From Last Time

How many ways are there to tile a $2 \times n$ board with dominoes?

How many ways to tile this:  

With these?
Today’s Keywords

• Dynamic Programming
• Log Cutting
CLRS Readings

• Chapter 15
Homework

• Hw4 Due Tonight at 11pm
  – Sorting
  – Written
Midterm

• Wednesday March 6 in class
  – Covers all content through last Monday
  – We will have a review session
    • Tonight! 7pm, Olsson 120
    • Will be recorded, so you’ll have it if you can’t make it
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:

\[ \text{Tile}(n) = \text{Tile}(n - 1) + \text{Tile}(n - 2) \]

\[ \text{Tile}(0) = \text{Tile}(1) = 1 \]
How to compute $Tile(n)$?

Tile(n):
  if n < 2:
    return 1
  return Tile(n-1)+Tile(n-2)

Problem?
Recursion Tree

Many redundant calls!
Run time: $\Omega(2^n)$

Better way: Use Memory!
Computing $Tile(n)$ with Memory

Initialize Memory $M$

$\text{Tile}(n)$:

if $n < 2$:
    return 1

if $M[n]$ is filled:
    return $M[n]$

$M[n] = \text{Tile}(n-1)+\text{Tile}(n-2)$

return $M[n]$
Computing $\text{Tile}(n)$ with Memory “Top Down”

Initialize Memory $M$

$\text{Tile}(n)$:

if $n < 2$:
    return 1

if $M[n]$ is filled:
    return $M[n]$

$M[n] = \text{Tile}(n-1)+\text{Tile}(n-2)$

return $M[n]$
Dynamic Programming

• Requires **Optimal Substructure**
  – Solution to larger problem contains the solutions to smaller ones

• Idea:
  1. Identify recursive structure of the problem
    • What is the “last thing” done?

\[
\begin{align*}
    n - 1 & \\
    n - 2 & \\
\end{align*}
\]
def myDCalgo(problem):

    if baseCase(problem):
        solution = solve(problem)

        return solution

    for subproblem in problem:  # After dividing
        subsolutions.append(myDCalgo(subproblem))

    solution = Combine(subsolutions)

    return solution
Generic Top-Down Dynamic Programming Soln

```python
mem = {}  
def myDPalgo(problem):
    if mem[problem] not blank:
        return mem[problem]
    if baseCase(problem):
        solution = solve(problem)
        mem[problem] = solution
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem))
    solution = OptimalSubstructure(subsolutions)
    mem[problem] = solution
    return solution
```
Computing $Tile(n)$ with Memory “Top Down”

Initialize Memory $M$

$Tile(n)$:

if $n < 2$:
    return 1

if $M[n]$ is filled:
    return $M[n]$

$M[n] = Tile(n-1)+Tile(n-2)$

return $M[n]$

Recursive calls happen in a predictable order
Better *Tile*(n) with Memory
“Bottom Up”

**Tile(n):**

Initialize Memory M

M[0] = 1
M[1] = 1

for i = 2 to n:

\[ M[i] = M[i-1] + M[i-2] \]

return M[n]
Dynamic Programming

• Requires **Optimal Substructure**
  – Solution to larger problem contains the solutions to smaller ones

• Idea:
  1. Identify recursive structure of the problem
     • What is the “last thing” done?
  2. Select a good order for solving subproblems
     • Usually smallest problem first
     • “Bottom up”
Log Cutting

Given a log of length $n$
A list (of length $n$) of prices $P$ ($P[i]$ is the price of a cut of size $i$)
Find the best way to cut the log

Select a list of lengths $\ell_1, \ldots, \ell_k$ such that:
$\sum \ell_i = n$
to maximize $\sum P[\ell_i]$  
Brute Force: $O(2^n)$
Greedy won’t work

- **Greedy algorithms** (next unit) build a solution by picking the best option “right now”
  - Select the most profitable cut first

<table>
<thead>
<tr>
<th>Length</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

**Greedy:** Lengths: 5, 1
Profit: 51

**Better:** Lengths: 2, 4
Profit: 54
Greedy won’t work

- **Greedy algorithms** (next unit) build a solution by picking the best option “right now”
  - Select the “most bang for your buck”
    - (best price / length ratio)

