Dynamic Programming

CIS 315
“We now turn to the two sledgehammers of the algorithms craft, dynamic programming and linear programming, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.”

-- Dasgupta, Papadimitriou, Vazirani

comments:
- one point is that dynamic programming is not “elegant”
- we won’t be covering linear programming here in 315
dynamic programming manifesto

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from the computed information.
rod cutting example

Given a rod of length $n$ inches and a table of prices $p_i$ for $i=1,2,...,n$ ($p_i$ is the price charged for a piece of length $i$ inches), determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces.

<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>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

with $n=10$
- two pieces of length 5: $p_5+p_5 = 20$
- no cuts: $p_{10}=30$
- six and four inches: $p_4+p_6 = 9+17 = 26$
crucial requirement

optimal substructure

For $r_n$, in an optimal solution, consider the first cut of length $i$. In that optimal solution, the remaining $n-i$ inches must be cut optimally, obtaining revenue $r_{n-i}$. 
first two steps of solution

**subproblem:**

\[ r_i \text{ is the maximum revenue obtainable from a rod of } i \text{ inches} \]

\[ (i=0,1,2,\ldots,n) \]

**recurrence:**

look at all possible first cuts:

\[ r_0 = 0 \text{ (base case)} \]

\[ r_n = \max_{1 \leq i \leq n} [p_i + r_{n-i}] \]
# fill out a table

<table>
<thead>
<tr>
<th>$r_0$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
<th>$r_9$</th>
<th>$r_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**using recurrence:** $r_1 = \text{MAX}[p_1+r_0] = 1+0 = 1$

<table>
<thead>
<tr>
<th>$r_0$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
<th>$r_9$</th>
<th>$r_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**$r_2 = \text{MAX}[p_1+r_1, p_2+r_0] = \text{MAX}[1+1, 5+0] = 5$**

<table>
<thead>
<tr>
<th>$r_0$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
<th>$r_9$</th>
<th>$r_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
$r_3 = \text{MAX}[p_1+r_2, p_2+r_1, p_3+r_0] = \text{MAX}[1+5, 5+1, 8+0] = 8$
$r_4 = \text{MAX}[p_1+r_3, p_2+r_2, p_3+r_1, p_4+r_0] = \text{MAX}[1+8, 5+5, 8+1, 9+0] = 10$

<table>
<thead>
<tr>
<th>\text{r}_0</th>
<th>\text{r}_1</th>
<th>\text{r}_2</th>
<th>\text{r}_3</th>
<th>\text{r}_4</th>
<th>\text{r}_5</th>
<th>\text{r}_6</th>
<th>\text{r}_7</th>
<th>\text{r}_8</th>
<th>\text{r}_9</th>
<th>\text{r}_{10}</th>
</tr>
</thead>
</table>
|   0   |   1   |   5   |   8   |   10  |\n
$r_5 = \text{MAX}[p_1+r_4, p_2+r_3, p_3+r_2, p_4+r_1, p_5+r_0] = \text{MAX}[1+10, 5+8, 8+5, 9+1, 10+0] = 13$
$r_6 = \text{MAX}[p_1+r_5, p_2+r_4, p_3+r_3, p_4+r_2, p_5+r_1, p_6+r_0] = \text{MAX}[1+13, 5+10, 8+8, 9+5, 10+1, 17+0] = 17$

<table>
<thead>
<tr>
<th>\text{r}_0</th>
<th>\text{r}_1</th>
<th>\text{r}_2</th>
<th>\text{r}_3</th>
<th>\text{r}_4</th>
<th>\text{r}_5</th>
<th>\text{r}_6</th>
<th>\text{r}_7</th>
<th>\text{r}_8</th>
<th>\text{r}_9</th>
<th>\text{r}_{10}</th>
</tr>
</thead>
</table>
|   0   |   1   |   5   |   8   |   10  | 13    | 17    |\n

\text{clearly an O(n^2) process this way}
array r[0...n] of int

r[0] = 0

for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max[ q, p[i]+r[j-i] ]
    r[j] = q

return r[n]
versus naïve recursive version

\[
\text{CutRod}(p,n) \\
\quad \text{if } n=0 \text{ return 0} \\
\quad q = -\infty \\
\quad \text{for } i = 1 \text{ to } n \\
\quad \quad q = \text{MAX}[ q, p[i]+\text{CutRod}(p, n-i) ] \\
\quad \text{return } q
\]

\(O(2^n)\) time according to text so memoize it
other example: Fibonacci Numbers

```python
def recFib (int n):
    if n==0 return 0
    if n==1 return 1
    n1 = recFib(n-1)
    n2 = recFib(n-2)
    return n1+n2

def:
F_0 = 0, F_1 = 1
F_k = F_{k-1} + F_{k-2}
```

**pure recursion:**
lots of recomputation, so time is exponential in \( n \)
memoized Fibonacci Numbers

```java
memoFib (int n)
    if n=0 return 0
    if n=1 return 1
    if F[n] = -1
        n1 = memoFib(n-1)
        n2 = memoFib(n-2)
        F[n] = n1+n2
    return F[n]
```

uses a global array F initialized to -1
- -1 means “not seen”
- could also use hash map

also called “top-down” based on the call structure

memoized version:
still uses recursion but stores computed values in a cache to avoid recomputation
iterative Fibonacci

```cpp
iterFib (int n)
    F[0] = 0
    F[1] = 1
    for i=2 to n
        F[i] = F[i-1]+F[i-2]
    return F[n]
```

stores values in array F; obviously does not need previous values, could just use the last two instead of whole array

also called “bottom-up”

**iterative version:** since all values are computed in other version, better to compute directly, avoids recursion overhead