Matrix Chain Multiplication

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Matrix chain multiplication

- **Matrix chain multiplication** (or **Matrix Chain** Ordering Problem, MCOP) is an **optimization problem** that can be solved using dynamic programming.
- Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
- For example, if the chain of matrices is ***(A1, A2, A3, A4)*** then we can fully parenthesize the product ***(A1A2A3A4)*** in **five** distinct ways:

1:-(A1(A2(A3A4))) , 2:-(A1((A2A3)A4)),

3:- ((A1A2)(A3A4)), 4:-((A1(A2A3))A4), 5:-(((A1A2)A3)A4)

Recalling 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: The following is a 4×5 matrix:

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

Recalling 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 given by

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

for $1 \leq i \leq p$ and $1 \leq j \leq r$.

Example: If

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

then

$$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}.$$

Remarks on Matrix Multiplication

- If AB is defined, BA may not be defined.
- Quite possible that $AB \neq BA$.
- Multiplication is recursively defined by

$$A_1 A_2 A_3 \cdots A_{s-1} A_s = A_1(A_2(A_3 \cdots (A_{s-1} A_s))).$$

- 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 parenthenization does not change result.

Direct Matrix Multiplication

MATRIX-MULTIPLY (A,B)

if *columns* [A] ≠ *rows* [B]

then error "incompatible dimensions"

else for $i \leftarrow 1$ **to** *rows* [A]

do for $j \leftarrow 1$ **to** *columns* [B]

do $C[i, j] \leftarrow 0$

for $k \leftarrow 1$ **to** *columns* [A]

do $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$

return C

Direct Matrix Multiplication A & B

Given a $p \times q$ matrix A and a $q \times r$ matrix B, the direct way of multiplying $C = AB$ is to compute each

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

for $1 \leq i \leq p$ and $1 \leq j \leq r$.

Complexity of Direct 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.

Direct Matrix Multiplication A B C

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

The number of multiplications needed are:

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

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

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

A big difference!

Implication: The multiplication "sequence" (parenthesization) is important!!

Matrix-chain multiplication(MCM)

- **Problem:** given $<A_1, A_2, \ldots, A_n>$, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.
- Suppose two matrices $A(p,q)$ and $B(q,r)$, compute their product $C(p,r)$ in $p \times q \times r$ multiplications.

Matrix-Chain multiplication

- Matrix multiplication is assosiative, and so all parenthesizations yield the same product.
- For example, if the chain of matrices is $\langle A_1 A_2 A_3 A_4 \rangle$ then the product $\langle A_1 A_2 \ldots A_4 \rangle$ can be fully paranthesized in **five** distinct way:

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

Matrix-Chain multiplication

- A_1 is 10 by 100 matrix
- A_2 is 100 by 5 matrix
- A_3 is 5 by 50 matrix
- A_4 is 50 by 1 matrix
- $A_1A_2A_3A_4$ is a 10 by 1 matrix

5 different orderings = 5 different parenthesizations

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

Each parenthesization is a different number of mults

Let $A_{ij} = A_i \cdots A_j$

Matrix-Chain multiplication

- A_1 is 10 by 100 matrix, A_2 is 100 by 5 matrix, A_3 is 5 by 50 matrix, A_4 is 50 by 1 matrix, $A_1A_2A_3A_4$ is a 10 by 1 matrix.
- $(A_1(A_2(A_3A_4)))$
  - $A_{34} = A_3A_4$, 250 mults, result is 5 by 1
  - $A_{24} = A_2A_{34}$, 500 mults, result is 100 by 1
  - $A_{14} = A_1A_{24}$, 1000 mults, result is 10 by 1
  - Total is 1750
- $((A_1A_2)(A_3A_4))$
  - $A_{12} = A_1A_2$, 5000 mults, result is 10 by 5
  - $A_{34} = A_3A_4$, 250 mults, result is 5 by 1
  - $A_{14} = A_{12}A_{34})$, 50 mults, result is 10 by 1
  - Total is 5300
- $(((A_1A_2)A_3)A_4)$
  - $A_{12} = A_1A_2$, 5000 mults, result is 10 by 5
  - $A_{13} = A_{12}A_3$, 2500 mults, result is 10 by 50
  - $A_{14} = A_{13}A_4$, 500 mults, results is 10 by 1
  - Total is 8000

Matrix-Chain multiplication

- A_1 is 10 by 100 matrix, A_2 is 100 by 5 matrix, A_3 is 5 by 50 matrix, A_4 is 50 by 1 matrix, $A_1A_2A_3A_4$ is a 10 by 1 matrix.
- $((A_1(A_2A_3))A_4)$
  - $A_{23} = A_2A_3$, 25000 mults, result is 100 by 50
  - $A_{13} = A_1A_{23}$, 50000 mults, result is 10 by 50
  - $A_{14} = A_{13}A_4$, 500 mults, results is 10 by
  - Total is 75500
- $(A_1((A_2A_3)A_4))$
  - $A_{23} = A_2A_3$, 25000 mults, result is 100 by 50
  - $A_{24} = A_{23}A_4$, 5000 mults, result is 100 by 1
  - $A_{14} = A_1A_{24}$, 1000 mults, result is 10 by 1
  - Total is 31000

Conclusion Order of operations makes a huge difference. How do we compute the minimum?

The Chain Matrix Multiplication Problem

Parenthesization A product of matrices is fully parenthesized if it is either

- a single matrix, or
- a product of two fully parenthesized matrices, surrounded by parentheses

Each parenthesization defines a set of n-1 matrix multiplications. We just need to pick the parenthesization that corresponds to the best ordering.

How many parenthesizations are there?

Let P(n) be the number of ways to parenthesize n matrices.

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

This recurrence is related to the Catalan numbers, and solves to

$$P(n) = \Omega(4^n/n^{3/2}).$$

Conclusion Trying all possible parenthesizations is a bad idea.

