Exercise 12.1  Algorithm design I (part of an exam in February 2011).

a) First, we sort the performances in ascending order according to their ending times. We then go through the performances in this order, and for each performance, we decide whether to schedule or reject it. If the start time of the current performance is smaller than the end time of the last scheduled performance, we reject the current performance. If the start time is greater or equal than the end time of the last accepted performance, we schedule the current performance and update the end time of the last scheduled performance.

b) \textbf{PerformancePlanning}(V):
\begin{itemize}
\item \textbf{input}: set of all performances $V$
\item \textbf{output}: maximum set of mutually compatible performances
\end{itemize}

$S \leftarrow \text{sort } V \text{ by ending times}$
$t_{\text{end}} \leftarrow 0$

\textbf{FOR EACH PERFORMANCE }$(s, e) \text{ IN } S$: \begin{itemize}
\item \textbf{IF } $s \geq t_{\text{end}}$: \begin{itemize}
\item \textbf{SCHEDULE PERFORMANCE }$(s, e)$
\item $t_{\text{end}} \leftarrow e$
\end{itemize}
\end{itemize}

c) We prove that an optimal solution exists that equals the one computed by our algorithm. We prove by induction that after each step there still is an optimal solution that does not contain the rejected performances (i.e., this invariant holds). Since the solution we compute contains all the other performances, it must be optimal.

\textbf{Base step}: In the beginning, none of the performances have been rejected. The invariant trivially holds.

\textbf{Inductive hypothesis}: We assume that the invariant holds before we reject a performance.

\textbf{Inductive step}: let $(s, e)$ be the next performance that is going to be rejected by our algorithm. By the inductive hypothesis, there is an optimal solution $L$ that does not contain the previously rejected performances, except maybe $(s, e)$. If $L$ does not contain $(s, e)$, the invariant still holds. Otherwise, there is a performance $(s', t_{\text{end}})$ chosen by our algorithm, which overlaps with $(s, e)$. This performance cannot be contained in $L$. Since $L$ contains none of the other rejected performances, we can substitute $(s, e)$ in $L$ with $(s', t_{\text{end}})$ without creating any overlaps. The new solution is obviously as large as $L$ and contains none of the previously discarded performances. The invariant still holds.

The running time of the algorithm is dominated by the sorting and is $O(n \log n)$. Afterwards, the algorithm performs only $O(n)$ steps, where the cost of each step is constant.
Exercise 12.2  Algorithm design II (part of an exam in February 2011).

a) The customer’s relationships can be represented as a directed tree. The root of the tree represents the bank, and all the other nodes represent customers. An edge from a node \( u \) to a node \( v \) in the tree means that the customer \( u \) recruited the customer \( v \).

We store the nodes of the tree in an array \( A \) of length \( n+1 \) (a node for every customer and a node for the bank). Each node stores a list \( L \) of his children, i.e., is a list of the customers that it recruited. Additionally, each node stores a visit time \( t_{in} \) and an exit time \( t_{out} \). We can initialize the lists by going through all the customers once and inserting each in the list of the node that recruited it. This requires a total of \( O(|S|) = O(n) \) steps.

Now we perform a depth-first search starting from the root. We set the visit time \( t_{in} \) of every node to the number of steps until that node is reached by the search. The exit time \( t_{out} \) is set to the number of steps until the node is exited by the search. The depth-first search requires linear time, hence the initialization requires \( O(n) \) steps.

The Successor-operation can now be easily implemented: A node \( a \) is a successor of \( b \) if its visit time lies between the visit and the exit times of \( b \). The corresponding pseudocode is the following:

```
SUCCESSOR?(a,b):
  input: two nodes a, b
  output: TRUE if a is a successor of b, otherwise FALSE
  return A[b].t_{in} < A[a].t_{in} < A[b].t_{out}
```

b) Here it is sufficient to mark the entry corresponding to the customer in the array as deleted if it has no successors, and otherwise raise an error. We also have to remove the node from the list of customers recruited by its predecessor. In order to make the operation DELETE run in constant time, we have to use doubly linked lists, and every node needs a reference to its entry \( P \) in the list of children of its predecessor. The initialization changes accordingly, and in addition, each node is initially set as unmarked.

```
DELETE(a):
  input: node a
  output: error message if a has a successor, otherwise mark a as deleted
  if A[a].L ≠ ∅ or A[a] is already marked as ,,deleted“:
    raise an error message
  else:
    mark A[a] as ,,deleted“
    delete A[a].P from the list of children of the predecessor of a
```

c) Since we now want to insert and delete, we can no longer store the elements in an array. Instead, we use a balanced search tree \( B \) in which the nodes are sorted by customer number. The balanced tree can be built in linear time, since we do not have to sort the customer numbers anymore. The initialization proceeds as before, and it still requires \( O(n) \) steps, if we avoid to search for nodes in \( B \). This can be achieved, for example, by using an array that contains a link to every customer in \( B \). The array can be forgotten after the initialization.

When we want to insert a node \( a \), we first need to insert a node in the balanced tree. Then, we have to find the recruiter \( B[b] \), and insert \( a \) in the list of its children \( B[b].L \). We also have to set the visit times \( B[a].t_{in} \) and \( B[a].t_{out} \) such that the sequences of visit and exit times corresponds to a depth-first search. If \( x \) is the last node attached to the list \( B[b].L \), it is also the child with the greatest exit time, so we can set the visit and exit times of \( B[a] \) such that \( x.t_{out} < B[a].t_{in} < B[a].t_{out} < B[b].t_{out} \). In this way, we ensure that the last child always has the largest exit time. We can manage this efficiently, by keeping track of last child \( x \) of \( B[b].L \). We use the fact that we can store any rational number efficiently.

Overall, the three operations now require \( O(\log n) \) steps each, because we have to find the node that corresponds to the customer in the search tree. The insertion and deletion operations require \( O(\log n) \) steps each.
Recruited\((a, b)\):

**input:** a new customer \(a\) and an existing customer \(b\) that recruited \(a\) (or the bank)

**output:** the customer \(a\) is stored in the balanced tree \(B\)

\[
B[a] \leftarrow \text{ADD KEY } a \text{ IN } B
\]

\[
B[a].L \leftarrow \emptyset
\]

\[
B[b] \leftarrow \text{FIND } b \text{ IN } B
\]

if \(B[b].L = \emptyset\):

\[
t_{\text{last}} \leftarrow B[b].t_{\text{in}}
\]

else:

\[
t_{\text{last}} \leftarrow B[b].L.t_{\text{out}}
\]

\[
B[a].t_{\text{in}} \leftarrow t_{\text{last}} + \frac{1}{2} (B[b].t_{\text{out}} - t_{\text{last}})
\]

\[
B[a].t_{\text{out}} \leftarrow t_{\text{last}} + \frac{3}{2} (B[b].t_{\text{out}} - t_{\text{last}})
\]

APPEND \(B[a]\) TO THE END OF \(B[b].L\)