CSC 364S Notes **University of Toronto, Spring, 2004**

Greedy Algorithms

Simple Knapsack Problem

"Greedy Algorithms" form an important class of algorithmic techniques. We illustrate the idea by applying it to a simplified version of the "Knapsack Problem". Informally, the problem is that we have a knapsack that can only hold weight C, and we have a bunch of items that we wish to put in the knapsack; each item has a specified weight, and the total weight of all the items exceeds C; we want to put items in the knapsack so as to come as close as possible to weight C, without going over. More formally, we can express the problem as follows.

Let $w_1, \ldots, w_d \in \mathbb{N}$ be weights, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \ldots, d\}$ let $K(S) = \sum_{i \in S} w_i$. (Note that $K(\emptyset) = 0$.)
Find:

$$M = \max_{S \subseteq \{1,\ldots,d\}} \{K(S) | K(S) \leq C\}$$

For large values of d, brute force search is not feasible because there are 2^d subsets of $\{1, \ldots, d\}$.

We can estimate M using the Greedy method:
We first sort the weights in decreasing (or rather nonincreasing order)

$$w_1 \geq w_2 \geq ... \geq w_d$$

We then try the weights one at a time, adding each if there is room. Call this resulting estimate $\overline{M}$.

It is easy to find examples for which this greedy algorithm does not give the optimal solution; for example weights $\{501, 500, 500\}$ with $C = 1000$. Then $\overline{M} = 501$ but $M = 1000$. However, this is just about the worst case:

Lemma 1 $\overline{M} \geq \frac{1}{2} M$

Proof:
We first show that if $\overline{M} \neq M$, then $\overline{M} > \frac{1}{2} C$; this is left as an exercise. Since $C \geq M$, the Lemma follows. □

The notion of a **polynomial-time** algorithm is basic to complexity theory, and to this course (see definition below). Roughly speaking, we regard an algorithm as *feasible* (or *tractable*) if and only if it runs in polynomial time. The above greedy algorithm runs in polynomial time (see below) and is feasible to execute for values of d in the thousands or even millions. On the other hand, the blind search algorithm takes more than 2^d steps, is not polynomial-time, and will never run on any physical computer (now or in the future) for values of d as small as 200 – the universe will expire first.

Unfortunately the greedy algorithm does not necessarily yield an optimal solution. This brings up the question: is there any polynomial-time algorithm that is guaranteed to find an optimal solution to the simple knapsack problem? This question will be studied in the next part of the course. The answer is that this knapsack problem is "NP hard" (assuming that the weights are given using binary or decimal notation), and hence is very unlikely to be solvable by a polynomial-time algorithm.

Definition 1 *An algorithm is* polynomial-time *iff there exists a k such that the running time $T(n)$ of the algorithm satisfies $T(n) \in O(n^k)$, where $n =$ input "size".*

The notation "$T(n) \in O(f(n))$" is defined in CLR (that is, Cormen, Leiserson, Rivest) in Section 2.1, and in the more recent version of the text (CLRS) in Section 3.1 . It means that for some positive constants c and n_0, $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Instead of "$T(n) \in O(f(n))$", we sometimes say "$T(n)$ is $O(f(n))$", or "$T(n) = O(f(n))$".

The running time (i.e. $T(d)$) for the knapsack problem with the above greedy algorithm is $O(d \log d)$, because first we sort the weights, and then go at most d times through a loop to determine if each weight can be added. So this particular greedy algorithm is a polynomial-time algorithm.

Lemma 2 *For any constant $c > 0$ and positive integer k, $n^k \in O(2^{cn})$ but $2^{cn} \notin O(n^k)$.*

Proof:
It is sufficient to show that $\lim_{n\to\infty} \frac{2^{cn}}{n^k} = \infty$. This can be proved using L'Hospital's rule. The derivative of 2^{cx} with respect to x is $(c \log_e 2)2^{cx}$. So if we differentiate any number of times, the resulting function still approaches ∞ as x approaches ∞. On the other hand, if we differentiate x^k just k times, the result is the constant $k!$. $\square$

Minimum Spanning Trees

An *undirected graph* G is a pair (V, E); V is a set (of vertices or nodes); E is a set of (undirected) edges, where an edge is a set consisting of exactly two (distinct) vertices. For convenience, we will sometimes denote the edge between u and v by $[u, v]$, rather than by $\{u, v\}$.
The *degree* of a vertex v is the number of edges touching v. A *path* in G between v_1 and v_k is a sequence $v_1, v_2, \ldots, v_k$ such that each $\{v_i, v_{i+1}\} \in E$. G is *connected* if between every pair of distinct nodes there is a path. A *cycle* is a closed path $v_1, \ldots, v_k, v_1$ with $k \geq 3$ such that $v_1, ..., v_k$ are all distinct. (Sometimes in the literature this is called a *simple cycle.*) A graph is *acyclic* if it has no cycle. A *tree* is a connected acyclic graph. A *spanning tree* of a connected graph G is a subset $T \subseteq E$ of the edges such that (V, T) is a tree. (In other words, the edges in T must connect all nodes of G and contain no cycles.)

