Algorithms Design and Analysis

Lecture 4

Beihang University

2017

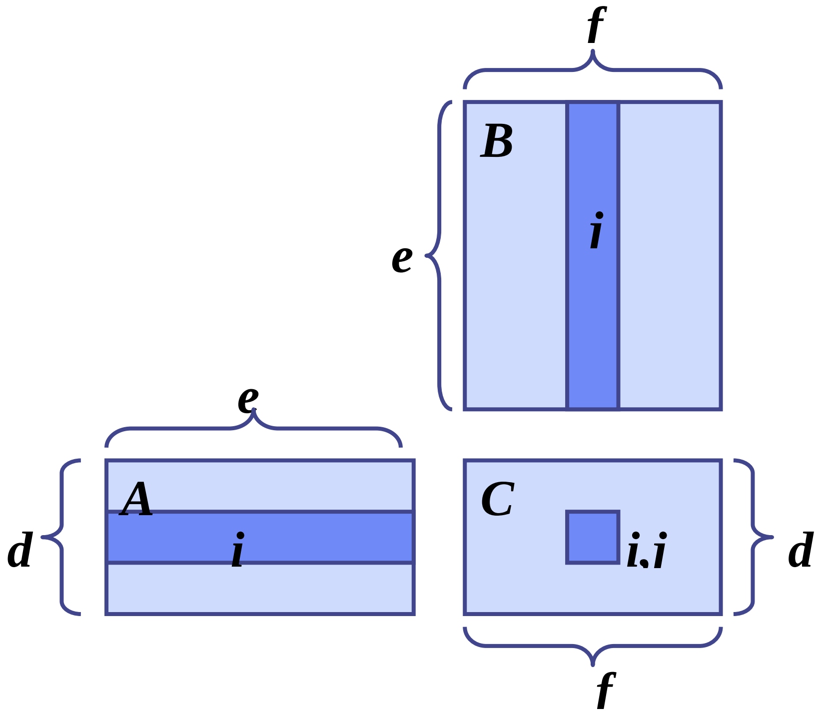
Dynamic Programming (Continue)

Example 2. Matrix Chain-Products (乘积)

Review: Matrix Multiplication

- C=A×B
- A is $d \times e$ and B is $e \times f$
- $C[i, j]=\sum_{k=0}^{e-1} A[i, k]\times B[k, j]$
- C is $d \times f$
- $O(d \times e \times f)$ time
- Matrix multiplication is associative (结合的)：

  (A×B)×C=A×(B×C)



The matrix multiplication order problem

- What is the best order to compute $A_1 \times A_2 \times \ldots \times A_n$ for $n>2$
  -where matrix A_i has cardinality (基数) $d_{i-1} \times d_i$
- Consider matrices A_1:30×1, A_2:1×40, A_3:40×10, A_4:10×25

Multiplication order	1st product		2nd product		3rd product		# Mults
$((A_1A_2)A_3)A_4$	30×1×40	+	30×40×10	+	30×10×25	=	**20700**
$A_1(A_2(A_3A_4))$	40×10×25	+	1×40×25	+	30×1×25	=	**11750**
$(A_1A_2)(A_3A_4)$	30×1×40	+	40×10×25	+	30×40×25	=	**41200**
$A_1((A_2A_3)A_4)$	1×40×10	+	1×10×25	+	30×1×25	=	**1400**

- What is the minimum number of multiplications? What is the order?

An Enumeration (列举) Approach

- Try all possible ways to parenthesize (加括号)

  ◊$(A_1 \times A_2 \times \cdots A_k)(A_{k+1} \times A_{k+2} \times \cdots A_n)$

  k matrices *n*-*k* matrices

- Calculate number of operations for each one
- Pick the one that is best

Let $P(n)$ be the number of different parenthesizations of n matrices.

$$P(n)=P(k)P(n\text{-}k) \; \textbf{?}$$

Recursive equation for the number of parenthesizations is:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n > 1 \end{cases}$$

Complexity analysis

- By induction, we can prove $P(n) = \binom{2n-2}{n-1} \Big/ n$.
- This is **exponential**!
- It is called the Catalan number, and is almost 4^n when n is large.
- This is a terrible (可怕的) algorithm!

