Let v be the vertex in C with the earliest starting time. If v has a parent, then *parent*(v) starts before v and thus cannot be in C.

Now let w be another vertex in C. Just before DFS(v) is called, every vertex in C is new, so there is a path of new vertices from v to w. Lemma 1 now implies that w is a descendant of v in the depth-first forest. Every vertex on the path of tree edges v to w lies in C; in particular, *parent*$(w) \in C$. □

The previous lemma implies that each strong component of a directed graph G defines a connected subtree of any depth-first forest of G. In particular, for any strong component C, the vertex in C with the earliest starting time is the lowest common ancestor of all vertices in C; we call this vertex the ***root*** of C.

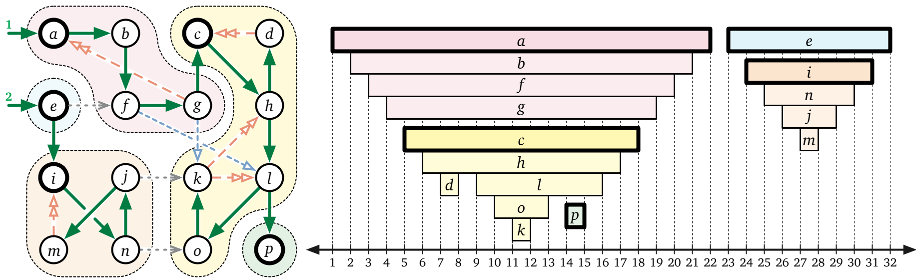

Figure 6.13. Strong components are contiguous in the depth-first forest.

I'll present two algorithms, both of which follow the same intuitive outline. Let C be any strong component of G that is a sink in *scc*(G); we call C a ***sink component***. Equivalently, C is a sink component if the reach of any vertex in C is precisely C. We can find all the strong components in G by repeatedly finding a vertex v in some sink component (somehow), finding the vertices reachable from v, and removing that sink component from the input graph, until no vertices remain. This isn't quote an algorithm yet, because it's not clear how to find a vertex in a sink component!

