1. (a) Order the requests by non-increasing client time, *i.e.*, so that $c_{i_1} \geqslant c_{i_2} \geqslant \ldots \geqslant c_{i_n}$.
   This takes time $\Theta(n \log n)$, for sorting.

   (b) We introduce some notation to make the proof easier to write. For any permutation $S = i_1, i_2, \ldots, i_n$ of $[1, 2, \ldots, n]$ and for $1 \leqslant k \leqslant n$, let $T_k(S) = s_{i_1} + s_{i_2} + \cdots + s_{i_k} + c_{i_k}$ denote the completion time for the k-th request. The overall completion time is then $T(S) = \max\{T_1(S), T_2(S), \ldots, T_n(S)\}$.

   To prove that the algorithm produces an optimal ordering, we show that every initial portion of the solution $(i_1, i_2, \ldots, i_k)$ can be extended to an optimal solution, by induction on k.

   Base Case: The empty sequence is extended by all optimal solutions.

   Ind. Hyp.: Suppose $k \geqslant 0$ and $i_1, \ldots, i_k$ can be extended to some optimal $S^* = i_1, \ldots, i_k, i^*_{k+1}, \ldots, i^*_n$.

   Ind. Step: Consider the greedy choice for the next index, i_{k+1}. If $i_{k+1} = i^*_{k+1}$ then we are done: S^* already extends $i_1, \ldots, i_k, i_{k+1}$.

   Else, let j be such that $i^*_j = i_{k+1}$. Since S^* starts with $i_1, \ldots, i_k$, and $i^*_{k+1} \neq i_{k+1}$, we know that $k + 2 \leqslant j \leqslant n$. Because of the way i_{k+1} is chosen, we also know that $c_{i_{k+1}} \geqslant c_{i^*_{k+1}}, c_{i_{k+1}} \geqslant c_{i^*_{k+2}}, \ldots, c_{i_{k+1}} \geqslant c_{i^*_n}$.

   Let $S' = i_1, \ldots, i_k, i_{k+1}, i^*_{k+2}, \ldots, i^*_{j-1}, i^*_{k+1}, i^*_{j+1}, \ldots, i^*_n$ (*i.e.*, S' is the same as S^* except that $i_{k+1} = i^*_j$ and i^*_{k+1} have been swapped). To compare $T(S')$ and $T(S^*)$, we examine the completion time of individual requests:

   - $T_1(S') = T_1(S^*), \ldots, T_k(S') = T_k(S^*)$ because both S' and S^* start with the same $i_1, \ldots, i_k$.
   - $$\begin{aligned} T_{k+1}(S') &= s_{i_1} + \cdots + s_{i_k} + s_{i_{k+1}} + c_{i_{k+1}} \\ &\leqslant s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i_{k+1}} + c_{i_{k+1}} \\ &= T_j(S^*) \\ T_{k+1}(S^*) &= s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + c_{i^*_{k+1}} \\ &\leqslant s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i_{k+1}} + c_{i^*_{k+1}} \\ &\leqslant s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i_{k+1}} + c_{i_{k+1}} \\ &= T_j(S^*) \end{aligned}$$
   - Similarly, $T_{k+2}(S') \leqslant T_j(S^*), \ldots, T_{j-1}(S') \leqslant T_j(S^*)$ and $T_{k+2}(S^*) \leqslant T_j(S^*), \ldots, T_{j-1}(S^*) \leqslant T_j(S^*)$.
   - $$\begin{aligned} T_j(S') &= s_{i_1} + \cdots + s_{i_k} + s_{i_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i^*_{k+1}} + c_{i^*_{k+1}} \\ &= s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i_{k+1}} + c_{i^*_{k+1}} \\ &\leqslant s_{i_1} + \cdots + s_{i_k} + s_{i^*_{k+1}} + s_{i^*_{k+2}} + \cdots + s_{i^*_{j-1}} + s_{i_{k+1}} + c_{i_{k+1}} \\ &= T_j(S^*) \end{aligned}$$
   - $T_{j+1}(S') = T_{j+1}(S^*), \ldots, T_n(S') = T_n(S^*)$ because both S' and S^* contain the same requests after request number j.