The structure of an optimal parenthesization

- Notation (记号): $A_{i..j}$ = result from computing $A_iA_{i+1}\ldots A_j$

  - ✓ Any parenthesization of $A_iA_{i+1}\ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$, i.e. $(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$.
  - ✓ Cost = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ together

- Suppose that an optimal parenthesization of $A_iA_{i+1}\ldots A_j$, called P, splits the product between A_k and A_{k+1}.
  - ✓The parenthesization of the sub-chain $A_iA_{i+1}\ldots A_k$ given by P must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$
  - ✓The parenthesization of the sub-chain $A_{k+1}A_{k+2}\ldots A_j$ given by P must be an optimal parenthesization of $A_{k+1}A_{k+2}\ldots A_j$

Minimal

Cost_$A_{1..6}$ + Cost_$A_{7..9}$+$d_0d_6d_9$

$A_1A_2A_3A_4A_5A_6A_7A_8A_9$

Suppose $((A_1A_2)(A_3((A_4A_5)A_6)))$ $((A_7A_8)A_9)$ is optimal

Then (A_1A_2) $(A_3((A_4A_5)A_6))$ must be optimal for $A_1A_2A_3A_4A_5A_6$

Otherwise, if $(A_1(A_2A_3))$ $((A_4A_5)A_6)$ is optimal for $A_1A_2A_3A_4A_5A_6$

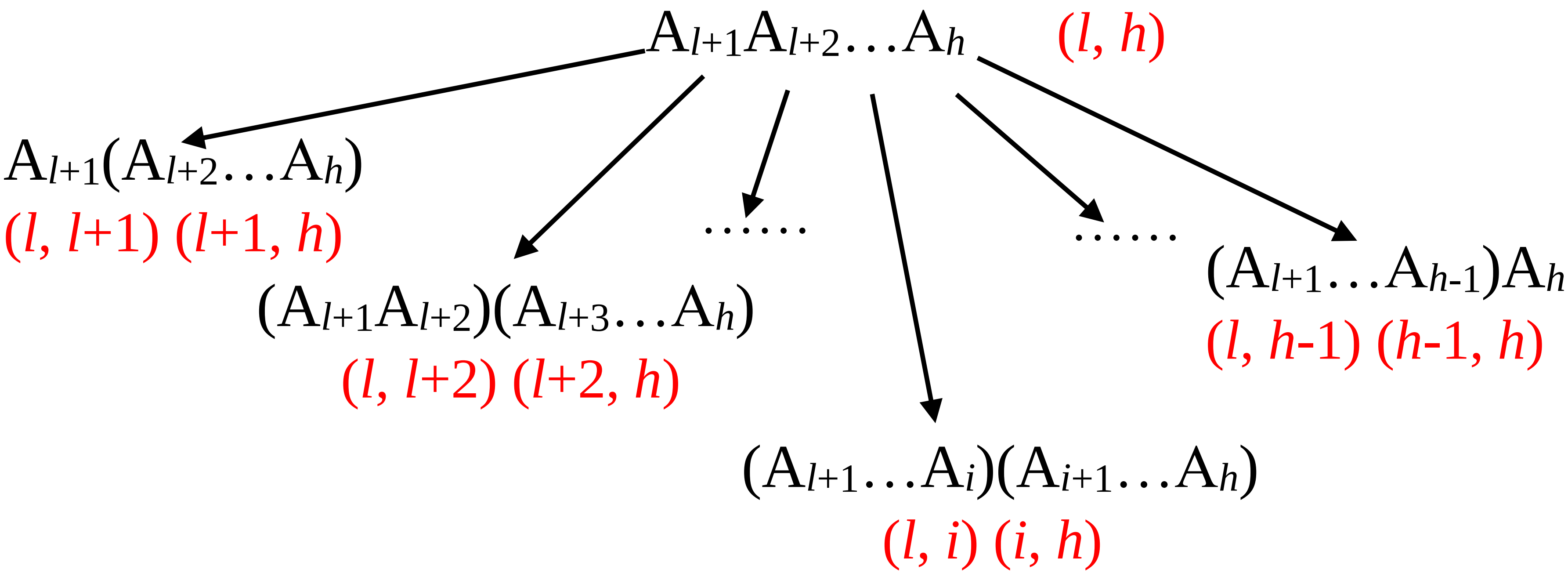
Then $((A_1(A_2A_3))((A_4A_5)A_6))$ will be better than

