Suppose $G$ has a unique source $s$ and a unique sink $t$. Describe and analyze an algorithm to find every $(s,t)$-cut vertex in $G$.

**Solution:** First, observe that every vertex $v$ can reach $t$: We can repeatedly follow outgoing edges starting at $v$, and $t$ will be the only vertex where there won’t be an outgoing edge to follow. Similarly, $s$ can reach every vertex $v$.

Now, consider a topological ordering $<$ of $G$’s vertices. Per the advice, we claim $v$ is an $(s,t)$-cut vertex if and only if there exists no edge $u\rightarrow v$ where $u < v$ and $v < w$. Suppose there does exist such an edge $u\rightarrow w$. Then, there is a path $P$ from $s$ to $u$, along $u\rightarrow w$, and from $w$ to $t$. Path $P$ does not include $v$, because its vertices must appear in topological order and $v$ is neither before $u$ nor after $w$ in ordering $<$. Therefore, $v$ is not an $(s,t)$-cut vertex. Now, suppose there is no such edge. Every path from $s$ to $t$ follows its vertices in topological order. We cannot “skip” $v$ along any path, because otherwise there would be an edge going directly from an earlier vertex to a later vertex in the ordering. We conclude that $v$ is an $(s,t)$-cut vertex in this case.

We now use the following algorithm based on the observation. We will scan the vertices in topological order, maintaining a counter `shortcuts` of the number of edges that start earlier than the current vertex and end later. We may conclude the current vertex is an $(s,t)$-cut vertex if and only if the counter is non-zero. The following algorithm outputs an array `cuts` containing all $(s,t)$-cut vertices of $G$.

```plaintext
FindCutVertices(V, E, s, t):
    count ← 0
    shortcuts ← 0
    for each vertex v in topological order
        for each edge u→v
            shortcuts ← shortcuts − 1
        if shortcuts > 0
            count ← count + 1
            cuts[count] ← v
        for each edge v→w
            shortcuts ← shortcuts + 1
    return cuts
```

Computing a topological order takes $O(V + E)$ time. The for loops go over every vertex once and every edge twice, so **the whole algorithm takes $O(V + E)$ time**.

**Rubric:** 10 points total: 5 points for the algorithm. 3 points for the proof of correctness. 2 points for running time analysis.
We say that an edge $e$ is **dangerous** if it is the heaviest edge in some cycle in $G$ and **useful** if it does not lie in any cycle in $G$.

(a) Prove that the minimum spanning tree of $G$ contains every useful edge.

**Solution:** Let $e = uv$ be any useful edge of $G$. The graph $G - e$ is disconnected, because a path $P$ in $G - e$ from $v$ to $u$ would imply $e$ is part of the cycle $P \circ e$. Every subgraph of $G - e$ is disconnected as well, so $G - e$ contains no spanning trees. Therefore, every spanning tree of $G$, including the minimum spanning tree, must include $e$. ■

**Rubric:** 2 points total.

(b) Prove that the minimum spanning tree of $G$ does not contain any dangerous edge.

**Solution:** Suppose the minimum spanning tree $T$ contains a dangerous edge $e = uv$. By the definition of dangerous, edge $e$ is the heaviest edge of some cycle $C$. Forest $T - e$ contains two components, one with vertex $u$ and the other with vertex $v$. If we follow $C - e$ from $u$ to $v$, we will encounter at least one edge $e'$ with one endpoint in each component. Therefore, $T' := T - e + e'$ is a spanning tree. Further, $w(T') = w(T) - w(e) + w(e') < w(T)$, because $e$ is the heaviest edge of $C$. Tree $T$ is not the minimum spanning tree after all. ■

**Rubric:** 3 points total.

(c) Describe and analyze an $O(V + E)$ time algorithm to determine if a given edge $uv$ belongs to a cycle.

**Solution:** Observe that $uv$ belongs to a cycle if and only if there is a path from $v$ to $u$ that does not use the edge $uv$. Therefore, we should check if $u$ is reachable from $v$ in the graph $G' := G - uv$. We call `WhateverFirstSearch(v)` in the graph $G'$ and return `TRUE` if and only if $u$ is marked. **Constructing $G'$ and running the search takes $O(V + E)$ time.** ■

**Rubric:** 2 points total: 1.5 points for the algorithm. 0.5 points for running time analysis.

(d) Describe and analyze an efficient implementation of the “reverse Kruskal” algorithm.

**Solution:** For each edge $e$ in decreasing weight order, we do the following. We check if $e$ is in a cycle. If so, we delete it from the graph; otherwise, we leave the edge alone and continue with the next edge in the list. When we are done looping over all the edges, we return what remains of the graph. Here’s some pseudocode:
ReverseKruskal\((V, E, w)\):
- sort \(E\) by decreasing weight
- for \(i \leftarrow 1\) to \(|E|\)
  - \(e \leftarrow i\)th heaviest edge in \(E\)
  - if \(e\) belongs to a cycle
    - remove \(e\) from \(E\)
- return \((V, E)\)

Correctness comes from the following observations: First, we never remove any edges from the minimum spanning tree. Further, after scanning the \(i\)th heaviest edge, each of the \(i\) heaviest edges has been either deleted or is useful. Indeed, consider when we scan the \(i\)th heaviest edge \(e\). Suppose it belongs to a cycle. Inductively, all heavier edges are useful and therefore not in that cycle. By part (b), we can remove \(e\) from \(G\) without harming the minimum spanning tree, and the second observation holds as well. Now, suppose \(e\) does not belong to a cycle. By definition, it is useful, so the observation holds.

The observations imply that by the time the for loop ends, every edge is useful, and the minimum spanning tree still exists in \(G\). From part (a), we know all the edges are in the minimum spanning tree. We conclude \(G\) itself is the minimum spanning tree by the time the algorithm ends.

Sorting the edges takes \(O(E \log E)\) time. Because \(G\) is connected, \(|E| \geq |V| - 1\), implying it takes \(O(E)\) time to determine if an edge is in a cycle. We do so \(|E|\) times, so the whole algorithm takes \(O(E^2)\) time.

Rubric: 3 points total: 2 points for the algorithm. 0.5 points for the proof of correctness. 0.5 points for running time analysis.