Matrix-chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied
  - That is, we want to compute the product $A_1A_2\ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  – Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication ...contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)
Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

MATRIX-MULTIPLY($A_{p \times q}$, $B_{q \times r}$)

1. **for** $i \leftarrow 1$ **to** p
2.   **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4.   **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute C
Number of scalar multiplications = pqr

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - n matrices $A_1, A_2, \ldots, A_n$ with size $p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n$
  - To determine the multiplication order such that of scalar multiplications is minimized.
  - To compute $A_i \times A_{i+1}$, we need $p_{i-1}p_ip_{i+1}$ scalar multiplications

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i\, A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

…contd

– **Key observation**: parenthesizations of the subchains $A_1A_2 \ldots A_k$ and $A_{k+1}A_{k+2} \ldots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2 \ldots A_n$ is optimal (why?)

– That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach

…contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \le k < j$

Dynamic Programming Approach

…contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ *for* $i \le k < j$
- $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

Dynamic Programming Approach

…contd

– But… optimal parenthesization occurs at one value of k among all possible $i \leq k < j$

– Check all these and select the best one

$$m[i, j] = \begin{cases} 0 & \text{if } i=j \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i<j \end{cases}$$

Dynamic Programming Approach

…contd

- To keep track of how to construct an optimal solution, we use a table *s*
- $s[i, j]$ = value of k at which $A_i A_{i+1} \dots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

MATRIX-CHAIN-ORDER($p[\,]$, 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
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
  - Each entry $s[i, j]=k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost

Example: $A_1 - 5 \times 4$ $A_2 - 4 \times 6$ $A_3 - 6 \times 2$ $A_4 - 2 \times 7$

Here dimensions are $p_0=5, p_1=4, p_2=6, p_3=2, p_4=7$

▪ For $i = j$ $M[i][j]=0$ $M[1][1]=0, M[2][2]=0, M[3][3]=0, M[4][4]=0$ For $i > j$ we does not calculate cost so $M[i][j]$ will be blank for $I > j$	▪ $M[1][2]$, here $i < j$ for $i=1, j=2$ $k=1$ because $i \le k \le j-1$ Use $M[i][j] = \min(M[i][k]+M[k+1][j]+P_{[i-1]}*p_{[k]}*p_{[j]})$ For $k=1$ $M[1][2]=M[1][1]+M[2][2]+p_0*p_1*p_2$ $= 0+0+5*4*6$ $=120$ Put value of minimum value of $M[1][2]$ in table $M[i][j]$ and put value of k in table $S[i][j]$.

	1	2	3	4
1	0			
2	-	0		
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0			
2	-	0		
3	-	-	0	
4	-	-	-	0

- M[2][3], here i < j for i=1,j=2 i = 2 j = 3

k=2 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=2

M[2][3]=M[2][2]+M[3][3]+p_1*p_2*p_3

= 0+0+4*6*2

=48

Put value of minimum value of M[2][3] in table M[i][j] and put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120		
2	-	0	48	
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1		
2	-	0	2	
3	-	-	0	
4	-	-	-	0

- M[1][3], here i < j for i=1,j=3

k=1 or 2 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=1

M[1][3]=M[1][1]+M[2][3]+p_0*p_1*p_3

= 0+48+5*4*2

=88

For k=2

M[1][3]=M[1][2]+M[3][3]+p_0*p_2*p_3

=120+0+5*6*2

=180.

Put value of minimum value of M[1][3] in table M[i][j]

And put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120		
2	-	0		
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1		
2	-	0		
3	-	-	0	
4	-	-	-	0

- M[3][4], here i < j for i=3,j=4

k=3 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=3

M[3][4]=M[3][3]+M[4][4]+p_2*p_3*p_4

= 0+0+6*2*7

=84

Put value of minimum value of M[3][4] in table M[i][j] and put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120		
2	-	0	48	
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1		
2	-	0	2	
3	-	-	0	3
4	-	-	-	0