$((A_1A_2)(A_3((A_4A_5)A_6)))$

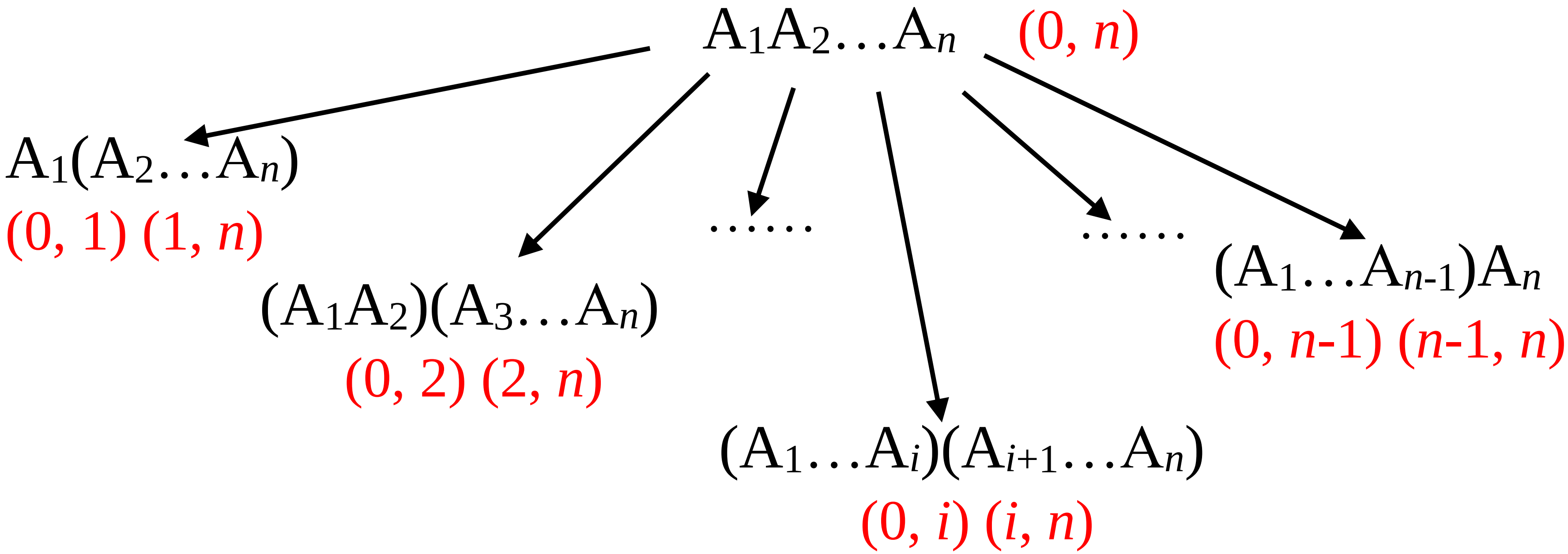
Characterize (刻画) the structure of an optimal solution

- Denote (表示) the problem $A_{l+1} \times A_{l+2} \times \ldots \times A_h$ as (l, h), i.e. $(d_l, d_{l+1}) \times (d_{l+1}, d_{l+2}) \times \ldots \times (d_{h-1}, d_h)$.
  - ✓ After choice of the last multiplication at i where $l<i<h$, i.e. $(A_{l+1} \times A_{l+2} \times \ldots \times A_i) \times (A_{i+1} \times \ldots \times A_h)$, the remaining subproblems are (l, i) and (i, h).

- Let $m[l, h]$ be the optimal cost of computing the problem (l, h). To determine it we need to check all possibilities for i.



