Chain Matrix Multiplication

Version of November 5, 2014
Outline

- Review of matrix multiplication.
- The chain matrix multiplication problem.
- A dynamic programming algorithm for chain matrix multiplication.
Matrix: An \( n \times m \) matrix \( A = [a[i,j]] \) is a two-dimensional array

\[
A = \begin{bmatrix}
a[1, 1] & a[1, 2] & \cdots & a[1, m-1] & a[1, m] \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
a[n, 1] & a[n, 2] & \cdots & a[n, m-1] & a[n, m]
\end{bmatrix},
\]

which has \( n \) rows and \( m \) columns.

Example

A \( 4 \times 5 \) matrix:

\[
\begin{bmatrix}
12 & 8 & 9 & 7 & 6 \\
7 & 6 & 89 & 56 & 2 \\
5 & 5 & 6 & 9 & 10 \\
8 & 6 & 0 & -8 & -1
\end{bmatrix}.
\]
The product $C = AB$ of a $p \times q$ matrix $A$ and a $q \times r$ matrix $B$ is a $p \times r$ matrix $C$ given by

$$c[i, j] = \sum_{k=1}^{q} a[i, k] b[k, j], \quad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$
The product $C = AB$ of a $p \times q$ matrix $A$ and a $q \times r$ matrix $B$ is a $p \times r$ matrix $C$ given by

$$c[i, j] = \sum_{k=1}^{q} a[i, k] b[k, j], \quad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$

**Complexity of Matrix multiplication**: Note that $C$ has $pr$ entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.
The product $C = AB$ of a $p \times q$ matrix $A$ and a $q \times r$ matrix $B$ is a $p \times r$ matrix $C$ given by

$$c[i, j] = \sum_{k=1}^{q} a[i, k] b[k, j], \quad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$

**Complexity of Matrix multiplication:** Note that $C$ has $pr$ entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

**Example**

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$
The product $C = AB$ of a $p \times q$ matrix $A$ and a $q \times r$ matrix $B$ is a $p \times r$ matrix $C$ given by

$$c[i, j] = \sum_{k=1}^{q} a[i, k] b[k, j], \quad \text{for} \ 1 \leq i \leq p \ \text{and} \ 1 \leq j \leq r$$

**Complexity of Matrix multiplication:** Note that $C$ has $pr$ entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

**Example**

$$A = \begin{bmatrix}
1 & 8 & 9 \\
7 & 6 & -1 \\
5 & 5 & 6
\end{bmatrix}, \quad B = \begin{bmatrix}
1 & 8 \\
7 & 6 \\
5 & 5
\end{bmatrix}, \quad C = AB = \begin{bmatrix}
102 & 101 \\
44 & 87 \\
70 & 100
\end{bmatrix}.$$
Matrix multiplication is associative, e.g.,
Matrix multiplication is associative, e.g.,

\[ A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3), \]

so parenthesization does not change result.
Matrix multiplication is **associative**, e.g.,

\[ A_1A_2A_3 = (A_1A_2)A_3 = A_1(A_2A_3), \]

so parenthesization does not change result.

Matrix multiplication is **NOT commutative**, e.g.,
Matrix multiplication is **associative**, e.g.,

\[ A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3), \]

so parenthesization does not change result.

Matrix multiplication is **NOT commutative**, e.g.,

\[ A_1 A_2 \neq A_2 A_1 \]
Given $p \times q$ matrix $A$, $q \times r$ matrix $B$ and $r \times s$ matrix $C$, $ABC$ can be computed in two ways:

- $(AB)C$
- $A(BC)$

The number of multiplications needed are:

- $\text{mult}[(AB)C] = pqr + prs$
- $\text{mult}[A(BC)] = qrs + pqs$

Example:

- For $p = 5$, $q = 4$, $r = 6$ and $s = 2$, $\text{mult}[(AB)C] = 180$
- $\text{mult}[A(BC)] = 88$

A big difference! Implication: Multiplication "sequence" (parenthesization) is important!!
Given $p \times q$ matrix $A$, $q \times r$ matrix $B$ and $r \times s$ matrix $C$, $ABC$ can be computed in two ways: $(AB)C$ and $A(BC)$.
Matrix Multiplication of $ABC$

- Given $p \times q$ matrix $A$, $q \times r$ matrix $B$ and $r \times s$ matrix $C$, $ABC$ can be computed in two ways: $(AB)C$ and $A(BC)$
- The number of multiplications needed are:

  \[
  \text{mult}[(AB)C] = pqr + prs, \\
  \text{mult}[A(BC)] = qrs + pqs.
  \]
Given \( p \times q \) matrix \( A \), \( q \times r \) matrix \( B \) and \( r \times s \) matrix \( C \), \( ABC \) can be computed in two ways: \((AB)C\) and \(A(BC)\).

