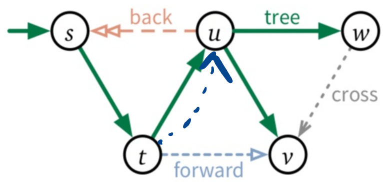
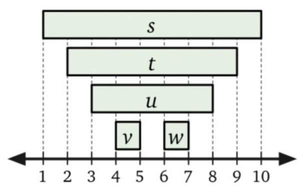
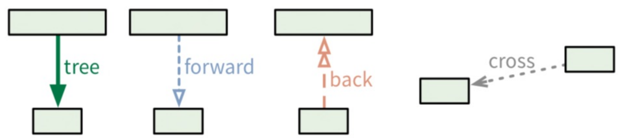
```
DFSAll(G):
    clock ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            clock ← DFS(v, clock)
```

```
DFS(v, clock):
    mark v
    clock ← clock + 1;  v.pre ← clock
    for each edge v→w
        if w is unmarked
            w.parent ← v
            clock ← DFS(w, clock)
    clock ← clock + 1;  v.post ← clock
    return clock
```







forward edge $u \to v$

$u.pre < v.pre < v.post < u.post$ &

it isn't a tree edge

cross edge $u \to v$

$v.pre < v.post < u.pre < u.post$

back edge $u \to v$

$v.pre < u.pre < u.post < v.post$

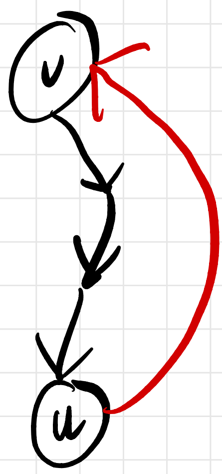
$\Rightarrow$ $u \to v$ is a back edge iff $u.post < v.post$

Detecting Cycles

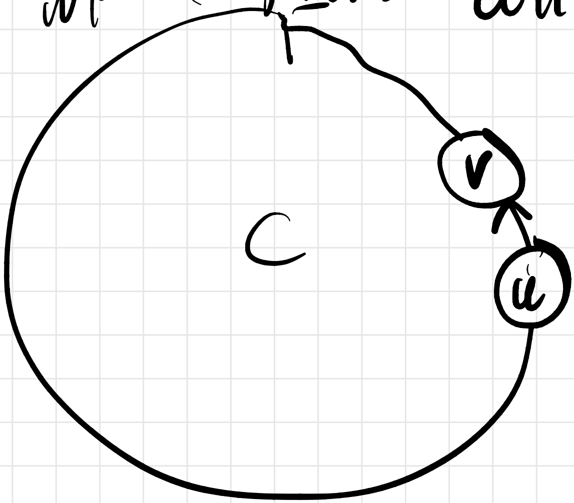
Lemma: Directed graph G has a cycle iff DFSAll(G) yields at least one back edge.

Proof: Suppose $u \to v$ is a back edge. There is a path P of tree edges from v to u

$P \circ u \to v$ is a cycle.



Suppose G has a cycle C. Let v be the first vertex of C we explore during DFSAll



u: immediately before v on C

During call DFS(v), we will reach every unmarked vertex reachable from v.

We call DFS(u).

$u \to v$ is a back edge

Cycle Detection Algo:

Run DFSAll(G) to find a postorder.

Return true iff $\exists$ an edge $u \to v$ s.t. $u.post < v.post$

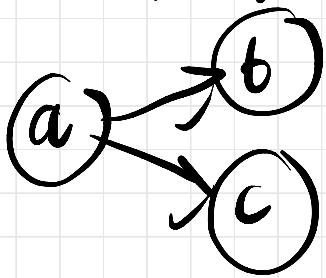
$O(V+E)$ time

Topological Sort

Given directed graph $G=(V,E)$ a <u>topological ordering</u> of its vertices is a total ordering of the vertices where $u < v$ <u>if</u> $u \to v \in E$.

i.e.

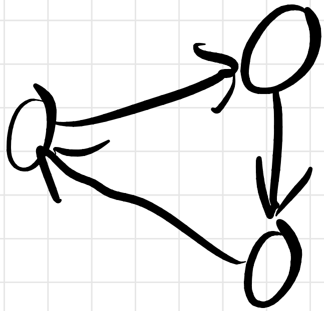
you can draw graph so edges only go left to right.



a b c

-or-

a c b



Cycle => no topological ordering.

Directed acyclic graph (DAG) if directed & no cycles.

Every DAG has a top. order!

Proof: Run DFSAll.

No back edges.

=> for all edges $u \to v$,

$v.post < u.post$.

