CSCI203

Algorithms and Data Structures

Dynamic Programming (II)

Lecturer: Dr. Fenghui Ren

Room 3.203

Email: fren@uow.edu.au

Dynamic Programming

- Previously we saw several “definitions” of Dynamic Programming:
  - Clever Brute Force;
  - Recursion + Memoization;
  - Tabular and bottom-up implementation
- Remember: dynamic programming is not an algorithm; it is a technique for constructing algorithms.

- This lecture we will examine some more problems that can be solved using dynamic programming techniques.

Dynamic Programming (DP)

- Typically applied to optimization problems

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Matrix Multiplication…

- An Example

- Let A and B be the following Matrices:
- $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$, $B = \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$
- Then C is calculated as follows:
- $c(1,1) = 1 \times -1 + 2 \times 2 = 3$
- $c(1,2) = 1 \times 2 + 2 \times -1 = 0$
- $c(2,1) = 3 \times -1 + 4 \times 2 = 5$
- $c(2,2) = 3 \times 2 + 4 \times -1 = 2$
- $c(3,1) = 5 \times -1 + 6 \times 2 = 7$
- $c(3,2) = 5 \times 2 + 6 \times -1 = 4$
- So:
- $C = \begin{bmatrix} 3 & 0 \\ 5 & 2 \\ 7 & 4 \end{bmatrix}$

Matrix Multiplication…

$$c_{11}=a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2 x 4 4 x 3 2 x 3

$$c_{22}=a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

$$\boldsymbol{A \times B = C}$$

$$\underset{(m \times n)}{A} \cdot \underset{(n \times k)}{B} = \underset{(m \times k)}{C}$$

product is defined

Matrix Multiplication…

- If A has m rows and n columns and B has n rows and k columns then C will have m rows and k columns.
- Matrix multiplication is not commutative: $AB \neq BA$ (BA may not even exist).

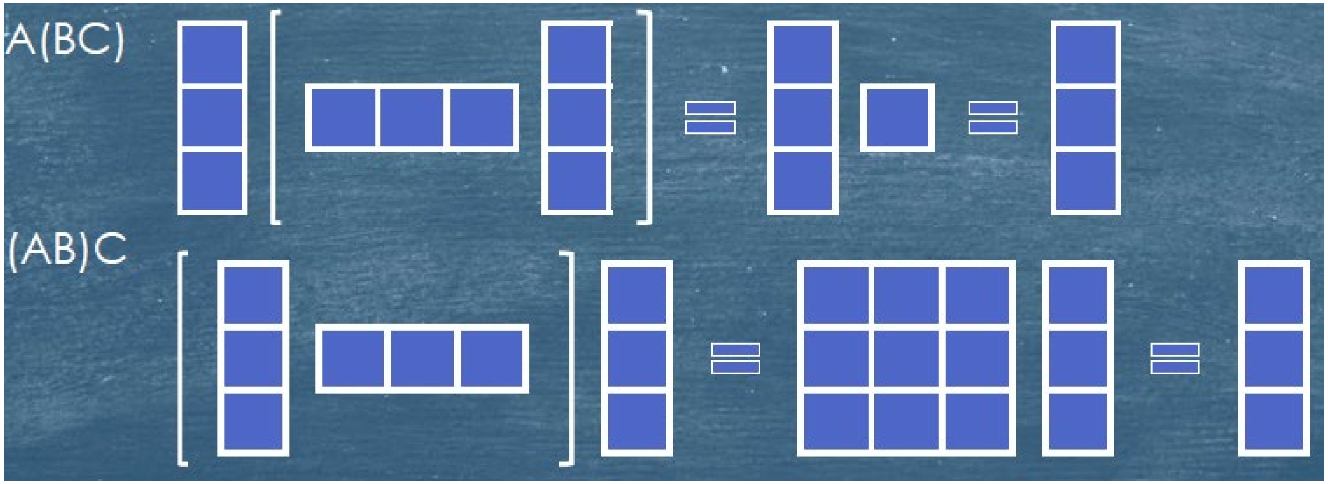
- The cost of a matrix multiplication depends on the sizes of A and B
- $A_{m \times n} \times B_{n \times k}$ will take $m \times n \times k$ multiplications and $m \times (n-1) \times k$ additions.
- The order in which multiple matrices are multiplied will effect the total cost.

