Euler tours in trees
Euler tours (intuition)

Draw a tree in the plane, from the root down.

Trace your finger around the boundary of the drawing.

What order does it reach each vertex (multiple times)?

Each edge traced twice $\Rightarrow$ # vertices in tour $= 2n - 1$. 

a, b, d, b, e, b, f, h, k, h, f, b, a, c, g, i, g, j, g, c, a
Euler tours (pseudocode)

To produce an Euler tour of a subtree rooted at $x$:

- Output $x$
- For each child $y$ of $x$:
  - Recursively produce an Euler tour for the subtree rooted at $y$
  - Output $x$ again

Like

- preorder (root comes before children),
- inorder (root comes between children), and
- postorder (root comes after children),
all merged into one
Representing trees succinctly
If we represent trees with an object for each node, we can handle:

- Find the root
- Check if a node is the root, or is a leaf
- Go to the parent of a node
- Go to the first child of a node
- Go to the next child of the same parent

... by including a pointer to each of these things in each node

Space is $O(n)$ words of memory ... but we can do better!
Arbitrary trees with ordered children

Write nested parentheses for each node, surrounding string representing the subtree below it.

Convert to binary: “(” = 0, “)” = 1

\[
\begin{array}{cccccc}
0 & 00101011 & 01 & 000111 & 01 & 1 \\
( ( ( ) ( ) ( ) ) ) ( ) ( ( ( ) ) ) ) ( )
\end{array}
\]

Left child and its three grandchildren

\[n\text{-node tree } \Rightarrow 2n\text{-bit binary number}\]

Open parenthesis: parent-to-child edge in Euler tour

Close parenthesis: child-to-parent edge
Binary trees

Parentheses don’t work: can’t tell if single child is left or right

Traverse tree in preorder (root first, then children)

For each node, write down two bits:
Does it have a left child (0 for no, 1 for yes)?
Does it have a right child (0 for no, 1 for yes)?

$n$-node tree $\Rightarrow 2n$-bit binary number
Both codes are near-optimal

In each case, $\#$ trees is a Catalan number

$$C_n = \binom{2n}{n}/n + 1 \approx \frac{4^n}{\sqrt{n^3\pi}}$$

($C_n$ for $n$-node binary trees, $C_{n-1}$ for ordered trees)

So in order to have a different code for each tree, $\#$ bits must be

$$\geq \log_2 C_n = 2n - O(\log n)$$
Blocking

One long sequence of parentheses is concise but difficult to navigate

Break it up into blocks of $\frac{1}{2} \log n$ parens (represented as a sequence of $\frac{1}{2} \log n$ bits)

There are $\sqrt{n}$ different sequences of $\frac{1}{2} \log n$ bits $\Rightarrow$ we can build a table indexed by these sequences that lets us navigate within a block in constant time per step (tables take much less than $n$ space to store)

There are $O(n/ \log n)$ blocks $\Rightarrow$ we can use a less space-efficient structure to navigate from block to block and the small number of blocks will make it more efficient

We will see this same blocking strategy repeatedly all week!
Log-shaving principle

Using blocks can reduce memory requirement from $S(n)$ to $2n$ bits + $S(n/\log n)$

Sometimes repeatable for $O(1)$ levels, saving $\log n$ in space each time
The level-ancestor problem
The level-ancestor problem

Data: A tree

Query:

- Given a vertex $v$ and a number $k$, find the ancestor of $v$ that is $k$ steps higher in the tree
- Equivalently: Given a vertex $v$ and a number $d$ find the ancestor whose depth (number of steps from root) is $d$

We could just follow parent links up the tree in time $O(k)$ but we want small query time even when $k$ is large
Inefficient solution

Each node $v$ stores $O(\log n)$ ancestors, the ones $k$ steps higher for $k = 1, 2, 4, 8, \ldots 2^i, \ldots$

To find the ancestor $k$ steps higher when $k$ is not a power of two:

- Decompose $k$ into a sum of powers of two (its binary representation)
- Use stored ancestor pointers to jump up by each power

Space $O(n \log n)$, query time $O(\log k)$

We can reduce space but first we need to speed up queries
An easy special case

If the tree is a path, rooted at left end:

- Store an array $A$ of the vertices in the path
- Each node records its position in the array
- If $V$ is in $A[i]$, its $k$-step ancestor is in $A[i - k]$

Linear space, constant query time
Decomposition into long paths

Each non-leaf node selects one child, the one leading to the deepest leaf

The selected edges form a system of paths covering all non-leaf nodes and some of the leaves

Remaining leaves form one-vertex paths

But if we use the path solution for these paths, how do we find ancestors on different paths?
Extended paths

Instead of disjoint paths, extend each path to be twice as long (or all the way to the root if there is no ancestor twice as high)

Store each extended path in an array, and for each tree vertex store both which array it belongs to and its position in the array

Total length of all arrays $\leq 2n \Rightarrow$ linear space

Can answer some but not all queries: when $v$ is $h$ steps above a leaf, we can find ancestors $h$ steps above $v$
Combined data structure

Store both power-of-2 ancestors and extended paths

To find the ancestor $k$ steps higher from $v$:

- Make one power-of-2 jump, the biggest one that would still be below the ancestor
  
  $\text{Jump amount} = 2^{\lfloor \log_2 k \rfloor}$

- You are now high enough above a leaf to use extended paths

Constant query time but space is still $O(n \log n)$

(The paths use linear space but the jump tables are too big)
Level ancestors and Euler tours

To find the ancestor of $v$ at height $d$:

- Look in the Euler tour of the tree, starting from the last copy of $v$ in the tour
- The next vertex with height $d$ is the ancestor
Log-shaving

Structure:

- Break Euler tour into blocks of length \( b = \frac{1}{2} \log_2 n \)
- Within block, depths differ by \( \pm 1 \); label each block by its pattern of \( \pm \) choices, a \((b - 1)\)-bit binary number
- Precompute tables for queries with answer in same block
- Store power-of-two tables only at the last vertex of each block

Query:

- Use table to find answer in \( v \)'s block (if it is there)
- If not, vertex \( u \) at block end has same level-\( d \) ancestor as \( v \)
- Use the jump table stored at \( u \) to find an ancestor \( w \) high enough that we can use the extended paths for \( w \)