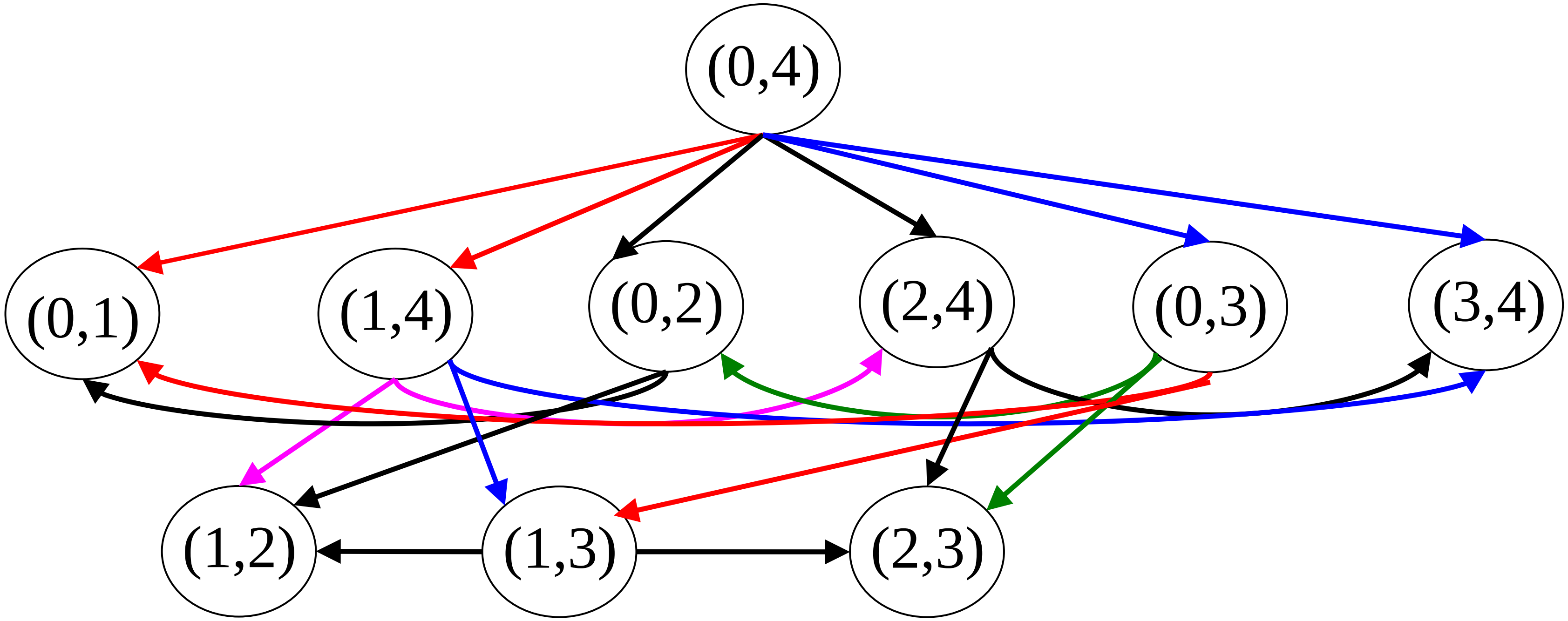
$$
\begin{aligned}
&m[l, h] \\
&= 0 && \text{if } l+1=h \\
&= \min_{l<i<h}(m[l, i]+m[i, h]+d_l d_i d_h) && \text{if } l+1<h
\end{aligned}
$$



Note: there are at least two subproblems of size n-1, i.e. (1, n) and (0, n-1).

Subproblem graph – recursive algorithm

The subproblem graph of $A_1 \times A_2 \times A_3 \times A_4$, i.e. subproblem (0,4):



- For problem (0, n), this creates $\Theta(n^2)$ vertices (顶点) in the subproblem graph

- Encounter (遭遇) each sub-problem many times in different branches of its recursion tree $\Rightarrow$ overlapping sub-problems

- A recursive algorithm takes exponential time.
  ✓Let T(n) denote the number of recursive calls for computing (0, n). Then T(2) = 2 and $T(n) \geq 2T(n-1)$. So $T(n) \geq 2^{n-1}$.

Reverse graphical order – dynamic programming

The dependency graph of $A_1 \times A_2 \times A_3 \times A_4$, i.e. subproblem (0,4):

