Warm Up

How many arithmetic operations are required to multiply a $n \times m$ Matrix with a $m \times p$ Matrix?

(don’t overthink this)
Warm Up

How many arithmetic operations are required to multiply a $n \times m$ Matrix with a $m \times p$ Matrix?
(don’t overthink this)

- $m$ multiplications and additions per element
- $n \cdot p$ elements to compute
- Total cost: $m \cdot n \cdot p$
Dynamic Programming

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

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

\[
\begin{align*}
\text{Til}_n &= \text{Til}_{n-1} + \text{Til}_{n-2} \\
\text{Til}_0 &= \text{Til}_1 = 1
\end{align*}
\]
How to compute $Tile(n)$?

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

Problem?
Many redundant calls!
Run time: $\Omega(2^n)$
Better way: Use Memory!
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 $M[n]$
Dynamic Programming

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

• Idea:
  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. Save the solution to each subproblem in memory
Top-Down Dynamic Programming Template

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 (optimal) solutions to smaller ones

• Idea:
  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
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>1</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>17</th>
<th>17</th>
<th>20</th>
<th>24</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length:</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>

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

Greedy: Lengths: 5, 1
Profit: 51

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

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

• Idea:

  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
1. Identify Recursive Structure

\[ P[i] = \text{value of a cut of length } i \]

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

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

\[ \text{best way to cut a log of length } n - \ell_k \]

\[ \ell_k \text{ Last Cut} \]
Dynamic Programming

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

• Idea:
  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
Top-Down Log Cutting

```python
mem = {}
def best_cut(length):
    if mem[length] not blank:
        return mem[length]
    if length == 0:
        solution = 0
        mem[length] = solution
        return solution
    for shorter ≤ length:
        subsolutions[shorter] = best_cut(shorter)
        solution = max{subsolutions[shorter] + price[shorter]}
        mem[length] = solution
        return solution
```
3. Bottom-up: solve subproblems in order

Solve Smallest subproblem first

\[ \text{Cut}(0) = 0 \]
3. Bottom-up: solve subproblems in order

Solve Smallest subproblem first

\[
\text{best_cut}(1) = \text{best_cut}(0) + P[1]
\]
3. Bottom-up: solve subproblems in order

Solve Smallest subproblem first

\[
best\_cut(2) = \max \begin{cases} 
best\_cut(1) + P[1] 
\end{cases} \]

\[
best\_cut(0) + P[2] 
\]
3. Bottom-up: solve subproblems in order

Solve Smallest subproblem first

\[ \text{best}_\text{cut}(3) = \max \begin{cases} \text{best}_\text{cut}(2) + P[1] \\ \text{best}_\text{cut}(1) + P[2] \\ \text{best}_\text{cut}(0) + P[3] \end{cases} \]
3. Bottom-up: solve subproblems in order

Solve Smallest subproblem first

\[
\text{best\_cut}(4) = \max \begin{cases} 
\text{best\_cut}(3) + P[1] \\
\text{best\_cut}(2) + P[2] \\
\text{best\_cut}(1) + P[3] \\
\text{best\_cut}(0) + P[4] 
\end{cases}
\]

Length:

<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>
Bottom-Up Log Cutting Pseudocode

```python
mem = {}
def best_cut(length):
    mem[0] = 0
    for sub=1 to length:  // smallest subproblem first
        best = 0
        for last-cut = 1 to sub:  //last cut
            best = max(best, mem[i-j] + price[j])
        mem[i] = 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
Top-Down Log Cutting

