Recap

Main Idea: To solve a big problem:
- Identify smaller subproblems s.t. a solution to the big problem can be derived from solutions to subproblems.
- Solve all subproblems from "small to large"
- Analyze runtime & Memory.

Alternative View: Recursion with Memoization (avoids doing the same subproblem again & again).

Example: "Big" problem: Given n calculate Fibn
Subproblems: for i=2,3,...,n calculate Fibi

Code:
F0=0, F1=1
for i=2,...,n
Fi = Fi-1 + Fi-2

Recursion w. Memoization

```
def fibMem(n):
    if n<=1: return n
    if n in Mem: return Mem[n]
    Mem[n] = fibMem(n-1) + fibMem(n-2)
    return Mem[n]
```
More examples from last time:
- Shortest path in a DAG
- Longest path in a DAG.

Problem 2: Longest Increasing Subsequence (LIS)

- **Input:** Array of \( n \) numbers, e.g. \( x_1, x_2, \ldots, x_n \)
  
  \( \{3, 2, 7, 4, 5, 6\} \)

- **Goal:** Find longest subsequence that is strictly increasing (non-consecutive)

  Greedy: not optimal.

  Optimal: \( 1, 2, 4, 5, 6 \)

  \( \rightarrow 1, 3, 4, 5, 6 \)

  Subproblems:

  First try \( \forall i = 1, \ldots, n \) : \( f(i) = \) longest increasing subsequence from \( f(1), \ldots, f(i-1) \) in \( x_1, \ldots, x_i \)

  Second try \( \forall i = 1, \ldots, n \) : \( f(i) = \) LIS in \( x_i, \ldots, x_n \) that includes \( x_i \).

  \( f(n) = \max(1, \max_{i < n} (f(i) + 1)) \)

  \( \forall i < n \) the longest subsequence that ends in \( x_i \).
n subproblems
each can be solved from prev ones with additional time $O(n^2)$.

$$f(i) = \max (1, \max_{j < i} (1 + f(j)))$$

Runtime: $O(n^2)$ time.
Memory: $O(n)$ memory.

Subproblems: $f(i) =$ longest subseq in $x_1, \ldots, x_i$ that uses $x_i$.

return $\max (f(1), f(2), \ldots, f(n))$. 

\[ x_i < x_j \]

\[ x_i < x_j \]
Problem 3: Edit Distance

Given two strings: \( S[1...n] \)  \( T[1...m] \)

Find fewest number of edits to turn \( S \) into \( T \).

Edits allowed:
1. Insert character to \( S \).
2. Delete " " from \( S \).
3. Substitute a character for another.

Example:
\[
S = "\text{snowy}" \\
T = "\text{sunny}"
\]

\[
s\text{n} \text{ow} y \\
\text{s}\overline{u}n\text{n} \text{ow} y \\
\text{sunny} \\
\text{sunny}
\]

- Cost of this alignment
\[
s - \text{nowy}^{3} \\
- S - \text{unny}
\]
**DP:**

- **Subproblems:**
  - For \(0 \leq i \leq n, 0 \leq j \leq m\):

    \[ f(i, j) = \text{Edit Distance}(s[1 \ldots i], t[1 \ldots j]) \]

    - Look at last char in optimal alignment:
      \[
      \begin{align*}
      s[i] & \quad T[j] \\
      - & \quad -
      \end{align*}
      \]

    \[
    f(i, j) = \min \left( 1 + f(i-1, j), 1 + f(i, j-1), f(i-1, j-1) + \delta_{i, j} \right)
    \]

    \[ \delta_{i, j} = \sum_{t=1}^{\infty} \]

    - **Edge Cases:**
      \[ f(i, 0) = i, \quad f(0, j) = j \]

    - **Runtime:** \(O(nm)\) subproblems \(\Rightarrow O(mn)\) time.

    - **Memory:** \(O(nm)\).
```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Memo:** $O(m)$
- Row by row
- $O(n)$
- Col by col

Graph:

- $f(i-1,j)$
- $H_{ij} - 1$
- $f(i,j)$
Example 4: Knapsack.

We're robbing a bank!
We find \( n \) items in the safe with weights \( w_1, \ldots, w_n \) & values \( v_1, \ldots, v_n \).
We have a bag that can carry at most \( W \) pounds.

Goal: Find most valuable choice that fits the knapsack.

Example:

\[
\begin{align*}
    w_1 &= 11 & v_1 &= 15 \\
    w_2 &= 10 & v_2 &= 10 \\
    w_3 &= 10 & v_3 &= 10
\end{align*}
\]

\( N = 20 \)

Greedy: Pick every item that maximizes \( \frac{v_i}{w_i} \) not work.

DP:

\[
f(i, u) = \max \text{ value when packing items } 1, \ldots, i \text{ in a bag of capacity } u.
\]

\[
f(i, u) = \max(f(i-1, u), f(i-1, u-w_i) + v_i) \quad \text{if } w_i \leq u,
\]

\[
f(i, u) = \max(f(i-1, u), f(i-1, u-w_i) + v_i) \quad \text{o.w.}
\]
Runtime: $O(n \cdot W)$. Is this a polynomial-time algorithm? No!

A polynomial time alg. runs in time polynomial in the input length.

Here, the input is:

$$(v_1, ..., v_n), (w_1, ..., w_n), W$$

Input length: $O(n \cdot \log W)$

so runtime can be exp. in input length.

For example, if $W = 2^n$.

Runtime: $O(n \cdot 2^n)$

Input Length: $O(n^3)$ Gates.

This algorithm would be polynomial time if we further assume that $W$ is at most polynomial in $n$. 
Example 5: Shortest Path in General Graphs

\[ G = (V, E) \quad w : E \to \mathbb{Z} \quad \text{(either positive or negative)} \]

Assume: no negative cycles.

Given \( s, t \)

Goal: compute distance from \( s \) to \( t \).

Subproblems: distance from \( s \) to \( v \) for any \( v \in V \).

Doesn't work.

Subproblems: distance from \( s \) to \( v \) with at most \( i \) edges.

\[ \text{dist}(v, i) = \text{Distance from } s \text{ to } v \text{ with at most } i \text{ edges} \]

\[ \text{dist}(v, i) = \min \left( \text{dist}(v, i-1), \min_{u \in V : (u, v) \in E} \left( \text{dist}(u, i-1) + w(u, v) \right) \right) \]

Runtime: \# subproblems \( \times n \) choices for \( i \) \( \times n \) choices for \( v \).

For subproblem \( (v, i) \):

\[ \text{time} = O(1 + \text{indeg}(v)) \]

\[ \leq \sum_{v \in V} O(1 + \text{indeg}(v)) = n \cdot O(n + m) \]
More Great Problem in the Book

\[
((A_1 \cdot A_2) \cdot A_3) \cdot ((A_4 \cdot A_5) \cdot A_6)
\]

APSP: \( O(n^3) \) algorithm.

TSP: \( 2^n \) time alg. \( \Rightarrow \) Better than brute-force \( O(n!) \) line.

Independent sets in Trees ...