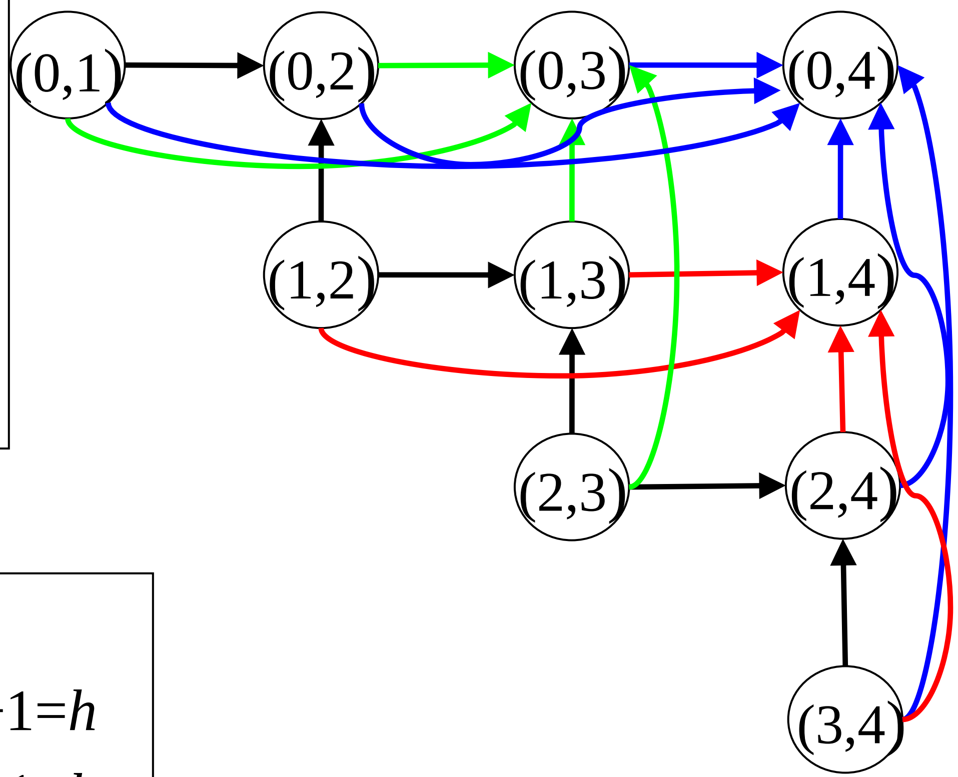
$m[2, 4] = \min_{2<i<4}\{$ // A_3A_4

$m[2, 3]+m[3, 4]+d_2d_3d_4\}$

$m[1, 4]=\min_{1<i<4}\{$ // $A_2A_3A_4$

$m[1, 2]+m[2, 4]+d_1d_2d_4;$ // $A_2(A_3A_4)$

$m[1, 3]+m[3, 4]+d_1d_3d_4\}$ // $(A_2A_3)A_4$

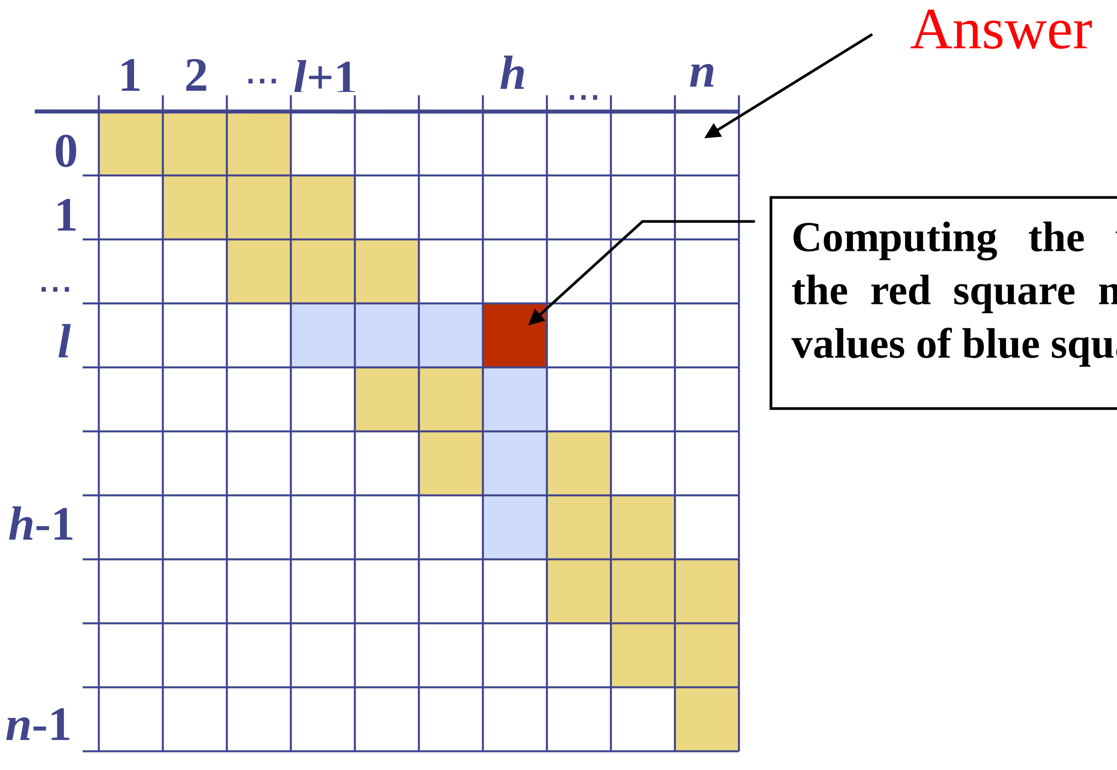


$m[l, h]$

$= 0$ if $l+1=h$

$= \min_{l<i<h}(m[l, i]+m[i, h]+d_l d_i d_h)$ if $l+1<h$

$$m[l, h] = \min_{l<i<h}(m[l, i] + m[i, h] + d[l]d[i]d[h])$$



Computing the value of the red square needs the values of blue squares.

Dynamic Programming Algorithm

MatrixChain(d, *n*)

for (l=n-1; l>0; l––) { // iterate (迭代) over rows (行)

  for (h=l+1; $h \le n$; h++) { // iterate over the colums (列)

  if (h–l=1) bestcost =0; **else** bestcost = ∞;

  for (i=l+1; i<h; i++) { // consider l<i<h

    a = m[l, i]; // look-up solution

    b = m[i, h]; // look-up solution

    c = d[l]d[i]d[h]; // cost of last multiplicationat position i

    bestcost=min(bestcost, a+b+c); }

   m[l, h] = bestcost; } // store obtained result

}

return m[0, n];

Complexity analysis: $O(n \times n \times n) = O(n^3)$.

$0+0+1\times40\times10=400$

$0+0+40\times10\times25=10000$

$\min_{1<i<4}\{$
$0+10000+1\times40\times25=11000,$
$400+0+1\times10\times25=650\}$
$=650$

$(A_2A_3)A_4$

(0,1) (0,2) (0,3) (0,4)

0 (1,3) (1,4)

