You are given the list of players and their ranking in each of the \( m \) games. Our goal is to design and analyze an algorithm that produces an overall ranking/ordering of the \( n \) players that is consistent with the individual game rankings, or correctly reports that no such ranking exists.

(a) What vertices should we use for our graph?

**Solution:** We want to order the players in some way, so we'll use the players themselves as the vertices.

**Rubric:** 1 point total.

(b) What edges should we use in our graph? Are they directed or undirected edges?

**Solution:** For every game \( G \), let \( A, B, \) and \( C \) be its players ranked first, second, and third, respectively. We add a **directed edge** from \( A \) to \( B \), form \( A \) to \( C \), and from \( B \) to \( C \). Therefore, there is now a directed edge from any player to any other player they outranked at least once.

**Rubric:** 2 points total.

(c) What standard algorithm should we call on our graph?

**Solution:** We want a ranking consistent with the directed edges, so we'll run the \( O(V + E) \) time topological sort we saw in class.

**Rubric:** 2 points total.

(d) How do we translate the output of the algorithm into the final rankings of our players?

**Solution:** If the topological sort fails, then there was a directed cycle in our graph. This represents a sequence of players \( \langle A, B, \ldots, Z, A \rangle \) where each successive player beat the other at least once. Player \( A \) cannot outrank themselves in the final ordering, so we report there is no legal ranking.

Otherwise, the topological ordering is the ranking of players from best to worst. There are no edges going from a lessor player to a better one, so we haven't contradicted any one game.

**Rubric:** 2 points total.

(e) What is the running time of the graph algorithm **in terms of** \( V \) and \( E \), the number of vertices and edges.

**Solution:** A topological sort can be performed in \( O(V + E) \) time.
(f) What is the running time of the whole algorithm in terms of the input parameters $n$, the number of players, and $m$, the number of games?

**Solution:** We have $V = n$, because there are $n$ players, and $E = 3m$, because there are three edges created per game. The total running time is $O(n + m)$. ■

**Rubric:** 2 points total. -1 for excluding one of $n$ or $m$. 

---

Rubric: 1 point total.
(a) Prove that the \textit{maximum} spanning tree of $G$ contains the widest paths between \textit{every} pair of vertices.

\textbf{Solution:} Suppose the maximum spanning tree $T$ \textit{did not} contain the widest path between two vertices $s$ and $t$. Let $uv$ be the lightest edge on the unique $s$, $t$-path in $T$. Edge $uv$ is not the widest path between $u$ and $v$ either; otherwise, we could walk from $s$ to $u$, along a wider path from $u$ to $v$, and then from $v$ to $s$ to get a wider path from $s$ to $t$. Removing $uv$ from $T$ disconnects $T$ into two pieces. There is at least once edge $e$ from the widest $uv$ path that connects these two pieces, and $w(e) > w(uv)$. Tree $T' = T - uv + e$ weights more than $T$, contradicting $T$ being a maximum spanning tree.

\textbf{Rubric:} 3 points total.

(b) Suppose $B$ is the bottleneck distance between $s$ and $t$.

i. Prove that deleting any edge with weight less than $B$ does not change the bottleneck distance between $s$ and $t$.

\textbf{Solution:} Every edge on the widest $s$ to $t$ path survives the deletion, because they are all heavier than the deleted edges. Therefore, the bottleneck distance is at least $B$ after the deletion. No new paths are created by the deletion, including wider ones, so the bottleneck distance is at most $B$ after the deletion as well.

\textbf{Rubric:} 1 point total.

ii. Prove that contracting any edge with weight \textit{greater} than $B$ does not change the bottleneck distance between $s$ and $t$.

\textbf{Solution:} Consider the widest path between $s$ and $t$ before the contraction. After the contraction, every edge on the path, other than the contracted one perhaps, remains, possibly replaced by wider ones. Therefore, the bottleneck distance is at least $B$ after the contraction. Now, suppose the bottleneck distance actually increased, and let $p$ be the new wider path between $s$ and $t$. There was a path from $s$ to $t$ containing every edge of $p$ and possibly the contracted edge. If it did contain the contracted edge, that edge is heavier than $B$. Therefore, there was a path from $s$ to $t$ that was wider than $B$ before contraction, contradicting the definition of $B$.

\textbf{Rubric:} 2 points total.

