CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/CSE214_2018

Lecture 8

Dynamic Programming

Based on Cevdet Aykanat's and Mustafa Ozdal's Lecture Notes - Bilkent

Introduction

- An algorithm design paradigm like divide-and-conquer
- "Programming": A tabular method (not writing computer code)

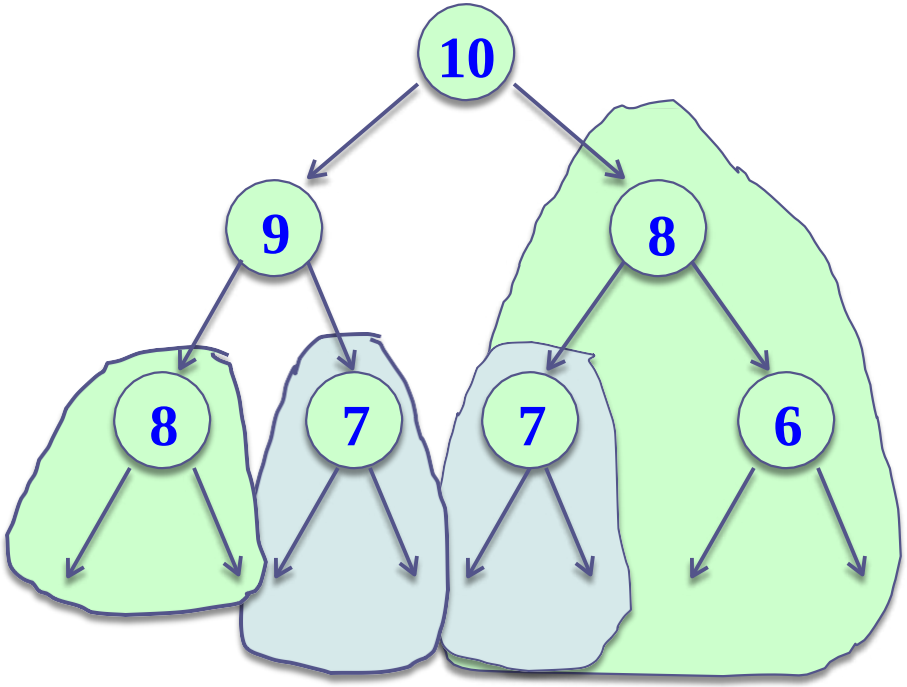
  Older sense of planning or scheduling, typically by filling in a table
- Divide-and-Conquer (DAC): subproblems are independent
- Dynamic Programming (DP): subproblems are not independent
- Overlapping subproblems: subproblems share sub-subproblems
  - In solving problems with overlapping subproblems
    - A DAC algorithm does redundant work
      - Repeatedly solves common subproblems
    - A DP algorithm solves each problem just once
      - Saves its result in a table

Example: Fibonacci Numbers (Recursive Solution)

Reminder:
F(0) = 0 and F(1) = 1
F(n) = F(n-1) + F(n-2)

REC-FIBO(n)

```
if n < 2
    return n
else
    return REC-FIBO(n-1)
           + REC-FIBO(n-2)
```



Overlapping subproblems in different recursive calls. Repeated work!

Example: Fibonacci Numbers (Recursive Solution)

Recurrence:

$$T(n) = T(n-1) + T(n-2) + 1$$

➔ exponential runtime

Recursive algorithm inefficient because it recomputes the same F(i) repeatedly in different branches of the recursion tree.

Example: Fibonacci Numbers (Bottom-up Computation)

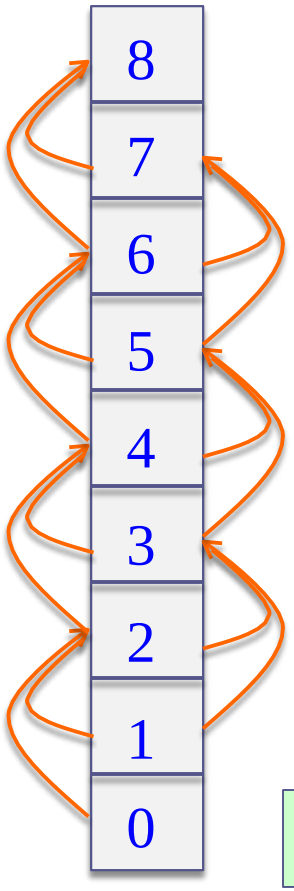
Reminder:

F(0) = 0 and F(1) = 1

F(n) = F(n-1) + F(n-2)

ITER-FIBO(n)

```
F[0] = 0
F[1] = 1
for i = 2 to n do
    F[i] = F[i-1] + F[i-2]
return F[n]
```



Runtime: $\Theta(n)$

Optimization Problems

- DP typically applied to optimization problems
- In an optimization problem
  - There are many possible solutions (feasible solutions)
  - Each solution has a value
  - Want to find an optimal solution to the problem
    - A solution with the optimal value (min or max value)
  - Wrong to say "the" optimal solution to the problem
    - There may be several solutions with the same optimal value

Development of a DP Algorithm

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from the information computed in Step 3

Example: Matrix-chain Multiplication

- *Input*: a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices
- *Aim*: compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- A product of matrices is fully parenthesized if
  - It is either a single matrix
  - Or, the product of two fully parenthesized matrix products surrounded by a pair of parentheses.

$$(A_i(A_{i+1}A_{i+2} \ldots A_j))$$

$$((A_iA_{i+1}A_{i+2} \ldots A_{j-1})A_j)$$

$$((A_iA_{i+1}A_{i+2} \ldots A_k)(A_{k+1}A_{k+2} \ldots A_j)) \qquad \text{for } i \le k < j$$

  - *All parenthesizations yield the same product; matrix product is associative*

Matrix-chain Multiplication: An Example Parenthesization

- *Input*: $\langle A_1, A_2, A_3, A_4 \rangle$
- 5 distinct ways of full parenthesization

  $(A_1(A_2(A_3A_4)))$

  $(A_1((A_2A_3)A_4))$

  $((A_1A_2)(A_3A_4))$

  $((A_1(A_2A_3))A_4)$

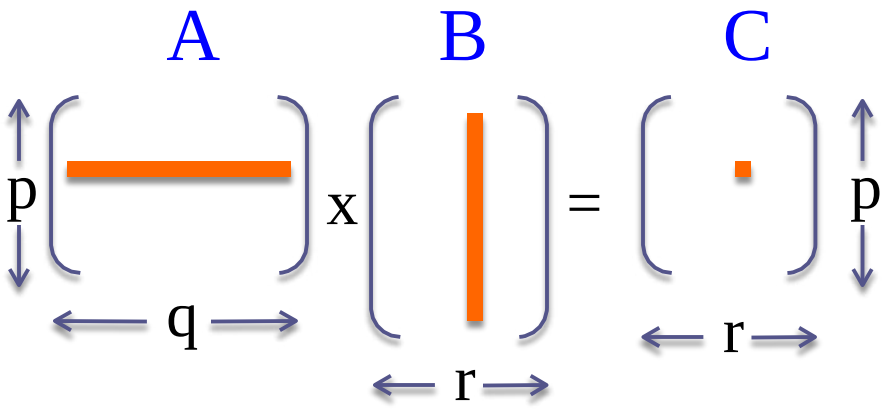
  $(((A_1A_2)A_3)A_4)$

- The way we parenthesize a chain of matrices can have a dramatic effect on the cost of computing the product

Reminder: Matrix Multiplication

MATRIX-MULTIPLY(A, B)

```
if cols[A]≠rows[B] then
error("incompatible dimensions")
for i ←1 to rows[A] do
    for j←1 to cols[B] do
        C[i,j] ← 0
        for k←1 to cols[A] do
            C[i,j]← C[i,j]+A[i,k]·B[k,j]
return C
```



rows(A) = p
cols(A) = q

rows(B) = q
cols(B) = r

rows(C) = p
cols(C) = r

Reminder: Matrix Multiplication

MATRIX-MULTIPLY(A, B)

```
if cols[A]≠rows[B] then
error("incompatible dimensions")
for i ←1 to rows[A] do
    for j←1 to cols[B] do
        C[i,j] ← 0
        for k←1 to cols[A] do
            C[i,j]← C[i,j]+A[i,k]·B[k,j]
return C
```

A: p x q
B: q x r

C: p x r

# of mult-add ops
= rows[A] x cols[B] x cols[A]

# of mult-add ops = p x q x r

Matrix Chain Multiplication: Example

A_1: 10x100 A_2: 100x5 A_3: 5x50

Which paranthesization is better? $(A_1A_2)A_3$ or $A_1(A_2A_3)$?