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      C ← ∅
      count ← count + 1
      v ← any vertex in a sink component of G      ⟨⟨Magic!⟩⟩
      for all vertices w in reach(v)
          w.label ← count
          add w to C
      remove C and its incoming edges from G
```

Figure 6.14. Almost an algorithm to compute strong components.

Koraraju and Sharir's Algorithm

At first glance, finding a vertex in a sink component *quickly* seems quite difficult. However, it's actually quite easy to find a vertex in a *source* component—a strong component of G that corresponds to a *source* in *scc*(G)—using depth-first search.

Lemma 6.3. *The last vertex in any postordering of G lies in a source component of G.*

Proof: Fix a depth-first traversal of G, and let v be the last vertex in the resulting postordering. Then DFS(v) must be the last direct call to DFS made by the wrapper algorithm DFSAll. Moreover, v is the root of one of the trees in the depth-first forest, so any node x with $x.post > v.pre$ is a descendant of v. Finally, v is the root of its strong component C.

For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Then x can reach y, and y can reach v, so x can reach v. Because v is the root of C, vertex y is a descendant of v, and thus $v.pre < y.pre$. The edge $x \to y$ guarantees that $y.pre < x.post$ and therefore $v.pre < x.post$. It follows that x is a descendant of v. But then v can reach x (through tree edges), contradicting our assumption that $x \notin C$. □

It is easy to check (hint, hint) that $rev(scc(G)) = scc(rev(G))$ for any directed graph G. Thus, the *last* vertex in a postordering of $rev(G)$ lies in a *sink* component of the original graph G. Thus, if we traverse the graph a second time, where the wrapper function follows a reverse postordering of $rev(G)$, then each call to DFS visits exactly one strong component of G.[4]

Putting everything together, we obtain the algorithm shown in Figure 6.15, which counts and labels the strong components of any directed graph in $O(V+E)$ time. This algorithm was discovered (but never published) by Rao Kosaraju in 1978, and later independently rediscovered by Micha Sharir in 1981.[5] The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of $rev(G)$, pushing each vertex onto a stack when it is finished. In the second phase, we perform a *whatever*-first traversal of the original graph G, considering vertices in the order they appear on the stack. The algorithm labels each vertex with the root of its strong component (with respect to the second depth-first traversal).

Figure 6.16 shows the Koraraju-Sharir algorithm running on our example graph. With only minor modifications to the algorithm, we can also compute the strong component graph $scc(G)$ in $O(V + E)$ time.

[4]Again: A reverse postordering of $rev(G)$ is not the same as a postordering of G.

[5]There are rumors that the same algorithm appears in the Russian literature even before Kosaraju, but I haven't found a reliable source for that rumor yet.

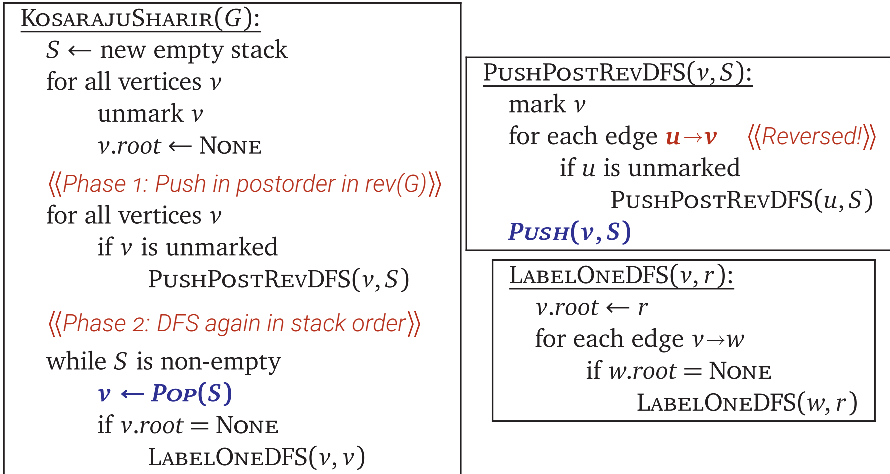

Figure 6.15. The Kosaraju-Sharir strong components algorithm

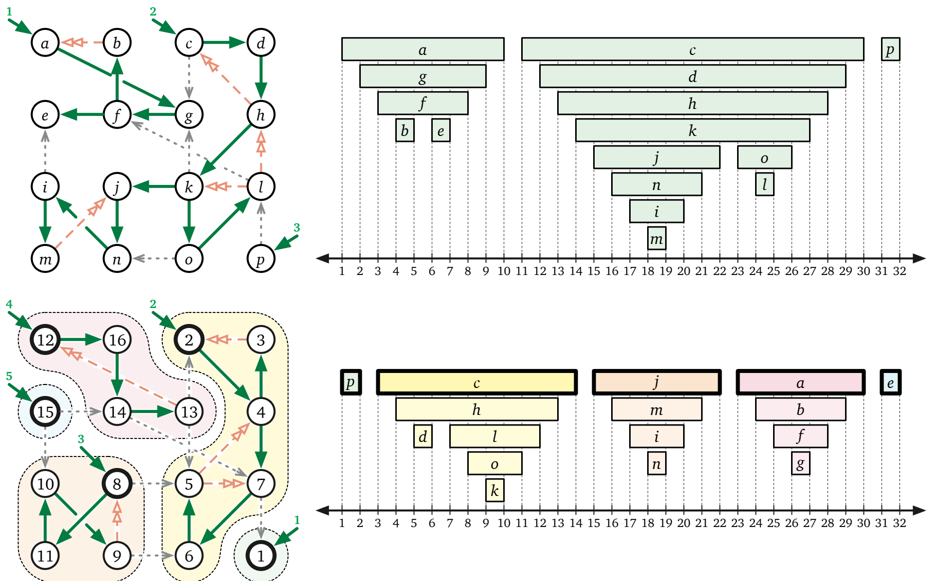

Figure 6.16. The Kosaraju-Sharir algorithm in action. Top: Depth-first traversal of the reversed graph. Bottom: Depth-first traversal of the original graph, visiting root vertices in reversed postorder from the first traversal.

♥Tarjan's Algorithm

