Relevant Reading: Chapter 7, esp. 7.32 onward.

**Review:** Graphs, DAGs (Directed Acyclic Graphs).

Any graph: \( G = (V, E) \) where \( V \): set of vertices,
\( E \subseteq V \times V \): set of edges.

Want: Given a DAG, for all \( v \in V \), store in \( \text{lp}(v) \) the length of the longest path starting at \( v \).

**Recurrence equation:**
\[
\text{lp}(v) = 1 + \max_{(v, w) \in E} \text{lp}(w)
\]

Easy from here:

Driver \( (G = (V, E)) \):
1. global `visited` \[ \text{init to } F \]
2. for each \( v \) in \( V \) do
   - if `visited` \[ v \] = \[ F \] then \( \text{lp}(v) \).
3. end for.
LP(x) {
    Visited[x] ← T
    v. lpath ← 0
    for each out-neighbor w of x do // ie. ∀ w s.t. (v, w) ∈ E.
        if Visited[w] = F then
            LP(w)
        end if
        if v. lpath < w. lpath + 1 then
            v. lpath ← w. lpath + 1.
        end if
    end for
}

Problem: Suppose edges in DAG have some cost given by Cost(v, w) ≠ (v, w) ∈ E. Cost of path is sum of cost of edges. Want: ∀ v ∈ V, store in v.maxpath the maximum-cost path starting at v.

Recurrence Equation:

v.maxpath = max w.maxpath + Cost(v, w)

Code is same as before except in LP(x) lines 7 & 8:

if v. lpath < w. lpath + Cost(v, w) then
    v. lpath ← w. lpath + Cost(v, w).
end if
Problem: Test if an undirected graph is a tree. If not, report a cycle.

Starting point: Test if an undirected graph is a tree.

Driver(s) \{
  global hascycle = F
  global visited[1:n] init to F
  global parent[1:n] init to 0
  for each v in V do
    if visited[v] = F and hascycle = F then
      TestCycle(v)
    end if
  end for
  return hascycle.
\}

TestCycle(v) \{
  visited[v] = T
  for each neighbor w of v do
    if w \neq Parent[v] then
      if visited[w] = F then
        Parent[w] \leftarrow v
        TestCycle(w)
      else
        hascycle \leftarrow T
      end if
    end if
  end for
\}
Why this works:

Graph has a cycle $\iff$ we revisit a node.

Need to keep track of parent because when we recurse on $w$, $(w,v) \in E$ also and $\text{visited}[v] = T$ already. Don't want to report a cycle in this case.

Eg:

```
&  v
\& w
\& x
```

$\text{TestCycle}(v)$:

```
L = \text{TestCycle}(w).
L = \text{visited}(v) = \text{true}
\implies \text{sets hascycle} = T.
```

Don't want this!

Runtime: visit each node and each edge once:

$\Rightarrow O(\mathcal{V} + \mathcal{E})$

How to find cycle? $\text{Parent}[1:n]$ remembers order of visits. If we revisit $v$, we have stored a path from $i$ to $v$. (backtrack using $\text{Parent}[1:n]$).

Code is same except for direction doesn't matter.

Line 8: $\text{visited}[v] = T$.

```
\begin{cases}
\text{print } w, \text{ print } v \\
\text{line 8.} \\
\text{if } \text{visited}[v] = T \\
\text{print } u \\
\text{replaces} \\
\text{u} \leftarrow \text{Parent}[v] \\
\text{while } u \neq w \\
\text{print } u \\
\text{u} \leftarrow \text{Parent}[u]
\end{cases}
```
E.g. Alg visits 1, 2, 3, 4, 5, 3 & finds cycle. Parent = [0, 1, 2, 3, 4] v = 5 w = 3.

Alg prints 3, 5, 4 \[ w \text{ } v \text{ } \text{Parent}[v] \] stops because Parent[4] = w.

Problem Suppose DFS is run on following DAG, but without remembering visited nodes using visited[].
Determine # of recursive calls (including initial call).

\[ V = \{ \text{v}_0, \ldots, \text{v}_n, \text{w}_1, \ldots, \text{w}_n, \text{x}_1, \ldots, \text{x}_n \} \]
\[ |V| = 3n + 1. \]
\[ E = \{ (\text{v}_{i-1}, \text{w}_i), (\text{v}_{i-1}, \text{x}_i), (\text{w}_i, \text{v}_i), (\text{x}_i, \text{v}_i) \mid i = 1, \ldots, n \} \]

E.g. \( n = 2 \).
\( n = 1. \)
\( n = 0. \)

\[ R(1) = 5 = 3 + 2R(0). \]
\[ R(0) = 1. \]
\[ R(0) = 1. \]
\[ R(n) = 3 + 2R(n-1). \]

"Remove \( x_0, w_1, x_1 \), remaining is \((3(n-1)+1)\)-node graph with same structure: SUBPROBLEM \( w \)/" starting/root vertex \( v_1 \).

\[ \Rightarrow 3 \text{ calls } \begin{bmatrix} \text{for } x_0, w_1, x_1 \end{bmatrix} \]

+ 2 calls to SUBPROBLEM of size \( n-1 \).