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 $\text{Tile}(n)$?

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 $\text{Tile}(n)$ with Memory

Initialize Memory $M$
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 \( M \)

\[
\text{Tile}(n): \\
\begin{cases} 
\text{if } n < 2: \\
\quad \text{return } 1 \\
\text{if } M[n] \text{ is filled:} \\
\quad \text{return } M[n] \\
\quad M[n] = \text{Tile}(n-1) + \text{Tile}(n-2) \\
\text{return } M[n] 
\end{cases}
\]

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?

Generic Divide and Conquer Solution

```python
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem)
        return solution
    for subproblem of problem:
        if 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
```
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

<table>
<thead>
<tr>
<th>Price</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Select a list of lengths \( \ell_1, ..., \ell_k \) 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>Price</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Greedy: Lengths 1, 1
Profit: 11
Better: Lengths 2, 8
Profit: 16
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>Price</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Greedy:** Lengths 4, 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

\[ P[i] = \text{value of a cut of length } i \]
\[ \text{Cut}(n) = \text{value of best way to cut a log of length } n \]
\[ \text{Cut}(n) = \max \left\{ \begin{array}{l}
\text{Cut}(n-1) + P[1] \\
\text{Cut}(n-2) + P[2] \\
\vdots \\
\text{Cut}(0) + P[n] 
\end{array} \right\} \]

Best way to cut a log of length \( n \)
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

\[ \text{Cut}(0) = 0 \]

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

Solve Smallest subproblem first

\[
\text{Cut}(0) = \max \left\{ \text{Cut}(1) + p_1, \text{Cut}(0) + p_2 \right\}
\]

Length: 0.0%

2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[
\text{Cut}(1) = \max \left\{ \text{Cut}(2) + p_1, \text{Cut}(1) + p_2, \text{Cut}(0) + p_3 \right\}
\]

Length: 0.0%

2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

\[
\text{Cut}(2) = \max \left\{ \text{Cut}(3) + p_1, \text{Cut}(2) + p_2, \text{Cut}(1) + p_3, \text{Cut}(0) + p_4 \right\}
\]

Length: 0.0%
Log Cutting Pseudocode

Initialize Memory $C$

\text{Cut}(n):$
\begin{align*}
C[0] &= 0 \quad \text{Run Time: } O(n^2) \\
\text{for } i=1 \text{ to } n: \\
&\quad \text{best} = 0 \\
&\quad \text{for } j = 1 \text{ to } i: \\
&\quad \quad \text{best} = \max(\text{best}, C[i-j] + P[j]) \\
&\quad C[i] = \text{best} \\
&\quad \text{return } C[n]
\end{align*}

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):$
\begin{align*}
C[0] &= 0 \\
\text{for } i=1 \text{ to } n: \\
&\quad \text{best} = 0 \\
&\quad \text{for } j = 1 \text{ to } i: \\
&\quad \quad \text{if } \text{best} < C[i-j] + P[j]: \\
&\quad \quad \quad \text{best} = C[i-j] + P[j] \\
&\quad \quad \quad \text{Choices}[j]=i \\
&\quad C[i] = \text{best} \\
&\quad \text{return } C[n]
\end{align*}

Gives the size of the last cut
Reconstruct the Cuts

- Backtrack through the choices

```
Backtracking Pseudocode

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"