Combinatorial Optimization
CSE 301

Lecture 2

Dynamic Programming

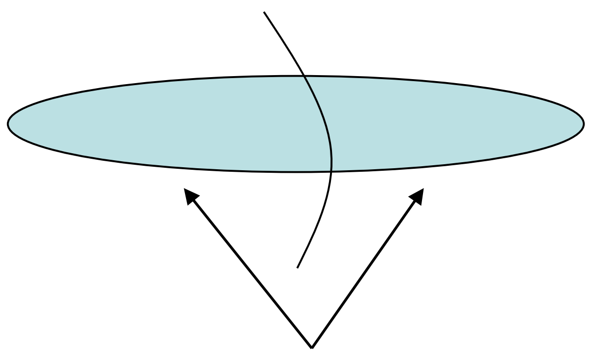
Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

DP - Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

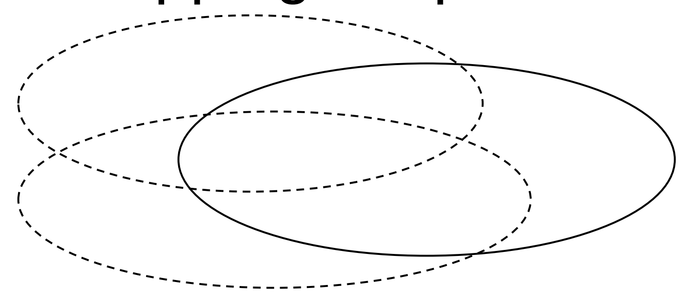
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Matrix-chain Multiplication

- Suppose we have a sequence or chain A_1, A_2, …, A_n of *n* matrices to be multiplied
  - That is, we want to compute the product $A_1A_2 \ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)

Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY($A_{p\times q}$, $B_{q\times r}$)

1. **for** $i \leftarrow 1$ **to** p
2. **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4. **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute C

Number of scalar multiplications = pqr

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-Chain Multiplication

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

$$
\begin{array}{ccccccccc}
A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\
p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n
\end{array}
$$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$p_{i-1}p_kp_j$

$$= A_{i\ldots k}\, A_{k+1\ldots j} \qquad \text{for } i \le k < j$$

m[i, k] m[k+1,j]

- Assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \cdots A_j$ at k ($i \le k < j$)

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

min # of multiplications to compute $A_{i\ldots k}$

min # of multiplications to compute $A_{k+1\ldots j}$

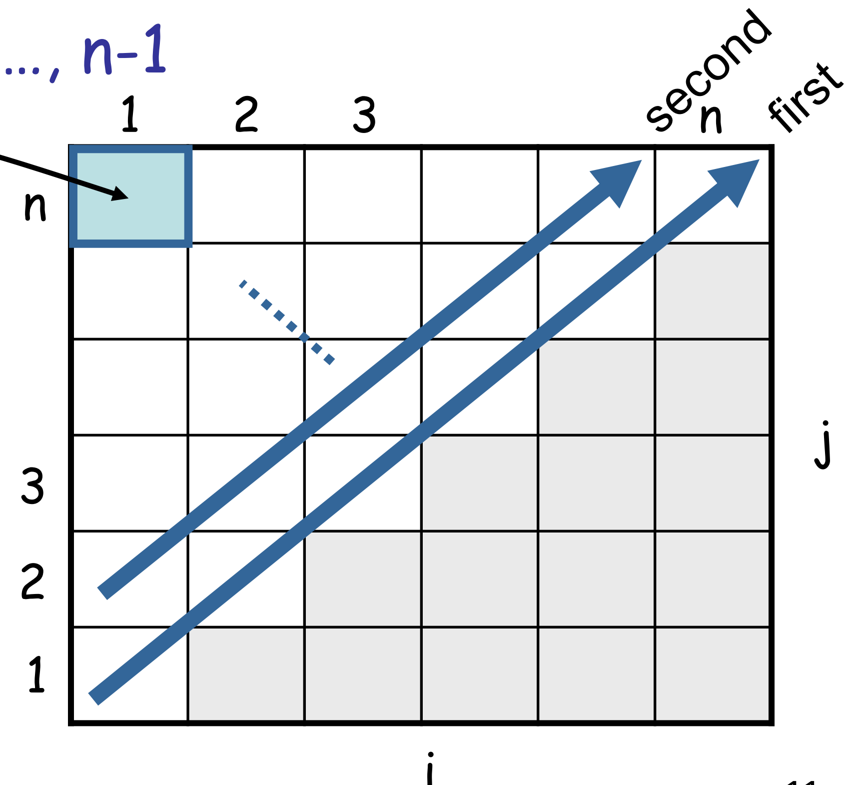
# of multiplications to compute $A_{i\ldots k}A_{k\ldots j}$

