095946- ADVANCED ALGORITHMS AND PARALLEL PROGRAMMING

Fabrizio Ferrandi

a.a. 2021-2022

- **Dynamic Programming**
  - Longest common subsequence
  - Optimal substructure
  - Overlapping subproblems

Dynamic Programming

- The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another.
- The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.
- The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics.
  - This usage is the same as that in the phrases linear programming and mathematical programming, a synonym for mathematical optimization.

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- **Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.**

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- **Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$, find a longest subsequence common to them both.**

"a" ***not*** **"the"**

x:	A	B	C	B	D	A	B
y:	B	D	C	A	B	A	

Dynamic programming

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- **Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.**

“a” *not* “the”

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \,..\, m]$ to see if it is also a subsequence of $y[1 \,..\, n]$.

Brute-force LCS algorithm

Check every subsequence of $x[1 \, . \, . \, m]$ to see if it is also a subsequence of $y[1 \, . \, . \, n]$.

Analysis

- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$
$= \text{exponential time.}$

Towards a better algorithm

Simplification:

1. **Look at the *length* of a longest-common subsequence.**
2. **Extend the algorithm to find the LCS itself.**

Towards a better algorithm

Simplification:

1. **Look at the *length* of a longest-common subsequence.**
2. **Extend the algorithm to find the LCS itself.**

Notation: **Denote the length of a sequence s by $|s|$.**

Towards a better algorithm

Simplification:

1. **Look at the *length* of a longest-common subsequence.**
2. **Extend the algorithm to find the LCS itself.**

Notation: **Denote the length of a sequence s by $|s|$.**

Strategy: **Consider *prefixes* of x and y.**

- **Define** $c[i, j] = |\mathrm{LCS}(x[1\,.\,.\,i], y[1\,.\,.\,j])|$.
- **Then,** $c[m, n] = |\mathrm{LCS}(x, y)|$.

Recursive formulation

Theorem.

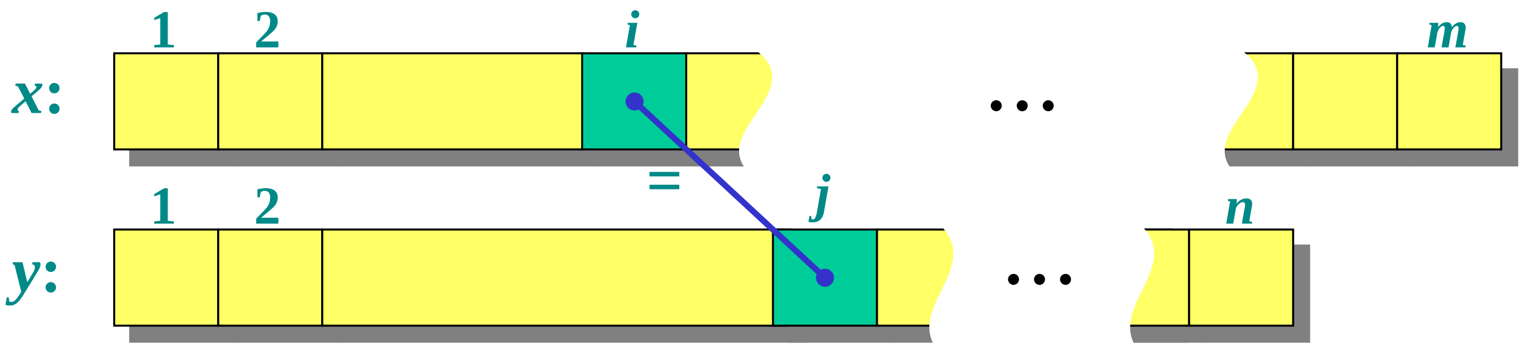
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \textbf{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \textbf{otherwise.} \end{cases}$$

Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \textbf{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \textbf{otherwise.} \end{cases}$$

