Algorithms and Data Structures:
Dynamic Programming; Matrix-chain multiplication
Algorithmic Paradigms

Divide and Conquer

*Idea:* Divide problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance.

*Examples:* Mergesort, Quicksort, Strassen’s algorithm, FFT.

Greedy Algorithms

*Idea:* Find solution by always making the choice that looks optimal at the moment — don’t look ahead, never go back.

*Examples:* Prim’s algorithm, Kruskal’s algorithm.

Dynamic Programming

*Idea:* Turn recursion upside down.

*Example:* Floyd-Warshall algorithm for the all pairs shortest path problem.
Dynamic Programming - A Toy Example

Fibonacci Numbers

\[
F_0 = 0, \\
F_1 = 1, \\
F_n = F_{n-1} + F_{n-2} \quad \text{(for } n \geq 2\).
\]

A recursive algorithm

**Algorithm** \textsc{Rec-Fib}(n)

1. if \( n = 0 \) then
2. return 0
3. else if \( n = 1 \) then
4. return 1
5. else
6. return \textsc{Rec-Fib}(n − 1) + \textsc{Rec-Fib}(n − 2)

Ridiculously slow: \textbf{exponentially many} repeated computations of \textsc{Rec-Fib}(j) for small values of \( j \).

ADS: lects 12 and 13 – slide 3 –
Why is the recursive solution so slow?
Running time $T(n)$ satisfies

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1) \geq F_n \sim (1.6)^n.$$ 

**BOARD:** We show $F_n \geq \frac{1}{2} (3/2)^n$ for $n \geq 8$. 

*ADS: lects 12 and 13 – slide 4 –*
Fibonacci Example (cont’d)

Dynamic Programming Approach

**Algorithm** DYN-FIB(n)

1. \( F[0] = 0 \)
2. \( F[1] = 1 \)
3. for \( i \leftarrow 2 \) to \( n \) do
   4. \( F[i] \leftarrow F[i - 1] + F[i - 2] \)
5. return \( F[n] \)

Build “from the bottom up”

Running Time

\( \Theta(n) \)

Very fast in practice - just need an array (of linear size) to store the \( F(i) \) values.
Multiplying Sequences of Matrices

Recall

Multiplying a \((p \times q)\) matrix with a \((q \times r)\) matrix (in the standard way) requires

\[ pqr \]

multiplications.

We want to compute products of the form

\[ A_1 \cdot A_2 \cdot \ldots \cdot A_n. \]

How do we set the parentheses?
Example

Compute

\[
\begin{array}{cccc}
A & \cdot & B & \cdot & C & \cdot & D \\
30 \times 1 & & 1 \times 40 & & 40 \times 10 & & 10 \times 25 \\
\end{array}
\]

Multiplication order \((A \cdot B) \cdot (C \cdot D)\) requires

\[
30 \cdot 1 \cdot 40 + 40 \cdot 10 \cdot 25 + 30 \cdot 40 \cdot 25 = 41,200
\]
multiplications.

Multiplication order \(A \cdot ((B \cdot C) \cdot D)\) requires

\[
1 \cdot 40 \cdot 10 + 1 \cdot 10 \cdot 25 + 30 \cdot 1 \cdot 25 = 1,400
\]
multiplications.
The Matrix Chain Multiplication Problem

Input:
Sequence of matrices $A_1, \ldots, A_n$, where $A_i$ is a $p_{i-1} \times p_i$-matrix

Output:
Optimal number of multiplications needed to compute $A_1 \cdot A_2 \cdots A_n$, and an optimal parenthesisation to realise this

Running time of algorithms will be measured in terms of $n$. 
Solution “Attempts”

Approach 1: Exhaustive search (CORRECT but SLOW).
Try all possible parenthesisations and compare them. Correct, but extremely slow; running time is $\Omega(3^n)$. UGLY PROOF

Approach 2: Greedy algorithm (INCORRECT).
Always do the cheapest multiplication first. Does not work correctly — sometimes, it returns a parenthesisation that is not optimal:

Example: Consider

$$A_1 \cdot A_2 \cdot A_3$$

$$3 \times 100 \quad 100 \times 2 \quad 2 \times 2$$

Solution proposed by greedy algorithm: $A_1 \cdot (A_2 \cdot A_3)$ with

$$100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000 \text{ multiplications.}$$

Optimal solution: $(A_1 \cdot A_2) \cdot A_3$ with

$$3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612 \text{ multiplications.}$$

ADS: lects 12 and 13 – slide 9 –
Solution “Attempts” (cont’d)

**Approach 3:** Alternative greedy algorithm (INCORRECT).
Set outermost parentheses such that cheapest multiplication is done last.

