1. **Clearly** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist.

(a) A depth-first spanning tree rooted at $r$

**Solution:** There are *many* correct solutions; here are four.

(b) A breadth-first spanning tree rooted at $r$

**Solution:** There are exactly two correct solutions.

**Rubric (a) and (b)):** 2½ points each:
- No credit if the graph is not a spanning tree
- −1 for each misplaced (or misdirected) edge

(c) A topological order (list vertices below)

**Solution:** There are many correct solutions; here are four:

- $r, s, t, u, z, v, w, x, y$
- $r, u, s, t, z, w, v, x, y$
- $r, s, z, u, v, t, x, y, w$
- $r, s, t, z, u, v, x, w, y$

**Rubric:** 2½ points:
- No credit if any vertices are missing
- −1 for each pair out of order

(d) The strongly connected components (circle each component)

**Solution:** Because the graph is a DAG, every vertex is in its own strong component.
2. Describe and analyze an algorithm to solve arbitrary obtuse-angle mazes. You are given a connected undirected graph \( G \), whose vertices are points in the plane and whose edges are line segments, along with two vertices Start and Finish. Your algorithm should return True if \( G \) contains a walk from Start to Finish that has only obtuse angles, and False otherwise.

**Solution:** Let \( G = (V, E) \) be the input graph. Imagine moving a token along a valid walk through \( G \). At any time during the walk, the current state of the token is described by its current vertex and its previously vertex (if any). More formally, we reduce the angle-maze problem to a reachability problem in a new directed graph \( G' = (V', E') \) as follows.

- \( V' = \{ (u, v) \mid uv \in E \} \cup \{ \text{Start} \} \) — Each vertex \( (u, v) \) indicates that the token is currently at vertex \( v \) and its previously vertex \( u \). These are ordered pairs; \( G' \) has two distinct vertices for each edge in \( G \). In addition, we retain the Start vertex from \( G \), because the token initially has no “previous” location. There are \( 2E + 1 = O(E) \) vertices altogether.
  
  In fact, because \( G \) is planar, Euler’s formula implies that \( E \leq 3V - 6 \); on the other hand, because \( G \) is connected, we have \( E \geq V - 1 \). So the bound \( |V'| = O(V) \) is also correct, but it’s not actually better in this case.

- There are two types of edges:
  - Starting edges \( \{ \text{Start} \rightarrow (\text{Start}, v) \mid (\text{Start}, v) \in E \} \) — At the beginning, the token can follow any edge out of the Start vertex.
  - Regular edges \( \{ (u, v) \rightarrow (v, w) \mid 90^\circ < \angle uvw \leq 180^\circ \} \) — At every vertex in the walk after Start, the token can either continue straight or make an obtuse-angle turn.

Edges are directed. The total number of edges is \( O(E^2) \).

The number of edges is actually \( \sum_v O(\deg^2 v) \), which is better for reasonable graphs; however, in the worst case, the number of edges could actually be \( \Omega(E^2) \).

Suppose \( G \) is a tree with \( n \) leaves regularly spaced around a circle and one interior vertex at the center of the circle; then every vertex in \( G' \) has degree roughly \( n/2 \).

- We can construct \( G' \) in \( O(E^2) \) time by brute force.
- We need to decide if any vertex of the form \( (v, \text{Finish}) \) is reachable in \( G' \) from the Start vertex.
- We can solve this problem using whatever-first search in \( G' \). Specifically, we mark every vertex in \( G' \) that is reachable from Start, and then scan through all vertices \( (v, \text{Finish}) \) to see if any is marked.
- The reachability algorithm runs in \( O(V' + E') = O(E^2) \) time.

