Exercise  (Scheduling activities among the fewest halls)

**Problem** Given n activities, with activity i for 1 ≤ i ≤ n described by time interval \([s_i, f_i]\), and n available halls, assign activities to halls so that

- two activities whose time intervals intersect are never assigned to the same hall, and
- the total number of halls used is minimized.

**Definitions**

- Let \( N := \{1, 2, \ldots, n\} \).
- A function \( H : A \rightarrow N \) is a partial assignment if
  - \( A \subseteq N \), and
  - for distinct \( i, j \in A \) \((H(i) = H(j) \Rightarrow [s_i, f_i] \cap [s_j, f_j] = \emptyset)\).
- A total assignment is a partial assignment with \( A = N \).
- A total assignment \( H : N \rightarrow N \) extends a partial assignment \( G : A \rightarrow N \) if \( H \) restricted to domain \( A \) agrees with \( G \).

**Remarks**

- A solution to our problem is a total assignment \( H : N \rightarrow N \) that minimizes \( |H(N)| \). (Here \( H(N) := \{H(i) : i \in N\} \).)
- Our greedy procedure repeatedly extends a partial assignment until it is total.
Exercise contd

Greedy procedure

(1) Sort the start times \( s_1, s_2, \ldots, s_n \) and finish times \( f_1, f_2, \ldots, f_n \).

(2) Merge the sorted lists of start and finish times into one sorted list of events, placing start events before finish events in case of ties. Record with each event its activity number and whether it is a start or finish event.

(3) Build a min-heap \( H \) of halls \{1, 2, \ldots, n\} prioritized by hall number. (\( H \) stores the currently available halls.)

(4) Scan the merged list of events from earliest to latest. For each event \( (e, i) \) of type \( e \) caused by activity \( i \), do the following:

(a) If \( e \) is a start event, set \( h(i) := \text{Extract}(H) \) (i.e. assign to \( i \) the lowest-numbered available hall).

(b) If \( e \) is an end event, do \( \text{Insert}(h(i), H) \) (i.e. free up hall \( h(i) \)).

(5) Return the assignment \( h \).

Analysis

Step (1) takes \( O(n \log n) \) time worst-case.
Steps (2) and (3) take \( \Theta(n) \) time.
Parts (a) and (b) of Step (4) each take \( O(\log n) \) time, so Step (4) takes \( O(n \log n) \) total time.
Thus the whole procedure takes \( O(n \log n) \) time.
Exercise cont.

Correctness

Lemma

Number the activities 1, 2, ..., n in order of increasing start time, and let H be the partial assignment of activities 1, 2, ..., i} obtained by the greedy procedure. Suppose H has an extension to an optimal total assignment. Then H together with the greedy assignment for activity i+1 has an extension to an optimal total assignment.

Proof

Let $H^*$ be an optimal total assignment that extends H, $h^*$ be $H^*(i+1)$, and $h$ be the greedy assignment for activity i+1.

If $h^* = h$, the lemma holds.

If $h^* \neq h$, change $H^*$ into $\tilde{H}$ by exchanging halls $h^*$ and $h$ as follows. Whenever $H^*(j) = h^*$ for an activity $j \in \{i+1, i+2, \ldots, n\}$, set $\tilde{H}(j) = h$. Whenever $H^*(j) = h$ for an activity $j \in \{i+1, \ldots, n\}$, set $\tilde{H}(j) = h^*$.

$\tilde{H}$ is a total assignment that extends H together with the greedy assignment of activity i+1, and it uses no more halls than $H^*$, so it is optimal.

Theorem

The greedy procedure finds an optimal assignment.

Proof

By the lemma using induction on the number of activities.
Exercise contd

Remark

- Note that the approach that repeatedly finds a maximum-cardinality non-intersecting subset of activities, assigns them to a hall, removes them from the input, and iterates, is not correct, as shown by the following counterexample:

```
1
2
3
4
```

The above approach partitions the activities into 3 halls:

\{1,3\}, \{2\}, \{4\}.

But an optimal solution partitions them into 2 halls:

\{1,4\}, \{2,3\}.
Problem

Using the fewest coins to make change.

(a) Given an amount $x$ to make into change, we wish to find nonnegative integers $q, n, d, p$ such that

$$x = 25q + 10d + 5n + p$$

and $q + d + n + p$ is minimum.

The following greedy algorithm always uses as much as possible of the next largest coin.