The number of multiplications needed are:

\[
\text{mult}[(AB)C] = pqr + prs, \\
\text{mult}[A(BC)] = qrs + pqs.
\]

Example

For \( p = 5 \), \( q = 4 \), \( r = 6 \) and \( s = 2 \),

\[
\text{mult}[(AB)C] = 180, \\
\text{mult}[A(BC)] = 88.
\]

A big difference!
Given \( p \times q \) matrix \( A \), \( q \times r \) matrix \( B \) and \( r \times s \) matrix \( C \), \( ABC \) can be computed in two ways: \((AB)C\) and \(A(BC)\)

The number of multiplications needed are:

\[
\text{mult}[(AB)C] = pqr + prs,
\]

\[
\text{mult}[A(BC)] = qrs + pqs.
\]

**Example**

For \( p = 5, q = 4, r = 6 \) and \( s = 2 \),

\[
\text{mult}[(AB)C] = 180,
\]

\[
\text{mult}[A(BC)] = 88.
\]

A big difference!

**Implication**: Multiplication “sequence” (parenthesization) is important!!
Review of matrix multiplication.
Outline

- Review of matrix multiplication.
- The chain matrix multiplication problem.
Review of matrix multiplication.

The chain matrix multiplication problem.

A dynamic programming algorithm for chain matrix multiplication.
### Definition (Chain matrix multiplication problem)

Given dimensions $p_0, p_1, \ldots, p_n$, corresponding to matrix sequence $A_1, A_2, \ldots, A_n$ in which $A_i$ has dimension $p_{i-1} \times p_i$, determine the “multiplication sequence” that minimizes the number of scalar multiplications in computing $A_1 A_2 \cdots A_n$.

- i.e., determine how to parenthesize the multiplications.
The Chain Matrix Multiplication Problem

Definition (Chain matrix multiplication problem)

Given dimensions $p_0, p_1, \ldots, p_n$, corresponding to matrix sequence $A_1, A_2, \ldots, A_n$ in which $A_i$ has dimension $p_{i-1} \times p_i$, determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1A_2\cdots A_n$.

i.e., determine how to parenthesize the multiplications.

Example

$A_1A_2A_3A_4 = (A_1A_2)(A_3A_4) = A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) = (A_1A_2)(A_3)(A_4) = (A_1(A_2A_3))(A_4)$

Exhaustive search: $\Omega(4^n/n^{3/2})$. 
The Chain Matrix Multiplication Problem

Definition (Chain matrix multiplication problem)

Given dimensions \(p_0, p_1, \ldots, p_n\), corresponding to matrix sequence \(A_1, A_2, \ldots, A_n\) in which \(A_i\) has dimension \(p_{i-1} \times p_i\), determine the “multiplication sequence” that minimizes the number of scalar multiplications in computing \(A_1A_2\cdots A_n\).

- i.e., determine how to parenthesize the multiplications.

Example

\[
A_1A_2A_3A_4 = (A_1A_2)(A_3A_4) = A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) = ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4)
\]

Exhaustive search: \(\Omega(4^n/n^{3/2})\).

Question

Is there a better approach?
Definition (Chain matrix multiplication problem)

Given dimensions $p_0, p_1, \ldots, p_n$, corresponding to matrix sequence $A_1, A_2, \ldots, A_n$ in which $A_i$ has dimension $p_{i-1} \times p_i$, determine the “multiplication sequence” that minimizes the number of scalar multiplications in computing $A_1 A_2 \cdots A_n$.

i.e., determine how to parenthesize the multiplications.

Example

$A_1 A_2 A_3 A_4 = (A_1 A_2)(A_3 A_4) = A_1(A_2(A_3 A_4)) = A_1((A_2 A_3)A_4) = (A_1 A_2)(A_3 A_4) = (A_1(A_2 A_3))(A_4)$

Exhaustive search: $\Omega(4^n / n^{3/2})$.

Question

Is there a better approach?

Yes – DP
Review of matrix multiplication.
Outline

- Review of matrix multiplication.
- The chain matrix multiplication problem.
Review of matrix multiplication.
The chain matrix multiplication problem.
A dynamic programming algorithm.
Developing a Dynamic Programming Algorithm

Step 1: Define Space of Subproblems

Original Problem: Determine minimal cost multiplication sequence for $A_1..A_n$.

Subproblems: For every pair $1 \leq i \leq j \leq n$:

Determine minimal cost multiplication sequence for $A_i..A_j = A_iA_{i+1} \cdots A_j$.

Note that $A_i..A_j$ is a $p_i-1 \times p_j$ matrix.

There are $\binom{n}{2} = \Theta(n^2)$ such subproblems. (Why?)

How can we solve larger problems using subproblem solutions?
Step 1: Define Space of Subproblems

- Original Problem:
  Determine minimal cost multiplication sequence for $A_1..n$. 

Note that $A_i..j$ is a $p_{i-1} \times p_j$ matrix.
There are $\binom{n}{2} = \Theta(n^2)$ such subproblems. (Why?)
Developing a Dynamic Programming Algorithm

Step 1: Define Space of Subproblems

- **Original Problem:**
  Determine minimal cost multiplication sequence for $A_{1..n}$.

- **Subproblems:** For every pair $1 \leq i \leq j \leq n$:
  Determine minimal cost multiplication sequence for
  $A_{i..j} = A_i A_{i+1} \cdots A_j$.

- Note that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.
Step 1: Define Space of Subproblems

- **Original Problem:**
  Determine minimal cost multiplication sequence for $A_{1..n}$.

