6.4 Knapsack Problem
Knapsack Problem

Knapsack problem.
- Given n objects and a "knapsack."
- Item i weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \{ 3, 4 \} has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\[ W = 11 \]

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \{ 5, 2, 1 \} achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Dynamic Programming: False Start

**Def.** $OPT(i) = \text{max profit subset of items } 1, \ldots, i.$

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{ 1, 2, \ldots, i-1 \}$

- **Case 2:** $OPT$ selects item $i$.
  - accepting item $i$ does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before $i$, we don't even know if we have enough room for $i$

**Conclusion.** Need more sub-problems!
Dynamic Programming: Adding a New Variable

Def. \( \text{OPT}(i, w) = \text{max profit subset of items 1, ..., i with weight limit } w \).

- **Case 1**: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \{ 1, 2, ..., i-1 \} using weight limit \( w \)

- **Case 2**: \( \text{OPT} \) selects item \( i \).
  - new weight limit = \( w - w_i \)
  - \( \text{OPT} \) selects best of \{ 1, 2, ..., i-1 \} using this new weight limit

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max\{ \text{OPT}(i-1, w), \ v_i + \text{OPT}(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

Input: n, w₁,…,w₈, v₁,…,v₈

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wᵢ > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vᵢ + M[i-1, w-wᵢ]}

return M[n, W]
Knapsack Algorithm

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**OPT:** \{ 4, 3 \}

value = 22 + 18 = 40

W = 11
Knapsack Problem: Running Time

**Running time.** $\Theta(nW)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

**Knapsack approximation algorithm.** There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]