An earlier but considerably more subtle linear-time algorithm to compute strong components was published by Bob Tarjan in 1972.[6] Intuitively, Tarjan's algorithm identifies a *source* component of G, "deletes" it, and then "recursively" finds the remaining strong components; however, the entire computation happens during a single depth-first search.

Fix an arbitrary depth-first traversal of some directed graph G. For each vertex v, let $\boldsymbol{low(v)}$ denote the smallest *starting time* among all vertices reachable from v by a path of tree edges followed by *at most one* non-tree edge. Trivially, $low(v) \le v.pre$, because v can reach itself through zero tree edges followed by zero non-tree edges. Tarjan observed that sink components can be characterized in terms of this *low* function.

Lemma 6.4. *A vertex v is the root of a sink component of G if and only if $low(v) = v.pre$ and $low(w) < w.pre$ for every proper descendant w of v.*

Proof: First, let v be a vertex such that $low(v) = v.pre$. Then there is no edge $w{\rightarrow}x$ where w is a descendant of v and $x.pre < v.pre$. On the other hand, v cannot reach any vertex y such that $y.pre > v.post$. It follows that v can reach only its descendants, and therefore any descendant of v can reach only descendants of v. In particular, v cannot reach its parent (if it has one), so v is the root of its strong component.

Now suppose in addition that $low(w) < w.pre$ for every descendant w of v. Then each descendant w can reach another vertex x (which must be another descendant of v) such that $x.pre < w.pre$. Thus, by induction, every descendant of v can reach v. It follows that the descendants of v comprise the strong component C whose root is v. Moreover, C must be a sink component, because v cannot reach any vertex outside of C.

On the other hand, suppose v is the root of a sink component C. Then v can reach another vertex w if and only if $w \in C$. But v can reach all of its descendants, and every vertex in C is a descendant of v, so v's descendants comprise C. If $low(w) = w.pre$ for any other node $w \in C$, then w would be another root of C, which is impossible. □

Computing $low(v)$ for every vertex v via depth-first search is straightforward; see Figure 6.17.

Lemma 6.4 implies that after running FindLow, we can identify the root of *every* sink component in $O(V + E)$ time (by a global whatever-first search),

[6]According to legend, Kosaraju apparently discovered his algorithm *during* an algorithms lecture; he was supposed to present Tarjan's algorithm, but he forgot his notes, so he had to make up something else on the fly. The only thing I find surprising about this story is that nobody tells it about Sharir or Tarjan.

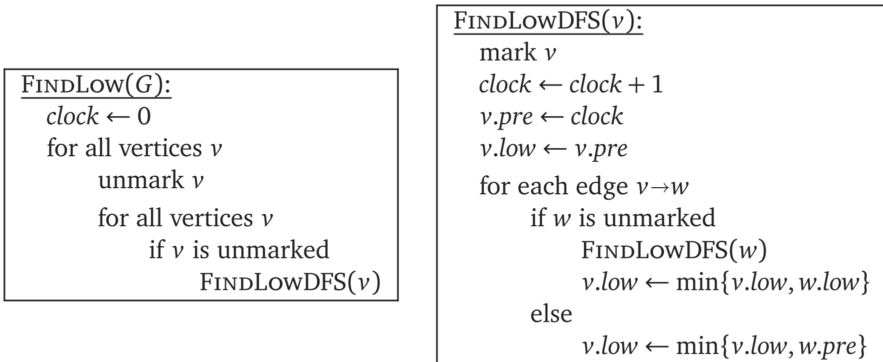

Figure 6.17. Computing *low*(v) for every vertex v.

and then mark and delete those sink components in $O(V + E)$ additional time (by calling whatever-first search at each root), and then recurse. Unfortunately, the resulting algorithm might require V iterations, each removing only a single vertex, naively giving us a total running time of $O(VE)$.

To speed up this strategy, Tarjan's algorithm maintains a stack of vertices (separate from the recursion stack). Whenever we start a new vertex v, we push it onto the stack. Whenever we finish a vertex v, we compare $v.low$ with $v.pre$. Then the *first* time we discover that $v.low = v.pre$, we know three things:

- Vertex v is the root of a sink component C.
- All vertices in C appear consecutively at the top of the stack.
