1 Dynamic Programming
   - Longest Common Subsequence
   - Dynamic Programming Hallmark # 1: Optimal Substructure
   - Dynamic Programming Solution to LCS
   - Dynamic Programming Hallmark # 2: Overlapping subproblems
   - The 0/1 Integer Knapsack Problem
Dynamic Programming is a design technique like divide-and-conquer

Example: Longest Common Subsequence (LCS)

Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both:

\[
x: \quad A \quad B \quad C \quad B \quad D \quad A \quad B
\]
\[
y: \quad B \quad D \quad C \quad A \quad B \quad A
\]

\[
BCBA = \text{LCS}(x, y)
\]
Check every subsequence of \(x[1 \ldots m]\) to see if it is also a subsequence of \(y[1 \ldots n]\).

**Analysis:**

- There are \(2^m\) possible subsequences of \(x\), since each bit-vector of length \(m\) represents a distinct subsequence of \(x\).
- Checking each one of them into \(y\) takes \(O(n)\) time.
- So, worst-case running time is \(O(n \cdot 2^m)\).
- An exponential running time is impractical.
A Better Algorithm

Simplification:
- Look at the length of a longest common subsequence
- Extend the algorithm to find the LCS itself

Notation: Let $|s|$ denote the length of a sequence $s$

 Proposed Strategy: Consider prefixes of $x$ and $y$
- Define $c[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])|$  
- Then, $\text{LCS}(x, y) = c[m, n]$
Recursive Formulation

**Theorem:**

\[
c[i, j] = \begin{cases} 
  c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j] \\
  \max\{c[i - 1, j], c[i, j - 1]\} & \text{otherwise}
\end{cases}
\]

**Proof:** Case \( x[i] = y[j] \)

Let \( z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j]) \), where \( c[i, j] = k \). Then \( z[k] = x[i] \). Otherwise, \( z \) could be extended by \( x[i] \). Moreover, \( z[1 \ldots k - 1] = \text{LCS}(x[1 \ldots i - 1], y[1 \ldots j - 1]) \).
Claim: \( z[1 \ldots k - 1] = \text{LCS}(x[1 \ldots i - 1], y[1 \ldots j - 1]) \)

Proof of Claim by Contradiction:

- Suppose \( w \) is a longer common subsequence of \( x[1 \ldots i - 1] \) and \( y[1 \ldots j - 1] \). That is, \(|w| > k - 1\).

- Then, cut and paste: \( w \cdot z[k] \) (\( w \) concatenated by \( z[k] \)) is also a common subsequence of \( x[1 \ldots i] \) and \( y[1 \ldots j] \). Since \(|w \cdot z[k]| > k\), we have reached a contradiction, proving the above claim.

- So, \( c[i - 1, j - 1] = k - 1 \), which implies that \( c[i, j] = c[i - 1, j - 1] + 1 \).

Case 2 is proven with a similar argument.
Dynamic Programming: Hallmark # 1

**Optimal substructure**

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of $z$ is an LCS of a prefix of $x$ and a prefix of $y$. 
Recursive Algorithm for LCS

LCS(x, y, i, j)
1: if \(x[i] = y[j]\) then
2: \(c[i, j] \leftarrow \text{LCS}(x, y, i - 1, j - 1) + 1\)
3: else \(c[i, j] = \max\{\text{LCS}(x, y, i - 1, j), \text{LCS}(x, y, i, j - 1)\}\)

Worst-case: When \(x[i] \neq y[j]\), the algorithm evaluates two subproblems, each one with only one parameter decremented.
The height of the recursion tree is $m + n$. It seems that the work is exponential because we are solving the same subproblems over and over. We need to remember subproblems once we solve them!
Overlapping subproblems
A recursive solution contains a “small” number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths $m$ and $n$ is only $mn$. 
**Memoization Algorithm**

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS\((x, y, i, j)\)
1:  if \(c[i,j] = NIL\) then
2:    if \(x[i] = y[j]\) then
3:      \(c[i,j] \leftarrow \text{LCS}(x, y, i - 1, j - 1) + 1\)
4:    else \(c[i,j] = \max\{\text{LCS}(x, y, i - 1, j), \text{LCS}(x, y, i, j - 1)\}\)