$$\underset{10}{\left[\begin{array}{c}100\\A_1\end{array}\right]} \times \underset{100}{\left[\begin{array}{c}5\\A_2\end{array}\right]} = \underset{10}{\left[\begin{array}{c}5\\A_1A_2\end{array}\right]}$$

# of ops: 10 . 100 . 5 = 5000

$$\underset{10}{\left[\begin{array}{c}5\\A_1A_2\end{array}\right]} \times \underset{5}{\left[\begin{array}{c}50\\A_3\end{array}\right]} = \underset{10}{\left[\begin{array}{c}50\\A_1A_2A_3\end{array}\right]}$$

# of ops: 10 . 5 . 50 = 2500

Total # of ops: 7500

Matrix Chain Multiplication: Example

A_1: 10x100 A_2: 100x5 A_3: 5x50

Which paranthesization is better? $(A_1A_2)A_3$ or $A_1(A_2A_3)$?

$$\underset{100}{\left[\begin{array}{c}5\\ A_2\end{array}\right]} \times \underset{5}{\left[\begin{array}{c}50\\ A_3\end{array}\right]} = \underset{100}{\left[\begin{array}{c}50\\ A_2A_3\end{array}\right]}$$

# of ops: 100 . 5 . 50 = 25000

$$\underset{10}{\left[\begin{array}{c}100\\ A_1\end{array}\right]} \times \underset{100}{\left[\begin{array}{c}50\\ A_2A_3\end{array}\right]} = \underset{10}{\left[\begin{array}{c}50\\ A_1A_2A_3\end{array}\right]}$$

# of ops: 10 . 100 . 50 = 50000

Total # of ops: 75000

Matrix Chain Multiplication: Example

A_1: 10x100 A_2: 100x5 A_3: 5x50

Which paranthesization is better? $(A_1A_2)A_3$ or $A_1(A_2A_3)$?

In summary: $(A_1A_2)A_3$ ➔ # of multiply-add ops: 7500

$A_1(A_2A_3)$ ➔ # of multiple-add ops: 75000

➔ First parenthesization yields 10x faster computation

Matrix-chain Multiplication Problem

Input: A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices,

where A_i is a $p_{i-1} \times p_i$ matrix

Objective: Fully parenthesize the product

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

such that the number of scalar mult-adds is minimized.

Counting the Number of Parenthesizations

- ***Brute force approach***: exhaustively check all parenthesizations
- $P(n)$: # of parenthesizations of a sequence of n matrices
- We can split sequence between k^{th} and $(k+1)^{st}$ matrices for any $k=1, 2, \ldots, n-1$, then parenthesize the two resulting sequences independently, i.e.,

$$(A_1 A_2 A_3 \ldots A_k)(A_{k+1} A_{k+2} \ldots A_n)$$

- We obtain the recurrence

$$P(1) = 1 \text{ and } P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

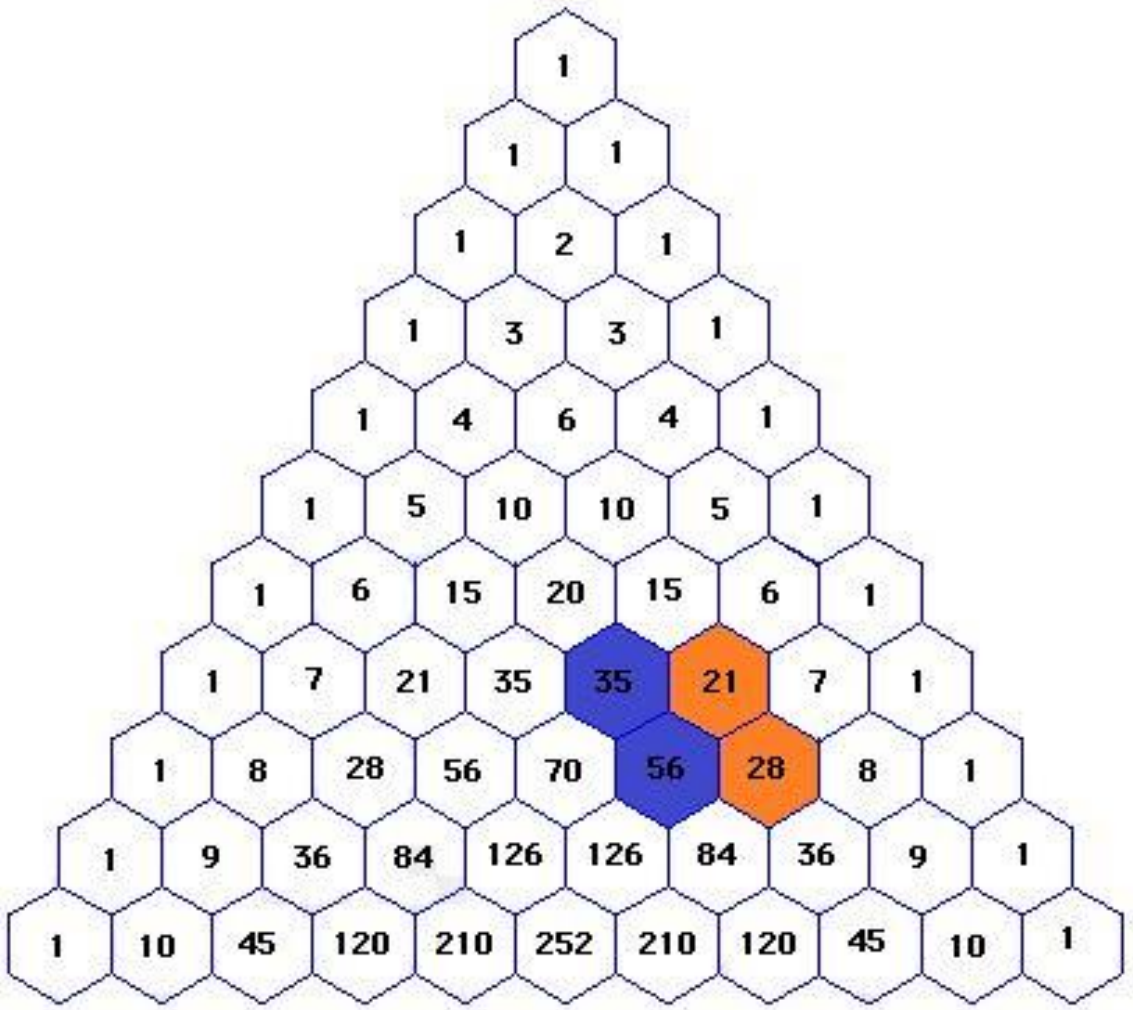
Number of Parenthesizations: $\sum_{k=1}^{n-1} P(k)P(n-k)$

- The recurrence generates the sequence of Catalan Numbers
- Solution is $P(n) = C(n-1)$ where

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$$

- The number of solutions is exponential in n
- Therefore, brute force approach is a poor strategy

Catalan Numbers Example



The Structure of Optimal Parenthesization

Notation: $A_{i..j}$: The matrix that results from evaluation of the product: $A_i\ A_{i+1}\ A_{i+2}\ \ldots\ A_j$

Observation: Consider the last multiplication operation in any parenthesization: $(A_1\ A_2\ \ldots\ A_k)\ .\ (A_{k+1}\ A_{k+2}\ \ldots\ A_n)$

There is a k value ($1 \leq k < n$) such that:

First, the product $A_{1..k}$ is computed

Then, the product $A_{k+1..n}$ is computed

Finally, the matrices $A_{1..k}$ and $A_{k+1..n}$ are multiplied

Step 1: Characterize the Structure of an Optimal Solution

- An optimal parenthesization of product $A_1 A_2 \ldots A_n$ will be:

  $(A_1 A_2 \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_n)$ for some k value

- The cost of this optimal parenthesization will be:

  Cost of computing $A_{1..k}$

  \+ Cost of computing $A_{k+1..n}$

  \+ Cost of multiplying $A_{1..k} \cdot A_{k+1..n}$

Step 1: Characterize the Structure of an Optimal Solution

- ***Key observation***: Given optimal parenthesization

$$(A_1A_2A_3 \ldots A_k) \cdot (A_{k+1}A_{k+2} \ldots A_n)$$

  – Parenthesization of the subchain $A_1A_2A_3 \ldots A_k$
  – Parenthesization of the subchain $A_{k+1}A_{k+2} \ldots A_n$

  should both be optimal

Thus, optimal solution to an instance of the problem contains optimal solutions to subproblem instances

i.e., optimal substructure within an optimal solution exists.