```python
mem = {}
choices = {}
def best_cut(length):
    if mem[length] not blank:
        return mem[length]
    if length == 0:
        solution = 0
        mem[length] = solution
        return solution
    for last_cut ≤ length:
        subsolutions[length- last_cut] = best_cut(length- last_cut)
        solution = max{subsolutions[length- last_cut] + price[last_cut]}
        mem[length] = solution
        choices[length] = best last_cut
    return solution
```

Gives the size of the last cut
Reconstruct the Cuts

• Backtrack through the choices

For a log of length 10, the best choice for the last cut’s length is 3.
Backtracking Pseudocode

\[ i = n \]

while \( i > 0 \):
    print choices[i]
    \[ i = i - \text{choices}[i] \]
Matrix Chaining

• Given a sequence of Matrices \((M_1, \ldots, M_n)\), what is the most efficient way to multiply them?
Order Matters!

\[ c_1 = r_2 \]
\[ c_2 = r_3 \]

\[
\begin{align*}
(M_1 \times M_2) \times M_3 &= (c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3 \text{ operations}
\end{align*}
\]
Order Matters!

\[ c_1 = r_2 \]
\[ c_2 = r_3 \]

\[ M_1 \times (M_2 \times M_3) \]

- uses \( c_1 \cdot r_1 \cdot c_3 + (c_2 \cdot r_2 \cdot c_3) \) operations
Order Matters!

c_1 = r_2
c_2 = r_3

- \((M_1 \times M_2) \times M_3\)
  - uses \((c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3\) operations
  - \((10 \cdot 7 \cdot 20) + 20 \cdot 7 \cdot 8 = 2520\)

- \(M_1 \times (M_2 \times M_3)\)
  - uses \(c_1 \cdot r_1 \cdot c_3 + (c_2 \cdot r_2 \cdot c_3)\) operations
  - \(10 \cdot 7 \cdot 8 + (20 \cdot 10 \cdot 8) = 2160\)

\[M_1 = 7 \times 10\]
\[M_2 = 10 \times 20\]
\[M_3 = 20 \times 8\]
\[c_1 = 10\]
\[c_2 = 20\]
\[c_3 = 8\]
\[r_1 = 7\]
\[r_2 = 10\]
\[r_3 = 20\]
Dynamic Programming

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

• Idea:

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

2. Save the solution to each subproblem in memory

3. Select a good order for solving subproblems
   • “Top Down”: Solve each recursively
   • “Bottom Up”: Iteratively solve smallest to largest
1. Identify the Recursive Structure of the Problem

\[ \text{Best}(1,n) = \text{cheapest way to multiply together} \ M_1 \ \text{through} \ M_n \]
1. Identify the Recursive Structure of the Problem

\[ Best(1,n) = \text{cheapest way to multiply together } M_1 \text{ through } M_n \]

\[ Best(1,2) + Best(2,4) + r_1 r_2 c_4 \]

\[ Best(1,4) = \min \]

\[ M_1 \times M_2 \times M_3 \times M_4 \]
1. Identify the Recursive Structure of the Problem

\[ \text{Best}(1,n) = \text{cheapest way to multiply together } M_1 \text{ through } M_n \]