- **Subproblems:** For every pair $1 \leq i \leq j \leq n$:
  Determine minimal cost multiplication sequence for $A_{i..j} = A_i A_{i+1} \cdots A_j$.
  Note that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

- There are $\binom{n}{2} = \Theta(n^2)$ such subproblems. (Why?)

- How can we solve larger problems using subproblem solutions?
At the last step of any optimal multiplication sequence (for a subproblem), there is some $k$ such that the two matrices $A_{i..k}$ and $A_{k+1..j}$ are multiplied together.
At the last step of any optimal multiplication sequence (for a subbroblem), there is some $k$ such that the two matrices $A_{i..k}$ and $A_{k+1..j}$ are multiplied together. That is,

$$A_{i..j} = (A_i \cdots A_k) (A_{k+1} \cdots A_j)$$
At the last step of *any* optimal multiplication sequence (for a subproblem), there is some \( k \) such that the two matrices \( A_{i..k} \) and \( A_{k+1..j} \) are multiplied together. That is,

\[
A_{i..j} = (A_i \cdots A_k) (A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.
\]
At the last step of any optimal multiplication sequence (for a subproblem), there is some \(k\) such that the two matrices \(A_{i..k}\) and \(A_{k+1..j}\) are multiplied together. That is,

\[
A_{i..j} = (A_i \cdots A_k) (A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.
\]

Question: How do we decide where to split the chain (what is \(k\)?)

ANS: Can be any \(k\). Need to check all possible values.

ANS: \(A_{i..k}\) and \(A_{k+1..j}\) must be computed optimally, so we can apply the same procedure recursively.
At the last step of any optimal multiplication sequence (for a subproblem), there is some $k$ such that the two matrices $A_{i..k}$ and $A_{k+1..j}$ are multiplied together. That is,

$$A_{i..j} = (A_i \cdots A_k) \ (A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$ 

**Question**

How do we decide where to split the chain (what is $k$)?

**ANS:** Can be *any* $k$. Need to check all possible values.
At the last step of any optimal multiplication sequence (for a subproblem), there is some $k$ such that the two matrices $A_{i..k}$ and $A_{k+1..j}$ are multiplied together. That is,

$$A_{i..j} = (A_i \cdots A_k) (A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$ 

**Question**

How do we decide where to split the chain (what is $k$)?

**ANS:** Can be any $k$. Need to check all possible values.

**Question**

How do we parenthesize the two subchains $A_{i..k}$ and $A_{k+1..j}$?
At the last step of any optimal multiplication sequence (for a subproblem), there is some $k$ such that the two matrices $A_{i..k}$ and $A_{k+1..j}$ are multiplied together. That is,

$$A_{i..j} = (A_i \cdots A_k) (A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$ 

**Question**

How do we decide where to split the chain (what is $k$)?

**ANS:** Can be *any* $k$. Need to check all possible values.

**Question**

How do we parenthesize the two subchains $A_{i..k}$ and $A_{k+1..j}$?

**ANS:** $A_{i..k}$ and $A_{k+1..j}$ must be computed optimally, so we can apply the same procedure *recursively*. 
If the “optimal” solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at the final step, then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in the optimal solution must also be optimal.
If the “optimal” solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at the final step, then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in the optimal solution must also be optimal.

- If parenthesization of $A_{i..k}$ was not optimal, it could be replaced by a cheaper parenthesization, yielding a cheaper final solution, contradicting optimality.
If the “optimal” solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at the final step, then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in the optimal solution must also be optimal.

- If parenthesization of $A_{i..k}$ was not optimal, it could be replaced by a cheaper parenthesization, yielding a cheaper final solution, contradicting optimality.

- Similarly, if parenthesization of $A_{k+1..j}$ was not optimal, it could be replaced by a cheaper parenthesization, again yielding contradiction of cheaper final solution.
Relationships among subproblems

Step 2: Constructing optimal solutions from optimal subproblem solutions.

For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. This optimum cost must satisfy the following recursive definition.

$$m[i,j] = \begin{cases} 
0 & i = j, \\
\min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_i - 1 p_k p_j) & i < j
\end{cases}$$

$A_{i..j} = A_{i..k} A_{k+1..j}$
Step 2: Constructing optimal solutions from optimal subproblem solutions.

For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. This optimum cost must satisfy the following recursive definition:
Step 2: Constructing optimal solutions from optimal subproblem solutions.

For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. This optimum cost must satisfy the following recursive definition.

$$m[i, j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & i < j \end{cases}$$
Step 2: Constructing optimal solutions from optimal subproblem solutions.

For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of
multiplications needed to compute $A_{i..j}$. This optimum cost
must satisfy the following recursive definition.

$$m[i, j] = \begin{cases} 
0 & i = j, \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & i < j
\end{cases}$$

$$A_{i..j} = A_{i..k}A_{k+1..j}$$
Proof of Recurrence

Proof.

If \( j = i \), then \( m[i, j] = 0 \) because, no multiplications are required.

If \( i < j \), note that, for every \( k \), calculating \( A_{i..k} \) and \( A_{k+1..j} \) optimally and then finishing by multiplying \( A_{i..k}A_{k+1..j} \) to get \( A_{i..j} \) uses \((m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)\) multiplications.