Step 2: A Recursive Solution

Step 2: Define the value of an optimal solution recursively in terms of optimal solutions to the subproblems

Assume we are trying to determine the min cost of computing $A_{i..j}$

$m_{i,j}$: min # of scalar multiply-add operations needed to compute $A_{i..j}$

Note: The optimal cost of the original problem: $m_{1,n}$

How to compute $m_{i,j}$ recursively?

Step 2: A recursive Solution

Base case: $m_{i,i} = 0$ (single matrix, no multiplication)

Let the size of matrix A_i be $(p_{i-1} \times p_i)$
Consider an optimal parenthesization of chain $A_i \ldots A_j$:

$$(A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$$

The optimal cost:

$$m_{i,j} = m_{i,k} + m_{k+1,\,j} + p_{i-1} \times p_k \times p_j$$

where: $m_{i,k}$: Optimal cost of computing $A_{i..k}$

$m_{k+1,j}$: Optimal cost of computing $A_{k+1..j}$

$p_{i-1} \times p_k \times p_j$: Cost of multiplying $A_{i..k}$ and $A_{k+1\ldots j}$

Step 2: A Recursive Solution

In an optimal parenthesization:

k must be chosen to minimize m_{ij}

The recursive formulation for m_{ij}:

$$m_{ij} = \begin{cases} 0 & \text{if } i=j \\ \underset{i \le k < j}{\text{MIN}} \{ m_{ik} + m_{k+1,\,j} + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

Step 2: A Recursive Solution

- The m_{ij} values give the costs of optimal solutions to subproblems
- In order to keep track of how to construct an optimal solution
  - Define s_{ij} to be the value of k which yields the optimal split of the subchain $A_{i..j}$

    That is, $s_{ij} = k$ such that

    $$m_{ij} = m_{ik} + m_{k+1,\,j} + p_{i-1} p_k p_j$$ holds

Direct Recursion: Inefficient!

Recursive matrix-chain order

RMC(p, i, j)

if $i = j$ **then**
  return 0

$m[i, j] \leftarrow \infty$

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

  $q \leftarrow \text{RMC}(p, i, k) + \text{RMC}(p, k+1, j) + p_{i-1} p_k p_j$

  if $q < m[i, j]$ **then**

    $m[i, j] \leftarrow q$

return $m[i, j]$

Computing the Optimal Cost (Matrix-Chain Multiplication)

An important observation:

- We have relatively few subproblems
  - one problem for each choice of i and j satisfying $1 \le i \le j \le n$
  - total $n + (n-1) + \ldots + 2 + 1 = \frac{1}{2} n(n+1) = \Theta(n^2)$ subproblems
- We can write a recursive algorithm based on recurrence.
- However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree
- This property, overlapping subproblems, is the second important feature for applicability of dynamic programming

Computing the Optimal Cost (Matrix-Chain Multiplication)

Compute the value of an optimal solution in a bottom-up fashion

- matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$
- the input is a sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where length[p] = $n + 1$

Procedure uses the following auxiliary tables:

- $m[1 \ldots n, 1 \ldots n]$: for storing the $m[i, j]$ costs
- $s[1 \ldots n, 1 \ldots n]$: records which index of k achieved the optimal cost in computing $m[i, j]$

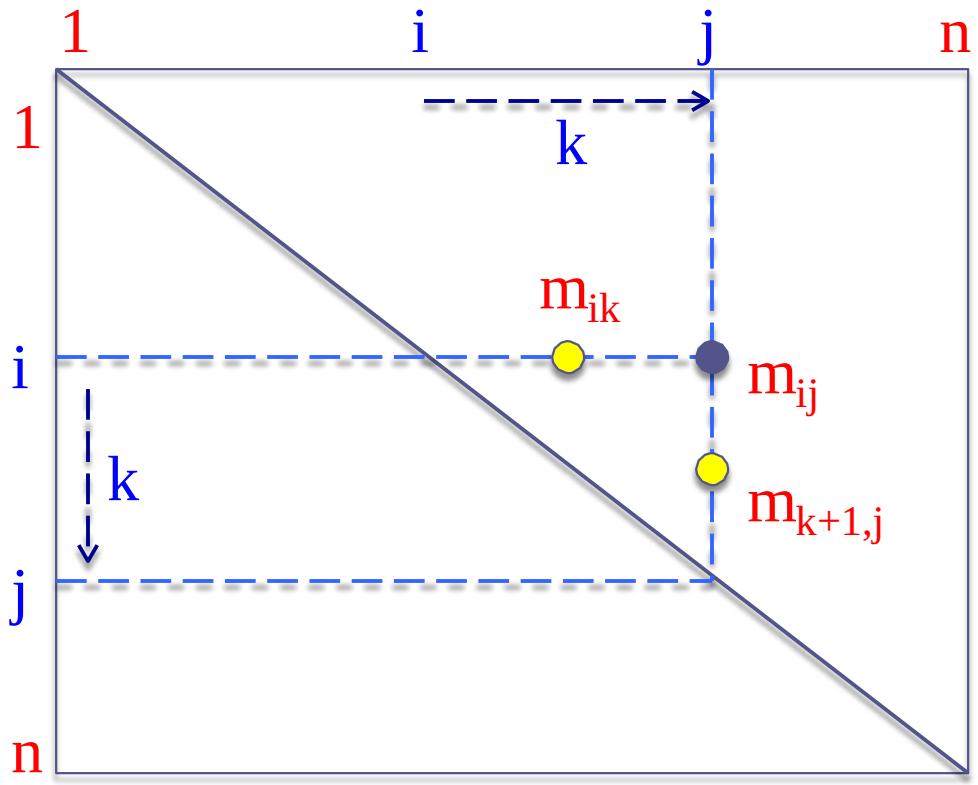
Bottom-up computation

$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$

How to choose the order in which we process m_{ij} values?

Before computing m_{ij}, we have to make sure that the values for m_{ik} and $m_{k+1,j}$ have been computed for all k.

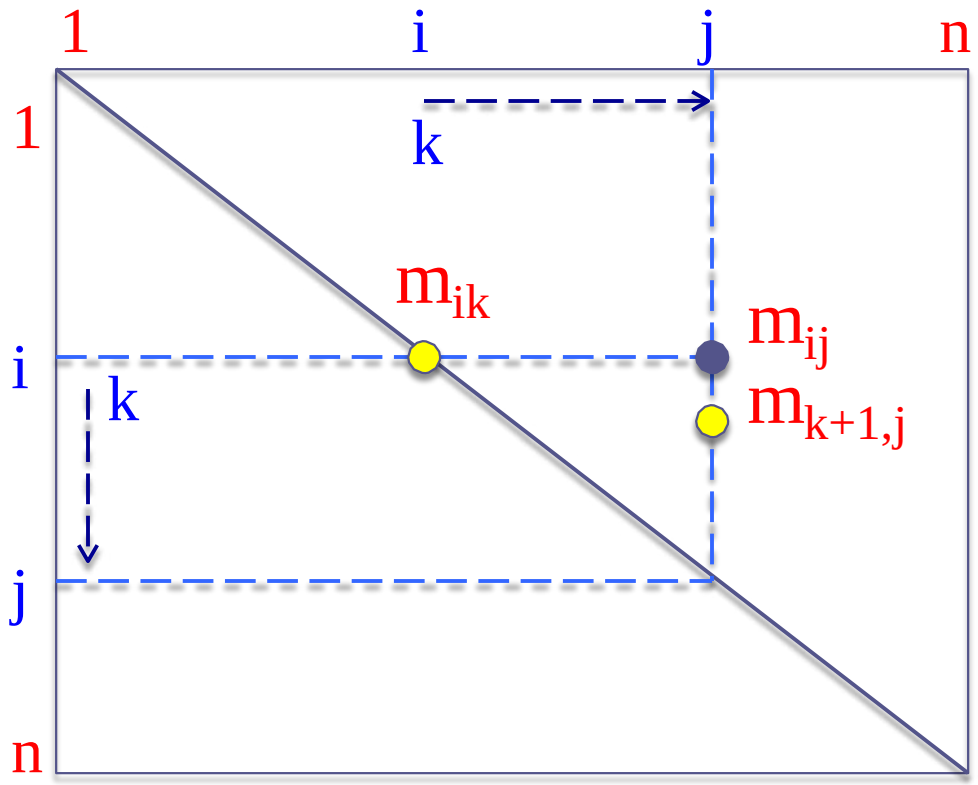
$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$