<table>
<thead>
<tr>
<th>Length</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

**Greedy**: Lengths: 5, 1
Profit: 51

**Better**: Lengths: 2, 4
Profit: 54
Dynamic Programming

• Idea:

1. Identify recursive structure of the problem
   • What is the “last thing” done?
2. Select a good order for solving subproblems
   • Usually smallest problem first
   • “Bottom up”
1. Identify Recursive Structure

\[ P[i] = \text{value of a cut of length } i \]
\[ \text{Cut}(n) = \text{value of best way to cut a log of length } n \]

\[ \text{Cut}(n) = \max \begin{cases} 
\text{Cut}(n - 1) + P[1] \\
\text{Cut}(n - 2) + P[2] \\
\vdots \\
\text{Cut}(0) + P[n] 
\end{cases} \]
Dynamic Programming

• Idea:
  1. Identify recursive structure of the problem
    • What is the “last thing” done?
  2. Select a good order for solving subproblems
    • Usually smallest problem first
    • “Bottom up”
2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[ \text{Cut}(0) = 0 \]
2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[ \text{Cut}(1) = \text{Cut}(0) + P[1] \]
2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[
Cut(2) = \max \left\{ \begin{array}{ll}
Cut(1) + P[1] \\
Cut(0) + P[2]
\end{array} \right.
\]

Cut(i):

<table>
<thead>
<tr>
<th></th>
<th>0</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>
</table>
Length: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Cut(2):

- Red: 1
- Green: 2

Length: 2
2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[ \text{Cut}(3) = \max \left\{ \text{Cut}(2) + P[1], \text{Cut}(1) + P[2], \text{Cut}(0) + P[3] \right\} \]

<table>
<thead>
<tr>
<th>Cut(i):</th>
<th>0</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>Length:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Length: 3

Cut(3) = \max \left\{ \text{Cut}(2) + P[1], \text{Cut}(1) + P[2], \text{Cut}(0) + P[3] \right\}

Cut(3) = \max \left\{ \text{Cut}(2) + P[1], \text{Cut}(1) + P[2], \text{Cut}(0) + P[3] \right\}

Cut(3) = \max \left\{ \text{Cut}(2) + P[1], \text{Cut}(1) + P[2], \text{Cut}(0) + P[3] \right\}
2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[
\text{Cut}(4) = \max \left\{ \begin{array}{l}
\text{Cut}(3) + P[1] \\
\text{Cut}(2) + P[2] \\
\text{Cut}(1) + P[3] \\
\text{Cut}(0) + P[4]
\end{array} \right. 
\]

<table>
<thead>
<tr>
<th>Cut(i):</th>
<th>0</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>Length</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

\[
\text{Cut}(4) = \max \left\{ 4 + P[3], 3 + P[2], 2 + P[1], 1 + P[0] \right. 
\]

4
Log Cutting Pseudocode

Initialize Memory C

Cut(n):

\[ C[0] = 0 \]

for i = 1 to n:

\[ \text{best} = 0 \]

for j = 1 to i:

\[ \text{best} = \max(\text{best}, \ C[i-j] + P[j]) \]

\[ C[i] = \text{best} \]

return C[n]

Run Time: \( O(n^2) \)
How to find the cuts?

• This procedure told us the profit, but not the cuts themselves
• Idea: remember the choice that you made, then backtrack
Remember the choice made

Initialize Memory C, Choices

$\text{Cut}(n)$:

$C[0] = 0$

for $i=1$ to $n$:

$\text{best} = 0$

for $j = 1$ to $i$:

if $\text{best} < C[i-j] + P[j]$:  

$\text{best} = C[i-j] + P[j]$

$\text{Choices}[i]=j$  \text{Gives the size of the last cut}$

$C[i] = \text{best}$

return $C[n]$
Reconstruct the Cuts

• Backtrack through the choices
Backtracking Pseudocode

```python
i = n
while i>0:
    print Choices[i]
    i = i - Choices[i]
```
Dynamic Programming

• Requires **Optimal Substructure**
  – Solution to larger problem contains the solutions to smaller ones

• Idea:
  1. Identify recursive structure of the problem
     • What is the “last thing” done?
  2. Select a good order for solving subproblems
     • Usually smallest problem first
     • “Bottom up”