\[ \text{Best}(1,4) = \min \left\{ \text{Best}(1,2) + \text{Best}(2,4) + r_1 r_2 c_4, \text{Best}(1,2) + \text{Best}(3,4) + r_1 r_3 c_4 \right\} \]
1. Identify the Recursive Structure of the Problem

Best(1, n) = cheapest way to multiply together $M_1$ through $M_n$

Best(1, 4) = min

\[
\begin{align*}
\text{Best}(1, 2) + \text{Best}(2, 4) + r_1 r_2 c_4 \\
\text{Best}(1, 2) + \text{Best}(3, 4) + r_1 r_3 c_4 \\
\text{Best}(1, 3) + r_1 r_4 c_4
\end{align*}
\]
1. Identify the Recursive Structure of the Problem

• In general:

\[
Best(i, j) = \text{cheapest way to multiply together } M_i \text{ through } M_j
\]

\[
Best(i, j) = \min_{k=i}^{j-1}(Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j)
\]

\[
Best(i, i) = 0
\]

\[
Best(1, n) = \min \begin{cases} 
Best(1,2) + Best(2,4) + r_1 r_2 c_4 \\
Best(1,2) + Best(3, n) + r_1 r_3 c_n \\
Best(1,3) + Best(4, n) + r_1 r_4 c_n \\
Best(1,4) + Best(5, n) + r_1 r_5 c_n \\
\vdots \\
Best(1, n - 1) + r_1 r_n c_n
\end{cases}
\]
Dynamic Programming

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

• Idea:
  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. **Save the solution to each subproblem in memory**
  3. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
2. Save Subsolutions in Memory

• In general:

\[
Best(i, j) = \text{cheapest way to multiply together } M_i \text{ through } M_j
\]

\[
Best(i, j) = \min_{k=i}^{j-1} \left( Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j \right)
\]

\[
Best(i, i) = 0
\]

\[
Best(1, n) = \min \left( Best(2, n) + r_1 r_2 c_n, \right. \]
\[
\left. Best(1,2) + Best(3, n) + r_1 r_3 c_n, \right.
\]
\[
\left. Best(1,3) + Best(4, n) + r_1 r_4 c_n, \right.
\]
\[
\left. Best(1,4) + Best(5, n) + r_1 r_5 c_n, \right\ldots
\]
\[
\left. Best(1, n - 1) + r_1 r_n c_n \right)
\]
Dynamic Programming

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

• Idea:
  1. Identify the recursive structure of the problem
     • What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
3. Select a good order for solving subproblems

- In general:

\[ \text{Best}(i, j) = \text{cheapest way to multiply together } M_i \text{ through } M_j \]

\[ \text{Best}(i, j) = \min_{k=i}^{j-1} \left( \text{Best}(i, k) + \text{Best}(k + 1, j) + r_i r_{k+1} c_j \right) \]

\[ \text{Best}(i, i) = 0 \]

\[ \text{Best}(1, j) = \min_{k=i}^{j-1} \left( \text{Best}(1, k) + \text{Best}(k + 1, j) + r_i r_{k+1} c_j \right) \]

Save to M[i][j]

Read from M if present

\[ \text{Best}(1, n) = \min \left( \text{Best}(1, n - 1) + r_1 r_n c_n \right) \]

\[ \text{Best}(1, 2) + \text{Best}(3, n) + r_1 r_3 c_n \]

\[ \text{Best}(1, 3) + \text{Best}(4, n) + r_1 r_4 c_n \]

\[ \text{Best}(1, 4) + \text{Best}(5, n) + r_1 r_5 c_n \]

...
Top-Down Matrix Chain

