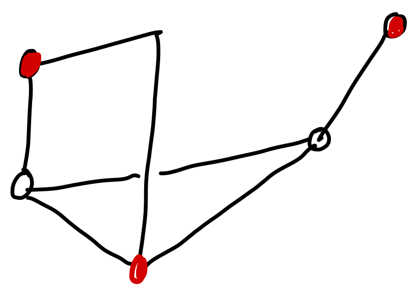
LECTURE 5 (SEPTEMBER 9th)

Dynamic Programming on Trees & DAGs

We will wrap up talking about dynamic programming today
We will see some examples where the memoization structure is not a table.
This is to reinforce the point that dynamic programming is not about filling tables but about smart recursion

Maximum Independent Set Problem

Given a graph G, find a set of vertices such that there is no edges between them and the set is as large as possible

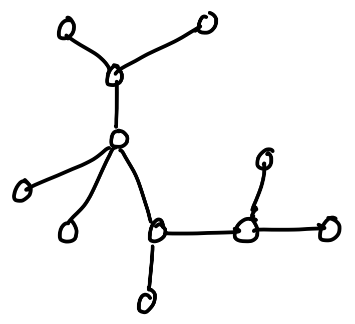


e.g. ● form an independent set

This is an NP-hard problem, so we are not going to be able to give an efficient algorithm for this problem.

But there are many kinds of graphs where this problem is easy, e.g. trees

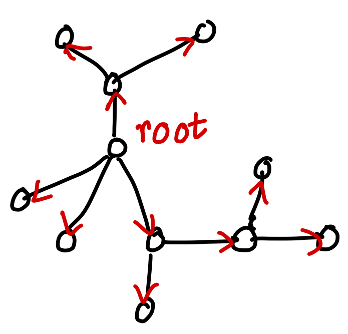
Trees are connected graphs that do not have any cycles



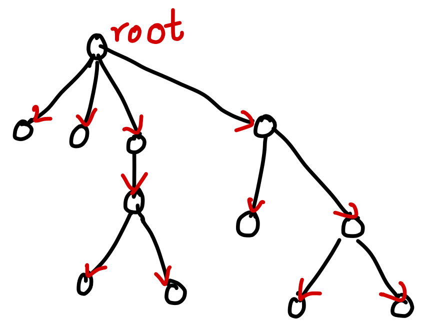
When we think about dynamic programming, we want to find a recursive structure in some problems.

In an array or a sequence, this is a bit straightforward, but what about trees? Here there is no obvious recursive structure and we are just trying to find a set.

We can impose a recursive structure on the tree by rooting it – choose a designated vertex as the root, and directing all of the edges going away from the root. Then, we can draw the tree from top to bottom, where the top is the root.



can be drawn as



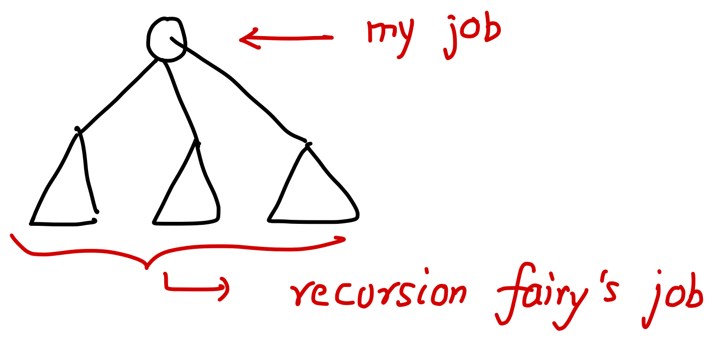
Every node has pointers to it's children now and every node except the root has a parent.