If a connected G has a cycle, then there is more than one spanning tree for G, and in general G may have exponentially many spanning trees, but each spanning tree has the same number of edges.

Lemma 3 *Every tree with n nodes has exactly $n - 1$ edges.*

The proof is by induction on n, using the fact that every (finite) tree has a *leaf* (i.e. a node of degree one).

We are interested in finding a minimum cost spanning tree for a given connected graph G, assuming that each edge e is assigned a *cost* $c(e)$. (Assume for now that the cost $c(e)$ is a nonnegative real number.) In this case, the cost $c(T)$ is defined to be the sum of the costs of the edges in T. We say that T is a *minimum cost spanning tree* (or an optimal spanning tree) for G if T is a spanning tree for G, and given any spanning tree T' for G, $c(T) \leq c(T')$.

Given a connected graph $G = (V, E)$ with n vertices and m edges $e_1, e_2, \ldots, e_m$, where $c(e_i)$ = "cost of edge e_i", we want to find a minimum cost spanning tree. It turns out (miraculously) that in this case, an obvious greedy algorithm (Kruskal's algorithm) always works. Kruskal's algorithm is the following: first, sort the edges in increasing (or rather nondecreasing) order of costs, so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$; then, starting with an initially empty tree T, go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

Kruskal's Algorithm:

Sort the edges so that: $c(e_1) \le c(e_2) \le \ldots \le c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..m$
(*) if $T \cup \{e_i\}$ has no cycle then
        $T \leftarrow T \cup \{e_i\}$
    end if
end for

But how do we test for a cycle (i.e. execute (*))? After each execution of the loop, the set T of edges divides the vertices V into a collection $V_1 \ldots V_k$ of *connected components*. Thus V is the disjoint union of $V_1 \ldots V_k$, each V_i forms a connected graph using edges from T, and no edge in T connects V_i and V_j, if $i \neq j$.

A simple way to keep track of the connected components of T is to use an array $D[1..n]$ where $D[i] = D[j]$ iff vertex i is in the same component as vertex j. So our initialization becomes:

$T \leftarrow \emptyset$
for $i : 1..n$
    $D[i] \leftarrow i$
end for

To check whether $e_i = [r, s]$ forms a cycle with T, check whether $D[r] = D[s]$. If not, and we therefore want to add e_i to T, we merge the components containing r and s as follows:

$k \leftarrow D[r]$
$l \leftarrow D[s]$
for $j : 1..n$
    if $D[j] = l$ then
        $D[j] \leftarrow k$
    end if
end for

The complete program for Kruskal's algorithm then becomes as follows:

```
Sort the edges so that: c(e_1) ≤ c(e_2) ≤ ... ≤ c(e_m)
T ← ∅
for i : 1..n
    D[i] ← i
end for
for i : 1..m
    Assign to r and s the endpoints of e_i
    if D[r] ≠ D[s] then
      T ← T ∪ {e_i}
      k ← D[r]
      l ← D[s]
      for j : 1..n
          if D[j] = l then
             D[j] ← k
          end if
      end for
    end if
end for
```

We wish to analyze the running of Kruskal's algorithm, in terms of n (the number of vertices) and m (the number of edges); keep in mind that $n-1 \leq m$ (since the graph is connected) and $m \leq \binom{n}{2} < n^2$. Let us assume that the graph is input as the sequence $n, I_1, I_2, \ldots, I_m$ where n represents the vertex set $V = \{1, 2, \ldots, n\}$, and I_i is the information about edge e_i, namely the two endpoints and the cost associated with the edge. To analyze the running time, let's assume that any two cost values can be either added or compared in one step. The algorithm first sorts the m edges, and that takes $O(m \log m)$ steps. Then it initializes D, which takes time $O(n)$. Then it passes through the m edges, checking for cycles each time and possibly merging components; this takes $O(m)$ steps, plus the time to do the merging. Each merge takes $O(n)$ steps, but note that the total number of merges is the total number of edges in the final spanning tree T, namely (by the above lemma) $n-1$. Therefore this version of Kruskal's algorithm runs in time $O(m \log m + n^2)$. Alternatively, we can say it runs in time $O(m^2)$, and we can also say it runs in time $O(n^2 \log n)$. Since it is reasonable to view the size of the input as n, this is a polynomial-time algorithm.