- M[2][4], here i < j for i=2,j=4

k=2 or 3 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=2

M[2][4]=M[2][2]+M[3][4]+p_1*p_2*p_4

= 0+84+4*6*7

=252

For k=3

M[2][4]=M[2][3]+M[4][4]+p_1*p_3*p_4

= 48+0+4*2*7

=104

Put value of minimum value of M[1][3] in table M[i][j]

And put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120	88	
2	-	0	48	
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1	1	
2	-	0	2	
3	-	-	0	3
4	-	-	-	0

M[i][j]=

	1	2	3	4
1	0	120	88	
2	-	0	48	104
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1	1	
2	-	0	2	3
3	-	-	0	3
4	-	-	-	0

- M[1][4], here i < j for i=1,j=4

k=1 or 2 or 3 because $i \le k \le j-1$

Use M[i][j] =min(M[i][k]+M[k+1][j]+$p_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=1

M[1][4]=M[1][1]+M[2][4]+p_0*p_1*p_4

= 0+104+5*4*7

=244

For k=2

M[1][4]=M[1][2]+M[3][4]+p_0*p_2*p_4

= 120+84+5*6*7

=414

For k=3

M[1][4]=M[1][3]+M[4][4]+p_0*p_3*p_4

= 88+0+5*2*7

=158

Put value of minimum value of M[1][3] in table M[i][j]

And put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120	88	158
2	-	0	48	104
3	-	-	0	84
4	-	-	-	0

S[i][j]=

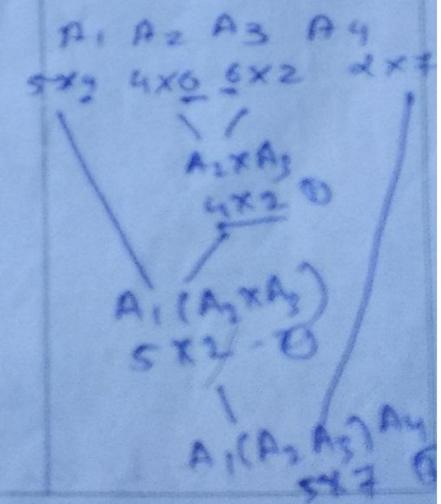
	1	2	3	4
1	0	1	1	3
2	-	0	2	3
3	-	-	0	3
4	-	-	-	0

1 $4\times2\times6 = 48$

2 $5\times2\times4 = 40$

3 $5\times7\times2 = 70$

158



```
PRINT_OPTIMAL (A, i, j,S)
If i=j
     print Ai
else
     print '('
     PRINT_OPTIMAL (A, i, S[i,j])
     PRINT_OPTIMAL (A, S[i,j]+1, j)
     print ')'
```

A 1 4

(A 1 3   A 4 4)

((A 1 1   A 2 3) . A_4)

((A_1 (A 2 2   A 3 3)) A_4)

((A_1 (A_2 A_3)) A_4)

