**Dynamic Programming**

- Choice at each step depends on solutions to subproblems
- Solved bottom up, progressing from smaller subproblems to larger ones or top down but memoizing

**Greedy**

- Take choice that looks best at each step, then solve remaining subproblem (this choice may depend on choices so far, but not on future choices.)
- Usually solves top down.
- Can usually make greedy choice more efficiently (fewer alternatives) than in dynamic programming

**How to design a greedy algorithm**

1. Describe the optimization problem as one in which we make a choice and are left with one subproblem.
2. Prove that there is an optimal solution that makes the greedy choice.
3. Demonstrate optimal substructure by showing that by combining the greedy choice with the solution to the subproblem we arrive at an optimal solution.
Optimal Substructure

A problem has this property if an optimal solution contains optimal solutions to subproblems.

To show a problem has this property with a
- dynamic programming approach usually requires the analysis of multiple subproblems at each level.
- greedy approach is usually easier, requiring analysis of one subproblem combined with the greedy choice. Implicitly uses induction.

Once you have formulated a problem to show it has optimal substructure, you will be tempted to design a dynamic programming algorithm when a greedy algorithm may suffice.
Fractional knapsack problem

We have n items, each with weight $w_i$ and value $v_i$.

Choose fractions of items to put in knapsack whose:

- sum of $v_i$ is maximum
- sum of $w_i \leq W$, the capacity of knapsack

Greedy solution:

1. Take as much as possible of item with the greatest value per pound $\frac{v_i}{w_i}$
2. If knapsack not full, repeat with item having next greatest $\frac{v_i}{w_i}$

Complexity?

Requires sorting the n items in ascending order of $\frac{v_i}{w_i}$.

$O(n \log n)$
Example (from book)

Items sorted by \( \frac{v_i}{w_i} \):

\[
\begin{align*}
\$60 & \quad \frac{60}{10} & \frac{60}{20} & \frac{60}{30} \\
\$100 & \quad \frac{100}{20} & \frac{100}{20} & \frac{100}{20} \\
\$120 & \quad \frac{120}{30} & \frac{120}{30} & \frac{120}{30} \\
\end{align*}
\]

Knapsack: 6, 5, 4

Greedy steps:

\[
\begin{align*}
& \frac{2}{3} \cdot \frac{120}{20} \\
& + \frac{100}{20} \\
& + \frac{60}{20} \\
& = \$240
\end{align*}
\]

What if we cannot take fractions of items?

\[
\begin{align*}
& \frac{100}{20} \\
& \frac{60}{20} \\
& = \$160
\end{align*}
\]

Can't pack item 3.

Not optimal solution, which is \$220
Greedy works for fractional knapsack, but not for 0-1 knapsack.

So, must devise a dynamic programming approach!

Let's try to define a recurrence relation for the value of all items in the knapsack for different sizes of subproblems.

Subproblem size determined by remaining items and remaining knapsack capacity.

Subproblem having remaining items 1,..., i and remaining capacity w:

\[ c[i, w] \]

If \( i = 0 \) or \( w = 0 \) \( c[i, w] = 0 \)

If \( w_i > w \) (Item i will not fit) \( c[i, w] = c[i-1, w] \)

Otherwise (Item i will fit) \( c[i, w] = \max(v_i + c[i-1, w-w_i], c[i-1, w]) \)

Don't put it in

Put it in

How would you fill in the c table?