3. Computing the Optimal Costs

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

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

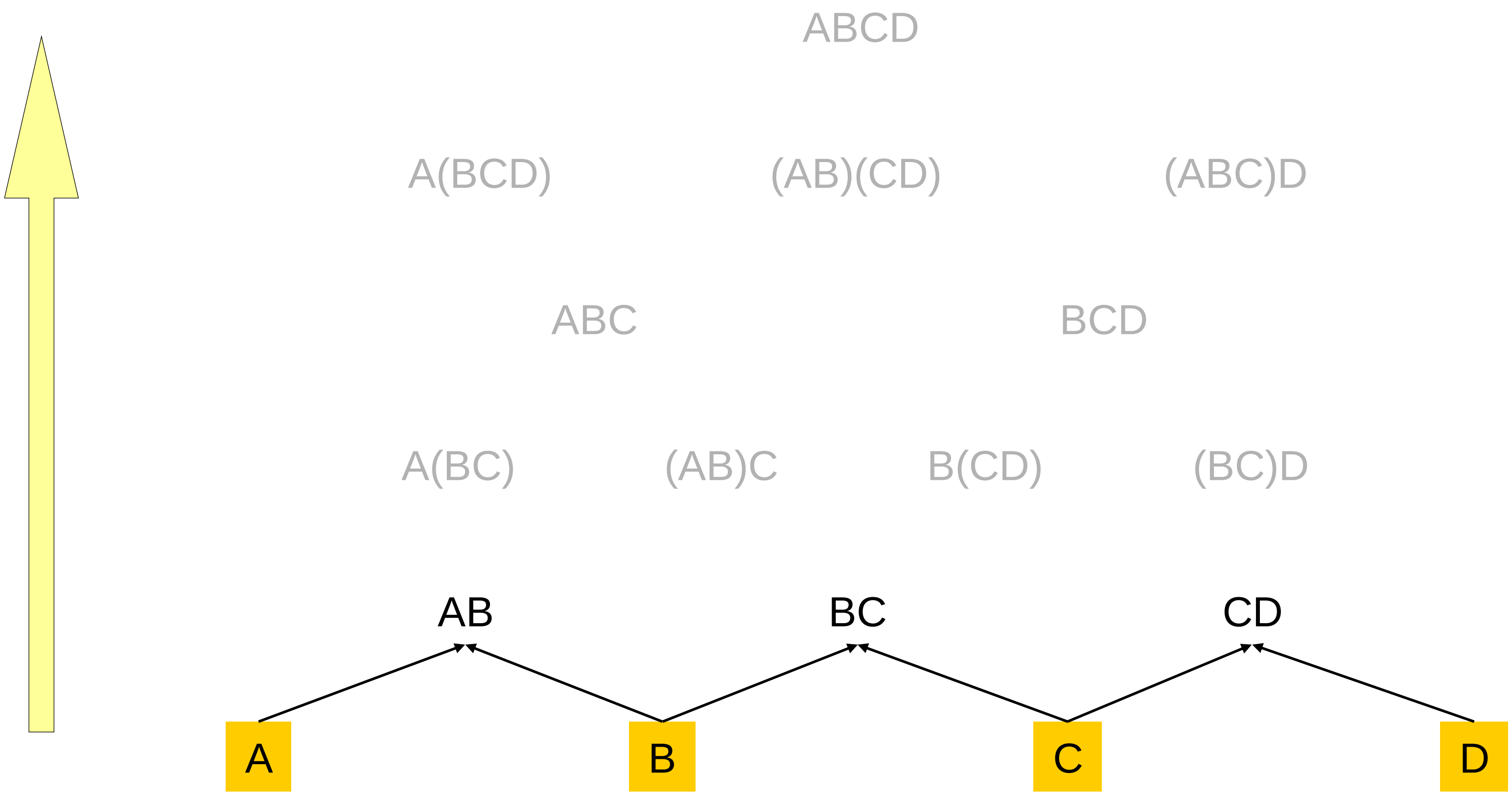
$m[1, n]$ gives the optimal solution to the problem



