# Matrix Multiplication

- Matrix Multiplication
  - **A**: $n * l$ matrix, **B**: $l * m$ ➔ **C**: $n * m$ matrix

$$\mathbf{C} = \mathbf{AB} = \left[c_{ij}\right]$$

$$c_{ij} = \sum_{k=1}^{l} a_{ik} b_{kj}$$

  - example

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

# Matrix Multiplication: A x B = C

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

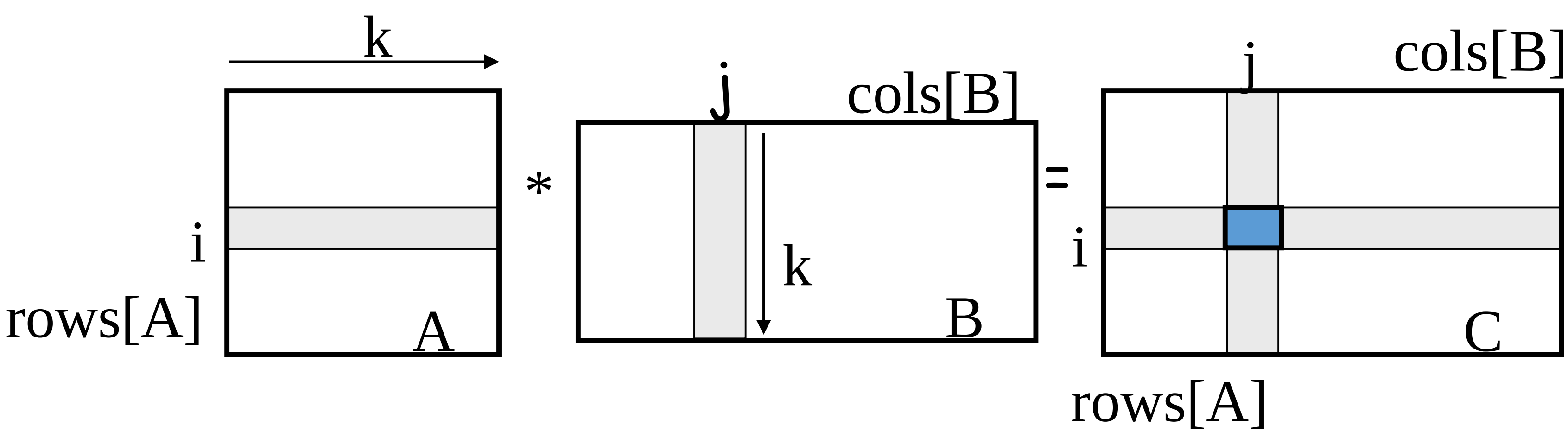
$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \end{bmatrix}$$

Matrix multiplication is **not commutative**!

$$\mathbf{AB} \neq \mathbf{BA}$$

# MATRIX-MULTIPLY(A, B)

```
if columns[A] ≠ rows[B]
  then error "incompatible dimensions"
  else for i ← 1 to rows[A]
           do for j ← 1 to columns[B]
                  do C[i, j] = 0
                     for k ← 1 to columns[A]
                         do C[i, j] ← C[i, j] + A[i, k] B[k, j]
```



# Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied

  Ex) $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

  $= (A_1 \cdot (A_2 \cdot A_3))$

- Which one of these orderings should we choose?
  - <u>The order in which we multiply the matrices has a significant impact on the cost of evaluating the product</u>

# Example

$$A_1 \cdot A_2 \cdot A_3$$

- $A_1$: 10 x 100 $A_2$: 100 x 5 $A_3$: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5{,}000$ $(10 \times 5)$

$$((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2{,}500$$

Total: <u>7,500 scalar multiplications</u>

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25{,}000$ $(100 \times 50)$

$$(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50{,}000$$

Total: <u>75,000 scalar multiplications</u>

➔ <u>significant impact on the cost of evaluating the product</u>

# Matrix-Chain Multiplication: Problem Statement

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

$$A_1 \cdot A_2 \quad \cdots \quad \mathbf{A_i} \cdot \mathbf{A_{i+1}} \quad \cdots \quad A_n$$

$$p_0 \text{ x } p_1 \quad p_1 \text{ x } p_2 \quad p_{i-1} \text{ x } p_i \quad p_i \text{ x } p_{i+1} \quad p_{n-1} \text{ x } p_n$$

