6.851

André Schulz' Notes

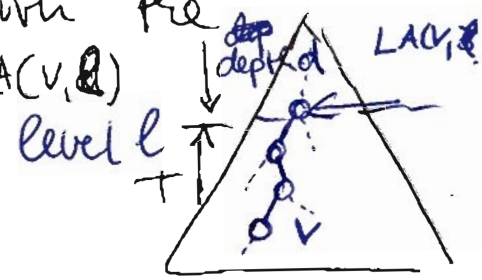
Lecture 8

The level ancestor problem
LAQ

[Dietz '91]
[Bender Farach-Colton 03 (Simplified)]

Problem: Input: rooted Tree T with n vertices

Query: For node v, return the depth ~~level~~ d-~~th~~ ancestor LA(v,d)



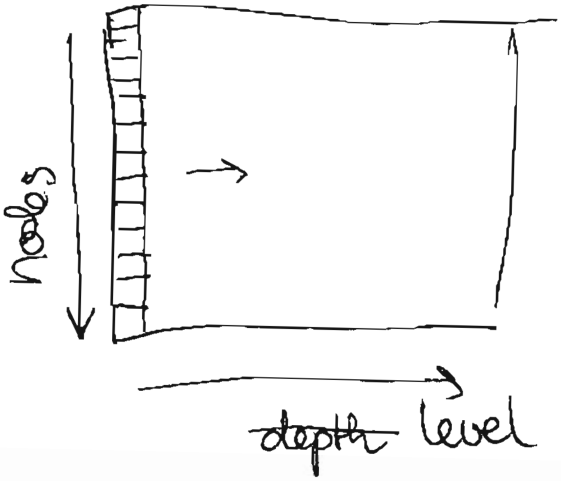
Equivalent LA(v,d) ← depth
$d+\ell = \text{depth}(v)$

Notation: An algorithm that needs $f(n)$ preprocessing & $g(n)$ query time is denoted as $\langle f(n), g(n) \rangle$

Goal: Find a $\langle O(n), O(1) \rangle$ Algorithm

↳ we present a series of improvements to achieve the $\langle O(n), O(1) \rangle$ bound

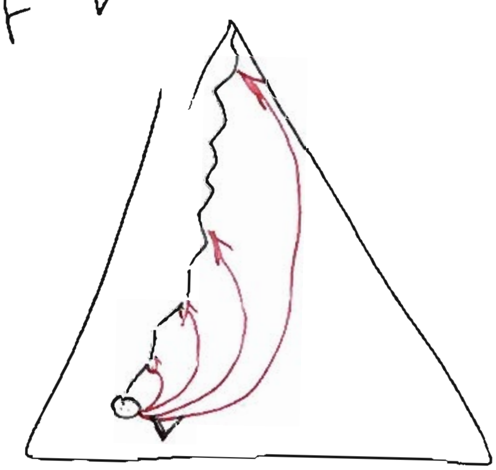
Alg A	$\langle O(n^2), O(1) \rangle$
	Table look-up



Array that can be filled column by column using a simple dynamic program.

Alg B	$\langle O(n \log n), O(\log n) \rangle$
	Jump - pointers

- we store pointers at every node v that jump up to the 1, 2, 4, 8, ... depth(v) ancestor of v



using the pointers 1x we get at least halfway to LA(v, d)

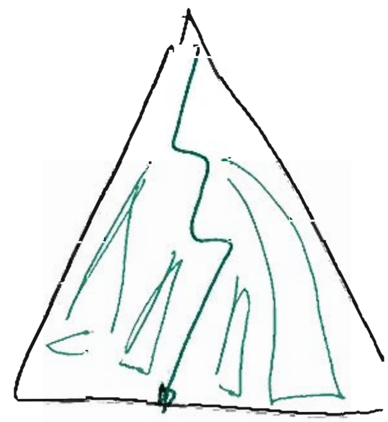
↳ O(log(n)) query time

→ we need O(n log n) storage, computing is easy, using dynamic programming (we fill a n × log n Array)

