15	18	42	18	20	18	15	0
0	1	0	1	2	3	2	3	2	1	2	3	2	3	2	1	0
									$\Uparrow$							

Figure 2: The Euler tour of the tree in Figure 1, showing the elements and their depths. The box encloses the two elements we want to find the LCA of and the arrow points to the element that is both their LCA and the RMQ of either row over the enclosed range.

By taking the RMQ$\pm$1 over the sequence of depths between the two elements we want to find the LCA of, we can find the index of the element with the least depth. This element is precisely the LCA.

3.2 Apply Indirection

Now that we've reduced the problem to RMQ±1, we need to solve RMQ±1 in $O(1)$ time. This time we won't reduce to LCA. Instead, we'll apply indirection as shown in Figure 3. We split our list of n elements into $2n/\lg n$ groups, each of size $1/2 \lg n$. From each group we find the minimum element and produce a summary structure of the $2n/\lg n$ minimums, which we store in another RMQ array.

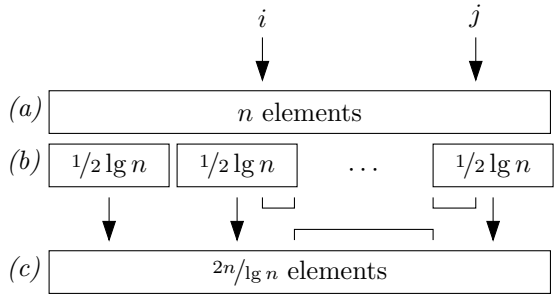

Figure 3: Applying indirection. *(a)* The n element list. We wish to find the minimum element between elements i and j. *(b)* The same elements, this time split into $2n/\lg n$ groups, each of size $1/2 \lg n$. *(c)* A summary RMQ data structure on the minimum element of each group.

The minimum between i and j is thus the minimum of

1. the elements in i's group between i and the end of the group,
2. the elements in j's group between the beginning of the group and j,
3. and the minimums of all of the groups between i's group and j's group.

In other words,

$$RMQ(i,j) = \min \begin{cases} RMQ(i, \infty) \text{ in } i\text{'s group} \\ RMQ(-\infty, j) \text{ in } j\text{'s group} \\ RMQ(> i\text{'s group}, < j\text{'s group}) \text{ in the summary structure} \end{cases}$$

3.3 Use a Lookup Table for Groups

By constructing a lookup table mapping from a tuple of $\langle$group type, i, $j\rangle$ to the index of the minimum element in the group between elements i and j, we can answer a query for the minimum element of any range in any group in $O(1)$ time.

We begin by observing that the index of the minimum element is invariant under translation. Thus, we can subtract k from every element of the group such that the first element is translated to 0 without affecting the index of the minimum element in the group.

This leads to $2^{1/2 \lg n} = \sqrt{n}$ distinct group types. Note that this is only true because we're solving the ±1 case and not the general case. Thus, each element can be represented as one of two values indicating its delta from the previous element. i and j can each take on one of $1/2 \lg n$ possible

values, leading to a total of $(1/2 \lg n)^2$ possible queries. The result of a query requires $O(\lg \lg n)$ bits, the number of bits required to represent the index of the minimum element.

All together, this means the lookup table requires $O(\sqrt{n} \lg^2 n \lg \lg n) = o(n)$ bits.

3.4 Using RMQ for the Summary Structure

We've dealt with finding the RMQ of i's group and j's group, but we still have to find the RMQ over the summary structure. Note that the elements of this structure may not satisfy the ± 1 property, so general RMQ must be used. Using RMQ recursively would fail to achieve $O(1)$ queries. Using the trivial $O(n^2)$ RMQ algorithm is insufficient to achieve the desired bounds. We need something smarter.

Instead, we'll use something similar to the trivial algorithm, but with a *sparse table*. Instead of storing the answers for any start point and any interval length, we'll only store the answers for any start point and intervals of length 2^i, where $i = 0, \ldots, \lg n$. This leads to n choices for the starting point and $\lg n$ choices for the interval length, yielding $O(n \lg n)$ space. Furthermore, this table can be computed in $O(n \lg n)$ time using dynamic programming.

This is sufficient for computing RMQ because overlapping intervals will not affect the minimum. As illustrated in Figure 4, any interval of length k can be broken down into at most two overlapping ranges of length $2^{\lfloor \lg k \rfloor}$.

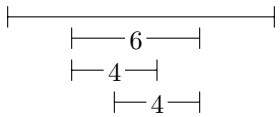

Figure 4: A range over 6 elements can be broken down into two ranges over $2^2 = 4$ elements.

Thus, we now have an $O(n \lg n)$ space and time algorithm for computing a structure supporting $O(1)$ RMQ queries over the summary structure from section 3.2. Since the summary structure actually contains $n' = O(n/\lg n)$ elements, the space and preprocessing time is linear in the original n.

4 Summary

Computing the Cartesian tree for the original array takes $O(n)$ time and space. Using indirection, computing the structures for LCA takes $O(n)$ time. Thus, construction of the RMQ structure takes $O(n)$ time. Each step of the query takes $O(1)$ time and there are a constant number of steps, so queries are supported in $O(1)$ time.

References

[1] M. A. Bender, M. Farach-Colton, *The LCA Problem Revisited*, LATIN 2000: 88-94.

[2] R. Cole, R. Hariharan, *Dynamic LCA Queries on Trees*, SODA 1999: 235-244.

[3] H. N. Gabow, J. L. Bentley, R. E. Tarjan, *Scaling and Related Techniques for Geometry Problems*, STOC 1984: 135-143.

[4] D. Harel, R. E. Tarjan, *Fast Algorithms for Finding Nearest Common Ancestors*, SIAM Journal on Computing, 13(2): 338-355, 1984.