Dynamic Programming approach

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution bottom-up
4. Construct an optimal solution from the computed information

Structure of an optimal solution If the outermost parenthesization is

$$((A_1 A_2 \cdots A_i)(A_{i+1} \cdots A_n))$$

then the optimal solution consists of solving A_{1i} and $A_{i+1,n}$ optimally and then combining the solutions.

Proof

Structure of an optimal solution If the outermost parenthesization is

$$((A_1 A_2 \cdots A_i)(A_{i+1} \cdots A_n))$$

then the optimal solution consists of solving A_{1i} and $A_{i+1,n}$ optimally and then combining the solutions.

Proof: Consider an optimal algorithm that does not solve A_{1i} optimally. Let x be the number of multiplications it does to solve A_{1i}, y be the number of multiplications it does to solve $A_{i+1,n}$, and z be the number of multiplications it does in the final step. The total number of multiplications is therefore

$$x + y + z.$$

But since it is not solving A_{1i} optimally, there is a way to solve A_{1i} using $x' < x$ multiplications. If we used this optimal algorithm instead of our current one for A_{1i}, we would do

$$x' + y + z < x + y + z$$

multiplications and therefore have a better algorithm, contradicting the fact that our algorithms is optimal.

Proof

Proof: Consider an optimal algorithm that does not solve A_{1i} optimally. Let x be the number of multiplications it does to solve A_{1i}, y be the number of multiplications it does to solve $A_{i+1,n}$, and z be the number of multiplications it does in the final step. The total number of multiplications is therefore

$$x + y + z.$$

But since it is not solving A_{1i} optimally, there is a way to solve A_{1i} using $x' < x$ multiplications. If we used this optimal algorithm instead of our current one for A_{1i}, we would do

$$x' + y + z < x + y + z$$

multiplications and therefore have a better algorithm, contradicting the fact that our algorithms is optimal.

Meta-proof that is not a correct proof Our problem consists of subproblems, assume we didn't solve the subproblems optimally, then we could just replace them with an optimal subproblem solution and have a better solution.

Recursive solution

In the enumeration of the $P(n) = \Omega(4^n/n^{3/2})$ subproblems, how many unique subproblems are there?

Answer: A subproblem is of the form A_{ij} with $1 \leq i, j \leq n$, so there are $O(n^2)$ subproblems!

Notation

- Let A_i be p_{i-1} by p_i.
- Let $m[i, j]$ be the cost of computing A_{ij}

If the final multiplication for A_{ij} is $A_{ij} = A_{ik}A_{k+1,j}$ then

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

We don't know k a priori, so we take the minimum

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Direct recursion on this does not work! We must use the fact that there are at most $O(n^2)$ different calls. What is the order?

Developing a Dynamic Programming Algorithm

Step 1: Determine the structure of an optimal solution (in this case, a parenthesization).

Decompose the problem into subproblems: For each pair $1 \leq i \leq j \leq n$, determine the multiplication sequence for $A_{i..j} = A_i A_{i+1} \cdots A_j$ that minimizes the number of multiplications.

Clearly, $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

Original Problem: determine sequence of multiplication for $A_{1..n}$.

Developing a Dynamic Programming Algorithm

Step 1: Determine the structure of an optimal solution (in this case, a parenthesization).

High-Level Parenthesization for $A_{i..j}$

For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k. That is,

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

Example

$$A_{3..6} = (A_3(A_4A_5))(A_6) = A_{3..5}A_{6..6}.$$

Here $k = 5$.

Developing a Dynamic Programming Algorithm

Step 1 – Continued: Thus the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

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

  (Search all possible values of k)

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

  (Problem has optimal substructure property that $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively)

Developing a Dynamic Programming Algorithm

Step 1 – Continued:

Optimal Substructure Property: If final "optimal" solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at final step then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in final optimal solution must also be optimal for the subproblems "standing alone":

If parenthisization of $A_{i..k}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, leading to a contradiction.

Similarly, if parenthisization of $A_{k+1..j}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, also leading to a contradiction.

Developing a Dynamic Programming Algorithm

Step 2: Recursively define the value of an optimal solution.

As with the 0-1 knapsack problem, we will store the solutions to the subproblems in an array.

For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive definition.

Developing a Dynamic Programming Algorithm

Step 2: Recursively define the value of an optimal solution.

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

Proof: Any optimal sequence of multiplication for $A_{i..j}$ is equivalent to some choice of splitting

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

for some k, where the sequences of multiplications for $A_{i..k}$ and $A_{k+1..j}$ also are optimal. Hence

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

Developing a Dynamic Programming Algorithm

Step 2 – Continued: We know that, for some k

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

We don't know what k is, though

But, there are only $j - i$ possible values of k so we can check them all and find the one which returns a smallest cost.

Therefore

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

Developing a Dynamic Programming Algorithm

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.
$m[i, j]$ only defi ned for $i \leq j$.

The important point is that when we use the equation

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

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k + 1, j]$. For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$. Hence, the algorithm should fi ll the table in increasing order of the length of the matrix-chain.

Developing a Dynamic Programming Algorithm

That is, we calculate in the order

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

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

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

$$\vdots$$

$$m[1,n-1], m[2,n]$$

$$m[1,n]$$

Developing a Dynamic Programming Algorithm

When designing a dynamic programming algorithm there are two parts:

1. Finding an appropriate optimal substructure property and corresponding recurrence relation on table items. Example:

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

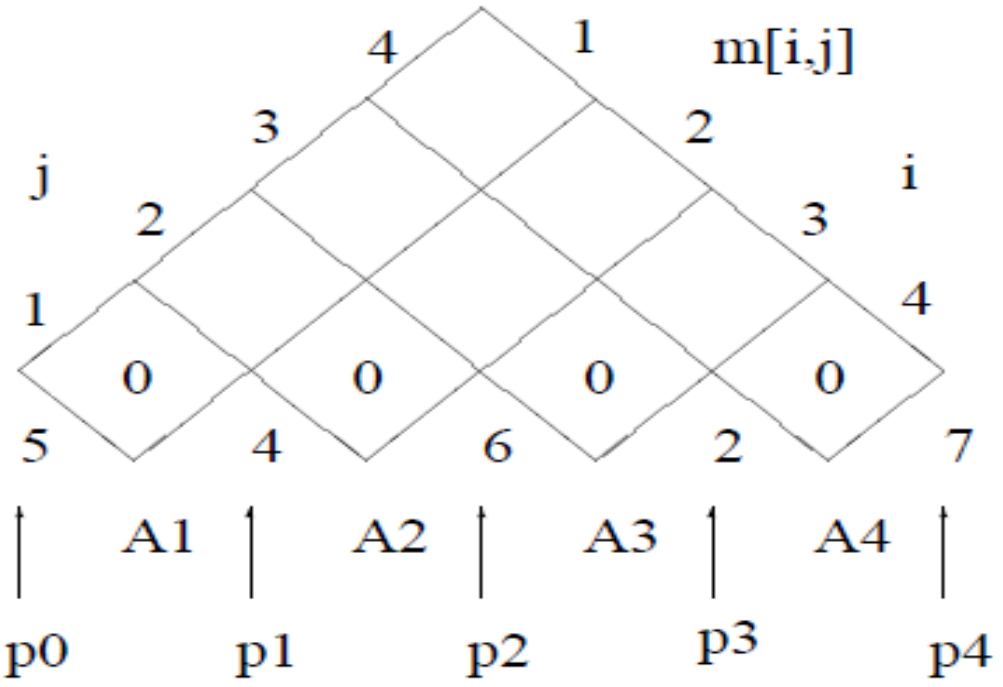
2. Filling in the table properly.
This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated.

   In our example this means that by the time $m[i,j]$ is calculated all of the values $m[i,k]$ and $m[k+1,j]$ were already calculated.

Example for the Bottom-Up Computation

Example: Given 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 $m[1,4]$.

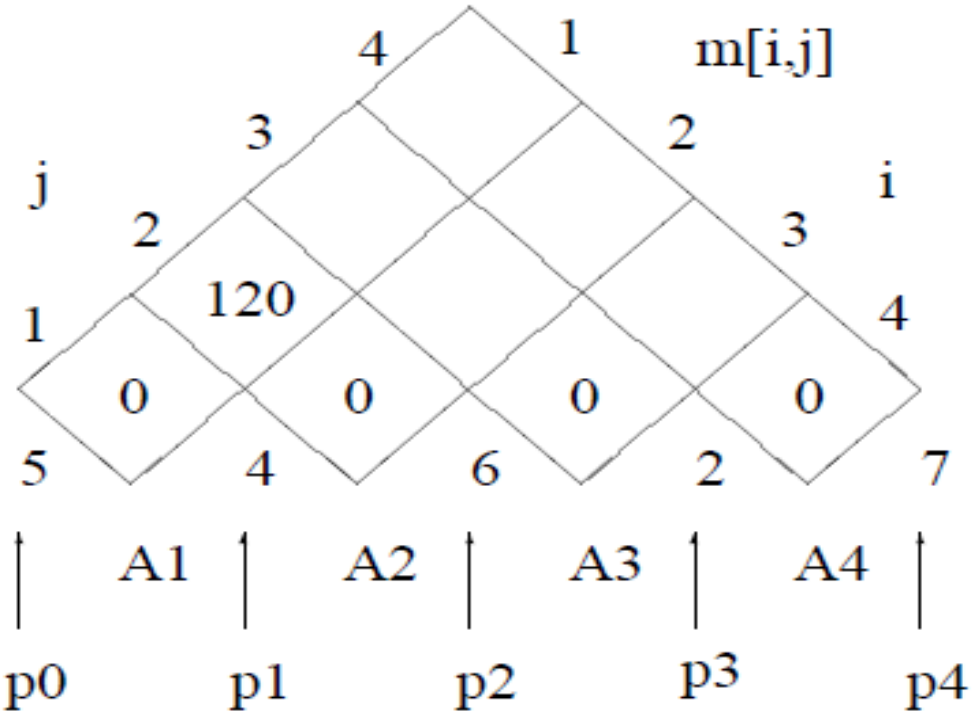
S0: Initialization



Example for the Bottom-Up Computation (cont.)

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

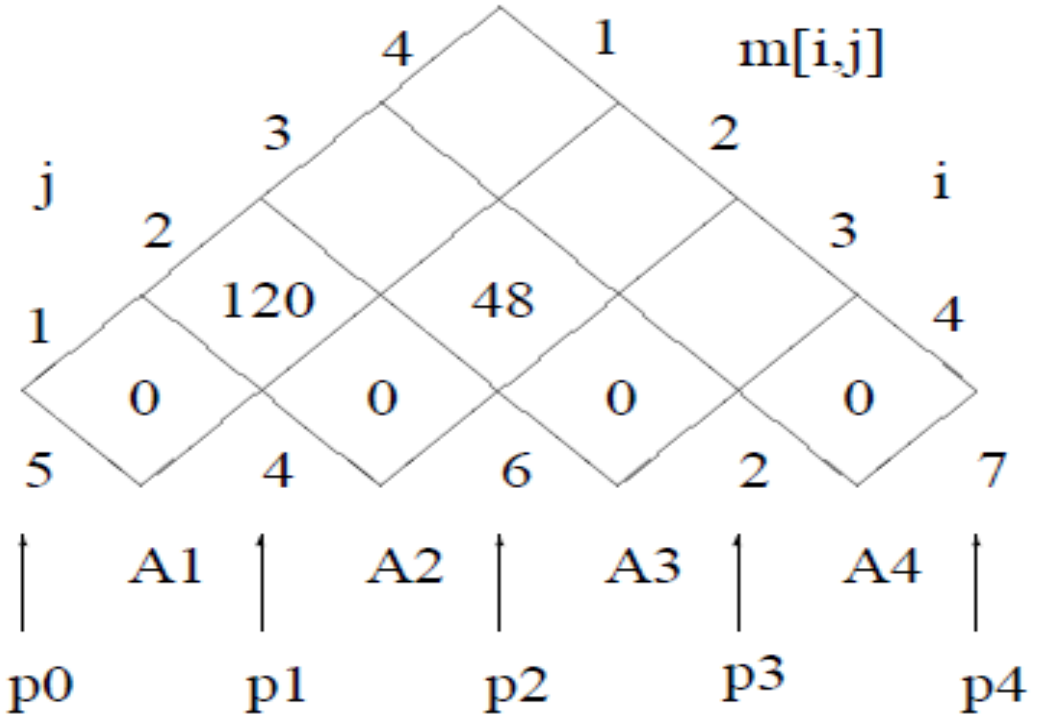
$$m[1,2] = \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2)$$
$$= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120.$$



