

# Dynamic Programming



# Matrix Chain-Products (not in book)



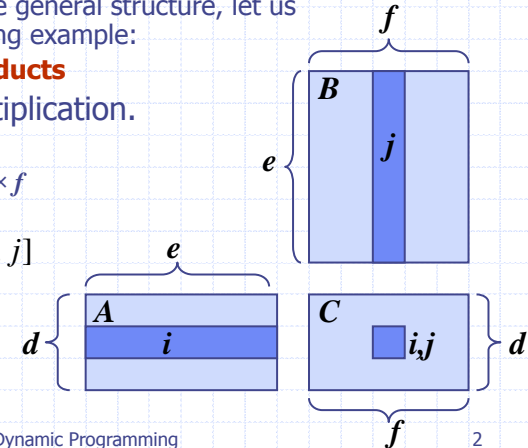
- ◆ **Dynamic Programming** is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**

- ◆ Review: Matrix Multiplication.

- $C = A * B$
- $A$  is  $d \times e$  and  $B$  is  $e \times f$

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] * B[k, j]$$

- $O(def)$  time



# Matrix Chain-Products



## ◆ Matrix Chain-Product:

- Compute  $A = A_0 * A_1 * \dots * A_{n-1}$
- $A_i$  is  $d_i \times d_{i+1}$
- Problem: How to parenthesize?

## ◆ Example

- B is  $3 \times 100$
- C is  $100 \times 5$
- D is  $5 \times 5$
- $(B * C) * D$  takes  $1500 + 75 = 1575$  ops
- $B * (C * D)$  takes  $1500 + 2500 = 4000$  ops

# An Enumeration Approach



## ◆ Matrix Chain-Product Alg.:

- Try all possible ways to parenthesize  $A = A_0 * A_1 * \dots * A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

## ◆ Running time:

- The number of paranthesizations is equal to the number of binary trees with  $n$  nodes
- This is **exponential**!
- It is called the Catalan number, and it is almost  $4^n$ .
- This is a terrible algorithm!

## A Greedy Approach



- ◆ Idea #1: repeatedly select the product that uses (up) the most operations.
- ◆ Counter-example:
  - A is  $10 \times 5$
  - B is  $5 \times 10$
  - C is  $10 \times 5$
  - D is  $5 \times 10$
  - Greedy idea #1 gives  $(A*B)*(C*D)$ , which takes  $500+1000+500 = 2000$  ops
  - $A*((B*C)*D)$  takes  $500+250+250 = 1000$  ops

## Another Greedy Approach



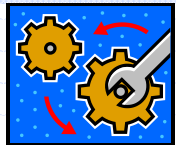
- ◆ Idea #2: repeatedly select the product that uses the fewest operations.
- ◆ Counter-example:
  - A is  $101 \times 11$
  - B is  $11 \times 9$
  - C is  $9 \times 100$
  - D is  $100 \times 99$
  - Greedy idea #2 gives  $A*((B*C)*D)$ , which takes  $109989+9900+108900=228789$  ops
  - $(A*B)*(C*D)$  takes  $9999+89991+89100=189090$  ops
- ◆ The greedy approach is not giving us the optimal value.

## A "Recursive" Approach



- ◆ Define **subproblems**:
  - Find the best parenthesization of  $A_i * A_{i+1} * \dots * A_j$ .
  - Let  $N_{i,j}$  denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is  $N_{0,n-1}$ .
- ◆ **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index  $i$ :  $(A_0 * \dots * A_i) * (A_{i+1} * \dots * A_{n-1})$ .
  - Then the optimal solution  $N_{0,n-1}$  is the sum of two optimal subproblems,  $N_{0,i}$  and  $N_{i+1,n-1}$  plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

## A Characterizing Equation



- ◆ The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- ◆ Let us consider all possible places for that final multiply:
  - Recall that  $A_i$  is a  $d_i \times d_{i+1}$  dimensional matrix.
  - So, a characterizing equation for  $N_{i,j}$  is the following:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- ◆ Note that subproblems are not independent--the **subproblems overlap**.

# A Dynamic Programming Algorithm



- ◆ Since **subproblems overlap**, we don't use recursion.
- ◆ Instead, we construct optimal subproblems "bottom-up."
- ◆  $N_{i,i}$ 's are easy, so start with them
- ◆ Then do length 2,3,... subproblems, and so on.
- ◆ The running time is  $O(n^3)$