This running time can be improved to $O(m \log m)$ (equivalently $O(m \log n)$) by using a more sophisticated data structure to keep track of the connected components of T; this is discussed on page 570 of CLRS (page 505 of CLR).

Correctness of Kruskal's Algorithm

It is not immediately clear that Kruskal's algorithm yields a spanning tree at all, let alone a minimum cost spanning tree. We will now prove that it does in fact produce an optimal spanning tree. To show this, we reason that after each execution of the loop, the set T of edges can be expanded to an optimal spanning tree using edges that have not yet been considered. Hence after termination, since all edges have been considered, T must itself be a minimum cost spanning tree.

We can formalize this reasoning as follows:

Definition 2 *A set T of edges of G is* promising after stage i *if T can be expanded to a optimal spanning tree for G using edges from $\{e_{i+1}, e_{i+2}, \ldots, e_m\}$. That is, T is promising after stage i if there is an optimal spanning tree T_{opt} such that $T \subseteq T_{opt} \subseteq T \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$.*

Lemma 4 *For $0 \leq i \leq m$, let T_i be the value of T after i stages, that is, after examining edges $e_1, \ldots, e_i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq m$:*

$P(i)$: T_i is promising after stage i.

Proof:
We will prove this by induction. $P(0)$ holds because T is initially empty. Since the graph is connected, there exists *some* optimal spanning tree T_{opt}, and
$T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_1, e_2, \ldots, e_m\}$.

For the induction step, let $0 \leq i < m$, and assume $P(i)$. We want to show $P(i+1)$. Since T_i is promising for stage i, let T_{opt} be an optimal spanning tree such that
$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$. If e_{i+1} is rejected, then $T_i \cup \{e_{i+1}\}$ contains a cycle and $T_{i+1} = T_i$. Since $T_i \subseteq T_{opt}$ and T_{opt} is acyclic, $e_{i+1} \notin T_{opt}$. So
$T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Now consider the case that $T_i \cup \{e_{i+1}\}$ does *not* contain a cycle, so we have $T_{i+1} = T_i \cup \{e_{i+1}\}$.
If $e_{i+1} \in T_{opt}$, then we have $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.
So assume that $e_{i+1} \notin T_{opt}$. Then according to the Exchange Lemma below (letting T_1 be T_{opt} and T_2 be T_{i+1}), there is an edge $e_j \in T_{opt} - T_{i+1}$ such that $T'_{opt} = T_{opt} \cup \{e_{i+1}\} - \{e_j\}$ is a spanning tree. Clearly $T_{i+1} \subseteq T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. It remains to show that T'_{opt} is optimal. Since $T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$ and $e_j \in T_{opt} - T_{i+1}$, we have $j > i+1$. So (because we sorted the edges) $c(e_{i+1}) \leq c(e_j)$, so $c(T'_{opt}) = c(T_{opt}) + c(e_{i+1}) - c(e_j) \leq c(T_{opt})$. Since T_{opt} is optimal, we must in fact have $c(T'_{opt}) = c(T_{opt})$, and T'_{opt} is optimal.

This completes the proof of the above lemma, except for the Exchange Lemma.

Lemma 5 *(Exchange Lemma) Let G be a connected graph, let T_1 be any spanning tree of G, and let T_2 be be a set of edges not containing a cycle. Then for every edge $e \in T_2 - T_1$ there is an edge $e' \in T_1 - T_2$ such that $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G.*

Proof:
Let T_1 and T_2 be as in the lemma, and let $e \in T_2 - T_1$. Say that $e = [u, v]$. Since there is a path from u to v in T_1, $T_1 \cup \{e\}$ contains a cycle C, and it is easy to see that C is the only cycle in $T_1 \cup \{e\}$. Since T_2 is acyclic, there must be an edge e' on C that is not in T_2, and hence $e' \in T_1 - T_2$. Removing a single edge of C from $T_1 \cup \{e\}$ leaves the resulting graph acyclic but still connected, and hence a spanning tree. So $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G. $\square$

We have now proven Lemma 4. We therefore know that T_m is promising after stage m; that is, there is an optimal spanning tree T_{opt} such that $T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset = T_m$, and so $T_m = T_{opt}$. We can therefore state:

Theorem 1 *Given any connected edge weighted graph G, Kruskals algorithm outputs a minimum spanning tree for G.*

Discussion of Greedy Algorithms

Before we give another example of a greedy algorithm, it is instructive to give an overview of how these algorithms work, and how proofs of correctness (when they exist) are constructed.