Alg C	$\langle O(n), O(\sqrt{n}) \rangle$
	Long Path decomposition

- We break T into disjoint paths as follows

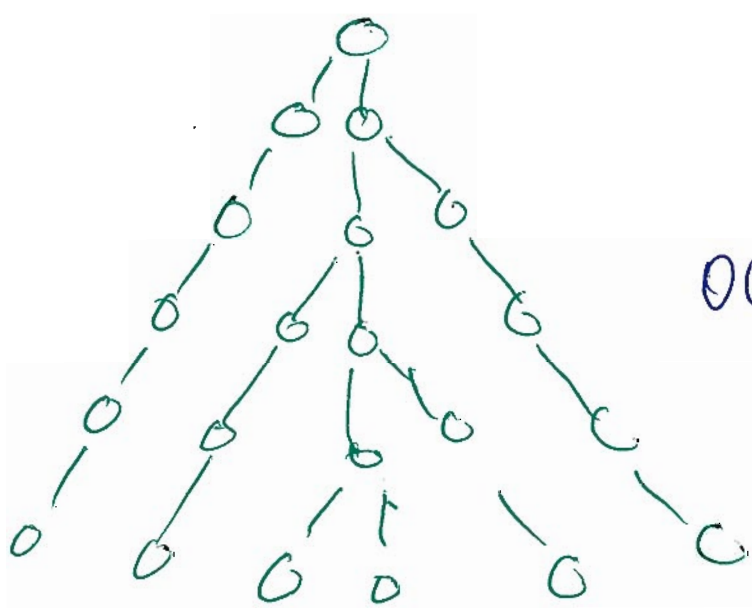
→ ~~select~~ remove the largest path (root → leaf) & recurse on the connected components



→ Store every path as an array and "connect" it to its "parent"

→ connect every node in T to its entry in the path decomposition

→ max depth of the path decomposition is $\Theta(\sqrt{n})$

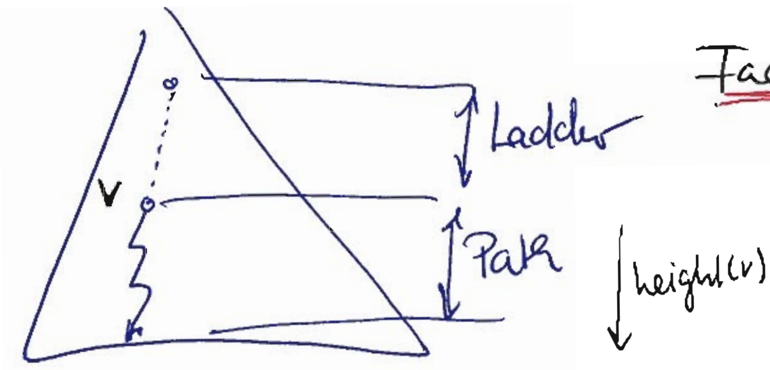


$O(\sqrt{n})$ Example

Storage + prep. $= O(n)$

Alg D	⟨O(n), O(log n)⟩
	Ladder Decomposition

- use the longest path decomposition, but extend every path by ~~its length~~ a factor of 2 (or less if the root "comes")



Fact: v lies on a path in the decomposition of length at least height(v) !

→ using the ladder we double the height or reach the root

↳ O(log n) query time

Alg E	⟨O(n log n), O(1)⟩
	Ladders + jump pointers

Ladder : exponentially increasing hops up the tree
jump pointers : exponentially decreasing hops up the tree