Greedy change (x) begin

$q := \lfloor \frac{x}{25} \rfloor$

$x := 25q$

$d := \lfloor \frac{x}{10} \rfloor$

$x := 10d$

$n := \lfloor \frac{x}{5} \rfloor$

$x := 5n$

$p := x$

return $q, d, n, p$

end
Proposition. The greedy algorithm for U.S. coins is optimal.

Proof. Let $g, d, n, p$ be the greedy solution, and $g^*, d^*, n^*, p^*$ be an optimal solution.

Certainly $q^* < g$. Suppose $q^* < g$. Then $x - 25q^* > 25$ cents is distributed among $d^*, n^*, p^*$. If in this distribution there is a subset of the coins adding up to 25¢, replacing this subset by one quarter reduces the number of coins, a contradiction. If no subset of the distribution adds up to 25¢, the distribution must use > 3 dimes (if the distribution of > 25¢ uses ≤ 2 dimes, a subset must add up to 25¢, as can be shown by case analysis.) Replacing these three dimes by a quarter and a nickle reduces the number of coins, a contradiction. So we must have $q^* = g$.

Given that $q^* = g$, we know $d^* < d$. Suppose $d^* < d$.

Then $x - 25q^* - 10d^* > 10$ cents is distributed among $n^*, p^*$. In any distribution of > 10¢ among nickles and pennies, there is a subset of the coins adding up to 10¢. (This can be proven by case analysis.) Taking this subset and replacing it by one dime reduces the number of coins, a contradiction. Hence $d^* = d$.

The same reasoning shows $n^* = n$, and hence $p^* = p$. Thus the greedy solution is optimal.
(b) Given coins in denominations $c_0, c_1, \ldots, c_k$ for integers $c_1, k \geq 1$, the greedy algorithm generalizes to the following.

Greedy change $(x, c, k, U, L)$ begin

\[ \text{for } i := k \text{ down to } 0 \text{ do begin} \]
\[ L[i] := \min \left\{ \frac{x}{c_i}, U[i] \right\} \]
\[ x := c_i L[i] \]
end

if $x > 0$ then begin
\[ \text{output } "\text{Cannot make change.}" \]
\[ \text{halt} \]
end
end

Note that a key property in the proof of correctness of the greedy algorithm for U.S. coins is that when $x \geq d$ cents, for some denomination $d$, is distributed among coins smaller than $d$, there is always a subset of the coins that adds up to exactly $d$ cents. We will prove that this property holds for coins in denominations $c_0, c_1, \ldots, c_k$, from which it follows, as before, that the greedy algorithm is correct.
Lemma. For \( c > 1, k > 1 \), let \( x \geq c^k \) cents be distributed among coins in denominations \( c^0, c^1, \ldots, c^{k-1} \). Then there is a subset of these coins that sums to exactly \( c^k \).

Proof. We proceed by induction on \( k \). Certainly the lemma holds for \( k = 1 \). So, assuming it holds for \( 1, 2, \ldots, k-1 \), we show it holds for \( k \).

Let \( x \geq c^k \) cents be distributed among coins smaller than \( c^k \). Consider how much is distributed among denomination \( c^{k-1} \), and call the amount \( d \). If \( d \geq c^{k-1} \), the lemma holds, since then \( d + c^{k-1} \geq c^k \) for some integer \( i \), which implies \( i > c \), so taking \( i \) of the coins forming \( d \) will make exactly \( c \cdot c^{k-1} = c^k \) cents.

Now suppose \( d < c^k \). Then

\[
x - d \geq c^k - d \geq c^{k-1}
\]

cents must be distributed among coins smaller than \( c^{k-1} \).

By the induction hypothesis, there is a subset of these coins summing to exactly \( c^{k-1} \).

Collect this subset. If now \( d + c^{k-1} = c^k \), we're done. Otherwise, \( d + c^{k-1} < c^k \), and the remaining

\[
x - (d + c^{k-1}) \geq c^k - (d + c^{k-1}) > c^{k-1}
\]

cents is again distributed among smaller coins; by the hypothesis, there is a subset summing to exactly \( c^{k-1} \).
Collect this subset. If now \( d + 2ck - 1 = c^k \), we're done. Otherwise, repeat the above process. Eventually, because \( x \geq c^k \), we must collect an amount from the smaller coins that is a multiple of \( c^{k-1} \) and when added to \( d \) gives \( c^k \) — at which point, we're done.

**Theorem** The greedy algorithm for coins \( c, c', \ldots, c^k \) is optimal.

**Proof** Again let \( d_0, d_1, \ldots, d_k \) be the amounts in the greedy solution, and \( d_0^*, d_1^*, \ldots, d_k^* \) be the amounts in an optimal solution.