m_{ij} must be processed after m_{ik} and $m_{j,k+1}$

Reminder: m_{ij} computed only for $j > i$

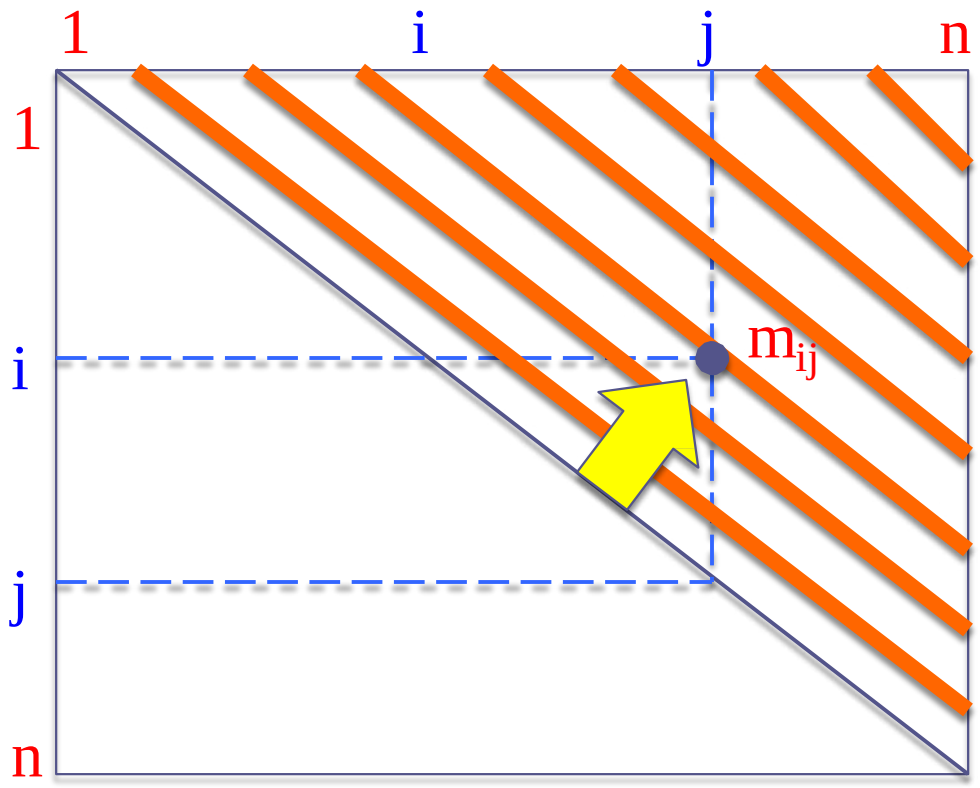
$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$



m_{ij} must be processed after m_{ik} and $m_{j,k+1}$

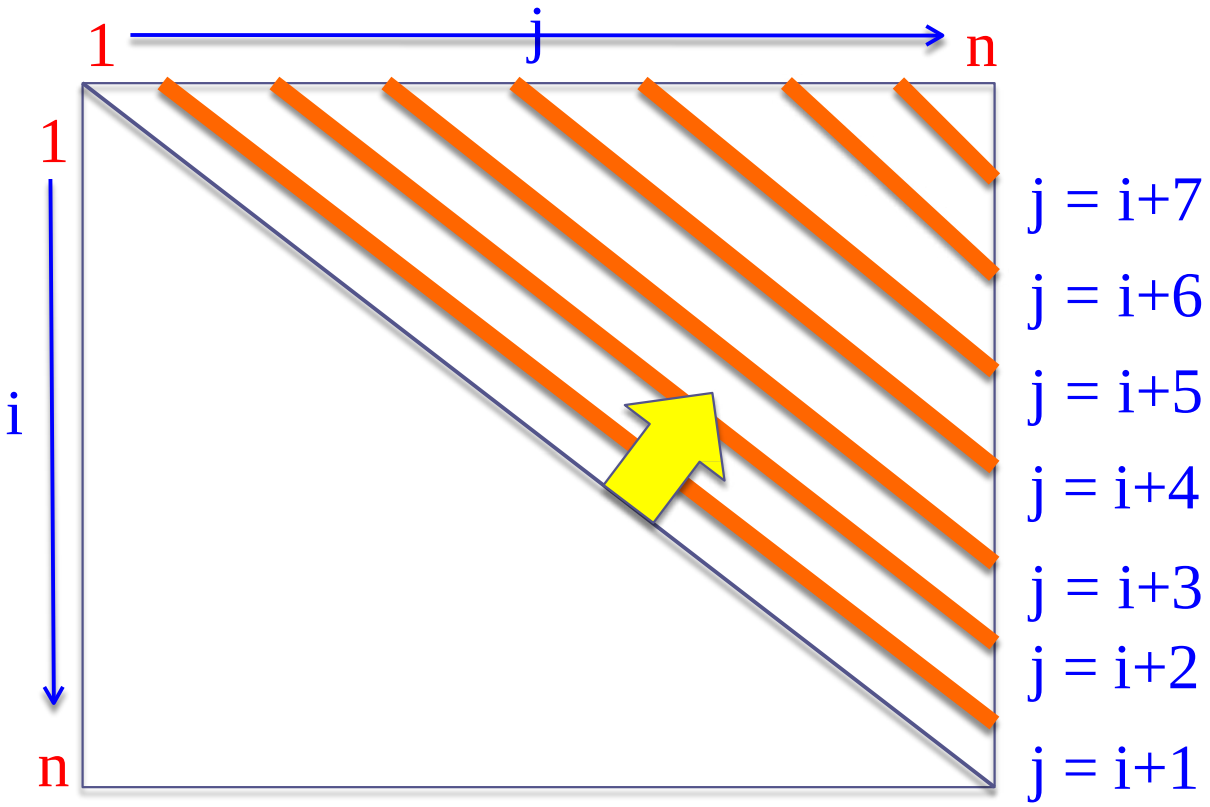
How to set up the iterations over i and j to compute m_{ij}?

$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$

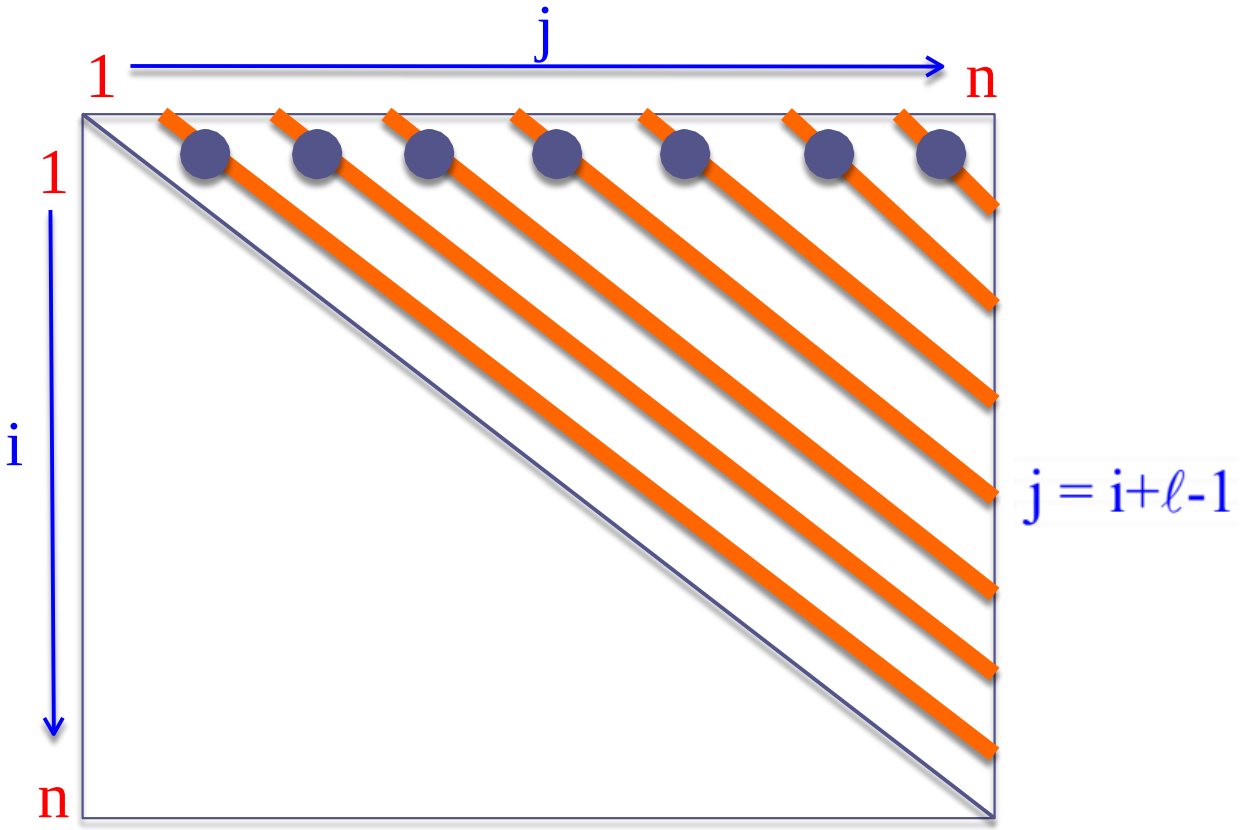


If the entries m_{ij} are computed in the shown order, then m_{ik} and $m_{k+1,j}$ values are guaranteed to be computed before m_{ij}.