   Hence,

$$\begin{aligned} T(S') &= \max\{T_1(S'), \ldots, T_k(S'), T_{k+1}(S'), T_{k+2}(S'), \ldots, T_{j-1}(S'), T_j(S'), T_{j+1}(S'), \ldots, T_n(S')\} \\ &\leqslant \max\{T_1(S'), \ldots, T_k(S'), T_j(S^*), T_{j+1}(S'), \ldots, T_n(S')\} \\ &= \max\{T_1(S^*), \ldots, T_k(S^*), T_j(S^*), T_{j+1}(S^*), \ldots, T_n(S^*)\} \\ &= \max\{T_1(S^*), \ldots, T_k(S^*), T_{k+1}(S^*), T_{k+2}(S^*), \ldots, T_{j-1}(S^*), T_j(S^*), T_{j+1}(S^*), \ldots, T_n(S^*)\} \\ &= T(S^*). \end{aligned}$$

   Since S^* was optimal, this means $T(S') = T(S^*)$. So S' is optimal and extends $i_1, \ldots, i_{k+1}$.

   Since every initial portion of the greedy permutation can be extended to an optimal solution, in particular, the final permutation itself is optimal.

2. (a) Counter-example: $G = (V, E)$ with $V = \{a, b, c\}$, $E = \{(a, b), (b, c)\}$, $c(a, b) = c(b, c) = 1$, and $L = \{b\}$. Since G is already a tree, there is only one spanning tree of G (G itself), and b is not a leaf in this tree.

   (b) Consider the graph $G = (V, E)$ with $V = \{a, b, c\}$, $E = \{(a, b), (b, c), (c, a)\}$, $c(a, b) = c(b, c) = 1$, $c(a, c) = 2$, and $L = \{b\}$. Then G contains exactly one MST: $T = \{(a, b), (b, c)\}$, but T is not a solution to the MST with Fixed Leaves problem because b is not a leaf in T.

   With the same input, there are two optimal solutions to the MST with Fixed Leaves problem: $T_1 = \{(a, c), (a, b)\}$ and $T_2 = \{(a, c), (b, c)\}$. Neither of these is a MST in G.

   (c)
```
# This is a variation of Kruskal's algorithm: we construct the tree edge-by-edge,
# starting with the nodes in L.
# Handle the only special case when two nodes of L must be connected to each other.
if V = {a, b}, E = {(a, b)}, L = {a, b}:   return {(a, b)}
# Now, handle the general case.
T ← ∅
# Start by selecting one ege (v, u) for each node v ∈ L, where u ∉ L and c(v, u) is minimum.
for v ∈ L:
    c ← ∞
    for (v, u) ∈ E:
        # Remove all edges adjacent to v from E, to ensure v is a leaf in T.
        E ← E − {(v, u)}
        if u ∉ L and c(v, u) < c:
            e ← (v, u)
            c ← c(v, u)
    # At this point, e is a minimum-cost edge connecting v to V − L.
    T ← T ∪ {e}
Now, run Kruskal's algorithm on the remaining graph, starting from the edges already in T.
return T
```

   The algorithm's complexity is $\mathcal{O}(m \log m)$, same as Kruskal's, since that is the part of the algorithm that takes the longest.

   (d) Let $G_0 = G - L$ (G_0 is G with every node of L removed, and every edge that contains a node from L removed).

   Consider any optimal solution T to the MST with Fixed Leaves problem on input G, L. Now consider $T' = T - \{(v, u) : v \in L\}$ (T' is T with its leaves removed).

   Claim: T' is a MST in G_0.

   Proof: For a contradiction, suppose T^* is a spanning tree of G_0 and $c(T^*) < c(T')$. Then $T^* \cup \{(v, u) \in T : v \in L\}$ forms a spanning tree of the original graph G whose total cost is smaller than that of T and where each node of L is a leaf. This contradicts the fact that T is an optimal solution for the original input.