The optimal way of calculating \( A_{i..j} \) uses no more than the worst of these \( j - i \) ways so

\[
m[i, j] \leq \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j).
\]

\( A_{i..j} = A_{i..k}A_{k+1..j} \)
Proof.

For the other direction, note that an optimal sequence of multiplications for $A_{i..j}$ is equivalent to splitting $A_{i..j} = A_{i..k}A_{k+1..j}$ for some $k$, where the sequences of multiplications to calculate $A_{i..k}$ and $A_{k+1..j}$ are also optimal. Hence, for that special $k$,

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j.$$
Proof of Recurrence (II)

Proof.

For the other direction, note that an optimal sequence of multiplications for $A_{i..j}$ is equivalent to splitting $A_{i..j} = A_{i..k}A_{k+1..j}$ for some $k$, where the sequences of multiplications to calculate $A_{i..k}$ and $A_{k+1..j}$ are also optimal. Hence, for that special $k$,

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j.$$

Combining with the previous page, we have just proven

$$m[i, j] = \begin{cases} 
0 & i = j, \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & i < j.
\end{cases}$$
Step 3: Bottom-up computation of $m[i, j]$. 

Recurrence:

$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_i - 1 p_k p_j)$
Step 3: Bottom-up computation of $m[i, j]$.  

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$
Step 3: Bottom-up computation of $m[i, j]$.

Recurrence:

$$m[i, j] = \min_{i \leq k < j} \left( m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \right)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.
Step 3: Bottom-up computation of $m[i, j]$.

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.

An easy way to ensure this is to compute them in increasing order of the size $(j - i)$ of the matrix-chain $A_{i..j}$:

$m[1, 2]$, $m[2, 3]$, $m[3, 4]$, $\ldots$, $m[n - 3, n - 2]$, $m[n - 2, n - 1]$, $m[n - 1, n]$
Step 3: Bottom-up computation of $m[i, j]$.

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.

An easy way to ensure this is to compute them in increasing order of the size $(j - i)$ of the matrix-chain $A_{i..j}$:

Step 3: Bottom-up computation of $m[i, j]$.  

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.

An easy way to ensure this is to compute them in increasing order of the size $(j - i)$ of the matrix-chain $A_{i..j}$:

$m[1, 2], m[2, 3], m[3, 4], \ldots, m[n - 3, n - 2], m[n - 2, n - 1], m[n - 1, n]
\qquad m[1, 3], m[2, 4], m[3, 5], \ldots, m[n - 3, n - 1], m[n - 2, n]
\qquad m[1, 4], m[2, 5], m[3, 6], \ldots, m[n - 3, n]$
Developing a Dynamic Programming Algorithm

Step 3: Bottom-up computation of $m[i, j]$.

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.

An easy way to ensure this is to compute them in increasing order of the size $(j - i)$ of the matrix-chain $A_i..j$:

$m[1, 2], m[2, 3], m[3, 4], \ldots, m[n - 3, n - 2], m[n - 2, n - 1], m[n - 1, n]$  
$m[1, 3], m[2, 4], m[3, 5], \ldots, m[n - 3, n - 1], m[n - 2, n]$  
$m[1, 4], m[2, 5], m[3, 6], \ldots, m[n - 3, n]$  
$\ldots$  
$m[1, n - 1], m[2, n]$
Step 3: Bottom-up computation of $m[i, j]$.

Recurrence:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j)$$

Fill in the $m[i, j]$ table in an order, such that when it is time to calculate $m[i, j]$, the values of $m[i, k]$ and $m[k + 1, j]$ for all $k$ are already available.

An easy way to ensure this is to compute them in increasing order of the size $(j - i)$ of the matrix-chain $A_{i..j}$:

- $m[1, 2], m[2, 3], m[3, 4], \ldots, m[n - 3, n - 2], m[n - 2, n - 1], m[n - 1, n]$
- $m[1, 3], m[2, 4], m[3, 5], \ldots, m[n - 3, n - 1], m[n - 2, n]$
- $m[1, 4], m[2, 5], m[3, 6], \ldots, m[n - 3, n]$
- $\ldots$
- $m[1, n - 1], m[2, n]$
- $m[1, n]$
Example

A chain of four matrices $A_1, A_2, A_3$ and $A_4$, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$. 

Example for the Bottom-Up Computation
Example

A chain of four matrices $A_1$, $A_2$, $A_3$ and $A_4$, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

S0: Initialization
Step 1: Computing $m[1, 2]$
Step 1: Computing $m[1, 2]$

By definition

$$m[1, 2] = \min_{1 \leq k \leq 2} (m[1, k] + m[k + 1, 2] + p_0 p_k p_2)$$

$$= m[1, 1] + m[2, 2] + p_0 p_1 p_2 = 120.$$
Step 1: Computing $m[1, 2]$

By definition

$$
m[1, 2] = \min_{1 \leq k < 2} (m[1, k] + m[k + 1, 2] + p_0 p_k p_2)
$$

$$= m[1, 1] + m[2, 2] + p_0 p_1 p_2 = 120.
$$
Step 2: Computing $m[2, 3]$

By definition

$$m[2, 3] = \min_{2 \leq k < 3} (m[2, k] + pm[k + 1, 3] + p_1 p_k p_3)$$

Step 2: Computing $m[2, 3]$

By definition

$$m[2, 3] = \min_{2 \leq k < 3} (m[2, k] + pm[k + 1, 3] + p_1 p_k p_3)$$

Step 3: Computing $m[3, 4]$

By definition

$$m[3, 4] = \min_{3 \leq k < 4} (m[3, k] + m[k + 1, 4] + p_2 p_k p_4)$$
Step 3: Computing $m[3, 4]$