Certainly \( d_k^* \leq d_k \). Suppose \( d_k^* < d_k \). Then

\[
\frac{x - d_k^*}{d_k} > \frac{d_k - d_k^*}{c^k}
\]

cents is distributed among coins smaller than \( c^k \). By the lemma, there is a subset of these coins summing to exactly \( c^k \) cents. Replacing the subset by one \( c^k \) coin reduces the number of coins, a contradiction. So \( d_k^* = d_k \).

Continuing with the smaller coins shows \( d_{k-1}^* = d_{k-1} \), \( \ldots, d_0^* = d_0 \). Hence the greedy solution is optimal.
Pr contd

(3) For denominations 1, 5, 10, 20, 25 (i.e., the U.S. coins with a 20¢ coin added), the greedy algorithm is not optimal. Consider making 40¢. The greedy algorithm uses 1 quarter, 1 dime, and 1 nickel, using three coins. But 40¢ can be made from two 20¢ coins. So greedy is not optimal.

Observations

1. While our proof shows the greedy algorithm is optimal for coins c_0, c_1, ..., c_k even when the number of coins available is limited, the greedy algorithm is not optimal for U.S. coins when the number of coins is limited. Consider making 30¢ from 1 quarter, 3 dimes, 0 nickels, 5 pennies. Greedy uses 1 quarter, 5 pennies, using 6 coins. Optimal is 3 dimes, using three coins. (If there were 0 pennies, the greedy algorithm would not even find a feasible solution.)

2. It is natural to search for a necessary and sufficient condition for the greedy algorithm to be optimal. There is some indication that finding a simple characterization is unlikely: the general problem of simply deciding whether there is a subset of coins adding up to a given amount of change, for an arbitrary set of coins, is NP-complete: it is equivalent to a version of 0-1 knapsack.
Exercise: Implicit heap with Insert and ExtractMin

In an implicit binary heap with $n$ elements, Insert and ExtractMin both take $O(\log n)$ time. We show that we can view Insert as taking $O(\log n)$ amortized time and ExtractMin as taking $O(1)$ amortized time.

To measure the actual time, we count the number of comparisons performed by these operations. Consider the following potential function for a heap $H$:

$$\Phi(H) := 2 \sum_{v \in H} \text{depth}(v).$$

An Insert $(k, H)$ places the new element $k$ at $A[n+1]$, increments $n$, and bubbles $A[n+1]$ up to the root as needed. Let the height of the node that $k$ ends up at be $h$. Then the number of comparisons performed is $h+1$. So the amortized time is

$$a_{\text{Insert}} := t_{\text{Insert}} + \Delta \Phi_{\text{Insert}}$$

$$= h+1 + 2 \lfloor \log n \rfloor$$

$$\leq 3 \lfloor \log n \rfloor + 1$$

$$= O(\log n).$$

An ExtractMin $(H)$ exchanges $A[n]$ with $A[i]$, decrements $n$, and bubbles $A[i]$ down with a call to Heapify. Let the depth of the node that $A[i]$ ends up at be $d$. Then the number of comparisons
Exercise cont'd

performed is at most $2(d+1)$. Thus the amortized time for an ExtractMin is

$$a_{\text{ExtractMin}} = t_{\text{ExtractMin}} + \Delta \overline{\text{ExtractMin}}$$

$$\leq 2(d+1) - 2 \lfloor \log n \rfloor$$

$$\leq 2 \lceil \log n \rceil + \log n + 2 - 2 \lfloor \log n \rfloor \quad (*)$$

$$= 2$$

$$= O(1).$$

Thus we can view Insert and ExtractMin as taking $O(\log n)$ and $O(1)$ amortized time, respectively.

(Notice why we chose the factor of 2 in the definition of $\overline{\text{ExtractMin}}$: to obtain the needed cancellation in $(*)$. )
Problem \hfill (Amortized weight-balanced trees)

Definition For a node $x$ in a search tree, let
\begin{align*}
s(x) &= \text{size of subtree rooted at } x, \\
\ell(x) &= \text{left child of } x, \\
r(x) &= \text{right child of } x.
\end{align*}

For a given constant $\alpha$, where $\frac{1}{2} < \alpha < 1$, a tree is $\alpha$-balanced if at every node $x$,
\begin{align*}
s(\ell(x)) &\leq \alpha \cdot s(x), \quad \text{and} \\
s(r(x)) &\leq \alpha \cdot s(x).
\end{align*}

Idea After an Insert or Delete in a search tree, the largest subtree that is not $\alpha$-balanced is reorganized to make it $\frac{1}{2}$-balanced. (If we start with an empty tree and follow this rule, the tree is always $\alpha$-balanced.)