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

Matrix-chain Multiplication

- Matrix multiplication is associative A(BC)=(AB)C.

- Let A, B and C be three matrices with sizes 3×1, 1×3 and 3×1 respectively:
- ABC can be computed as A(BC) or (AB)C.



6 multiplications

18 multiplications

Matrix-chain multiplication

- Multiplication of four matrices
  - $< A_1, A_2, A_3, A_4 >$
- We can define five distinct ways to perform the calculation using parentheses:

$$\left(A_1\left(A_2(A_3A_4)\right)\right), \left(A_1\left((A_2A_3)A_4\right)\right), \left((A_1A_2)(A_3A_4)\right), \left(\left(A_1(A_2A_3)\right)A_4\right), \left(\left((A_1A_2)A_3\right)A_4\right)$$

- How we parenthesize a chain of matrices can have a dramatic impact on the cost of computing the product.

Generalization

- If we need to evaluate the product of n matrices:
  - $A_1 \times A_2 \times A_3 \times \cdots \times A_n$ or $A_1 A_2 A_3 \cdots A_n$

- We can perform this in $O(4^n)$ different ways, each has a potentially different cost.
- What is the minimum cost for the overall computation.
  - What is the optimum sequence of matrix multiplications to perform?
- Once again, we can solve this with dynamic programming.

DP: Parenthesization

- We can restate the problem as follows:
- Given a sequence of n matrices; find the optimal locations for $n-1$ pairs of balanced parentheses, such that each pair contains exactly two matrices or parenthesized sets of matrices.
- E.g. given matrices $ABCD$, possible parenthesizations are:
  - $A(B(CD)), A((BC)D), (AB)(CD), (A(BC))D, ((AB)C)D$
- Let us approach this problem in the same way as we have used with the other problems.

Notations

- Given a chain $< A_1, A_2, \cdots, A_n >$ of matrices, where for $i = 1,2, \cdots, n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications

- $A_{i \ldots j}$ denotes $A_i A_{i+1} \cdots A_j$, $i \leq j$

- Note: *we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.*

Step 1. Structure an optimal parenthesization

- Suppose to optimally parenthesize $A_i \cdots A_j$
- Let's say an optimal split is between A_k and A_{k+1}
- The problem becomes two sub-problems
  - $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$
- We must ensure that we search for the correct place to split the product
  - We have considered all possible places so that we are sure of having examined the optimal one

Step 2. A recursive solution

- Let $m[i,j]$ be the minimum number of scalar multiplications needed for $A_{i \dots j}$
  - For the full problem $A_{1 \dots n}$, it would be $m[1,n]$

- Lets' examine $m[i,j]$

Step 2. A recursive solution…

- If $i = j$, $A_{i\cdots i} = A_i$, no scalar multiplications, $m[i,i] = 0$, for $i = 1,2,\cdots,n$
- If $i < j$ and optimal split at k, i.e. $A_{i\cdots k}$ and $A_{k+1\cdots j}$
  - Scalar multiplications for $A_{i\cdots k}$ is $m[i,k]$
  - Scalar multiplication for $A_{k+1\cdots j}$ is $m[k+1,j]$
  - The dimension of $A_{i\cdots k}$ is $p_{i-1} \times p_k$
  - The dimension of $A_{k+1\cdots j}$ is $p_k \times p_j$
  - Scalar multiplications for $A_{i\cdots k}A_{k+1\cdots j}$ is $p_{i-1}p_kp_j$

Step 2. A recursive solution…

- Thus, we have

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

- This recursive assume we know the value of k, which we do not.
- There are $j-i$ possible values for k, i.e.
  - $k = i, i+1, \cdots, j-1$
- We need to check all of them and find the best

