



Theory of algorithms (8th lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- Dynamic programming
 - General knowledge
 - Developing a dynamic-programming algorithm
- Examples
 - Matrix-chain multiplication
 - Longest common subsequence
- Exercises



Dynamic programming

- The **divide-and-conquer** algorithms partition the problem into **disjoint** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- **Dynamic programming**
 - Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. In contrast, dynamic programming applies when the subproblems **overlap**, that is, when subproblems share subsubproblems.
 - It solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.
 - It is typically applied to solve **optimization problems**. Such problems usually have many possible solutions and each solution has a value.
 - The goal is to find a solution with the optimal (minimum or maximum) value. Such a solution is called *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.



Dynamic programming

Developing a dynamic-programming algorithm typically follows a sequence of four steps:

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.



Matrix-chain multiplication

MATRIX-MULTIPLY(A, B)

```
1  if  $A.columns \neq B.rows$ 
2      error „incompatible dimensions”
3  else
4      let  $C$  be a new  $A.rows \times B.columns$  matrix
5      for  $i = 1$  to  $A.rows$ 
6          for  $j = 1$  to  $B.columns$ 
7               $c_{i,j} = 0$ 
8              for  $k = 1$  to  $A.columns$ 
9                   $c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}$ 
10     return  $C$ 
```

Remarks: Two matrices A and B can be multiplied only if they are **compatible**: the number of columns of A must equal the number of rows of B . If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix.

Efficiency: The time to compute C is dominated by the number of scalar multiplications in line 9, which is pqr . The costs will be expressed of the number of scalar multiplications.



Matrix-chain multiplication

It is given a sequence (chain) A_1, A_2, \dots, A_n of n matrices to be multiplied and their product $A_1 A_2 \dots A_n$ has to be compute.

Matrix multiplication is **associative**, and so all parenthesizations yield the same product.

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 parentheses.

Example: If the sequence is A_1, A_2, A_3, A_4 , then $A_1 A_2 A_3 A_4$ product can be fully parenthesized in five distinct ways:

$$(A_1 (A_2 (A_3 A_4))),$$

$$(A_1 ((A_2 A_3) A_4)),$$

$$((A_1 (A_2 A_3)) A_4),$$

$$((A_1 A_2)(A_3 A_4)),$$

$$(((A_1 A_2) A_3) A_4).$$

Example: Let $n=3$ and the dimensions of the matrices A_1, A_2, A_3 are 10×100 , 100×5 and 5×50 , respectively.

$((A_1 A_2) A_3)$ parenthesization: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$ scalar multiplications.

$(A_1 (A_2 A_3))$ parenthesization: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$ scalar multiplications.

Matrix-chain multiplication

The **matrix-chain multiplication problem** as follows: Given a chain A_1, A_2, \dots, A_n of n matrices, where for $i=1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \dots A_n$ in a way that minimizes the number of scalar multiplications.

Remark: The number of solutions is exponential in n .

The steps of the solution by dynamic-programming:

1. Characterize the structure of an optimal solution.
 - Let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product of $A_i A_{i+1} \dots A_j$. The optimal parenthesizing split $A_i A_{i+1} \dots A_j$ product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.
 - The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.
 - Note that $A_{i..k}$ and $A_{k+1..j}$ subproblems must have an optimal parenthesization as well, thus an optimal solution to the problem can be constructed from optimal solutions to subproblems.



Matrix-chain multiplication

2. Recursively define the value of an optimal solution:
 - The subproblem: determining the minimum cost of parenthesizing $A_i A_{i+1} \dots A_j$ for $1 \leq i \leq j \leq n$.
 - Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, for the full problem, the lowest cost way to compute $A_{1..n}$ would thus be $m[1, n]$.
 - If $i=j$, the problem is trivial, the chain consists of just one matrix $A_{i..i}=A_i$, so that no scalar multiplications are necessary to compute the product.
 - If $i < j$, let us assume that the optimal parenthesization split the product $A_i A_{i+1} \dots A_j$ between A_k and A_{k+1} , where $i \leq k < j$. The dimension of each matrix A_i is $p_{i-1} \times p_i$, the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications.

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

- There are only $j-i$ possible values for k , namely $k=i, i+1, i+2, \dots, j-1$. Since the optimal parenthesization must use one of these values for k , we need only check them all to find the best:

$$m[i, j] = 0, \quad \text{if } i=j,$$

$$m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}, \quad \text{if } i < j.$$

Remarks: To construct an optimal solution (step 4) we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \dots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ is satisfied.



