Greedy or Dynamic Programming?

1. Divide problem into smaller versions of the same problem whose solutions are easily combined to form solution to original problem.

2. Are the solutions to the subproblems independent?

   Yes  No

   Greedy  Dynamic Programming

Example from Chapter 16: Activity selection

Given set of activities \( S = \{a_1, a_2, \ldots, a_n\} \) start times \( s_i \) and finish times \( f_i \), find maximum-sized subset of compatible (non-overlapping in time) activities.

Divide into smaller versions:

Activities sorted by finish time

\[
S_{27} = \{a_4, a_5, a_7\}
\]

\[
S_{ij} = \{a_k \mid s_k \geq f_i, f_k \leq s_j\}
\]
Activities sorted by finish time

\[ S_i = \sum a_i, \quad a_i \geq f_i \]

\[ S_{ij} = \sum a_i \mid f_i \leq s_j \]

Let the count of the number of activities in the solution for set of activities in \( S_{ij} \) be \( c[i,j] \).

We can split this count into two parts for subsets \( S_{ik} \) and \( S_{kj} \) with activity \( a_k \). Finding the optimal \( a_k \) sets up this recurrence relation:

\[
c[i,j] = \begin{cases} 
0, & \text{if } S_{ij} = \emptyset \\
\max \left( c[i,k] + 1 + c[k,j] \right), & \text{otherwise} 
\end{cases}
\]

From this recurrence relation, you should be able to define and implement a recursive function to solve the activity selection problem using this dynamic programming approach and analyze its time complexity.

But we should have first checked the subproblems for independence. A little thought (and reading Section 16.1) shows that they are. The solutions to \( S_{ik} \) and \( S_{kj} \) can be concatenated to find solution to \( S_{ij} \).
Remember that we assume activities are sorted by their finish times.

Instead of solving the subproblems before making a choice, as implied by this recurrence relation

\[
C[i, j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset \\
\max \{ C[i, k] + 1 + C[k+1, j], \text{ otherwise} \}
\end{cases}
\]

Let's try making the choice first, then solve the subproblem(s). How should we do this?

Well, for the activity selection problem, maybe we should first pick the activity that finishes first. This leaves the most room for additional activities. So, we can add a, to our solution, then solve the slightly smaller problem \( S_{in} \). We can say that picking a, solves the problem \( S \) if we add a fictitious activity \( a_0 \) with \( s_0 = f_0 = 0 \).

Our text gives this recursive version of the algorithm:

```
Select_Activities (S, f, k, n): # solve S_{kn}
    m = k + 1
    while m ≤ n and s[m] < f[k]:
        m = m + 1
    if m ≤ n:
        return {am3} \cup Select_Activity (S_f, m, n)
    else:
        return \emptyset
```

What would call tree look like?

See page 420
Theorem 16.1 (page 418)

For subproblem $S_{kn}$, let $a_m$ have earliest finish time in $S_{kn}$. Then $a_m$ is included in some maximum-sized subset of compatible activities in $S_{kn}$.

Can replace $y$ with $x$ and still have solution.
What is time complexity? What should we count?

Since this recursive function is almost tail-recursive, we should be able to convert it to an iterative function.

Select Activity \((s, f)\):  
\[
\begin{align*}
   n &= s.\text{length} \\
   A &= \emptyset \\
   k &= 1 \\
   \text{for } m = 2 \text{ to } n: \\
   &\quad \text{if } s[m] \geq f[k]: \\
   &\quad\quad A = A \cup \{a_m\} \\
   &\quad\quad k = m \\
\end{align*}
\]
return \(A\)

What if each activity has a weight and we want set of compatible activities with largest sum of weights?