# What is the number of possible parenthesizations?

- Brute force approach
  - Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

# 1. The Structure of an Optimal Parenthesization

- Notation:

$$A_{i\ldots j} = A_i\ A_{i+1} \cdots A_j, \quad i \le j$$

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between $A_k$ and $A_{k+1}$, where $i \le k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\ A_{i+1} \cdots A_j \\ &= A_i\ A_{i+1} \cdots A_k\ A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\ A_{k+1\ldots j} \end{aligned}$$

# Optimal Substructure

$$\mathbf{A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}}$$

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

# 2. A Recursive Solution

- Subproblem:

  determine the minimum cost of parenthesizing

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

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$
  - full problem ($A_{1..n}$) : $m[1, n]$
  - $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] = 0$, for $i = 1, 2, \ldots, n$

# 2. A Recursive Solution

- Consider the subproblem of parenthesizing

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

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

- Assume that the optimal parenthesization splits the product

$A_i A_{i+1} \cdots A_j$ at k ($i \le k < j$)

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

# 2. A Recursive Solution (cont.)

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

- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i\, A_{i+1} \cdots A_j$ becomes:

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

# 3. Computing the Optimal Costs

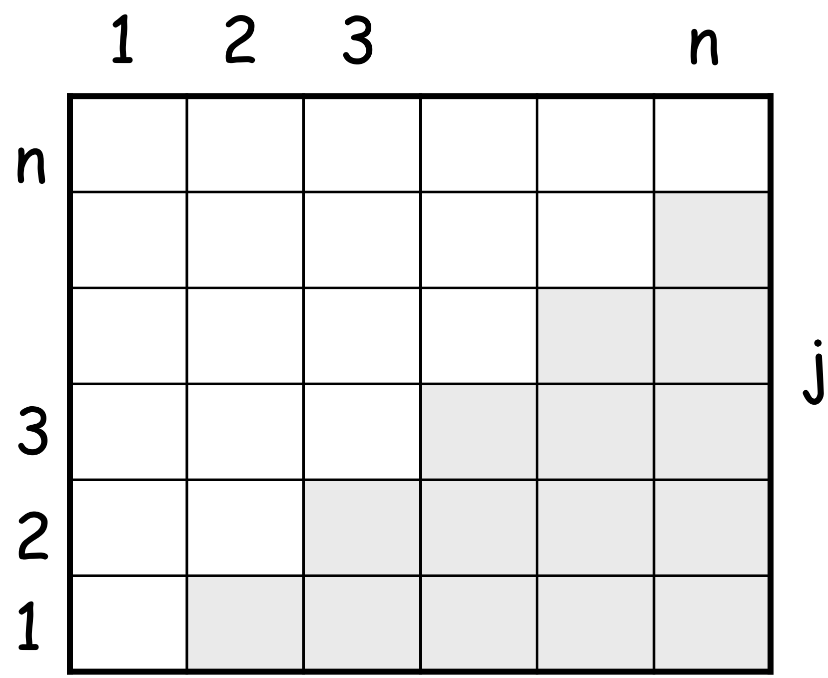
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Computing the optimal solution recursively takes exponential time!

- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  - Parenthesize $A_{i\ldots j}$ for $1 \le i \le j \le n$
  - One problem for each choice of i and j



# 3. Computing the Optimal Costs

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

- How do we fill in the tables m[1..n, 1..n]?
  - Determine which entries of the table are used in computing m[i, j]

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

  - Subproblems' size is one less than the original size
  - **Idea:** fill in m such that it corresponds to solving problems of increasing length