Example for the Bottom-Up Computation (cont.)

Stp 2: Computing $m[2,3]$ By definition

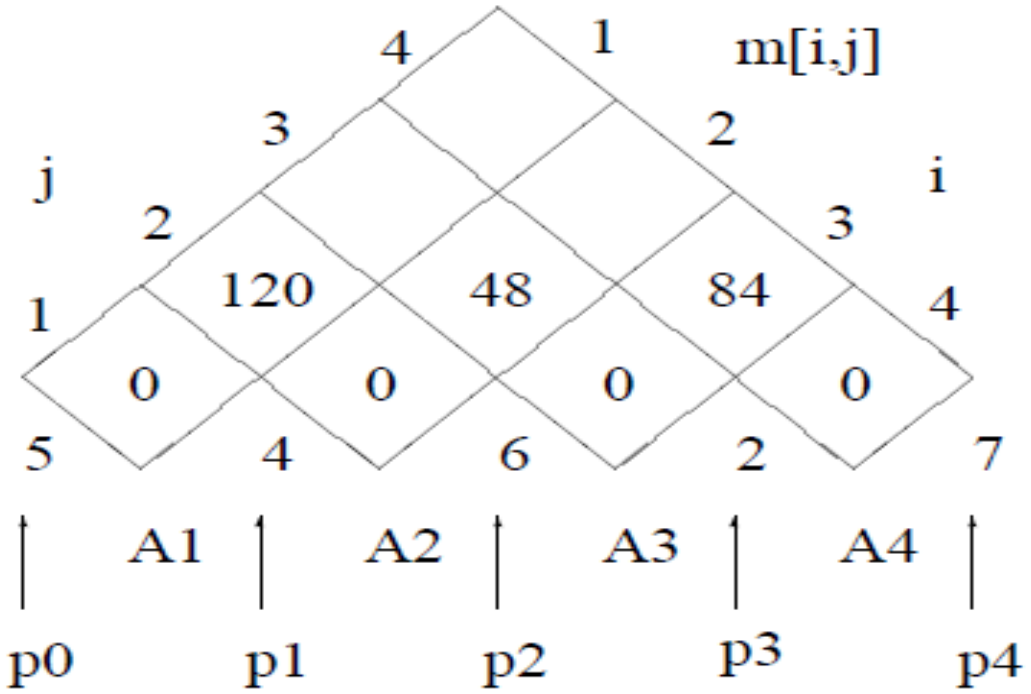
$$m[2,3] = \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1 p_k p_3)$$
$$= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48.$$



Example for the Bottom-Up Computation (cont.)

Stp3: Computing $m[3,4]$ By definition

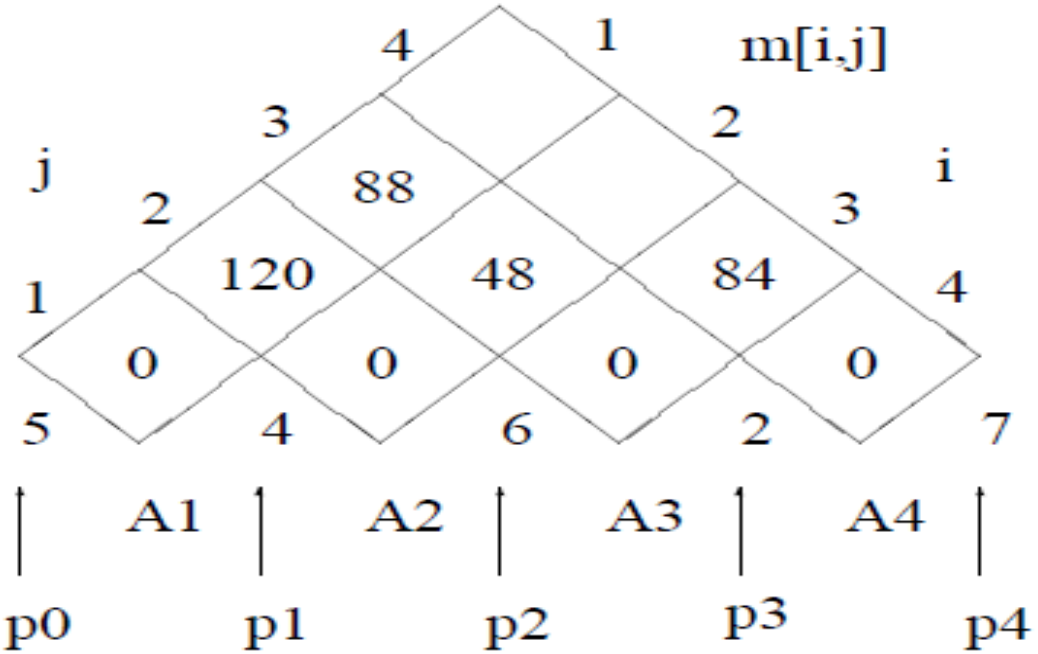
$$m[3,4] = \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4)$$
$$= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84.$$



Example for the Bottom-Up Computation (cont.)