Again, because \( E = \Theta(V) \), the running time can also be bounded by \( O(V^2) \) and \( O(VE) \), but in this case, those are not actually better bounds than \( O(E^2) \). ■

**Rubric:** 10 points: standard graph-reduction rubric from HW6

- 2 points vertices
- 2 for edges (−1 for forgetting “directed”)
- 2 for correct problem (reachability)
- 2 for correct algorithms (whatever-first search)
  - ½ for “depth” or “breadth” instead of “whatever”
  - 1 for Dijkstra
- 2 for time analysis

**Solution:** We use a modified binary-search algorithm, similar to the algorithm to find local minima that we discussed in Lab 6. We can present the same algorithm either iteratively or recursively; the recursive formulation treats the input arrays $A$ and $B$ as global variables,

```plaintext
FINDCROSS($A[1..n], B[1..n]$):
    $lo \leftarrow 1$
    $hi \leftarrow n$
    while $lo \leq hi - 374$
        $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
        if $A[mid] < B[mid]$
            $lo \leftarrow mid$
        else
            $hi \leftarrow mid$
    search $A[lo..hi]$ and $B[lo..hi]$ by brute force

FINDCROSS($lo, hi$):
    if $hi - lo \leq 374$
        search $A[lo..hi]$ and $B[lo..hi]$ by brute force
    else
        $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
        if $A[mid] < B[mid]$
            return FINDCROSS($mid, hi$)
        else
            return FINDCROSS($lo, mid$)
```

In either formulation, the algorithm runs in $O(\log n)$ time. (Obviously there’s nothing special about the number 374 here.)

**Rubric:** 10 points = 3 for base case + 5 for recursive cases + 2 for running time. A correct $O(n)$-time solution is worth 5 points; a slower correct solution is worth 3 points; scale partial credit. These are not the only correct solutions. Common errors:

- $-1$ for infinite loop when $hi = lo + 1$.
- Max 5 points for “binary search” with no other details.
4. You have a collection of \( n \) lockboxes and \( m \) gold keys. Each key unlocks at most one box. Without a matching key, the only way to open a box is to smash it with a hammer. Your baby brother has locked all your keys inside the boxes! Luckily, you know which keys (if any) are inside each 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.

**Solution:** This problem reduces to reachability in a directed graph \( G = (V, E) \) defined as follows:

- The vertices \( V \) are the nonempty boxes. (Boxes that do not contain keys do not correspond to vertices.)
- The edges correspond to useful keys. Specifically, \( G \) contains the directed edge \( u \to v \) if and only if (1) box \( u \) contains a key to box \( v \) and (2) box \( v \) has at least one key inside it. (Any key that opens an empty box, or does not open a box at all, does not correspond to an edge.)
- Suppose your little brother has chosen box \( s \). We need to determine if every other box is reachable from \( s \) in \( G \).
  
  If \( G \) contains a directed path \( s \to v_1 \to v_2 \to \cdots \to v_\ell \), we can open all boxes on that path by smashing box \( s \) to get the key to box \( v_1 \), and then for all \( i > 1 \), unlocking \( v_{i-1} \) to get the key to \( v_i \). Thus, \( s \) is the only box we need to smash if and only if every other box is reachable from \( s \) in \( G \).
- We can solve this problem using whatever-first search from \( s \), and then verifying that every vertex of \( G \) (that is, every nonempty box) is marked.

The algorithm runs in \( O(V + E) = O(n + m) \) time.

**Rubric:** 5 points: standard graph-reduction rubric. \(-\frac{1}{2}\) for including empty boxes in the graph.
(b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

**Solution:** We build the same graph $G = (V, E)$ exactly as in part (a); in particular, we ignore any boxes that do not contain keys. Then we construct the strongly-connected component graph $SCC(G)$ using the Kosaraju-Sharir algorithm described in class. Finally, we return the number of sources (vertices with no incoming edges) in $SCC(G)$. The algorithm runs in $O(V + E) = O(n + m)$ time.

To prove that this algorithm is correct, we need to prove two claims:

**Claim 1.** We must smash at least one box in every source component.

**Proof:** For any vertex $v \in V$, let $S(v)$ denote the strong component containing $v$. The definition of $SCC(G)$ implies that if there is a walk in $G$ from vertex $u$ to vertex $v$, then there is a walk in $SCC(G)$ from component $S(u)$ to component $S(v)$. Recall from part (a) that we can open box $v$ if and only if $G$ contains a walk from some smashed box to $v$. (The smashed box might be $v$ itself.)

