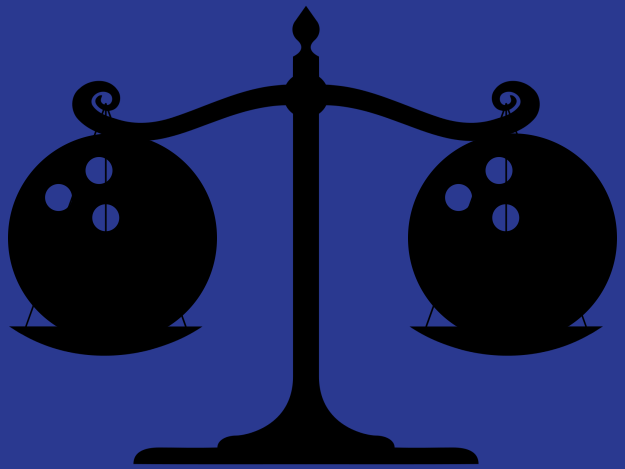
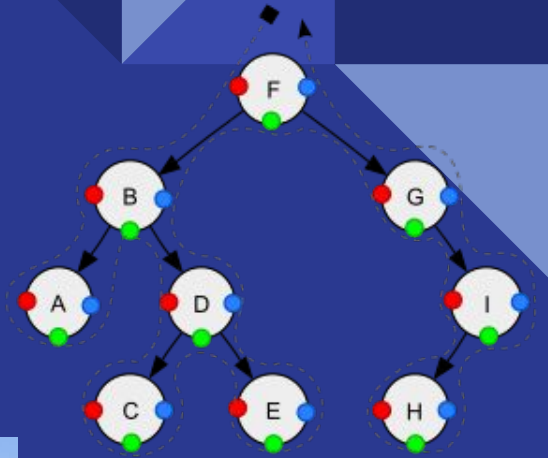




CS 9

Week 7 Problems

Andrew Benson
Ian Tullis

Announcements

- Our Tuesday meetings in Weeks 8 and 9 will be panels. So that all may speak freely, these will not be recorded. Please attend in person.
  - Panel 1: (tentatively) PM from Reddit, Engineer at Startup, + 1 more
  - Panel 2: (tentatively) Engineers from Jane Street, Two Sigma, + 1 more

- **We will adjust the required number of points from 25 to 23 to account for the lack of recordings.**

⭐ Problem 7-1: Bowling Balls

- You have a bunch of bowling balls ($3 \le N \le 10000$). Some miscreant removed the numbers on them, and they all look and feel kind of similar.
- You have been promised that:
  - at least two of the balls have different weights
  - at least two of the balls have the exact same weight
- You have a balance scale. You can put one bowling ball in each pan (the pans are too big to add more than this!) The scale tells you whether the left ball weighs more, the right ball weighs more, or they weigh exactly the same.

Write code / a strategy to, in as few weighings as possible **in the worst case**,

- **Warm-up:** Identify any pair of balls with different weights.
- **Harder:** Identify any pair of balls with the same weight.

Assume you have access to a function `compare(i, j)` that takes in indexes i, j and returns `-1` if i weighs more, `1` if j weighs more, and `0` if they weigh the same.

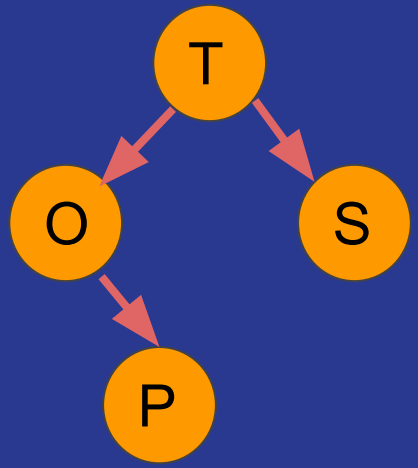
⭐⭐ Problem 7-2: Preposterous

- A *preorder* traversal of a binary tree does the following, starting from the root:
  - print the root value
  - print a preorder traversal of the left child tree, if any
  - print a preorder traversal of the right child tree, if any