- We do the preprocessing for Alg B & Alg D
  → $O(n \log n)$

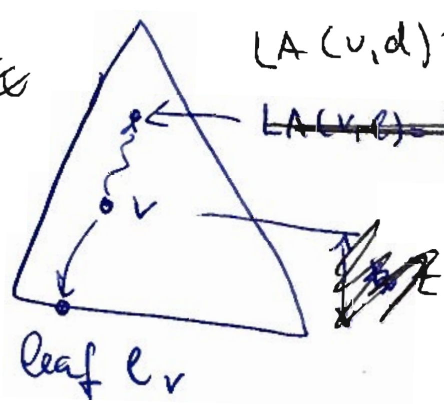
- A query does the following
  - use a jump pointer to reach v' } We jumped up at least $\ell/2$ nodes
  - climb a ladder from v' } v' lies on a path $\geq \frac{\ell}{2}$ ⇒ ladder goes up $\geq \frac{\ell}{2}$ over v'

  we reach $LA(v,\ell)$ da 1 jump + 1 ladder climbing

→ Small improvement:
- Store jump pointers only at leaves
- Store pointers for every node to some ~~le~~ leaf descendant

Jump pointers can be computed using the ladder decomposition, not! dynamic progr. - see later (*)

→ ~~So~~



$LA(v,d) = LA(\ell_v, d)$

~~$LA(v,\ell) = LA(\ell_v, \ell+z)$~~

Query: answer $LA(\ell_v, d)$

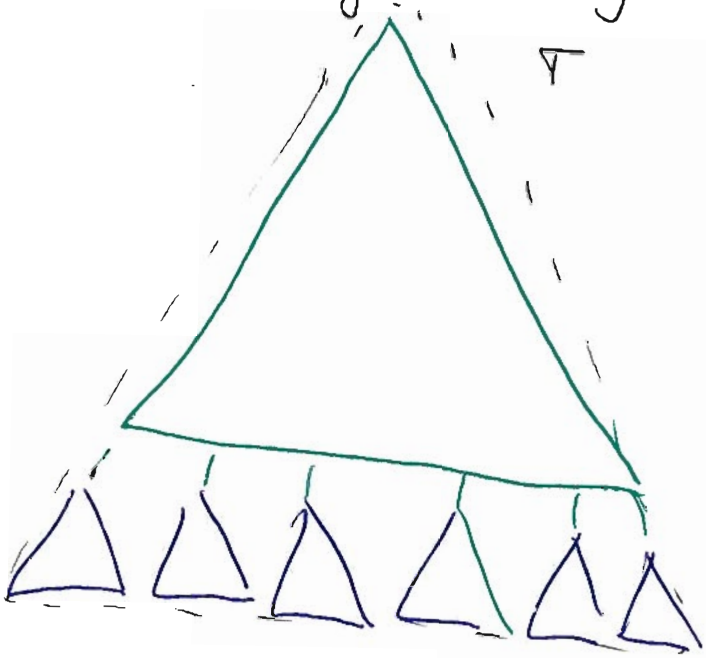
Preprocessing: $O(n + L \log n)$

L = # of leaves in T

Alg F	$\langle O(n), O(1)\rangle$
	Micro-Macrotree Algorithm

Idea: precompute small subtrees of T

↳ this gives only $O(n/\log n)$ leaves



Macrotree = remaining part

Microtrees = maximal trees of ~~###~~ $\leq \frac{1}{4}\log n$ nodes

- There are at most ~~Es gibt~~ $C_{\frac{1}{4}\log n}$ many distinct ~~trees~~ microtrees ← Catalan#

$$C_{\frac{1}{4}\log n} \leq 4^{\frac{1}{4}\log n} \leq \sqrt{n}$$

(more direct technique next slide)

- ~~The macrotree has at most~~
- we solve for any possible microtree the LA problem and store it ~~on~~ a look-up table

this takes $O(\sqrt{n} \cdot \log^2 n)$ time

- (*) leaf jump pointers can be computed in $O(\log n)$ time/lea
  using ladders $\Rightarrow$ total $O(n)$
  (every ladder doubles)

- Query $LA(v,d)$: if $v \in$ Macrotree
  - jump to leaf
  - jump-up-pointer (1x)
  - ladder climbing (1x)

  } (*)

  if $v \in$ Microtree
  - if e belongs to Micro-tree use look-up table
  - else go to the leaf in Macrotree and search with (*)

How to enumerate and bound the microtrees

→ every tree is specified by its up/down-edge sequence of a DFS ($|DFS| = 2 \cdot |E|$)