$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$



$$m_{ij} = \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \}$$




```
for ℓ=2 to n
    for i=1 to n-ℓ+1
        j = i + ℓ - 1
        ……
        m_ij = …
    ……
```

The Dynamic Programming Algorithm

```
Matrix-Chain(p, n)
{   for (i = 1 to n) m[i, i] = 0;
    for (l = 2 to n)
    {
        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; (Optimum in m[1, n])
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.

Algorithm for Computing the Optimal Costs

- The algorithm first computes

  $m[i, i] \leftarrow 0$ for $i = 1, 2, \ldots, n$ min costs for all chains of length 1

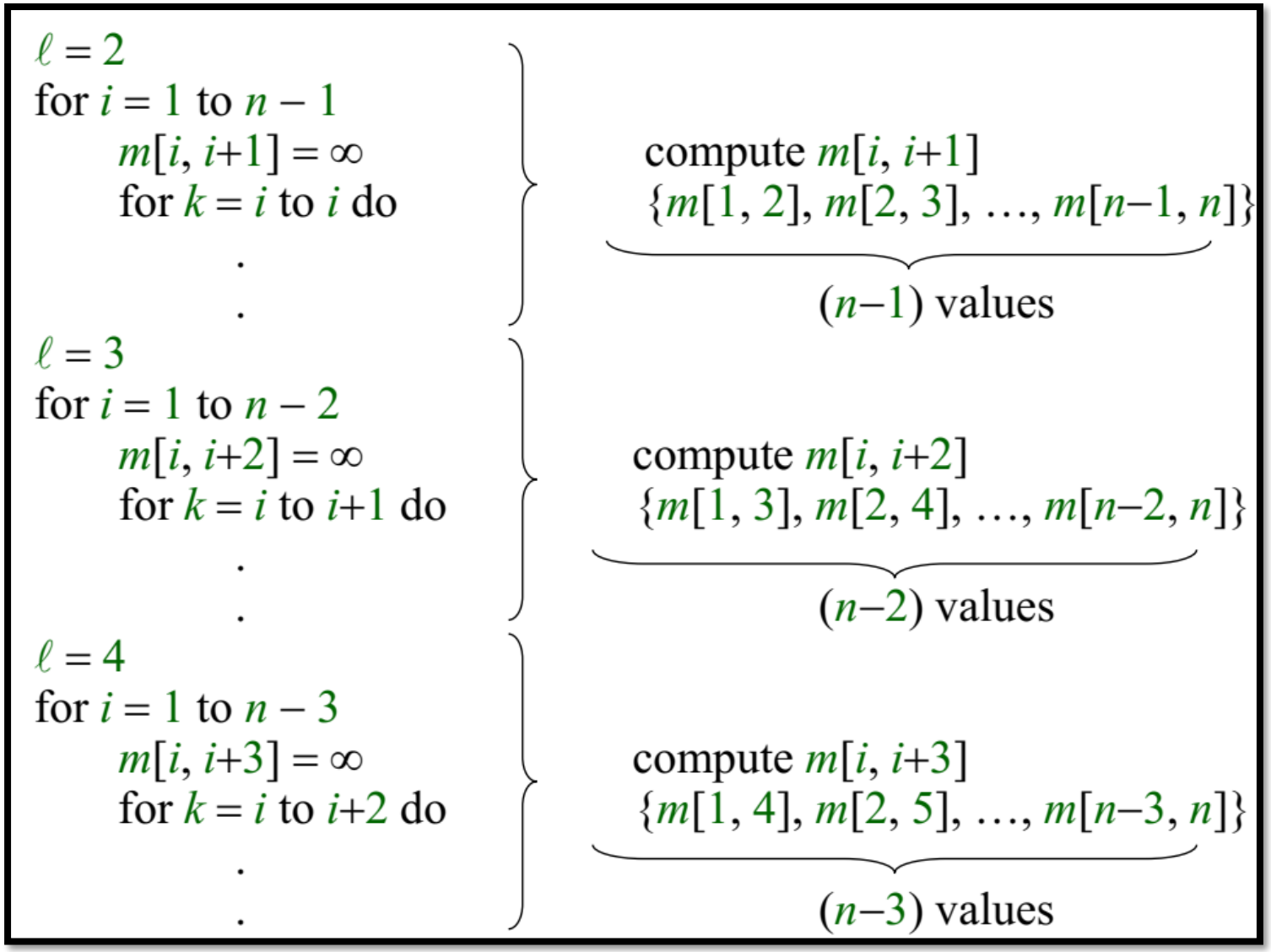
- Then, for $\ell = 2, 3, \ldots, n$ computes

  $m[i, i+\ell-1]$ for $i = 1, \ldots, n-\ell+1$ min costs for all chains of length ℓ

- For each value of $\ell = 2, 3, \ldots, n$,

  $m[i, i+\ell-1]$ depends only on table entries $m[i, k]$ & $m[k+1, i+\ell-1]$ for $i \le k < i+\ell-1$, which are already computed

Algorithm for Computing the Optimal Costs



Constructing an Optimal Solution: Compute $A_{1..n}$

The actual multiplication code uses the $s[i, j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i, j]$ is global to this recursive procedure. The procedure returns a matrix.

