1) **The GreedySchedule algorithm we described for the class scheduling problem is not the only greedy strategy we could have tried. For each of the following alternative greedy strategies, either prove that the resulting algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). [Hint: Three of these algorithms are actually correct.]**

⇨ **(a) Choose the course x that ends last, discard classes that conflict with x, and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 11:00 AM

- Course B: 10:00 AM - 12:00 PM

- Course C: 12:00 PM - 2:00 PM

The algorithm might choose Course C first (since it ends last), and then discard Courses A and B. However, the optimal choice is to choose both A and B.

⇨ **(b) Choose the course x that starts first, discard all classes that conflict with x, and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 11:00 AM

- Course B: 10:00 AM - 12:00 PM

- Course C: 11:30 AM - 1:30 PM

The algorithm might choose Course A first (since it starts first), and then discard Courses B and C. However, the optimal choice is to choose both B and C.

⇨ **(c) Choose the course x that starts last, discard all classes that conflict with x, and recurse.**

This strategy is correct and produces an optimal schedule. The algorithm selects the last starting course, which ensures the earliest possible end times for the remaining courses.

⇨ **(d) Choose the course x with the shortest duration, discard all classes that conflict with x, and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 10:00 AM

- Course B: 9:30 AM - 11:30 AM

- Course C: 11:00 AM - 1:00 PM

The algorithm might choose Course A first (shortest duration), and then discard Courses B and C. However, the optimal choice is to choose both B and C.

⇨ **(e) Choose a course x that conflicts with the fewest other courses, discard all classes that conflict with x, and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 11:00 AM

- Course B: 10:00 AM - 12:00 PM

- Course C: 10:30 AM - 11:30 AM

The algorithm might choose Course A first (fewest conflicts), and then discard Courses B and C. However, the optimal choice is to choose both B and C.

⇨ **(f) If no classes conflict, choose them all. Otherwise, discard the course with the longest duration and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 10:00 AM

- Course B: 10:30 AM - 12:00 PM

The algorithm might choose both A and B. However, the optimal choice is to choose only Course B.

⇨ **(g) If no classes conflict, choose them all. Otherwise, discard a course that conflicts with the most other courses and recurse.**

This strategy doesn't always produce an optimal schedule. Consider the following example:

- Course A: 9:00 AM - 10:00 AM

- Course B: 10:30 AM - 11:30 AM

- Course C: 10:30 AM - 12:00 PM

The algorithm might choose Course A first (fewest conflicts), and then discard Courses B and C. However, the optimal choice is to choose both B and C.

⇨ **(h) Let x be the class with the earliest start time, and let y be the class with the second earliest start time.**

• If x and y are disjoint, choose x and recurse on everything but x.

• If x completely contains y, discard x and recurse.

• Otherwise, discard y and recurse.

This strategy is correct and produces an optimal schedule. It handles cases where there is containment or overlap between two courses' time intervals.

(i) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that ends last, discard all classes that conflict with y, and recurse.

This strategy is correct and produces an optimal schedule. It prioritizes courses that don't conflict with others and then chooses the course with the latest end time.

In summary, three of the alternative greedy strategies (c, h, and i) are correct and always construct optimal schedules, while the others have scenarios where they fail to produce optimal schedules.

2) Let X be a set of n intervals on the real line. We say that a subset of intervals Y ⊆ X covers X if the union of all intervals in Y is equal to the union of all intervals in X. The size of a cover is just the number of intervals. Describe and analyze an efficient algorithm to compute the smallest cover of X. Assume that your input consists of two arrays L[1 .. n] and R[1 .. n], representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.

⇨ To find the smallest cover of a set of intervals X, we can use a greedy algorithm known as the "Interval Scheduling Algorithm." This algorithm iteratively selects the interval with the earliest ending point that does not overlap with any previously selected interval.

Here's how the algorithm works:

1. **Initialization:** Sort the intervals in ascending order based on their right endpoints.

