Dynamic programming vs greedy algorithms

Both are for optimization problems that have *optimal substructure*.

How are they different:

- Greedy algorithms make decisions that are locally optimal
- Greedy algorithms tend to be simpler, more straightforward to write
- The hard part of greedy algorithms is *determining whether an optimal greedy solution exists*

The activity selection problem (§16.1)

- Problem: S, the given, complete set of activities, $\{a_1, a_2, \ldots\}$.
- Subproblem: S_{ij}, the set of activities that fall between a_i and a_j.
- Solution to subproblem: A_{ij}, a "maximal" (in terms of cardinality) subset of S_{ij}

Claim (Theorem 16.1 in the book):

> *Let a_m be an activity with earliest finish time in S_k. There exists a maximal solution to S_k that includes a_m.*

Notation switch in the book: $S_k = S_{kn}$, $A_k = A_{kn}$.

Theorem 16.1. Let a_m be an activity with earliest finish time in S_k. There exists a maximal solution to S_k that includes a_m.

Let A_k be a maximal solution to subproblem S_k.

Suppose $a_m \notin A_k$

Let a_j be the element in A_k with earliest finish time.

Consider the set $(A_k - \{a_j\}) \cup \{a_m\}$.

Since

$$\begin{aligned} f_m &\leq f_j \\ &\leq s_x \end{aligned}$$

. . . for all $a_x \in A_k$, a_m *does not conflict with anything in* $(A_k - \{a_j\}) \cup \{a_m\}$.

$$\begin{aligned} |(A_k - \{a_j\}| &= |A_k| - 1 + 1 \\ &= |A_k| \end{aligned}$$

So $(A_k - \{a_j\}) \cup \{a_m\}$ *is also. maximal.* $\square$

Elements of the greedy strategy

find	1.	The optimal substructure	
develop	2.	A recursive solution	
prove	3/4.	The greedy choice	a. One subproblem remains b. It's safe to pick a local optimum
develop	5.	A recursive algorithm	
convert to	6.	An iterative algorithm	

Ex 16.2-1. Suppose we have items 1 through n, with v_i being the value of the whole thing and w_i being its weight. W is the capacity of the knapsack. Until the knapsack is full, (a) choose the item with highest value density ($\frac{v_i}{w_i}$) and take as much as will fit; (b) repeat with subknapsack $W - w_i$, assuming $w_i < W$.
Claim: For a given instance of the problem, there is a solution using this greedy approach.
Demonstration. Suppose $A = (a_1, a_2, \ldots a_n)$ is an optimal solution, indicated by the weight taken from each item. It must be that $a_1 + a_2 + \ldots a_n \leq W$, but assume that the total weight is in fact equal to W, since you can always increase the knapsack's value by adding something more. The value of the solution is

$$\sum_{k=1}^{n} a_i \frac{v_i}{w_i}$$

Suppose further that item m has the highest value density and that $a_m < w_m$. (We're also assuming $w_m < W$; the argument would be basically the same otherwise, just a little more complicated.)
Start removing items from the solution until you've removed $w_m - a_m$ weight, and then add the rest of item m. Since item m has the highest density, you now have a more valuable knapsack.

Ex 16.2-3. Always pick the smallest, most valuable one. This works because any solution that did not have the smallest, most valuable one can be made more valuable (without increaseing weight) by replacing one of the others with the smallest, most valuable one.

Ex 16.2-4. Let $A = \{a_1, a_2, \ldots a_n\}$ be the locations of the drinking fountains in miles from Grand Forks. (Let $a_0 = 0$ be Grand Forks and let a_{n+1} be Williston.) We will use these distances to identify them. We want an optimal solution $b_1, b_2, \ldots b_k$, a list of fountains to stop at.

Alternately, call that the set $B_{0,n+1}$, and in general let $B_{i,j}$ be a minimal set of fountains to stop at when leaving a_i with a full bottle and arrive at a_j; the set is "exclusive"—do not count a_i or a_j.

In all this, we're assuming that we leave Grand Forks with a full bottle and that the distance between fountains is always less than m.

To characterize the solution,

$$B_{i,j} = \min_{i<k<j} |\{a_k\} \cup B_{i,k} \cup B_{k,j}|$$

Our main idea is always to choose the furthest fountain within m miles.

$\textsc{Best-Station}(A, i, j)$ // precondition: $i \leq j \leq m$
    if $a_j - a_i < m$
        return $\emptyset$
    $k = i + 1$
    while $a_k - a_i < m$
        $k = k + 1$
    $k = k - 1$
    return $\{a_k\} \cup \textsc{Best} - \textsc{Station}(A, k, j)$

The top level call is to $\textsc{Best-Station}(A, 0, m + 1)$.

Lemma

Throughout the algorithm, $k > i$.

Proof. *Initially, $k = i + 1 > i$, and the loop only increases it. (Now the only way this lemma could be false is if the loop doesn't run at all and then we subtract one from k, leaving $k = i$. The rest of the proof is to show that doesn't happen.)*

The initial if statement of the algorithm guarantees that $a_j - a_i \geq m > 0$. so $a_i < a_j$, and so $a_i \neq a_m$. Moreover, there exists a_ℓ such that $a_\ell > a_i$ and $a_\ell - a_i < m$, by assumption that I made above.

Hence the while loop will execute at least once, and so $k \geq i+2$ on termination of the while loop. By substitution, $k \geq i + 1$ after the final assignment to k, and so $k > i$. $\square$

Now, why is this optimal? We will prove that there exists an optimal solution to $B_{i,j}$ that includes the futhest fountain within range.

Proof. *Let a_x be the furthest fountain within range, that is,*

$$\forall\ a_\ell \in A_{i,j},\ \textit{if } a_\ell - a_i \leq m \textit{ then } a_\ell \leq a_x$$

Suppose $C_{i,j}$ is a minimal solution. Let c_0 be the first fountain in $C_{i,j}$. It must be that $c_0 - a_i \leq m$, or else it wouldn't be a solution. So, either $c_0 = a_x$ or $c_0 < a_x$.

Case 1. *Suppose $c_0 = a_x$. Done.*
Case 2. *Suppose $c_0 < a_x$. Then let $C'_{i,j} = C_{i,j} - \{c_0\} \cup \{a_x\}$. (We're going to proove that $C'_{i,j}$ is an optimal solution.)*

Clearly $|C'_{i,j}| = |C_{i,j}|$, so, if it is a solution, it's optimal. Let c_1 be the next station in $C_{i,j}$ after c_0. $c_1 - c_0 < m$, or else $C_{i,j}$ would not be a solution. $c_1 - a_x < m$, since $c_0 < a_x$. So $C'_{i,j}$ is a solution.

Therefore, an optimal solution including a_x exists. $\square$