```
Mult(A, s, i, j)
{
    if (i < j)
    {
        X = Mult(A, s, i, s[i, j]);
              X is now A_i ··· A_k, where k is s[i, j]
        Y = Mult(A, s, s[i, j] + 1, j);
              Y is now A_{k+1} ··· A_j
        return X * Y;   multiply matrices X and Y
    }
    else return A[i];
}
```

To compute $A_1 A_2 \cdots A_n$, call Mult$(A, s, 1, n)$.

Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.
$m[i, j]$ only defi ned for $i \leq j$.

The important point is that when we use the equation

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

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k + 1, j]$.

Note that $k - i < j - i$ and $j - (k + 1) < j - i$ so, to ensure that $m[i, j]$ is evaluated after $m[i, k]$ and $m[k + 1, j]$ we simply let $\ell = 1, 2, \ldots, n - 1$ and calculate all the terms of the form $m[i, i + \ell]$, $i = 1, \ldots n - \ell$ before we calculate the terms of the form $m[i, i + \ell + 1]$, $i = 1, \ldots n - \ell - 1$. That is, we calculate in the order

$m[1, 2], m[2, 3], m[3, 4], \ldots, m[n - 3, n - 2], m[n - 2, n - 1], m[n - 1, n]$

$m[1, 3], m[2, 4], m[3, 5], \ldots, m[n - 3, n - 1], m[n - 2, n]$

$m[1, 4], m[2, 5], m[3, 6], \ldots, m[n - 3, n]$

$\vdots$

$m[1, n - 1], m[2, n]$

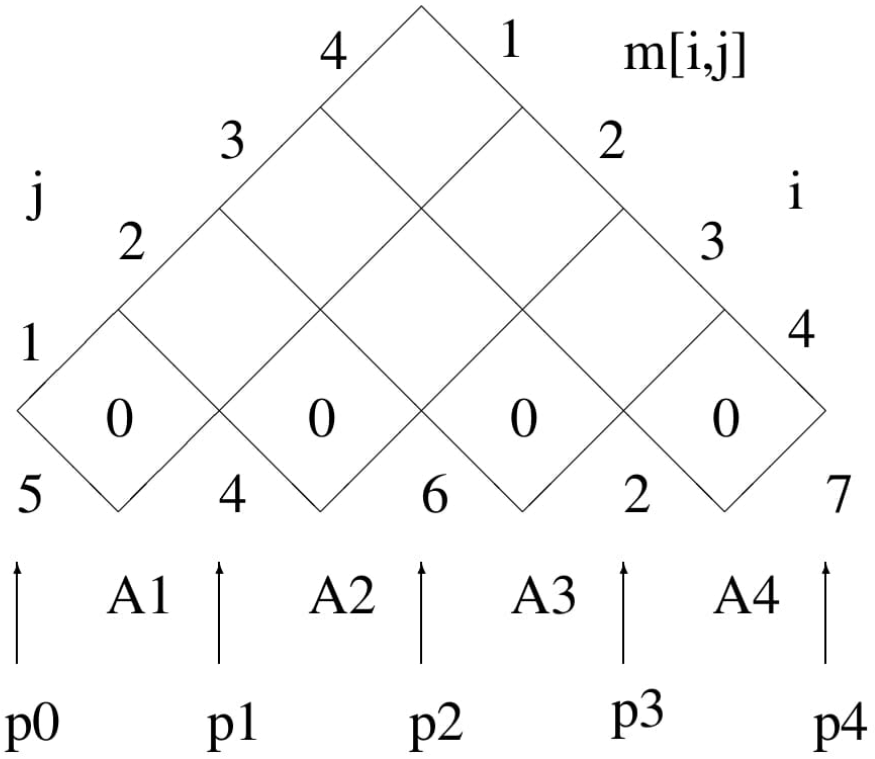
$m[1, n]$

Matrix-chain Multiplication Example by David Mount Professor in the Department of Computer Science and UMIACS.

Example for the Bottom-Up Computation

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

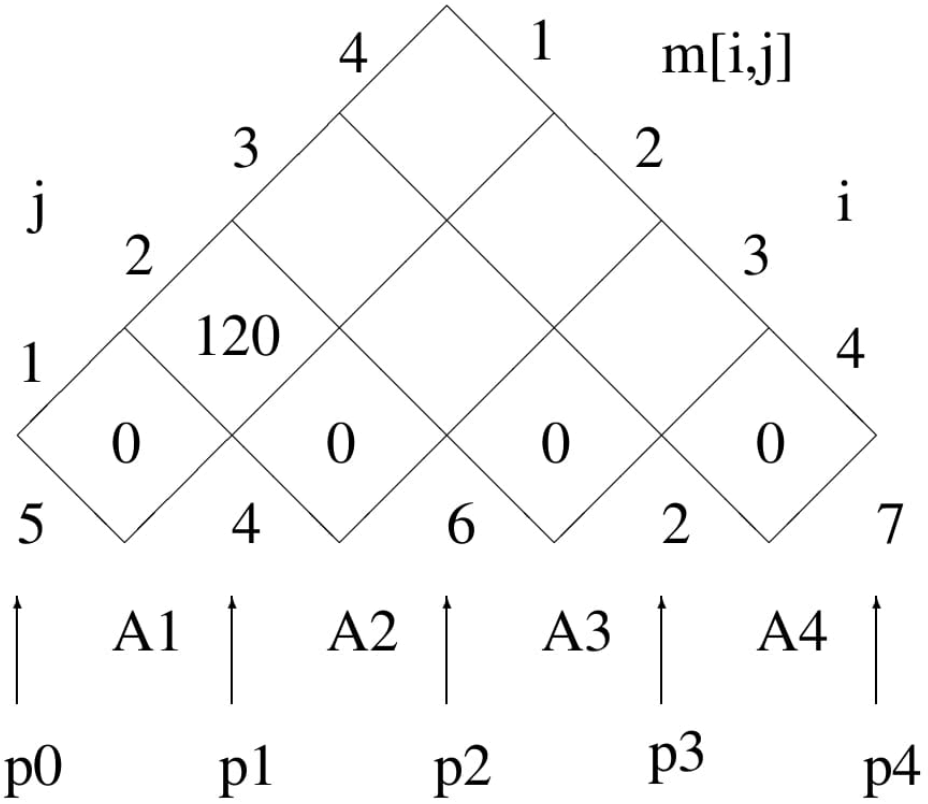
S0: Initialization



Example – Continued

Stp 1: Computing $m[1,2]$ By definition

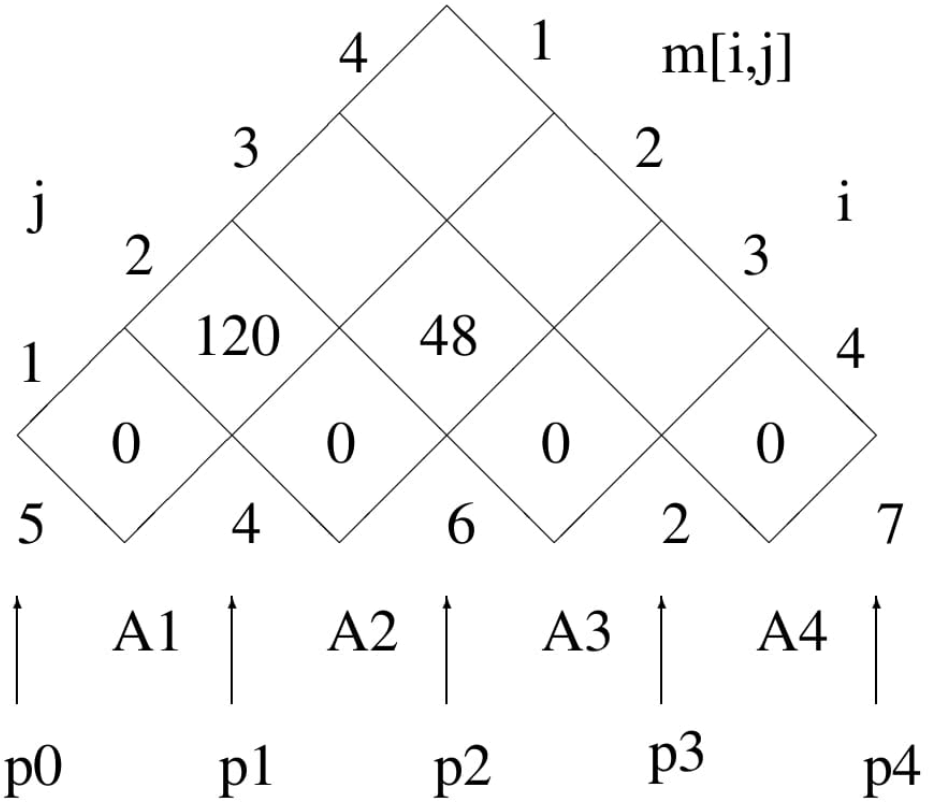
$$\begin{aligned} m[1,2] &= \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2) \\ &= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120. \end{aligned}$$



Example – Continued

Stp 2: Computing $m[2,3]$ By definition

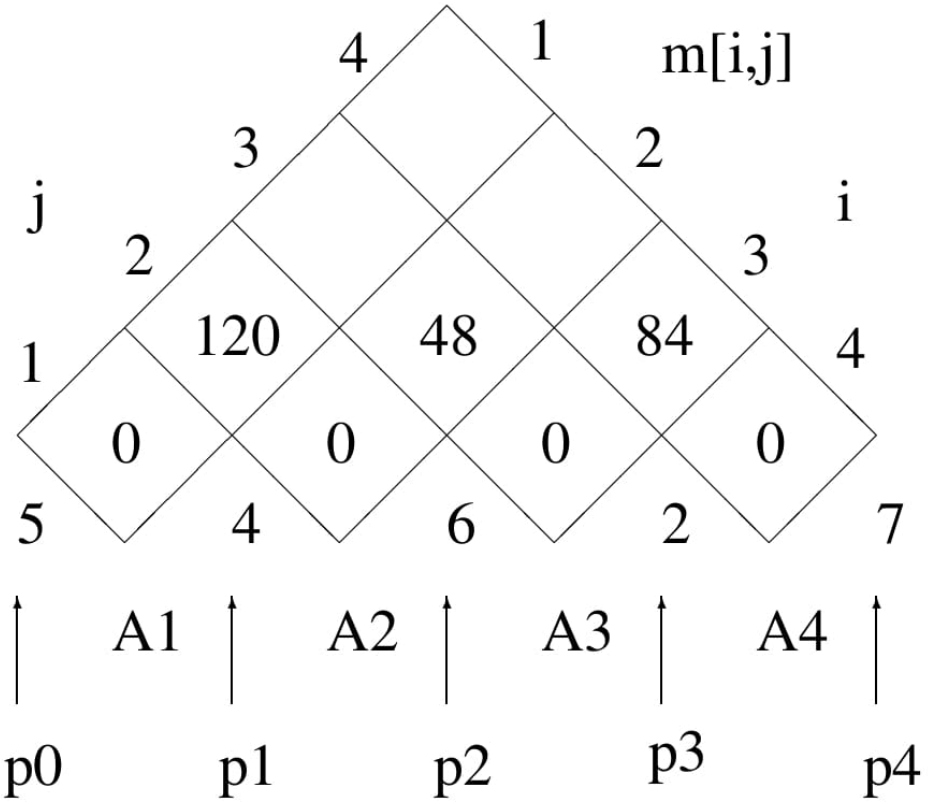
$$\begin{aligned} m[2,3] &= \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1 p_k p_3) \\ &= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48. \end{aligned}$$



Example – Continued

Stp3: Computing $m[3,4]$ By definition

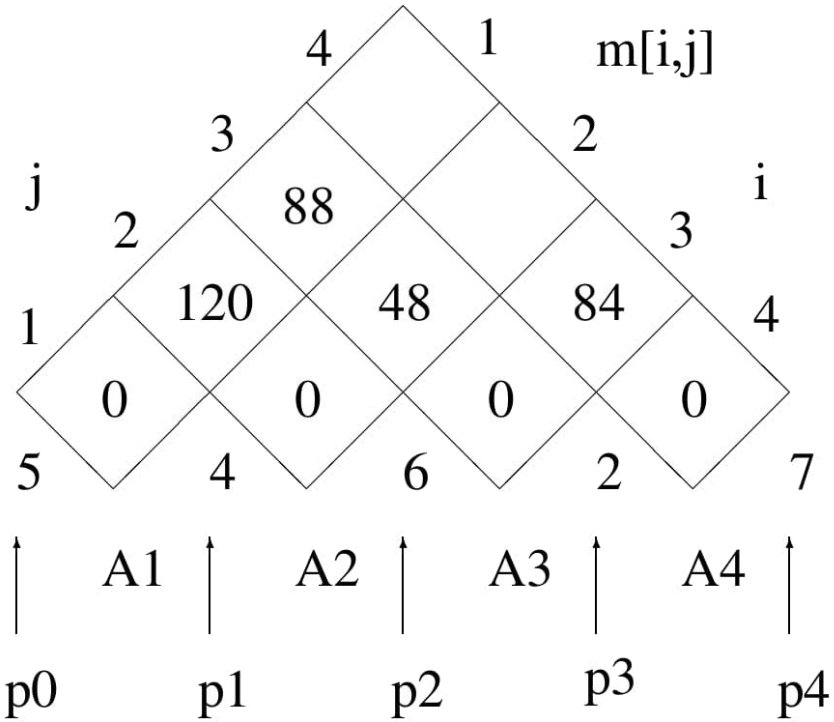
$$\begin{aligned} m[3,4] &= \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4) \\ &= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84. \end{aligned}$$



Example – Continued

Stp4: Computing $m[1,3]$ By definition

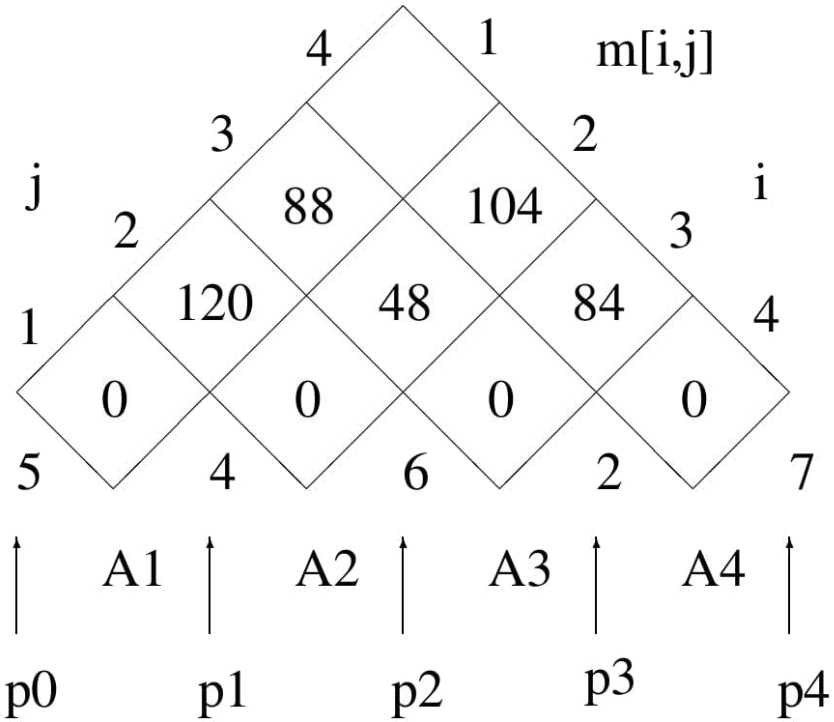
$$
\begin{aligned}
m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\
&= 88.
\end{aligned}
$$



Example – Continued

Stp5: Computing $m[2,4]$ By definition

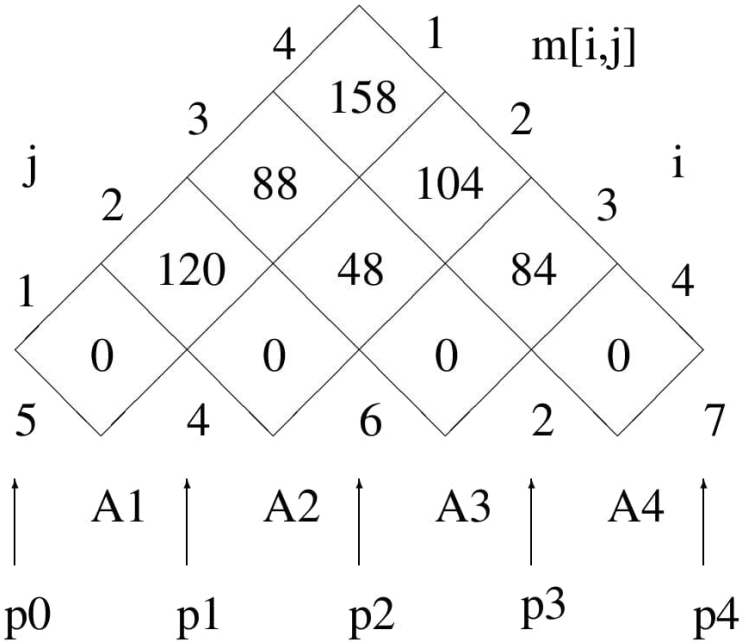
$$
\begin{aligned}
m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{array} \right\} \\
&= 104.
\end{aligned}
$$



Example – Continued

St6: Computing $m[1,4]$ By definition

$$
\begin{aligned}
m[1,4] &= \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + p_0 p_k p_4) \\
&= \min \left\{ \begin{array}{l} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{array} \right\} \\
&= 158.
\end{aligned}
$$



We are done!

Constructing an Optimal Solution: Compute $A_{1..n}$

Example of Constructing an Optimal Solution:

Compute $A_{1..6}$.

Consider the example earlier, where $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.

Mult$(A, s, 1, 6)$, $s[1, 6] = 3$, $(A_1A_2A_3)(A_4A_5A_6)$
Mult$(A, s, 1, 3)$, $s[1, 3] = 1$, $((A_1)(A_2A_3))(A_4A_5A_6)$
Mult$(A, s, 4, 6)$, $s[4, 6] = 5$, $((A_1)(A_2A_3))((A_4A_5)(A_6))$
Mult$(A, s, 2, 3)$, $s[2, 3] = 2$, $((A_1)((A_2)(A_3)))((A_4A_5)(A_6))$
Mult$(A, s, 4, 5)$, $s[4, 5] = 4$, $((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6))$

Hence the product is computed as follows

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

Summary

1. Identification of the optimal substructure property
2. Recursive formulation to compute the cost of the optimal solution
