1. Characterize structure of optimal solution.
2. Recursively define value of optimal solution.
3. Compute value of optimal sol. (bottom-up).
4. Construct actual optimal solution (e.g., shortest path vs. shortest path length from 3).

To design a DP algorithm:
- Optimal value may not be unique.
- Find solution of optimal value.

DP is usually applied to optimization problems.
Matrix-Chain Multiplication

- How to parenthesize the matrix product.
- Order to perform matrix multiplications.

- Output: an order to compute $A_1 A_2 \ldots A_n$ of matrices.
- Input: sequence $<A_1, A_2, \ldots, A_n>$ of matrices.

Operations (additions and multiplications) such that the total number of scalar operations is minimized.
Want: fully parenthesized $A_1A_2\ldots A_n$. s.t. minimize cost of multiplications.

- $A_i$: $p_i \times p_i$ size matrix.
- $A_1^{p_1}A_2^{p_2} \ldots A_n^{p_n}$.
- $A_1^{p_1}A_2^{p_2} \ldots A_n^{p_n}$ requires $p_1 p_2 \ldots p_n$ operations.

The way to parenthesize: big impact on cost of multiplications.

- $((A_1A_2)A_3)(A_4)$
- $(A_1(A_2A_3))A_4$
- $A_1(A_2)(A_3A_4)$
- $(A_1(A_2A_3))A_4$
- $(A_1(A_2A_3)A_4)$
- $A_1(A_2)(A_3)(A_4)$

Example: multiplication is associative.

No matter the order of multiplications, final result is the same parentheses.

Product of 2 fully parenthesized matrix products surrounded by
- A single matrix or
- A fully parenthesized matrix product.

Parenthesized Products
Counting # of parenthesizations

- check all possibilities

- $P(n)$ : # of parenthesizations

  - for $k = 1, 2, \ldots, n-1$ do:
    - split $A_1 \cdot A_2 \cdots A_n$ at $k, k+1$
    - parenthesize $A_1 \cdot A_2 \cdots A_k$ independently
      $A_{k+1} \cdot A_{k+2} \cdots A_n$

- Recurrence: $P(n) = \begin{cases} 
1 & \text{if } n=1 \\
\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2
\end{cases}$

- Solution: $P(n) = C(n-1)$

  - Sequence of Catalan numbers

  \[
  C(n) = \frac{1}{n+1} \binom{2n}{n} = \Theta \left( \frac{4^n}{n^{3/2}} \right)
  \]

  -> exponential
Structure of optimal solution

• Notation: \( A_{i,j} \) → matrix obtained after evaluating product \( A_i \cdot A_{i+1} \cdots A_j \).

• Optimal solution: split of \( A_1 \cdot A_2 \cdots A_n \) at some \( k, k+1 \), \( 1 \leq k < n \).

  • first compute \( A_{1,k} \), \( A_{k+1,n} \)
    \[ \text{multiply} \]
    \[ A_{1,n} \]

  • \( A_{1,k} \), \( A_{k+1,n} \) must be optimal parenthesisations

⇒ optimal solution contains within it

optimal solutions to subproblems

⇒ key ingredient for applying DPs!
2. **Recursive solution:**

- Define value of optimal solution recursively.

- **Subproblems:** compute minimum cost of
  \[ A_i; A_{i+1}; \ldots; A_j \], \( 1 \leq i \leq j \leq n \)

- Let \( m[i,j] \) be min # scalar multiplications to compute \( A_i \ldots j \)
  \( (m[1,n] is optimum for \( A_1 \ldots n \)) \)

- \( i = j \) \( \implies m[i,i] = 0 \)

- \( i < j \) \( \implies \) have split at \( A_k \), \( A_{k+1} \ldots A_j \), \( 1 \leq k < j \)
  \( \rightarrow (A_k; A_{i+1} \ldots A_j) \cdot (A_{k+1} \ldots A_j) \)
  \( \rightarrow m[i,j] = m[i,k] + m[k+1,j] + P_{i-k} P_k P_j \)

- But we don’t know \( K \)!

