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.

Review of Matrix Multiplication

Matrix: An $n \times m$ matrix $A = [a[i,j]]$ is a two-dimensional array

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

which has n rows and m columns.

Example

A 4×5 matrix:

$$\begin{bmatrix} 12 & 8 & 9 & 7 & 6 \\ 7 & 6 & 89 & 56 & 2 \\ 5 & 5 & 6 & 9 & 10 \\ 8 & 6 & 0 & -8 & -1 \end{bmatrix}.$$

Review of Matrix Multiplication

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

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

Review of Matrix Multiplication

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

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

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.

Review of Matrix Multiplication

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

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

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

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

Review of Matrix Multiplication

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

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

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

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

Remarks on Matrix Multiplication

- Matrix multiplication is associative, e.g.,

Remarks on Matrix Multiplication

- 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.

Remarks on Matrix Multiplication

- 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.,

Remarks on Matrix Multiplication

- 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.,

  $$A_1A_2 \neq A_2A_1$$

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:

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 $A(BC)$

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:

$$\begin{aligned} mult[(AB)C] &= pqr + prs, \\ mult[A(BC)] &= qrs + pqs. \end{aligned}$$

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:

$$\begin{aligned} mult[(AB)C] &= pqr + prs, \\ mult[A(BC)] &= qrs + pqs. \end{aligned}$$

Example

For $p = 5$, $q = 4$, $r = 6$ and $s = 2$,

$$\begin{aligned} mult[(AB)C] &= 180, \\ mult[A(BC)] &= 88. \end{aligned}$$

A big difference!

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:

$$\begin{aligned} mult[(AB)C] &= pqr + prs, \\ mult[A(BC)] &= qrs + pqs. \end{aligned}$$

Example

For $p = 5$, $q = 4$, $r = 6$ and $s = 2$,

$$\begin{aligned} mult[(AB)C] &= 180, \\ mult[A(BC)] &= 88. \end{aligned}$$

A big difference!

Implication: Multiplication "sequence" (parenthesization) is important!!

- Review of matrix multiplication.

Outline

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

Outline

- Review of matrix multiplication.
- The chain matrix multiplication problem.
- A dynamic programming algorithm for chain matrix multiplication.

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, ..., 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, ..., 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

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) = A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\ &= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4) \end{aligned}$$

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, ..., 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

$$
\begin{aligned}
A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) = A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\
&= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4)
\end{aligned}
$$

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

Question

Is there a better approach?

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, …, 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

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) = A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\ &= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4) \end{aligned}$$

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

Question

Is there a better approach?

Yes – DP

Outline

- 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.

Step 1: Define Space of Subproblems

Developing a Dynamic Programming Algorithm

Step 1: Define Space of Subproblems

- Original Problem:

  Determine minimal cost multiplication sequence for $A_{1..n}$.

Step 1: Define Space of Subproblems

- Original Problem:

  Determine minimal cost multiplication sequence for $A_{1..n}$.

- Subproblems: For every pair $1 \le i \le j \le n$:

  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 $1 \le i \le j \le n$:
  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?

Relationships among subproblems

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 multipled together.

Relationships among subproblems

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 multipled together. That is,

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

Relationships among subproblems

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 multipled together. That is,

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

Relationships among subproblems

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 multipled 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)?

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 multipled 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.

Relationships among subproblems

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 multipled 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 $A_{k+1..j}$?

Relationships among subproblems

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 multipled 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*.

Optimal Structure Property

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

Optimal Structure Property

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, constradicting optimality

Optimal Structure Property

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, constradicting 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.

Step 2: Constructing optimal solutions from optimal subproblem solutior

Relationships among subproblems

Step 2: Constructing optimal solutions from optimal subproblem solution

- 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 satisify the following recursive definition.

Relationships among subproblems

Step 2: Constructing optimal solutions from optimal subproblem solution

- 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 satisify the following recursive definition.

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

Step 2: Constructing optimal solutions from optimal subproblem solution

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

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

$$A_{i..j} = A_{i..k} A_{k+1..j}$$

Proof of Recurrence

Proof.

If $j = i$, then $m[i, j] = 0$ because, no mutiplications 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

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

$A_{i..j} = A_{i..k}A_{k+1..j}$ □

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,

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

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

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

□

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

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

Recurrence:

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

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

Recurrence:

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

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:

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

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.

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:

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

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.

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

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

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

Recurrence:

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

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.

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

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

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

Recurrence:

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

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.

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:

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

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.