A Greedy algorithm often begins with sorting the input data in some way. The algorithm then builds up a solution to the problem, one stage at a time. At each stage, we have a partial solution to the original problem – don't think of these as solutions to subproblems (although sometimes they are). At each stage we make some decision, usually to include or exclude some particular element from our solution; we never backtrack or change our mind. It is usually not hard to see that the algorithm eventually halts with some solution to the problem. It is also usually not hard to argue about the running time of the algorithm, and when it is hard to argue about the running time it is because of issues involved in the data structures used rather than with anything involving the greedy nature of the algorithm. The key issue is whether or not the algorithm finds an *optimal* solution, that is, a solution that minimizes or maximizes whatever quantity is supposed to be minimized or maximized. We say a greedy algorithm is optimal if it is guaranteed to find an optimal solution for every input.

Most greedy algorithms are not opitmal! The method we use to show that a greedy algorithm is optimal (when it is) often proceeds as follows. At each stage i, we define our partial solution to be *promising* if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage 0 is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follows. Say that stage $i + 1$ occurs, and that the partial solution after stage i is S_i and that the partial solution after stage $i+1$ is S_{i+1}, and we know that there is an optimal solution S_{opt} that extends S_i ; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1} . S_{i+1} extends S_i by making only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1}, and we can just let $S'_{opt} = S_{opt}$ and we are done. The hard part of the induction step is if S_{opt} does not extend S_{i+1}. In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1}, and
- S'_{opt} has value (cost, profit, or whatever it is we're measuring) at least as good as S_{opt}, so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any solution other than itself. Therefore, if we have proven the above, we know that the solution constructed must be optimal.

A Greedy Algorithm for Scheduling Jobs with Deadlines and Profits

The setting is that we have n jobs, each of which takes unit time, and a processor on which we would like to schedule them in as profitable a manner as possible. Each job has a profit associated with it, as well as a deadline; if the job is not scheduled by the deadline, then we don't get the profit. Because each job takes the same amount of time, we will think of a Schedule S as consisting of a sequence of job "slots" $1, 2, 3, \ldots$ where $S(t)$ is the job scheduled in slot t.
(If one wishes, one can think of a job scheduled in slot t as beginning at time $t - 1$ and ending at time t, but this is not really necessary.)

More formally, the input is a sequence $(d_1, g_1), (d_2, g_2), \cdots, (d_n, g_n)$ where g_i is a nonnegative real number representing the profit obtainable from job i, and $d_i \in \mathbb{N}$ is the deadline for job i; it doesn't hurt to assume that $1 \leq d_i \leq n$. (The reason why we can assume that every deadline is less than or equal to n is because even if some deadlines were bigger, every feasible schedule could be "contracted" so that no job was placed in a slot bigger than n.)

Definition 3 *A* schedule *S is an array: $S(1), S(2), ..., S(n)$ where $S(t) \in \{0, 1, 2, \cdots n\}$ for each $t \in \{1, 2, \cdots, n\}$.*

The intuition is that $S(t)$ is the job scheduled by S in slot t; if $S(t) = 0$, this means that no job is scheduled in slot t.

Definition 4 *S is* feasible *if*
(a) If $S(t) = i > 0$, then $t \leq d_i$. (Every scheduled job meets its deadline)
(b) If $t_1 \neq t_2$ and $S(t_1) \neq 0$, then $S(t_1) \neq S(t_2)$. (Each job is scheduled at most once.)

We define the *profit* of a feasible schedule S by
$P(S) = g_{S(1)} + g_{S(2)} + ... + g_{S(n)}$, where $g_0 = 0$ by definition.

Goal: Find a feasible schedule S whose profit $P(S)$ is as large as possible; we call such a schedule *optimal*.