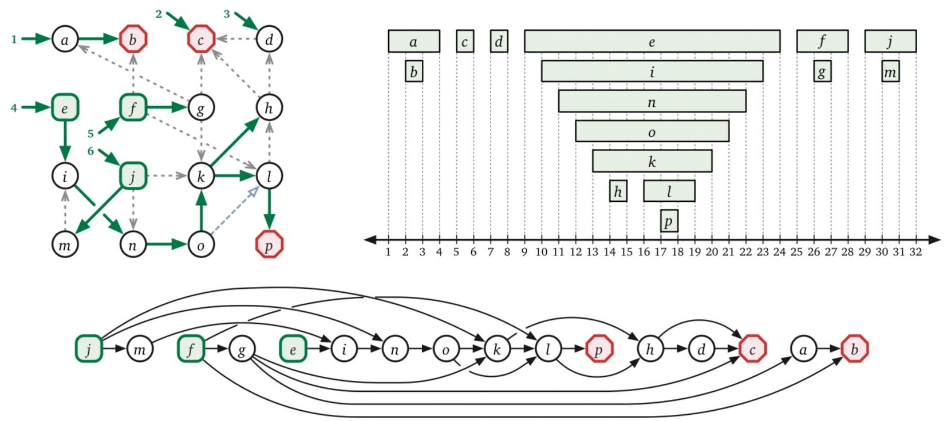
Consider the <u>reversed</u> postorder. $u < v$ for each edge $u \to v$, so it is a top. order.

Alg: Run DFSAll.
Return reversed post order.

$O(V+E)$ time

```
TopologicalSort(G):
    for all vertices v
        v.status ← New
    clock ← V
    for all vertices v
        if v.status = New
            clock ← TopSortDFS(v, clock)
    return S[1..V]
```

```
TopSortDFS(v, clock):
    v.status ← Active
    for each edge v→w
        if w.status = New
            clock ← TopSortDFS(w, clock)
        else if w.status = Active
            fail gracefully
    v.status ← Finished
    S[clock] ← v
    clock ← clock − 1
    return clock
```

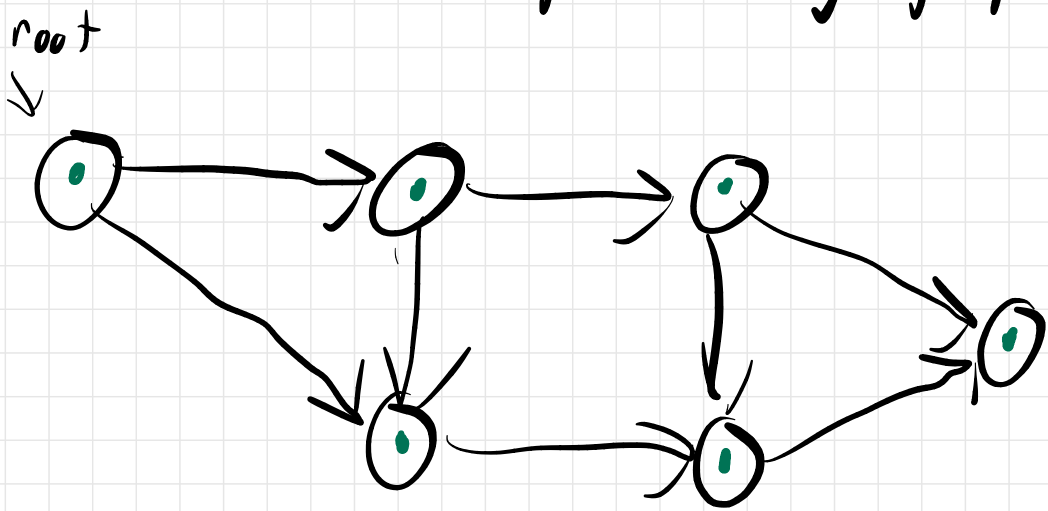

1 2 3 4 5 6
a b c d e f g h i j k l m n o p
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
j m f g e i n o k l p h d c a b

Dynamic Programming

Suppose we have a recurrence.
The <u>dependency graph</u>
has the subproblems
as its vertices + edges
$x \to y$ for each direct
call for a subproblem
y from a subproblem x.

Cycle $\Rightarrow$ infinite loop of
recursive calls.

So good recurrences have DAGs as dependency graphs.



If we do recursion with basic memoization, we're doing a depth-first search.

So the subproblems are solved in <u>post</u> order.

The dynamic programming algorithms are really solving each subproblem in postorder.

(usually we just say what the postorder is)

Longest Path problem

Given: A DAG $G=(V,E)$ with edge weights $\ell: E \to \mathbb{R}$.

Want length of longest path from given s to given t.

$LLP(v)$: length of longest path from v to t or $-\infty$ if no v-t path exists.

If $v = t$, $LLP(v) = 0$.

o.w.

$$LLP(v) = \begin{cases} 0 & \text{if } v = t \\ \max\limits_{v \to w \in E} \{\ell(v \to w) + LLP(w)\} & \text{o.w.} \end{cases}$$

(max over nothing $= -\infty$)

G is the dependency graph for LLP.

```
LongestPath(s, t):
  for each node v in postorder
    if v = t
      v.LLP ← 0
    else
      v.LLP ← −∞
      for each edge v→w
        v.LLP ← max{v.LLP, ℓ(v→w) + w.LLP}
  return s.LLP
```

$O(V+E)$ Time

Shortest Path in DAG.

max $\Rightarrow$ min

$-\infty \Rightarrow +\infty$

$O(V+E)$ time