→ gives 0/1 pattern $= 2^{\frac{2}{4} \cdot \log \frac{n}{4}} = \cancel{2^{\log \frac{n^2}{4}}}\ 2^{\log \sqrt{n}} = \sqrt{n}$

→ we define table string $w \in \{0,1\}^{\sqrt{n}}$ the associated tree as the longest valid prefix P

valid: #0 = #1
∀ prefixes of P : #0 > #1

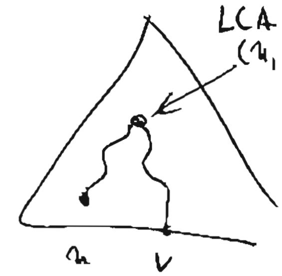
The LCA - Problem

[Bender, Farach-Colton 2000]

LCA

Input : Tree (rooted) with n nodes

Q : $LCA(u,v)$ returns the ~~least~~ lowest common ancestor of u, v

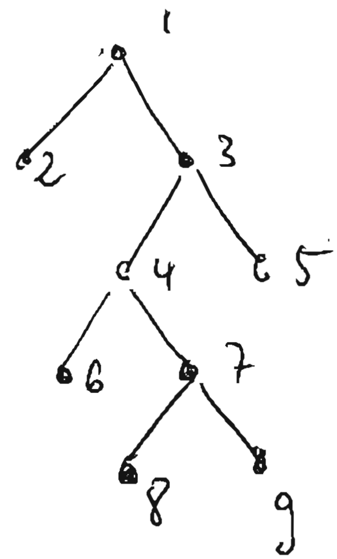


RMQ

Input : Array A

Q : $RMQ_A(i,j)$, index of the minimum element in $A[i,j]$

Lemma A $\langle f(n), g(n)\rangle$ Algorithm for RMQ induces a $\langle f(2n-1)+O(n), g(2n-1)+O(1)\rangle$ algorithm for LCA



E: Euler tour : Sequence of nodes traversed in DFS

$E = \{1,2,1,3,4,6,4,7,8,7,9,7,4,3,5,3,1\}$

L = Level array (for E)

$L = \{1,2,1,2,3,4,3,4,5,4,3,2,3,2,1\}$

$(|E| = |L| = 2n-1)$

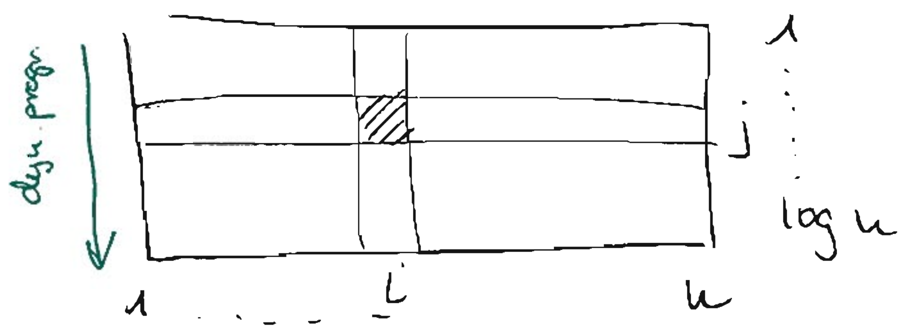
First occurrence of a node is stored in R (representatives)

$R = \{1, 2, 4, 5, 15, 6, 8, 11\}$

The LCA (u,v) is the shallowest node visited at the Euler Tour between $u \to v$, that is $E[RMQ_L(R[u], R[v])]$ $\square$

