Lecture 12: Chain Matrix Multiplication

CLRS Section 15.2

Outline of this Lecture

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

$\boxed{\textbf{Recalling Matrix Multiplication}}$

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 $1 \le i \le p$ and $1 \le j \le r$.

Example: If

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

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

$$\begin{aligned} & A_1 A_2 A_3 \cdots A_{s-1} A_s \\ & \quad = A_1(A_2(A_3 \cdots (A_{s-1} A_s))). \end{aligned}$$

- 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 AB

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 $1 \le i \le p$ and $1 \le j \le r$.

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 of ABC

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!!

The Chain Matrix Multiplication Problem

Given

dimensions $p_0, p_1, \ldots, p_n$

corresponding to matrix sequence A_1, A_2, ..., A_n

where A_i has dimension $p_{i-1} \times p_i$,

determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1 A_2 \cdots A_n$. That is, determine how to parenthisize the multiplications.

$$\begin{aligned} A_1 A_2 A_3 A_4 &= (A_1 A_2)(A_3 A_4) \\ &= A_1(A_2(A_3 A_4)) = A_1((A_2 A_3) A_4) \\ &= ((A_1 A_2) A_3)(A_4) = (A_1(A_2 A_3))(A_4) \end{aligned}$$

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

Question: Any better approach? Yes – DP

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

$$A_{3..6} = (A_3(A_4 A_5))(A_6) = A_{3..5} A_{6..6}.$$

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 $1 \le i \le j \le n$, 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 defined 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 fill the table in increasing order of the length of the matrix-chain.

That is, we calculate in the order

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

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

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

$\vdots$

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

$m[1, n]$

Dynamic Programming Design Warning!!

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:

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

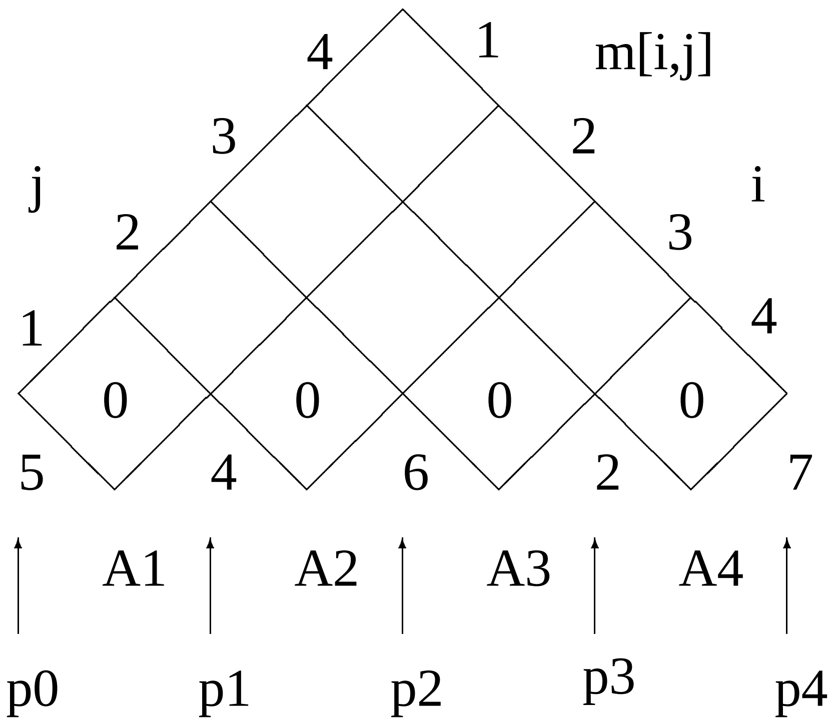
2. Filling in the table properly.
   This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated.

   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 $m[1, 4]$.