**Running Time Analysis:** \(T(n, m) \in \theta(m \cdot n)\) since the amount of work per table entry is constant.

**Space Analysis:** \(S(n, m) \in \theta(m \cdot n)\) since we only store the table.
### Dynamic Programming Algorithm

**Idea:**

1. Fill the table top left to bottom right
2. \( T(n, m) \in \theta(m \cdot n) \)
3. Reconstruct the LCS by tracing backwards
4. \( S(n, m) \in \theta(m \cdot n) \)
5. Exercise: reduce \( S(n, m) \) to \( O(\min\{m, n\}) \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Another Dynamic Programming Problem

The 0/1 Integer Knapsack Problem
The 0/1 Integer Knapsack Problem

- Given $n$ objects
- Each object has an integer weight $w_i$ and integer profit $p_i$
- You have a knapsack with an integer weight capacity $M$
- Problem: Find the subset of $n$ objects that fits in the knapsack and gives the maximum total profit
Examples of Possible Solutions

Say the knapsack has capacity $M = 20$:

<table>
<thead>
<tr>
<th>Object</th>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit $p_i$</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Weight $w_i$</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>14</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Possible solutions:

- Put items 1-3 in knapsack: Total weight is 20, and profit is 25
- Put items 1, 2, 4, and 6: Total weight now is 19, profit is 32
- Other possible solutions ...

How long does it take to evaluate all feasible solutions?
Mathematical Formulation of the Optimization Problem

MAXIMIZE

\[ p_1 \cdot x_1 + p_2 \cdot x_2 + \ldots + p_n \cdot x_n \]

such that (SUBJECT TO CONSTRAINT)

\[ w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n \leq M \]

where \( x_i \in \{0, 1\} \) for \( i \in \{1, 2, \ldots, n\} \)
A Dynamic Programming Solution

Define $f_i(y)$ to be the optimal solution to the subproblem:

$$\underset{\text{MAXIMIZE}}{\text{MAXIMIZE}}\ p_1 \cdot x_1 + p_2 \cdot x_2 + \ldots + p_i \cdot x_i$$

such that $w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_i \cdot x_i \leq y$

where $x_j \in \{0, 1\}$ for $j \in \{1, 2, \ldots, i\}$

Then we see the optimal substructure of the solution:

$$f_i(y) = \begin{cases} 
\max\{f_{i-1}(y), p_i + f_{i-1}(y - w_i)\} & \text{if } y \geq w_i \\
 f_{i-1}(y) & \text{if } y < w_i 
\end{cases}$$
Seeing the Optimal Substructure

- $f_1(y) =$ the maximum profit for capacity $y$ considering only object 1, where $x_1 \in \{0, 1\}$
- $f_2(y) =$ the maximum profit for capacity $y$ considering only objects 1 and 2, where $x_1, x_2 \in \{0, 1\}$
- Consider what happens when we consider object 3:
  - If $x_3 = 0$, this means we do not choose to include object 3 in the knapsack. So, maximum profit is what it used to be using objects 1, 2: $f_3(y) = f_2(y)$
  - Else, we choose to include, which means we only have $y - w_3$ capacity for objects 1, 2:
    - We do not know a priori whether $x_3$ should be 0 or 1
    - The only criterion is that $f_3(y) = \max\{f_2(y), f_2(y - w_3)\}$
Computing $f_i(y)$

- The optimal substructure dictates that we compute $f_{i-1}(y)$ for all capacities $y \in \{0, 1, \ldots, M\}$
- The recursion shows it is only necessary to save $f_i(y)$ and $f_{i-1}(y)$ for all possible values of $y$
- Basic Idea:
  - Set $f_0(y) = 0$ $\forall y \in \{0, 1, \ldots, M\}$
  - Compute $f_1(y)$ $\forall y \in \{0, 1, \ldots, M\}$
  - ...
  - Compute $f_n(y)$ $\forall y \in \{0, 1, \ldots M\}$
Let $p = (7, 6, 12, 3, 12, 16)$, $w = (2, 8, 10, 4, 14, 5)$, and $M = 20$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>10</th>
<th>...</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>$f_1$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>7</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>$f_2$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>13</td>
<td>...</td>
<td>13</td>
</tr>
<tr>
<td>$f_3$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question: How big is the matrix that stores solutions to subproblems?
A Simpler Version of the Knapsack Problem

What if one can take portions of one item?