Proof. **Case** $x[i] = y[j]$**:**

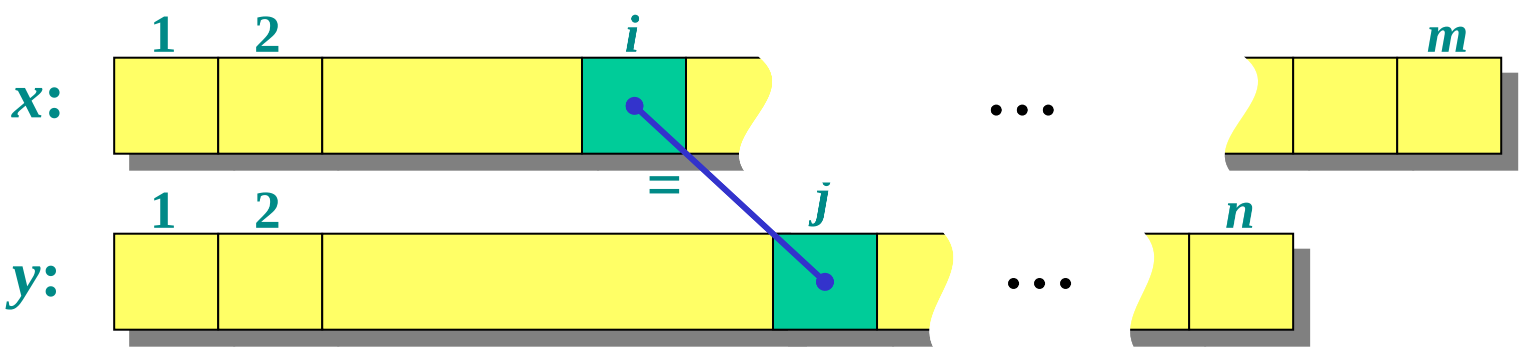


Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \textbf{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \textbf{otherwise.} \end{cases}$$

Proof. **Case $x[i] = y[j]$:**



Let $z[1 \,.\,.\, k] = \mathrm{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.

Proof (continued)

Claim: $z[1 \,..\, k-1] = \mathrm{LCS}(x[1 \,..\, i-1], y[1 \,..\, j-1])$.

Suppose w is a longer CS of $x[1 \,..\, i-1]$ and $y[1 \,..\, j-1]$, that is, $|w| > k-1$. Then, *cut and paste*: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \,..\, i]$ and $y[1 \,..\, j]$ with $|w \,||\, z[k]| > k$. Contradiction, proving the claim.

Proof (continued)

Claim: $z[1\,.\,.\,k–1] = \mathrm{LCS}(x[1\,.\,.\,i–1], y[1\,.\,.\,j–1])$.
Suppose w is a longer CS of $x[1\,.\,.\,i–1]$ and $y[1\,.\,.\,j–1]$, that is, $|w| > k–1$. Then, *cut and paste*: $w \,\|\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,.\,.\,i]$ and $y[1\,.\,.\,j]$ with $|w \,\|\, z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i–1, j–1] = k–1$, which implies that $c[i, j] = c[i–1, j–1] + 1$.

Other cases are similar.

□

Dynamic-programming hallmark #1

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

i = lunghezza di x e j è la lunghezza di y

LCS(x, y, i, j) **// ignoring base cases**

if $x[i] = y[j]$

then $c[i, j] \leftarrow$ **LCS**$(x, y, i-1, j-1) + 1$ add 1 to then optimal solution to the subproblem

else $c[i, j] \leftarrow \max\{$**LCS**$(x, y, i-1, j)$, **LCS**$(x, y, i, j-1)\}$