- A *postorder* traversal does the same kind of thing but in a different order:
  - print a postorder traversal of the left child tree, if any
  - print a postorder traversal of the right child tree, if any
  - print the root value

⭐⭐ Problem 7-2: Preposterous

- And for an *inorder* traversal:
  - print an inorder traversal of the left child tree, if any
  - print the root value
  - print an inorder traversal of the right child tree, if any



Preorder: T, O, P, S
Inorder: O, P, T, S
Postorder: P, O, S, T

⭐⭐ Problem 7-2: Preposterous

- Suppose there is some unknown binary tree with N nodes, where $1 \leq N \leq 10000$.
  - Not necessarily complete!
  - Not necessarily a binary search tree!
- All you are guaranteed is that all the values are all different. (Think of them as numbers rather than letters)
- You are given one list representing the preorder traversal of the tree, and another list representing the postorder traversal of the tree.
- Your goal is to produce the **inorder** traversal of the tree, or say it is ambiguous (i.e. there could be more than one).

⭐⭐ Problem 7-3: Travel Troubles

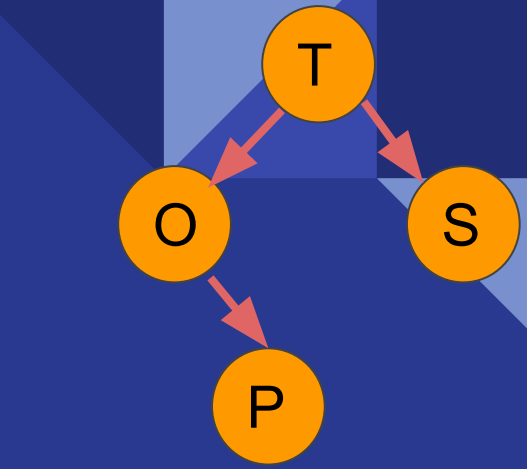
- You are in a magical country consisting of N towns ($2 \le N \le 1000$), some of which are connected by bidirectional toll roads (with at most one road directly connecting any two towns). But these are no ordinary toll roads -- each i-th road steals some of your current size, i.e., multiplies your current size by $1/K_i$.
- You are currently in town 1, and you want to get to town N. (It is guaranteed that there is at least one way to get there.)
- Given a list of the roads and their costs, what is the largest size (as a fraction of your original size) that you can have when you reach your destination?

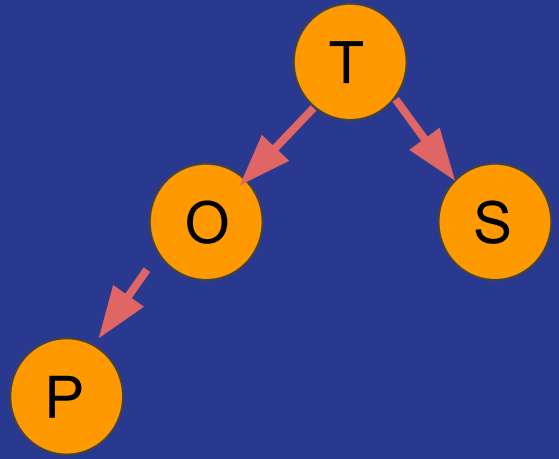
- **Warm-up:** $K_i = 2$ for all i.
- **Harder:** $2 \le K_i \le 1000$, for all i.
- **Even harder:** Now some roads multiply your size by K_i and some divide it by K_i. (In each case $2 \le K_i \le 1000$, and there is another parameter S_i for each road that is -1 if the road makes you shrink and 1 if the road makes you grow). The answer might be ∞, in which case you should report that (and loom and laugh ominously)

Solutions to 7-2 (discussed in-class)

Ambiguity!

