Dynamic programming
Advanced Algorithms

Master DSC/MLDM/CPS2

Amaury Habrard

amaury.habrard@univ-st-etienne.fr

Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet Saint-Étienne

amaury.habrard@univ-st-etienne.fr

Semester 1

Matrix chain multiplication (divide and conquer)

Ex1: Div and Conquer for Matrix chain multiplication

begin
  if $i = j$ **then**
    Return 0;
  $min \leftarrow \infty$;
  for $k \leftarrow i$ *to* $j - 1$ **do**
    $q_1 \leftarrow RecursiveMatrixChain(i, k)$;
    $q_2 \leftarrow RecursiveMatrixChain(k + 1, j)$;
    $q \leftarrow q_1 + q_2 + p_{i-1} p_k p_j$;
    if $min > q$ **then**
      $min \leftarrow q$;
  Return min;
end

Algorithme 1: Algorithm RecursiveMatrixChain(i,j)

For the sake of simplicty, the matrix chain is omitted in the algorithm

Ex1: complexity recursive approach

Let c be a constant for the costs of the operations used in the algo, the complexity is given by the recurrence:

$$T(n) = c + \sum_{k=1}^{n-1}(T(k) + T(n-k) + c), \text{ if } n \geq 2 \text{ and otherwise } T(n) = c.$$

This can be rewritten as: $T(n) = 2\sum_{i=1}^{n-1} T(i) + c(n-1)$.

A simple idea to show the exponential complexity:
$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + cn$, then $T(n) \geq 2T(n-1)$ which implies $\Omega(2^n)$.

Ex1 complexity (ctd)

We can prove the following upper bound by induction that $T(n)$is $O(n3^{n-1})$

$$
\begin{aligned}
T(n) &\leq 2\sum_{i=1}^{n-1} T(i) + cn \\
&\leq 2\sum_{i=1}^{n-1}(ci3^{i-1}) + cn \text{ by induction hypothesis} \\
&\leq 2cn\sum_{i=1}^{n-1} 3^{i-1} + cn \text{ since } i < n \\
&\leq 2cn(\frac{3^{n-1}-1}{2}) + c(n) = cn3^{n-1}
\end{aligned}
$$

with another analysis, we can get $O(4^n/n^{3/2})$.

Longest Common subsequence

Complexity recursive solution

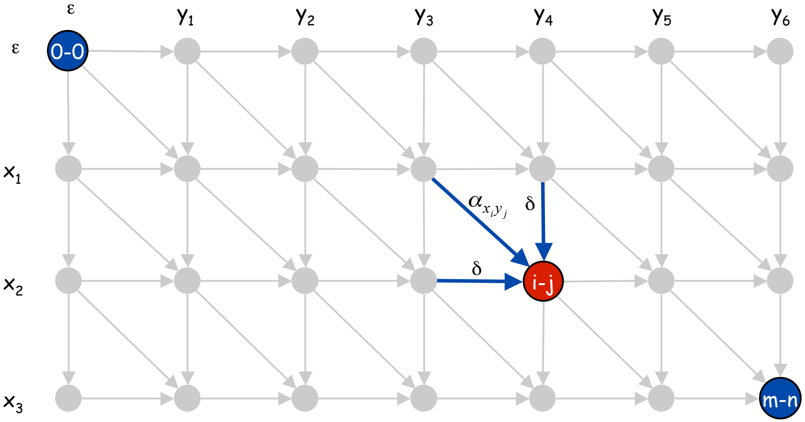
We consider the case where the two strings have no letter in common (worst case).

$$\begin{aligned} T(n,m) &= T(n,m-1)+T(n-1,m)+c \\ &= (T(n,m-2)+T(n-1,m-1))+ \\ &\quad (T(n-1,m-1)+T(n-2,m))+c' \\ &\geq 2T(n-1,m-1) \end{aligned}$$

thus the algo can need about $2^{\min(n,m)}$ operations, and is thus exponential when applied on two strictly different strings.

