Dynamic Programming:
Matrix Chain Multiplication

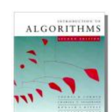
Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET

Matrix Chain Multiplication Problem

- Multiplying non-square matrices:
  - A is $p \times \boxed{q,}$ B is $\boxed{q} \times r$ must be equal
  - AB is $p \times r$ whose (i, j) entry is $\sum a_{ik}\, b_{kj}$

- Computing AB takes $p{\cdot}q{\cdot}r$ scalar multiplications and $p(q\text{–}1)r$ scalar additions (using basic algorithm).

- Suppose we have a sequence of matrices to multiply. What is the best order?



Matrix Chain Multiplication Problem

Given a sequence of matrices $A_1, A_2, \ldots, A_n$, then

Compute $C = A_1 . A_2 . \ldots, A_n$

- Different ways to compute C
  - $C = (A_1 A_2)((A_3 A_4)(A_5 A_6))$
  - $C = (A_1 (A_2 A_3)(A_4 A_5)) A_6$
- Matrix multiplication is associative
  - So output will be the same
- However, time cost can be very different
  - Example

Why Order Matters

- Suppose we have 4 matrices:
  - A, 30×01
  - B, 01×40
  - C, 40×10
  - D, 10×25

- $((AB)(CD))$: requires 41,200 multiplications

  [$(30\times1\times40) + (40\times10\times25) + (30\times40\times25) = 41{,}200$]

- $(A((BC)D))$: requires 1400 multiplications

  [$(1\times40\times10) + (1\times10\times25) + (30\times1\times25) = 1{,}400$]

Matrix Chain Multiplication Problem

Given a sequence of matrices $A_1, A_2, \ldots, A_n$, where A_i is $p_{i\text{-}1} \times p_i$:

1) What is minimum number of scalar multiplications required to compute $A_1 \cdot A_2 \cdot \ldots \cdot A_n$?
2) What order of matrix multiplications achieves this minimum?

- Fully parenthesize the product in a way that minimizes the number of scalar multiplications

(( ) ( )) (( ) (( ) (( ) ( ))))

No. of parenthesizations: ???

A Possible Solution

- Try all possibilities and choose the best one.
- Drawback is there are too many of them (exponential in the number of matrices to be multiplied)
- The number of parenthesizations is

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

- The solution to the recurrence is $\Omega(2^n)$

No. of parenthesizations: Exponential

- Need to be more clever - try dynamic programming !

Step 1: Optimal Substructure Property

- A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
- Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.
- We must also take care to ensure that the total number of distinct subproblems is a polynomial in the input size.

Step 1: Optimal Substructure Property

- Define $A_{i..j}$, $i \le j$, to be the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.
- If the problem is nontrivial, i.e, $i < j$, then to parenthesize $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$, split the product between A_k and A_{k+1} for some k, where $i \le k < j$.

- The cost of parenthesizing this way is
  - The cost of computing the matrix $A_{i..k}$ +
  - The cost of computing the matrix $A_{k+1..j}$ +
  - The cost of multiplying them together

- The optimal substructure of this problem is:

  An optimal parenthesization of $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ contains within it optimal parenthesizations of $A_i \cdot A_{i+1} \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_j$

  Proof ?

Overlapping Subproblem Property

- Two subproblems of the same problem are independent if they do not share resources.
- Two subproblems are *overlapping* if they are really the same subproblem that occurs as a subproblem of different problems.
- A problem exhibits *overlapping subproblem* if the number of subproblems is “small” in the sense that a recursive algorithm solves the same subproblems over and over, rather than always generating new subproblems.
- 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.