We shall consider the following greedy algorithm. This algorithm begins by sorting the jobs in order of decreasing (actually nonincreasing) profits. Then, starting with the empty schedule, it considers the jobs one at a time; if a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

Greedy:
Sort the jobs so that: $g_1 \geq g_2 \geq \ldots \geq g_n$
for $t : 1..n$
    $S(t) \leftarrow 0$ {Initialize array $S(1), S(2), ..., S(n)$}
end for
for $i : 1..n$
    Schedule job i in the latest possible free slot meeting its deadline;
    if there is no such slot, do not schedule i.
end for

Example. Input of **Greedy**:

Job i:	1	2	3	4	Comments
Deadline d_i:	3	2	3	1	(when job must finish by)
Profit g_i:	9	7	7	2	(already sorted in order of profits)

Initialize $S(t)$:

t	1	2	3	4
$S(t)$	0	0	0	0

Apply **Greedy**: Job 1 is the most profitable, and we consider it first. After 4 iterations:

t	1	2	3	4
$S(t)$	3	2	1	0

Job 3 is scheduled in slot 1 because its deadline $t = 3$, as well as slot $t = 2$, has already been filled.

$P(S) = g_3 + g_2 + g_1 = 7 + 7 + 9 = 23.$

Theorem 2 *The schedule output by the greedy algorithm is optimal, that is, it is feasible and the profit is as large as possible among all feasible solutions.*

We will prove this using our standard method for proving correctness of greedy algorithms. We say feasible schedule S' *extends* feasible schedule S iff for all t $(1 \leq t \leq n)$,
if $S(t) \neq 0$ then $S'(t) = S(t)$.

Definition 5 *A feasible schedule is* promising after stage i *if it can be extended to an optimal feasible schedule by adding only jobs from* $\{i+1, \cdots, n\}$.

Lemma 6 *For $0 \leq i \leq n$, let S_i be the value of S after i stages of the greedy algorithm, that is, after examining jobs $1, \cdots, i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq n$:*

$P(i)$: S_i is promising after stage i.

This Lemma implies that the result of **Greedy** is optimal. This is because $P(n)$ tells us that the result of **Greedy** can be extended to an optimal schedule using only jobs from $\emptyset$. Therefore the result of **Greedy** must be an optimal schedule.

Proof of Lemma: To see that $P(0)$ holds, consider any optimal schedule S_{opt}. Clearly S_{opt} extends the empty schedule, using only jobs from $\{1, \cdots, n\}$.

So let $0 \leq i < n$ and assume $P(i)$. We want to show $P(i+1)$. By assumption, S_i can be extended to some optimal schedule S_{opt} using only jobs from $\{i+1, \cdots, n\}$.

Case 1: Job $i+1$ cannot be scheduled, so $S_{i+1} = S_i$.
Since S_{opt} extends S_i, we know that S_{opt} does not schedule job $i+1$. So S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

Case 2: Job $i+1$ is scheduled by the algorithm, say at time t_0 (so $S_{i+1}(t_0) = i+1$ and t_0 is the latest free slot in S_i that is $\leq d_{i+1}$).

Subcase 2A: Job $i+1$ occurs in S_{opt} at some time t_1 (where t_1 may or may not be equal to t_0).

Then $t_1 \leq t_0$ (because S_{opt} extends S_i and t_0 is as large as possible) and $S_{opt}(t_1) = i+1 = S_{i+1}(t_0)$.

If $t_0 = t_1$ we are finished with this case, since then S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. Otherwise, we have $t_1 < t_0$. Say that $S_{opt}(t_0) = j \neq i+1$. Form S'_{opt} by interchanging the values in slots t_1 and t_0 in S_{opt}. Thus $S'_{opt}(t_1) = S_{opt}(t_0) = j$ and $S'_{opt}(t_0) = S_{opt}(t_1) = i+1$. The new schedule S'_{opt} is feasible (since if $j \neq 0$, we have moved job j to an earlier slot), and S'_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. We also have $P(S_{opt}) = P(S'_{opt})$, and therefore S'_{opt} is also optimal.

Subcase 2B: Job $i+1$ does not occur in S_{opt}.

Define a new schedule S'_{opt} to be the same as S_{opt} except for time t_0, where we define $S'_{opt}(t_0) = i+1$. Then S'_{opt} is feasible and extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

To finish the proof for this case, we must show that S'_{opt} is optimal. If $S_{opt}(t_0) = 0$, then we have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} \geq P(S_{opt})$. Since S_{opt} is optimal, we must have

$P(S'_{opt}) = P(S_{opt})$ and S'_{opt} is optimal. So say that $S_{opt}(t_0) = j$, $j > 0$, $j \neq i+1$. Recall that S_{opt} extends S_i using only jobs from $\{i+1, \cdots, n\}$. So $j > i+1$, so $g_j \leq g_{i+1}$. We have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} - g_j \geq P(S_{opt})$. As above, this implies that S'_{opt} is optimal. $\square$

We still have to discuss the running time of the algorithm. The initial sorting can be done in time $O(n \log n)$, and the first loop takes time $O(n)$. It is not hard to implement each body of the second loop in time $O(n)$, so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to $O(n \log n)$, but in any case it is a polynomial-time algorithm.