Let $S$ be any strong component of $G$ that is a source in $SCC(G)$, and let $v$ be any vertex in $S$. To open box $v$, we must smash at least one box $u$ such that $S(v)$ is reachable from $S(u)$ in $SCC(G)$. But because $S$ is a source in $SCC(G)$, the only strong component that can reach $S(v) = S$ is $S$ itself. Thus, to open $v$, we must smash at least one box in $S$. □

**Claim 2.** If we smash one box in every source component, we do not need to smash any other boxes.

**Proof:** Suppose we smash one box in every source component.

Let $v$ be any arbitrary box, and let $S(v)$ be the strong component of $G$ containing $v$. If we follow edges backward from $S(v)$ in $SCC(G)$, we eventually reach some source component $S$, because $SCC(G)$ is a dag. Let $u$ be the box in $S$ that we smash open. Because there is a walk from $S = S(u)$ to $S(v)$ in $SCC(G)$, there is a walk from $u$ to $v$ in $G$. We conclude that we can open $v$ without smashing any other boxes. □

**Rubric:** 5 points: standard graph-reduction rubric. The proof of correctness is not required. Again, $-\frac{1}{2}$ for including empty boxes in the graph.
5. Describe and analyze an algorithm to find the length of the longest palindrome subsequence of a given string \( A[1..n] \).

**Solution (dynamic programming):** Let \( LPS(i, j) \) denote the length of the longest palindrome subsequence of \( A[i..j] \). This function satisfies the following recurrence:

\[
LPS(i, j) = \begin{cases} 
0 & \text{if } i > j \\
1 & \text{if } i = j \\
\max \left\{ \begin{array}{ll}
LPS(i + 1, j) \\
LPS(i, j + 1)
\end{array} \right\} & \text{if } A[i] \neq A[j] \\
\max \left\{ \begin{array}{ll}
2 + LPS(i + 1, j - 1) \\
LPS(i + 1, j) \\
LPS(i, j + 1)
\end{array} \right\} & \text{otherwise}
\end{cases}
\]

We need to compute \( LPS(1, n) \).

We can memoize this function into an array \( LPS[1..n+1, 0..n] \), which we can fill with two nested loops, one decreasing \( i \) and the other increasing \( j \). (The nesting order doesn’t matter.) The resulting algorithm runs in \( O(n^2) \) time. ■

**Rubric:** 10 points: standard dynamic programming rubric. Max 7 points for \( O(n^3) \) time; max 5 points for slower polynomial time; scale partial credit. No penalty for greedily simplifying the last case of the recurrence to \( 2 + LPS(i + 1, j - 1) \) without proof.

**Solution (longest path):** We solve this problem by reducing it to a longest path problem in a dag \( G = (V, E) \) defined as follows:

- \( V = \{(i, j) | i \leq j\} \cup t \) — There is one vertex for each non-empty substring of \( A \), plus an artificial target vertex \( t \).
- The graph contains the following weighted directed edges, for all indices \( i \) and \( j \):
  - \( (i, j) \rightarrow (i + 1, j) \) with weight 0
  - \( (i, j) \rightarrow (i, j - 1) \) with weight 0
  - \( (i, j) \rightarrow (i + 1, j - 1) \) with weight 2 if \( A[i] = A[j] \) and weight 0 otherwise.
  - \( (i, i + 1) \rightarrow t \) with weight 2 if \( A[i] = A[i + 1] \) and weight 0 otherwise.
  - \( (i, i) \rightarrow t \) with weight 1

For every edge \( (i, j) \rightarrow (i', j') \), we have \( j - i > j' - i' \), so this graph is a dag.
- We need to compute the length of the longest path in \( G \) from \((1, n)\) to \( t \).
- We can find this length using depth-first search (or equivalently, dynamic programming) in \( G \), as described in class.

The algorithm runs in \( O(V + E) = O(n^2) \) time. ■

**Rubric:** 10 points: standard graph-reduction rubric. Max 7 points for \( O(n^3) \) time; max 5 points for slower polynomial time; scale partial credit. This is not the only correct formulation of this reduction.