Modified by: Daniel Gomez-Prado, University of Massachusetts Amherst
Then modified by Gokberk Cinbis for METU CENG 315

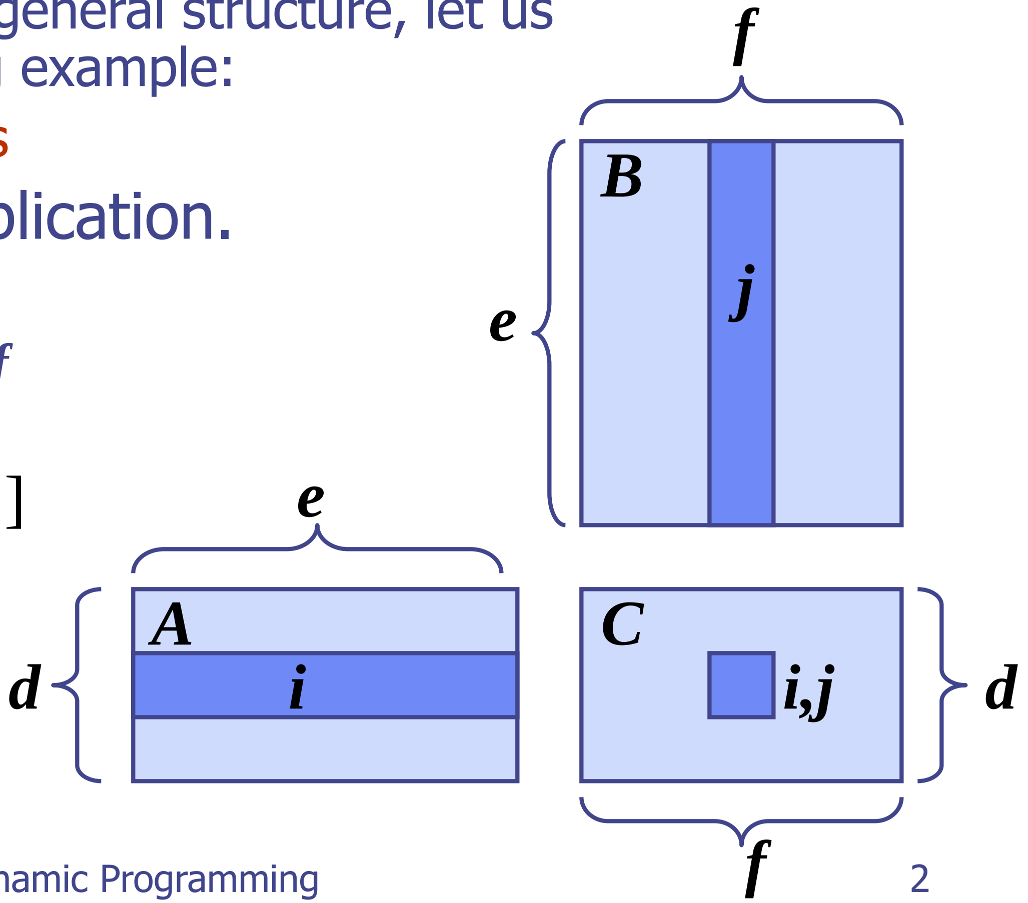
Dynamic Programming

Matrix Chain-Products

- 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 $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 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 $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- **Running time:**
  - The number of paranethesizations 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 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - But A*((B*C)*D) takes 500+250+250 = 1000 ops

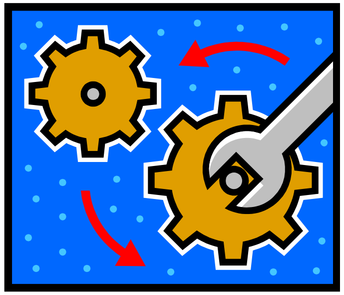
Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 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 $A_i*A_{i+1}*\ldots*A_j$.
  - 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: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - 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

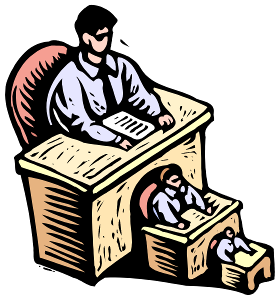


- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the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

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

- 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.
- Running time: $O(n^3)$

Algorithm *matrixChain*(*S*):