Alg A	$\langle O(n\log n), O(1)\rangle$
	SPARSE Table Algo. for RMQ A

We use a sparse look up table M

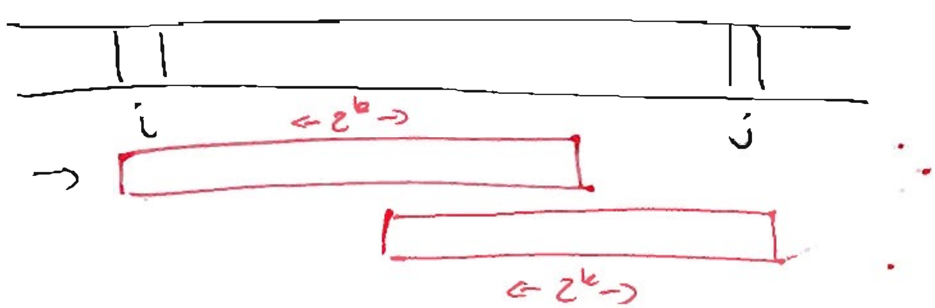


$M[i,j]$ stores the "min. index" $A[i, i+2^j-1]$

M has size $O(n\log n)$ and can be computed in $O(n\log n)$ time using dyn. progr.

$RMQ_A(i,j)$:

largest 2^k block between i,j



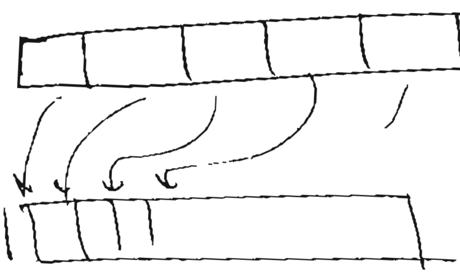
We find 2 Blocks that cover $A[i,j]$,
comparing their minima answers the RMQ query

Alg B	$\langle O(n), O(1)\rangle$
	Algorithm for RMQ±1

- RMQ±1 is RMQ but the entries $A[i], A[i+1]$ differ by ±1
- For solving LCA we have to solve RMQ±1

Idea: splitting A in blocks of size $\frac{\log n}{2}$

A



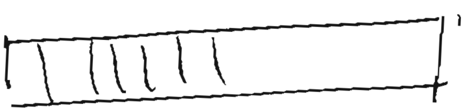
A'

$|A'| = O(n/\log n)$

A' stores the ~~RMQ±1 queries~~ minimum for each block

$|A'| = [1, \dots 2n/\log n]$

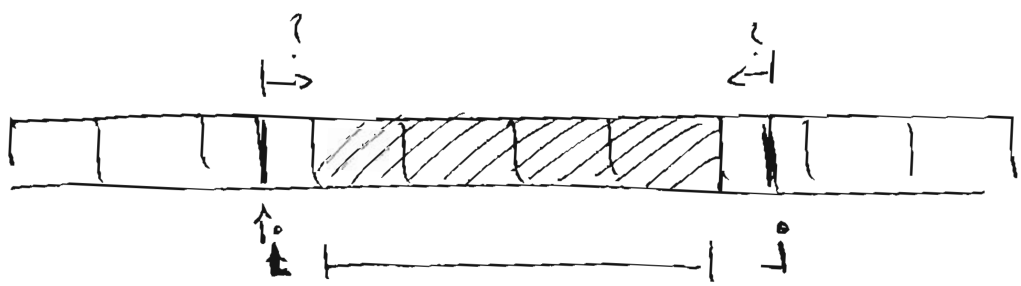
B'



$|B'|$

B' stores the index where the minimum is realized for each block

Queries:



RMQ here can be computed with Alg B for A'

We can compute $RMQ_A(i,j)$ by combining

1. $RMQ_A(i,k)$ k = end of block of i
2. $RMQ_A(k',j)$ k' = start of block of j
3 $RMQ_A(k,k')$ Alg A $\langle O(n), O(1)\rangle$ for $n' = n/\log n$

To answer 1. and 2. we precompute

<u>Fact</u>: If two blocks differ by the same number at each position they have the same RMQ answer

→ we can normalize all blocks

→ there are at most $2^{\log n/2} = \sqrt{n}$ normalized blocks

→ Store $\sqrt{n}$ tables of size $\log n \times \log n$

→ $O(\sqrt{n} \log^2 n)$ storage / time

Back to RMQ (general case)

- ~~The~~ $RMQ_A(i,j)$ equals the ~~LCA~~ (i,j) of the Cartesian tree of A !

- the cartesian tree can be build in linear time