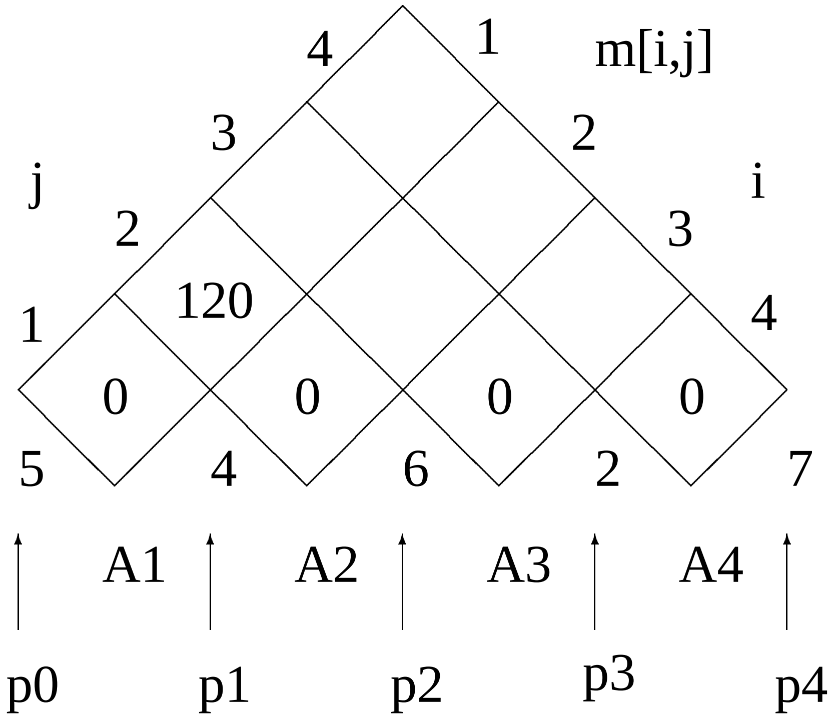
S0: Initialization



Example – Continued

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

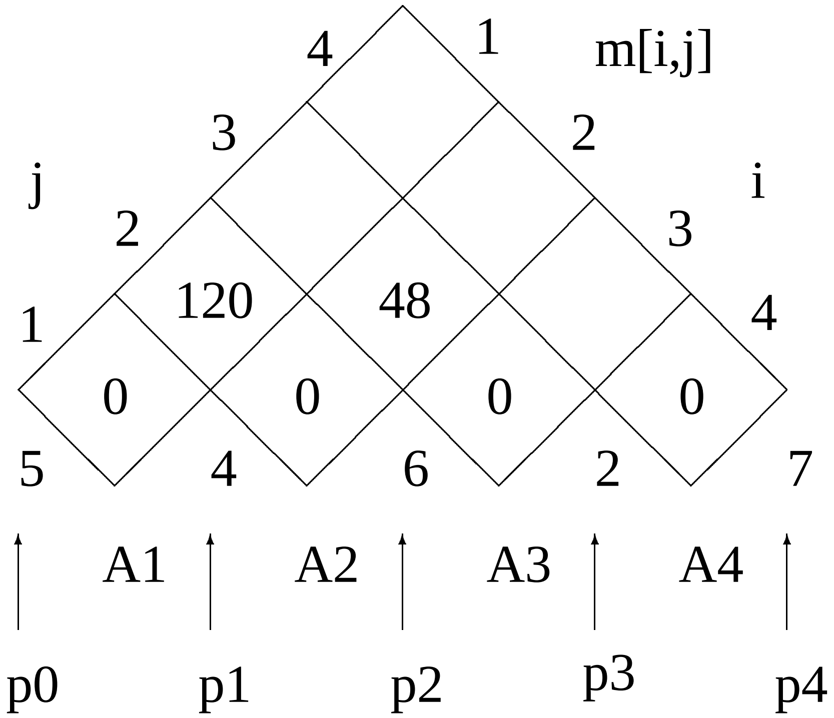
$$
\begin{aligned}
m[1,2] &= \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2) \\
&= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120.
\end{aligned}
$$