LCS: Linear

Let $c(i, j)$ be the optimal path from $(0, 0)$ to (i, j)



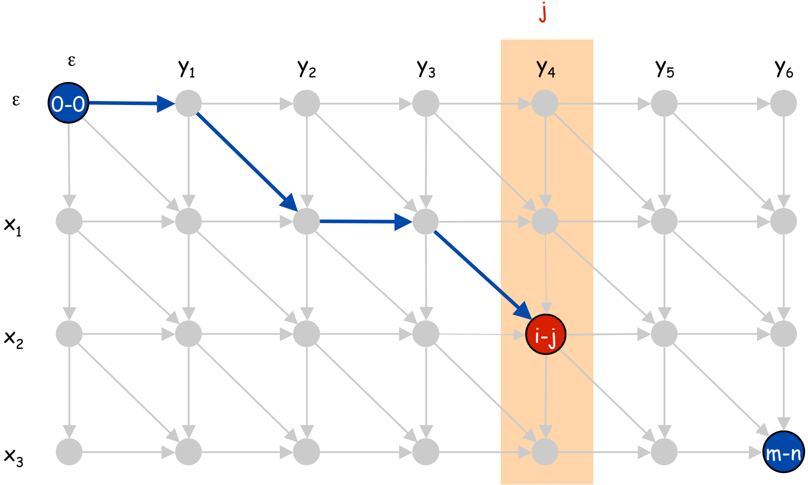
Longest common subsequence (linear space)

```
begin
    Create Array c[0..1, 0..n] where n = size(Y);
    Initialize c[0, j] = 0 for all j;
    for i from 1 to m = size(X) do
        c[1, 0] ← 0;
        for j = 1 to n do
            if x_i = y_j then
                c[1, j] ← c[0][j − 1] + 1;
            else
                if c[1, j − 1] < c[0, j] then  c[1, j] ← c[0, j];
                else  c[1, j] ← c[1, j − 1];
        Move line 1 of C to line 0: for j = 0 to n do  c[0, j] ← c[1, j];
end
```

$\Rightarrow$ We cannot find the LCS!

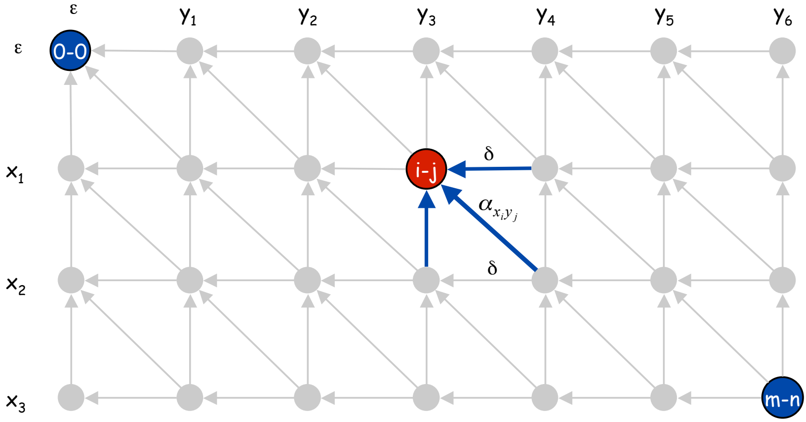
LCS: forward

Let (i, j) be the optimal path from $(0, 0)$ to (i, j)
Can compute $c(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space



LCS: backward

Let $g(i,j)$ be the optimal path from (i,j) to (m,n)
Can compute by reversing the edge orientations and inverting the roles of $(0,0)$ and (m,n)



LCS: backward

The same algorithm can be written in a bottom-up manner instead of top-down. We can rewrite the cost function as:

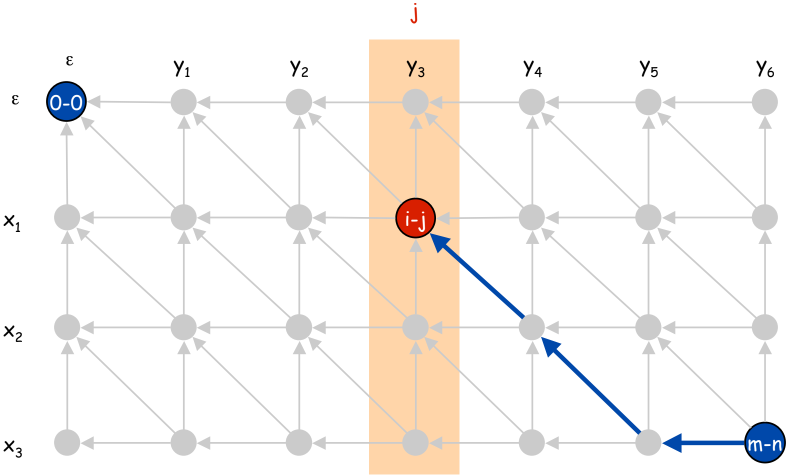
$$g[i,j] = \begin{cases} 0 & \text{if} i = m \text{ or } j = n \\ g[i+1, j+1] + 1 & \text{if } x_{i+1} = y_{j+1} \text{ and } i < m, j < n \\ \max(g[i+1, j], g[i, j+1]) & \text{otherwise} \end{cases}$$

g defines the size of the LCS between suffixes (from (i, j) to (n, m)). We can thus define a similar dynamic programming algo with a “backward” procedure.

In the following, we denote by $\mathbf{G_{XY}}$ the cost matrix computed by the algo (forward or backward).

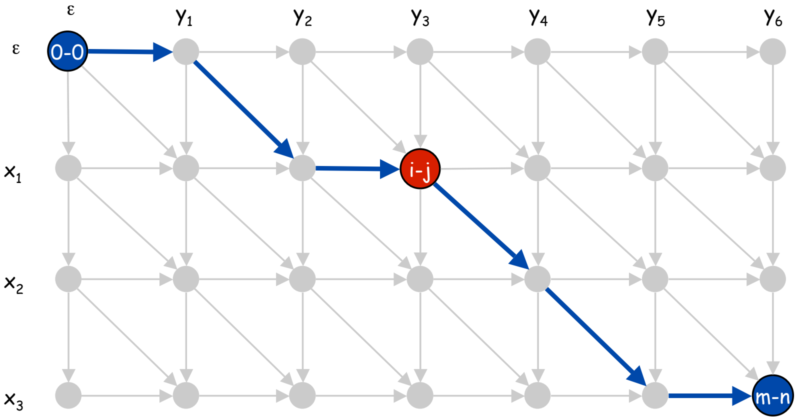
LCS: backward

Let $g(i, j)$ be the optimal path from (i, j) to (m, n)
$g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space



LCS: forward + backward

The cost of the shortest path that uses (i, j) is $c(i, j) + g(i, j)$.



LCS with divide and conquer

Property 1

The size of the LCS that passes through entry i, j is $c[i,j] + g[i,j]$.

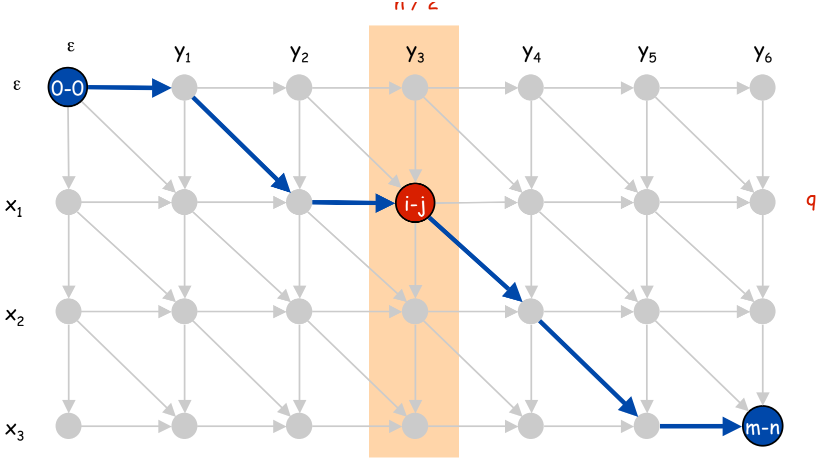
Proof. Let l_{ij} be the length of the LCS that passes through (i,j) in G_{XY}.