(c) Describe an algorithm to solve the following problem in $O(E)$ time: Given a connected undirected graph $G$, two vertices $s$ and $t$, and a weight $W$, is the bottleneck distance between $s$ and $t$ at least $W$?

\textbf{Solution:} We make a copy of the graph. In the copy, we remove all edges with weight strictly less than $W$. Then, we run \textsc{WhateverFirstSearch}(s) in the copy. If $t$ is marked, we say the bottleneck distance is at least $W$. Otherwise, the bottleneck distance is less than $W$. 

\textbf{Rubric:} 1 point total.
If the bottleneck distance is less than $W$, there is no path from $s$ to $t$ with edges all weighing at least $W$, meaning we destroy any path from $s$ to $t$ during the deletion and the algorithm correctly returns ‘no’. Otherwise, part (b)i guarantees the bottleneck distance will remain at least $W$ during the deletions and a path will exist.

It takes $O(E)$ time to delete the edges, and searching a connected graph takes $O(E)$ time as well, so the whole algorithm runs in $O(E)$ time.

**Rubric:** 1.5 points total.

(d) Describe an algorithm to compute a widest path between $s$ and $t$ in $O(E)$ time. You may assume you have access to a procedure $\text{CONTRACTBIGGER}(G, W)$ that contracts all edges of weight greater than input parameter $W$ in $O(E)$ time.

**Solution:** Per the hint, we’ll use a median-finding algorithm to binary search for the lightest edge on the widest path, reducing the number of edges by half in every iteration.

```
\text{FINDBOTTLENECK}(G, s, t):
    if $E = 1$
        return only $s$ to $t$ path
    else
        $W$ ← median edge weight found by MoMSELECT
        if bottleneck distance between $s$ and $t$ is at least $W$
            $E_\prec$ ← edges of weight less than $W$
            $G'$ ← connected component of $G - E_\prec$
            return $\text{FINDBOTTLENECK}(G', s, t)$
        else
            $E_\succ$ ← edges of weight greater than $W$
            $G'$ ← $\text{CONTRACTBIGGER}(G, W)$
            $p$ ← $\text{FINDBOTTLENECK}(G', s, t)$
            $G''$ ← $E_\succ + p$  \(\text{Include only contracted edges or those on} \ p\)\)
            return path to $t$ found by $\text{WHATEVERFIRSTSEARCH}(G'', s)$
```

We can learn if the bottleneck distance is at most $W$ from part (c). From part (b), the deletions or contractions don’t change the bottleneck distance, so the recursive calls find a path of the correct width. The extra code at the bottom is to reconstruct the final path if includes a contracted edge.

All steps outside the recursive calls run in $O(E)$ time. The number of edges during the recursive calls is half that of the original call, so the running time follows the recurrence $T(E) = T(E/2) + E$. We’ve seen before (using recursion trees) that this recurrence solves to $O(E)$.

**Rubric:** 2.5 points total. Accurately returning only the bottleneck distance instead of the widest path is worth full credit.
Describe a modification of the Prim-Jarník minimum spanning tree algorithm that runs in $O(V^2)$ time (independent of $E$) when the input graph is dense, using only simple data structures (and in particular, without using a Fibonacci heap).

**Solution:** We’ll use the Prim-Jarník algorithm from class that stores the vertices in the priority queue (called JARNík in Erickson 7, page 7). Instead of using a binary heap or a Fibonacci heap, we’ll use the following simple priority queue data structure:

Suppose we number the vertices 1 through $n$. Our priority queue is implemented as an array $queue[1..V]$. Entry $queue[i]$ holds the key of vertex $i$ if it is in the priority queue or $\infty$ otherwise. Before starting the loop in JARNíkInit, we set every entry of $queue$ to be $\infty$. To INSERT vertex $i$ with key $key(i)$, we set $queue[i] \leftarrow key(i)$ in constant time. To perform a DECREASEKey($i$, new), we again set $queue[i] \leftarrow new$. Finally, EXTRACTMin scans $queue$ for its minimum entry in $O(V)$ time, returns the index $i$ containing that minimum entry, and sets $queue[i] \leftarrow \infty$.

Each of the $O(V + E) = O(V^2)$ INSERT and DECREASEKey operations take $O(1)$ time. Each of the $V - 1$ EXTRACTMin operations take $O(V)$ time. The total running time of our implementation is $O(V^2)$.

**Rubric:** 10 points total: 7 for the algorithm and necessary justifications, 3 for the running time analysis.