15-451/651: Algorithms **CMU, Spring 2015**
Lecture #26: Lowest Common Ancestors and Range Min Queries April 27, 2015
Lecturer: Danny Sleator

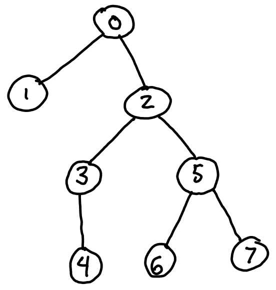
1 Introduction

This lecture is about two closely related problems: Lowest Common Ancestors (LCA) and Range Min Queries (RMQ).

The LCA Problem

Given a tree T with nodes numbered $0, 1, \ldots, n-1$ we want to preprocess it so we can answer queries of the following form:

$\text{LCA}_T(i, j)$: Compute the lowest common ancestor (LCA) of i and j in T. This is the unique node in the tree that is an ancestor of both i and j, and farthest from the root of the tree.



For example, in the tree above, $\text{LCA}(4, 7) = 2$ and $\text{LCA}(6, 0) = 0$.

The RMQ Problem

Given an array $A[0], A[1], \ldots, A[n-1]$ preprocess it so we can answer queries of the following form:

$\text{RMQ}_T(i, j)$: Compute an index k in $[i, j]$ where A attains its minimum value. That is, k is an element of $\text{argmin}_{k\in[i,j]} A[k]$

Any solution to either of these problems can be characterized by the preprocessing time and the query time. Thus we will evaluate solutions to these problems with with a pair of functions bracketed by $\langle$ and $\rangle$. The ideal bounds for both of these problems is $\langle O(n), O(1)\rangle$.

These notes explain how the LCA problem can be reduced to the RMQ problem in a way that preserves the complexity bounds. (It's also the case that the RMQ can be reduced to LCA, although this lecture does not cover this.)

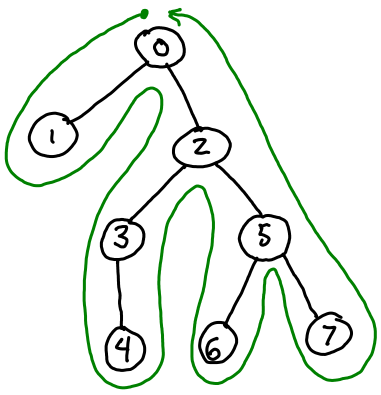
We also present two solutions to th RMQ problem:

Solution 1: $\langle O(n \log n), O(1)\rangle$
Solution 2: $\langle O(n), O(1)\rangle$

Solution 2 is actually for a special case of the RMQ problem where the each successive element of the input array differs from the previous one by exactly one.

2 How to reduce LCA to RMQ

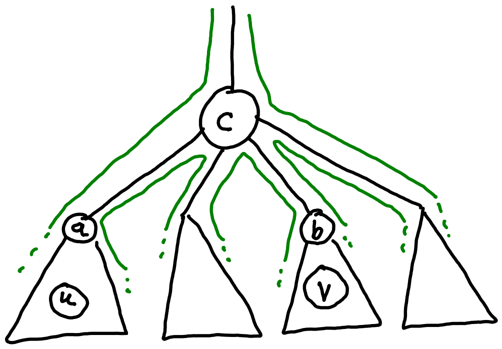
An *Euler tour* of a rooted tree visits the vertices of the tree in a specific order. A node can be visited more than once. The traversal starts at the root, then it visits the first subtree of the root (recursively), then the root, then second subtree of the root (recursively), then the root, etc. It ends with a visit to the root.



The Euler tour of this tree visits the nodes in the following order: (0,1,0,2,3,4,3,2,5,6,5,7,5,2,0). Note that if the tree has n nodes in it, then the Euler tour has $2n-1$ elements.

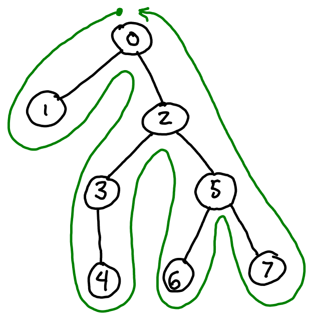
Lemma: Let T be a rooted tree, and let E be the Euler tour of T. Let i and j be two indices into the Euler tour. Then the $\mathrm{LCA}_T(E[i], E[j])$ occurs in the Euler tour between i and j.

Proof: Let $u = E[i]$ and $v = E[j]$, and let $c = \mathrm{LCA}_T(u, v)$. If $u = c$ or $v = c$ then the lemma is trivial. So assume that neither equality holds.



We know that u and v are in the subtree rooted at c. We also know that u and v are in *different* subtrees of the children of c (otherwise that child would be the $\mathrm{LCA}_T(u, v)$). (See figure above.) Let these subtrees be rooted at nodes called a and b. The Euler tour will include an instance of c between its traversal of the subtree rooted at a and the one rooted at b. (One property of the Euler tour is that all the nodes in a subtree form a contiguous block in the Euler tour.) Since u occurs during the traversal of a and v occurs during the traversal of b we know that c must occur between u and v in the Euler tour. Q.E.D.

