(a) Using an exchange argument, prove there exists a conflict-free schedule of maximum size which contains the class that starts last.

**Solution:** Let $s$ be the class that starts last. Suppose we have a maximal conflict-free schedule $X$ that does not include $s$. Let $t$ be the last class in $X$ to start. Since $s$ starts after $t$ does, $s$ cannot conflict with any class in the set $X \setminus \{t\}$. Thus, the schedule $X' = X \cup \{s\} \setminus \{t\}$ is also conflict-free. Since $X'$ has the same size as $X$, it is also maximal.

**Rubric:** 3 points total. Proof must resemble an exchange argument for credit (so no merely appealing to symmetry).

For parts (b) and (c), consider a weighted version of the class scheduling problem, where different classes offer a different number of credit hours (totally unrelated to the duration of the class lectures).

(b) Prove that the greedy algorithm suggested above—choose the class that starts last and recurse—does not always return an optimal solution for the weighted version of the problem.

**Solution:** Consider the following input that consists of only classes 1 and 2 that partially overlap. $S[1] = 0$, $F[1] = 2$, and $C[1] = 100$. $S[2] = 1$, $F[2] = 3$, and $C[2] = 1$. The optimal schedule includes class 1 only for 100 credit hours. The greedy algorithm takes class 2 for 1 credit hour and cannot add any other classes, making it vastly suboptimal.

**Rubric:** 2 points total. Simply drawing an example and indicating the number of credit hours for each class is sufficient for full credit.

(c) Describe and analyze a dynamic programming algorithm that always computes an optimal schedule.

**Solution:** Per the hint, we'll assume the classes are indexed in increasing order of starting time. Imagine choosing classes to take in increasing order by starting time and we're about to decide if we should take class $i$. If we don't take it, we can still choose from all later available classes. If we do take it, we have to choose from all classes that start after $i$ finishes. Those classes form a suffix of our sorted list, so it suffices to define recursive subproblems using only the index of the first class in the suffix.

We'll focus on computing the maximum number of credit hours available. Let $MaxCredits(i)$ denote the maximum number of credit hours for any conflict-free subset of classes $i$ through $n$. Our ultimate goal is to compute $MaxCredits(1)$ and its associated schedule. If $i > n$, then there are no classes to choose from and $MaxCredits(i) = 0$. If we choose to ignore a class $i$ with $i \leq n$, then we still have classes $i + 1$ through $n$ available and we can acquire $MaxCredits(i + 1)$ credit hours. If we choose to take class $i$, then let $k$ be the index of the
first class to start after \( i \) ends. All classes \( i + 1 \) through \( k - 1 \) conflict with \( i \) but \( k \) through
\( n \) don’t, so we can get our \( C[i] \) credit hours from class \( i \) plus however many we can get
from classes \( k \) through \( n \) for \( C[i] + \text{MaxCredits}(k) \) hours total. We see
\[
\text{MaxCredits}(i) = \begin{cases} 
0 & \text{if } i > n \\
\max \{ \text{MaxCredits}(i + 1), C[i] + \text{MaxCredits}(k) \} & \text{otherwise}
\end{cases}
\]
where \( k \) is equal to the minimum index \( j \) such that \( S[j] > F[i] \) or \( n + 1 \) if no suitable
choice for \( j \) exists.

Subproblems are parameterized by \( 1 \leq i \leq n+1 \), so we can use an array \( \text{MaxCredits}[1..n+1] \) to store subproblem solutions. Each entry depends upon at most two others with larger
parameter, so we’ll fill the array right to left. There are \( O(n) \) subproblems to solve. It
would take linear time to solve each if we manually checked every possible choice \( j \) to
compute \( k \). However, the starting times are sorted, so we can perform a binary search to
find \( k \) in \( O(\log n) \) time. The table can then be filled in \( O(n \log n) \) time. \textbf{Even including the
time needed to sort, the total running time will be} \( O(n \log n) \).