Overlapping Subproblem Property

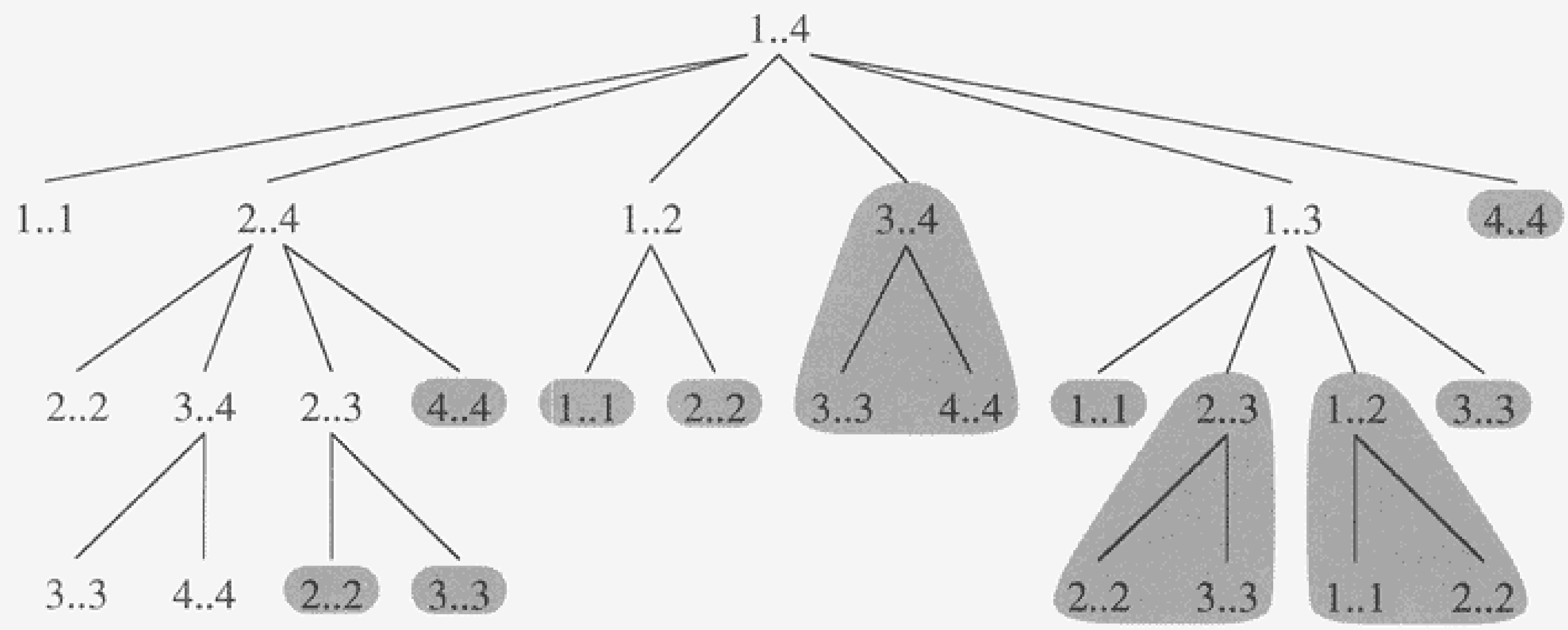


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(p, 1, 4).

Step 2: Develop a Recursive Solution

- Define $m[i, j]$ to be the minimum number of multiplications needed to compute $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.
  - Goal: Find $m[1, n]$
  - Basis: $m[i, i] = 0$
  - Recursion: How to define $m[i, j]$ recursively ?

- Consider all possible ways to split A_i through A_j into two pieces.
- Compare the costs of all these splits:
  - best case cost for computing the product of the two pieces
  - plus the cost of multiplying the two products
- Take the best one

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

Step 2: Develop a Recursive Solution

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i = j
    then return 0
m[i, j] ← ∞
for k ← i to j − 1
      do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                + p_{i−1} p_k p_j
          if q < m[i, j]
             then m[i, j] ← q
return m[i, j]
```

Step 2: Develop a Recursive Solution

- Let $T(n)$ be the time taken by Recursive-Matrix-Chain for n matrices.

$$T(1) \ge 1$$

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

For $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$. Thus, we have

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

$$\ge 2^{n-1}$$

Then $T(n) = \Omega(2^n)$

A Recursive Solution (Memoization)

- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.

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

A Recursive Solution (Memoization)

running time $O(n^3)$

Lookup-Chain(m, p, i, j)

    if $m[i, j] < \propto$

        return $m[i, j]$

    if $i = = j$

        $m[i, j] = 0$

    else for $k = i$ to $j - 1$

        q = Lookup-Chain(m, p, i, k) +

            Lookup-Chain(m, p, $k + 1$, j) + $p_{i-1} p_k p_j$

        if $q < m[i, j]$

            $m[i, j] = q$

    return $m[i, j]$

Step 3: Compute the Optimal Costs

Find Dependencies among Subproblems

m:

	1	2	3	4	5
1	0				
2	n/a				
3	n/a	n/a	0		
4	n/a	n/a	n/a		
5	n/a	n/a	n/a	n/a	0

GOAL !

computing the red square requires the blue ones: to the left and below.

Step 3: Compute the Optimal Costs

Find Dependencies among Subproblems

m:

	1	2	3	*j*	5
1	0				
i	n/a				
3	n/a	n/a	0		
4	n/a	n/a	n/a		
5	n/a	n/a	n/a	n/a	0

- Computing $m(i, j)$ uses
  - everything in same row to the left:

    $m(i, i), m(i, i+1), \ldots, m(i, j-1)$
  - and everything in same column below:

    $m(i+1, j), m(i+2, j), \ldots, m(j, j)$

Step 3: Compute the Optimal Costs

Identify Order for Solving Subproblems

- Solve the subproblems (i.e., fill in the table entries) this way:
  - go along the diagonal
  - start just above the main diagonal
  - end in the upper right corner (goal)

m:

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

Step 3: Compute the Optimal Costs

Identify Order for Solving Subproblems

- A1 (30×35)
- A2 (35×15)
- A3 (15×05)
- A4 (05×10)
- A5 (10×20)
- A6 (20×25)

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

Step 3: Compute the Optimal Costs

Identify Order for Solving Subproblems

- A1 (30×35)
- A2 (35×15)
- A3 (15×05)
- A4 (05×10)
- A5 (10×20)
- A6 (20×25)

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	**15125**
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

Step 3: Compute the Optimal Costs

Pseudocode

running time O(n^3)

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
      do m[i, i] ← 0
for l ← 2 to n                ▷ l is the chain length.
      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
return m and s
```

Step 4: Construct an Optimal Solution

- It's fine to know the cost of the cheapest order, but what is that cheapest order?
- Keep another array *s* and update it when computing the minimum cost in the inner loop
- After *m* and *s* have been filled in, then call a recursive algorithm on *s* to print out the actual order

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
    then print "A"_i
    else print "("
         PRINT-OPTIMAL-PARENS(s, i, s[i, j])
         PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
         print ")"
```