return $c[i, j]$

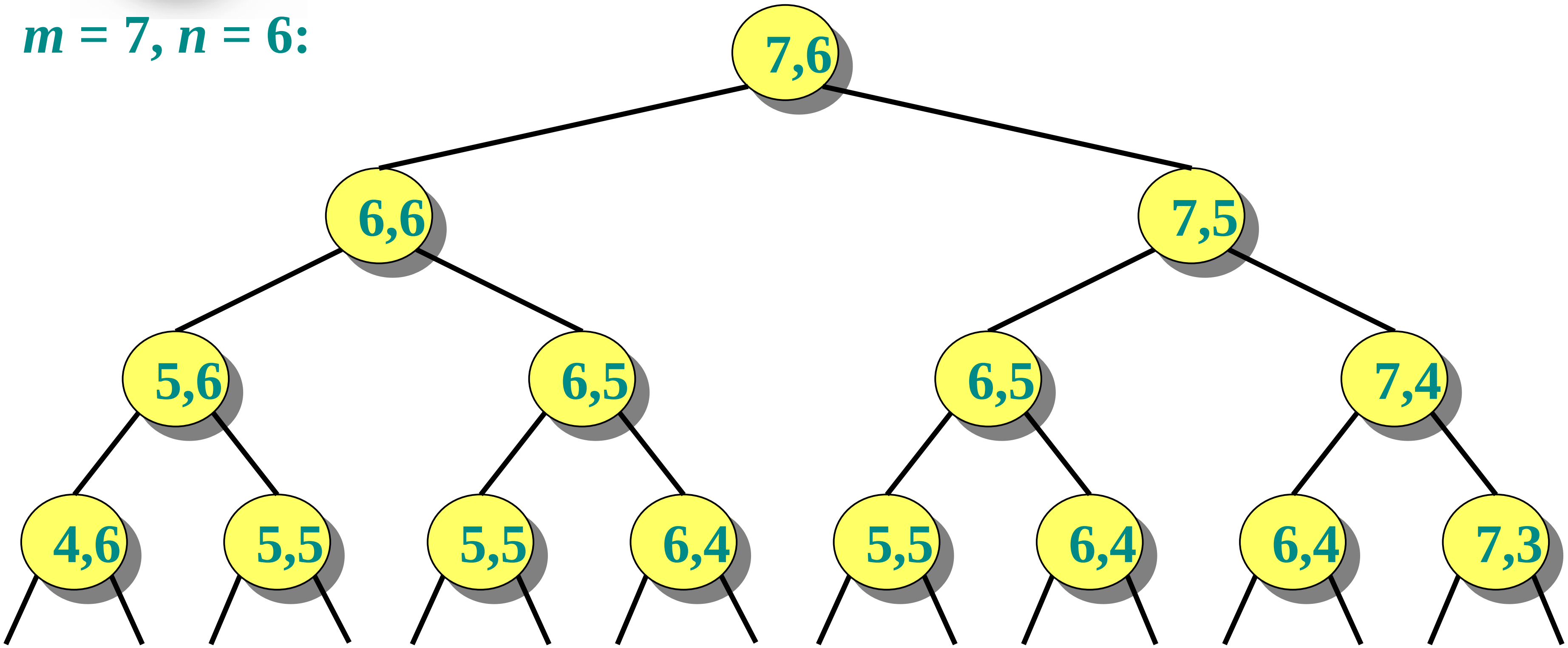
Recursive algorithm for LCS

$\mathrm{LCS}(x, y, i, j)$ // ignoring base cases
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j), \mathrm{LCS}(x, y, i, j-1)\}$
  return $c[i, j]$

