Exercise

Correctness of greedy algorithm for continuous knapsack

Lemma Number the items $1, \ldots, n$ so that

$$\frac{v_1}{w_1} \geq \cdots \geq \frac{v_n}{w_n},$$

and suppose fractions $f_1, \ldots, f_i$, for $0 \leq i < n$, are the fractions for items $1, \ldots, i$ in an optimal continuous knapsack of capacity W. Then fractions $f_1, \ldots, f_i, f_{i+1}$, where

$$f_{i+1} := \min\left\{ w_{i+1},\ W - \sum_{1 \leq j \leq i} f_j w_j \right\} \Big/ w_{i+1},$$

are the fractions for items $1, \ldots, i+1$ in an optimal knapsack.

Proof Let K^* be an optimal knapsack using fractions

$$f_1, \ldots, f_i,\ f^*_{i+1},\ f^*_{i+2},\ \ldots,\ f^*_n$$

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

$$\sum_{i+1 \leq k \leq j} f^*_k w_k \geq f_{i+1} w_{i+1}.$$

Consider the knapsack $\tilde{K}$ derived from K^* that has fractions

$$\underbrace{f_1, \ldots, f_i}_{\text{same as } K^*},\ \underbrace{f_{i+1}}_{\text{greedy}},\ \underbrace{\tilde{f}_{i+2}, \ldots, \tilde{f}_j}_{\text{new}},\ \underbrace{f^*_{j+1}, \ldots, f^*_n}_{\text{same as } K^*},$$

where $\tilde{f}_{i+2} := 0, \ldots, \tilde{f}_{j-1} := 0$, and

$$\tilde{f}_j := \left(\sum_{i+1 \leq k \leq j} f^*_k w_k - f_{i+1} w_{i+1} \right) \Big/ w_j.$$

Knapsack $\tilde{K}$ has the same total weight as K^*, so it is feasible, and since $\frac{v_{i+1}}{w_{i+1}} \geq \cdots \geq \frac{v_j}{w_j}$, it has total value at least as great as K^*, so it is also optimal. $\square$

(Greedy algorithms) 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

$$\frac{1}{n} \sum_{1 \le i \le n} c_i.$$

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

$$t_1 \le t_2 \le \cdots \le t_n .$$

We then execute the tasks in this order $1, 2, \cdots, n$, starting them at times

$$0,\ t_1,\ t_1+t_2,\ \cdots,\ \sum_{1 \le i < n} t_i .$$

(Equivalently, this greedy procedure executes next that 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 <u>partial solution</u> be a prefix of the listing of tasks in their order of execution.

A partial solution is <u>contained</u> in a complete solution if it is a prefix of the complete solution.

Correctness, cont'd.

Lemma

Suppose tasks $1, 2, \cdots, i$ 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 $1, 2, \cdots, i, i+1$ is contained in an optimal solution.

Proof

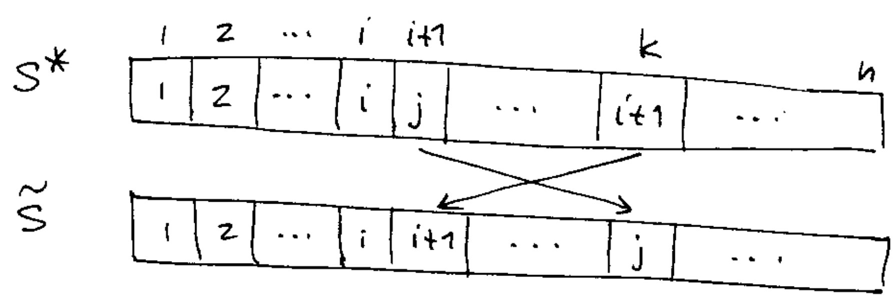
Let S^* be an optimal solution that contains the partial solution $1, 2, \cdots, i$.

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 $\tilde{S}$ by exchanging the positions of tasks $i+1$ and j, as follows.



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

$$c(S) = \frac{1}{n} \sum_{1 \le i \le n} \sum_{1 \le j \le i} t_{S[j]}$$
$$= \frac{1}{n} \sum_{1 \le i \le n} (n-i+1)\, t_{S[i]} .$$

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

$$c(S^*) - c(\tilde{S}) = \frac{1}{n}\Big(\big((n-i)\,t_j + (n-k+1)\,t_{i+1}\big) - \big((n-i)\,t_{i+1} + (n-k+1)\,t_j\big)\Big)$$
$$= \frac{1}{n}\Big((k-(i+1))\,t_j - (k-(i+1))\,t_{i+1}\Big)$$
$$= \frac{1}{n}\underbrace{\big(k-(i+1)\big)}_{>0}\underbrace{\big(t_j - t_{i+1}\big)}_{\ge 0 \text{ since } t_1 \le \cdots \le t_n}$$
$$\ge 0,$$

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

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

Theorem The greedy procedure finds an optimal schedule.