A rooted tree has a recursive structure we can use, i.e.,

$$\text{rooted tree} = \text{node} + \underbrace{\text{set of rooted trees}}_{\text{could be empty}}$$

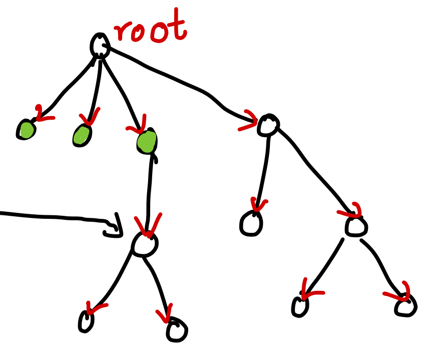
To design a recursive algorithm, we need to make some sort of decision about the root, ask the recursion fairy to handle the subtrees, and assemble the answer based on what the recursion fairy tells me.

So, intuitively, a recursive subproblem here is a subtree which we can identify with its root, i.e., we can say that we look at this subtree rooted at this particular node



For the maximum independent set, the question about the root we want to answer is whether it is in the maximum independent set or not. But since we do not know this, we will just do what we have been doing previously — assume it is and solve it recursively and assume it is not and solve recursively and then see which solution is bigger

Assume that the root is not in the independent set and we have included some children of the root in the independent set shown as ●



Suppose we recurse here, then we cannot include it in the independent set as we included the parent

So, there is some information we need to remember from past decisions in the recursive calls

How should we describe the subproblem that we are solving?

We need an additional bit, let's define

LIS(v,p) = size of largest independent set in a subtree rooted at v where v must be excluded from the independent set if $p=$ true

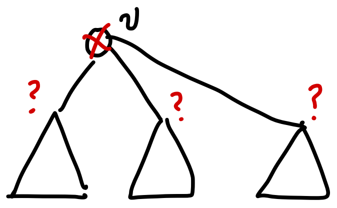
What's the top level of the recursion?

$$\text{Is it } \underbrace{\text{LIS(root, FALSE)}} \text{ or LIS(root, TRUE) ?}$$

↳ It's this one! FALSE just says that root is not excluded but it may or may not be in the independent set

The easiest way to think about this is in terms of two separate functions LIS(root, FALSE) and LIS(root, TRUE)

The picture we have is

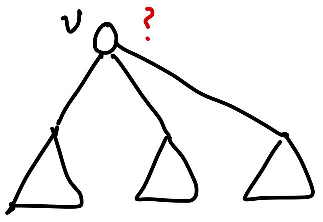


and we want to compute LIS $(v$, TRUE$)$ i.e., v is excluded

In this case, the children of v are not excluded

$$\text{So, LIS}(v, \text{TRUE}) = \sum_{w \text{ child of } v} \text{LIS}(w, \text{FALSE})$$

Similarly in the second case we know that v is not excluded but don't know if it is included We try both possibilities – if v is included or not and take the maximum



$$\text{LIS}(v, \text{FALSE}) = \max\left\{ \underbrace{\sum_{w \text{ child of } v} \text{LIS}(w,\text{FALSE})}_{\substack{v \text{ not included} \\ \text{in ind. set}}},\ \underbrace{1+\sum_{w \text{ child of } v} \text{LIS}(w,T)}_{\substack{v \text{ included in ind. set} \\ w \text{ not included}}} \right\}$$

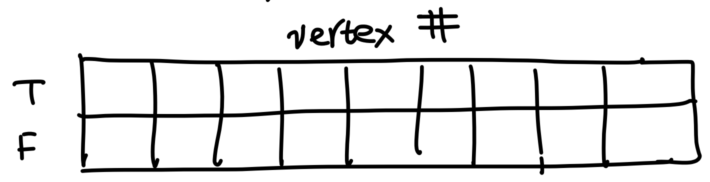
What is the base case? When the tree is a leaf, the expression above still make sense since if there are no children, the summation over children is zero and we get the right thing.

Now, we have a recurrence but we still want to avoid doing repeated work, so we want to memoize this function somehow

What data structure should we use ?