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 for the Bottom-Up Computation

Example

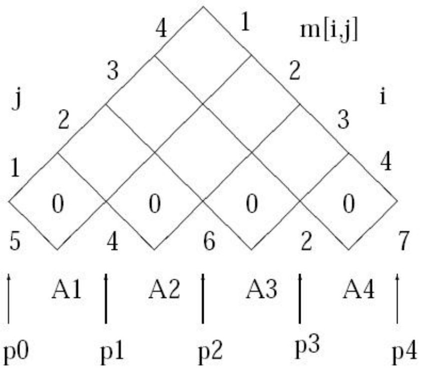
A chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

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



Example – Continued

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

Example – Continued

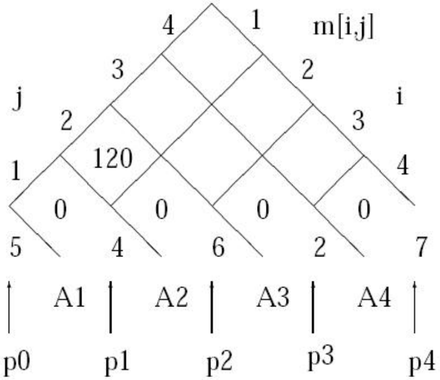
Step 1: Computing $m[1,2]$
By definition

$$\begin{aligned} m[1,2] &= \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2) \\ &= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120. \end{aligned}$$

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

By definition

$$
\begin{aligned}
m[1,2] &= \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2) \\
&= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120.
\end{aligned}
$$



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

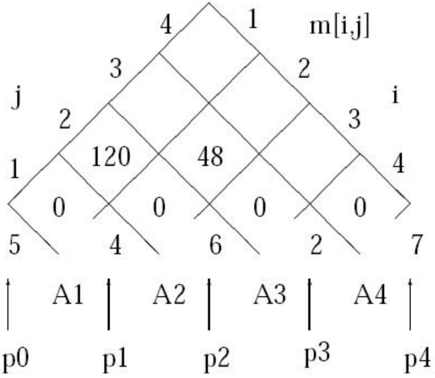
By definition

$$
\begin{aligned}
m[2,3] &= \min_{2 \le k < 3} (m[2,k] + pm[k+1,3] + p_1 p_k p_3) \\
&= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48.
\end{aligned}
$$

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

By definition

$$\begin{aligned} m[2,3] &= \min_{2\le k<3}\left(m[2,k] + pm[k+1,3] + p_1p_kp_3\right) \\ &= m[2,2] + m[3,3] + p_1p_2p_3 = 48. \end{aligned}$$



Example – Continued

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

By definition

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

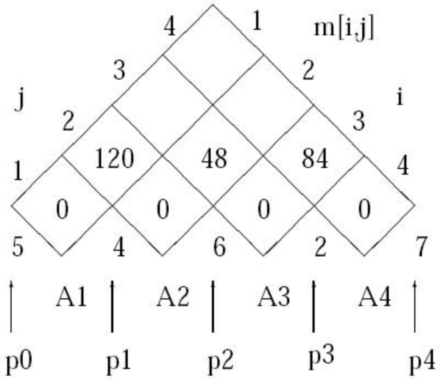
Step 3: Computing $m[3,4]$

By definition

$$\begin{aligned} m[3,4] &= \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4) \\ &= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84. \end{aligned}$$

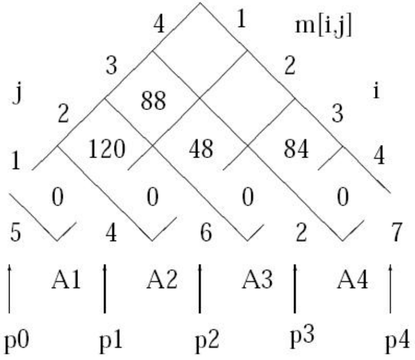
Step 3: Computing $m[3,4]$
By definition

$$\begin{aligned} m[3,4] &= \min_{3\le k<4} (m[3,k] + m[k+1,4] + p_2 p_k p_4) \\ &= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84. \end{aligned}$$



Step 4: Computing $m[1,3]$
By definition

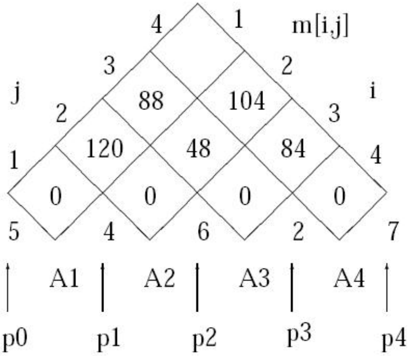
$$\begin{aligned} m[1,3] &= \min_{1\le k<3}(m[1,k]+m[k+1,3]+p_0p_kp_3) \\ &= \min\left\{\begin{array}{l} m[1,1]+m[2,3]+p_0p_1p_3 \\ m[1,2]+m[3,3]+p_0p_2p_3 \end{array}\right\} \\ &= 88. \end{aligned}$$



