Chapter 5, Dynamic Programming:
▶ Introduction and sample problems (this past Friday)
▶ Principles of DP (**Today**)
▶ DP algorithms, solutions to sample problems (Wednesday)
▶ Optimal BSTs (Friday)
▶ [Test 2, Wed Apr 6, *not* covering DP]

Today:
▶ Review of memoization
▶ Introduction of three problems (left over from previous class)
   ▶ 0-1 Knapsack
   ▶ Longest common subsequence
   ▶ Matrix multiplication
▶ Elements of dynamic programming
   ▶ Optimization problems
   ▶ Optimal substructure
   ▶ Dynamic programming algorithms
▶ Solution to the knapsack problem (time permitting)
Ex 6.5. Explain why this function can't use memoization:

idgen = -1

def make_unique_id(name):
    # global allows us to modify idgen inside this function
    global idgen
    idgen += 1
    return name + str(idgen)
Ex 6.6. Explain why this function can't use memoization:

def pick_at_random(seq):
    return seq[random.randint(0, len(seq)-1)]
Ex. 6.7. Explain why this function can't use memoization:

```python
def next_n_lines(n):
    lines = ''
    for i in range(n):
        lines += f.readline()
    return lines
```

```python
f = open('data', 'r')
```
Matrix multiplication.

\[
\begin{pmatrix}
2 & 8 \\
5 & 7 \\
\end{pmatrix}
\begin{pmatrix}
3 & 6 \\
1 & 4 \\
\end{pmatrix}
= 
\begin{pmatrix}
2 \cdot 3 + 8 \cdot 1 & 2 \cdot 6 + 8 \cdot 4 \\
5 \cdot 3 + 7 \cdot 1 & 5 \cdot 6 + 7 \cdot 4 \\
\end{pmatrix}
= 
\begin{pmatrix}
14 & 44 \\
22 & 58 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 3 & 12 \\
2 & 7 & 11 \\
\end{pmatrix}
\begin{pmatrix}
4 & 10 \\
8 & 6 \\
9 & 5 \\
\end{pmatrix}
= 
\begin{pmatrix}
1 \cdot 4 + 3 \cdot 8 + 12 \cdot 9 & 1 \cdot 10 + 3 \cdot 6 + 12 \cdot 5 \\
2 \cdot 4 + 7 \cdot 8 + 11 \cdot 9 & 2 \cdot 10 + 7 \cdot 6 + 11 \cdot 5 \\
\end{pmatrix}
= 
\begin{pmatrix}
136 & 88 \\
163 & 117 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 5 \\
6 & 8 & 9 \\
\end{pmatrix}
\begin{pmatrix}
3 \\
7 \\
4 \\
\end{pmatrix}
= 
\begin{pmatrix}
1 \cdot 3 + 2 \cdot 7 + 5 \cdot 4 & 6 \cdot 3 + 8 \cdot 7 + 9 \cdot 4
\end{pmatrix}
= 
\begin{pmatrix}
37 \\
110 \\
\end{pmatrix}
\]
Progression of dynamic-programming problems:

1. Problem statement . . . *recognizing optimal substructure*
2. Recursive characterization . . . *recognizing overlapping subproblems*
3. Dynamic programming algorithm
   Make a table for subproblems
   Initialize base cases in the table
   For all other subproblems / cells in the table
      For each option in the decision for that subproblem
         Lookup subsubproblem results and compare
         Record best choice for that subproblem
   Return minimum cost or maximum value for top-level problem
0-1 Knapsack.

Given a capacity \( c \) and the value and weight of \( n \) items in arrays \( V \) and \( W \), find a subset of the \( n \) items whose total weight is less than or equal to the capacity and whose total value is maximal.

<table>
<thead>
<tr>
<th>( V )</th>
<th>20</th>
<th>15</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

\( c = 7 \)

- \( \{2, 3\} \) weight 9 value 190 exceeds capacity
- \( \{1, 3\} \) weight 7 value 115 not optimal
- \( \{0, 1, 2\} \) weight 7 value 125 optimal
Knapsack
Let $B[i][j]$ be the value of the best way to fill remaining knapsack capacity $i$ using only items 0 through $j$. Then $B[c][n − 1]$ is the value-solution to the entire problem, that is,

$$B[c][n − 1] = \max_K \sum_{j=0}^{n-1} K[j]V[j]$$

In the general case we have the choice between

$$V[j] + B[i − W[j]][j − 1]$$

**value of the $j$th item**

**remaining capacity after taking the $j$th item**

The best way to fill the remaining capacity with the remaining items

versus

$$B[i][j − 1]$$

**The best way to fill the unchanged capacity with the remaining items**
Knapsack

\[ B[i][j] = \begin{cases} 
0 & \text{if } j = 0 \text{ and } W[0] > i \quad (0\text{th doesn’t fit}) \\
V[0] & \text{if } j = 0 \text{ and } W[0] \leq i \quad (0\text{th fits}) \\
B[i][j - 1] & \text{if } W[j] > i \quad (j\text{th doesn’t fit}) \\
\max \left\{ V[j] + B[i - W[j]][j - 1], B[i][j - 1] \right\} & \text{otherwise} \quad (j \text{ fits}) 
\end{cases} \]
Coming up:

*Catch up on projects...*

_Do Traditional RB project (suggested by Mon, Mar 28)_
_(Recommended: Do LL RB project for your own practice)_

**Due Mon, Mar 28 (end of day)**
_Read Section 6.3_
_Do Exercises 6. (16, 19, 23, 33)_
_Take quiz Tues, Mar 29_

**Due Wed, Mar 30 (classtime)**
_Read Section 6.4_
_Do Exercises 6. (20, 24, 32)_

**Due Fri, Apr 1 (end of day)**
_Do Project 6.1.b as a practice problem_
_Take quiz_