Matrix-chain multiplication

3. Computing the optimal costs.

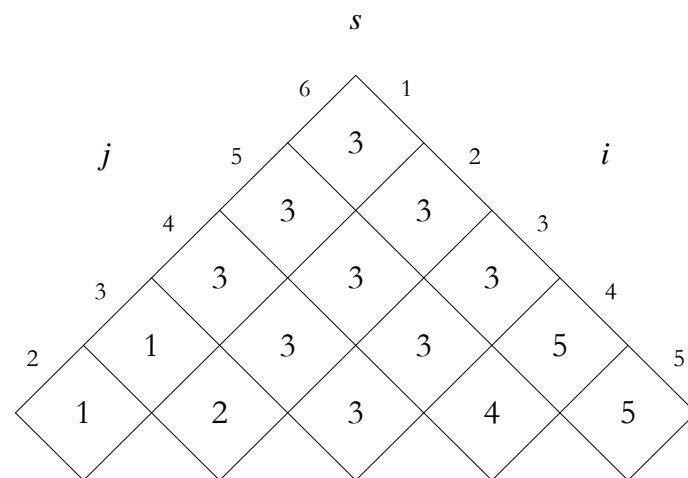
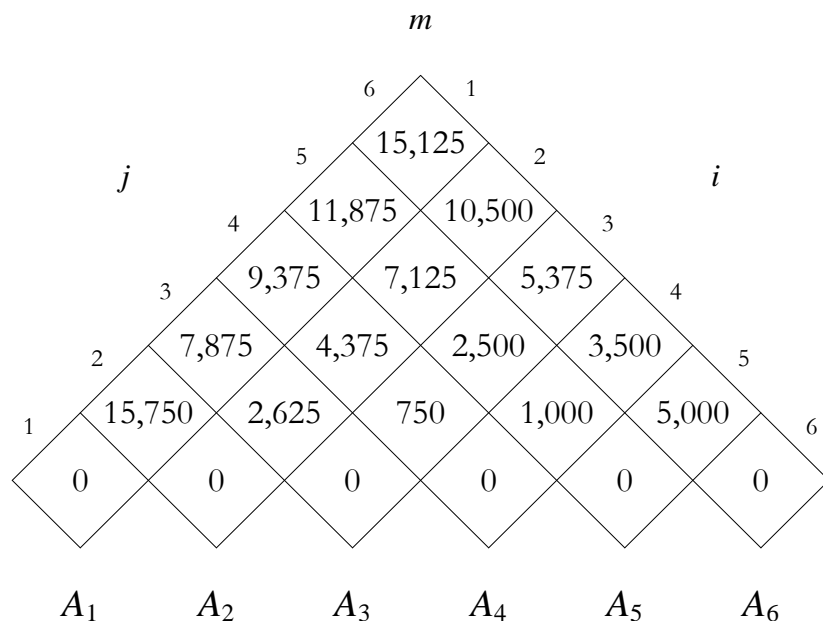
MATRIX-CHAIN-ORDER(p)

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14 return  $m$  and  $s$ 
```

Efficiency: The algorithm runs in $O(n^3)$ time and it requires $\Theta(n^2)$ space to store m and s tables.



Matrix-chain multiplication



$A_1 : 30 \times 35$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

$$m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000,$$

$$m[2, 5] = \min \{ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \dots \}$$

$$m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$$

The operation of MATRIX-CHAIN-ORDER ($n=6$)

Matrix-chain multiplication

4. Constructing an optimal solution.

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