Doesn’t work correctly either (Exercise!).

**Approach 4:** Recursive (Divide and Conquer) - (SLOW - see over).
Divide:

\[(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)\]

For all \(k\), recursively solve the two sub-problems and then take best overall solution.

For \(1 \leq i \leq j \leq n\), let

\[m[i, j] = \text{least number of multiplications needed to compute } A_i \cdots A_j\]

Then

\[m[i, j] = \begin{cases} 
0 & \text{if } i = j, \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & \text{if } i < j.
\end{cases}\]
The Recursive Algorithm (SLOW)

Running time $T(n)$ satisfies the recurrence

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) + \Theta(n).$$

This implies

$$T(n) = \Omega(2^n).$$
Dynamic Programming Solution

As before:

\[ m[i, j] = \text{least number of multiplications needed to compute } A_i \cdots A_j \]

Moreover,

\[ s[i, j] = (\text{the smallest}) \ k \text{ such that } i \leq k < j \text{ and } m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j. \]

\( s[i, j] \) can be used to reconstruct the optimal parenthesisation.

Idea

Compute the \( m[i, j] \) and \( s[i, j] \) in a bottom-up fashion.

TURN RECURSION UPSIDE DOWN :-)

ADS: lects 12 and 13 – slide 12 –
Implementation

**Algorithm** Matrix-Chain-Order(p)

1. \( n \leftarrow p.length - 1 \)
2. \( \text{for } i \leftarrow 1 \text{ to } n \text{ do} \)
3. \( \quad m[i, i] \leftarrow 0 \)
4. \( \text{for } \ell \leftarrow 2 \text{ to } n \text{ do} \)
5. \( \quad \text{for } i \leftarrow 1 \text{ to } n - \ell + 1 \text{ do} \)
6. \( \quad \quad j \leftarrow i + \ell - 1 \)
7. \( \quad m[i, j] \leftarrow \infty \)
8. \( \quad \text{for } k \leftarrow i \text{ to } j - 1 \text{ do} \)
9. \( \quad \quad q \leftarrow m[i, k] + m[k + 1, j] + p_{i - 1}p_kp_j \)
10. \( \quad \quad \text{if } q < m[i, j] \text{ then} \)
11. \( \quad \quad \quad m[i, j] \leftarrow q \)
12. \( \quad \quad s[i, j] \leftarrow k \)
13. \( \text{return } s \)

**Running Time:** \( \Theta(n^3) \)
Example

\[
A_1 \cdot A_2 \cdot A_3 \cdot A_4 \\
30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25
\]

Solution for \( m \) and \( s \)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1200</td>
<td>700</td>
<td>1400</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>400</td>
<td>650</td>
<td></td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>10000</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimal Parenthesisation

\[ A_1 \cdot ((A_2 \cdot A_3) \cdot A_4) \]

ADS: lects 12 and 13 – slide 14 –
Multiplying the Matrices

**Algorithm** Matrix-Chain-Multiply\((A, p)\)

1. \(n \leftarrow A.\text{length}\)
2. \(s \leftarrow \text{Matrix-Chain-Order}(p)\)
3. return Rec-Mult\((A, s, 1, n)\)

**Algorithm** Rec-Mult\((A, s, i, j)\)

1. if \(i < j\) then
2. \(C \leftarrow \text{Rec-Mult}(A, s, i, s[i, j])\)
3. \(D \leftarrow \text{Rec-Mult}(A, s, s[i, j] + 1, j)\)
4. return \((C) \cdot (D)\)
5. else
6. return \(A_i\)

*ADS: lects 12 and 13 – slide 15*
Problems

see Wikipedia:

[CLRS] Sections 15.2-15.3

1. Review the Edit-Distance Algorithm and try to understand why it is a dynamic programming algorithm.

2. Exercise 15.2-1 of [CLRS].