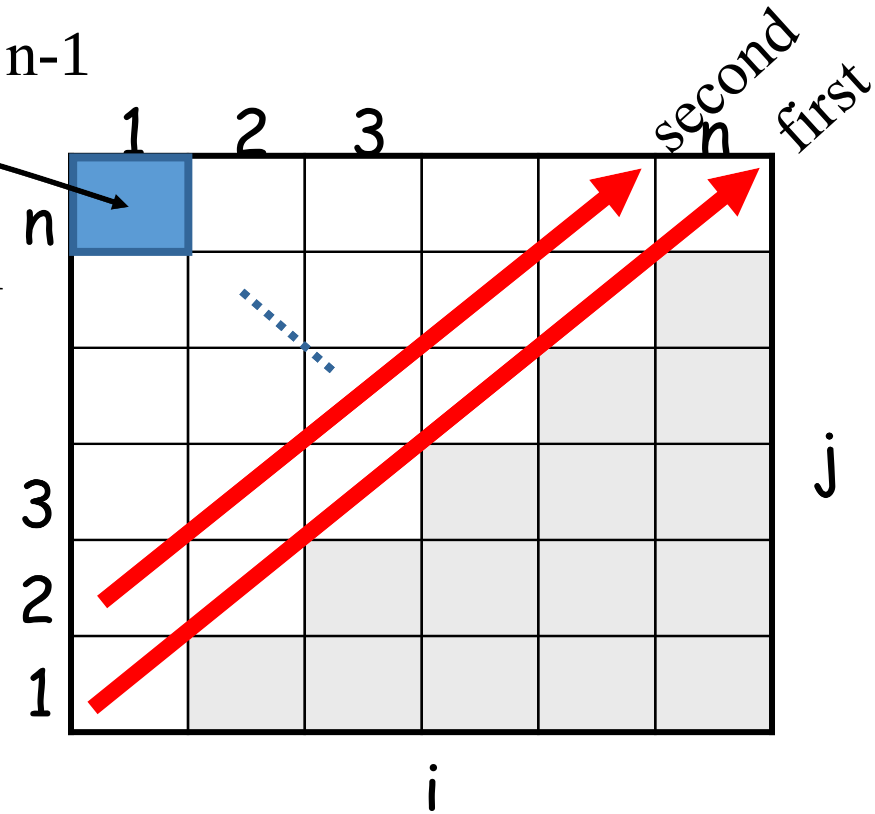
# 3. Computing the Optimal Costs (cont.)

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

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

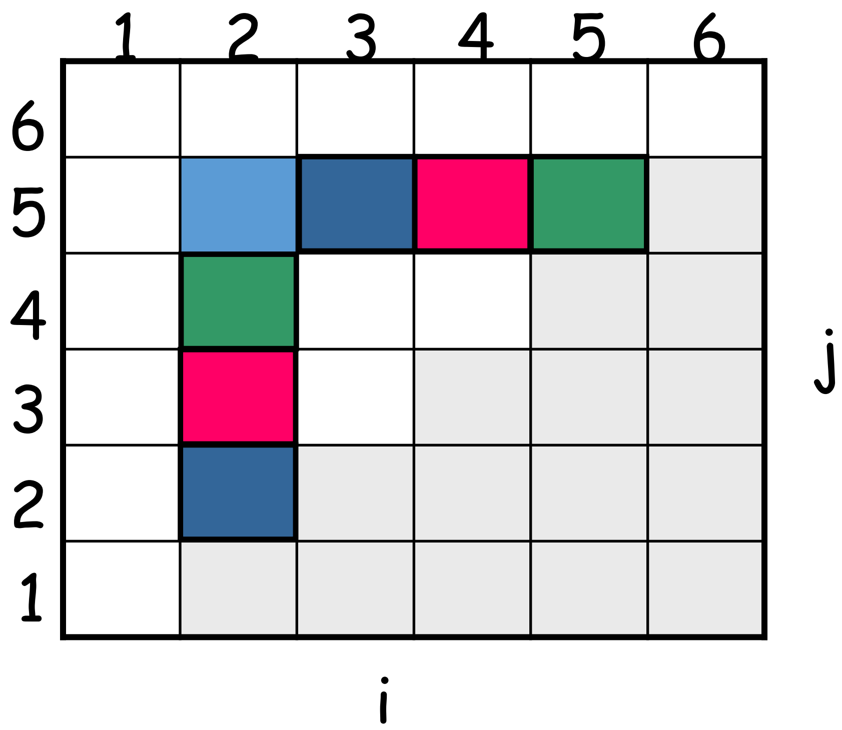
m[1, n] gives the optimal solution to the problem

Compute rows from bottom to top and from left to right



# Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$




- Values m[i, j] depend only on values that have been previously computed

# Example $\min\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$

Compute $A_1 \cdot A_2 \cdot A_3$

- $A_1$: 10 x 100 ($p_0$ x $p_1$)
- $A_2$: 100 x 5 ($p_1$ x $p_2$)
- $A_3$: 5 x 50 ($p_2$ x $p_3$)

| | 1 | 2 | 3 |
|---|---|---|---|
| 3 | 2<br>7500 | 2<br>25000 | 0 |
| 2 | 1<br>5000 | 0 | |
| 1 | 0 | | |