Worse case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

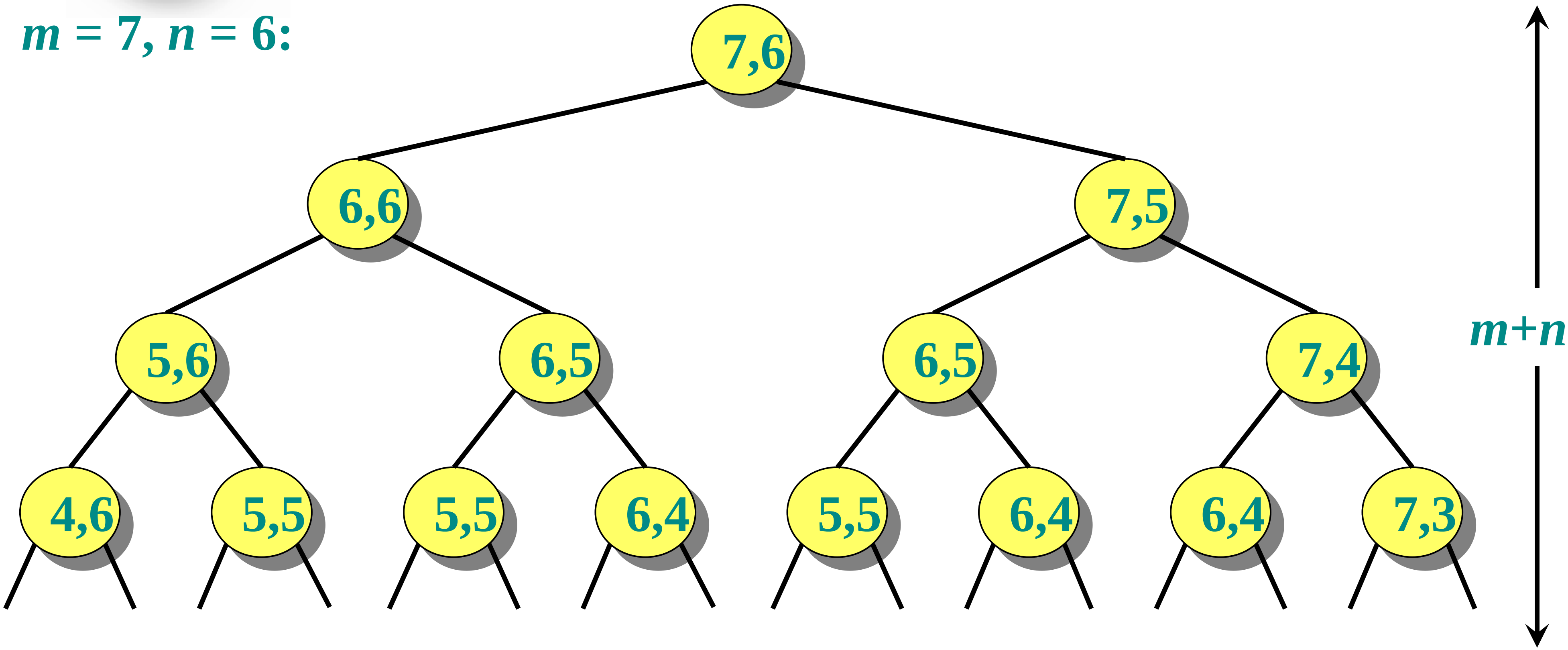
Recursion tree

***m* = 7, *n* = 6:**



Recursion tree

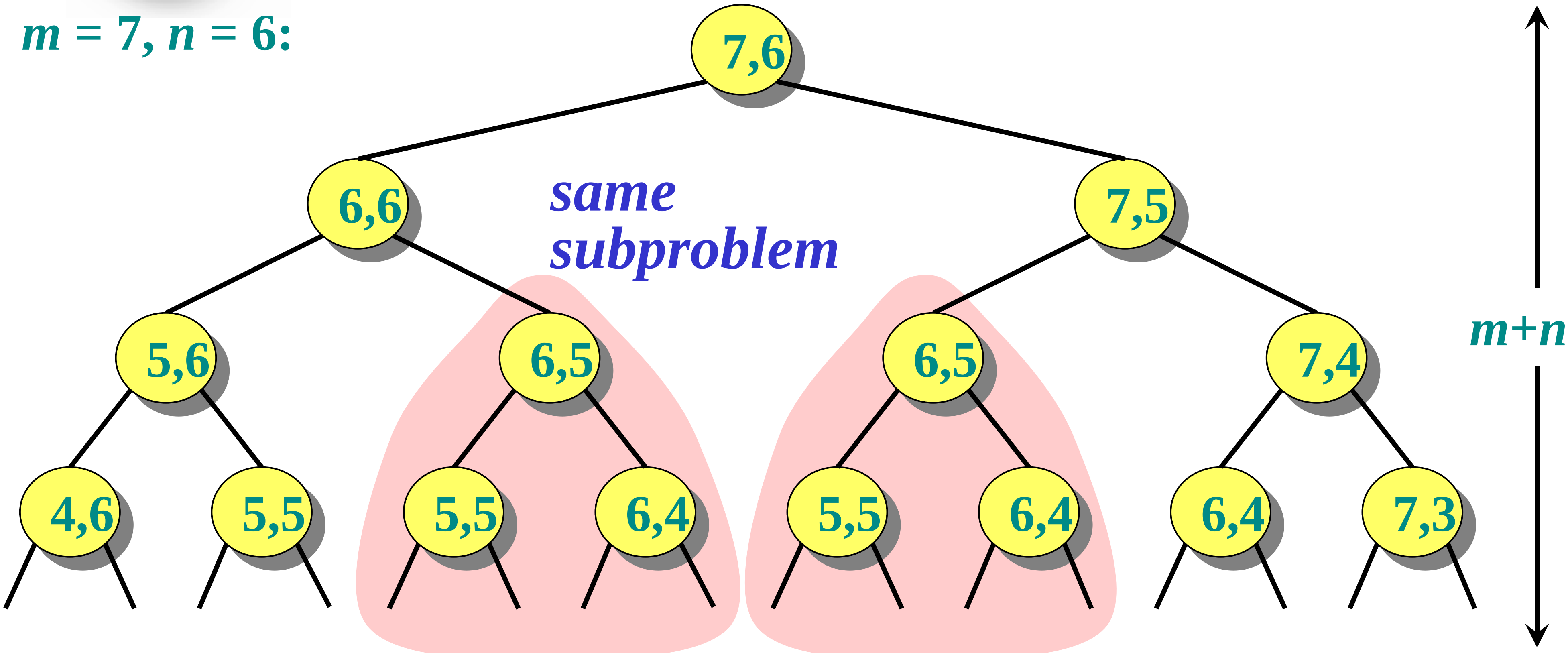
$m = 7, n = 6$:



