Exercise Correctness of greedy algorithm for continuous knapsack

Lemma Number the items $1, \ldots, n$ so that
\[
\frac{v_1}{w_1} > \ldots > \frac{v_n}{w_n},
\]
and suppose fractions $f_1, \ldots, f_n$ for $0 \leq i \leq n$, are
the fractions for items $1, \ldots, i$ in an optimal continuous
knapsack of capacity $W$. Then fractions $f_i, \ldots, f_i, f_{i+1}$,
where
\[
f_{i+1} := \min \left\{ \frac{w_{i+1}}{W} \left( W - \sum_{1 \leq j \leq i} f_j w_j \right) / w_{i+1} \right\},
\]
are the fractions for items $1, \ldots, i+1$ in an optimal knapsack.

Proof Let $K^*$ be an optimal knapsack using fractions
\[
f_i, \ldots, f_i, f_{i+1}, f_{i+2}, \ldots, f_n
\]
on items $1, \ldots, n$, and let $f_{i+1}$ be defined as in the lemma.
Since $K^*$ is optimal, there is a smallest index $j$ in
range $i+1, \ldots, n$ such that
\[
\sum_{i \leq k < j} f_k w_k \leq f_{i+1} w_{i+1}.
\]
Consider the knapsack $\tilde{K}$ derived from $K^*$ that has fractions
\[
f_i, \ldots, f_i, f_{i+1}, \tilde{f}_{i+2}, \ldots, \tilde{f}_j, f_{j+1}, \ldots, f_n,
\]
where $\tilde{f}_{i+2} = 0, \ldots, \tilde{f}_{j-1} = 0$, and
\[
\tilde{f}_j := \left( \sum_{i \leq k < j} f_k w_k - f_{i+1} w_{i+1} \right) / w_j.
\]
Knapsack $\tilde{K}$ has the same total weight as $K^*$, so it is feasible,
and since $f_{i+1} = \ldots = f_j$, it has total value at least as great as $K^*$,
so it is also optimal. □
Exercise \hspace{1cm} (Scheduling activities among the fewest halls)

Problem Given $n$ activities, with activity $i$ for $1 \leq i \leq n$ 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
  - $\forall$ distinct $i, j \in A \left( H(i) = H(j) \Rightarrow [s_i, f_i] \cap [s_j, f_j] = \emptyset \right)$.
- 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 cont'd

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'd

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 \( j \in \{i+1, i+2, ..., n\} \), set \( \tilde{H}(j) := h \). Whenever \( H^*(j) = h \) for an activity \( j \in \{i+1, ..., n\} \), 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 cont'd

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:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The above approach partitions the activities into 3 halls:
\[ \{1,3\}, \{2\}, \{4\} \].
But an optimal solution partitions them into 2 halls:
\[ \{1,4\}, \{2,3\} \].
Suppose we have a collection of \( n \) tasks that must be performed. For each task \( i \) we know \( t_i \), the length of time it takes to perform task \( i \). We can perform a task at any point in time that we choose, and we can perform them in any order, but we can only perform one task at a given moment.

The completion time of a task is the time at which we finish performing it. Design an efficient greedy algorithm that finds a sequence in which to perform the tasks that minimizes the average completion time for the \( n \) tasks. More formally, if \( c_i \) is the completion time of task \( i \) for a given sequence, the solution value for that sequence is

\[
\frac{1}{n} \sum_{1 \leq i \leq n} c_i.
\]

Analyze the running time of your algorithm, and prove that it finds an optimal solution using a greedy augmentation lemma of the type given in class.

**Algorithm**

We sort the tasks by increasing running time. 

Rename the sorted tasks so that 

\[ t_1 \leq t_2 \leq \cdots \leq t_n. \]

We then execute the tasks in this order \( 1, 2, \ldots, n \), starting them at times 

\[ 0, t_1, t_1+t_2, \ldots, t_1+t_2+\cdots+t_n. \]

(Equivalently, this greedy procedure executes next the task \( i \) that has the smallest \( t_i \) of all tasks not yet executed.)

**Analysis**

Sorting the tasks and determining their start times takes a total of \( O(n \log n) \) time for \( n \) tasks.

**Correctness**

Let a partial solution be a prefix of the listing of tasks in their order of execution.

A partial solution is contained in a complete solution if it is a prefix of the complete solution.
Correctness, cont.

**Lemma**

Suppose tasks 1, 2, ..., i form a partial solution contained in an optimal solution.

Let task i+1 be the next task executed by the greedy procedure.

Then partial solution 1, 2, ..., i, i+1 is contained in an optimal solution.

**Proof**

Let S* be an optimal solution that contains the partial solution 1, 2, ..., i.

If the next task S* executes is i+1, the lemma holds.

Suppose instead S* executes next task j > i+1.

Let k be the position in the ordering at which S* executes task i+1.