**Algorithm *matrixChain(S)*:**

**Input:** sequence  $S$  of  $n$  matrices to be multiplied

**Output:** number of operations in an optimal parenethization of  $S$

**for**  $i \leftarrow 1$  **to**  $n-1$  **do**

$N_{i,i} \leftarrow 0$

**for**  $b \leftarrow 1$  **to**  $n-1$  **do**

**for**  $i \leftarrow 0$  **to**  $n-b-1$  **do**

$j \leftarrow i+b$

$N_{i,j} \leftarrow +\text{infinity}$

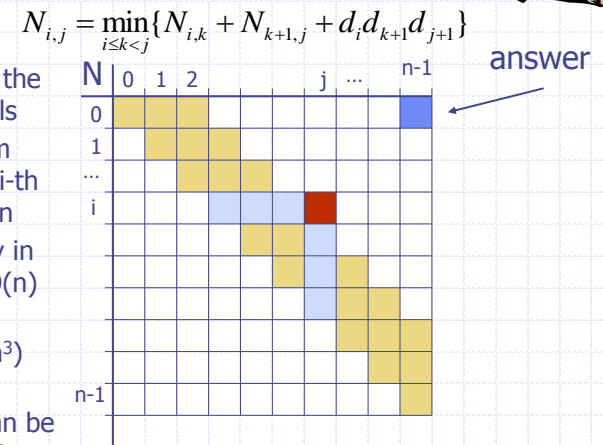
**for**  $k \leftarrow i$  **to**  $j-1$  **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

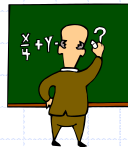
# A Dynamic Programming Algorithm Visualization



- ◆ The bottom-up construction fills in the  $N$  array by diagonals
- ◆  $N_{i,j}$  gets values from pervious entries in  $i$ -th row and  $j$ -th column
- ◆ Filling in each entry in the  $N$  table takes  $O(n)$  time.
- ◆ Total run time:  $O(n^3)$
- ◆ Getting actual parenthesization can be done by remembering "k" for each  $N$  entry



# The General Dynamic Programming Technique



- ◆ Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as  $j$ ,  $k$ ,  $l$ ,  $m$ , and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

# Subsequences

- ◆ A **subsequence** of a character string  $x_0x_1x_2\dots x_{n-1}$  is a string of the form  $x_{i_1}x_{i_2}\dots x_{i_k}$ , where  $i_j < i_{j+1}$ .
- ◆ Not the same as substring!
- ◆ Example String: ABCDEFGHIJK
  - Subsequence: ACEGJIK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

# 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 is {A,C,G,T})
- ◆ Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

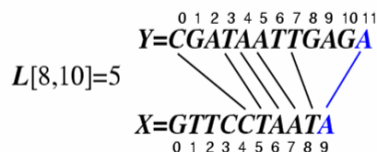
# A Poor Approach to the LCS Problem

- ◆ A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- ◆ Analysis:
  - If X is of length n, then it has  $2^n$  subsequences
  - This is an exponential-time algorithm!

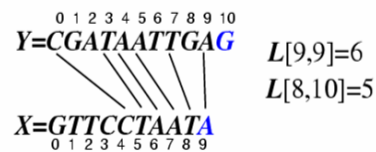
# A Dynamic-Programming Approach to the LCS Problem

- ◆ Define  $L[i,j]$  to be the length of the longest common subsequence of  $X[0..i]$  and  $Y[0..j]$ .
- ◆ Allow for -1 as an index, so  $L[-1,k] = 0$  and  $L[k,-1] = 0$ , to indicate that the null part of X or Y has no match with the other.
- ◆ Then we can define  $L[i,j]$  in the general case as follows:
  1. If  $x_i = y_j$ , then  $L[i,j] = L[i-1,j-1] + 1$  (we can add this match)
  2. If  $x_i \neq y_j$ , then  $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$  (we have no match here)

Case 1:



Case 2:



# An LCS Algorithm

**Algorithm** LCS(X,Y ):

**Input:** Strings X and Y with n and m elements, respectively

**Output:** For  $i = 0, \dots, n-1$ ,  $j = 0, \dots, m-1$ , the length  $L[i, j]$  of a longest string that is a subsequence of both the string  $X[0..i] = x_0x_1x_2\dots x_i$  and the string  $Y[0..j] = y_0y_1y_2\dots y_j$

```

for i = 1 to n-1 do
    L[i,-1] = 0
for j = 0 to m-1 do
    L[-1,j] = 0
for i = 0 to n-1 do
    for j = 0 to m-1 do
        if  $x_i = y_j$  then
            L[i, j] = L[i-1, j-1] + 1
        else
            L[i, j] = max{L[i-1, j] , L[i, j-1]}
return array L
    
```

# Visualizing the LCS Algorithm

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6

$Y = \text{CGATAATTGAGA}$   
 $X = \text{GTTCTAATA}$

# Analysis of LCS Algorithm

- ◆ We have two nested loops
  - The outer one iterates  $n$  times
  - The inner one iterates  $m$  times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is  $O(nm)$
- ◆ Answer is contained in  $L[n,m]$  (and the subsequence can be recovered from the L table).