3. Bottom-up computation of the table entries
4. Constructing the optimal solution by backtracing the table entries

Elements of Dynamic Programming

- When should we look for a DP solution to an optimization problem?
- Two key ingredients for the problem
  - Optimal substructure
  - Overlapping subproblems

DP Hallmark #1

Optimal Substructure

- A problem exhibits optimal substructure
  - if an optimal solution to a problem contains within it optimal solutions to subproblems
- Example: matrix-chain-multiplication

  Optimal parenthesization of $A_1A_2\ldots A_n$ that splits the product between A_k and A_{k+1},

  contains within it optimal soln's to the problems of parenthesizing $A_1A_2\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_n$

Optimal Substructure

Finding a suitable space of subproblems

- Iterate on subproblem instances
- Example: matrix-chain-multiplication
  - Iterate and look at the structure of optimal solutions to subproblems, sub-subproblems, and so forth
  - Discover that all subproblems consists of subchains of $\langle A_1, A_2, \ldots, A_n \rangle$
  - Thus, the set of chains of the form

$$\langle A_i, A_{i+1}, \ldots, A_j \rangle \text{ for } 1 \le i \le j \le n$$

  - Makes a natural and reasonable space of subproblems

DP Hallmark #2

Overlapping Subproblems

- Total number of distinct subproblems should be polynomial in the input size
- When a recursive algorithm revisits the same problem over and over again we say that the optimization problem has overlapping subproblems

Overlapping Subproblems

- DP algorithms typically take advantage of overlapping subproblems
  - by solving each problem once
  - then storing the solutions in a table

    where it can be looked up when needed
  - using constant time per lookup

Overlapping Subproblems

Recursive matrix-chain order

RMC(p, i, j)

if $i = j$ **then**
  return 0

$m[i, j] \leftarrow \infty$

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

  $q \leftarrow \text{RMC}(p, i, k) + \text{RMC}(p, k+1, j) + p_{i-1} p_k p_j$

  if $q < m[i, j]$ **then**

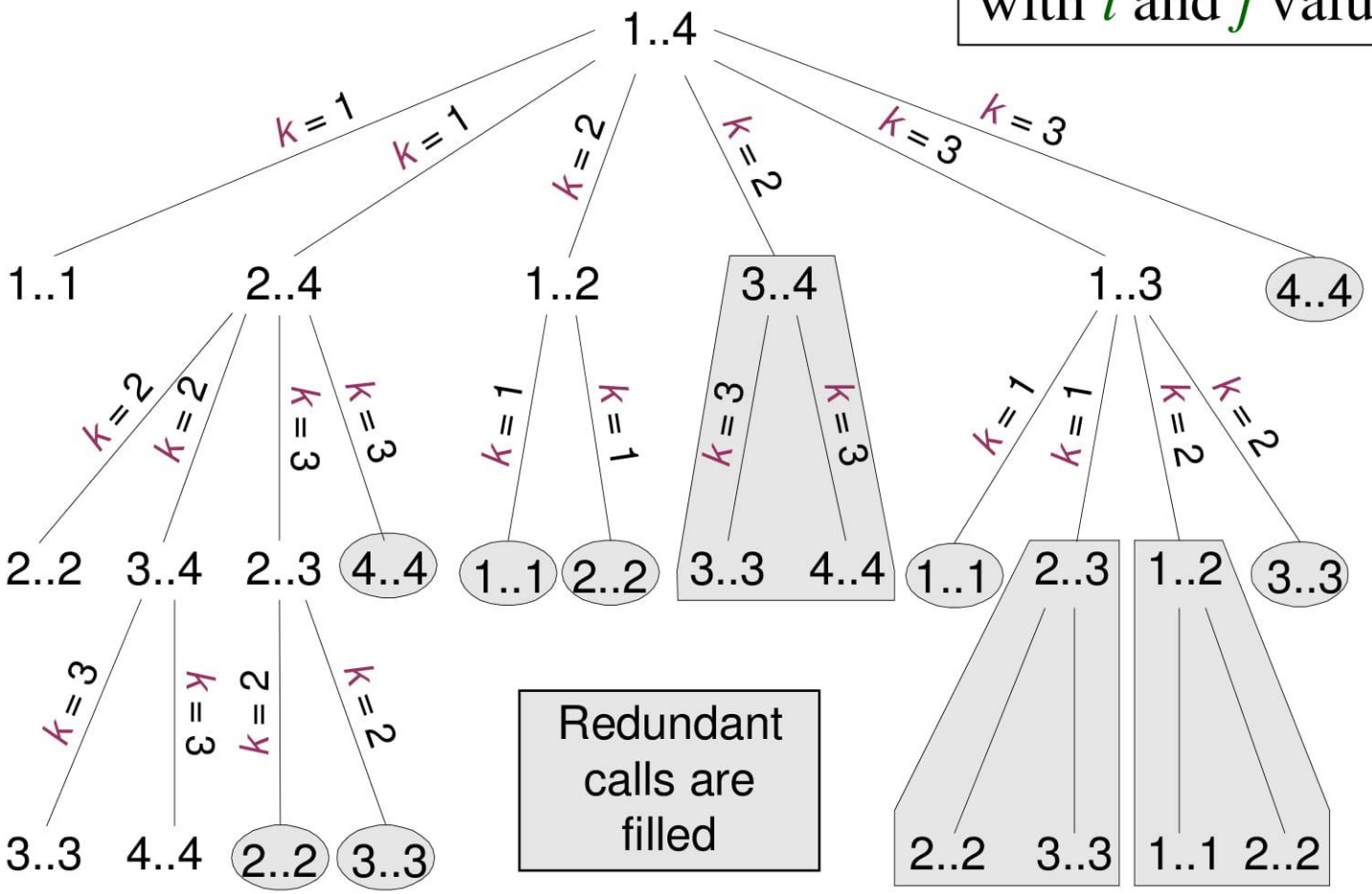
    $m[i, j] \leftarrow q$

return $m[i, j]$

Recursive Matrix-chain Order

Recursion tree for RMC(p,1,4)

Nodes are labeled with i and j values



Running Time of RMC

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1 \right) \text{ for } n > 1$$

- For $i = 1, 2, \ldots, n$ each term $T(i)$ appears twice
  - Once as $T(k)$, and once as $T(n-k)$
- Collect $n-1$ 1's in the summation together with the front 1

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

- Prove that $T(n) = \Omega(2^n)$ using the substitution method

Running Time of RMC: Prove that $T(n) = \Omega(2^n)$

- Try to show that $T(n) \geq 2^{n-1}$ (by substitution)

Base case: $T(1) \geq 1 = 2^0 = 2^{1-1}$ for $n = 1$

IH: $T(i) \geq 2^{i-1}$ for all $i = 1, 2, \ldots, n-1$ and $n \geq 2$

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n$$

$$= 2\sum_{i=0}^{n-2} 2^i + n = 2(2^{n-1} - 1) + n$$

$$= 2^{n-1} + (2^{n-1} - 2 + n)$$

$$\Rightarrow T(n) \geq 2^{n-1}$$ Q.E.D.

Running Time of RMC: $T(n) \geq 2^{n-1}$

Whenever

- a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly
- the total number of different subproblems is small

it is a good idea to see if DP can be applied

Memoization

- Offers the efficiency of the usual DP approach while maintaining top-down strategy
- Idea is to memoize the natural, but inefficient, recursive algorithm

Memoized Recursive Algorithm

- Maintains an entry in a table for the solution to each subproblem
- Each table entry contains a special value to indicate that the entry has yet to be filled in
- When the subproblem is first encountered its solution is computed and then stored in the table
- Each subsequent time that the subproblem encountered the value stored in the table is simply looked up and returned

Memoized Recursive Matrix-chain Order

