1. Suppose we are given a tree T with weighted edges. Describe and analyze an algorithm to find a matching in T with maximum total weight.

Solution (7/10): For any vertex v of the input tree T, we define two functions:

- $MWM(v)$ is the weight of a maximum weight matching in the subtree rooted at v, where v may or may not be incident to a matching edge.
- $MWM^*(v)$ is the weight of a maximum weight matching in the subtree rooted at v, *where v is **not** incident to a matching edge*.

We need to compute $MWM(root(T))$. These two functions satisfy the following mutual recurrences, where $wt(e)$ denotes the weight of edge e and $w \downarrow v$ denotes that w is a child of v.

$$MWM^*(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \displaystyle\sum_{w \downarrow v} MWM(w) & \text{otherwise} \end{cases}$$

$$MWM(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max \left\{ \begin{array}{c} MWM^*(v) \\ \displaystyle\max_{w \downarrow v} \left(\begin{array}{c} wt(vw) + MWM^*(w) \\ \displaystyle + \sum_{x \downarrow v, x \neq w} MWM(x) \end{array} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize these functions into two new fields at each node of T, and we can evaluate the functions in postorder. For each vertex v, evaluating $MWM^*(v)$ and $MWM(v)$ requires $O(\deg(v)^2)$ time, so the overall algorithm runs in $\boldsymbol{O(n^2)}$ ***time*** in the worst case. ■

Solution (10/10): We can speed up the previous solution by modifying the recurrence for MWM. The basic observation is that the sum over x in the previous recurrence is equal the sum in the recurrence for MWM^*, minus one term.

$$MWM(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max \left\{ \begin{array}{c} MWM^*(v) \\ \displaystyle + \max_{w \downarrow v} \left(\begin{array}{c} wt(vw) + MWM^*(w) \\ + MWM^*(v) - MWM(w) \end{array} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can still memoize these functions into two new fields at each node of T, and we can evaluate the functions in postorder.

For each vertex v, evaluating $MWM^*(v)$ and $MWM(v)$ with this new recurrence requires $O(\deg(v))$ time, so the overall algorithm runs in $\boldsymbol{O(n)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. Max 7 points for $O(n^2)$-time algorithms; scale partial credit. These are not the only correct solutions.

2. Suppose we are given a convex polygon P with an even number of vertices $p_0, p_1, \ldots, p_{2n-1}$, indexed in order around the boundary. Describe an algorithm to compute diagonals that partition P into convex quadrilaterals (4-sided polygons), where the area of the smallest quadrilateral is as large as possible.

Assume you have access to a subroutine QUADAREA(i, j, k, l) that computes the area of the quadrilateral with vertices p_i, p_j, p_k, p_l in $O(1)$ time.

Solution (find the best quad incident to an edge): For any indices i and j such that $0 \le i < j \le 2n-1$, let $P[i..j]$ denote the convex polygon with vertices $p_i, p_{i+1}, \ldots, p_j$. Let $MaxMinQ(i, j)$ denote the maximum, over all quadrangulations of $P[i..j]$, of the area of the smallest quadrilateral. We need to compute $MaxMinQ(0, 2n-1)$.

In every quadrangulation of $P[i..j]$, the line segment $p_i p_j$ must be an edge of some quadrilateral. The following recurrence considers all possible value for the other two vertices of this quadrilateral:

$$MaxMinQ(i,j) = \begin{cases} -\infty & \text{if } j-i \text{ is even} \\ \infty & \text{if } j = i+1 \\ \max\left\{ \min\left\{ \begin{array}{c} \text{QUADAREA}(i,i',j',j) \\ MaxMinQ(i,i') \\ MaxMinQ(i',j') \\ MaxMinQ(j',j) \end{array} \right\} \middle| \; i < i' < j' < j \right\} & \text{otherwise} \end{cases}$$

The first base case ensures that we only consider subproblems where $j-i$ is even; the second base case correctly handles the empty quadrangulation and simplifies the conditions of the main recursive case.

We can memoize this function into a two-dimensional array $MaxMinQ[0..2n-1, 0..2n-1]$, which we can fill with two nested loops, decreasing i in one loop and increasing j in the other. (It doesn't matter which of these is the outer loop.) Each entry $MaxMinQ[i, j]$ can be computed in $O(n^2)$ time (by looping over i' and j'), so the overall algorithm runs in $\boldsymbol{O(n^4)}$ ***time***. ■

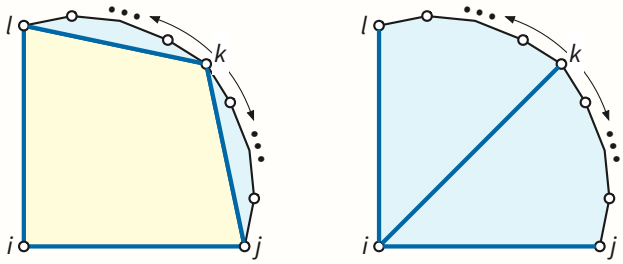
Solution (find the best quad(s) incident to a vertex): For this solution, the subproblems consider polygons defined by an interval of vertices of P, *plus one additional vertex*. Specifically, for any indices i, j, and l such that $0 \le i < j < l \le 2n-1$, let $P[i; j..k]$ denote the convex polygon with vertices $p_i, p_j, p_{j+1}, \ldots, p_l$. Let $MaxMinQ(i; j, l)$ denote the maximum, over all quadrangulations of $P[i; j..l]$, of the area of the smallest quadrilateral. We need to compute $MaxMinQ(0, 1, 2n-1)$.