$m[i, i] = 0$ for $i = 1, 2, 3$

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2$ $(A_1A_2)$

$= 0 + 0 + 10 * 100 * 5 = 5{,}000$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3$ $(A_2A_3)$

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = \underline{7{,}500} & ((A_1A_2)A_3) \end{cases}$$

# Matrix-Chain-Order

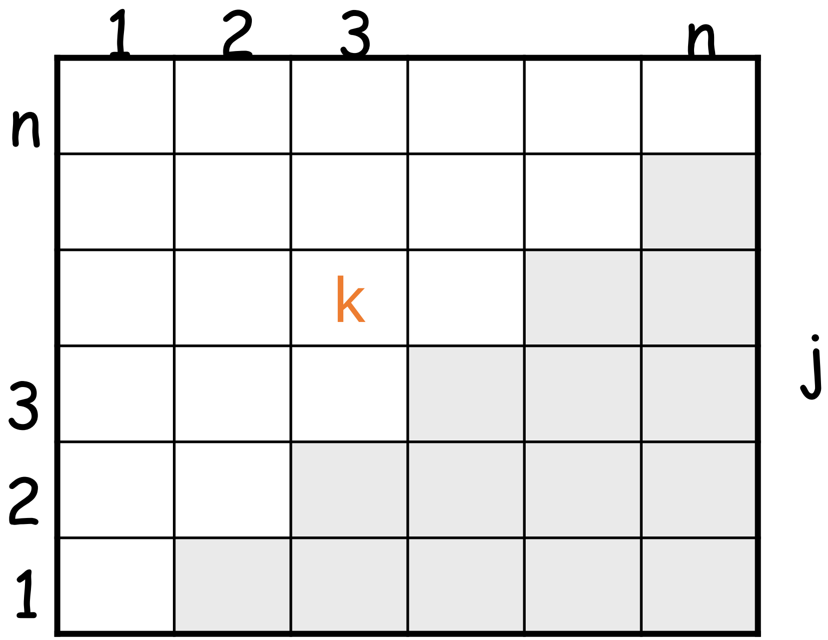
MATRIX-CHAIN-ORDER(p)

```
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
```

$O(N^3)$

# 4. Construct the Optimal Solution

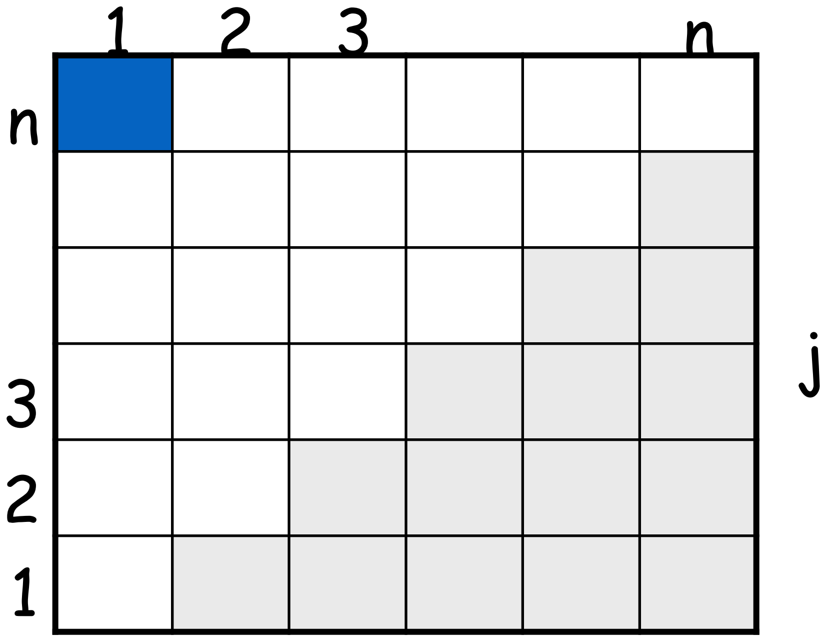
- In a similar matrix s we keep the optimal values of k
- s[i, j] = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between $A_k$ and $A_{k+1}$



# 4. Construct the Optimal Solution

- s[1, n] is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = s[1, n]

