Admin: Quiz 2 next Wed 11/12 in room 1-190, 7-9 pm

Readings: CLRS 15.1-15.4

Outline: Dynamic Programming (1/4)

- "Rod-cutting" problem
- Dynamic programming approaches:
  - top-down + memoization
  - bottom-up (smallest subproblems first)
Rod Cutting

We have a supplier who gives a rod of length, say, n.
(n may vary from rod to rod).
We want to cut rod into pieces & sell pieces for maximum revenue.
(Pieces have integer sizes 1, 2, ...)
Market determines price: let \( p_i \) = price for a piece of length \( i \);
(Assume cost of doing cuts is negligible)

\[
\begin{align*}
  i & | 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad \ldots \\
  p_i & | 3 \quad 4 \quad 10 \quad 11 \quad 7 \quad 15 \quad 15 \quad \ldots
\end{align*}
\]

Let \( r_n \) = maximum revenue we can get by cutting
up a piece of size \( n \):
\[
\begin{align*}
  r_1 & = 3 \quad (\text{no cutting possible} \implies \text{price } p_1) \\
  r_2 & = 6 \quad (3 + 3 \times 2 = r_1 + r_1) \\
  r_3 & = 10 \quad (p_3) \\
  r_4 & = 13 \quad (p_4 + p_3) \\
  r_5 & = 16 \quad (r_1 + r_4 = p_1 + p_1 + p_3) \\
  r_6 & = 20 \quad (p_3 + p_3) \\
  r_7 & = 23 \quad (p_3 + p_3 + p_1)
\end{align*}
\]

Given: \( p_1, p_2, \ldots \)
Compute: \( r_1, r_2, \ldots \)
How?
Not too hard:

\[ r_n = \max \left( p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_i \right) \]

cut into \( i \) \& \( n-i \)

& recurse on pieces

Suffices to consider only leftmost cut.

\[ r_0 = 0 \]

\[ r_n = \max \left( p_i + r_{n-i} \right) \]

\( \leq i \leq n \)

\( i \) \& \( n-i \)

only recurse on right-hand piece, since we are considering left-most cut position

Features to note: (characteristics of dyn. prog. problems in general):

- many related subproblems (e.g., \( r_1, r_2, \ldots \)) all of same type
- computing optimal solution for one (e.g., \( r_n \)) requires consideration of smaller subproblems: "optimal substructure"
- "overlapping subproblems": both \( r_6 \) and \( r_7 \) benefit from knowing solution to \( r_3 \), for example.
- a bit like divide \& conquer, except there is some optimization (i.e., maximization or minimization) involved; we are trying to find best way to divide problem up into subproblems, rather than just taking a fixed division.
First cut at python code:

```python
def r(p, n):
    if n == 0: return 0
    ans = p[n]
    for i in range(1, n):
        ans = max(ans, p[i] + r(p, n-i))
    return ans
```

Gives correct answers, but...

Slow (r(p, 27) takes ≈ 1 minute...) (dies for n > 30...)

Why?

Consider r₅:

---

Complexity $T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\sum_{i | n} T(i) & \text{if } n > 1 
\end{cases}$ = # calls

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n)$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

$T(n) = 2^n$ (prove it!)
But: we are re-computing some things over and over... there are only
n distinct problems here: \( r_1, \ldots, r_n \)!
Only compute each subproblem once:

Two ways of doing this, both save previously computed answers:

1. Top-down + "memoization" \( \{ \text{dynamic} \} \)
2. Bottom-up + array of pre-computed values \( \{ \text{programming} \} \)

1. Top-down + memoization

```python
def \( r(p, n) \):
    if \( n \) in memo: return memo[\( n \)]
    if \( n == 0 \): return 0
    \( ans = p[\( n \)] \)
    for \( i \) in range(1, \( n \)):
        ans = max(ans, \( p[i] + r(p, n-i) \))
    memo[\( n \)] = ans
    return ans
```

Complexity = \( \Theta(n^2) \) to compute \( r_n \)
2. Bottom-up: compute \( r_1, \ldots, r_n \), ...

\[
\begin{align*}
\text{for } k \text{ in range (1, n+1)}: \\
\quad \text{ans} &= r[k] \\
\quad \text{for } i \text{ in range (1, k)}: \\
\quad \quad \text{ans} &= \max(\text{ans}, \rho[i] + r[k-i]) \\
\quad r[k] &= \text{ans}
\end{align*}
\]

\( \text{time} = \Theta(n^2) \)

"Subproblem dependence graph"

vertices = subproblems to solve
edges = dependence between subproblems

\[ A \rightarrow B \]

solving A requires solving B first...
Note:

* Top-down + memoization is really just DFS on subproblem dependence graph
  \[ \text{time} = \Theta(V+E) \]

* Bottom-up method is just solving problems in order determined by reverse of topological sort
  (i.e. solve subproblems in order of increasing size...)
  \[ \text{time is still } \Theta(V+E) \]

Setting up subproblem dependence graph is good first step in working on D.P. problem.

Exercise: how to reconstruct optimal solution?

D.P. buzzwords to remember:

* subproblem dependence graph
* optimal substructure
* overlapping subproblems