2. **Greedy Selection:** Start with an empty set S to store the selected intervals. Iterate through the sorted intervals. For each interval, if its left endpoint is after or equal to the right endpoint of the last selected interval (or if it's the first interval), add it to the set S.

3. **Output:** The set S now contains the selected intervals that form the smallest cover of X.

Proof of Correctness:
The correctness of the Interval Scheduling Algorithm can be established by considering the optimal solution. Let O be the optimal set of intervals that

cover X. We'll show that the algorithm's solution S is not larger than O, which implies that S is optimal.

Consider any interval o from the optimal set O. Since the algorithm selects intervals with the earliest ending points, the first interval in S that covers or overlaps with o will also have an ending point that is at least as early as the ending point of o. Therefore, S includes an interval that is at least as good (or better) as o for covering X. Thus, S is not larger than O, and it's also a cover of X.

Algorithm Analysis:
- Sorting the intervals takes O(n log n) time.
- Iterating through the sorted intervals and selecting non-overlapping intervals takes O(n) time.
- Overall, the algorithm's time complexity is dominated by the sorting step, making it O(n log n) in the worst case.

In summary, the Interval Scheduling Algorithm provides an efficient and correct way to compute the smallest cover of a set of intervals X. It's a greedy algorithm that makes locally optimal choices by selecting intervals with the earliest ending points that do not overlap with previously selected intervals. The algorithm's correctness is proven by comparing its solution to an optimal solution, and its time complexity is O(n log n), primarily due to the sorting step.

3) **(a) For every integer n, find a frequency array f [1 .. n] whose Huffman code tree has depth n − 1, such that the largest frequency is as small as possible.**
(b) Suppose the total length N of the unencoded message is bounded by a polynomial in the alphabet size n. Prove that the any Huffman tree for the frequencies f [1 .. n] has depth O(log n).

⇨ **Part (a):**

For a Huffman code tree, the depth of the tree is determined by the frequencies of the symbols. To achieve a depth of n - 1 in the Huffman tree, we need to arrange the frequencies in a specific way. We want to minimize the maximum frequency while ensuring that the depths add up to n - 1.

Here's a way to construct such a frequency array f [1 .. n]:

1. Set f[i] = 1 for all i from 1 to n - 1.
2. Set f[n] = 2^(n - 1) - (n - 1).

This arrangement ensures that the Huffman tree has a depth of n - 1 while minimizing the maximum frequency.

Part (b):

To prove that any Huffman tree for frequencies f[1 .. n] has depth O(log n), we can use the concept of "weighted external path length," which is a measure of the expected depth of nodes in a Huffman tree.

The weighted external path length L(T) of a tree T is defined as the sum of the products of the depth of each leaf node and its frequency. Formally, L(T) = Σ (depth(i) * f[i]), where depth(i) is the depth of the i-th leaf node in tree T, and f[i] is its frequency.

For a Huffman tree, it's known that the weighted external path length is minimized. Since the total length N of the unencoded message is bounded by a polynomial in the alphabet size n, we can say that the frequencies f[1 .. n] are polynomially bounded as well.

Now, let's consider the weighted external path length L(T) for any Huffman tree T:

$$\begin{aligned} L(T) &= \Sigma\,(\text{depth}(i) * f[i]) \\ &\le d\_max * \Sigma\, f[i] \quad \text{(where d\_max is the maximum depth in T)} \\ &= d\_max * N \quad \text{(since } \Sigma\, f[i] = N\text{, the total length of the message)} \end{aligned}$$

Since N is polynomially bounded in n, the weighted external path length L(T) is also polynomially bounded in n.

For a Huffman tree, the number of leaf nodes is n, and the average weighted external path length (L(T) / n) is polynomially bounded. This implies that the average depth of a Huffman tree is O(log n).

Therefore, any Huffman tree for frequencies f[1 .. n] has depth O(log n), given that the total length of the unencoded message is bounded by a polynomial in the alphabet size n.