   Claim: The spanning tree generated by our algorithm is an optimal solution to the MST with Fixed Leaves problem on input G, L.

   Proof: In every optimal solution, each node of L must be connected to $V - L$ by exactly one edge. Any choice other than a minimum-cost edge would increase the cost of the resulting spanning tree and would be sub-optimal. Also, connecting one node of L to another node of L would make it impossible for both nodes to be connected to the rest of the graph and still be leaves (unless G consists of exactly one edge).

   Once these edges have been selected, and other edges to the vertices of L eliminated (to guarantee every node of L is a leaf), the remaining graph is simply G_0 and Kruskal's algorithm is guaranteed to find a MST of G_0. Thus, there is no spanning tree of G where each node of L is a leaf and with a smaller cost.

3. (a) **Greedy strategy:** Repeatedly pick the largest coin remaining.

   Counter-example: Consider input $A = 30, c_1 = 25, c_2 = 10, c_3 = 10, c_4 = 10$. The greedy algorithm would pick $c_1 = 25$ and be unable to finish making change, even though $\{c_2, c_3, c_4\}$ is a solution.

   (b) **Step 0:** Describe the recursive structure of sub-problems.

   For every optimal solution $i_1 < i_2 < \cdots < i_k$, either $i_k = m$ or $i_k < m$.

   If $i_k = m$, then $\{i_1, i_2, \ldots, i_{k-1}\}$ must be an optimal solution for input $A - c_m, c_1, \ldots, c_{m-1}$ (if there were a better solution for input $A - c_m, c_1, \ldots, c_{m-1}$, we could simply add c_m to it and get a better overall solution).

   If $i_k < m$, then $\{i_1, i_2, \ldots, i_k\}$ must be an optimal solution for input $A, c_1, \ldots, c_{m-1}$ (trivially).

   Step 1: Define an array that stores optimal values for arbitrary sub-problems.

   Define $N[k, a]$ to be the minimum number of coins required to make change for amount a using coins $c_1, \ldots, c_k$, for $0 \leqslant a \leqslant A, 0 \leqslant k \leqslant m$. (Let $N[k, a] = \infty$ when the problem has no solution for input $a, c_1, \ldots, c_k$.)

   Step 2: Give a recurrence relation for the array values. (We include justifications for each case of the recurrence.)

   $N[k, 0] = 0$ for $0 \leqslant k \leqslant m$ (no coin is necessary to make change for amount 0).

   $N[0, a] = \infty$ for $1 \leqslant a \leqslant A$ (it is impossible to make change for a positive amount without using any coins).

   $N[k, a] = N[k-1, a]$ if $c_k > a$, for $1 \leqslant k \leqslant m, 1 \leqslant a \leqslant A$ (coin c_k cannot be used to make change for amount a if $c_k > a$).

   $N[k, a] = \min\{N[k-1, a], 1 + N[k-1, a - c_k]\}$ if $c_k \leqslant a$, for $1 \leqslant k \leqslant m, 1 \leqslant a \leqslant A$ (any optimal solution either does not use c_k, or it does).

   Step 3: Write a bottom-up algorithm to compute the array values, following the recurrence.

```
# Simply fill in the array values, following the recurrence.
N[0,0] := 0
for a := 1,2,...,A:  N[0,a] := oo
for k := 1,2,...,m:
    N[k,0] := 0
    for a := 1,2,...,A:
        N[k,a] := N[k-1,a]
        if c_k <= a and 1 + N[k-1, a-c_k] < N[k,a]:
            N[k,a] := 1 + N[k-1, a-c_k]
```

   Step 4: Use the computed values to reconstruct an optimal solution.

```
# Idea: for every k,a, use coin c_k iff N[k,a] != N[k-1,a].
if N[m,A] = oo:  return {}
S = {}
a := A
for k := m,m-1,...,1:
    if N[k,a] != N[k-1,a]:
        S := S u {k}
        a := a - c_k
return S
```

   The worst-case runtime of the entire algorithm is $\Theta(mA)$, for filling in the values of $N[k, a]$ (reconstructing the solution takes only time $\Theta(m)$).