0 (2,4)

0

$d_0=30,\ d_1=1,\ d_2=40,\ d_3=10,\ d_4=25$

$$m[l, h] = \begin{cases} 0 & \text{if } l+1=h \\ \min_{l<i<h}(m[l, i]+m[i, h]+d_l d_i d_h) & \text{if } l+1<h \end{cases}$$

$0+0+30\times1\times40=1200$

$\min_{0<i<3}\{$
$0+400+30\times1\times10=\color{red}{700}$,
$1200+0+30\times40\times10=13200\}$
$=700$

0	(0,2)	(0,3)	(0,4)
	0	**400**	**650**
		0	**10000**
			0

$d_0=30,\ d_1=1,\ d_2=40,\ d_3=10,\ d_4=25$

$m[l, h]$
$= 0 \qquad$ if $l+1=h$
$= \min_{l<i<h}(m[l, i]+m[i, h]+d_l d_i d_h) \qquad$ if $l+1<h$

$\min_{0<i<4}\{$

$0+650+30\times1\times25=$1400,

$1200+10000+30\times40\times25=32000,$

$700+0+30\times10\times25=8200\}$

$=1400$

$A_1((A_2A_3)A_4)$

0	**1200**	**700**	(0,4)
	0	**400**	**650**
		0	**10000**
			0

$d_0=30,\ d_1=1,\ d_2=40,\ d_3=10,\ d_4=25$

$$m[l, h] = \begin{cases} 0 & \text{if } l+1=h \\ \min_{l<i<h}(m[l, i]+m[i, h]+d_l d_i d_h) & \text{if } l+1<h \end{cases}$$

Example 3. Longest Common (共同的) Subsequence (子序列) (LCS)

- A subsequence of a sequence *S* is obtained by deleting zero or more symbols from *S*. For example, the following are all subsequences of "president": pred, sdn, predent.

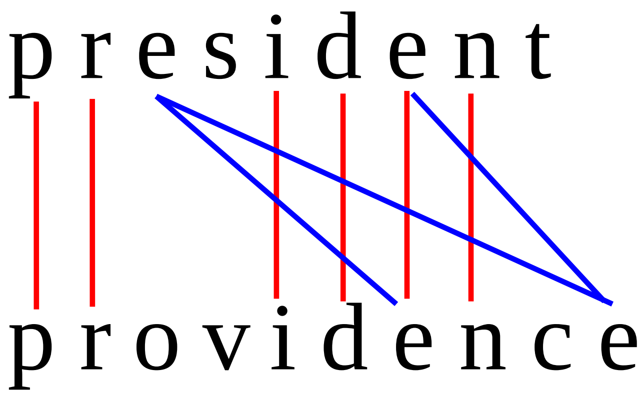
- The longest common subsequence problem is to find a maximum-length common subsequence between two sequences.