Clearly, any solution must go from $(0,0)$ to (i,j) and then from (i,j) to (m,n). Thus its length is at least $c(i,j) + g(i,j)$, and so we have $l_{ij} \geq c(i,j) + g(i,j)$.

On the other hand, consider the solution that consists of a maximum size subsequence from $(0,0)$ to (i,j), followed by a maximum size solution from (i,j) to (m,n). This path has length $c(i,j) + g(i,j)$, and so we have $l_{ij} \leq c(i,j) + g(i,j)$. It follows that $l_{ij} = c(i,j) + g(i,j)$.

LCS: forward + backward

Let q that maximizes $c(q, n/2) + g(q, n/2)$, then the optimal path from $(0,0)$ to (m,n) uses $(q, n/2)$



LCS with divide and conquer (ctd)

Property 2

Let k be any number in $\{0, \ldots, n\}$, and let q be an index that maximizes the quantity $c(q,k) + g(q,k)$. Then there is an optimal solution that passes through the entry (q,k).

Proof. Let $l*$ denotes the length of the optimal solution in G_{XY}.

Now fix a value of $k \in \{0, \ldots, n\}$. The optimal solution must use some entry in the kth column of G_{XY} -let's suppose it is entry (p,k)- and thus by Property 1 $l* = c(p,k) + g(p,k) \geq \max_q c(q,k) + g(q,k)$.

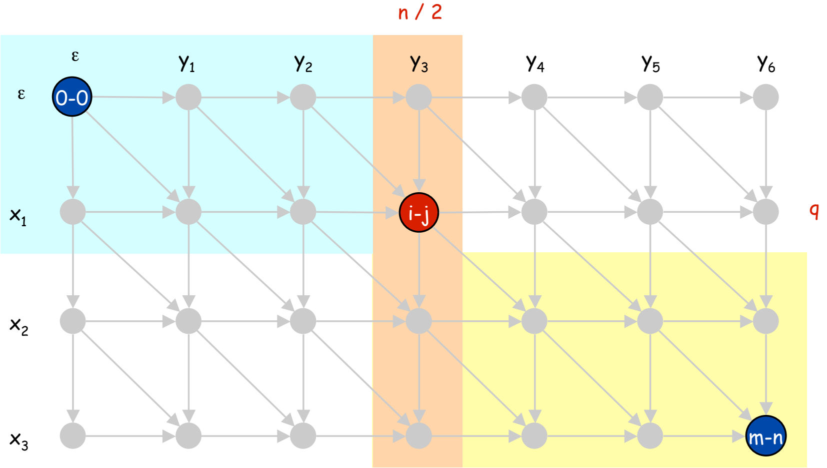
Now consider the index q that achieves the maximum in the right-hand side of this expression; we have $l* \geq c(q,k) + g(q,k)$. By Property 1 again, the optimal solution using the entry (q,k) has length $c(q,k) + g(q,k)$, and since $l*$ is the maximum length of any optimal solution we have $l* \leq c(q,k) + g(q,k)$.

It follows that $l* = c(q,k) + g(q,k)$. Thus the optimal solution using the entry (q,k) has length $l*$.

LCS: Divide + conquer

Divide: find q that maximizes $c(q, n/2) + g(q, n/2)$ using Dynamic programming

Conquer: recursively compute the LCS



Algo

```
begin
    Let m = size(X) and n = size(Y);
    if m ≤ 2 or n ≤ 2 then
        Compute the solution using LCS-LENGTH algo and store it in P;
    else
        Call Space-efficient-LCS-LENGTH(X,Y[1:n/2]);
        Call Backward-Space-Efficient-LCS-LENGTH(X,Y[n/2+1:n]);
        Let q be the index maximizing c(q, n/2) + g(q, n/2);
        Add (q, n/2) to global list P;
        Divide-and-Conquer-LCS(X[1:q],Y[1:n/2]);
        Divide-and-Conquer-LCS(X[q+1:m],Y[n/2+1:n]);
    return P;
end
```

Algorithme 2: Divide-and-Conquer-LCS(X,Y)

$\Rightarrow$ Uses $O(m+n)$ space !! We need to check the time complexity is still in $O(mn)$.