```
1  if  $j == i$ 
2      write " $A$ ";
3  else
4      write "("
5      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
6      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
7      write ")"
```

Remarks:

- The initial call PRINT-OPTIMAL-PARENS($s, 1, n$).
- With the matrices in the example the call PRINT-OPTIMAL-PARENS($s, 1, 6$) prints the parenthesization $((A_1(A_2 A_3))((A_4 A_5) A_6))$.



Exercises

- How many scalar multiplications do we need to compute the product matrix, where the sequence of dimensions is $(5, 2, 3, 4)$. Give the value of the best case and the value of the worst case.
- Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $(5, 2, 3, 4, 2)$.
- Give m and s tables that MATRIX-CHAIN-ORDER(p) computes with $p=(5, 2, 3, 5, 4, 2)$.
- Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices A_1, A_2, \dots, A_n , the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j . The initial call would be MATRIX-CHAIN-MULTIPLY($A, s, 1, n$).



Matrix-chain multiplication

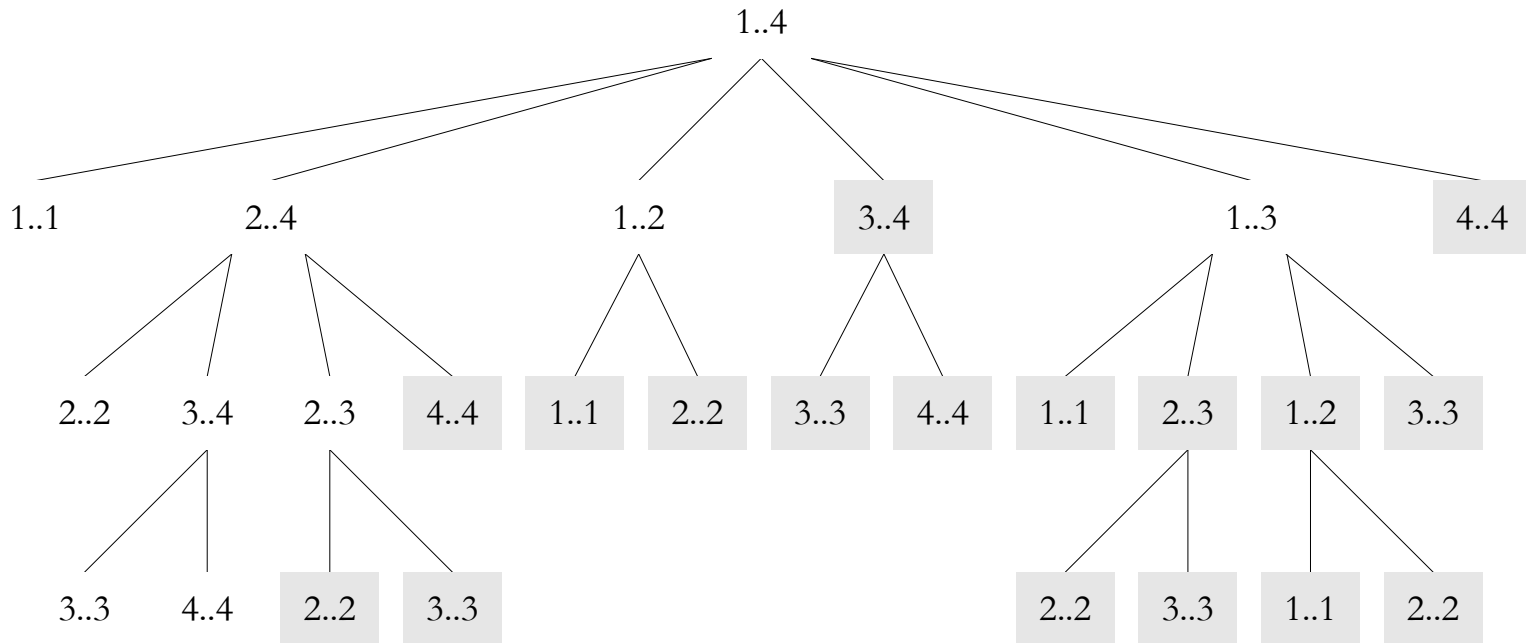
RECURSIVE-MATRIX-CHAIN(p, i, j)

```
1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j-1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k) +$ 
           $\text{RECURSIVE-MATRIX-CHAIN}(p, k+1, j) + p_{i-1} p_k p_j$ 
6      if  $q < m[i, j]$ 
7           $m[i, j] = q$ 
8  return  $m[i, j]$ 
```

Efficiency: The time to compute $m[1, n]$ by this recursive function is at least exponential in n .



Matrix-chain multiplication



The recursion tree for the computation of **RECURSIVE-MATRIX-CHAIN**(p , 1, 4)

Matrix-chain multiplication

MEMOIZED-MATRIX-CHAIN(p)

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  be a new table
2  for  $i = 1$  to  $n$ 
3      for  $j = i$  to  $n$ 
4           $m[i, j] = \infty$ 
5  return LOOKUP-CHAIN( $m, p, 1, n$ )

```

LOOKUP-CHAIN(m, p, i, j)

```

1  if  $m[i, j] < \infty$ 
2      return  $m[i, j]$ 
3  if  $i == j$ 
4       $m[i, j] = 0$ 
5  else
6      for  $k = i$  to  $j - 1$ 
7           $q = \text{LOOKUP-CHAIN}(p, i, k) + \text{LOOKUP-CHAIN}(p, k + 1, j) + p_{i-1} p_k p_j$ 
8          if  $q < m[i, j]$ 
9               $m[i, j] = q$ 
10 return  $m[i, j]$ 

```

Efficiency: Like the bottom-up dynamic-programming algorithm MATRIX-CHAIN-ORDER the procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time and it requires $\Theta(n^2)$ memory space.