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

Step 2. A recursive solution…

- We need to check all of them and find the best

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

- $m[i,j]$ values gives the costs of optimal solutions to a subproblems, does not provide the information to construct a optimal solution, i.e. where to split
- $s[i,j]$ to be a value of k which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization, i.e. $s[i,j]$ =k

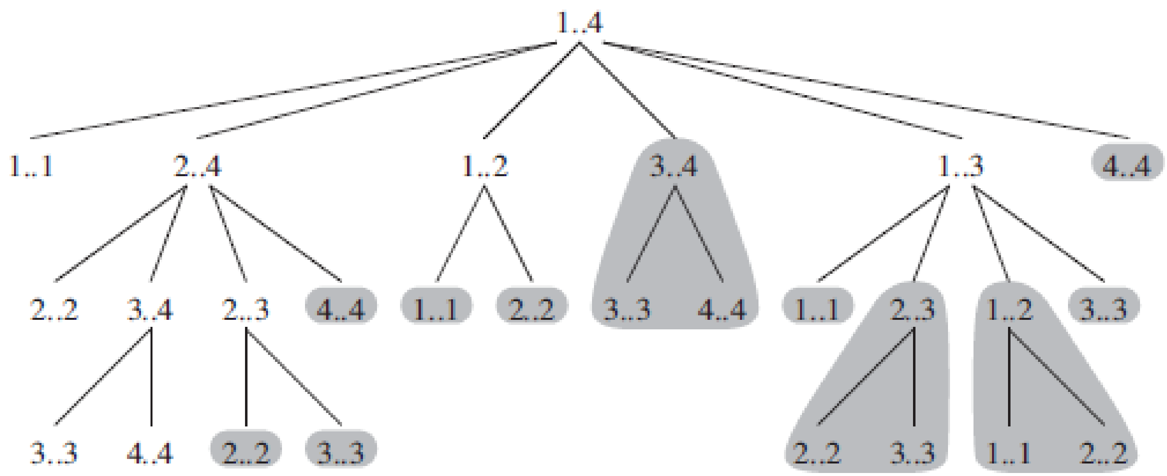
Step 3. Computing the optimal costs

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

- Given the above cost formulae, a recursive algorithm can be written to calculate the minimum cost $m[1,n]$ for $A_1 A_2 \cdots A_n$
- $p =< p_{i-1}, p_i, \cdots, p_n >$ dimensions of the matrices in the chain

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    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]
        m[i, j] = q
return m[i, j]
```

Step 3. Computing the optimal costs



RECURSIVE-MATRIX-CHAIN (p, 1,4)

- The recursive takes exponential time.
- It is no better than the brute-force method.

Step 3. Computing the optimal costs…

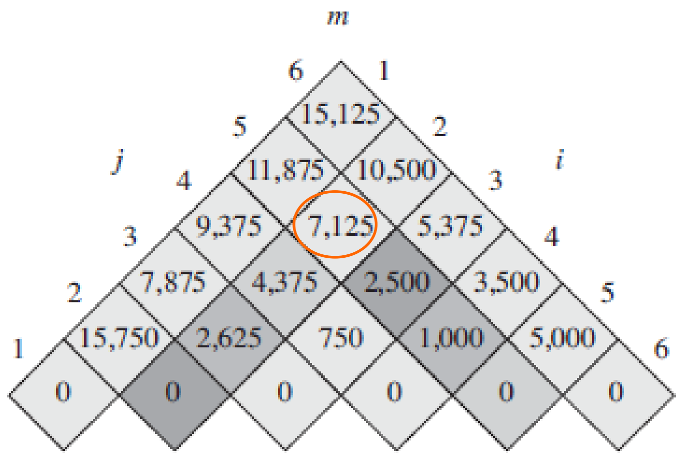
$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases}$$

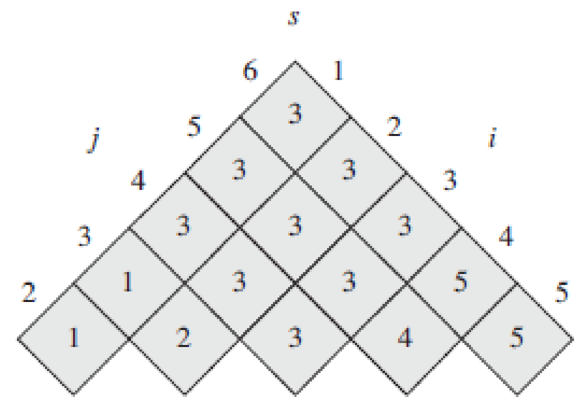
- Compute the optimal cost by using a tabular, bottom-up approach
  - Dimension of A_i is $p_{i-1} \times p_i$,
  - $p =< p_0, p_1, \cdots, p_n >$ and $p.length = n + 1$