(a) Proposition An arbitrary $n$-node subtree can be made $\frac{1}{2}$-balanced in $\Theta(n)$ time using $\Theta(n)$ space.

Proof We first show that a tree $T$ is $\frac{1}{2}$-balanced iff, at every node $x \in T$,
\begin{equation}
|s(\ell(x)) - s(r(x))| \leq 1. \tag{*}
\end{equation}

Then, to make an arbitrary tree $\frac{1}{2}$-balanced, we simply construct a tree over the same node set that meets condition (*).
Problem contd.
(a) contd.

Proposition: At every node in a $\frac{1}{2}$-balanced tree, inequality (*) holds.

Proof: For an arbitrary node $x$ in a $\frac{1}{2}$-balanced tree, let $L := s(l(x))$ and $R := s(r(x))$. Then by definition,

\[
\begin{align*}
&\begin{cases}
    L \leq \frac{1}{2} (1 + L + R) \\
    R \leq \frac{1}{2} (1 + L + R)
\end{cases} \\
\implies &\begin{cases}
    \frac{1}{2} L \leq \frac{1}{2} (1 + R) \\
    \frac{1}{2} R \leq \frac{1}{2} (1 + L)
\end{cases} \\
\implies &\begin{cases}
    L \leq 1 + R \\
    R \leq 1 + L
\end{cases} \\
\implies &\begin{cases}
    L - R \leq 1 \\
    R - L \leq 1
\end{cases} \\
\implies |L - R| \leq 1.
\]

Thus the following $\frac{1}{2}$-balancing procedure is correct.
Problem: cont$^d$

(a) cont $^d$

function Make Half Balanced $(x)$ begin

\begin{align*}
n &:= \text{Size}[x] & \text{Returns a \( \frac{1}{2} \)-balanced subtree} \\
\text{Traverse the subtree} & \quad \text{rooted at} \ x. \ \text{Uses an} \\
\text{rooted at} \ x, \ \text{storing} & \quad \text{auxiliary array} \ A[1:n]. \\
\text{its nodes in symmetric order} & \quad \text{in} \ A[1:n].
\end{align*}

return Make Half Balanced Helper $(A, 1, n)$
end

function Make Half Balanced Helper $(A, i, j)$ begin

if $i < j$ then begin

\begin{align*}
m &:= \left\lceil \frac{i+j}{2} \right\rceil \\
x &:= A[m] \\
\text{Size} [x] &:= j - i + 1 \\
\text{Left} [x] &:= \text{Make Half Balanced Helper} (A, i, m-1) \\
\text{Right} [x] &:= \text{Make Half Balanced Helper} (A, m+1, j)
\end{align*}

return $x$
end else

return Nil
end

The space is clearly $\Theta(n)$. The time for the recursive
helper function is $T(n) = 2T(\frac{n}{2}) + \Theta(1) = \Theta(n)$.
So the total time to construct the tree is $\Theta(n)$. \[\]

\[\]
(b) **Proposition** A Find in an $n$-node $\alpha$-balanced tree takes $O(\log n)$ time.

**Proof** The worst-case time is

$$T(n) = \Theta(1) + T(\alpha n)$$

$$= \Theta(\log \alpha^{-1} \log n) \quad \text{by Master Thm}.$$  

$$= \Theta(\log n).$$  

(c) **Proposition** The following potential function,

$$\Phi(T) := c \sum_{x \in T} \Delta(x),$$

where $c > 0$ and

$$\Delta(x) := |s(l(x)) - s(r(x))|,$$

is always positive, and has value $0$ on a $1/2$-balanced tree.

**Proof** Since $\Delta(x) > 0$ for all $x$, $\Phi(T) > 0$ for all $T$.

In a $1/2$-balanced tree, $\Delta(x) \leq 1$ for all $x \in T$ by Part (a). Thus the sum over all nodes $x \in T$ with $\Delta(x) > 1$ is empty, so $\Phi(T) = 0$ by definition.
(d) **Proposition**  The amortized time to rebalance a subtree
is $O(1)$ for $c \geq \frac{1}{2x-1}$.