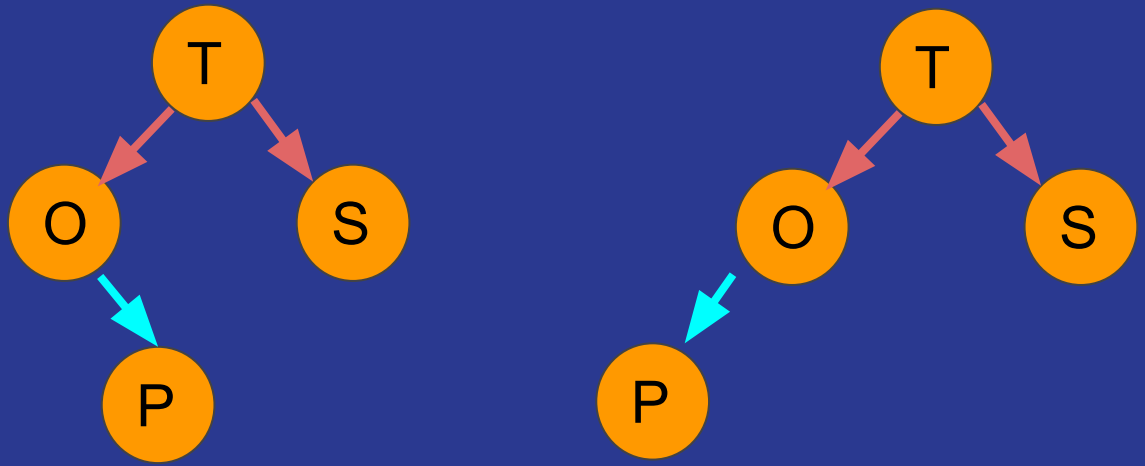
- Our example from earlier: we are given [T, O, P, S] as the preorder list and [P, O, S, T] as the postorder list.
- This case is ambiguous!
  - We could produce [O, P, T, S], which accords with the tree we showed earlier.
  - But [P, O, T, S] is also consistent with the given data – check out that other tree…





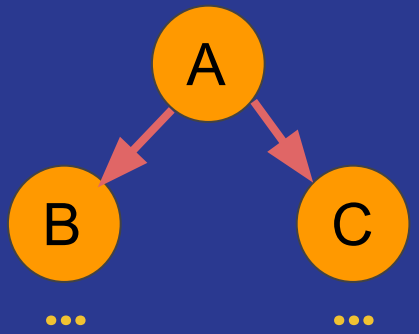
When is there ambiguity?

- Whenever a node has exactly one child. Then we could flip that child from being on the left to being on the right, or vice versa. This would not change either the preorder or postorder traversals, but it would change the inorder traversal.



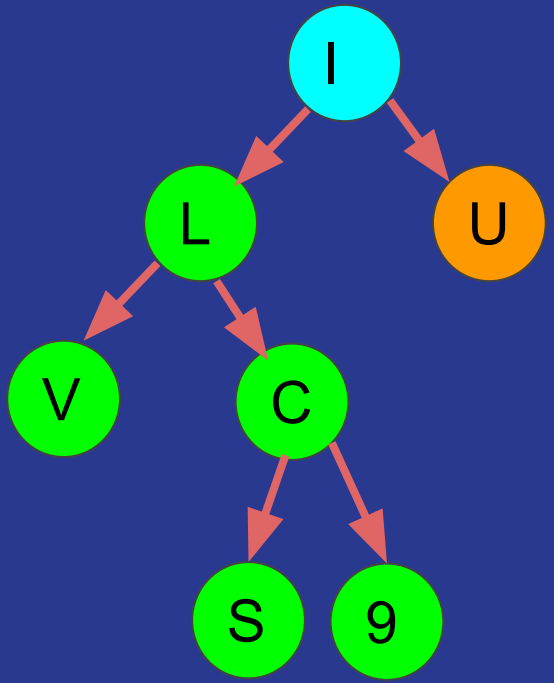
When is there *not* ambiguity?

- As long as each node has either zero or two children.
  - Zero: Pre, post, in for the node are all trivially the same.
  - Two: Pre will give A Pre(B) Pre(C), post will give Post(B) Post(C) A. It is clear that the B subtree comes before the C subtree, so we can return In(B) A In(C) as the sole inorder traversal.