Stp4: Computing $m[1,3]$ By definition

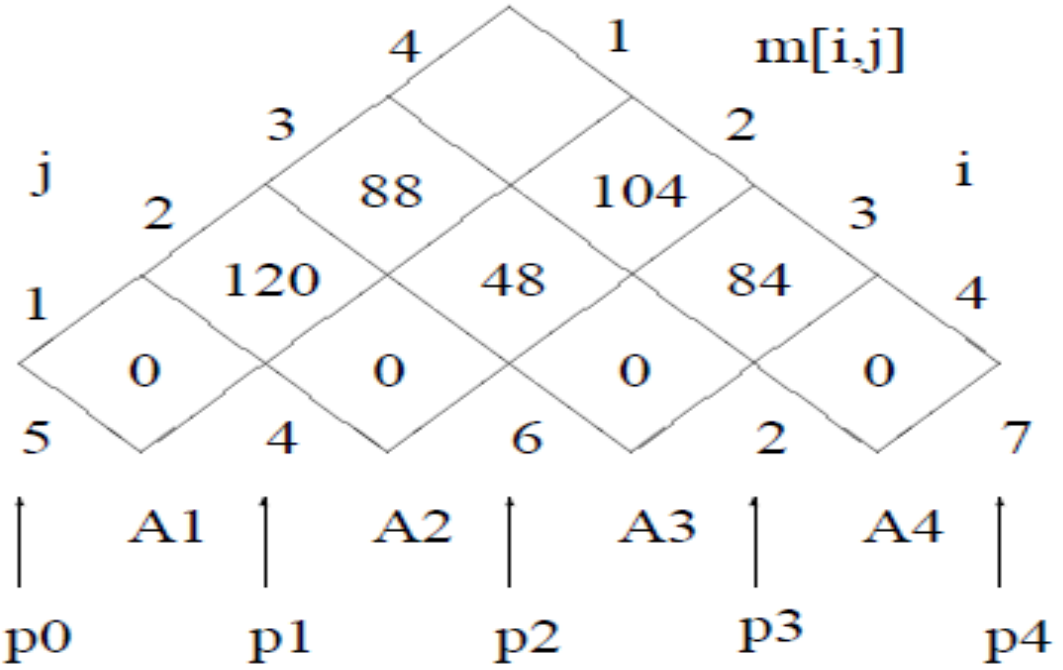
$$
\begin{aligned}
m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\
&= 88.
\end{aligned}
$$



Example for the Bottom-Up Computation (cont.)

Stp5: Computing $m[2,4]$ By definition

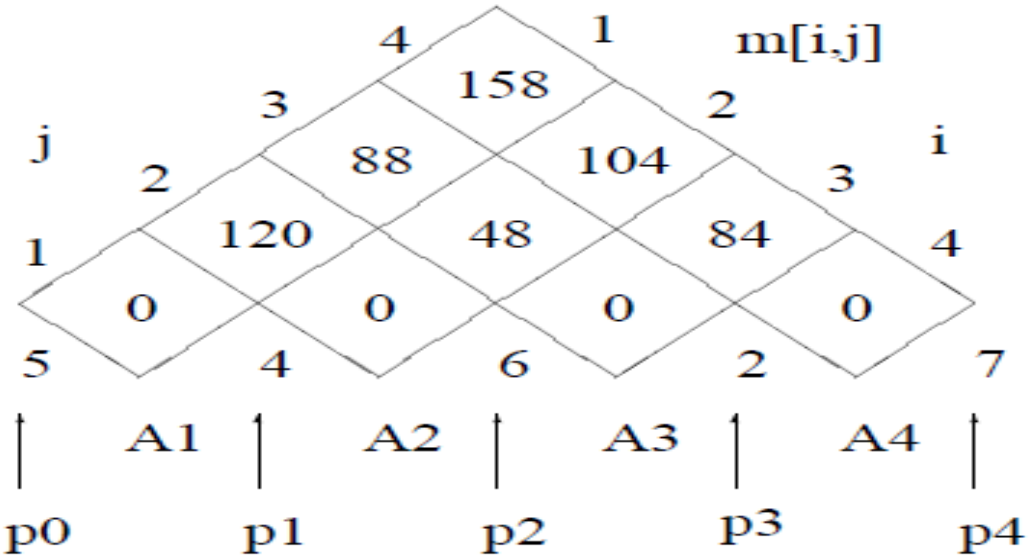
$$
\begin{aligned}
m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{array} \right\} \\
&= 104.
\end{aligned}
$$



Example for the Bottom-Up Computation (cont.)

St6: Computing $m[1,4]$ By definition

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



We are done!

Developing a Dynamic Programming Algorithm

Step 4: Construct an optimal solution from computed information – extract the actual sequence.

Idea: Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes k for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$. The array $s[1..n, 1..n]$ can be used recursively to recover the multiplication sequence.

How to Recover the Multiplication Sequence?

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

Do this recursively until the multiplication sequence is determined.

Developing a Dynamic Programming Algorithm

Step 4: Construct an optimal solution from computed information – extract the actual sequence.

