Formally, you’re given three arrays \( S[1..n] \), \( F[1..n] \), and \( C[1..n] \) where \( S[i] \), \( F[i] \), and \( C[i] \) respectively give the start time, end time, and credit hours for course \( i \). You must compute a credit-maximal conflict-free schedule defined as a subset \( X \subseteq \{1,2,…,n\} \) that maximizes \( \sum_{i \in X} C[i] \) so that for each \( i, j \in X \), either \( S[i] > F[j] \) or \( S[j] > F[i] \).

(a) Prove that the greedy algorithm given in class—choose the class that ends first and recurse—does not always return a credit-maximal conflict-free schedule.

**Solution:** Suppose there are only two classes to choose from, one with 2 credit hours and one with 1 credit hour. If the low credit one finishes first, then the greedy algorithm will choose only that class for 1 hour total, but the optimal schedule will take the other class for 2 credit hours.

Given the formal definitions, we can say \( S = \langle 0, 1 \rangle \), \( F = \langle 2, 3 \rangle \), and \( C = \langle 1, 2 \rangle \).

**Rubric:** 3 points total

(b) Describe an algorithm to compute a credit-maximal conflict-free schedule in \( O(n^2) \) time.

**Solution:** We will go with a dynamic programming solution. Suppose the classes are sorted in increasing order of starting time (this supposition can be enforced in \( O(n \log n) \) time by running a merge sort on \( S \) and permuting \( F \) and \( C \) to match). For all integers \( i \) such that \( 1 \leq i \leq n+1 \), let \( \text{MaxCredit}(i) \) be the maximum number of credit hours from a credit-maximal conflict-free schedule using only classes \( 1 \) through \( i \) (so \( \text{MaxCredit}(n+1) = 0 \)). Let \( \text{Next}(i) = \min\{ \min\{ j \mid S[j] > F[i] \}, n+1 \} \) be the first class to start after class \( i \) finishes or \( n+1 \) if no such class exists. \( \text{MaxCredit} \) can be defined recursively as follows.

\[
\text{MaxCredit}(i) = \begin{cases} 
0 & \text{if } i = n+1 \\
\max\{\text{MaxCredit}(i+1), C[i] + \text{MaxCredit}(\text{Next}(i))\} & \text{otherwise}
\end{cases}
\]

Indeed, for \( i < n+1 \), we can either skip class \( i \), leaving all remaining classes from \( i \) through \( n \) available, or we can take class \( i \). In the latter case, we must skip all classes that start before class \( i \) finishes, and then try to maximize our credit hours from the remaining classes \( \text{Next}(i) \) through \( n \).

We can store the recursive answers in an array \( \text{MaxCredit}[1..n+1] \). \( \text{MaxCredit}[n+1] \) is set to 0 as a base case. Every other entry depends upon one or two entries to its right, so we can fill the array from right to left (high index to low). There are \( O(n) \) entries to fill. However, it takes \( O(n) \) time to compute \( \text{Next}(i) \) since we need to loop through classes \( i + 1 \) through \( n \). Therefore, computing all the entries takes \( O(n^2) \) time total.

We can compute the actual list of classes in \( O(n) \) time after filling the table if we also mark each entry \( i \) with whether or not the max had us take class \( i \).

**Rubric:** 7 points total: 3 points for the recurrence, -1 for wrong base case, off-by-one errors, or failing to sort first; 2 points for filling a data structure; 2 points for the analysis. The recurrence must be correct to get credit for the data structure or analysis. Merely computing the maximum number of credit hours obtainable is still worth full credit.
Describe and analyze a greedy algorithm to compute the smallest full path of $X$ as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in $X$. Don’t forget to prove your greedy algorithm is correct!

**Solution:** We will use the following greedy approach to compute a full path $Y$ of $X$: Of all the intervals covering the leftmost uncovered point (initially all points are uncovered), include in $Y$ the one with the greatest right value and recurse. This approach can be implemented (and better explained) using the procedure `GREEDY_PATH`:

```
GREEDY_PATH($X_L[1..n], X_R[1..n]$):
    sort $X_L$ and permute $X_R$ to match
    count ← 0
    prev ← $-\infty$  // Where we ended last
    i ← 1
    while i ≤ n
        start ← max{prev, $X_L[i]$}
        farthest ← i
        while i + 1 ≤ n and max{prev, $X_L[i+1]$} = start
            i ← i + 1
            if $X_R[i] > X_R[farthest]$  
                farthest ← i
        count ← counts + 1
        $Y[count] ← farthest$
        prev ← $X_R[farthest]$
        i ← i + 1
    return $Y[1..count]$
```

To explain further, the outer while loop looks for an interval to include in the path. The inner while loops selects the interval that ends farthest to the right among all intervals covering the leftmost point not already covered by current members of $Y$. Note that since we take the furthest of those intervals into $Y$, all the others considered by the inner while loop have been completely covered and it is fine to ignore them in future iterations of the outer loop.

We still need to prove it is correct to take the interval ending farthest to the right among those intervals containing the leftmost uncovered point (initially, all points are uncovered). Given that fact, the algorithm must be correct by induction on intervals covering the points not covered by that first choice. Let $Y$ be a minimal full path with intervals ordered left to right by their leftmost point. Let $f$ be the first interval in $Y$, and let $g$ be the interval ending farthest to the right among all intervals containing the leftmost uncovered point. If we replace $f$ with $g$ in $Y$, we will still cover everything covered by $f$ without increasing the size of $Y$, proving our algorithm’s first choice is correct.

The algorithm performs an $O(n \log n)$ time sort at the beginning. Otherwise, there is a constant amount of work to do for each of $n+1$ values of $i$, meaning the rest of the algorithm takes $O(n)$ time. The total running time is $O(n \log n)$.

---

**Rubric:** 10 points total: 5 points for the algorithm; 3 points for the proof; 2 points for the analysis