By definition

$$m[3, 4] = \min_{3 \leq k < 4} (m[3, k] + m[k + 1, 4] + p_2 p_k p_4)$$

Step 3: Computing \( m[3, 4] \)

By definition

\[
m[3, 4] = \min_{3 \leq k < 4} (m[3, k] + m[k + 1, 4] + p_2p_kp_4)
\]

\[
\]
Step 4: Computing $m[1, 3]$

By definition

$$m[1, 3] = \min_{1 \leq k < 3} (m[1, k] + m[k + 1, 3] + p_0p_kp_3)$$

$$= \min \left\{ m[1, 1] + m[2, 3] + p_0p_1p_3, m[1, 2] + m[3, 3] + p_0p_2p_3 \right\}$$

$$= 88.$$
Example – Continued

Step 5: Computing $m[2, 4]$

By definition

$$m[2, 4] = \min_{2 \leq k < 4} (m[2, k] + m[k + 1, 4] + p_1 p_k p_4)$$

$$= \min \left\{ m[2, 2] + m[3, 4] + p_1 p_2 p_4, m[2, 3] + m[4, 4] + p_1 p_3 p_4 \right\}$$

$$= 104.$$
Step 6: Computing $m[1, 4]$

By definition

$$m[1, 4] = \min_{1 \leq k < 4} (m[1, k] + m[k + 1, 4] + p_0 p_k p_4)$$

$$= \min \left\{ \begin{array}{c}
m[1, 1] + m[2, 4] + p_0 p_1 p_4 \\
m[1, 2] + m[3, 4] + p_0 p_2 p_4 \\
m[1, 3] + m[4, 4] + p_0 p_3 p_4 \\
\end{array} \right\}$$

$$= 158.$$
Constructing a Solution

- $m[i, j]$ only keeps costs but not actual multiplication sequence.
- To solve problem, need to reconstruct multiplication sequence that yields $m[1, n]$.
- Solution: similar to previous DP algorithm(s) keep an auxillary array $s[*, *]$.
- $s[i, j] = k$ where $k$ is the index that achieves minimum in

$$m[i, j] = \min_{i \leq k < j} \left( m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \right).$$
Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n,1..n]$, where $s[i,j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.

How to Recover the Multiplication Sequence using $s[i,j]$?

- $s[1,n]$ ($A_1 \cdots A_n$)
- $s[1,s[1,n]]$ ($A_1 \cdots A_{s[1,n]}$)
- $s[s[1,n]+1,n]$ ($A_{s[1,n]+1} \cdots A_n$)

... Apply recursively until multiplication sequence is completely determined.
Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.

Question

How to Recover the Multiplication Sequence using $s[i, j]$?
Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.

Question

How to Recover the Multiplication Sequence using $s[i, j]$?

$$s[1, n] \quad (A_1 \cdots A_{s[1,n]}) (A_{s[1,n]+1} \cdots A_n)$$
Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.

Question

How to Recover the Multiplication Sequence using $s[i, j]$?

\[
s[1, n] \quad (A_1 \cdots A_{s[1, n]}) (A_{s[1, n]+1} \cdots A_n)
\]
\[
s[1, s[1, n]] \quad (A_1 \cdots A_{s[1, s[1, n]]}) (A_{s[1, s[1, n]]+1} \cdots A_{s[1, n]})
\]
Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k} A_{k+1..j}$.

Question

How to Recover the Multiplication Sequence using $s[i, j]$?

\[
\begin{align*}
s[1, n] & \quad (A_1 \cdots A_{s[1,n]}) (A_{s[1,n]+1} \cdots A_n) \\
s[1, s[1, n]] & \quad (A_1 \cdots A_{s[1,s[1,n]]}) (A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\
s[s[1, n] + 1, n] & \quad (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]}) (A_{s[s[1,n]+1,n]+1} \cdots A_n)
\end{align*}
\]
Developing a Dynamic Programming Algorithm

Step 4: Constructing optimal solution

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes $k$ for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.

Question

How to Recover the Multiplication Sequence using $s[i, j]$?

$$
\begin{align*}
  s[1, n] & : (A_1 \cdots A_{s[1,n]}) (A_{s[1,n]+1} \cdots A_n) \\
  s[1, s[1, n]] & : (A_1 \cdots A_{s[1,s[1,n]]}) (A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\
  s[s[1, n] + 1, n] & : (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]}) (A_{s[s[1,n]+1,n]+1} \cdots A_n) \\
  \vdots & : \vdots
\end{align*}
$$
**Step 4: Constructing optimal solution**

**Idea:** Maintain an array \( s[1..n, 1..n] \), where \( s[i, j] \) denotes \( k \) for the optimal splitting in computing \( A_{i..j} = A_{i..k}A_{k+1..j} \).