Greedy Summary: In general, greedy algorithms are very fast. Unfortunately, for some kinds of problems they only do not always yield an optimal solution (such as for Simple Knapsack). However for other problems (such as the scheduling problem above, and finding a minimum cost spanning tree) they *always* find an optimal solution. For these problems, greedy algorithms are great.

Scheduling Jobs with Deadlines and Profits, on Multiple Processors

The input to this problem, as before, is a sequence of n (unit duration) jobs $(d_1, g_1), (d_2, g_2), \cdots, (d_n, g_n)$ where d_i and g_i are the deadline and profit of job i. The difference is that we may now have more than one processor on which we are allowed to schedule jobs, so that our optimal profit may be higher. Of course, we are not allowed to schedule the same job on more than one processor.

We therefore also have as input a positive integer m denoting the number of processors, where $m \leq n$. A schedule is now defined to be a function $S : \{1, 2, \cdots, m\} \times \{1, 2, \cdots, n\} \rightarrow \{0, 1, 2, \cdots, n\}$. The intuition is that $S(k, t)$ is the job scheduled on processor k in slot t; $S(k, t) = 0$ means that no job is scheduled in slot t of processor k. We say S is feasible if the following two properties hold:
(a) If $S(k, t) = i > 0$ then $t \leq d_i$; that is, no job can be scheduled after its deadline.
(b) If $S(k, t) = S(k', t') > 0$, then $k = k'$ and $t = t'$; that is, no job can be scheduled more than once.

We define the profit of a (feasible) schedule S by

$$P(S) = \sum_{1 \leq k \leq m,\, 1 \leq t \leq n} g_{S(k,t)}$$

where $g_0 = 0$ by definition. The goal is to find a feasible schedule with as high a profit as possible.

Exercise: Give a polynomial time Greedy algorithm for this problem. Prove that your algorithm is optimal. Analyze the running time of your algorithm.

"Activity" Scheduling

(This development will be similar to that in CLR.)
An *activity* is defined to be a pair (s, f) of nonnegative integers such that $s < f$; the intuition is that s is the starting time of the activity and f is the finishing time of the activity. We will be given a sequence of activities, and we wish to schedule as many of them as possible (on one processor) so that no two scheduled activities overlap. If (s, f) and (s', f') are activities, we say they *do not overlap* if $f \leq s'$ or $f' \leq s$.

More formally, the input is a sequence of n activities, $(s_1, f_1), (s_2, f_2), \cdots, (s_n, f_n)$. A schedule is defined to be a set $A \subseteq \{1, 2, \cdots, n\}$ such that for all $i, j \in A$, if $i \neq j$ then (s_i, f_i) does not overlap (s_j, f_j). The goal is to find a schedule A such that $|A|$ (the size of A) is as big as possible. (Since we are only interested in how many activities are in A, we are in effect treating each activity as if it yields unit profit. More complicated versions of this problem allow different activities to yield different profits.)

Our greedy algorithm will work as follows. First, we will sort the activities according to nondecreasing finish times. Then we will go through the activities, one at a time, scheduling each activity if possible. Before formally stating the algorithm, we give the following exercise.

Exercise: Prove that if, instead of sorting by nondecreasing finish times, we sort by nondecreasing start times, then the algorithm would not work.
Prove that if, instead of sorting by nondecreasing finish times, we sort by nondecreasing job size (that is, $f - s$), then the algorithm would not work.

We now give code for the algorithm. We will use a variable e to keep track of the last finish time of an activity added to A, where $e = 0$ if A is empty. That is, since the jobs are sorted according to finish time, e is the earliest start time at which we can add an activity to A.

Greedy:
Sort the activities so that: $f_1 \leq f_2 \leq \ldots \leq f_n$
$A \leftarrow \emptyset$
$e \leftarrow 0$
for $i : 1..n$
    if $s_i \geq e$ then
        $A \leftarrow A \cup \{i\}$
        $e \leftarrow f_i$
    end if
end for

Analyzing the running time of the algorithm, we see that the $n \log n$ time for the sort dominates the linear time for the loop, so the total time is $O(n \log n)$.

We now want to prove the algorithm is optimal.

