Problem 1. (50 points)
Let $I_1, I_2, \ldots, I_n$ be a set of closed intervals on the real line, with $I_i = [a_i, b_i]$. Design an efficient greedy algorithm to compute the smallest set $S$ of points such that each interval contains at least one point. Analyze the time complexity of your algorithm and prove that it always produces the optimal solution.

Answer: Here is the proposed greedy algorithm

**Algorithm** MinimalPointsSet$(I_1 = (a_1, b_1), \ldots, I_n = (a_n, b_n))$

1. Sort $I_1, I_2, \ldots, I_n$ by the right-endpoint $b_i$
2. $S \leftarrow \{b_1\}$
3. $i \leftarrow 1$
4. for $j \leftarrow 2$ to $n$ do
   - if $a_j > b_i$ do
     - $S \leftarrow S \cup \{b_j\}$
     - $i \leftarrow j$
5. 

Time complexity: $O(n \log n)$ due to sorting.

Clearly, the solution produced by the algorithm covers all intervals since an interval is removed only after a point within it has been chosen.

Greedy choice: We first need to prove that first point chosen belongs to some optimal solution. Assume that the solution produced by the algorithm is a sequence of points $S = b_1, \ldots, b_k$ and that the optimal solution is $S' = b'_1, \ldots, b'_k$, where both sequences are ordered in ascending order. Let us compare $b_1$ with $b'_1$. We claim that $b_1 \geq b'_1$. In fact, $b_1$ is the rightmost point that be chosen in order to hit interval $I_1$, and the optimal solution clearly has to hit that interval as well. We also claim that if $I_j$ is an interval hit by $b'_1$, then $I_j$ is also hit by $b_1$. In fact, for $I_j$ to be hit by $b'_1$, it is necessary that $a_j \leq b'_1$ which implies that $a_j \leq b_1$ which implies that $b_1$ hits $I_j$ (since $b_j \geq b_1$ by the choice of $b_1$). Therefore choosing $b'_1$ instead of $b_1$ does not improve the solution.

Optimal substructure: After our algorithm have chosen the first point, we are left with a subset of intervals to cover. We need to prove that if $S'$ is optimal for the set $\{i \in S : b_1 \not\in I_i\}$, then $S = \{b_1\} \cup S'$ is optimal for the original problem. If we could find a solution $T = \{b_1\} \cup T'$ such that $|T| < |S|$, then $T'$ would be a better solution for the set $\{i \in S : b_1 \not\in I_i\}$ contradicting the optimality of $S'$.

Problem 2. (50 points)
In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. They seek an algorithm that will enable them to make change of $n$ units using the minimum number of coins.

1. (20 pts) The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of $n$ units using US denominations. Prove its correctness and analyze its time complexity.

2. (10 pts) Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$. 

3. (20 pts) Give dynamic programming algorithm that correctly determines the minimum number of coins needed to make change of \( n \) units using denominations \( \{d_1, \ldots, d_k\} \). Analyze its running time.

**Answer:** Here is the greedy algorithm.

**Inputs:** number of units to make change for \( n \)

**Outputs:** number of half dollars, quarter, dimes, nickels, and pennies to use \((c_{50}, c_{25}, c_{10}, c_5, c_1)\).

**Algorithm** MAKECHANGE\( (n) \)

\[
\begin{align*}
c_{50} & = n \div 50 \\
n & = n \mod 50 \\
c_{25} & = n \div 25 \\
n & = n \mod 25 \\
c_{10} & = n \div 10 \\
n & = n \mod 10 \\
c_5 & = n \div 5 \\
n & = n \mod 5 \\
c_1 & = n
\end{align*}
\]

**return** \((c_{50}, c_{25}, c_{10}, c_5, c_1)\)

Because the algorithm always performs 10 calculations, its worst-case running time is \( O(1) \).

**Proof of Optimality:** Assume that the best non-greedy solution for a given instance of the problem is \((b_{50}, b_{25}, b_{10}, b_5, b_1)\), where \( n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_5 + b_1 \). We show that the greedy solution is as good as or better than the best solution. The greedy solution is \((c_{50}, c_{25}, c_{10}, c_5, c_1)\).

Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If \( b_{50} < c_{50} \) then \( 25b_{25} + 10b_{10} + 5b_5 + b_1 \geq 50 \). To satisfy the given inequality these are all the possibilities.

1. if \( b_{25} \geq 2 \), replace with 1 half-dollar
2. if \( b_{25} = 1 \) we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins
3. if \( b_{25} = 0 \) we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If \( b_{50} = c_{50} \) and \( b_{25} < c_{25} \) then \( 10b_{10} + 5b_5 + b_1 \geq 25 \). These are the possibilities.

1. if \( b_{10} \geq 3 \), replace with 1 quarter and 1 nickel
2. if \( b_{10} = 2 \) we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter
3. if \( b_{10} = 1 \) we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter
4. if \( b_{10} = 0 \) we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if \( b_{50} = c_{50}, b_{25} = c_{25}, b_{10} = c_{10}, \) and \( b_5 < c_5 \).

2) We can show that the greedy algorithm doesn’t work for all possible denominations by giving a counter-example. If \( n = 12 \) and \( (d_1, d_2, d_3) = (1, 6, 10), \) then the greedy algorithm would return \( (c_{10}, c_6, c_1) = (1, 0, 2) \). However, the optimal solution is \( (c_{10}, c_6, c_1) = (0, 2, 0) \).

3) Given a list of \( k \) coin values, \((d_1, d_2, \ldots, d_k)\), and a number \( n \), we want to find the integers \((c_{d_1}, c_{d_2}, \ldots, c_{d_k})\) such that
\[
\sum_{i=1}^{k} c_{d_i} d_i = n
\]
and that \( \sum_{i=1}^{k} c_{d_i} \) is minimal.

Our subproblems consist of the optimal change set for 1 through \( n \). To keep track of the optimal solution for each subproblem we will use an array called \( sumc \) which is indexed by subproblem. (i.e. \( sumc[i] \) contains the least number of coins needed to make change for \( i \) units. \( sumc[d_1] = 1, sumc[d_2] = 1, \ldots, sumc[d_k] = 1 \)

**Inputs:** denominations \((d_1, d_2, \ldots, d_k)\), units \( n \)

**Outputs:** the count of each denomination \((c_{d_1}, c_{d_2}, \ldots, c_{d_k})\).

**Algorithm** MakeChange\((n, (d_1, d_2, \ldots, d_k))\)

```plaintext
for i ← 1 to n do
    sumc[i] ← ∞
for j ← 1 to k do
    sumc[d_j] ← 1; coin[d_j] ← j
// calculate sumc[i] for 1 ≤ i ≤ n
for i ← 1 to n do
    for j ← 1 to k do
        temp ← sumc[i - d_j] + 1
        if temp < sumc[i] then
            sumc[i] ← temp; coin[i] ← j
// determine if it is possible to make change
if sumc[n] = 1 then return impossible
else // generate answer
    for j ← 1 to sumc[n] do
        c_{d_j} = 0 // initialization
    // traverse through coins used to make best change
    total ← n
    while total > 0 do
        cd_{coin[total]} ← cd_{coin[total]} + 1
        total ← total - d_{coin[total]}
    return (c_{d_1}, c_{d_2}, \ldots, c_{d_k})
```

The running time of the above algorithm is \( O(nk) \). Note that this algorithm is pseudo-polynomial.