Form a new solution S by exchanging the positions of tasks i+1 and j, as follows.

\[
\begin{array}{cccccccc}
  S^* & 1 & 2 & \cdots & i & i+1 & k & \cdots \\
  \text{S} & 1 & 2 & \cdots & i & j & \cdots & i+1 \\
\end{array}
\]

Notice that the average completion time c(S) of a schedule S is,
Proof, cont'd.

\[ c(S) = \frac{1}{n} \sum_{i \in U} \sum_{j \in i} t S_{ij} \]

\[ = \frac{1}{n} \sum_{i \in U} (n - i + 1) t S_{i(i)} . \]

Since schedules \( S^* \) and \( \tilde{S} \) only differ at positions \( i+1 \) and \( k \),

\[ c(S^*) - c(\tilde{S}) = \frac{1}{n} \left( (n-i) t_j + (n-k+1) t_{i+1} \right) \]

\[ - \left( (n-i) t_{i+1} + (n-k+1) t_j \right) \]

\[ = \frac{1}{n} \left( (k - (i+1)) t_j - (k - (i+1)) t_{i+1} \right) \]

\[ = \frac{1}{n} \left( k - (i+1) \right) (t_j - t_{i+1}) \]

\[ \geq 0, \quad \text{since } t_i \leq \ldots \leq t_n \]

which implies \( c(\tilde{S}) \leq c(S^*) \).

Thus \( \tilde{S} \) is an optimal solution that contains the partial solution \( 1, 2, \ldots, i+1 \).

\[ \square \]

**Theorem**  The greedy procedure finds an optimal schedule.

**Proof**  By the lemma, using induction on the number of iterations.

\[ \square \]
Exercise

Simulating a queue with two stacks

To simulate the operations Create, Put, and Get on a queue $Q$, we use two stacks, Front[$Q$] and Rear[$Q$], as follows:

```
<table>
<thead>
<tr>
<th>Front[$Q$]</th>
<th>Rear[$Q$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...00</td>
<td>00...0</td>
</tr>
</tbody>
</table>
```

first element -> last element

Our implementation is below:

```plaintext
function Create() begin
    Q := Memory();
    Front[$Q$] := Stack();
    Rear[$Q$] := Stack();
    return Q
end

procedure Put(x, Q) begin
    Push(x, Rear[$Q$]);
end

function Get(Q) begin
    if Empty(Front[$Q$]) then
        while not Empty(Rear[$Q$]) do
            Push(Pop(Rear[$Q$]), Front[$Q$]);
        return Pop(Front[$Q$]);
    end
end
```
Exercise cont'd

For the analysis, let us measure the actual time by the number of Pushes and Pops, and take as our potential function

$$\Phi(Q) := 2 \cdot \text{Size}(\text{Rear}[Q]).$$

Then

$$a_{\text{Put}} := t_{\text{Put}} + \Delta \Phi_{\text{Put}}$$
$$= 1 + 2$$
$$= O(1).$$

Suppose $x$ elements are moved from Rear $[Q]$ to Front $[Q]$ by a Get. Then

$$a_{\text{Get}} := t_{\text{Get}} + \Delta \Phi_{\text{Get}}$$
$$= (1 + 2x) - 2x$$
$$= O(1).$$

Thus we can simulate a queue with two stacks in $O(1)$ amortized time.
Problem (Deleting the larger half)

Implement the following operations on a set $S$ of numbers,

- Insert $(x, S) \rightarrow$ Add $x$ to $S$;
- Delete Larger Half $(S) \rightarrow$ Delete the largest $\frac{|S|+1}{2}$ elements from $S$;

so both operations take $O(1)$ amortized time.

Solution Sketch

We implement these operations as follows:

- Insert $(x, S) \rightarrow$ Put $x$ onto a singly-linked, unordered list $L$.
- Delete Larger Half $(S) \rightarrow$ Compute the median element $x$ of $S$.

Every element $y$ in $S$, with $y \geq x$, delete from $L$.

For our amortized analysis we use the charging method.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual time</th>
<th>Amortized time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Delete Larger Half</td>
<td>$2n$</td>
<td>0</td>
</tr>
</tbody>
</table>

We take the actual time for Delete Larger Half, which (i) computes the median and then (ii) does a scan to delete elements, to be $2n$.

An Insert takes 1 unit of actual time, but receives 5 units of amortized time; we store the 4 units of credit on the inserted element.
Solution out:

For DeleteLargerHalf, let us assume every element has 4 units of credit on it before the operation.
To execute DeleteLargerHalf, we use 2 units from every element. (So now every element has 2 units remaining on it.)
Take the remaining 2 units from every deleted element and place those units on the elements not deleted. Now every element left in S has 4 units of credit again (as there are at least as many elements deleted as not deleted). So the credit assumption is maintained.
Problem (Amortized weight-balanced trees)

Definition For a node \( x \) in a search tree, let

\[
\begin{align*}
    s(x) &:= \text{size of subtree rooted at } x, \\
    l(x) &:= \text{left child of } x, \\
    r(x) &:= \text{right child of } x.
\end{align*}
\]

For a given constant \( \alpha \), where \( \frac{1}{2} < \alpha < 1 \), a tree is balanced if at every node \( x \),

\[
s(l(x)) \leq \alpha s(x), \quad \text{and} \quad s(r(x)) \leq \alpha s(x). \]

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 if, at every node \( x \in T \),

\[
| s(l(x)) - s(r(x)) | \leq 1. \tag{*}
\]

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

(a) cont.