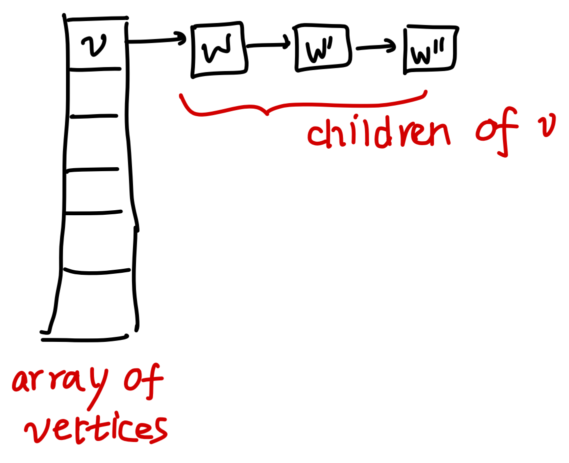
We need a data structure that takes as input a node in the tree and returns an integer.

We can use a 2D array for instance



but depending on how the tree is represented, it may not be easy to write in this array. For example, if each node in the tree only has pointers to each children but not a node id. So, what do we do ?

If the tree is represented in an adjacency list



Then, we can assume that the node id is the index in the array to the left

But not all trees are represented as adjacency lists, sometimes we only have pointers to the children. What should we do ?

One possibility is to do a pre-order traversal and store the vertex record in the hash table

Another possibility is to assume that we have the capability to add new information to the data structure itself. For example, we can add new fields in the record for each node "v. forbidden" and "v. allowed" that store the values we need. So, the memoization data structure is the tree itself. If we don't have this ability, we can create a new tree data structure to store the original tree where we can add new fields as well.

The last thing we need is the evaluation order! Let's think about the running time to figure this out

$$\text{LIS}(v, \text{TRUE}) = \sum_{w \text{ child of } v} \text{LIS}(w, \text{FALSE})$$

$$\text{LIS}(v, \text{FALSE}) = \max \left\{ \sum_{\substack{w \text{ child} \\ \text{of } v}} \text{LIS}(w, \text{FALSE}) , \; 1 + \sum_{\substack{w \text{ child} \\ \text{of } v}} \text{LIS}(w, T) \right\}$$

$\sum_{w} = 0$ if v is a leaf

What is the running time? Now each of the recursive calls look up in $O(1)$ time the values they need in the data structure. We only need to look at LIS (w, TRUE) once when we are computing LIS (parent(w), FALSE). Similarly, we only need to look up LIS (w, FALSE) twice once while computing LIS (parent(w), TRUE) and LIS (parent(w), FALSE). Each of these three recursive calls happen once for each node, so the total number of times we need to read from the memoization structure is three times the number of nodes.

So, we

read	from the memoization structure	$\leq$ 3 times per node
write	to the memoization structure	$\leq$ 2 times per node

For each node v, to compute the sum in the recurrence, we will have a for loop over the children of v and in each for loop we are going to read/write from the memoization structure.

The total amount of time is going to be $O(1)$ per node which absorbs the overhead of read/write/for loops/max, etc.

So, overall running time $O(n)$ provided that we evaluate things in the correct order. Again there are several options

- Reverse level order
- postorder $\rightarrow$ recommended because this naturally corresponds to the natural recursion on the tree. Also, because post-order is the natural outcome of depth-first search

So, the overall algorithm looks like

```
At each node $v$ in post-order
    compute LIS$(v, *)$ from LIS $(w, *)$ for children $w$
```

The recursive algorithm does the depth-first search already, so the only thing we need to do before we return from the recursive call for v is to write LIS(v, TRUE) and LIS(v, FALSE) in our data structure. Then, the parent call and retrieve those values and we will not need them afterwards.