Example of Finding the Multiplication Sequence:

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

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

Hence the final multiplication sequence is

$$(A_1(A_2A_3))((A_4A_5)A_6).$$

The Dynamic Programming Algorithm

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

The Dynamic Programming Algorithm

```
Matrix-Chain(p, n)
{   for (i = 1 to n) m[i, i] = 0;
    for (l = 2 to n)
    {
        for (i = 1 to n − l + 1)
        {
            j = i + l − 1;
            m[i, j] = ∞;
            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;
                }
            }
        }
    }
    return m and s; (Optimum in m[1, n])
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.

Algorithm to Print the result:

```
PRINT-OPTIMAL-PARENS (s, i, j)
1   if i=j
2       then print "A_i"
3       else print " ( "
4            PRINT-OPTIMAL-PARENS (s, i, s[i,j])
5            PRINT-OPTIMAL-PARENS (s, s[i,j]+1, j)
6            Print " ) "
```

Matrix-Chain multiplication

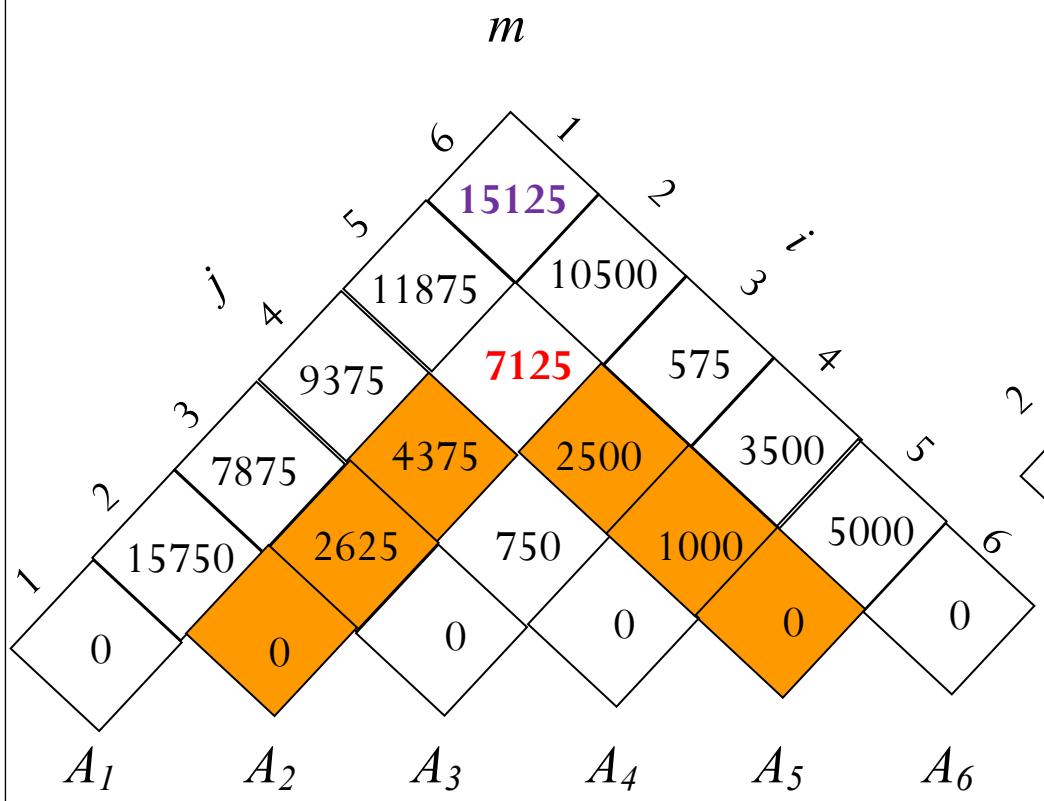
An example:

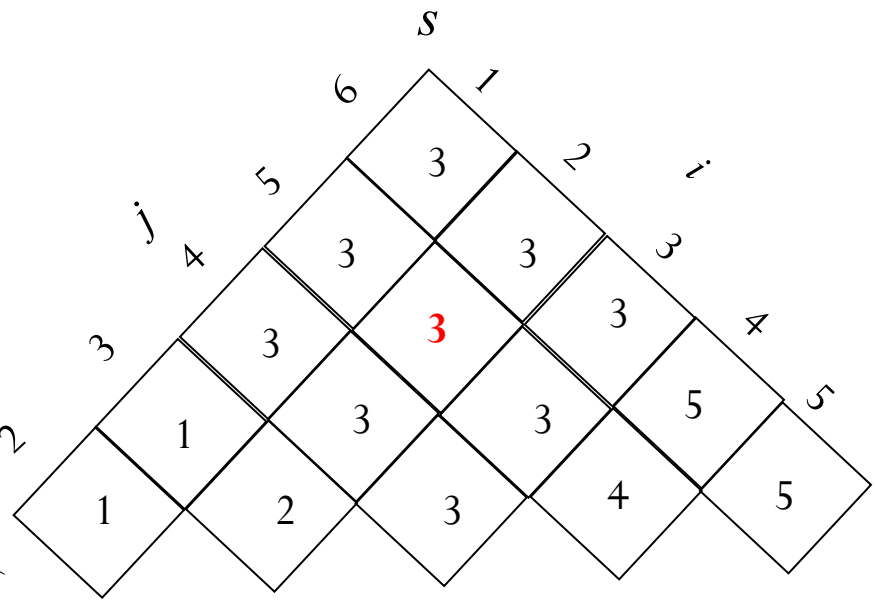
matrix	dimension
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

Matrix-Chain multiplication (cont.)





$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

Constructing an Optimal Solution: Compute $A_{1..n}$

The actual multiplication code uses the $s[i, j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i, j]$ is global to this recursive procedure. The procedure returns a matrix.

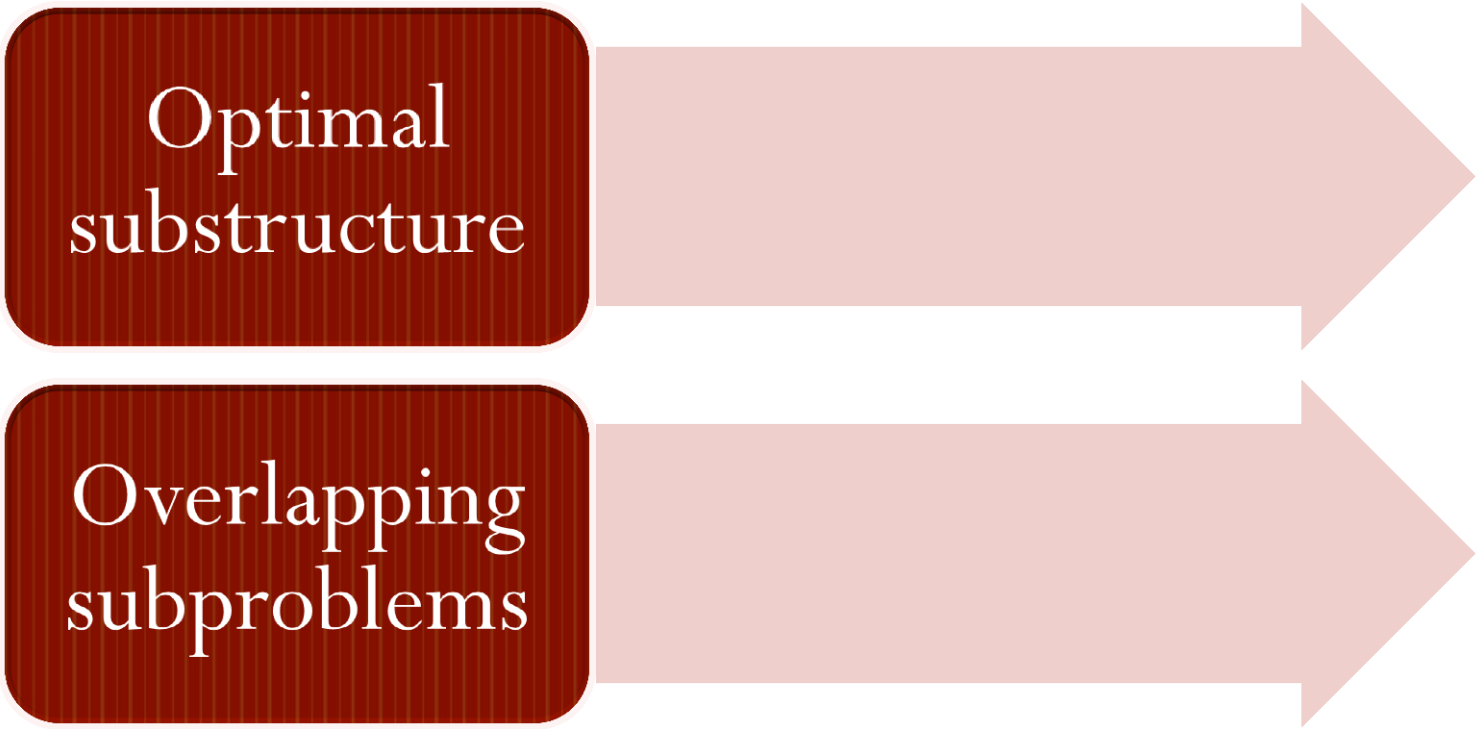
```
Mult(A, s, i, j)
{
    if (i < j)
    {
        X = Mult(A, s, i, s[i, j]);
            X is now A_i ··· A_k, where k is s[i, j]
        Y = Mult(A, s, s[i, j] + 1, j);
            Y is now A_{k+1} ··· A_j
        return X * Y;   multiply matrices X and Y
    }
    else return A[i];
}
```

To compute $A_1 A_2 \cdots A_n$, call Mult$(A, s, 1, n)$.

Elements of Dynamic Programming

When should we apply the method of Dynamic Programming?

Two key ingredients:



Elements of dynamic programming (cont.)

Optimal substructure(OS):

- A problem exhibits os if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits os, it is a good clue that dynamic programming might apply.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.
- Dynamic programming uses optimal substructure in a bottom-up fashion.

