(a) Describe and analyze an algorithm to compute the maximum-weight spanning tree of a given edge-weighted graph.

**Solution:** Let $G = (V, E)$ be the given undirected graph with edge weights $w : E \rightarrow \mathbb{R}$. Let $w' : E \rightarrow \mathbb{R}$ be the negation of $w$ so that for all edges $e \in E$, $w'(e) = -w(e)$. We compute the weights $w'$ in $O(E)$ time and then compute and return a minimum weight spanning tree of $G$ using weights $w'$ using any of the algorithms from lecture. Computing the spanning tree takes $O(E \log V)$ time, so the total running time is $O(E \log V)$.

This algorithm is correct, because the lighter a spanning tree is using weights $w'$, the heavier it is using weights $w$.

**Rubric:** 5 points total: 4 points for the algorithm; 1 point for the analysis.

(b) Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of a given edge-weighted graph.

**Solution:** Let $G = (V, E)$ be the given undirected graph with edge weights $w : E \rightarrow \mathbb{R}$. To begin, we will assume every edge has non-negative weight. We compute a maximum weight spanning tree $T = (V, E')$ of $G$ using the algorithm from part (a). Then we compute and return edge subset $F = E \setminus E'$. Computing the maximum weight spanning tree takes $O(E \log V)$ time. There are several ways to compute $F$ in no additional time including modify the maximum weight spanning tree algorithm to remember every edge we do not add to the spanning tree or marking vertices added to the tree and setting $F$ to be every unmarked vertex. The total running time is $O(E \log V)$.

This algorithm is correct (assuming non-negative weights), because the edges outside any feedback edge set form an acyclic subgraph. Maximizing the weight of these edges by building a maximum weight spanning tree minimizes the weight of the edges in the feedback edge set.

If $G$ does contain some negative weight edges, we include all of them in an edge subset $F'$. Then, we compute the components in $G - F'$ using COUNTANDLABEL from lecture, compute minimum weight feedback edge sets for each of those components, and return all the edges in these feedback edge sets and $F'$. Computing $F'$ and the components of $G - F'$ takes $O(V + E)$ time total. Every edge is a member of at most one component, so computing all the feedback edge sets takes $O(E \log V)$ time total. The running time of the algorithm is still $O(E \log V)$.

In this case, correctness follows because we can add a negative weight edge to any feedback edge set and get an even lighter feedback edge set. Therefore, we should include all negative weight edges. After that, we're just trying to remove cycles from the remaining components as cheaply as possible.

**Rubric:** 5 points total assuming non-negative edge weights: 4 points for the algorithm; 1 point for the analysis. +2 points extra credit for correctly handling the negative edge weight case.
Suppose you are given an integer $k$ and an array $A[1..n]$ of $n$ distinct integers, sorted in increasing order. Describe and analyze a recursive algorithm to determine whether there is an index $i$ such that $A[i] = i + k$ in $O(\log n)$ time.

**Solution:** The divide-and-conquer procedure $\text{Find}(k, A[1..n])$ determines if there is an index $i$ such that $A[i] = i + k$ given that $A$ is an array of $n$ distinct integers, sorted in increasing order.

$$
\text{Find}(k, A[1..n]):
\begin{align*}
&\text{if } n = 0 \\
&\quad \text{return ‘no’} \\
&\text{else} \\
&\quad m \leftarrow \lfloor n/2 \rfloor \\
&\quad \text{if } A[m] = m + k \\
&\quad \quad \text{return ‘yes’} \\
&\quad \text{else if } A[m] > m + k \\
&\quad \quad \text{return } \text{Find}(k, A[1..m-1]) \\
&\quad \text{else} \\
&\quad \quad \text{return } \text{Find}(k+m, A[m+1..n])
\end{align*}
$$

The algorithm is a binary search with constant time per recursive call, so it takes $O(\log n)$ time overall.