But how do we know where the B subtree ends and the C subtree begins?

Subtree location

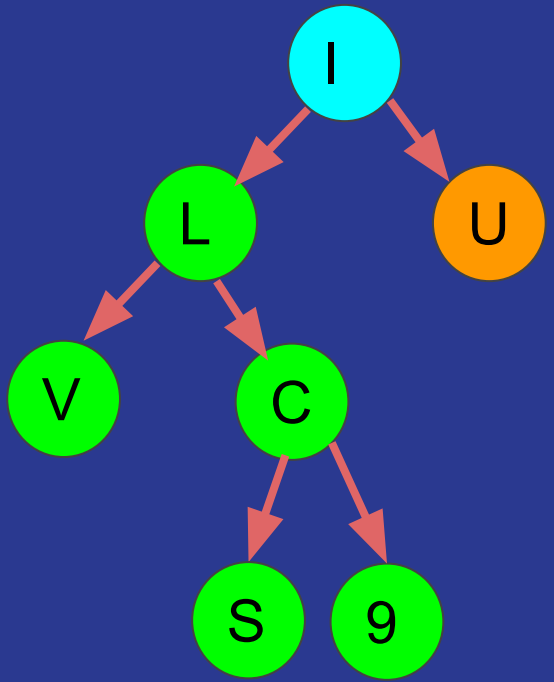


Pre: ILVCS9U

Post: VS9CLUI

- The root is the first element of Pre and the last element of Post.
- The first child is the second element of Pre.

Subtree location



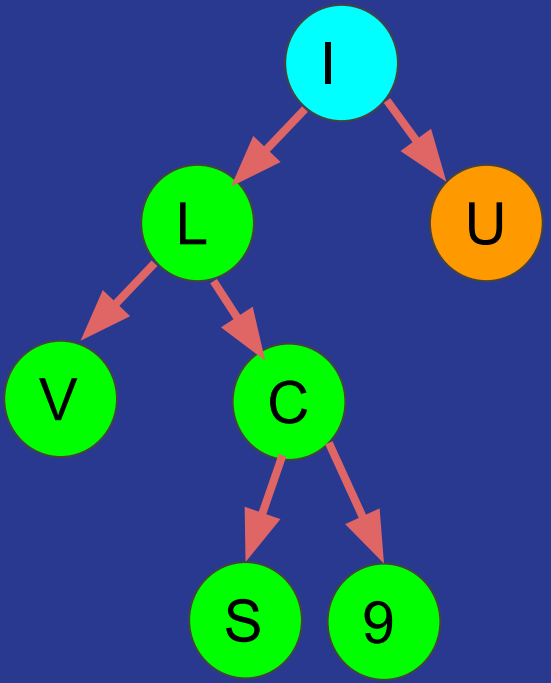
Pre: ILVCS9U

Post: VS9CLUI

- Walk through Post and find that first child.
- Then recurse on the subparts.

In(LVCS9, VS9CL) + I + In(U, U)

And so on…



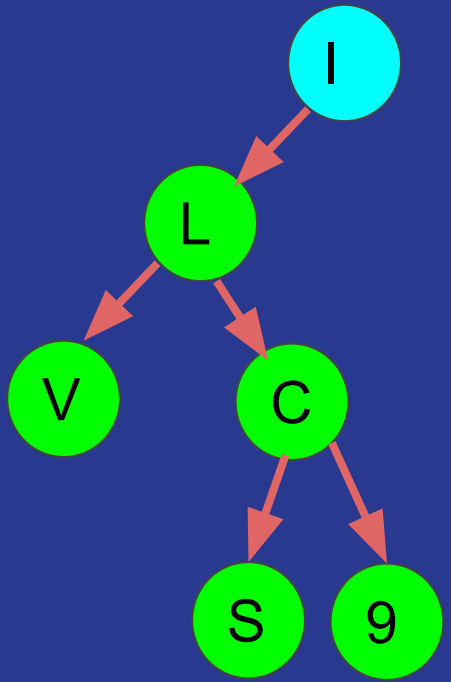
ln(**LVCS9**, **VS9CL**) + **I** + ln(**U**, **U**)

ln(**V**, **V**) + **L** + ln(**CS9**, **S9C**) + **I** + **U**

V + **L** + ln(**S**, **S**) + **C** + ln(**9**, **9**) + **I** + **U**

V + **L** + **S** + **C** + **9** + **I** + **U**

What if a child were missing?