**Question**

How to Recover the Multiplication Sequence using \( s[i, j] \)?

\[
\begin{align*}
  s[1, n] & \quad (A_1 \cdots A_{s[1,n]}) (A_{s[1,n]+1} \cdots A_n) \\
  s[1, s[1, n]] & \quad (A_1 \cdots A_{s[1,s[1,n]]}) (A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\
  s[s[1, n] + 1, n] & \quad (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]}) (A_{s[s[1,n]+1,n]+1} \cdots A_n) \\
  \vdots & \quad \vdots \\
\end{align*}
\]

Apply recursively until multiplication sequence is completely determined.
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed.
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

$$s[1, 6] = 3$$
### Example (Finding the Multiplication Sequence)

Consider \( n = 6 \). Assume array \( s[1..6, 1..6] \) has been properly constructed. The multiplication sequence is recovered as follows.

\[
s[1, 6] = 3 \quad (A_1 A_2 A_3)(A_4 A_5 A_6)
\]
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

\[
\begin{align*}
  s[1, 6] &= 3 \quad (A_1A_2A_3)(A_4A_5A_6) \\
  s[1, 3] &= 1 
\end{align*}
\]
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

\[
\begin{align*}
s[1, 6] &= 3 \quad (A_1 A_2 A_3)(A_4 A_5 A_6) \\
s[1, 3] &= 1 \quad (A_1 (A_2 A_3))
\end{align*}
\]
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

\[
\begin{align*}
  s[1, 6] &= 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\
  s[1, 3] &= 1 & (A_1 (A_2 A_3)) \\
  s[4, 6] &= 5 \\
\end{align*}
\]
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

\[
\begin{align*}
    s[1, 6] &= 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\
    s[1, 3] &= 1 & (A_1 (A_2 A_3)) \\
    s[4, 6] &= 5 & ((A_4 A_5) A_6)
\end{align*}
\]
**Example (Finding the Multiplication Sequence)**

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

$$
\begin{align*}
   s[1, 6] &= 3 & (A_1A_2A_3)(A_4A_5A_6) \\
   s[1, 3] &= 1 & (A_1(A_2A_3)) \\
   s[4, 6] &= 5 & ((A_4A_5)A_6)
\end{align*}
$$

Hence the final multiplication sequence is
Example (Finding the Multiplication Sequence)

Consider $n = 6$. Assume array $s[1..6, 1..6]$ has been properly constructed. The multiplication sequence is recovered as follows.

$$
\begin{align*}
    s[1, 6] &= 3 \quad (A_1A_2A_3)(A_4A_5A_6) \\
    s[1, 3] &= 1 \quad (A_1(A_2A_3)) \\
    s[4, 6] &= 5 \quad ((A_4A_5)A_6)
\end{align*}
$$

Hence the final multiplication sequence is

$$(A_1(A_2A_3))((A_4A_5)A_6).$$
Matrix-Chain\((p, n)\): // I is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] =
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
Matrix-Chain\((p, n)\):  // l is length of sub-chain

for \(i = 1\) to \(n\) do \(m[i, i] = 0;\)

; for \(l = \)
The Dynamic Programming Algorithm

Matrix-Chain\( (p, n) \): // \( l \) is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0;
\]
\[
\text{for } l = 2 \text{ to } n \text{ do }
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \( \leq n \) values. Hence the time complexity is \( O(n^3) \). Space complexity is \( \Theta(n^2) \).
Matrix-Chain($p, n$): // $l$ is length of sub-chain

```plaintext
for $i = 1$ to $n$ do $m[i, i] = 0$
for $l = 2$ to $n$ do
  for $i = 1$ to $n - l + 1$
    $j = i + l - 1$
    $m[i, j] = \infty$
    for $k = i$ to $j - 1$
      $q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$
      if $q < m[i, j]$
        $m[i, j] = q$
        $s[i, j] = k$
```

Complexity: The loops are nested three levels deep. Each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$. Space complexity is $\Theta(n^2)$. 
Matrix-Chain\((p, n)\):  // \(l\) is length of sub-chain

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
&\text{for } l = 2 \text{ to } n \text{ do} \\
&\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do} \\
&\quad\quad j =
\end{align*}
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
Matrix-Chain\((p, n)\): // l is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
, \\
\text{for } l = 2 \text{ to } n \text{ do} \\
\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do} \\
\quad \quad j = i + l - 1; \\
\quad m[i, j] = \\
\]
The Dynamic Programming Algorithm

**Matrix-Chain** \((p, n)\): // \(l\) is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0;
\]
\[
; \text{ }
\]
\[
\text{for } l = 2 \text{ to } n \text{ do }
\]
\[
\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do }
\quad \quad j = i + l - 1;
\quad m[i, j] = \infty;
\quad \text{for } k =
\]

**Complexity:** The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
The Dynamic Programming Algorithm

Matrix-Chain($p, n$): // $l$ is length of sub-chain

```
for $i = 1$ to $n$ do $m[i, i] = 0$
;
for $l = 2$ to $n$ do
    for $i = 1$ to $n - l + 1$ do
        $j = i + l - 1$
        $m[i, j] = \infty$
        for $k = i$ to
```

Complexity: The loops are nested three levels deep. Each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$. Space complexity is $\Theta(n^2)$. 