Compute rows from bottom to top and from left to right
In a similar matrix s we keep the optimal values of k

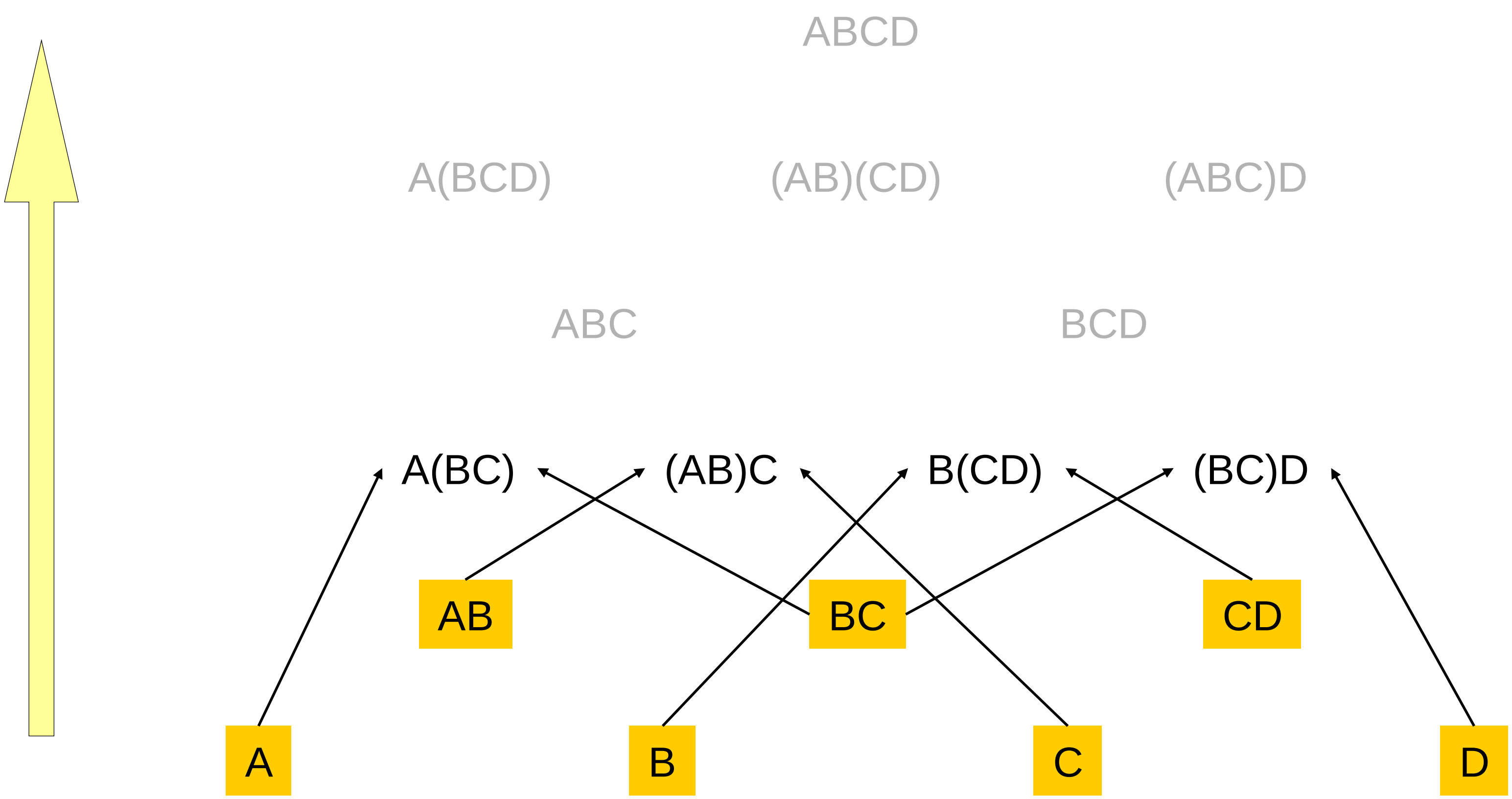
Multiply 4 Matrices: $A \times B \times C \times D$ (1)