- The *deepest* vertex in C on the stack is v.

At this point, we can identify the vertices in C by popping them off the stack one by one, stopping when we pop v.

We could delete the vertices in C and recursively compute the strong components of the remaining graph, but that would be wasteful, because we would repeat *verbatim* all computation done before visiting v. Instead, we *label* each vertex in C, identifying v as the root of its strong component, and then ignore labeled vertices for the rest of the depth-first search. Formally, this modification changes the definition of *low*(v) to the smallest starting time among all vertices ***in the same strong component as v*** that v can reach by a path of tree edges followed by at most one non-tree edge. But to prove correctness, it's easier to observe that ignoring labeled vertices leads the algorithm to exactly the same behavior as actually deleting them.

Finally, Tarjan's algorithm is shown in Figure 6.18, with the modifications from FINDLOW (Figure 6.17) indicated in bold red. The running time of the algorithm can be split into two parts. Each vertex is pushed onto S once and popped off S once, so the total time spent maintaining the stack (the red stuff) is $O(V)$. If we ignore the stack maintenance, the rest of the algorithm is just a

standard depth-first search. We conclude that the algorithm runs in $O(V+E)$ ***time***.

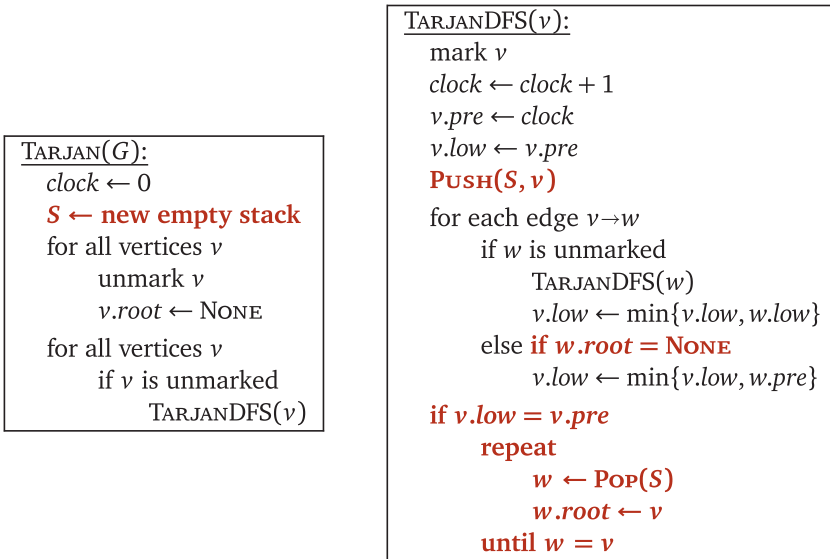

Figure 6.18. Tarjan's strong components algorithm.

Exercises

Depth-first search, topological sort, and strong components

0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.

   (b) Prove that for every directed graph G, the strong component graph $scc(G)$ is acyclic.

   (c) Prove that $scc(rev(G)) = rev(scc(G))$ for every directed graph G.

   (d) Fix an arbitrary directed graph G. For any vertex v of G, let $S(v)$ denote the strong component of G that contains v. For all vertices u and v of G, prove that v is reachable from u in G if and only if $S(v)$ is reachable from $S(u)$ in $scc(G)$.

   (e) Suppose S and T are two strong components in a directed graph G. Prove that either $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$, or $finish(u) > finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. A directed graph G is *semi-connected* if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both).

   (a) Give an example of a dag with a unique source that is *not* semi-connected.

(b) Describe and analyze an algorithm to determine whether a given directed *acyclic* graph is semi-connected.

(c) Describe and analyze an algorithm to determine whether an arbitrary directed graph is semi-connected.

2. The police department in the city of Shampoo-Banana has made every street in the city one-way. Despite widespread complaints from confused motorists, the mayor claims that it is possible to legally drive from any intersection in Shampoo-Banana to any other intersection.

   (a) The city needs to either verify or refute the mayor's claim. Formalize this problem in terms of graphs, and then describe and analyze an algorithm to solve it.

   (b) After running your algorithm from part (a), the mayor reluctantly admits that she was ~~lying~~ misinformed. Call an intersection x *good* if, for any intersection y that one can legally reach from x, it is possible to legally drive from y back to x. Now the mayor claims that over 95% of the intersections in Shampoo-Banana are good. Describe and analyze an efficient algorithm to verify or refute her claim.