Now here's how we can solve the LCA using RMQ. We construct three arrays. The Euler tour (explained above) is put into an array called E. Then there's an array for the depth, which we call L, which stores for each node in the Euler tour array, the depth of that node in the tree. (The root has depth 0.) Finally we construct the first occurrence array, called H. $H[i]$ is the index of the first occurrence of i in the Euler tour. These arrays can all be computed in linear time. Here are the arrays for the tree above:



Euler tour

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E:	0	1	0	2	3	4	3	2	5	6	5	7	5	2	0

Depth

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L:	0	1	0	1	2	3	2	1	2	3	2	3	2	1	0

First occurrence in E

	0	1	2	3	4	5	6	7
H:	0	1	3	4	5	8	9	11

Now we can solve the LCA problem. We first build the RMQ data structure for the array L. (We use L because the LCA is going to be the shallowest node on that region of the Euler tour.) Here's the algorithm:

$$\mathrm{LCA}_T(u, v) = E[\mathrm{RMQ}_L(H[u], H[v])]$$

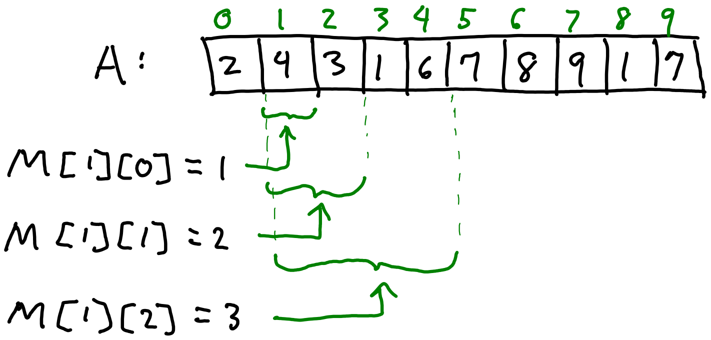
Note: This assumes that $H[u] \leq H[v]$. If this is not the case then we have to swap u and v.

3 An $\langle O(n \log n), O(1) \rangle$ Solution to RMQ

The algorithm described here is called the "sparse table" method. The input is $A[0 \cdots n-1]$. We compute a 2-D table $M[0 \cdots n-1][0 \cdots \lceil \log_2 n \rceil]$. Here's what this is going to mean:

$$M[i][j] = \text{The index of a minimum value in } A[i \cdots i + 2^j - 1].$$

In other words, this is the answer to the RMQ for a range of length 2^j starting from i. Here's an example:

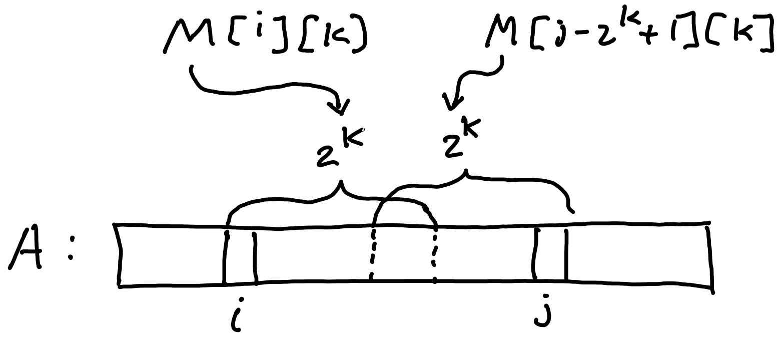


Here's a recurrence we can use to compute this table in $O(1)$ time per entry. Note that $M[i][j]$ will only be needed when $i + 2^j - 1 \le n - 1$.

```
M[i][j] =   if $j = 0$ then $i$ else
            let $m_1 = M[i][j-1]$ in
            let $m_2 = M[i + 2^{j-1}][j-1]$ in
            if $A[m_1] \le A[m_2]$ then $m_1$ else $m_2$
```

So this gives us the solution to a RMQ when the length of the query is a power of 2. For arbitrary lengths we can compute the answer with just two lookups in the table. These two intervals overlap. This is not a problem since the minimum value in a set with duplicates is the same as the minimum of a set without duplicates. Here is the code and and illustration.

```
$\text{RMQ}_A(i,j)$ =   let $k = \lfloor \log_2(j - i + 1) \rfloor$ in
                 let $m_1 = M[i][k]$ in
                 let $m_2 = M[j - 2^k + 1][k]$ in
                 if $A[m_1] \le A[m_2]$ then $m_1$ else $m_2$
```



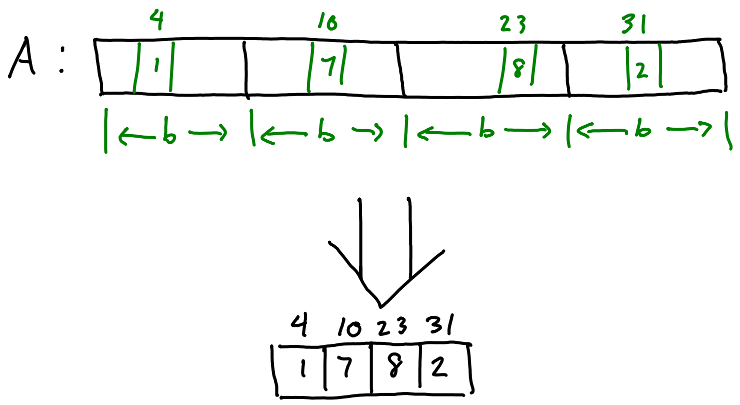
The solution overall is $O(n \log n)$ space and time, and lookups are $O(1)$.