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

By definition

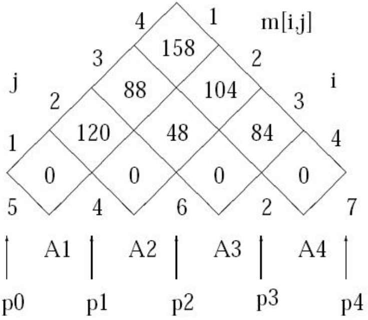
$$\begin{aligned} m[2,4] &= \min_{2\le k<4}\left(m[2,k]+m[k+1,4]+p_1p_kp_4\right) \\ &= \min\left\{\begin{array}{l} m[2,2]+m[3,4]+p_1p_2p_4 \\ m[2,3]+m[4,4]+p_1p_3p_4 \end{array}\right\} \\ &= 104. \end{aligned}$$



Step 6: Computing $m[1,4]$

By definition

$$
\begin{aligned}
m[1,4] &= \min_{1 \le k < 4} \left(m[1,k] + m[k+1,4] + p_0 p_k p_4 \right) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{array} \right\} \\
&= 158.
\end{aligned}
$$



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

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

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 $A_{i..j} = A_{i..k}A_{k+1..j}$.

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 $s[i, j]$?

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 $s[i,j]$?

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

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 $s[i,j]$?

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

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 $s[i, j]$?

$$\begin{array}{ll} s[1,n] & (A_1 \cdots A_{s[1,n]})\,(A_{s[1,n]+1} \cdots A_n) \\ s[1,s[1,n]] & (A_1 \cdots A_{s[1,s[1,n]]})\,(A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\ s[s[1,n]+1,n] & (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]})\,(A_{s[s[1,n]+1,n]+1} \cdots A_n) \end{array}$$

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 $s[i,j]$?

$$\begin{array}{ll}
s[1,n] & (A_1 \cdots A_{s[1,n]})\,(A_{s[1,n]+1} \cdots A_n) \\
s[1,s[1,n]] & (A_1 \cdots A_{s[1,s[1,n]]})\,(A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\
s[s[1,n]+1,n] & (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]})\,(A_{s[s[1,n]+1,n]+1} \cdots A_n) \\
\vdots & \vdots
\end{array}$$

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 $s[i, j]$?

$$
\begin{array}{ll}
s[1, n] & (A_1 \cdots A_{s[1,n]})\,(A_{s[1,n]+1} \cdots A_n) \\
s[1, s[1, n]] & (A_1 \cdots A_{s[1,s[1,n]]})\,(A_{s[1,s[1,n]]+1} \cdots A_{s[1,n]}) \\
s[s[1, n] + 1, n] & (A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]})\,(A_{s[s[1,n]+1,n]+1} \cdots A_n) \\
\vdots & \vdots
\end{array}
$$

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.

$$s[1,6] = 3 \quad (A_1A_2A_3)(A_4A_5A_6)$$

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.

$$
\begin{array}{ll}
s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\
s[1,3] = 1 &
\end{array}
$$

Step 4...

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.

$$\begin{array}{ll} s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\ s[1,3] = 1 & (A_1 (A_2 A_3)) \end{array}$$

Step 4...

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.

$$\begin{array}{ll} s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\ s[1,3] = 1 & (A_1 (A_2 A_3)) \\ s[4,6] = 5 & \end{array}$$

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.

$$\begin{array}{ll} s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\ s[1,3] = 1 & (A_1 (A_2 A_3)) \\ s[4,6] = 5 & ((A_4 A_5) A_6) \end{array}$$

Step 4...

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.

$$\begin{array}{ll} s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\ s[1,3] = 1 & (A_1 (A_2 A_3)) \\ s[4,6] = 5 & ((A_4 A_5) A_6) \end{array}$$

Hence the final multiplication sequence is

Step 4...

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.

$$\begin{array}{ll} s[1,6] = 3 & (A_1 A_2 A_3)(A_4 A_5 A_6) \\ s[1,3] = 1 & (A_1(A_2 A_3)) \\ s[4,6] = 5 & ((A_4 A_5) A_6) \end{array}$$

Hence the final multiplication sequence is

$$(A_1(A_2 A_3))((A_4 A_5) A_6).$$

The Dynamic Programming Algorithm

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

for $i = 1$ **to** n **do** $m[i, i] =$

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 $l =$

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**

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**

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**
    $j =$

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] =
```

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] = ∞;
        for k =
```

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**

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**
            $q =$

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**
            $q = m[i, k] +$

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**
      $q = m[i, k] + m[k + 1, j] +$

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] = ∞;
        for k = i to j − 1 do
            q = m[i, k] + m[k + 1, j] + p[i − 1] *
```

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] = ∞;
        for k = i to j − 1 do
            q = m[i, k] + m[k + 1, j] + p[i − 1] ∗ p[k] ∗
```

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**

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

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] = ∞;
        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] =
```

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**

        $m[i, j] = q$;

        $s[i, j] =$

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] = ∞;
        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
end
return
```

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] = ∞;
        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
end
return m and s; (Optimum in
```

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] = ∞;
        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
end
return m and s; (Optimum in m[1, n])
```

Complexity: The loops are nested three levels deep.

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] = ∞;
        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
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.

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] = ∞;
        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
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 $O(n^3)$.

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] = ∞;
        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
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 $\Theta(n^2)$.