   For full credit, both algorithms should run in linear time.

3. A vertex v in a connected undirected graph G is called a ***cut vertex*** if the subgraph $G - v$ (obtained by removing v from G) is disconnected.

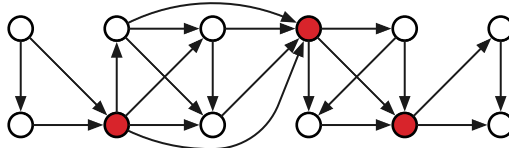

Figure 6.19. A dag with three cut vertices.

   (a) Describe a linear-time algorithm that determines, given a graph G and a vertex v, whether v is a cut vertex in G. What is the running time to find all cut vertices by trying your algorithm for each vertex?

   (b) Describe and analyze an algorithm that correctly determines whether a given directed *acyclic* graph contains a cut vertex. *[Hint: This is a warmup.]*

   (c) Let T be an arbitrary spanning tree of G, rooted at an arbitrary vertex r. For each vertex v, let T_v denote the subtree of T rooted at v; for example, $T_r = T$. Prove that v is a cut vertex of G if and only if G does not have an edge with exactly one endpoint in T_v.

(d) Describe a linear-time algorithm to identify every cut vertex in G. *[Hint: Let T be a depth-first spanning tree of G.]*

4. An edge e in a connected undirected graph G is called a ***cut edge*** if the subgraph $G - e$ (obtained by removing e from G) is disconnected.

(a) Given G and edge e describe a linear-time algorithm that determines whether e is a cut edge or not. What is the running time to find all cut edges by trying your algorithm for each edge?

(b) Let T be an arbitrary spanning tree of G. Prove that *every* cut edge of G is also an edge in T. This claim implies that G has at most $V - 1$ cut edges. How does this information improve your algorithm from part (a) to find all cut-edges?

(c) Now suppose we root T at an arbitrary vertex r. For any vertex v, let T_v denote the subtree of T rooted at v; for example, $T_r = T$. Let uv be an arbitrary edge of T, where u is the parent of v. Prove that uv is a cut edge of G if and only if uv is the only edge in G with exactly one endpoint in T_v.

(d) Describe a linear-time algorithm to identify every cut edge in G. *[Hint: Let T be a depth-first spanning tree of G.]*

5. The ***transitive closure*** G^T of a directed graph G is a directed graph with the same vertices as G, that contains any edge $u{\to}v$ if and only if there is a directed path from u to v in G. A ***transitive reduction*** of G is a graph with the smallest possible number of edges whose transitive closure is G^T. The same graph may have several transitive reductions.

(a) Describe an efficient algorithm to compute the transitive closure of a given directed graph.

(b) Prove that a directed graph G has a *unique* transitive reduction if and only if G is acyclic.

(c) Describe an efficient algorithm to compute a transitive reduction of a given directed graph.

6. One of the oldest algorithms for exploring arbitrary connected graphs was proposed by Gaston Tarry in 1895, as a procedure for solving mazes.[7] The input to Tarry's algorithm is an undirected graph G; however, for ease of presentation, we formally split each undirected edge uv into two directed edges $u{\to}v$ and $v{\to}u$. (In an actual implementation, this split is trivial;

[7] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, by Christian Wiener in 1873, and (implicitly) by Leonhard Euler in 1736. Wiener's algorithm is equivalent to depth-first search in a connected undirected graph.

the algorithm simply uses the given adjacency list for G *as though* G were directed.)