So, now you might realize that we don't need to write these two values down since we are not going to need them except for the parent. So we can just return the two values to the parent recursive call. So, what we can do for dynamic programming over trees where the evaluation order is post-order is the following equivalent thing

LIS(v) returns both values LIS(v, FALSE) & LIS(v, TRUE)

and the parent call can combine these values to compute its return values

So, we have three different implementations here

$\boxed{1}$ Dynamic programming with a table via post-order

$\boxed{2}$ Memoized recursion

$\boxed{3}$ Recursion returns multiple values

Running time is $O(n)$ for all of them, so you can use any of them

This is different than array based DP algorithms we saw in previous lecture because there filling the tables correctly simplifies the algorithm and also makes it faster, but when you are dealing with trees or DAGs we can't really avoid the recursion, so either of the above three options will roughly be equivalent.

Let's emphasize one aspect of this algorithm - that the thing that ties all the implementations together is depth-first search.

But depth-first search works for other graphs as well

Memoized recursion vs depth-first search — how do they compare?

As long as the pattern of dependencies between the recursive calls is a directed acyclic graph, memoized recursion & depth-first search are the same.

Let's say we pass in some input x to a recursive subroutine called memoize and we want to record the value in an array value[x]. We will invoke the recursive subproblems y and based on the value that we get from those calls, we will compute value[x] and then return the value.

```
MEMOIZE($x$):
  if $value[x]$ is undefined
      initialize $value[x]$

      for all subproblems $y$ of $x$
          MEMOIZE($y$)
          update $value[x]$ based on $value[y]$
      finalize $value[x]$
```

Here is how depth-first search works on a general dag

- Initially all vertices are unmarked, and the first time we visit a vertex we mark it. We do some computation at that node that we call previsit and then recursively do DFS on each successor of v, followed by some postprocessing postvisit(x). We mark the vertices so that we don't do the repeated work if we reach the same vertex again.

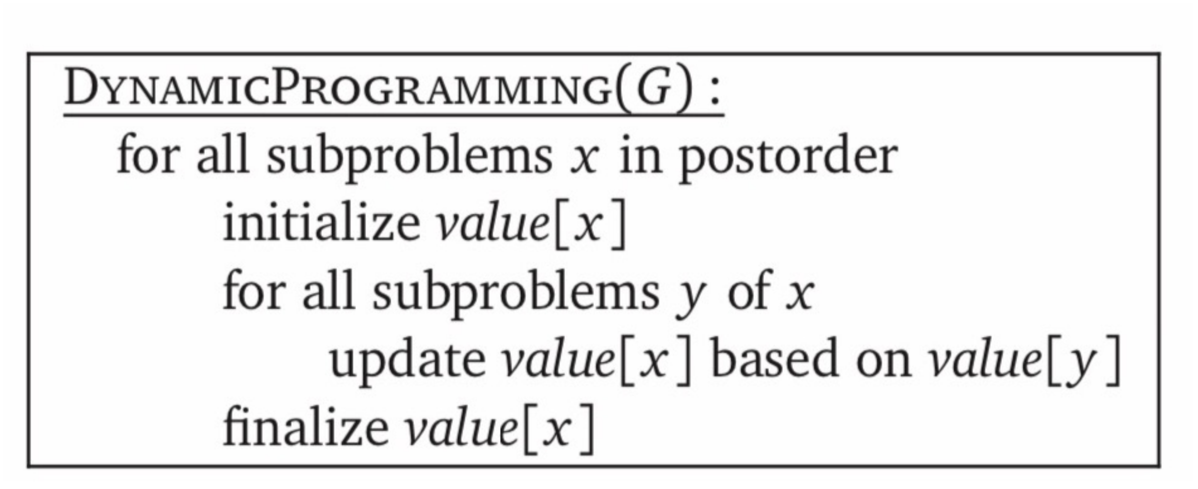
```
DFS($v$):
  if $v$ is unmarked
      mark $v$
      PREVISIT($x$)
      for all edges $v \to w$
          DFS($w$)

      POSTVISIT($x$)
```

Consider a DAG where the nodes are the recursive subproblems in memoization, then DFS on this DAG is equivalent to memoization

DFS still works even if graph is not a DAG, but it's not equivalent to memoization if graph is not a DAG