Example – Continued

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

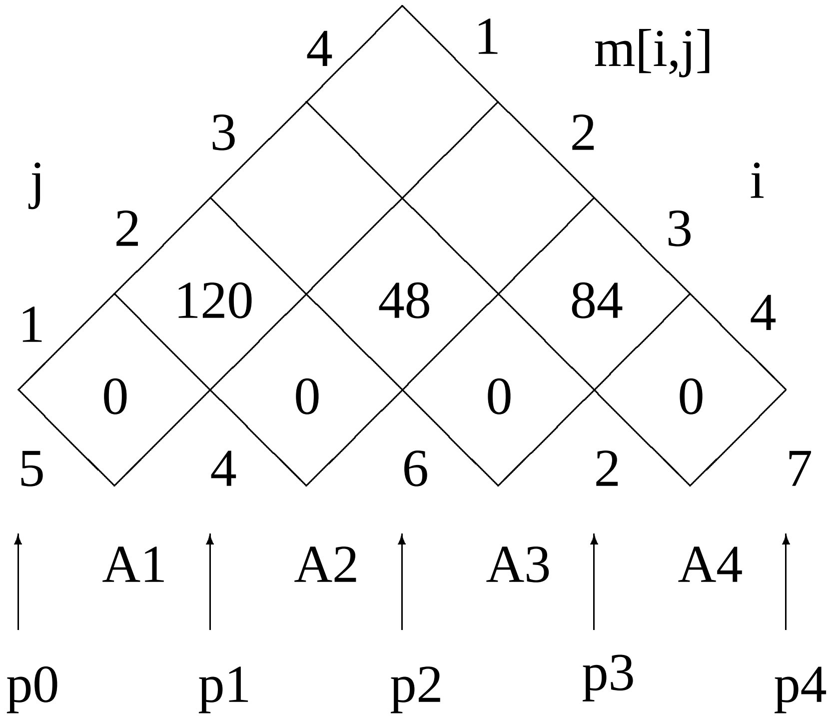
$$\begin{aligned} m[2,3] &= \min_{2\le k<3}\left(m[2,k]+m[k+1,3]+p_1p_kp_3\right) \\ &= m[2,2]+m[3,3]+p_1p_2p_3 = 48. \end{aligned}$$



Example – Continued

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

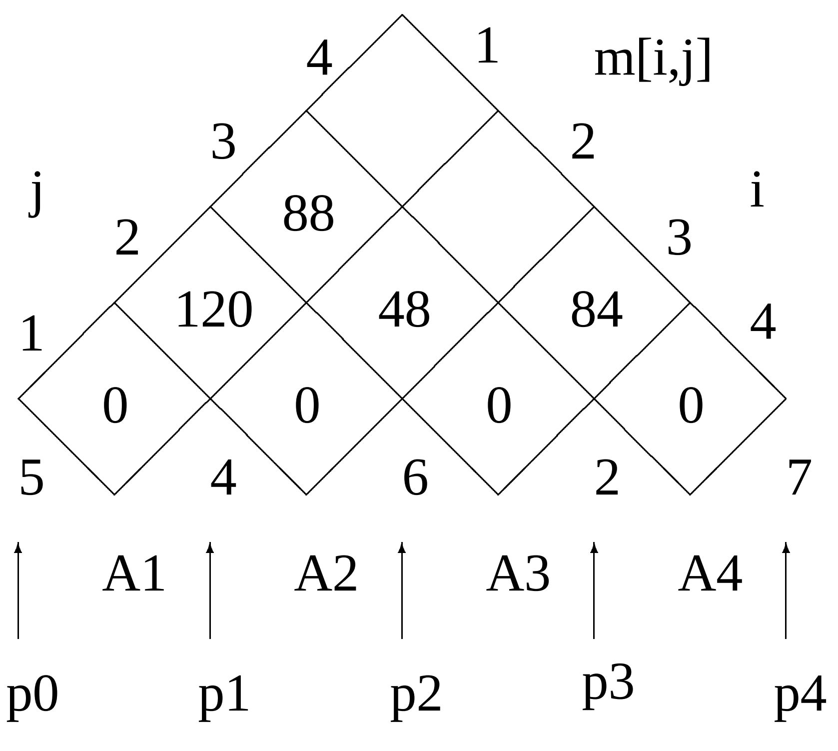
$$\begin{aligned} m[3,4] &= \min_{3\le k<4}\left(m[3,k]+m[k+1,4]+p_2p_kp_4\right) \\ &= m[3,3]+m[4,4]+p_2p_3p_4 = 84. \end{aligned}$$