**Proof**  We take the actual time to rebalance an
$m$-node subtree $T$ rooted at $x$ to be $t = m$, following
Part (a). To bound the change in potential, we
separately consider an Insert and a Delete. Before
the operation, $T$ is $\alpha$-balanced, but afterwards,
it is not. For $T$ before the operation, let
$L := s(l(x)), R := s(r(x))$, and suppose w.l.o.g.
that $L > R$ (since if $L = R$, node $x$ must be
$\frac{1}{2}$-balanced), and that on an Insert, the node is
added to the subtree rooted at $l(x)$. Then

$$\begin{cases}
L \leq \alpha m, \\
L+1 \notin \alpha (m+1)
\end{cases}$$

\[ \Rightarrow \begin{cases}
L \leq \alpha m \\
L > \alpha (m+1) - 1
\end{cases} \]

\[ \Rightarrow \begin{cases}
L \leq \lfloor \alpha m \rfloor \\
L \geq \lceil \alpha (m+1) \rceil
\end{cases} \]

\[ \Rightarrow \lfloor \alpha m \rfloor \leq \lceil \alpha (m+1) \rceil \leq L \leq \lfloor \alpha m \rfloor 
\]

\[ \Rightarrow L = \lfloor \alpha m \rfloor . \]

Similarly, suppose w.l.o.g. that $L > R$ and that
on a Delete, the node is removed from the subtree
rooted at $r(x)$.
Then

\[
\begin{cases}
L \leq \alpha m, \\
L \neq \alpha (m-1)
\end{cases}
\]

\[\Rightarrow L = \lfloor \alpha m \rfloor \quad \text{(reasoning as before)}\]

So for both an insert and a delete,

\[
\begin{align*}
|L-R| &= L-R \\
&= L - (m-L-1) \\
&= 2L - m + 1 \\
&= 2\lfloor \alpha m \rfloor - m + 1 \\
&\geq 2(\alpha m - 1) - m + 1 \\
&= (2\alpha - 1)m - 1.
\end{align*}
\]

Thus the amortized time to rebalance at \( x \) is

\[
\begin{align*}
t + E(T'(x)) - E(T(x)) &= m - E(T(x)) \\
&\leq m - c \Delta(x) \\
&= m - c |L-R| \\
&\leq m - c (2\alpha - 1)m - 1,
\end{align*}
\]

which is in turn upper bounded by the constant \( c = O(1) \)

when
Problem cont'd
(d) cont'd

Proof cont'd

\[ m - c((2\alpha - 1)m - 1) \leq c \]
\[
(1 - c(2\alpha - 1))m \leq 0 \\
1 - c(2\alpha - 1) \leq 0 \quad \text{since } m \geq 1 \\
\frac{c}{2\alpha - 1} \\
\]

Thus for this choice of \( c \), the amortized time to rebalance a subtree is \( O(1) \).

\[ \square \]

(e) Proposition An Insert or Delete on an \( \alpha \)-balanced tree of \( n \) nodes takes \( O(\log n) \) amortized time.

Proof We divide the amortized time for an Insert or Delete into search time and rebalancing time. The actual search time is \( O(\log n) \) by Part (b).

For the rebalancing time, there are two cases.

Case 1: Both before and after the Insert or Delete, \( T \) is \( \alpha \)-balanced.

In this case the actual rebalancing time is 0.

Let \( P \) be the path in \( T \) from the inserted or deleted node to the root. The change in potential is

\[ \Delta(T') - \Delta(T) = c \sum_{x \in P, \Delta(x') > 1} \Delta(x') - c \sum_{x \in P, \Delta(x) > 1} \Delta(x) \]

\[ \leq 2c |P|, \]
which is $O(\log n)$, as $|P| = O(\log n)$ by Part (b).
Thus the amortized rebalancing time is $O(\log n)$.

Case 2: Before the Insert or Delete, $T$ is $\alpha$-balanced, but afterwards it is not.

In this case the amortized time is $O(1)$ by Part (d), considering just the change in potential in the subtree rooted at the highest non-$\alpha$-balanced node.

Let $P$ be the path in $T$ from this node, $x$, to the root. Then the additional change in potential outside $T(x)$ is

$$c \sum_{x \in P, \Delta(x) > 1} \Delta(x) - c \sum_{x \in P, \Delta(x) > 1} \Delta(x) = O(\log n).$$

So the amortized rebalancing time is again $O(\log n)$.

Thus the total amortized time for an Insert or Delete is $O(\log n) + O(\log n) = O(\log n)$.  □