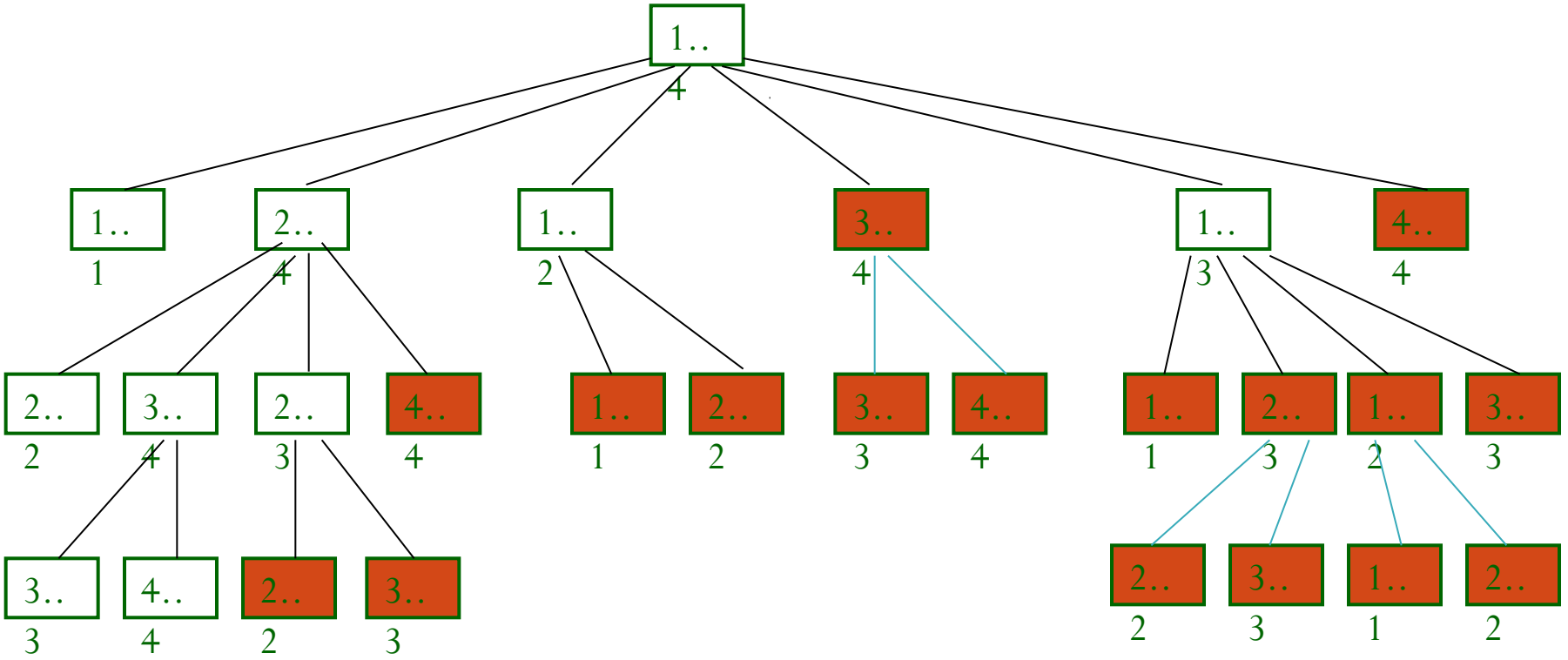
Elements of dynamic programming (cont.)

Overlapping subproblems:

- When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has ***overlapping subproblems***.
- In contrast , a ***divide-and-conquer*** approach is suitable usually generates brand new problems at each step of recursion.
- Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.

Elements of dynamic programming (cont.)

Overlapping subproblems: (cont.)



The recursion tree of RECURSIVE-MATRIX-CHAIN(*p*, 1, 4). The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(*p*, 1, 4).

Recursive Matrix-Chain multiplication

RECURSIVE-MATRIX-CHAIN (p, i, j)

1 **if** $i = j$

2 **then return** 0

3 $m[i,j] \leftarrow \infty$

4 **for** $k \leftarrow i$ **to** j-1

5 **do** $q \leftarrow$ RECURSIVE-MATRIX-CHAIN (p, i, k)

+ RECURSIVE-MATRIX-CHAIN (p, k+1, j)+ $\boldsymbol{p_{i-1} p_k p_j}$

6 **if** $q < m[i,j]$

7 **then** $m[i,j] \leftarrow q$

8 **return** $\boldsymbol{m[i,j]}$

Elements of dynamic programming (cont.)

Overlapping subproblems: (cont.)

Time to compute $m[\,i,j]$ by RECURSIVE-MATRIX-CHAIN:

We assume that the execution of lines 1-2 and 6-7 take at least unit time.

$$T(1) \geq 1,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1$$

$$= 2 \sum_{i=1}^{n-1} T(i) + n$$

Elements of dynamic programming (cont.)

Overlapping subproblems: (cont.)

WE guess that $T(n) = \Omega(2^n)$.

Using the substitution method with $T(n) = 2^{n-1}$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1} 2^i \\
&= 2\sum_{i=0}^{n-2} 2^i + n \\
&= 2(2^{n-1} - 1) + n \\
&\geq 2^{n-1}
\end{aligned}
$$

Matrix Chain Multiplication (Memoization)

- There is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy.
- The idea is to ***memoize*** the the natural, but inefficient, recursive algorithm.
- We maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.
- An entry in a table for the solution to each subproblem is maintained.

Matrix Chain Multiplication (Memoization)

- Eech table entry initially contains a special value to indicate that the entry has yet to be filled.
- When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table.
- Each subsequent time that the problem is encountered, the value stored in the table is simply looked up and returned.

Matrix Chain Multiplication (Memoization)