- Compute the costs in the bottom-up manner
  - First we consider AB, BC, CD
  - No need to consider AC or BD



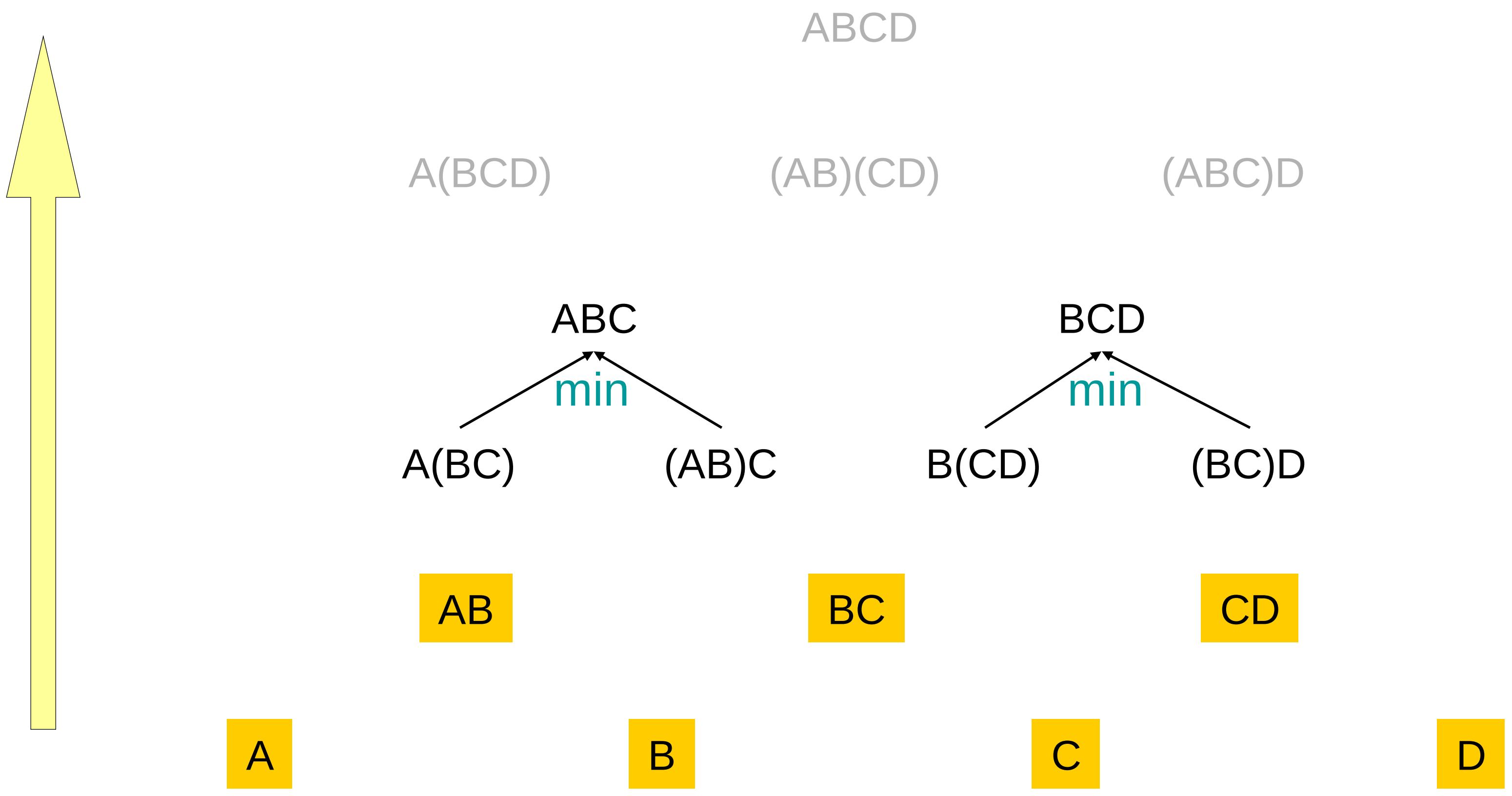
Multiply 4 Matrices: A×B×C×D (2)