Height $= m + n \Rightarrow$ **work potentially exponential.**

Recursion tree

$m = 7, n = 6$:



NB che i sottoproblemi con il colore rosa, sono gli stessi problemi, quindi non ha senso considerare entrambi i sotto problemi

Height $= m + n \Rightarrow$ **work potentially exponential,**
but we're solving subproblems already solved!

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

Overlapping subproblems
A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization algorithm

Memoization: **After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.**

Memoization algorithm

Memoization: **After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.**

$$
\begin{array}{l}
\textbf{LCS}(x, y, i, j) \\
\quad \textbf{if } c[i, j] = \text{NIL} \quad \text{(sarebbe il NULL)} \\
\quad\quad \textbf{then if } x[i] = y[j] \\
\quad\quad\quad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
\quad\quad\quad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{LCS}(x, y, i, j-1)\}
\end{array}
\left.\vphantom{\begin{array}{l}a\\b\end{array}}\right\} \textit{same as before}
$$

Memoization algorithm

Memoization: **After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.**

$$
\begin{aligned}
&\textbf{LCS}(x, y, i, j)\\
&\quad \textbf{if } c[i, j] = \text{NIL}\\
&\qquad \textbf{then if } x[i] = y[j]\\
&\qquad\quad \textbf{then } c[i, j] \leftarrow \textbf{LCS}(x, y, i\text{–}1, j\text{–}1) + 1\\
&\qquad\quad \textbf{else } c[i, j] \leftarrow \max\{\textbf{LCS}(x, y, i\text{–}1, j),\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{LCS}(x, y, i, j\text{–}1)\}
\end{aligned}
$$

same as before

Time $= \Theta(mn) =$ **constant work per table entry.**
Space $= \Theta(mn)$.

quando all'esame chiede la complessità di LCS scrivi anche la Space complexity

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

questa è la matrice c

c(i,j) -> ved come è costruita sopra e si capisce.
considera che i parte dalla fine e j dall'inizio

		j A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
(i-1) B	0	1	2	2	3	3	3	4
i A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Time = $\Theta(mn)$.

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

adesso io voglio la sottosequenza non la sua cardinalità.

partendo dal 4 in basso a dx vedo la sottosequenza massima (4 elementi) da ricostruire -> vedi la cosa in verde.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:
Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.
Exercise: $O(\min\{m, n\})$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Acknowledge

Based on Introduction to Algorithms CLRS