```
1   MEMOIZED-MATRIX-CHAIN(p)
2   n←length[p]-1
3   for i←1  to  n
4       do for j←i  to  n
           do  m[i,j] ←∞
return LOOKUP-CHAIN(p,1,n)
```

Matrix Chain Multiplication (Memoization)

```
LOOKUP-CHAIN(p,1,n)
1    if   m[i,j] < ∞
2         then return m[i,j]
3    if i=j
4         then m[i,j] ←0
5         else for k←1  to  j-1
6              do q← LOOKUP-CHAIN(p,i,k)
                          + LOOKUP-CHAIN(p,k+1,j) + p_{i-1} p_k p_j
7              if q < m[i,j]
8                   then  m[i,j] ←q
9    return m[i,j]
```

Example

$$\underset{30 \times 1}{A_1} \cdot \underset{1 \times 40}{A_2} \cdot \underset{40 \times 10}{A_3} \cdot \underset{10 \times 25}{A_4}$$

Solution for m and s

m	1	2	3	4
1	0	1200	700	1400
2		0	400	650
3			0	10 000
4				0

s	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

Optimal Parenthesisation

$$A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$


Thanks for Your Attention!

Exercises

Exercise

1. What will be the time complexity of the matrix chain multiplication using dynamic programming?
2. What are 2 things required in order to successfully use the dynamic programming technique?
3. What happens when a top down approach of dynamic programming is applied to any problem?
4. Describe a dynamic programming algorithm to find the maximum product of a contiguous sequence of positive numbers A[1..n].
5. Given a sequence of matrices such that any *matrix* may be multiplied by the previous matrix, find the best association such that the result is obtained with the minimum number of arithmetic operations.

The Matrix Product Parenthesization

- Suppose we need to multiply a series of matrices: **A1 × A2 × A3 × . . . × An**. Given the dimensions of these matrices, what is the best way to parenthesize them, assuming for simplicity that standard matrix multiplication is to be used.

Cost of the matrix multiplication:

Example $\Rightarrow$ $\langle A_1 A_2 A_3 \rangle$

$A_1 : \quad 10 \times 100$

$A_2 : \quad 100 \times 5$

$A_3 : \quad 5 \times 50$

Matrix-Chain multiplication

If we multiply $((A_1A_2)A_3)$ we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications.

If we multiply $(A_1(A_2A_3))$ we perform $100 \cdot 5 \cdot 50 = 25\,000$ scalar multiplications to compute the 100×50 matrix product A_2A_3, plus another $10 \cdot 100 \cdot 50 = 50\,000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75 000 scalar multiplications.

The Chain Matrix Multiplication Problem

- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.
- Let P(n) denote the number of alternative parenthesizations of a sequence of n matrices:

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

- The solution to the recursion is $\Omega(2^n)$. So brute-force will not work.