- \( j - 1 \) possible values for \( K \):
  \( i, i + 1, \ldots, j - 1 \)

- Optimal paranthesization has one of them
  \( \rightarrow \) check all

\[ m[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min \{ m[i,k] + m[k+1,j] + P_{i-k} P_k P_j \} & \text{if } i < j 
\end{cases} \]
Note: $m[i,j]$ gives cost. To obtain actual parenthesisation, define $S[i,j] = k$ for optimal split of within $A_j$.

3. **Computing optimal cost**

- Recursive algorithm to compute $m[1,n]$ is exponential time!

- **Key observation:**
  - "few" subproblems: one for each pair $i,j$:
    \[
    1 \leq i \leq j \leq n
    \]
  
    \[
    \Rightarrow \binom{n}{2} + n = \Theta(n^2) \text{ subproblems}
    \]

- A recursive algorithm may encounter a subproblem many times

  \[
  \Rightarrow \text{overlapping subproblems}
  \]

  \[
  \downarrow
  \]

  **Key ingredient for applying DP!**
Perform 3rd step of DP: compute optimal cost 

\[
A_i \rightarrow P_{i-1} \times P_i
\]

Input: \(<P_0, P_1, \ldots, P_n>\)

Data structures: \(m[i..n, 1..n]\) to store \(m[i, j]\)
\(s[i..n, 1..n]\) to store \(s[i, j]\)

**MCO** \((p)\) {

\[
\begin{align*}
n &= \text{length}(p) - 1; \\
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] &= \infty; \\
\text{for } l = 2 \text{ to } n \text{ do } \\
\quad \text{for } i = 1 \text{ to } n-l+1 \text{ do } \\
\quad \quad j &= i + l - 1; \\
\quad m[i, j] &= \infty; \\
\quad \text{for } k = i \text{ to } j-1 \text{ do } \\
\quad \quad g &= m[i, k] + m[k+1, j] + P_{i-1} P_k P_j; \\
\quad \quad \text{if } g < m[i, j] \text{ then } \\
\quad \quad \quad m[i, j] &= g; \ s[i, j] = k
\end{align*}
\]

}\)

i.e.: first \(m[i, i]\), \(i = 1, n\)

\[
\begin{align*}
\text{next } (l=2): m[i, i+1], \quad i = 1, n-1 \\
\text{next } (l=3): m[i, i+2], \quad i = 1, n-2 \\
\text{etc. } \Rightarrow [m[i, j]] \text{ depends on already computed } [m[k+1, j]]
\end{align*}
\]
Example:
\[ A_1 : 20 \times 35 \]
\[ A_2 : 35 \times 15 \]
\[ A_3 : 15 \times 5 \]
\[ A_4 : 5 \times 10 \]
\[ A_5 : 10 \times 20 \]
\[ A_6 : 20 \times 25 \]

\[ p_0 = 25 \]
\[ p_1 = 25 \]
\[ p_2 = 15 \]
\[ p_3 = 5 \]
\[ p_4 = 10 \]
\[ p_5 = 20 \]
\[ p_6 = 25 \]

\[ m[1,3] \rightarrow A_{1,2} \cdot A_3 : 15 \times 750 + 30 \times 15 \times 5 \]
\[ A_1 \cdot A_{2,3} : 2625 + 30 \times 35 \times 5 \]

\[ m[2,5] = \min \left\{ \begin{array}{c}
  m[2,2] + m[2,5] + p_2 p_5 = 13000 \cr
  m[2,2] + m[4,5] + p_4 p_5 = 1725 \cr
  m[2,4] + m[5,5] + p_4 p_5 = 11 \times 275
\end{array} \right. \]

\[ \Rightarrow (A_1 (A_2 A_3)) (A_4 A_5) A_6 \]

4. Constructing optimal solution. To have value from step 3
   - How to multiply matrices?!
   - Use \( S[i..n, i..n] \)
   - \( S[i, j] \) records \( K_i \) of \( i \leq K_j \), for optimal parenthesisation of \( A_i..j \)
   - Final multiplication in \( A_i..n \) is \( A_{i..x_{1..n}} A_{i+1..n} \)
Elements of Dynamic Programming

- When to look for a DP solution

- **Optimal substructure**
  - Optimal solution contains within it optimal solutions to subproblems
  - To show optimality of subproblems
    - Proof by contradiction

- **Overlapping subproblems**
  - # of distinct subproblems is polynomial
  - Solve each subproblem once

- Next class: All pairs shortest paths