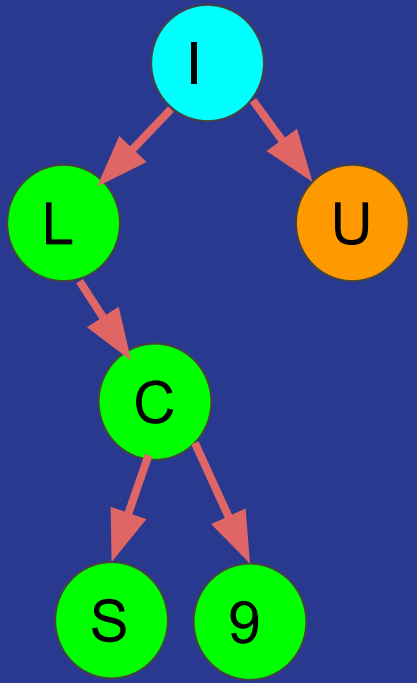
Pre: ILVCS9

Post: VS9CLI

In(LVCS9, VS9CL) + I + In(,)

No second child!
Return "AMBIGUOUS"

Works for missing left children



In(**LCS9**, **S9CL**) + **I** + In(**U**, **U**)

In(**CS9**, **S9C**) + **L** + In(,) + **I** + **U**

No second child!
Return "AMBIGUOUS"

Code

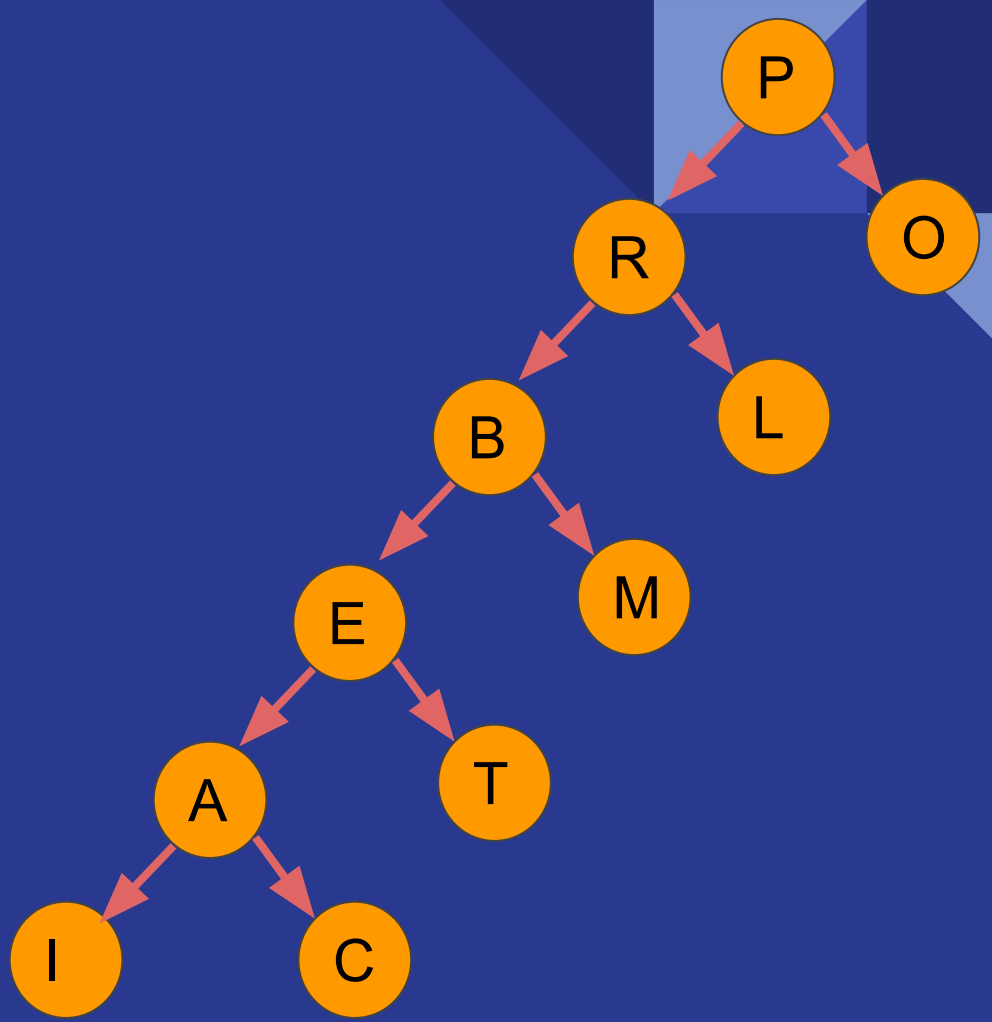
```
def solve(pre, post):
    # Base case: leaf node
    if len(pre) == 1:
        return [pre[0]]
    # Otherwise, there is at least one child.
    root = pre[0]
    first_child = pre[1]
    i = 0
    # Find the corresponding child in the postorder traversal.
    while post[i] != first_child:
        i += 1
    if i == len(post) - 2: # then there is no second child
        return "AMBIGUOUS"
    first_child_inorder = solve(pre[1:i+2], post[0:i+1])
    second_child_inorder = solve(pre[i+2:], post[i+1:-1])
    if first_child_inorder != "AMBIGUOUS" and second_child_inorder != "AMBIGUOUS":
        return first_child_inorder + [root] + second_child_inorder
    else:
        return "AMBIGUOUS"
```