Time Complexity: $O(m, n)$

Let $T(m,n)$ denote the max running time of the algo. The algo performs $O(mn)$ for the two calls to the space-efficient algo. Then, it runs recursively on strings of size q and $n/2$ and on strings of size $m-q$ and $n/2$. Thus for some constant c and some choice of index q we have:

$$\begin{array}{rcl} T(m,n) & \leq & cmn + T(q, n/2) + T(m-q, n/2), \\ T(m,2) & \leq & cm, \\ T(2,n) & \leq & cn. \end{array}$$

In a first step, to simplify the analysis, let us assume that $m = n$ and $q = n/2$, thus we can write:
$T(n) \leq 2T(n/2) + cn^2$
This recurrence implies a $O(n^2)$ (same techniques as in Exercise sheet 1)

Time Complexity: $O(m, n)$ (ctd)

Now, we know that when $m = n$, the running time grows like n^2. From this analysis, we can guess that the general recurrence grows like mn, *ie* we guess that $T(m, n) \leq kmn$ for some constant k, and we try to prove it by induction.
When $m \leq 2$, $n \leq 2$, the hypothesis is true as long as $k \geq c/2$. Now assume $T(m', n') \leq km'n'$ for pairs (m', n') with $m' < m$ and $n' < n$, we have:

$$\begin{array}{rcl} T(m,n) & \leq & cmn + T(q, n/2) + T(m-q, n/2) \\ & \leq & cmn + kqn/2 + k(m-q)n/2 \text{ by induction hypothesis} \\ & = & cmn + kqn/2 + kmn/2 - kqn/2 \\ & = & (c + k/2)mn. \end{array}$$

Thus, the inductive step will work if we choose $k = 2c$, which completes the proof.

Longest Monotonically Increasing Subsequence

Ex: Longest Monotonically Increasing Subsequence

1. If we consider the empty string as a possible answer, then we can have 2^n subsequences: the number of subsets of symbols
2. $(2,7)$ $(2,4)$, $(2,6)$ (twice), $(2,8)$, $(7,8)$, $(4,6)$ (twice), $(4,8)$, $(6,8)$, $(1,8)$, $(1,6)$, $(2,7,8)$, $(2,4,6)$ (twice), $(2,4,8)$, $(2,6,8)$, $(2,7,8)$, $(2,4,6,8)$. The last one is thus the answer of the problem.

Ex: Longest Monotonically Increasing Subsequence (ctd)

3. Let $L(j)$ be the length of the longest increasing subsequence of $x_1, \ldots, x_j$. We can then define the following recursive definition:

$$L(j) = 1 + max\{L(i) | i = 1, \ldots, j-1 \text{ and } x_i < x_j\}$$

This gives a relation with a subproblem of smaller size. The optimality can be easily proved by contradiction. Then the algorithm is:

$L(0) \leftarrow 0$; $x_0 \leftarrow -\infty$;
$L(1) \leftarrow 1$;
for $j = 2, \ldots, n$ **do**
  $L(j) \leftarrow 1 + max\{L(i) | i = 0, \ldots, j-1 \text{ and } x_i < x_j\}$;

The answer corresponds then to $\max_j L(j)$. The complexity is in $O(n^2)$ since at each step we need to find the max of at most n elements.

4. Add a new array of size n to the previous algorithm, let's say q. At step j, stores in $q[j]$ the index i that corresponds to the maximum of $\{L(i)|i = 0, \ldots, j-1 \text{ and } x_i < x_j\}$, we add this line to the previous loop:
$q[j] \leftarrow \texttt{argmax}_i\{L(i)|i = 0, \ldots, j-1 \texttt{ and } x_i < x_j\}$
Then the following algorithm returns the correct longest increasing subsequence.