- Compute the costs in the bottom-up manner
  - Then we consider A(BC), (AB)C, B(CD), (BC)D
  - No need to consider (AB)D, A(CD)



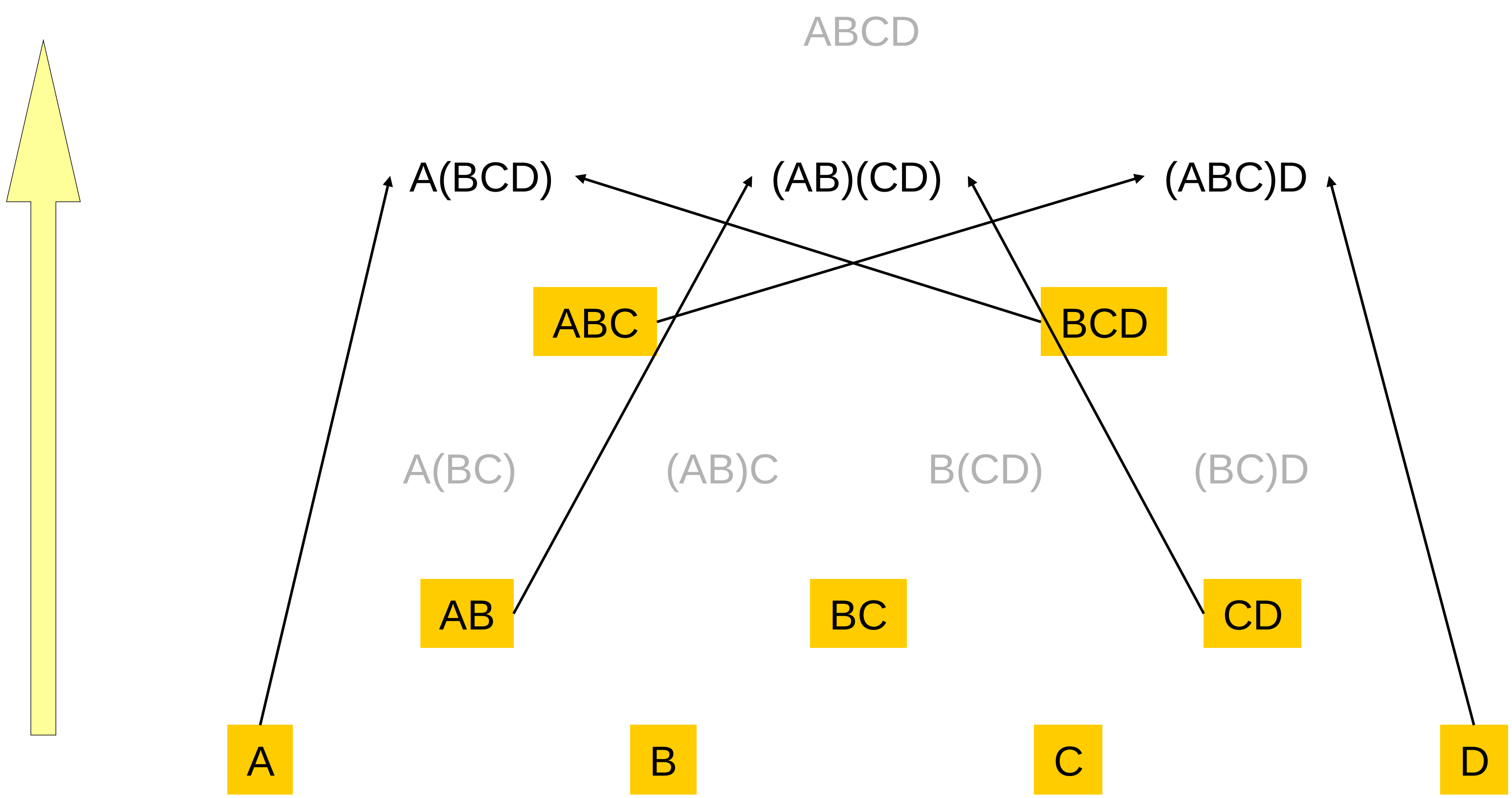
Multiply 4 Matrices: A×B×C×D (3)

