Outine

- Rod Cutting
- ***Matrix-Chain Multiplication***
- Elements of Dynamic Programming
- Longest Common Subsequence
- Optimal Binary Search Trees

Matrix-Chain Multiplication (1/4)

- We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product $A_1 A_2 \ldots A_n$.
- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

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

Matrix multiplication is associative, and so all parenthesizations yield the same product.

Matrix-Chain Multiplication (2/4)

- How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.
- Consider first the cost of multiplying 2 matrices.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr.

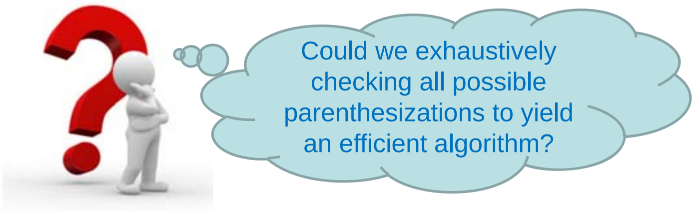
Matrix-Chain Multiplication (3/4)

- To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices.
  - Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively.
  - $((A_1 A_2) A_3)$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$
  - $(A_1 (A_2 A_3))$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$

Matrix-Chain Multiplication (4/4)

- We state the matrix-chain multiplication problem as follows: given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

➡ Our goal is only to determine an order for multiplying matrices that has the lowest cost.



Counting the Number of Parenthesizations

- Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$.

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

- Exercise 15.2-3 shows that the solution to the recurrence is $\Omega(2^n)$.

➡ The number of solutions is exponential in n, and the brute-force method of exhaustive search makes for a poor strategy.

Applying Dynamic Programming

We shall follow the four-step sequence:

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.
4. Construct an optimal solution from computed information.

Step 1: The Structure of an Optimal Parenthesization (1/3)

- For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \ldots A_j$.
  - To parenthesize the product $A_i A_{i+1} \ldots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.
  - The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$ + the cost of computing $A_{k+1..j}$ + the cost of multiplying them together.

Step 1: The Structure of an Optimal Parenthesization (2/3)

- Suppose that to optimally parenthesize $A_i A_{i+1} \ldots A_j$, we split the product between A_k and A_{k+1}.
- Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \ldots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \ldots A_k$. Why?
- A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \ldots A_j$ in the optimal parenthesization of $A_i A_{i+1} \ldots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \ldots A_j$.

Step 1: The Structure of an Optimal Parenthesization (3/3)

- We can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

Step 2: A Recursive Solution

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

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

- we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \ldots A_j$ in an optimal parenthesization.

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

Step 3: Computing the Optimal Costs (1/3)

- The recursive algorithm also takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product!

➡ A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree.

➡ Instead of computing the solution to the recurrence recursively, we compute the optimal cost by using a tabular, bottom-up approach.

Step 3: Computing the Optimal Costs (2/3)

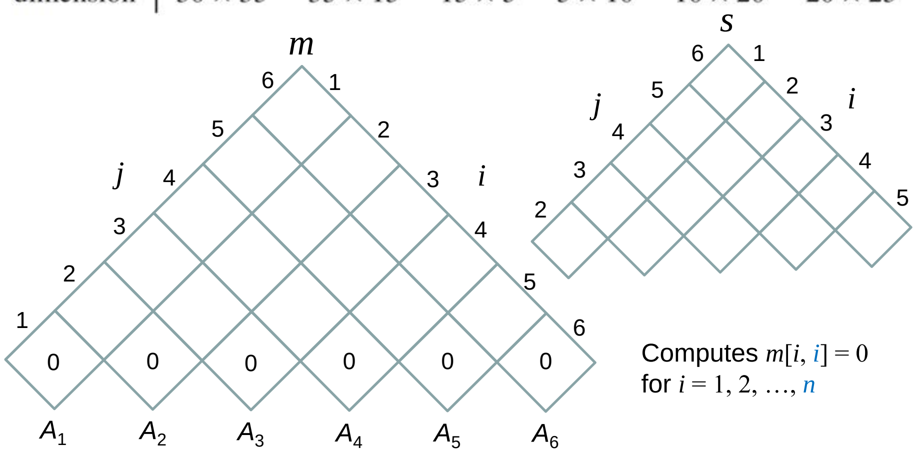
- We shall implement the tabular, bottom-up method in the procedure MATRIXCHAIN-ORDER.
  - It assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$.
  - Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$.
  - The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1..n-1, 2..n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.
  - We shall use the table s to construct an optimal solution.

Step 3: Computing the Optimal Costs (3/3)

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n              // l is the chain length
    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
```


Example (1/6)

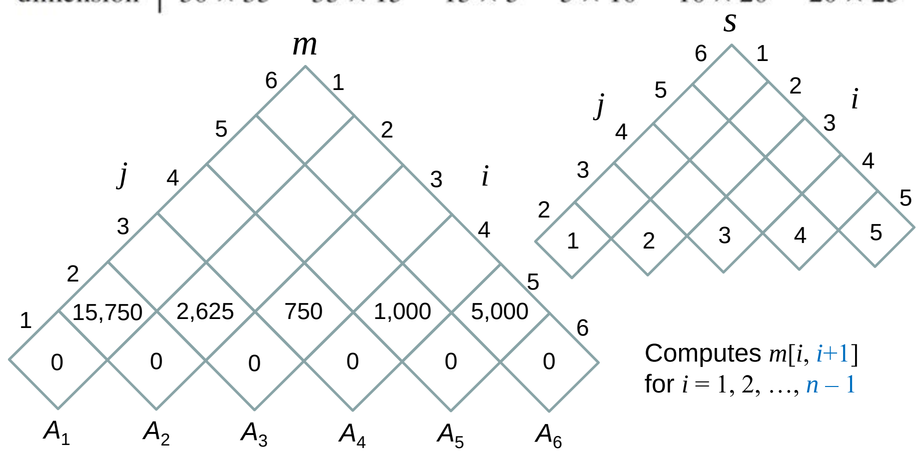
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



Computes $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

Example (2/6)

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



Computes $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$

$$\min_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} \quad \text{if } i < j$$