Input: i: index, q:array, x: sequence of integers

if $i > 0$ **then**

  $s \leftarrow$ PRINT(q[i]);

  return $s.' '.x_i$;

Algorithme 3: Algorithm PRINT

Ex: Longest Monotonically Increasing Subsequence (Ctd)

5. (easy) sort the sequence of numbers and apply Longest Common subsequence algorithm between this sequence and the original input

Printing neatly

Printing neatly - 1 (extras)

We define $extras[i,j] = M - \sum_{k=i}^{j} l_k - (j-i)$ the number of extra spaces at a line containing words i through j (note to fill this array, you can apply a dynamic programming approach too).
The cost of including a line containing words i through j:

$$lc[i,j] = \begin{cases} \infty & \text{if } extras[i,j] < 0 \text{ (ie words do not fit)} \\ 0 & \text{if } j = n \text{ and } extras[i,j] \geq 0 \text{ (last line costs 0)} \\ (extras[i,j])^3 & \text{otherwise} \end{cases}$$

By using ∞ when the words do not fit on a line we prevent such an arrangement to be a solution, and by making the cost 0 for the last line we prevent the arrangement on the last line to influence the sum.

Printing neatly - 1 (extras) Ctd

We want to minimize the sum lc over all the lines of the paragraph. The subproblems are how to optimally arrange words $1, \ldots, j$, where $j = 1, \ldots, n$.
Consider an optimal arrangement of words $1, \ldots, j$. Suppose we know that the last line ends with word j and begins with word i. The preceding lines contain words $1, \ldots, i-1$. These lines must contain an optimal arrangement, otherwise we could take a better arrangement for these lines which would lead to a better solution than the optimal one (contradiction).
Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains words $i, \ldots, j$ then:

$$c[j] = c[i-1] + lc[i, j].$$

We set $c[0] = 0$, which leads to $c[1] = lc[1, 1]$.

Printing neatly - 2

We have to find which words will be on the last line for a subproblem $c[j]$: try all the possibilities and take the best one. We store each cut for a line in a tab p to be able to find the arrangement: the last line starts at word $p[n]$ through word n, the previous line starts at word $p[p[n]]$ through word $p[n]-1$, and so on....

$$c[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j}(c[i-1] + lc[i,j]) & \text{if } j > 0. \end{cases}$$

Printing neatly - 3 - fill c iteratively, but we need before *extras* and *lc*

```
begin
    Compute extras[i,j] 1 ≤ i,j ≤ n;
    Compute lc[i,j] 1 ≤ i,j ≤ n;
    c[0] ← 0;
    for j ← 1,...,n do
        c[j] ← min_{1≤i≤j}(c[i−1] + lc[i,j]);
        p[j] ← k s.t. min_{1≤i≤j}(c[i−1] + lc[i,j]) = c[k−1] + lc[k,j];
    Return p;
end
```

This algorithm is clearly in $O(n^2)$, pay attention to the fact that **extras must be computed in** $O(n^2)$ which can be done **also using DP**.

We can do a bit better in $O(nM)$: at most $\lceil M/2 \rceil$ words can fit on a line (each word is at least one character long and there is a space between each word). Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i,j] = \infty$, thus we only need to compute and store *extras* ans lc for $j - i + 1 \leq \lceil M/2 \rceil$.

Printing neatly - 4- Print!

The printed output of GIVE-LINES is a triple (k, i, j) indicating that words $i, \ldots, j$ are printed on line k.

begin
  $i \leftarrow p[j]$;
  if $i == 1$ **then** $k \leftarrow 1$;
  else $k \leftarrow$ GIVE-LINES(p,$i-1$)+1;
  print(k,i,j) //we print words i to j on line k;
  return k;
end

Algorithme 4: GIVE-LINES(p, j)

The initial call is GIVE-LINES(p, n). Since the input value j decreases in each recursive call, the algo takes $O(n)$ time.