We’ll prove the algorithm correct by induction on $n$. If $n = 0$, then there are no elements at all, so the algorithm is correct to return ‘no’. For larger $n$, assume inductively that the algorithm is correct on arrays of length $n’$ where $0 \leq n’ < n$. If $A[m] = m + k$, then $m$ is the index we are looking for and the algorithm is correct to return ‘yes’. If $A[m] > m + k$, then for every index $i > m$, $A[i] > i + k$, because the array contains distinct integers sorted in increasing order; the elements grow too quickly for any of the indices to the right to work for us. The algorithm is correct to consider only elements of lessor index than $m$, and the recursive call is correct by induction. Finally, if $A[m] < m + k$, then for every index $i < m$, $A[i] < i + k$. The algorithm is correct to consider only elements of greater index than $m$. However, the recursive call will not recognize the same indices as the call on $A[1..n]$, so the algorithm adds $m$ to $k$ in the recursive call so the sum of the real indices from $A[1..n]$ and $k$ is correctly computed.

**Rubric:** 10 points extra credit total (scaled to 1/8 of the midterm): 5 points for the algorithm; 3 points for the proof; 2 points for the analysis
For all integers \( i, j \) such that \( 0 \leq i \leq m \) and \( 0 \leq j \leq n \), let \( SCS(i, j) \) denote the length of the shortest common supersequence of sequences \( A[1 .. i] \) and \( B[1 .. j] \).

(a) Give a recurrence definition or describe a simple recursive algorithm for computing \( SCS(i, j) \). You do not need to analyze the algorithm if you choose to describe one. Don’t forget to explain why your solution is correct.

**Solution:** \( SCS \) can be defined recursively as follows.

\[
SCS(i, j) = \begin{cases} 
  j & \text{if } i = 0 \\
  i & \text{if } j = 0 \\
  1 + \min \{SCS(i-1, j), SCS(i, j-1)\} & \text{if } A[i] \neq B[j] \\
  1 + \min \{SCS(i-1, j), SCS(i, j-1), SCS(i-1, j-1)\} & \text{otherwise}
\end{cases}
\]

The shortest common supersequence of \( A[1 .. i] \) and \( B[1 .. j] \) must contain every character of both sequences, so if \( i = 0 \), then it is equal to \( B[1 .. j] \) and has length \( j \), and if \( j = 0 \) then it is equal to \( A[1 .. i] \) and has length \( i \). If \( i, j > 0 \), then there are two possibilities. If \( A[i] \neq B[j] \), then the shortest common supersequence needs to cover one of those two characters. The recurrence counts 1 for that character, and considers the better option of that character coming from \( A \) and continuing to work with the remaining \( i - 1 \) characters of \( A \) or that character coming from \( B \) and continuing to work with the remaining \( j - 1 \) characters of \( B \). If \( A[i] = B[j] \), then there is a third option. That common character could be the end of the shortest common supersequence, covering both the last character of \( A \) and \( B \). For this case, we still add 1 for that last character, but now we only have the first \( i - 1 \) characters of \( A \) and the first \( j - 1 \) characters of \( B \) left to cover.

(b) Use your solution from part (a) to describe and analyze an \( O(n^2) \) time dynamic programming algorithm to compute the length of the shortest common supersequence of \( A[1 .. m] \) and \( B[1 .. n] \). Your solution to part (a) must be correct to receive any credit for this part.

**Solution:** We can store the recursive answers from \( SCS(i, j) \) in a 2-dimensional array \( SCS[0 .. m][0 .. n] \). Each entry \( SCS[0][j] \) is set to \( j \) and each entry \( SCS[i][0] \) is set to \( i \) as the base cases. Every other entry depends upon entries of lower \( i \) or \( j \) index, so we can fill the array row-by-row, column-by-column in increasing order of indices (so we use an outer for loop over \( i \) increasing and an inner for loop over \( j \) increasing). There are \( O(mn) \) entries to fill in constant time each, so the total running time is \( O(mn) \). (Saying the running time is \( O(n^2) \) is fine for full credit, but you should still explain filling the whole 2D array.)

**Rubric:** 5 points extra credit total (scaled to 1/16 of the midterm): 3 points for the algorithm or recurrence; 2 points for proof of correctness.