- Compute the costs in the bottom-up manner
- Select minimum cost matrix calculations of ABC & BCD



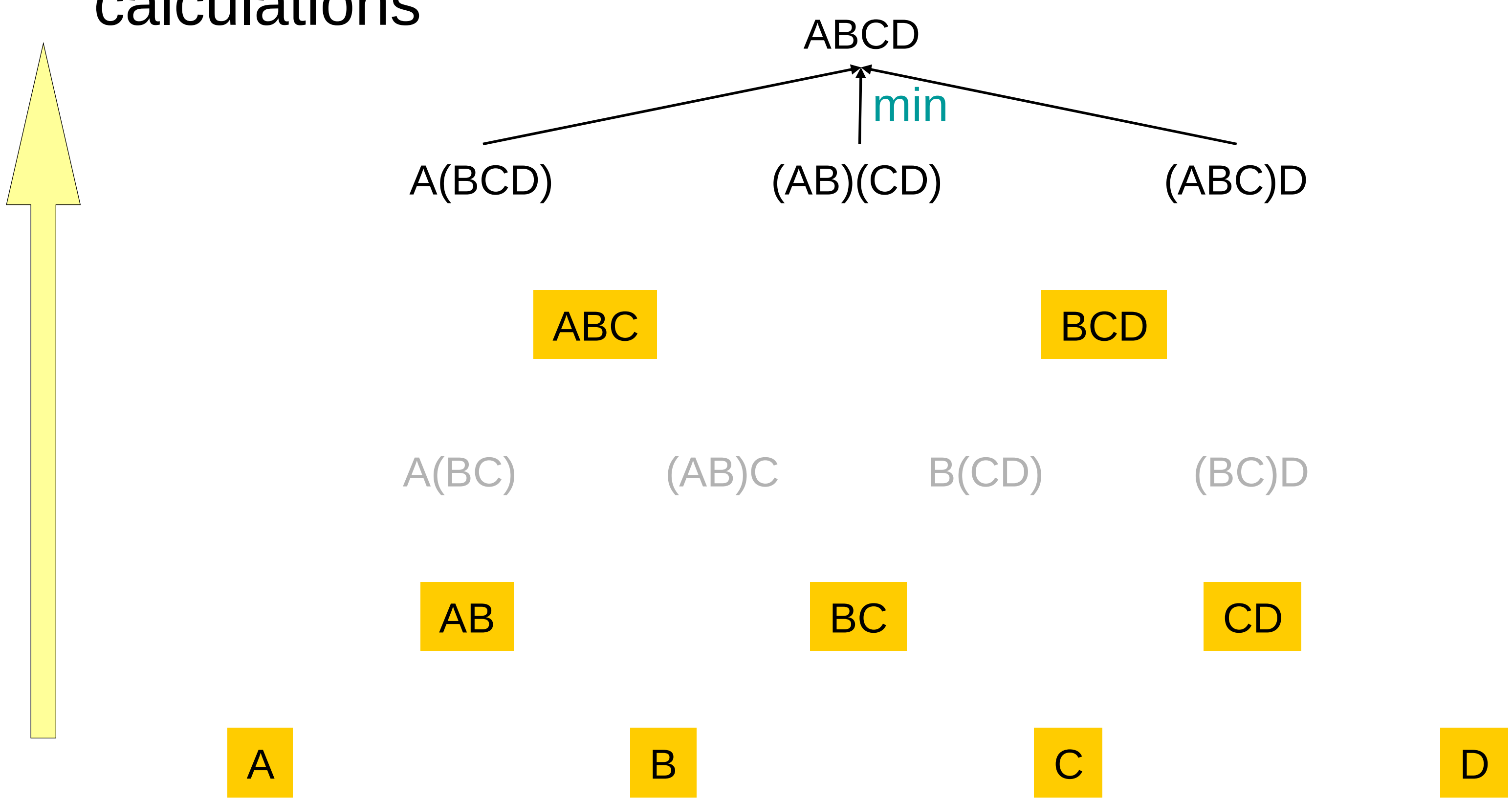
Multiply 4 Matrices: $A \times B \times C \times D$ (4)

- Compute the costs in the bottom-up manner
  - We now consider A(BCD), (AB)(CD), (ABC)D