Theorem 3 *This Greedy algorithm outputs an optimal (that is, largest possible) schedule.*

We will prove this using our standard method for proving correctness of greedy algorithms. Let A_i and e_i be the values of A and e after the body of the 'for' loop has been executed i times. It is easy to prove by induction on i that for all i, $0 \leq i \leq n$:
(*) $A_i \subseteq \{1, 2, \cdots, i\}$ and
(**) $e_i = \max\{f_j \mid j \in A_i\}$ (where $\max \emptyset = 0$).

The Theorem clearly follows from the following Lemma.

Lemma 7 *For $0 \leq i \leq n$, A_i is* promising *after stage i, that is, there exists an optimal schedule A_{opt} such that $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \cdots, n\}$.*

Proof of Lemma: Clearly A_0 is promising after stage 0.

So let $0 \leq i < n$ and assume that A_i is promising after stage i. We want to show A_{i+1} is promising after stage $i+1$. Let A_{opt} be an optimal schedule such that
$A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \cdots, n\}$. Clearly (*) and (**), together with the fact that the finish times are in sorted order, imply that $e_i \leq f_{i+1}$.

Case 1: $s_{i+1} < e_i$, so $A_{i+1} = A_i$.
Then, since e_i is the finish time of an activity in A_i and $e_i \leq f_{i+1}$, we see that activity $i+1$ overlaps an activity in A_i, so $i+1 \notin A_{opt}$. So $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$.

Case 2: $s_{i+1} \geq e_i$, so $A_{i+1} = A_i \cup \{i+1\}$.

Subcase 2A: $i+1 \in A_{opt}$.
Then $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$.

Subcase 2B: $i+1 \notin A_{opt}$.
Since $s_{i+1} \geq e_i$, (**) implies that activity $i+1$ does not overlap any activity in A_i. A_{opt} cannot equal A_i, for then $A_{opt} \cup \{i+1\}$ would be a larger schedule than A_{opt}. So let $u \geq i+2$ be the activity in $A_{opt} - A_i$ with the smallest finish time. Consider the activities in $A_{opt} - A_i$ other than u: since these all have start times $\geq f_u$, and since $f_u \geq f_{i+1}$, we see that none of these activities overlap activity $i+1$. Let $A'_{opt} = (A_{opt} - \{u\}) \cup \{i+1\}$. A'_{opt} is a schedule, and since it has the same size as A_{opt}, it is an optimal schedule. We also clearly have $A_{i+1} \subseteq A'_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$. $\square$

Activity Scheduling on Multiple Processors

For this problem, the input will consist of a positive integer m in addition to $(s_1, f_1), (s_2, f_2), \cdots, (s_n, f_n)$, where $m \leq n$ denotes the number of processors. Of course, we are not allowed to schedule overlapping jobs on the same processor.

More formally, a schedule is now defined to be a sequence $A = (A_1, A_2, \cdots, A_m)$ where each $A_k \subseteq \{1, 2, \cdots, n\}$ and where the following two properties hold:
(a) For all k, $1 \leq k \leq m$, and for all $i, j \in A_k$, if $i \neq j$ then activity i does not overlap activity j.
(b) For all k_1, k_2, $1 \leq k_1, k_2 \leq m$, if $k_1 \neq k_2$ then $A_{k_1} \cap A_{k_2} = \emptyset$.

The *size* of schedule A is defined to be $|A_1| + |A_2| + \cdots + |A_m|$. The goal is to find a schedule with as large a size as possible.

Exercise: Give a polynomial time Greedy algorithm for this problem. Prove that your algorithm is optimal. Analyze the running time of your algorithm.

Making Change

Canadian coins come in the following denominations: 1, 5, 10, 25, 100, 200. Let's assume that when we give somebody change we want to use as few coins as possible. For example, if we wanted to create the value 143, we *could* use:
3 quarters, 6 dimes, 1 nickel and 3 pennies,
for a total of 13 coins. A better way would be to use a *greedy* algorithm where we make change by, at each stage, using as big a coin as possible to make change for the remaining amount. For example, to create the value 143, we first use a loonie, followed by a quarter,

followed by a dime, followed by a nickel, followed by 3 pennies, for a total of only 7 coins. It turns out that this greedy algorithm always yields an optimal solution, for the case of Canadian coins.

It is important to realize that there are other currencies for which this greedy algorithm is *not* optimal. For example, assume we modify our currency by getting rid of the nickel. Say that we want to create the value 30. The greedy algorithm will use 1 quarter and 5 pennies, whereas the optimal method uses 3 dimes.