Version of November 5, 2014
The Dynamic Programming Algorithm

Matrix-Chain($p, n$): // l is length of sub-chain

```plaintext
for $i = 1$ to $n$ do $m[i, i] = 0$;
;
for $l = 2$ to $n$ do
    for $i = 1$ to $n - l + 1$ do
        $j = i + l - 1$;
        $m[i, j] = \infty$;
        for $k = i$ to $j - 1$ do
            $q = $
```

Complexity: The loops are nested three levels deep. Each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$. Space complexity is $\Theta(n^2)$.
Matrix-Chain\((p, n)\):
// l is length of sub-chain

```plaintext
for i = 1 to n do m[i, i] = 0;
;
for l = 2 to n do
   for i = 1 to n - l + 1 do
      j = i + l - 1;
      m[i, j] = \infty;
      for k = i to j - 1 do
         q = m[i, k] +
```

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
Matrix-Chain\((p, n)\): // l is length of sub-chain

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
&\text{;}
\end{align*}
\]

\[
\begin{align*}
&\text{for } l = 2 \text{ to } n \text{ do } \\
&\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
&\qquad j = i + l - 1; \\
&\qquad m[i, j] = \infty; \\
&\qquad \text{for } k = i \text{ to } j - 1 \text{ do } \\
&\qquad\qquad q = m[i, k] + m[k + 1, j] +
\end{align*}
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
The Dynamic Programming Algorithm

Matrix-Chain\((p, n)\): // \(l\) is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0;
\]

\[
\text{for } l = 2 \text{ to } n \text{ do }
\]

\[
\text{for } i = 1 \text{ to } n - l + 1 \text{ do }
\]

\[
j = i + l - 1;
\]

\[
m[i, j] = \infty;
\]

\[
\text{for } k = i \text{ to } j - 1 \text{ do }
\]

\[
q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j];
\]

\[
\text{if } q < m[i, j] \text{ then } m[i, j] = q;
\]

\[
s[i, j] = k;
\]

\text{return } m[1, n]; (Optimum in } m[i, j].

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
Matrix-Chain($p, n$): // l is length of sub-chain

\[
\begin{align*}
\text{for } i = 1 \text{ to } n & \text{ do } m[i, i] = 0; \\
, \\
\text{for } l = 2 \text{ to } n & \text{ do } \\
& \text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
& \quad j = i + l - 1; \\
& \quad m[i, j] = \infty; \\
& \quad \text{for } k = i \text{ to } j - 1 \text{ do } \\
& \quad \quad q = m[i, k] + m[k + 1, j] + p[i - 1] \cdot p[k] \cdot p[j] \\
\end{align*}
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
Matrix-Chain\((p, n)\):  // \(l\) is length of sub-chain

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
&\text{for } l = 2 \text{ to } n \text{ do} \\
&\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do} \\
&\qquad j = i + l - 1; \\
&\qquad m[i, j] = \infty; \\
&\qquad \text{for } k = i \text{ to } j - 1 \text{ do} \\
&\qquad\quad q = m[i, k] + m[k + 1, j] + p[i - 1] \times p[k] \times p[j];
\end{align*}
\]
Matrix-Chain \((p, n)\): // \(l\) is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
, \\
\text{for } l = 2 \text{ to } n \text{ do } \\
\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
\qquad j = i + l - 1; \\
\qquad m[i, j] = \infty; \\
\qquad \text{for } k = i \text{ to } j - 1 \text{ do } \\
\qquad \qquad q = m[i, k] + m[k + 1, j] + p[i - 1] \times p[k] \times p[j]; \\
\qquad \text{if } q < m[i, j] \text{ then } \\
\qquad \qquad m[i, j] = q;
\]
Matrix-Chain\((p, n)\): // l is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0;
\]
\[
\text{for } l = 2 \text{ to } n \text{ do }
\]
\[
\text{for } i = 1 \text{ to } n - l + 1 \text{ do}
\]
\[
\text{\quad j = i + l - 1;}
\]
\[
\text{\quad m[i, j] = \infty;}
\]
\[
\text{\quad for } k = i \text{ to } j - 1 \text{ do}
\]
\[
\quad q = m[i, k] + m[k + 1, j] + p[i - 1] \ast p[k] \ast p[j];
\]
\[
\quad \text{if } q < m[i, j] \text{ then}
\]
\[
\quad \quad m[i, j] = q;
\]
\[
\quad \quad s[i, j] =
\]
The Dynamic Programming Algorithm

**Matrix-Chain**($p, n$): // l is length of sub-chain

```
for i = 1 to n do m[i, i] = 0;

for l = 2 to n do
    for i = 1 to n - l + 1 do
        j = i + l - 1;
        m[i, j] = \infty;
        for k = i to j - 1 do
            q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j];
            if q < m[i, j] then
                m[i, j] = q;
                s[i, j] = k;
        end
    end
end
return m and s (Optimum in m[1, n])
```

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).
The Dynamic Programming Algorithm