In any partition of $P[i; j..l]$ into quadrilaterals, there are two possibilities for the local structure at p_i:

- Exactly one quadrilateral touches p_i. This quadrilateral has vertices p_i, p_j, p_k,

and p_k, for some index $j < k < l$.

- At least two quadrilaterals touch p_i. In this case, the partition uses the diagonal between p_i and p_k, for some index $j < k < l$.



This case analysis implies that the function *MaxMinQ* satisfies the following recurrence:

$$MaxMinQ(i;j,l) = \begin{cases} -\infty & \text{if } l-j \text{ is odd} \\ \max\left\{ \left. \begin{array}{l} \min\left\{ \begin{array}{l} MaxMinQ(i;j,k) \\ MaxMinQ(i;k,l) \end{array} \right\} \\ \min\left\{ \begin{array}{l} \textsc{QuadArea}(i,j,k,l) \\ MaxMinQ(j;j+1,k) \\ MaxMinQ(k;k+1,l) \end{array} \right\} \end{array} \right| j < k < l \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a three-dimensional array $MaxMinQ[0..2n-1, 0..2n-1, 0..2n-1]$, which we can fill with three nested loops, decreasing the first two indices i and j and increasing the third index l. (It doesn't matter how these loops are nested.) Each entry $MaxMinQ[i,j,l]$ can be computed in $O(n)$ time (by looping over all possible values of k), so the overall algorithm runs in $\boldsymbol{O(n^4)}$ ***time***. ■

Rubric: 10 points, standard dynamic programming rubric. These are not the only correct solutions.

This is not the fastest algorithm known for this problem. Specifically, the first solution can be modified to run in $O(n^3)$ time by exploiting the following geometric lemma: *The smallest quadrilateral in any partition uses at least two edges of P.* This lemma (which requires a proof!) essentially lets us search for indices i' and j' in independent subproblems. Feel free to ask Jeff for details in office hours.

3. Describe and analyze an algorithm to determine who wins the game between the Doctor and River, assuming both players play perfectly.

Solution: We start by topologically sorting the input dag G, because that's *always* the first thing one does with a dag. Topological sort labels the vertices with integers from 1 to V, so that every edge points from a lower label to a higher label. Because s is the only source, its label is 1, and because t is the only sink, its label is V.

We represent the two players with booleans: TRUE means the Doctor, and FALSE means River. For any vertices d and r and any boolean *who*, let $WhoWins(d, r, who)$ denote the winning player when the Doctor's token starts on d, River's token tarts on r, player *who* moves first, and both players play perfectly. We need to compute $WhoWins(s, t, \text{TRUE})$.

If the game is not over, then the Doctor wins moving first if and only if *at least one* move by the Doctor leads to a position where the Doctor wins moving second, and the Doctor wins moving second if and only if *every* move by River leads to a position where the Doctor wins moving first. (This is the recursive *definition* of "play perfectly", for *any* finite two-player game that cannot end in a draw.) Thus, the function *WhoWins* can be computed by the following recursive algorithm:

```
WhoWins(d, r, who):
  if d = r
      return TRUE
  else if d = t or r = s
      return FALSE
  else if who = TRUE
      return ⋁_{d→d'} WhoWins(d', r, FALSE)
  else if who = FALSE
      return ⋀_{r'→r} WhoWins(d, r', TRUE)
```

Thanks to our initial topological sort, we can memoize this function into a $V \times V \times 2$ array, indexed by the variables d, r, and *who* in that order. We can fill the array with two nested for loops, decreasing d in one loop and increasing r in the other, considering both players inside the inner loop. The nesting order of the two for-loops doesn't matter. Explicit pseudocode appears on the next page.