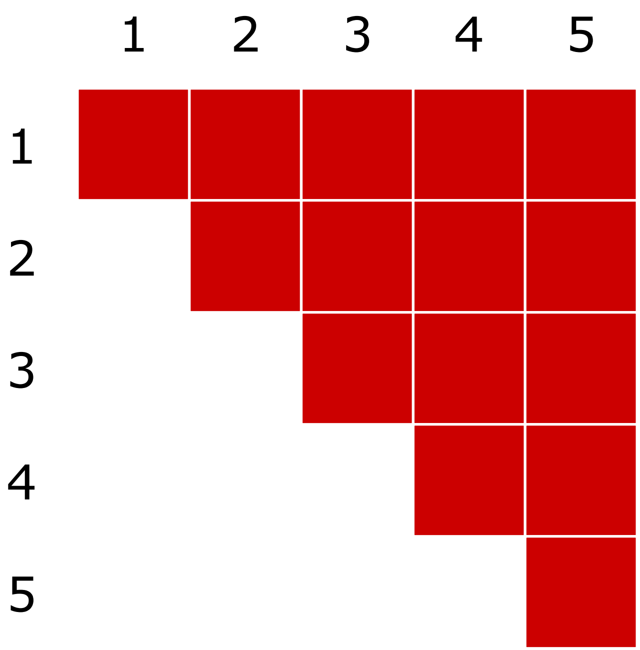
Multiply 4 Matrices: $A \times B \times C \times D$ (5)

- Compute the costs in the bottom-up manner
  - Finally choose the cheapest cost plan for matrix calculations

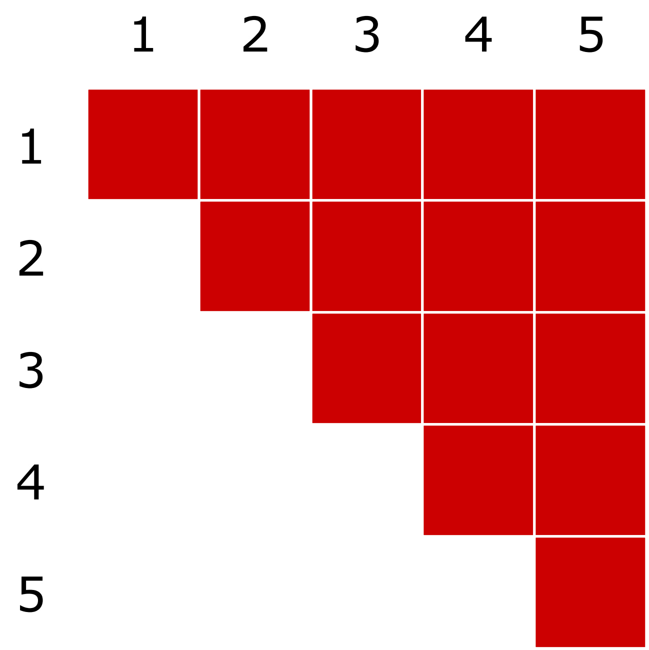


Example: Step 1

- $q = 5, \quad r = (10, 5, 1, 10, 2, 10)$
  - $[10 \times 5] \times [5 \times 1] \times [1 \times 10] \times [10 \times 2] \times [2 \times 10]$



$c(i,j), i \le j$



$kay(i,j), i \le j$

Example: Step 2

– s = 0, $[10\times 5]\times[5\times 1]\times[1\times 10]\times[10\times 2]\times[2\times 10]$
– c(i,i) = 0

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

c(i,j), $i \le j$

	1	2	3	4	5
1					
2					
3					
4					
5					

kay(i,j), $i \le j$

Example: Step 3

– $s = 0$, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$

– $c(i,i+1) = r_i r_{i+1} r_{i+2}$

– $kay(i,i+1) = i$

	1	2	3	4	5
1	0	50			
2		0	50		
3			0	20	
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1			
2			2		
3				3	
4					4
5					

kay(i,j), i ≤ j

Example: Step 4

- $s = 1$, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(i,i+2) = \min\{c(i,i) + c(i+1,i+2) + r_i r_{i+1} r_{i+3},$
  $c(i,i+1) + c(i+2,i+2) + r_i r_{i+2} r_{i+3}\}$