Matrix-Chain \((p, n)\): // \(l\) is length of sub-chain

\[
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
\]

\[
\text{for } l = 2 \text{ to } n \text{ do } \\
\text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
\hspace{1cm} j = i + l - 1; \\
\hspace{1cm} m[i, j] = \infty; \\
\hspace{1cm} \text{for } k = i \text{ to } j - 1 \text{ do } \\
\hspace{2cm} q = m[i, k] + m[k + 1, j] + p[i - 1] \times p[k] \times p[j]; \\
\hspace{2cm} \text{if } q < m[i, j] \text{ then } \\
\hspace{3cm} m[i, j] = q; \\
\hspace{3cm} s[i, j] = k; \\
\hspace{2cm} \text{end} \\
\hspace{1cm} \text{end} \\
\text{end} \\
\text{return } m \text{ and } s; \text{ (Optimum in } m[i, j])
\]
The Dynamic Programming Algorithm

Matrix-Chain\((p, n)\): // \(l\) is length of sub-chain

\[
\begin{align*}
\text{for } & i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
& \\
\text{for } & l = 2 \text{ to } n \text{ do} \\
& \quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do} \\
& \quad \quad j = i + l - 1; \\
& \quad m[i, j] = \infty; \\
& \quad \text{for } k = i \text{ to } j - 1 \text{ do} \\
& \quad \quad q = m[i, k] + m[k + 1, j] + p[i - 1] \times p[k] \times p[j]; \\
& \quad \quad \text{if } q < m[i, j] \text{ then} \\
& \quad \quad \quad m[i, j] = q; \\
& \quad \quad \quad s[i, j] = k; \\
& \quad \text{end} \\
& \quad \text{end} \\
& \text{end} \\
\text{return } m \text{ and } s; \ (\text{Optimum in } m[1, n])
\end{align*}
\]

Complexity: The loops are nested three levels deep.
Matrix-Chain\((p, n)\), // \(l\) is length of sub-chain

\[
\begin{align*}
  &\text{for } i = 1 \text{ to } n \text{ do } m[i, i] = 0; \\
  &\text{; } \\
  &\text{for } l = 2 \text{ to } n \text{ do } \\
  &\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
  &\quad\quad j = i + l - 1; \\
  &\quad\quad m[i, j] = \infty; \\
  &\quad\quad \text{for } k = i \text{ to } j - 1 \text{ do } \\
  &\quad\quad\quad q = m[i, k] + m[k + 1, j] + p[i - 1] \cdot p[k] \cdot p[j]; \\
  &\quad\quad\quad \text{if } q < m[i, j] \text{ then } \\
  &\quad\quad\quad\quad m[i, j] = q; \\
  &\quad\quad\quad\quad s[i, j] = k; \\
  &\quad\quad\quad \text{end} \\
  &\quad\quad \text{end} \\
  &\quad \text{end} \\
  &\text{return } m \text{ and } s; \text{ (Optimum in } m[1, n])
\end{align*}
\]

**Complexity:** The loops are nested three levels deep. Each loop index takes on \(\leq n\) values.
The Dynamic Programming Algorithm

Matrix-Chain\((p, n)\): // \(l\) is length of sub-chain

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n \text{ do } m[i, i] = 0; \\
&; \\
\text{for } l &= 2 \text{ to } n \text{ do } \\
&\quad \text{for } i &= 1 \text{ to } n - l + 1 \text{ do } \\
&\quad & j = i + l - 1; \\
&\quad & m[i, j] = \infty; \\
&\quad & \text{for } k &= i \text{ to } j - 1 \text{ do } \\
&\quad & \quad q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]; \\
&\quad & \quad \text{if } q < m[i, j] \text{ then } \\
&\quad & \quad \quad m[i, j] = q; \\
&\quad & \quad \quad s[i, j] = k; \\
&\quad & \quad \quad \text{end} \\
&\quad \text{end} \\
&\text{end} \\
\text{return } m \text{ and } s; \text{ (Optimum in } m[1, n])
\end{align*}
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\).
The Dynamic Programming Algorithm

Matrix-Chain\((p, n): // \text{l is length of sub-chain}\)

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do } m[i, i] &= 0; \\
\text{for } l = 2 \text{ to } n \text{ do } \\
\quad \text{for } i = 1 \text{ to } n - l + 1 \text{ do } \\
\quad\quad j &= i + l - 1; \\
\quad\quad m[i, j] &= \infty; \\
\quad\quad \text{for } k = i \text{ to } j - 1 \text{ do } \\
\quad\quad\quad q &= m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]; \\
\quad\quad\quad \text{if } q < m[i, j] \text{ then } \\
\quad\quad\quad\quad m[i, j] &= q; \\
\quad\quad\quad\quad s[i, j] &= k; \\
\quad\quad\quad \text{end} \\
\quad\quad \text{end} \\
\quad \text{end} \\
\text{return } m \text{ and } s; \text{ (Optimum in } m[1, n])
\end{align*}
\]

Complexity: The loops are nested three levels deep. Each loop index takes on \(\leq n\) values. Hence the time complexity is \(O(n^3)\). Space complexity is \(\Theta(n^2)\).