For instance,

Sequence 1: president

Sequence 2: providence (天意)

Its LCS is priden

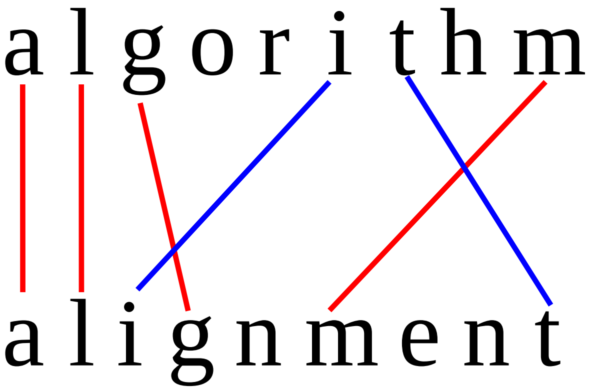


Another example:

Sequence 1: algorithm

Sequence 2: alignment (队列)

One of its LCS is algm.



Application (应用): comparison of two molecular sequence

Molecular (分子) sequence data are at the heart of Computational Biology (计算生物学)

- DNA sequences
- RNA sequences
- Protein (蛋白质) sequences

We can think of these sequences as strings of letters

- DNA & RNA: alphabet of 4 letters (A,T,C,G) (A,U,C,G)
- Protein: alphabet of 20 letters

Two DNA sequences

Sequence X = **A T C T G A T**

Sequence Y = **T G C A T A**

Similarity Degree (相似程度)

We can define a parameter to determine the similarity degree between two sequences as follows:

$$S(\mathrm{X}, \mathrm{Y})=2|\mathrm{LCS}(\mathrm{X}, \mathrm{Y})|/(|\mathrm{X}|+|\mathrm{Y}|).$$

For example, for the above two sequences, we have

$$S(\mathrm{X}, \mathrm{Y})=2\times 4|/(7+6|)\approx 0.6.$$

An enumeration approach

Enumeration algorithm will compare each subsequence of X with the symbols in Y

- If |X| = m, |Y| = n, then there are 2^m subsequences of X; we must compare each with Y (n comparisons)
- So the running time of the enumeration algorithm is $O(n2^m)$

For example, given two sequences X = **ATC** and Y = **TCAG**, the subsequences of X include **ATC**, **AT**, **AC**, **TC**, **A**, **T**, **C** and the empty string.

Basic Definitions

- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively, where $|X| = m$, $|Y| = n$. For example, X=abcbd, $|X|=5$, X_2=ab, X_3=abc.
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

How to compute LCS

- When computing $c[i,j]$, we consider the following two cases:
- **First case:** Let $X_i = a_1\ a_2\ \ldots\ a_{i-1}\ a_i$ and $Y_j = b_1\ b_2\ \ldots\ b_{j-1}\ b_j$. If $a_i=b_j$, then the length of LCS of X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1}, plus 1. That is to say, we have $c[i,j]= c[i-1,j-1]+1$. For example, given X_3=abc, Y_2=ac, we have $c[3,2]=2$, X_2=ab, Y_1=a and $c[2,1]=1$.

- **Second case:** If $a_i \neq b_j$, then the length of LCS(X_i, Y_j) is the maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j). That is to say, we have $c[i,j]= \max\{c[i-1,j], c[i,j-1]\}$.

Why not take the length of LCS(X_{i-1}, Y_{j-1})?

- For example, X_3=abc, Y_2=ab. We have X_2=ab, Y_1=a, $c[3,2]=2$ and $c[2,1]=1$.