4 Improving this to $\langle O(n), O(1)\rangle$

By the use of some clever tricks and a simplifying assumption about the input array, the preprocessing time and space can be reduced to $O(n)$ while preserving the $O(1)$ query time.

The assumption that we will make is that the array A has the property that each successive element of it differs from the previous one by ± 1. This holds for the LCA array L that we used in the reduction from LCA to RMQ. Thus the technique is applicable to LCA, and gives a $\langle O(n), O(1)\rangle$ solution to LCA.

The first trick is to divide the array A into blocks. Let b be the block size. (You will see later how this is picked.)



For each block we compute an index of a minimum value in the block. We replace the block by that single minimum value, along with its index. We can apply the sparse table algorithm of the previous section to this data. This will allow us to do RMQ queries that respect the block boundaries.

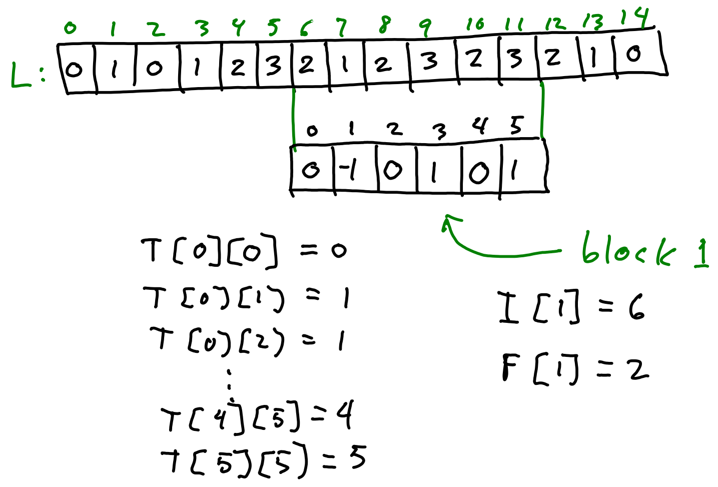
Let's pick the block size $b = \lfloor \frac{\log n}{2} \rfloor$. The length of the array fed into the sparse table method is n/b. So the time to construct the sparse table (and the space used) is:

$$O(\frac{n}{b} \log \frac{n}{b}) = O(\frac{n}{\log n} \log n) = O(n)$$

A general query will have a starting point in the middle of a block, span zero or more complete blocks, and have an ending point in the middle of a block. The middle part of this can be handled using the method described in the previous paragraph. It only remains to show how to handle the parts of a query that span part of a block. Once this is done, the (up to) three parts of the query can easily be combined to compute the answer.

Handling the Blocks

Let the blocks be numbered, starting with 0 on the left. Let k be a block number. We define the start index of a block $I(k) = k * b$. We also have $F(k) = A[I(k)]$, the first number in a block. We now replace the numbers in the block, by subtracting $F(k)$ from all the numbers in block k. This is called normalizing a block. We will also build a table $T[0 \cdots b-1][0 \cdots b-1]$. This table stores the RMQ answers for all i and j, where i and j are indices starting from the start of the block. The figure below gives an example.



This data structure allows us to answer queries within a block in $O(1)$ time. But how much space is needed for this and how much time does it take to compute it? The time and space is $O(b^2) = O((\log n)^2)$. At first glance it appears we need to do this for $O(n/(\log n))$ blocks, so we have not dealt with the $O(n \log n)$ problem.

This problem is solved when you realize that the number of possible different normalized blocks is only 2^{b-1}. (Every block begins with a 0, and each subsequent number differs from the previous one by ± 1.) So instead of storing the entire table for each of the n/b blocks, we only need to do it for $2^{b-1} \leq 2^{\frac{\log n}{2}} = \sqrt{n}$ blocks. So the space and time used to construct these tables is $O(\sqrt{n} b^2) = O(\sqrt{n}(\log n)^2) = o(n)$

We need another array for the block, called $BN[k]$. This is the pattern of $b-1$ bits that define the ups and downs of the block. This will tell us, for each block, which of the $T[][]$ tables we pre-computed to use for the query.

Given all of this, here's the algorithm for RMQ within a block k.

$$\mathrm{RMQ}_k(i,j) = I(k) + T_{BN[k]}[i][j]$$

It's easy to combine all these pieces together to give the complete solution.

Refrences

`https://www.topcoder.com/community/data-science/data-science-tutorials /range-minimum-query-and-lowest-common-ancestor/`