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,

\[
\begin{cases}
L &\leq \frac{1}{2} (1 + L + R) \\
R &\leq \frac{1}{2} (1 + L + R)
\end{cases}
\]

\[
\Leftrightarrow \begin{cases}
\frac{1}{2} L &\leq \frac{1}{2} (1 + R) \\
\frac{1}{2} R &\leq \frac{1}{2} (1 + L)
\end{cases}
\]

\[
\Leftrightarrow \begin{cases}
L &\leq 1 + R \\
R &\leq 1 + L
\end{cases}
\]

\[
\Leftrightarrow \begin{cases}
L - R &\leq 1 \\
R - L &\leq 1
\end{cases}
\]

\[
\Leftrightarrow |L - R| \leq 1.
\]

Thus the following \( \frac{1}{2} \)-balancing procedure is correct.
function Make Half Balanced \((x)\) begin

\(n := \text{Size}[x]\) . Returns a \(\frac{1}{2}\)-balanced subtree root at \(x\). Uses an auxiliary array \(A[1:n]\).

Traverse the subtree root at \(x\), storing its nodes in symmetric order in \(A[1:n]\).

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

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

if \(i \leq j\) then begin

\(m := \left\lfloor \frac{i + j}{2} \right\rfloor\)

\(x := A[m]\)

\(\text{Size}[x] := j - i + 1\)

\(\text{Left}[x] := \text{Make Half Balanced Helper} (A, i, m-1)\)

\(\text{Right}[x] := \text{Make Half Balanced Helper} (A, m+1, j)\)

return \(x\)

end else

return Nil

end

The space is clearly \(\Theta(n)\). The time for the recursive helper function is \(T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)\). So the total time to construct the tree is \(\Theta(n)\). \(\square\)
Problem contd

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

**Proof** The worst-case time is

\[
T(n) = \Theta(1) + T(\alpha n)
= \Theta(n \log \alpha^{-1} \log n) \quad \text{by Master Thm}
= \Theta(\log n).
\]

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

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

where \( c > 0 \) and

\[
\Delta(x) := \left| s(l(x)) - s(r(x)) \right|
\]

is always positive, and has value 0 on a \( \frac{1}{2} \)-balanced tree.

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

In a \( \frac{1}{2} \)-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

$$\begin{align*}
L &\leq \alpha m, \text{ but } \\
L + 1 &\neq \alpha (m+1) \\
\Rightarrow &\quad \begin{cases}
L &\leq \alpha m \\
L &> \alpha (m+1) - 1
\end{cases} \\
\Rightarrow &\quad \begin{cases}
L &\leq \lceil \alpha m \rceil \\
L &\geq \lfloor \alpha (m+1) \rfloor
\end{cases} \\
\Rightarrow &\quad \lceil \alpha m \rceil \leq \lfloor \alpha (m+1) \rfloor \leq L \leq \lceil \alpha m \rceil \\
\Rightarrow &\quad L = \lfloor \alpha m \rfloor.
\end{align*}$$

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

(d) cont.

Proof cont.

Then

$$\begin{align*}
L &\leq \alpha m, \text{ but} \\
L &\neq \alpha (m-1)
\end{align*}$$

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

So for both an insertion and a delete,

$$\left| L-R \right| = L-R$$

$$= L - (m - L - 1) \quad \text{since } m = 1 + L + R$$

$$= 2L - m + 1$$

$$= 2 \lfloor \alpha m \rfloor - m + 1$$

$$\geq 2 (\alpha m - 1) - m + 1$$

$$= (2\alpha - 1) m - 1.$$  

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

$$t + \Phi(T'(x)) - \Phi(T(x)) = m - \Phi(T(x)) \quad \text{since } \Phi(T(x)) = c \quad \text{by Part (c)}$$

$$\leq m - c \Delta(x) \quad \text{since }$$

$$\Phi(T(x)) \geq c \Delta(x)$$

$$= m - c \left| L-R \right|$$

$$\leq m - c \left( (2\alpha - 1) m - 1 \right), \quad \text{by the above}$$

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

Proof cont'd

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

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

(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

\[ \Delta (T') - \Delta (T) = c \sum_{x \in P} \Delta (x') - c \sum_{x \in P} \Delta (x) \]
\[ \Delta (x') > 1 \quad \Delta (x) > 1 \]
\[ \leq 2c |P| , \]
Problem cont'd

e) cont'd

Proof cont'd

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

$$c \sum_{x \in P, \Delta(x) > 1} \Delta(x) = O(\log n) \quad \text{(as above)}$$

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

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