Example – Continued

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

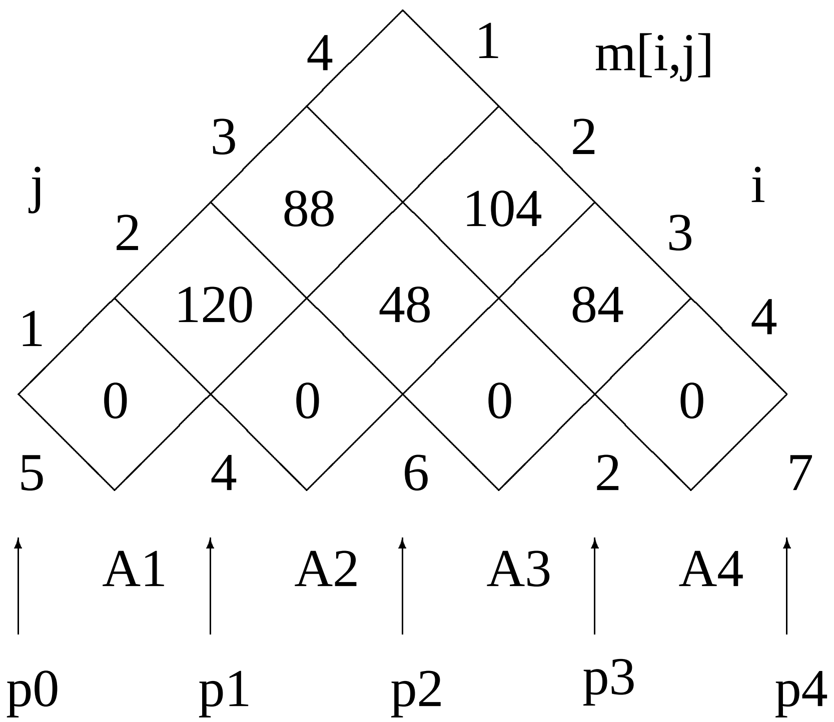
$$
\begin{aligned}
m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\
&= 88.
\end{aligned}
$$



Example – Continued

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

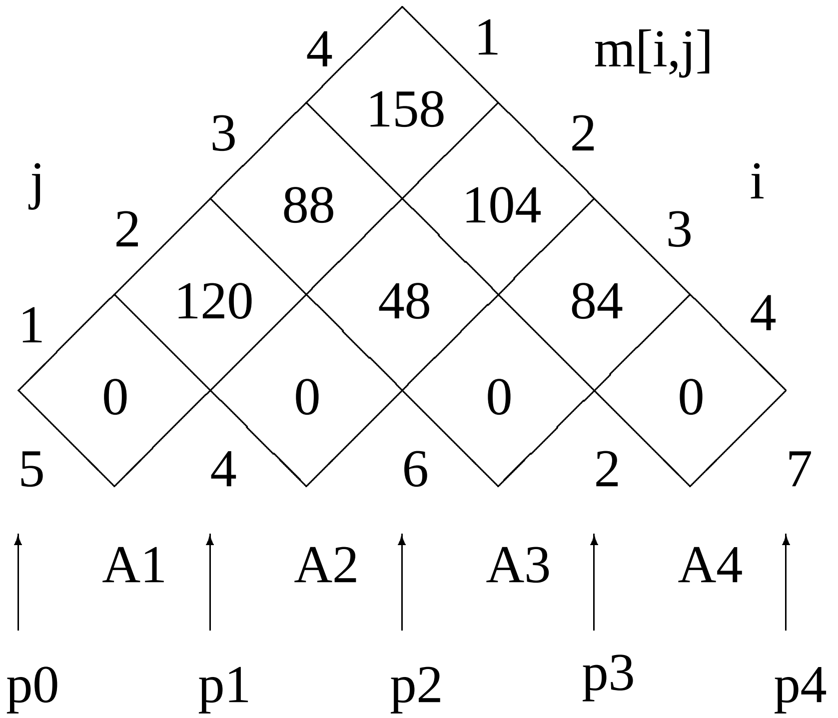
$$
\begin{aligned}
m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{array} \right\} \\
&= 104.
\end{aligned}
$$



Example – Continued

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.

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

Hence the final multiplication sequence is

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

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 $n - l + 1$)
    {
      $j = i + l - 1$;
      $m[i, j] = \infty$;
      for ($k = i$ to $j - 1$)
      {
        $q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$;
        if ($q < m[i, j]$)
        {
          $m[i, j] = q$;
          $s[i, j] = k$;
        }
      }
    }
  }
  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 $\Theta(n^2)$.

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$(A, s, 1, n)$.

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

Example of Constructing an Optimal Solution:

Compute $A_{1..6}$.

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

Mult$(A, s, 1, 6)$, $s[1, 6] = 3$, $(A_1A_2A_3)(A_4A_5A_6)$
Mult$(A, s, 1, 3)$, $s[1, 3] = 1$, $((A_1)(A_2A_3))(A_4A_5A_6)$
Mult$(A, s, 4, 6)$, $s[4, 6] = 5$, $((A_1)(A_2A_3))((A_4A_5)(A_6))$
Mult$(A, s, 2, 3)$, $s[2, 3] = 2$, $((A_1)((A_2)(A_3)))((A_4A_5)(A_6))$
Mult$(A, s, 4, 5)$, $s[4, 5] = 4$, $((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6))$

Hence the product is computed as follows

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