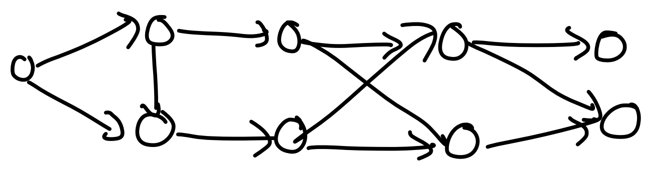
Similarly if we look at dynamic programming



```
DYNAMICPROGRAMMING($G$):
    for all subproblems $x$ in postorder
        initialize $value[x]$
        for all subproblems $y$ of $x$
            update $value[x]$ based on $value[y]$
        finalize $value[x]$
```

it again ends up being a post-order traversal of the graph where we update the value in that order. But post-order is defined in terms of DFS. Post-order is the same as reverse topological sort order.

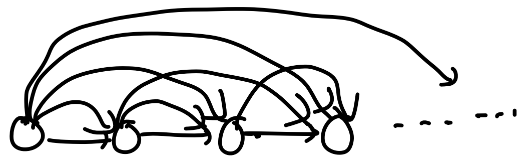
For example if the DAG is



then the order in DP is right to left.

For example, consider the string splitting problem as a DAG.

G has a node for each prefix/index i
It has an edge $i \to j$ where $i<j$
and the graph looks like



Each of the edges here represent a recursive call in the string splitting problem

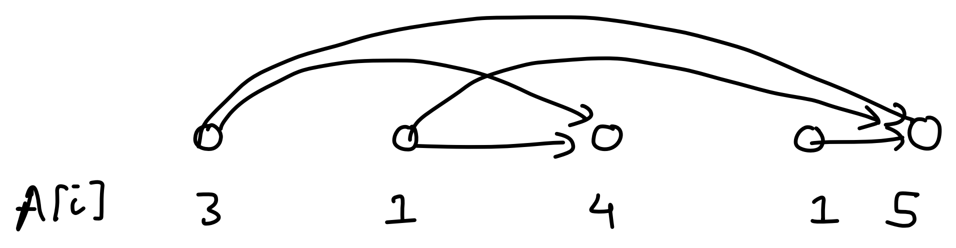
This is just a transitive tournament we saw in HW0.

So, when we are memoizing this into an array, we can identify each node with its position in the topological order and update the array in reverse topological order

To give another example, Let's take the longest increasing subsequence problem. One of the ways we formulated it was

$$LIS(i) = \text{length of LIS of } A[i \ldots n] \text{ including } A[i]$$

Again the dependency graph will have node for every i but the edges are only those indices $j>i$ where $A[j]>A[i]$ since those are the recursive subproblems we consider



Any path in this graph is an increasing subsequence

So, another way of solving LIS is to find the longest path in this DAG which can also be solved with DFS

The punchline here is that sequence dynamic programming is finding an optimal path in a DAG

If it's easier for you to think of this way, feel free — either one is fine

More generally if we want to compute the longest path in a DAG from a source node to another node t, here is a dynamic programming formulation

$$LLP(v) = \begin{cases} 0 & \text{if } v = t, \\ \max\left\{\ell(v\to w) + LLP(w) \,\middle|\, v\to w \in E\right\} & \text{otherwise}, \\ -\infty & \text{if } v \text{ is a sink and } v \neq t \end{cases}$$

If the underlying graph is not a DAG, we will get stuck in a loop, so this doesn't work if the graph is not a DAG which is NP-hard anyway

Here is a pseudocode description of the algorithm:

```
LONGESTPATH($v, t$):
  if $v = t$
      return 0
  if $v.LLP$ is undefined
      $v.LLP \leftarrow -\infty$
      for each edge $v\to w$
          $v.LLP \leftarrow \max\{v.LLP,\ \ell(v\to w) + \text{LONGESTPATH}(w,t)\}$
      return $v.LLP$
```

You can just invoke this subroutine as a black-box for HW/exams

Running Time $= O(|V| + |E|)$