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

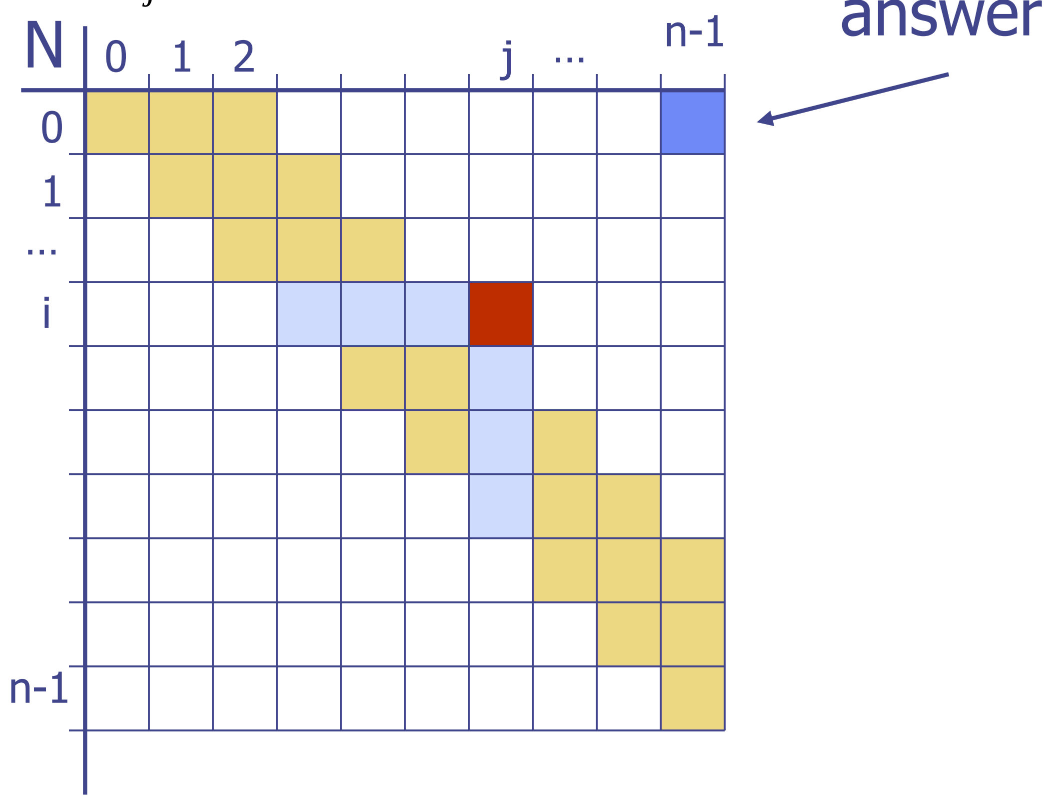
Output: number of operations in an optimal paranethization of *S*

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**
    $j \leftarrow i+b$
    $N_{i,j} \leftarrow$ +**infinity**
    for $k \leftarrow$ i **to** $j-1$ **do**
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

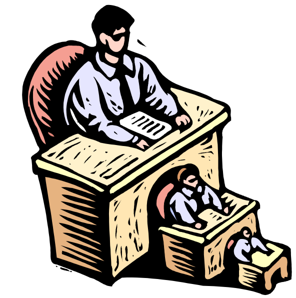
A Dynamic Programming Algorithm Visualization

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

- 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



Matrix Chain algorithm



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

Input: sequence S of n matrices to be multiplied

Output: # of multiplications in optimal parenthesization of S

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

  $N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n-1$ **do** // b is # of ops in S

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

    $j \leftarrow i+b$

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

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

      sum = $N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}$

      if (sum $< N_{i,j}$) **then**

        $N_{i,j} \leftarrow$ sum

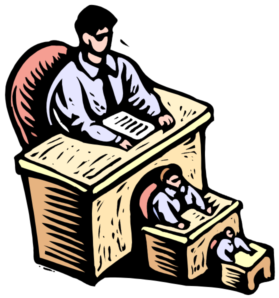
        $O_{i,j} \leftarrow k$

return $N_{0,n-1}$

- **Example: ABCD**
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10

N	0	1	2	3
0	**0** A	**500** 0 AB	**500** 0 A(BC)	**1000** 2 (A(BC))D
1		**0** B	**250** 0 BC	**500** 1 (BC)D
2			**0** C	**500** 0 CD
3				**0** D

Recovering operations



- **Example: ABCD**
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10

N	0	1	2	3
0	**0** A	**500** **0** AB	**500** **0** A(BC)	**1000** **2** (A(BC))D
1		**0** B	**250** **0** BC	**500** **1** (BC)D
2			**0** C	**500** **0** CD
3				**0** D

// return expression for multiplying
// matrix chain A_i through A_j

exp(*i*,*j*)
    if (*i*=*j*) then // base case, 1 matrix
        return ‘A_i’
    else
        ***k* = O[*i*,*j*]** // see red values on left
        S1 = exp(*i*,*k*) // 2 recursive calls
        S2 = exp(*k*+*1*,*j*)
        return ‘(‘ S1 S2 ‘)’

Conclusions

- Dynamic programming is a useful technique for solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time
  - Naïve algorithm: $O(4^n)$
  - DP: $O(n^3)$