```python
mem = {}
def best(i, j):
    if mem[i][j] not blank:
        return mem[i][j]
    if i == j:
        solution = 0
        mem[i][j] = solution
        return solution
    for i <= last_mult < j:
        subsolutions[last_mult] = best(i, last_mult) + best(last_mult, j) + r_i c_{last} c_j
        solution = min{subsolutions[last_mult]}
    mem[i][j] = solution
    return solution
```
3. Select a good order for solving subproblems

\[ \text{Best}(i, j) = \min_{k=i}^{j-1} \left( \text{Best}(i, k) + \text{Best}(k + 1, j) + r_ir_{k+1}c_j \right) \]

\[ \text{Best}(i, i) = 0 \]
3. Select a good order for solving subproblems

\[ \text{Best}(i, j) = \min_{k=i}^{j-1} \left( \text{Best}(i, k) + \text{Best}(k+1, j) + r_i r_{k+1} c_j \right) \]

\[ \text{Best}(i, i) = 0 \]

\[ \text{Best}(1,2) = \min \left\{ \text{Best}(1,1) + \text{Best}(2,2) + r_1 r_2 c_2 \right\} \]
3. Select a good order for solving subproblems

\[
Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j)
\]

\[
Best(i, i) = 0
\]

\[
Best(2,3) = \min \left\{ Best(2,2) + Best(3,3) + r_2 r_3 c_3 \right\}
\]
3. Select a good order for solving subproblems

\[
Best(i, j) = \min_{k=i}^{j-1} \left( Best(i, k) + Best(k+1, j) + r_i r_{k+1} c_j \right) \]

\[
Best(i, i) = 0
\]
3. Select a good order for solving subproblems

\[ Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k + 1, j) + r_ir_{k+1}c_j) \]

\[ Best(i, i) = 0 \]

\[ r_1r_2c_3 = 30 \cdot 35 \cdot 5 = 5250 \]

\[ r_1r_3c_3 = 30 \cdot 15 \cdot 5 = 2250 \]

\[ Best(1,3) = \min \begin{cases} 0 & 2625 \\ Best(1,1) + Best(2,3) + r_1r_2c_3 & 15750 \\ Best(1,2) + Best(3,3) + r_1r_3c_3 & \end{cases} \]
3. Select a good order for solving subproblems

To find $\text{Best}(i, j)$: Need all preceding terms of row $i$ and column $j$

Conclusion: solve in order of diagonal

$\text{Best}(i, j) = \min_{k=i}^{j-1} (\text{Best}(i, k) + \text{Best}(k+1, j) + r_i r_{k+1} c_j)$

$\text{Best}(i, i) = 0$
Matrix Chaining

\[
Best(i, j) = \min_{k=i}^{j-1} \left( Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j \right)
\]

\[
Best(i, i) = 0
\]

\[
Best(1,6) = \min \left\{ \begin{array}{l}
Best(1,1) + Best(2,6) + r_1 r_2 c_6 \\
Best(1,2) + Best(3,6) + r_1 r_3 c_6 \\
Best(1,3) + Best(4,6) + r_1 r_4 c_6 \\
Best(1,4) + Best(5,6) + r_1 r_5 c_6 \\
Best(1,5) + Best(6,6) + r_1 r_6 c_6
\end{array} \right. 
\]

\[
\begin{array}{cccccc}
& 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 15750 & 7875 & 9375 & 11875 & 15125 & \\
0 & 2625 & 4375 & 7125 & 10500 & \\
0 & 750 & 2500 & 5375 & &
\end{array}
\]
Run Time

1. Initialize $Best[i, i]$ to be all 0s
2. Starting at the main diagonal, working to the upper-right, fill in each cell using:
   1. $Best[i, i] = 0$
   2. $Best[i, j] = \min_{k=i}^{j-1} (Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j)$

$\Theta(n^2)$ cells in the Array
$\Theta(n)$ options for each cell

$\Theta(n^3)$ overall run time

Each “call” to Best() is a O(1) memory lookup
Backtrack to find the best order

“remember” which choice of $k$ was the minimum at each cell

$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k + 1, j) + r_ir_{k+1}c_j)$$

$Best(i, i) = 0$

$$Best(1, 6) = \min \begin{cases} 
Best(1,1) + Best(2, 6) + r_1r_2c_6 \\
Best(1,2) + Best(3, 6) + r_1r_3c_6 \\
Best(1,3) + Best(4, 6) + r_1r_4c_6 \\
Best(1,4) + Best(5, 6) + r_1r_5c_6 \\
Best(1,5) + Best(6, 6) + r_1r_6c_6 
\end{cases}$$
Matrix Chaining

\[ \text{Best}(i, j) = \min_{k=i}^{j-1} (\text{Best}(i, k) + \text{Best}(k + 1, j) + r_i r_{k+1} c_j) \]

\[ \text{Best}(i, i) = 0 \]

\[ \text{Best}(1, 6) = \min \]

\[ \begin{align*}
\text{Best}(1, 1) + \text{Best}(2, 6) + r_1 r_2 c_6 \\
\text{Best}(1, 2) + \text{Best}(3, 6) + r_1 r_3 c_6 \\
\text{Best}(1, 3) + \text{Best}(4, 6) + r_1 r_4 c_6 \\
\text{Best}(1, 4) + \text{Best}(5, 6) + r_1 r_5 c_6 \\
\text{Best}(1, 5) + \text{Best}(6, 6) + r_1 r_6 c_6
\end{align*} \]
Storing and Recovering Optimal Solution

• Maintain table choice[i,j] in addition to mem table
  – choice[i,j] = k means the best “split” was right after $M_k$
  – Work backwards from value for whole problem, choice[1,n]
  – Note: choice[i,i+1] = i because there are just 2 matrices

• From our example:
  – choice[1,6] = 3. So $[M_1 M_2 M_3] [M_4 M_5 M_6]$
  – We then need choice[1,3] = 1. So $[(M_1) (M_2 M_3)]$
  – Also need choice[4,6] = 5. So $[(M_4 M_5) M_6]$
  – Overall: $[(M_1) (M_2 M_3)] [(M_4 M_5) M_6]$