```
MemoizedMatrixChain(p)
    n ← length[p] −1
    for i ←1 to n do
        for j ←1 to n do
            m[i, j] ← ∞
    return LookupC(p, 1, n)
```

```
LookupC(p, i, j)
    if m[i, j] = ∞ then
        if i = j then
            m[i, j] ← 0
        else
            for k ← i to j −1 do
                q ← LookupC(p, i, k) + LookupC(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]
```

▷Shaded subtrees are looked-up rather than recomputing

Memoized Recursive Algorithm

- The approach assumes that
  - The set of all possible subproblem parameters are known
  - The relation between the table positions and subproblems is established
- Another approach is to memoize
  - by using hashing with subproblem parameters as *key*

Dynamic Programming vs Memoization Summary

- Matrix-chain multiplication can be solved in $O(n^3)$ time
  - by either a top-down memoized recursive algorithm
  - or a bottom-up dynamic programming algorithm
- Both methods exploit the overlapping subproblems property
  - There are only $\Theta(n^2)$ different subproblems in total
  - Both methods compute the solution to each problem once
- Without memoization the natural recursive algorithm runs in exponential time since subproblems are solved repeatedly

Dynamic Programming vs Memoization Summary

In general practice

- If all subproblems must be solved at once
  - a bottom-up DP algorithm always outperforms a top-down memoized algorithm by a constant factor

  because, bottom-up DP algorithm
    - Has no overhead for recursion
    - Less overhead for maintaining the table
- DP: Regular pattern of table accesses can be exploited to reduce the time and/or space requirements even further
- Memoized: If some problems need not be solved at all, it has the advantage of avoiding solutions to those subproblems

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

Source: 2004 Goodrich, Tamassia

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 xi=yj, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If xi≠yj, 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,…,n-1, j = 0,...,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = $x_0x_1x_2\ldots x_i$ and the string Y [0.. j] = $y_0y_1y_2\ldots y_j$

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

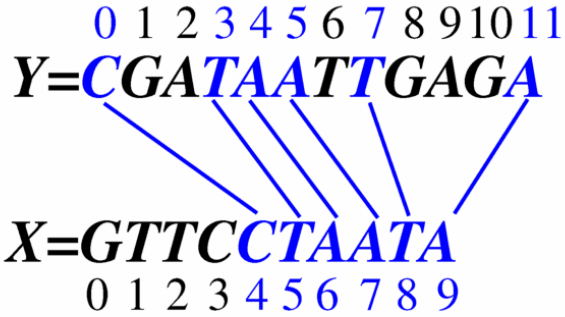
        else

            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**



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).