	1	2	3	4	5
1	0	50			
2		0	50		
3			0	20	
4				0	200
5					0

$c(i,j)$, $i \le j$

	1	2	3	4	5
1		1			
2			2		
3				3	
4					4
5					

$kay(i,j)$, $i \le j$

Example: Step 5

- $s = 1$, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(2,4) = \min\{c(2,2) + c(3,4) + r_2r_3r_5,\ c(2,3) + c(4,4) + r_2r_4r_5\}$
- $c(3,5) = \ldots$

	1	2	3	4	5
1	0	50	150		
2		0	50	30	
3			0	20	40
4				0	200
5					0

$c(i,j),\ i \le j$

	1	2	3	4	5
1		1	2		
2			2	2	
3				3	3
4					4
5					

$kay(i,j),\ i \le j$

Example: Step 6

- $s = 2$, $[10\times 5]\times[5\times 1]\times[1\times 10]\times[10\times 2]\times[2\times 10]$
- $c(i,i+3) = \min\{c(i,i) + c(i+1,i+3) + r_i r_{i+1} r_{i+4},$
  $c(i,i+1) + c(i+2,i+3) + r_i r_{i+2} r_{i+4},$
  $c(i,i+2) + c(i+3,i+3) + r_i r_{i+3} r_{i+4}\}$

	1	2	3	4	5
1	0	50	150	90	
2		0	50	30	90
3			0	20	40
4				0	200
5					0

$c(i,j)$, $i \le j$

	1	2	3	4	5
1		1	2	2	
2			2	2	2
3				3	3
4					4
5					

$kay(i,j)$, $i \le j$

Example: Step 7

– $s = 3$, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$

– $c(i,i+4) = \min\{c(i,i) + c(i+1,i+4) + r_i r_{i+1} r_{i+5},$
$c(i,i+1) + c(i+2,i+4) + r_i r_{i+2} r_{i+5},\ c(i,i+2) + c(i+3,i+4) +$
$r_i r_{i+3} r_{i+5},\ c(i,i+3) + c(i+4,i+4) + r_i r_{i+4} r_{i+5}\}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

$c(i,j),\ i \le j$

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	3
4					4
5					

$kay(i,j),\ i \le j$

Example: Step 8

- Optimal multiplication sequence
  - kay(1,5) = 2
    - $M_{15} = M_{12} \times M_{35}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	3
4					4
5					

kay(i,j), i ≤ j

Example: Step 9

– $M_{15} = M_{12} \times M_{35}$

– kay(1,2) = 1 ▶ $M_{12} = M_{11} \times M_{22}$

$\rightarrow M_{15} = (M_{11} \times M_{22}) \times M_{35}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

Example: Step 10

- $M_{15} = (M_{11} \times M_{22}) \times M_{35}$
- Kay(3,5) = 4 ▶ $M_{35} = M_{34} \times M_{55}$

$\rightarrow M_{15} = (M_{11} \times M_{22}) \times (M_{34} \times M_{55})$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

Example: Step 11

– $M_{15} = (M_{11} \times M_{22}) \times (M_{34} \times M_{55})$

– Kay(3,4) = 3 ▶ $M_{34} = M_{33} \times M_{44}$

$\rightarrow M_{15} = (M_{11} \times M_{22}) \times ((M_{33} \times M_{44}) \times M_{55})$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

Memoized Matrix-Chain

Alg.: MEMOIZED-MATRIX-CHAIN(p)

1. $n \leftarrow$ length[p] - 1
2. **for** $i \leftarrow 1$ **to** n
3. **do for** $j \leftarrow i$ **to** n
4. **do** $m[i, j] \leftarrow \infty$
5. **return** LOOKUP-CHAIN(p, 1, n)

Initialize the m table with large values that indicate whether the values of m[i, j] have been computed

Top-down approach

Memoized Matrix-Chain

Alg.: LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

```
if m[i, j] < ∞
    then return m[i, j]
if i = j
    then m[i, j] ← 0
    else for k ← i to j - 1
            do q ← LOOKUP-CHAIN(p, i, k) +
                   LOOKUP-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]
```

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

Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - Some subproblems do not need to be solved
  - Easier to implement and to think