"So what's the running time?"

This can be $O(n^2)$!

- We repeatedly search the postorder list and find that the left subtree has the biggest size possible (the right subtree has size 1)

- We could mitigate this by searching from *both* ends at once, using the fact that the next-to-last entry in the Post list is the right child.



How can we salvage this?

- The answer itself is O(*n*), and it seems like there "should" be a way to hit that complexity…
- We waste so much time looking up where elements are in the Post list! Sometimes over and over…
- What if we do one initial pass over the Post string and make a hash table that lets us look up the position in the string of each element?
- This cuts the work of each subproblem down to O(1), and indeed makes the entire process O(*n*).
- (We have to do some math to figure out pointer offsets for the subproblems, but this is all constant time.)

What if there are repeated elements?

Uh hey thanks for coming to CS9! We're out of time

(If you do want to think on this, and you come up with a cool solution, please post it on Ed)

Don't let Ian forget to say the password!!!!!!

Solutions to 7-3

Travel Troubles: Solution

- If all K_i values are 2, then all that matters is *how many* roads you take. Then this just becomes breadth-first search.
- In the second version, we'd love to use a shortest-paths algorithm like Dijkstra's Algorithm, but that only works for adding up costs, not multiplying them! If only we had some way of making multiplication into addition…
- We do! We can take the logarithm of every K_i. The log of a product is the sum of the logs of the individual values.
- So we do Dijkstra's on these log-costs to get a smallest possible answer, then re-exponentiate the final answer to get the total factor that we *shrink* by. (This works because log is strictly increasing, and these values are all positive.)

Travel Troubles: Solution

- In the third version, we can't pull the same trick with Dijkstra's because now there would be both negative and positive edge values after we take the logs of all edges. Except in certain edge situations (of which this is not one), Dijkstra's does not work correctly with negative edge values.

- However, the Bellman-Ford algorithm does, and it will also report an infinite cycle (i.e. if we can just go around and around a cycle with a net positive impact on our size, and then get arbitrarily large).