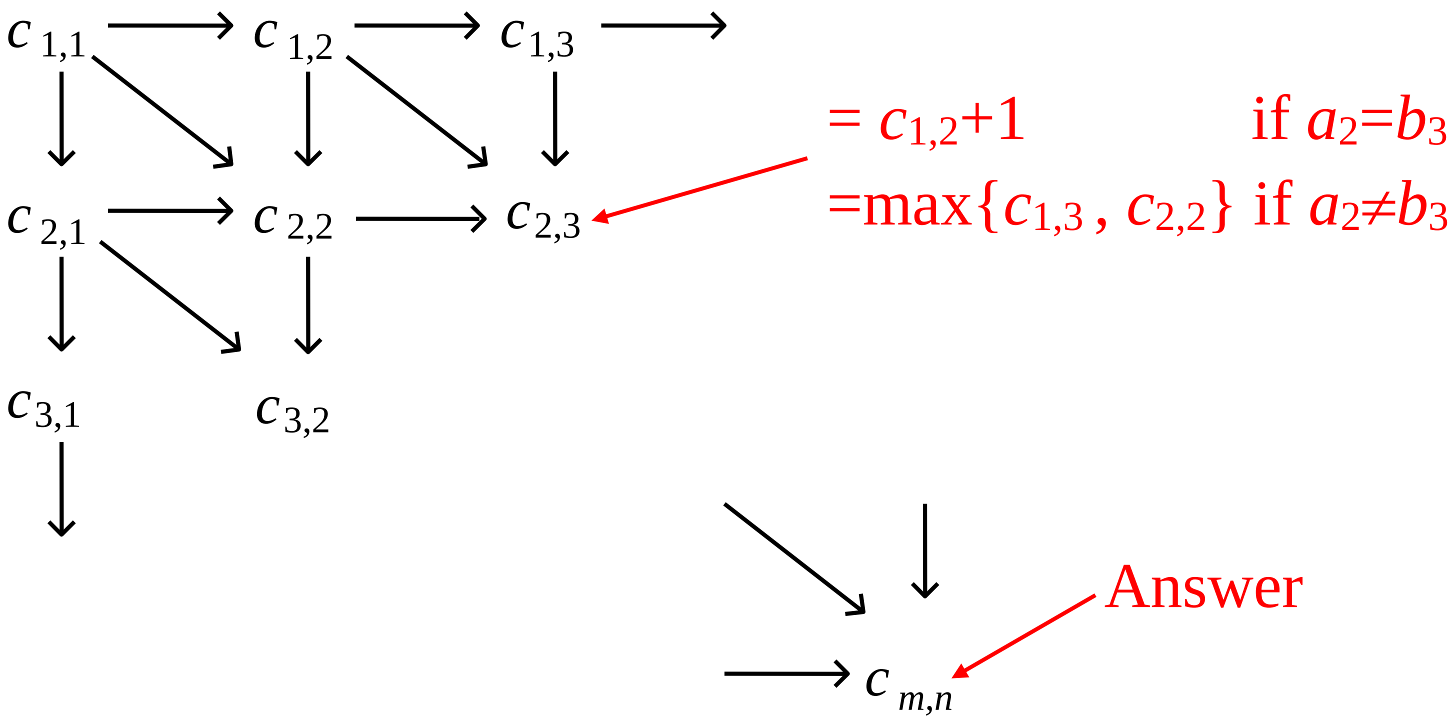
Combining the above two cases, we have

$$
\begin{aligned}
c[i,j] &= c[i-1,j-1]+1 && \text{if } a_i = b_j \\
&= \max\{c[i-1,j], c[i,j-1]\} && \text{if } a_i \neq b_j
\end{aligned}
$$

Base cases

- We start with $i = j = 0$ (empty substrings of X and Y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$.

The dynamic programming approach



LCS-Length(X, Y)

1. m = length(X) // get the number of symbols in X
2. n = length(Y) // get the number of symbols in Y
3. for $i = 1$ to m c[i, 0] = 0 // base case: Y_0
4. for $j = 1$ to n c[0, j] = 0 // base case: X_0
5. for $i = 1$ to m // Iterated over rows
6.   for $j = 1$ to n // Iterated over columns
7.   if ($x_i = y_j$)
8. $c[i, j] = c[i\text{-}1, j\text{-}1] + 1$
9.   else $c[i, j] = \max(c[i\text{-}1, j], c[i, j\text{-}1])$
10. return $c[m, n]$

Complexity analysis: $O(mn)$.

LCS Example

X = b a c a d, Y = a c c b a d c b

			Y							
			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	1	1	1	2	2	2	2
X	c	0	1	2	2	2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	4	4

After all elements have been found, we can trace (追溯) back to find the longest common subsequence of X and Y.

Homework 7

对于矩阵链乘积(Matrix Chain-Products)问题，请问是否存在其它的子问题计算次序，使得可以保证在计算到每一个子问题时，其所需要的子问题的解已经存在。如果有，请给出这样的计算次序。

Homework 8

用动态规划方法求解背包问题。