```
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
```

Complexity $O(n^3)$

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





$$
m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 \,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}
$$
$$
= 7125 \,.
$$

- The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$.

Step 4. Constructing an optimal solution

- The table $s[1 \dots n-1, 2 \dots n]$ gives us the information we need to do so.
- Each entry $s[i,j]$ records a value of k such that an optimal parenthesization of $A_{i \dots j}$ splits the product between A_k and A_{k+1}.
  - $A_{1 \cdots n} \rightarrow A_{1 \cdots s[1,n]} A_{s[1,n]+1, \dots n}$ split at $s[1,n]$
  - $A_{1 \dots s[1,n]} \rightarrow$ split at $s\big[1, s[1,n]\big]$
  - $A_{s[1,n]+1 \dots n} \rightarrow$ split at $s[s[1,n]+1, n]$
  - .....

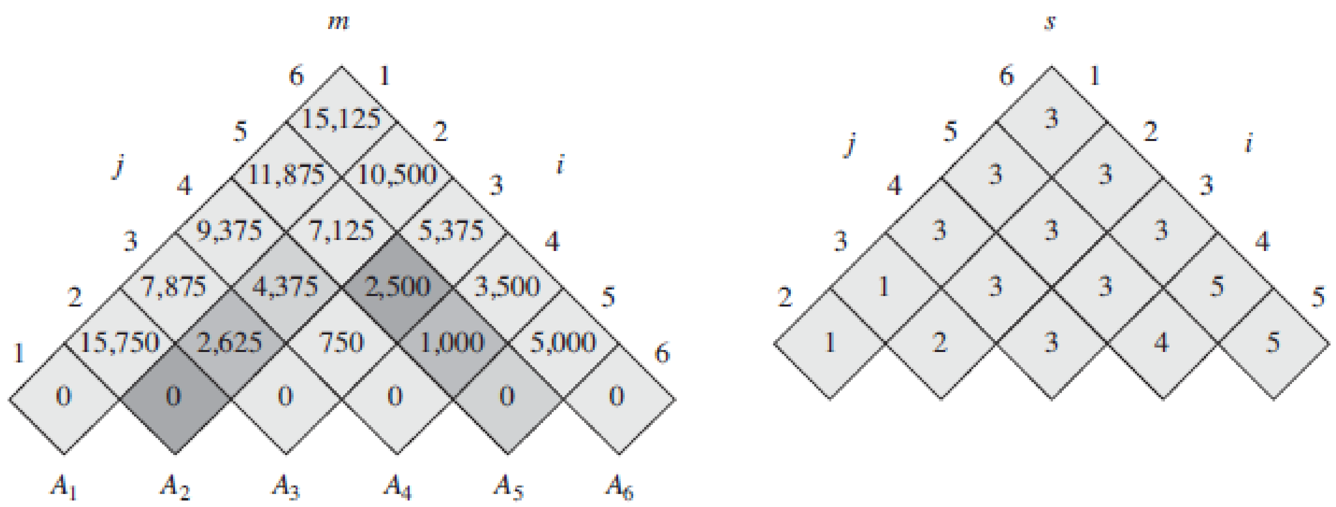
Step 4. Constructing an optimal solution

- An optimal parenthesization of $A_1 A_2 \cdots A_n$ is

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

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

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



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

Related References

- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 15.2