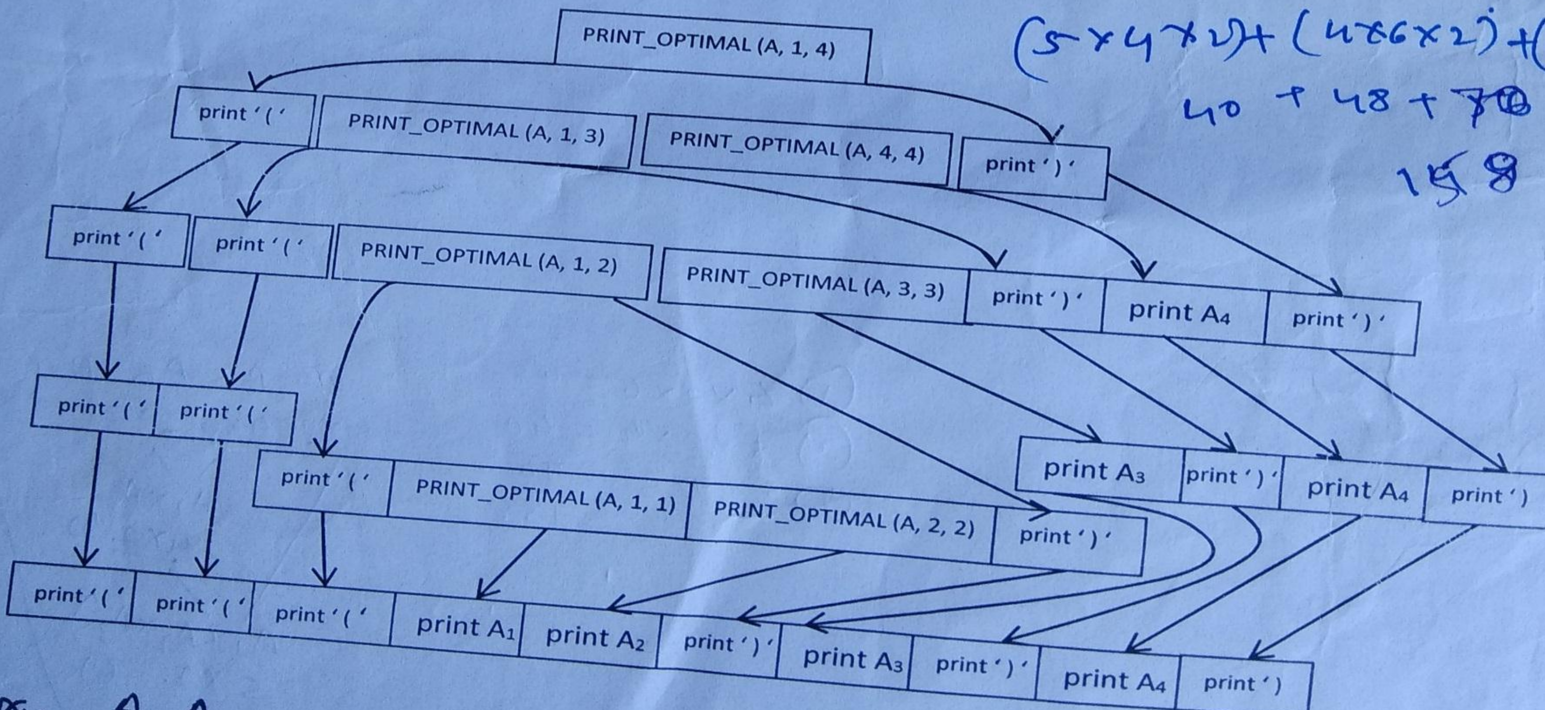
Here we are having 4 matrices let us denote it as A1,A2,A3,A4 so Calling PRINT_OPTIMAL (A, 1, 4)

<u>Note :For values of k see matrix S[i][j]</u>

$(5\times4\times2)+(4\times6\times2)+(5\times2\times7)$

$40+48+70$

158



Ex. $A_1\ A_2 - - A_6$

$A_1 - 30\times35$ $A_2 - 35\times15$

$A_3 - 15\times5$ $A_4 - 5\times10$

$A_5 - 10\times20$ $A_6 - 20\times25$

(2) sequence of dimension is
$(5, 10, 3, 12, 5, 50, 6)$

Example

- Show how to multiply this matrix chain optimally

- Solution on the board
  - Minimum cost 15,125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- **Example: 1** The initial set of dimensions are 5, 4, 6, 2, 7 meaning that we are multiplying A1 (5 × 4) times A2 (4 × 6) times A3 (6 × 2) times A4 (2 × 7).

- Here P0=5, P1=4, P2=6, P3=2, P4=7.
- For all i, m[i,i] = 0 ,

  m[1,1] = 0, m[2,2] = 0, m[3,3] = 0, m[4,4] = 0.
- Now we will fill up the table horizontally from left to right, assuming i <= k <= j-1