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

□

Problem (Deleting the larger half)

Implement the following operations on a set S of numbers,

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

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

Solution sketch

We implement these operations as follows:

- $Insert(x, S)$: Put x onto a singly-linked, unordered list L.
- $Delete\ Larger\ Half(S)$: 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:

Operation	Actual time	Amortized time
Insert	1	5
Delete Larger Half	$2n$	0

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 cont:

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 — Variation on a sorted array to support both binary search and insert

A sorted array of n elements allows us to find any given element in $O(\log n)$ time using binary search, but inserting a new element can take as much as $O(n)$ time. In this variation, instead of maintaining a sorted array $A[1..n]$ we maintain $k = \lceil \lg(n+1) \rceil$ arrays $A_1, A_2, \dots, A_k$ where $|A_i| = 2^{i-1}$. Array A_i is full iff bit $i-1$ is 1 in the binary representation of n; otherwise A_i is empty. Further, each A_i is sorted, but we do not enforce any relation between elements from different A_i.

(a) We can implement the Find operation on this structure as follows. The operation returns the location of the elements if found (given by the index of the list and the element's position within the list), and $(0,0)$ otherwise.

```
function Find(x) begin
    for i := 1 to k do begin
        Use binary search on A_i.
        if x is found at index j in A_i then
            return (i,j)
    end
    return (0,0)
end
```

In the worst case, when each A_i is full and all are searched, this takes time

$$\sum_{1 \le i \le k} \Theta(\log 2^i) = \sum_{1 \le i \le k} \Theta(i) = \Theta(k^2) = \Theta(\log^2 n).$$

Problem contd

(b) We can insert an element into this structure as follows. The procedure is similar to incrementing a binary counter.

procedure Insert (x) **begin**

Find the longest prefix $A_1, A_2, \cdots, A_i$ of full lists.
Merge these lists in sequence using an auxiliary array $B[1..n]$ to hold intermediate results.
Merge x into the final result in B.
Copy B into A_{i+1}, and mark lists $A_1, \cdots, A_i$ as empty

end

The time is dominated by the time to perform the i-1 merges. The time to merge the j^{th} list A_j with the merge of the preceding lists is linear in 2^{j-1}, the length of A_j, plus the total length of all preceding lists $A_1, \cdots, A_{j-1}$. This gives a total time of

$$\theta\left(\sum_{1<j\le i}\left(2^{j-1} + \sum_{1\le k\le j-1} 2^{k-1}\right)\right) = \theta\left(\sum_{1<j\le i}\left(2^{j-1}+2^{j-1}-1\right)\right)$$

$$= \theta\left(\sum_{0\le j\le i} 2^j\right)$$

$$= \theta(2^i).$$

In the worst case, $i = k = \theta(\log n)$, so the time for Insert is $\theta(2^{\log n}) = \theta(n)$ worst-case.

We determine the amortized time for an Insert using averaging. The analysis is similar to that for incrementing a binary counter. Consider a series of m Inserts. The time for a merge that spills list A_i into A_{i+1} is $\theta(2^i)$, but this only happens $\lfloor m/2^i \rfloor$ times during the series. Thus the total time for the m Inserts is

$$\sum_{1\le i\le k} \theta(2^i)\cdot\left\lfloor\frac{m}{2^i}\right\rfloor = \sum_{1\le i\le k}\theta\left(2^i\,\frac{m}{2^i}\right) = \sum_{1\le i\le k}\theta(m) = \theta(mk).$$

By averaging, the amortized time for an Insert is

Problem cont'd

(b) (cont'd)

$$\frac{1}{m}\,\Theta(mk) = \Theta(k) = \Theta(\log n).$$

It may be tempting to try keeping the elements in sorted order across the lists so that $A_1 \prec A_2 \prec \cdots \prec A_k$. Interestingly, this only makes matters worse: when the number of elements in binary is $100\cdots01$, for instance, we may be forced to insert into A_k instead of A_1, which can cause the whole structure to be reorganized, ruining the amortized time bound.

(c) There is no satisfactory way to implement Delete on this data structure. The reorganization of the lists caused by removing an element is similar to the effect of a decrement on a binary counter. To decrement a binary counter, we

- find the leftmost 1,
- change the 1 to a 0, and
- set all preceding 0's to 1's.

Our implementation of Delete is similar.

procedure Delete (x) **begin**

1 Find the list A_j containing element x.

2 Let A_{i+1} be the first nonempty list.
Choose an arbitrary element y from A_{i+1}.

3 Redistribute the elements of A_{i+1} other than y among lists $A_1, A_2, \ldots, A_j$.

4 Remove element x from A_j.
Insert y into A_j.

end

(The description above is for the general case in which $A_j \neq A_{i+1}$; the case in which $A_j = A_{i+1}$ is slightly simpler.)

Problem contd

(c, contd)

The time for step 1 is

$$\sum_{1 \le k \le j} \Theta(\log 2^k) = \sum_{1 \le k \le j} \Theta(k) = \Theta(j^2).$$

The time for step 2 is $\Theta(i)$. The time for step 3 is $\Theta(2^i)$. Finally, the time for step 4 is $\Theta(2^j)$. As $i \le j$, the total time is $\Theta(2^j)$.

In the worst case, $j = k = \Theta(\log n)$, so this is

$$\Theta(2^{\log n}) = \Theta(n)$$

time worst-case. Moreover, the presence of Delete destroys the amortized time bound on Insert, so we can place no better time bound for an Insert or Delete than $O(n)$. (For an argument justifying this, see the solution to Exercise 18.1-2.)