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



# 4. Construct the Optimal Solution

- s[i, j] = value of k such that the optimal parenthesization of $A_i\, A_{i+1} \cdots A_j$ splits the product between $A_k$ and $A_{k+1}$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

j

i

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\, A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\, A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\, A_{6..6}$

# 4. Construct the Optimal Solution

PRINT-OPT-PARENS(s, i, j)

**if** i = j

   **then** print “A”$_i$

   **else** print “(”

        PRINT-OPT-PARENS(s, i, s[i, j])

        PRINT-OPT-PARENS(s, s[i, j] + 1, j)

        print “)”

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

i

j

# Example: $A_1 \cdot \cdot \cdot A_6$

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

**s[1..6, 1..6]**

| j \ i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

P-O-P(s, 1, 6)  s[1, 6] = 3
i = 1, j = 6  "("  P-O-P (s, 1, 3)  s[1, 3] = 1
  i = 1, j = 3  "("  P-O-P(s, 1, 1)  $\Rightarrow$ "$A_1$"
    P-O-P(s, 2, 3) s[2, 3] = 2
    i = 2, j = 3  "("  P-O-P (s, 2, 2) $\Rightarrow$ "$A_2$"
      P-O-P (s, 3, 3) $\Rightarrow$ "$A_3$"
      ")"
  ")"
…

# 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 ← length[p] – 1
2. **for** i ← 1 **to** n
3. **do for** j ← i **to** n
4. **do** m[i, j] ← ∞
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)

**if** m[i, j] $< \infty$<br>
        **then return** m[i, j]<br>
**if** i = j<br>
    **then** m[i, j] $\leftarrow$ 0<br>
    **else for** k $\leftarrow$ i **to** j – 1<br>
            **do** q $\leftarrow$ LOOKUP-CHAIN(p, i, k) +<br>
                LOOKUP-CHAIN(p, k+1, j) + $p_{i-1}p_kp_j$<br>
                **if** q < m[i, j]<br>
                    **then** m[i, j] $\leftarrow$ q<br>
**return** m[i, j]

Running time is $O(n^3)$

# 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

# Matrix-Chain Multiplication (Summary)

- Both the dynamic programming approach and the memoized algorithm can solve the matrix-chain multiplication problem in $O(n^3)$
- Both methods take advantage of the overlapping subproblems property
- There are only $\Theta(n^2)$ different subproblems
  - Solutions to these problems are computed only once
- Without memoization the natural recursive algorithm runs in exponential time

# Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

# Parameters of Optimal Substructure

- How many subproblems are used in an optimal solution for the original problem
  - Assembly line: One subproblem (the line that gives best time)
  - Matrix multiplication: Two subproblems (subproducts $A_{i..k}$, $A_{k+1..j}$)
- How many choices we have in determining which subproblems to use in an optimal solution
  - Assembly line: Two choices (line 1 or line 2)
  - Matrix multiplication: $j - i$ choices for $k$ (splitting the product)

# Parameters of Optimal Substructure

- Intuitively, the running time of a dynamic programming algorithm depends on two factors:
  - Number of subproblems overall
  - How many choices we look at for each subproblem
- Assembly line
  - $\Theta(n)$ subproblems (n stations)
  - 2 choices for each subproblem

$\Theta(n)$ overall

- Matrix multiplication:
  - $\Theta(n^2)$ subproblems ($1 \leq i \leq j \leq n$)
  - At most n-1 choices

$\Theta(n^3)$ overall

# Longest Common Subsequence

- Given two sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y
- E.g.:

  $X = \langle A, B, C, B, D, A, B \rangle$
- Subsequences of X:
  - A subset of elements in the sequence taken in order

  $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

# Subsequences

- A ***subsequence*** of a character string $x_0x_1x_2\ldots x_{n-1}$ is a string of the form $x_{i1}x_{i2}\ldots x_{ik}$, where $i_j < i_{j+1}$.

- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGJIK
  - Subsequence: DFGHK
  - <u>Not subsequence: DAGH</u>

# The Longest Common Subsequence (LCS) Problem

- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y
- Has applications to DNA similarity testing (alphabet in DNA is {A,C,G,T})
- Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

# Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)

- $\langle B, C, A \rangle$, however is not a LCS of X and Y

# Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are $2^m$ subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n2^m)$

# Making the choice

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$

- Choice: include one element into the common sequence (E) and solve the resulting subproblem

$X = \langle A, B, D, G \rangle$

$Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

# Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ ,we define the i-th prefix of X, for i = 0, 1, 2, …, m

$$X_i = \langle x_1, x_2, \ldots, x_i \rangle$$

- c[i, j] = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

# A Recursive Solution

Case 1: $x_i = y_j$

e.g.: $X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$

$$c[i, j] = \quad c[i - 1, j - 1] + 1$$

- Append $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$
- Must find a LCS of $X_{i-1}$ and $Y_{j-1}$ $\Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

# A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$\mathbf{c[i, j] = \max \{ c[i - 1, j], c[i, j-1] \}}$$

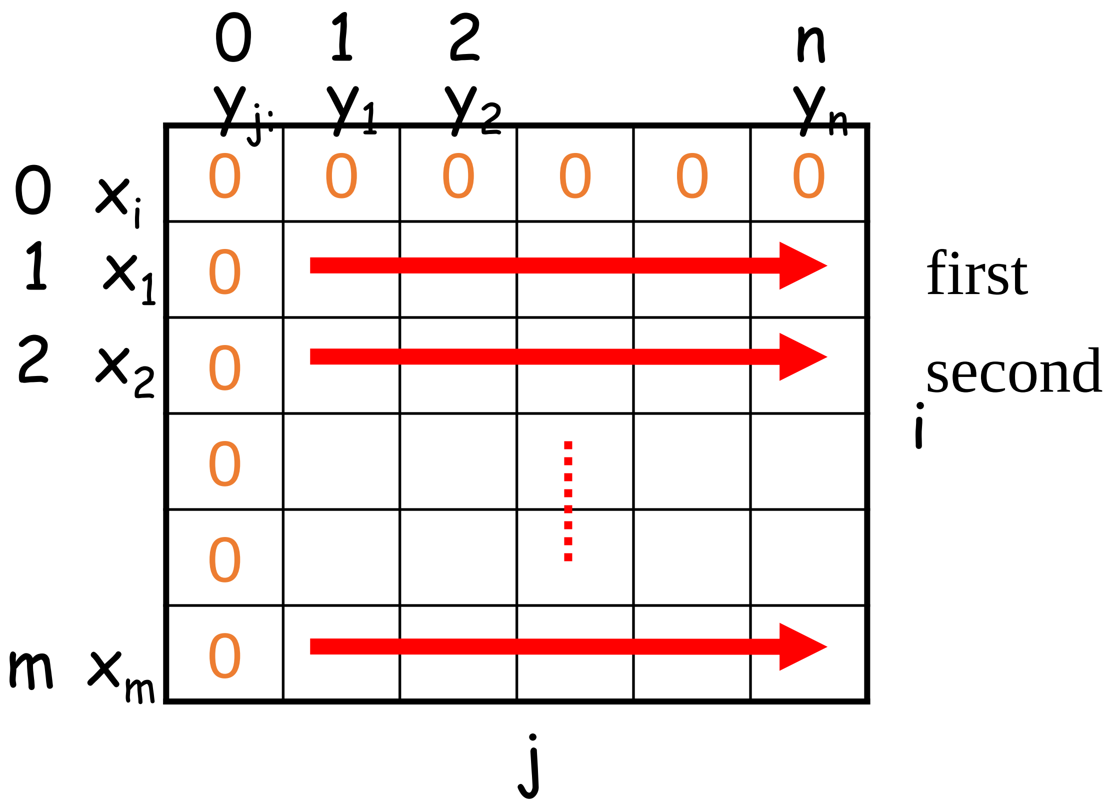
- Must solve two problems
  - find a LCS of $X_{i-1}$ and $Y_j$: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
  - find a LCS of $X_i$ and $Y_{j-1}$: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$
- Optimal solution to a problem includes optimal solutions to subproblems

# Overlapping Subproblems

- To find a LCS of X and Y
  - we may need to find the LCS between X and $Y_{n-1}$ and that of $X_{m-1}$ and Y
  - Both the above subproblems has the subproblem of finding the LCS of $X_{m-1}$ and $Y_{n-1}$
- Subproblems share subsubproblems