- let i=1, j=2, k=1

  → m[1,2] = m[i, k] + m[k + 1, j] + pi−1pkpj

  = m[1,1] + m[2,2] +P0P1P2

  = 0+0+5*4*6

  m[1,2]=120

- let i=2, j=3, k=2

  → m[2,3] = m[2,2] + m[3,3] +P1P2P3

  = 0+0+4*6*2

  m[2,3]=48

Same as for m[3,4]=84

Let i=1, j=3, k=1, or k=2

m[1,3] = min{(m[1,1]+m[2,3]+P0P1P3), (m[1,2]+m[3,3]+P0P2P3) }

= min{88,180}

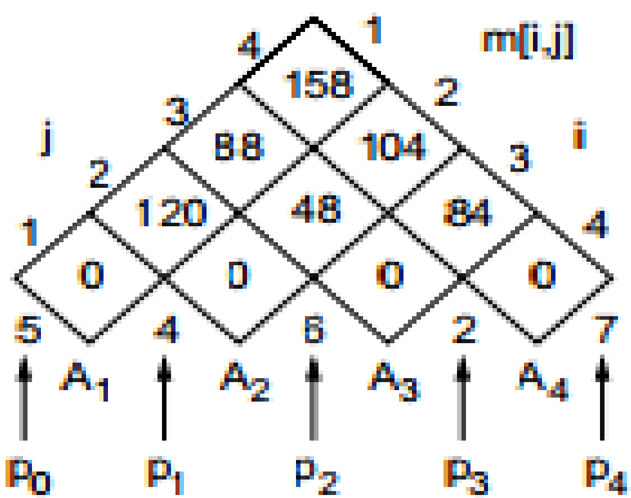
m[1,3] = 88

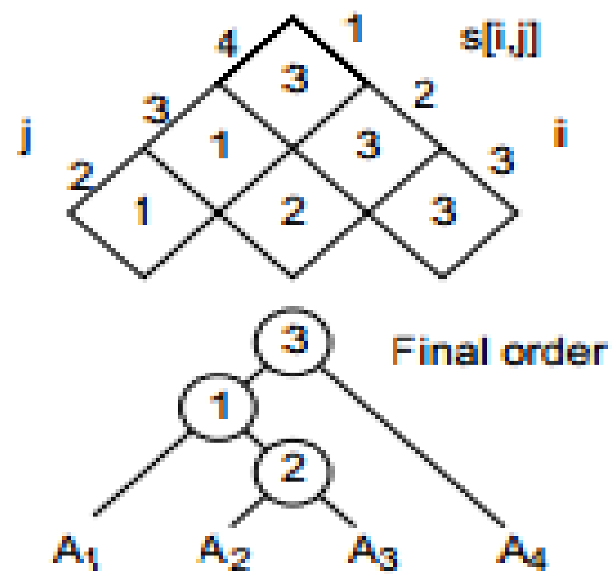
Same as…

m[2,4] = min(252,104)=104

m[1,4] = min(244,414,158)=158

Answer : The optimal sequence is **((A1(A2A3))A4)**.





Matrix Chain Multiplication

- n matrices $A_1, A_2, \ldots, A_n$ with size
  $p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n$
  To determine the multiplication order such that of scalar multiplications is minimized.
- To compute $A_i \times A_{i+1}$, we need $p_{i-1}p_ip_{i+1}$ scalar multiplications

$$m(i,\ j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \left\{ m(i,\ k) + m(k+1,\ j) + p_{i-1}p_kp_j \right\} & \text{if } i < j \end{cases}$$

- E.g. A1:15*20, A2:20*5, A3:5*10
- P0=15, p1=20, p2=5, p3=10

Cont…

- m[1,2]=m[1,1]+m[2,2]+P0P1P2

  =0+0+15*20*5

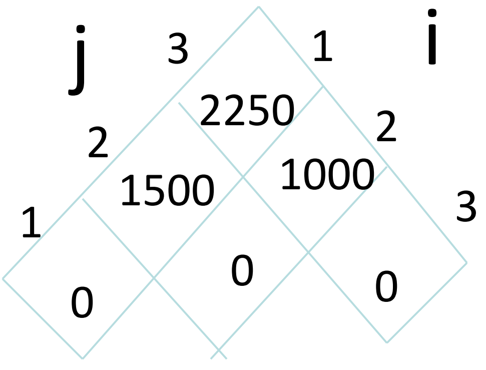
  =1500
- m[2,3]=m[2,2]+m[3,3]+P1P2P3

  =0+0+20*5*10

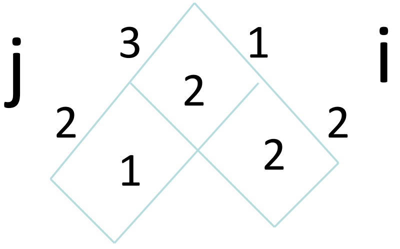
  =1000
- m[1,3]=m[1,2]+m[3,3]+P0P2P3

  =1500+0+15*5*10

  =2250



For m



For k

Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of “president”: pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.