Longest common subsequence

A subsequence of a given sequence is just the given sequence with zero or more elements left out.

Formally, given a sequence $X=(x_1, x_2, \dots, x_m)$, another sequence $Z=(z_1, z_2, \dots, z_k)$ is a **subsequence** of X if there exists a strictly increasing sequence (i_1, i_2, \dots, i_k) of indices of X such that for all $j=1, 2, \dots, k$ we have $x_{i_j} = z_j$.

Example: $Z=(B, C, D, B)$ is a subsequence of $X=(A, B, C, B, D, A, B)$ with corresponding index sequence $(2, 3, 5, 7)$.

Given two sequences X and Y , a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y .

Example: If $X=(A, B, C, B, D, A, B)$ and $Y=(B, D, C, A, B, A)$, the sequence (B, C, A) is a common subsequence of both X and Y . The sequence (B, C, A) is not a **longest** common subsequence of X and Y , however, since it has length 3 and the sequence (B, C, B, A) which is also common to both X and Y , has length 4.

The **longest-common-subsequence problem**: For given two sequences $X=(x_1, x_2, \dots, x_m)$ and $Y=(y_1, y_2, \dots, y_n)$ it has to be find a maximum length common subsequence of X and Y .

Remark: For the abbreviation of the longest common subsequence the LCS will be used.



Longest common subsequence

For a given sequence $X=(x_1, x_2, \dots, x_m)$ the i th **prefix** of X is $X_i=(x_1, x_2, \dots, x_i)$, for $i=0, 1, \dots, m$.

Example: If $X=(A, B, C, B, D, A, B)$, then $X_4=(A, B, C, B)$ and X_0 is the empty sequence.

Theorem (Optimal substructure of an LCS): Let $X=(x_1, x_2, \dots, x_m)$ and $Y=(y_1, y_2, \dots, y_n)$ be sequences, and let $Z=(z_1, z_2, \dots, z_k)$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

Corollary: The LCS problem has the optimal subproblem property, so it can be efficiently solved with dynamic programming.

Let $c[i, j]$ be the length of an LCS of the sequences X_i and Y_j . From the previous theorem:

$$\begin{aligned} c[i, j] &= 0, & \text{if } i=0 \text{ or } j=0, \\ c[i, j] &= c[i-1, j-1] + 1, & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ c[i, j] &= \max\{c[i, j-1], c[i-1, j]\}, & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{aligned}$$



Longest common subsequence

LCS-LENGTH(X, Y)

```
1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i-1, j-1] + 1$ 
12              $b[i, j] = „↖”$ 
13         else
14             if  $c[i-1, j] \geq c[i, j-1]$ 
15                  $c[i, j] = c[i-1, j]$ 
16                  $b[i, j] = „↑”$ 
17             else
18                  $c[i, j] \leftarrow c[i, j-1]$ 
19                  $b[i, j] \leftarrow „←”$ 
20 return  $c$  and  $b$ 
```

Efficiency: It requires $\Theta(mn)$ time and $\Theta(mn)$ memory space.



Longest common subsequence

		j	0	1	2	3	4	5	6
		y_j		<i>B</i>	<i>D</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>A</i>
i	x_i		0	0	0	0	0	0	0
0	x_i		0	0	0	0	0	0	0
1	<i>A</i>		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	<i>B</i>		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	<i>C</i>		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	<i>B</i>		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	<i>D</i>		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	<i>A</i>		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	<i>B</i>		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

The operation of LCS-LENGTH



Longest common subsequence

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

```
1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == \nwarrow$ 
4      PRINT-LCS( $b, X, i-1, j-1$ )
5      write  $x_i$ 
6  else if  $b[i, j] == \uparrow$ 
7      PRINT-LCS( $b, X, i-1, j$ )
8  else
9      PRINT-LCS( $b, X, i, j-1$ )
```

Remark: The initial call is PRINT-LCS ($b, X, X.length, Y.length$).

Efficiency: The procedure takes time $O(m+n)$.



Exercises

- Determine an LCS of $(1, 0, 0, 1, 0, 1, 0, 1)$ and $(0, 1, 0, 1, 1, 0, 1, 1, 0)$.
- Give c and b result tables and the result LCS of LCS-LENGTH with below given X and Y .
 - $X=(1, 0, 0, 1, 0)$
 - $Y=(0, 1, 0, 1)$
- Can LCS-LENGTH be written without table b ? What is PRINT-LCS in this case?
- Can the $\Theta(mn)$ memory space be reduced if we need only the length of the LCS (without LCS itself)?

