6.006 Lecture 19: Dynamic Programming I

- Rod-Cutting Problem
- Top-down & Bottom-up DP
- General characteristics of problems that DP can solve

CLRS 15.1 - 15.4
Rod-Cutting Problem

- A metal rod of length $n$
- We can cut it into integer-size pieces

$n = 9$

- A piece of size $i$ sells for $p_i$ dollars
- Cutting costs us nothing

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>11</td>
<td>7</td>
<td>15</td>
<td>15</td>
<td>...</td>
</tr>
</tbody>
</table>

- Goal: maximize revenue $r_i$ from a rod of size $n$ (by cutting it up)

\[
\begin{align*}
    r_1 &= 3 \quad (= p_1) \\
    r_2 &= 6 \quad (p_1 + p_1) \\
    r_3 &= 10 \quad (p_3) \\
    r_4 &= 13 \quad (p_3 + p_1) \\
    r_5 &= 16 \quad (p_3 + p_1 + p_1 = r_4 + p_1) \\
    r_6 &= 20 \quad (p_3 + p_3)
\end{align*}
\]
Algorithm Idea

- Consider the leftmost piece and recurse

\[ n = 9 \]

- Iterate over all possible sizes for leftmost piece:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \]

\[ r_0 = 0 \]

Possible implementation:

```python
def r(p, n):
    if n == 0:
        return 0
    ans = p[n]  # No cutting; \( r_n \leq p_n + r_0 \)
    for i in range(1, n):
        ans = max(ans, p[i] + r(p, n-i))
    return ans
```

Correct but slowuuuuu!!!
why is this slow?

\[ T(n) = \# \text{ calls to } r(p, n) \]

\[ T(1) = 1 \quad \text{never makes a recursive call} \]

\[ T(n) = 1 + \sum_{i=1}^{n-1} T(i) \]

\[ r(p, n) \quad r(p, n-i) \quad \text{in the loop} \]

\[
\begin{array}{cccccccc}
 n & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots \\
 T(n) & 1 & 2 & 4 & 8 & 16 & 32 & 64 & \ldots \\
\end{array}
\]

\[ T(n) = 2^n \]

Inefficient because we recompute same \( r_i \)'s!

Solution: memorize (or build up)

\[ \text{memo} = \{3\} \]

\[ \text{def } r(p, n): \]
\[ \quad \text{if } n \text{ in memo: return memo}[n] \]
\[ \quad \text{ans} = p[n] \]
\[ \quad \text{for } i \text{ in range(1, } n): \]
\[ \quad \quad \text{ans} = \max(\text{ans}, p[i] + r(p, n-i)) \]
\[ \quad \text{memo}[n] = \text{ans} \]
\[ \quad \text{return } \text{ans} \]

"top-down"

\[ \text{running time} = \Theta(n^2) \]
OR:
\[ r = [0] + (n+1) \# a list! \]
for \( k \) in range \((1, n+1)\):
    \[ \text{ans} = p[k] \]
    for \( i \) in range \((1, k)\):
        \[ \text{ans} = \max(\text{ans}, p[i] + r[k-i]) \]
    \[ r[k] = \text{ans} \]

"bottom-up",
also \( \Theta(n^2) \)

Subproblem dependence graph

vertices: \# subproblems \( r_i \)
edges: \( A \rightarrow B \) if
    solving \( A \) requires
    solving \( B \) first

Top-down (+ memorization): DFS on this graph
Bottom-Up: solve in reverse topological order

both \( \Theta(n+E) \)
but don't build the graph! 
Characteristics of Dynamic Programming:

- Many related subproblems
- Optimal solution for one (sub)problem requires optimal solution of smaller subproblems contains
- Different subproblems benefit from knowing solution to same subproblem (so memorize)