Time analysis: For any node v in G, let $indeg(v)$ denote the number of edges entering v (the *in-degree* of v), and let and $outdeg(v)$ denote the number of edges leaving v (the *out-degree* of v). For almost every pair of vertices d and r, our algorithm considers all $outdeg(d)$ possible moves for the Doctor and then all $indeg(r)$ possible moves for River. Thus, the total running time of our algorithm is at most

$$\sum_{d=1}^{V}\sum_{r=1}^{V} O\left(outdeg(d) + indeg(r)\right).$$

Ignoring the big-Oh constant, we can evaluate this sum in two pieces:

$$\sum_{d,r} outdeg(d) = \sum_{r}\left(\sum_{d} outdeg(d)\right) = \sum_{r} E = VE$$

$$\sum_{d,r} indeg(r) = \sum_{d}\left(\sum_{r} indeg(r)\right) = \sum_{d} E = VE$$

Less formally, our algorithm considers all VE pairs (Doctor's position, River's move) and all VE pairs (Doctor's move, River's position), spending $O(1)$ time on each pair. We conclude that our algorithm runs in ***O(VE) time***.

```
WHOWINS(V, E):
  label vertices of G in topological order
  for d ← V down to 1
      for r ← 1 to V
          if d = r
              WhoWins[d, r, TRUE] ← TRUE
              WhoWins[d, r, FALSE] ← TRUE
          else if d = t or r = s
              WhoWins[d, r, TRUE] ← FALSE
              WhoWins[d, r, FALSE] ← FALSE
          else
              doctor ← FALSE
              for all edges d→v
                  doctor ← doctor ∨ WhoWins[v, r, FALSE]
              WhoWins[d, r, TRUE] ← doctor
              river ← FALSE
              for all edges v→r
                  river ← river ∧ WhoWins[d, v, TRUE]
              WhoWins[d, r, FALSE] ← river
  return WhoWins[s, t, TRUE]
```

■

Rubric: 10 points, standard dynamic programming rubric. This solution is more detailed than necessary for full credit. −1 for looser time bounds like $O(V^3)$ or $O(V^2D)$ where D is maximum degree.

This is not the only correct description of this algorithm. For example, we can simplify the recurrence slightly by asking whether the *first* player wins, and then NANDing the results of recursive calls, instead of alternating between ANDs and ORs. More significantly, instead of topologically sorting the dag at the beginning, we can discover the correct traversal orders on the fly via depth-first search. Thus, we can rewrite the first three lines of WHOWINS as follows:

```
WHOWINS(V, E):
  for all vertices d in postorder                ⟨⟨via DFS(G, s)⟩⟩
      for all vertices r in reverse postorder    ⟨⟨via DFS(rev(G), t)⟩⟩
          ⟨⟨and so on⟩⟩
```

No penalty for implicitly assuming that the input graph has more than 10^{100} vertices; we can solve the small cases by brute force in $O(1)$ time. In particular, no penalty for implicitly assuming that the input graph has more than ***two*** vertices. (If the graph has only one or two vertices, then *both* players win!) Finally, no penalty for assuming without proof that $V = O(E)$; because the graph G has only one source, it must be connected, and therefore $E \geq V - 1$.

Solution (configuration graph search): First we construct the configuration graph $H = (V', E')$, which contains a vertex for every possible game configuration and a directed edge for every legal move. Specifically:

- $V' = V \times V \times \{0, 1\}$. Each vertex (d, r, who) represents the configuration where the Doctor's token is at node d, River's token is at node r, and it is the Doctor's turn if and only if $who = 1$. The number of vertices in H is $2V^2$.
- $E' = \big\{(d, r, 1) \to (d', r, 0) \;\big|\; d \to d' \in E\big\} \cup \big\{(d, r, 0) \to (d, r', 1) \;\big|\; r' \to r \in E\big\}$. Each edge represents a legal move by the appropriate player. The number of edges in H is $\sum_r \sum_d outdeg(d) + \sum_d \sum_r indeg(r) = 2VE$.

If H contains a directed cycle, it must have the form

$$(d_0, r_0, 1) \to (d_1, r_0, 0) \to (d_1, r_1, 1) \to \cdots \to (d_k, r_k, 1) \to (d_0, r_k, 0) \to (d_0, r_0, 1).$$

But then the original input dag would contain the cycles $d_0 \to d_1 \to \cdots \to d_k \to d_0$ and $r_0 \to r_k \to \cdots \to r_1 \to r_0$, which is impossible. We conclude that H is also a dag.

For each vertex (d, r, who) let $WhoWins(d, r, who) = \textsc{True}$ if the Doctor wins from the configuration (d, r, who) if both players play perfectly; otherwise, let $WhoWins(d, r, who) = \textsc{False}$. We need to compute $WhoWins(s, t, 1)$. We can evaluate this function recursively as follows:

```
WhoWins(d, r, who):
  if d = r
      return True
  else if d = t or r = s
      return False
  else if who = 1
      return ⋁_{(d,r,1)→(d',r,0)} WhoWins(d', r, 0)
  else if who = 0
      return ⋀_{(d,r,0)→(d,r',1)} WhoWins(d, r', 1)
```

We can memoize this function into the vertices of H themselves, and we can evaluate this function at every vertex of H by considering the vertices in reverse topological order (or equivalently, in DFS postorder). The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(VE)}$ ***time***. ■

Rubric: 10 points = 5 for correctly defining H (= 2 for vertices + 2 for edges + 1 for proving H is a dag) + 5 for solving the problem on H (dynamic programming rubric)