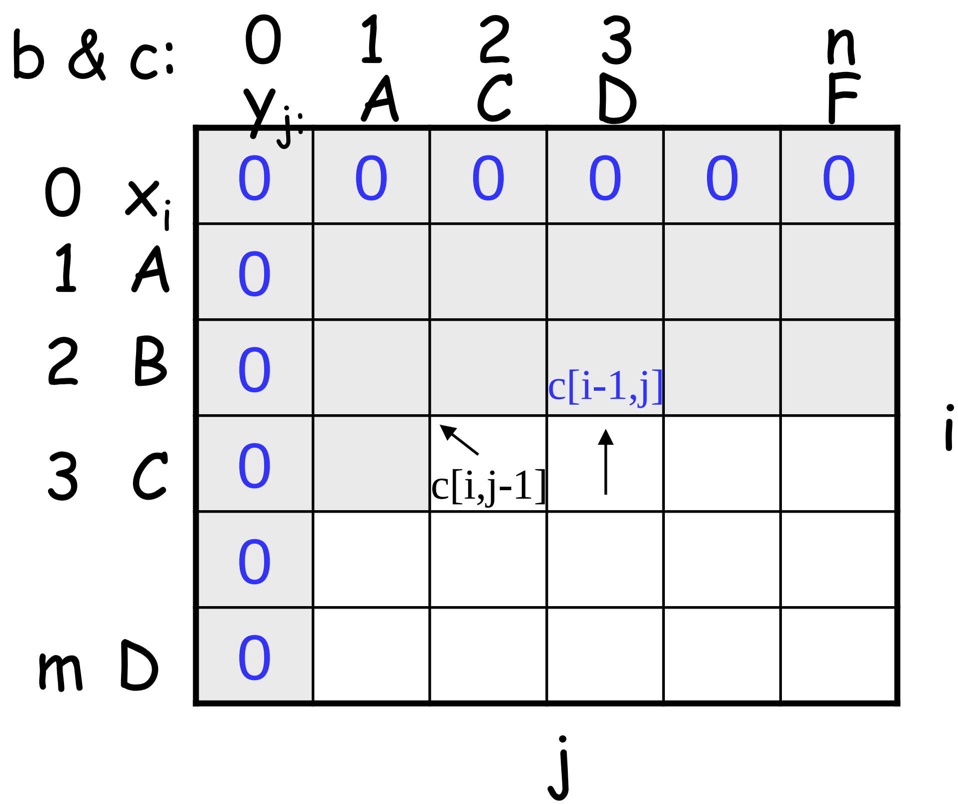
# 3. Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$



# Additional Information

$$c[i,j] = \begin{cases} 0 & \text{if } i,j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$




matrix b[i, j]:

- For a subproblem [i, j], it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$
  
  b[i, j] = "↖"
- Else, if c[i - 1, j] ≥ c[i, j-1]
  
  b[i, j] = " ↑ "
  
  else
  
  b[i, j] = " ← "

# LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j - 1] + 1
                 b[i, j ] ← "    "
            else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                     b[i, j] ← "↑"
                else c[i, j] ← c[i, j - 1]
                     b[i, j] ← "←"
return c and b
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

# Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

If $x_i = y_j$

$b[i, j] = $ "↖"

Else if $c[i - 1, j] \geq c[i, j-1]$

$b[i, j] = $ "↑"

else

$b[i, j] = $ "←"

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | B | D | C | A | B | A |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

# 4. Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a " ↖" in b[i, j] ⇒ $x_i = y_j$ is an element of the LCS

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | B | D | C | A | B | A |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

# PRINT-LCS(b, X, i, j)

**if** i = 0 or j = 0<br>
  **then return**<br>
**if** b[i, j] = "↖"<br>
 **then** PRINT-LCS(b, X, i - 1, j - 1)<br>
   print $x_i$<br>
**elseif** b[i, j] = "↑"<br>
   **then** PRINT-LCS(b, X, i - 1, j)<br>
   **else** PRINT-LCS(b, X, i, j - 1)

Running time: $\Theta(m + n)$

Initial call: PRINT-LCS(b, X, length[X], length[Y])

# Improving the Code

- What can we say about how each entry c[i, j] is computed?
  - It depends only on c[i -1, j - 1], c[i - 1, j], and c[i, j - 1]
  - Eliminate table b and compute in O(1) which of the three values was used to compute c[i, j]
  - We save $\Theta(mn)$ space from table b
  - However, we do not asymptotically decrease the auxiliary space requirements: still need table c

# Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row  being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows

# Applications of Dynamic Programming

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations Research(OR).
  - Computer science:  theory, graphics, AI, systems, ….