The following pseudocode computes the optimal number of credit hours. I’ve also
added some code return the best schedule as well.

\[
\begin{align*}
\text{BestSchedule}(S[1..n], F[1..n], C[1..n]): \\
&\text{Sort } S[1..n] \text{ and permute } F[1..n] \text{ and } C[1..n] \text{ to match.} \\
&\text{MaxCredits}[n+1] \leftarrow 0 \\
&\text{for } i \leftarrow n \text{ down to } 1 \\
&\text{skip} \leftarrow \text{MaxCredits}[i+1] \\
&k \leftarrow \min j \text{ s.t. } S[j] > F[i] \text{ or } n + 1 \text{ if no such } j \text{ exists (computed by binary search)} \\
&\text{take} \leftarrow C[i] + \text{MaxCredits}[k] \\
&\text{if } \text{skip} > \text{take} \\
&\text{MaxCredits}[i] \leftarrow \text{skip} \\
&\text{include}[i] \leftarrow \text{FALSE} \\
&\text{next}[i] \leftarrow i + 1 \\
&\text{else} \\
&\text{MaxCredits}[i] \leftarrow \text{take} \\
&\text{include}[i] \leftarrow \text{TRUE} \\
&\text{next}[i] \leftarrow k \\
&\text{count} \leftarrow 0 \\
&i \leftarrow 1 \\
&\text{while } i \leq n \\
&\text{if } \text{include}[i] \\
&\text{count} \leftarrow \text{count} + 1 \\
&X[count] \leftarrow i \\
&i \leftarrow \text{next}[i] \\
&\text{return } X[1..\text{count}]
\end{align*}
\]

\textbf{Rubric:} 5 points total: 2.5 points for the recurrence; −1 point for no justification,
−0.5 points for messing up base cases. 1.5 points for memoization and the iterative
algorithm. 1 point for the running time analysis. Max 4 points for a \( \Theta(n^2) \) worst-case
running time. Recurrence must be reasonable to get algorithm and running time points.
An algorithm that merely computes the maximum number of credit hours is still worth
full credit.
Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color \( X \).

**Solution:** Per the hint, we’ll start by sorting the intervals by their left endpoint. Now, consider a procedure that colors intervals in this order. We’ll use positive integers as our colors. Suppose we’re about to color interval \( i \). There will be some subset of already colored intervals overlapping with interval \( i \), and we cannot use their colors to color interval \( i \). Instead, we’d like to greedily choose the “smallest” color for \( i \) that does not conflict with the overlapping intervals.

I claim there does in fact exist a proper coloring of the intervals consistent with the earlier choices where the coloring uses as few colors as possible, and interval \( i \) gets the smallest available color \( c \). Consider an optimal coloring \( C \) that is consistent with the choices made before coloring interval \( i \). If \( C \) assigns color \( c \) to \( i \), then we are done. Otherwise, it assigns some other color \( c' \) to \( i \). Consider the following exchange to create a different coloring \( C' \) based on \( C \). We take all intervals \( i \) through \( n \) with color \( c' \) in \( C \) and instead color them \( c \) in \( C' \). Also, we take all intervals \( i \) through \( n \) with color \( c \) in \( C \) and instead color them \( c' \) in \( C' \). None of these intervals we just changed overlapped with any other intervals of color \( c \) or \( c' \), so we created no conflicts with other intervals. Likewise, the intervals originally colored \( c \) don’t overlap and the intervals originally colored \( c' \) don’t overlap, so we created no conflicts between intervals we just recolored. This exchange did not increase the total number of colors (and may have decreased it if \( C \) was not using color \( c \) originally), so \( C' \) is a new optimal schedule that does give color \( c \) to \( i \).

The optimal coloring given by the above approach uses a number of colors equal to the maximum color it assigns. In turn, this maximum color is equal to the maximum number of intervals that all overlap at a common point. To efficiently compute this value, we’ll loop through the sorted list of intervals, maintaining a list of active intervals as we go. We’ll use a priority queue \( Q \) of interval finish times for the list of active intervals so we can efficiently remove intervals from the list that are no longer active. (The actual intervals indices don’t matter, so we’ll store only keys in this priority queue.)

```plaintext
MinimizeColors(L[1..n],R[1..n]):
Sort L[1..n] and permute R[1..n] to match
colors ← 0
Q ← an empty priority queue
for i ← 1 to n
    while NotEmpty(Q) and Min(Q) < L[i]
        ExtractMin(Q)
        Insert(Q,R[i])
    if colors < Size(Q)
        colors ← Size(Q)
return colors
```

Each finish time gets added to and removed from the priority queue exactly once. Other operations include sorting the arrays and a linear number of constant time or priority queue operations. If we use a min-heap with \( O(\log n) \) time operations for our priority queue, the total running time is \( O(n \log n) \).
**Rubric:** 10 points total: 4 points for the algorithm; 4 points for arguing that greedily coloring or returning the max number of overlapping intervals is correct; 2 points for running time analysis. A correct $O(n^2)$ time algorithm is worth 8 points.