We wish to discuss this greedy algorithm for an arbitrary currency. Say that we have k distinct coin denominations of integer values $C[1] = 1 < C[2] < \ldots < C[k]$. (Note that because we have a coin of value 1, it will be possible to create any nonnegative integer amount.) We are given an input $n \in \mathbb{N}$, and we wish to create a sequence of coin values $S = u_1, u_2, \ldots, u_m$ that adds up to n such that m is as small as possible. Here is code for the greedy algorithm discussed above. In this algorithm we use $\circ$ to denote concatenation of sequences, and we assume the empty sequence adds up to 0. We use the variable s to denote the amount of change that our algorithm has produced so far.

$S \leftarrow$ empty sequence; $\quad s \leftarrow 0$
while $s < n$ do
$\qquad u \leftarrow$ the largest member of $\{1, 2, \ldots, k\}$ such that $C[u] \leq n - s$
$\qquad S \leftarrow S \circ C[u]; \quad s \leftarrow s + C[u]$
end while

Assuming the currency C (and hence k) is constant, this algorithm clearly runs in time $O(n)$. (If C as well as n is an input, then it is easy to implement this algorithm so that it runs in time $O(n+k)$.) As we have seen, it is not guaranteed to produce optimal outputs. The following lemma shows one case of C where the algorithm can be proven to be optimal.

Lemma 8 *Let $k = 4$, and let $C[1] = 1, C[2] = 7, C[3] = 13, C[4] = 19$. Then the greedy change-making algorithm above always produces an output (S) of coins summing to n using as few coins as possible.*

Proof: It is easy to see that at the end of each each stage i, S is a sequence of i coins adding up to $s \leq n$. We claim, furthermore, that at the end of each stage S is *promising* in the sense that there is a way of extending S to an optimal (in the sense of using as few coins as possible) sequence for n. This is clearly sufficient for proving the Lemma.

We will prove this for every stage i by induction on i. It is trivial true initially, that is after stage 0. So assume it is true at stage i, and that stage $i+1$ occurs; we want to prove it is true after stage $i+1$.

Let $a_1, a_2, \ldots, a_i$ be the sequence the greedy algorithm produces at the end of stage i, and say that they sum to $s < n$. By the induction hypothesis we can let $a_1, a_2, \ldots, a_i, b_{i+1}, \ldots, b_w$ be an optimal solution (to the problem of making change for n); assume that $b_{i+1} \geq b_{i+2} \geq \ldots \geq b_w$. Say that a_{i+1} is the coin the greedy algorithm produces in stage $i+1$; a_{i+1} is the largest denomination coin $\leq n-s$. If $b_{i+1} = a_{i+1}$ then $a_1, a_2, \ldots, a_i, b_{i+1}, \ldots, b_w$ is an optimal solution that extends $a_1, a_2, \ldots, a_i, a_{i+1}$ and we are done. So assume $b_{i+1} < a_{i+1}$. Let t be the smallest number such that $b_{i+1} + b_{i+2} + \ldots + b_t \geq a_{i+1}$. Note that if $b_t = 1$, then $b_{i+1} + b_{i+2} + \ldots + b_t = a_{i+1}$; if we replaced the subsequence $b_{i+1}, \ldots, b_t$ by the single coin a_{i+1}, then we would obtain a *shorter* sequence summing to n, contradicting the optimality of $a_1, a_2, \ldots, a_i, b_{i+1}, \ldots, b_w$. So we can assume that $b_{i+1}, \ldots, b_t$ doesn't contain a coin of value 1. Since $1 < b_{i+1} < a_{i+1}$, we have only two cases for a_{i+1}.

Case 1: $a_{i+1} = 19$.

Case 1.0: $b_{i+1}, \ldots, b_t$ doesn't contain the coin 13. Then $b_{i+1}, \ldots, b_t$ consists of exactly three occurrences of the coin of value 7 and nothing else. We can replace these three coins by $(19, 1, 1)$, and have an optimal solution extending $a_1, a_2, \ldots, a_{i+1}$, namely
$a_1, a_2, \ldots, a_{i+1}, 1, 1, b_{i+4}, \ldots, b_w$.

Case 1.1: $b_{i+1}, \ldots, b_t$ contains the coin 13 exactly one time. (Exercise)

Case 1.2: $b_{i+1}, \ldots, b_t$ contains the coin 13 exactly two times. (Exercise)

Case 2: $a_{i+1} = 13$. (Exercise) $\square$

Exercise: Prove that the greedy change-making algorithm produces optimal results in the case of Canadian currency:
$k = 6$ and $C[1] = 1, C[2] = 5, C[3] = 10, C[4] = 25, C[5] = 100, C[6] = 200$.