```
TARRY(G):
    unmark all vertices of G
    color all edges of G white
    s ← any vertex in G
    RECTARRY(s)
```

```
RECTARRY(v):
    mark v                          ⟨⟨"visit v"⟩⟩
    if there is a white arc v→w
        if w is unmarked
            color w→v green
        color v→w red              }⟨⟨"traverse v→w"⟩⟩
        RECTARRY(w)                }
    else if there is a green arc v→w
        color v→w red              }⟨⟨"traverse v→w"⟩⟩
        RECTARRY(w)                }
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red and recursively calls RECTARRY(w). Unlike our earlier graph traversal algorithm, Tarry's algorithm can mark same vertex multiple times.

(a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

(b) Prove that no directed edge in G is traversed more than once.

(c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

(d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s)+1$ times. This claim immediately implies that TARRY(G) terminates.

(e) Prove that the last vertex visited by TARRY(G) is the starting vertex s.

(f) For every vertex v that TARRY(G) visits, prove that all edges into v and out of v are red when TARRY(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

(g) Prove that TARRY(G) visits every vertex of G. This claim and the previous claim imply that TARRY(G) traverses every edge of G exactly once.

7. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      v.pre ← clock;  clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      v.post ← clock;  clock ← clock + 1
      RECTARRY2(w, clock)
```

Prove or disprove the following claim: When TARRY2(G) halts, the green edges define a spanning tree and the labels $v.pre$ and $v.post$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that TARRY2 produces the same *output* as depth-first search, even though it visits the edges in a completely different order.

8. You have a collection of n lock-boxes and m gold keys. Each key unlocks *at most* one box. However, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having one matching key in your hand), or smash it to bits with a hammer.

   Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

9. Suppose you are teaching an algorithms course. In your second midterm, you give your students a drawing of a graph and ask then to indicate a breadth-first search tree and a depth-first search tree rooted at a particular vertex. Unfortunately, once you start grading the exam, you realize that the graph you gave the students has several such spanning trees—far too many

to list. Instead, you need a way to tell whether each student's submission is correct!

In each of the following problems, suppose you are given a connected graph G, a start vertex s, and a spanning tree T of G.

(a) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

(b) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: It's not enough for T to be an unweighted shortest-path tree. Yes, this is the right chapter for this problem!]*

(c) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: Solve part (b) first.]*

(d) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

10. Several modern programming languages, including JavaScript, Python, Perl, and Ruby, include a feature called ***parallel assignment***, which allows multiple assignment operations to be encoded in a single line of code. For example, the Python code `x,y = 0,1` simultaneously sets `x` to 0 and `y` to 1. The values of the right-hand side of the assignment are all determined by the *old* values of the variables. Thus, the Python code `a,b = b,a` swaps the values of `a` and `b`, and the following Python code computes the nth Fibonacci number:

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev+curr, n-1
    return curr
```

Suppose the interpreter you are writing needs to convert every parallel assignment into an equivalent sequence of individual assignments. For example, the parallel assignment `a,b = 0,1` can be serialized in either order—either `a=0; b=1` or `a=0; b=1`—but the parallel assignment `x,y = x+1,x+y` can only be serialized as `y=x+y; x=x+1`. Serialization may require one or more additional temporary variables; for example, serializing `a,b = b,a` requires one temporary variable, and serializing `x,y = x+y,x-y` requires two temporary variables.

(a) Describe an algorithm to determine whether a given parallel assignment can be serialized without additional temporary variables.

(b) Describe an algorithm to determine whether a given parallel assignment can be serialized with *exactly one* additional temporary variable.