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

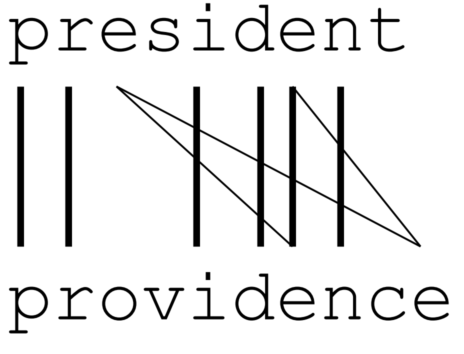
LCS

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



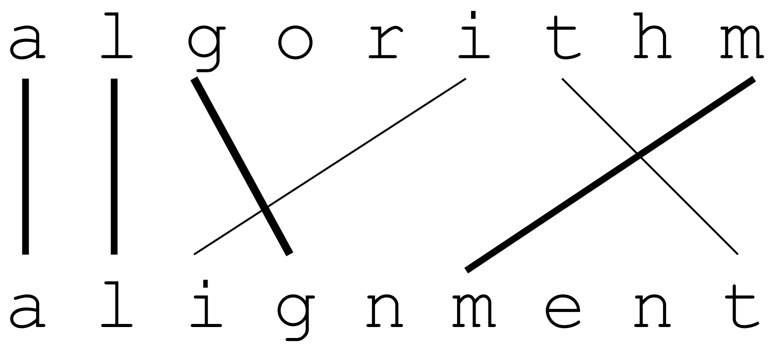
LCS

Another example:

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm.



How to compute LCS?

- Let A=$a_1a_2\ldots a_m$ and $B=b_1b_2\ldots b_n$.
- *len*(*i, j*): the length of an LCS between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$
- With proper initializations, *len*(*i, j*) can be computed as follows.

$$len(i,j)=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ len(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j\ , \\ \max(len(i,j-1), len(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j. \end{cases}$$

procedure *LCS-Length(A, B)*

for $i \leftarrow 0$ **to** m **do** $len(i,0) = 0$

for $j \leftarrow 1$ **to** n **do** $len(0,j) = 0$

for $i \leftarrow 1$ **to** m **do**

    for $j \leftarrow 1$ **to** n **do**

        if $a_i = b_j$ **then** $\left[\begin{array}{l} len(i,j) = len(i-1,j-1)+1 \\ prev(i,j) = \text{"}\nwarrow\text{"} \end{array}\right.$

        else if $len(i-1,j) \geq len(i,j-1)$

            then $\left[\begin{array}{l} len(i,j) = len(i-1,j) \\ prev(i,j) = \text{"}\uparrow\text{"} \end{array}\right.$

            else $\left[\begin{array}{l} len(i,j) = len(i,j-1) \\ prev(i,j) = \text{"}\leftarrow\text{"} \end{array}\right.$

return *len* and *prev*

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Running time and memory: O(*mn*) and O(*mn*).

The backtracing algorithm

procedure *Output-LCS(A, prev, i, j)*

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

if *prev(i, j)=*"$\nwarrow$" **then** $\left[\begin{array}{l} Output-LCS(A, prev, i-1, j-1) \\ \text{print} \quad a_i \end{array}\right.$

else if *prev(i, j)=*"$\uparrow$" **then** *Output-LCS(A, prev, i-1, j)*

else *Output-LCS(A, prev, i, j-1)*

i		*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0		0	0	0	0	0	0	0	0	0	0	0
1	*p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	*r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	*e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	*s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	*i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	*d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	*e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	*n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	*t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Output: *priden*