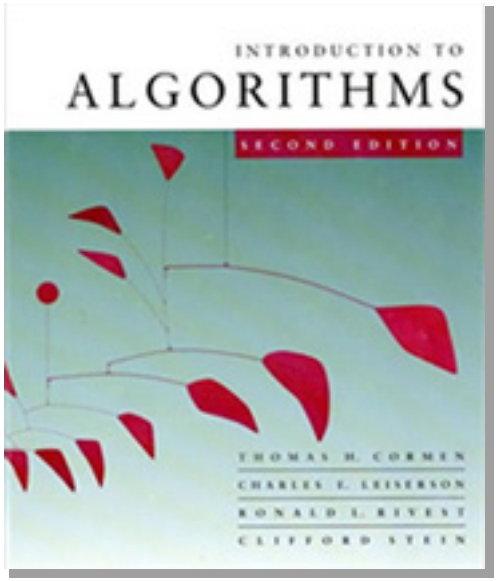
Algoritmalara Giriş

6.046J/18.401J

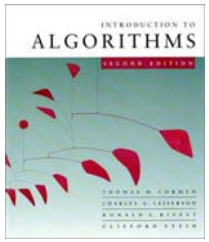


Ders 15

Dinamik Programlama

- En uzun ortak altdizi(LCS)
- En uygun altyapı
- Altproblemleri aşma

Prof. Charles E. Leiserson

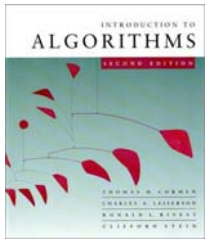


Dinamik Programlama

Böl ve fethet gibi bir teknik tasarlayın.

Örnek: *En uzun ortak altdizi (LCS)*

- İki tane, $x[1 \,..\, m]$ve $y[1 \,..\, n]$ dizisi verilmiş, ikisinde de ortak olan en uzun altdiziyi bulun.



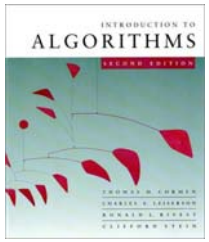
Dinamik Programlama

Böl ve fethet gibi bir teknik tasarlayın.

Örnek:*En uzun ortak altdizi (LCS)*

- İki tane, x[1 . . m]ve y[1 . . n] dizisi verilmiş, ikisinde de ortak olan en uzun altdiziyi bulun.

En uzun altdizi tek değildir.



Dinamik Programlama

Böl ve fethet gibi bir teknik tasarlayın.

Örnek:En uzun ortak altdizi (LCS)

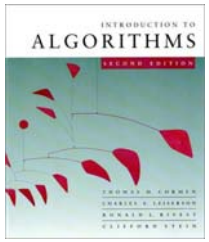
- İki tane, x[1 . . m]ve y[1 . . n] dizisi verilmiş,

ikisinde de ortak olan en uzun altdiziyi bulun.

En uzun altdizi tek değildir.

x: A B C B D A B

y: B D C A B A



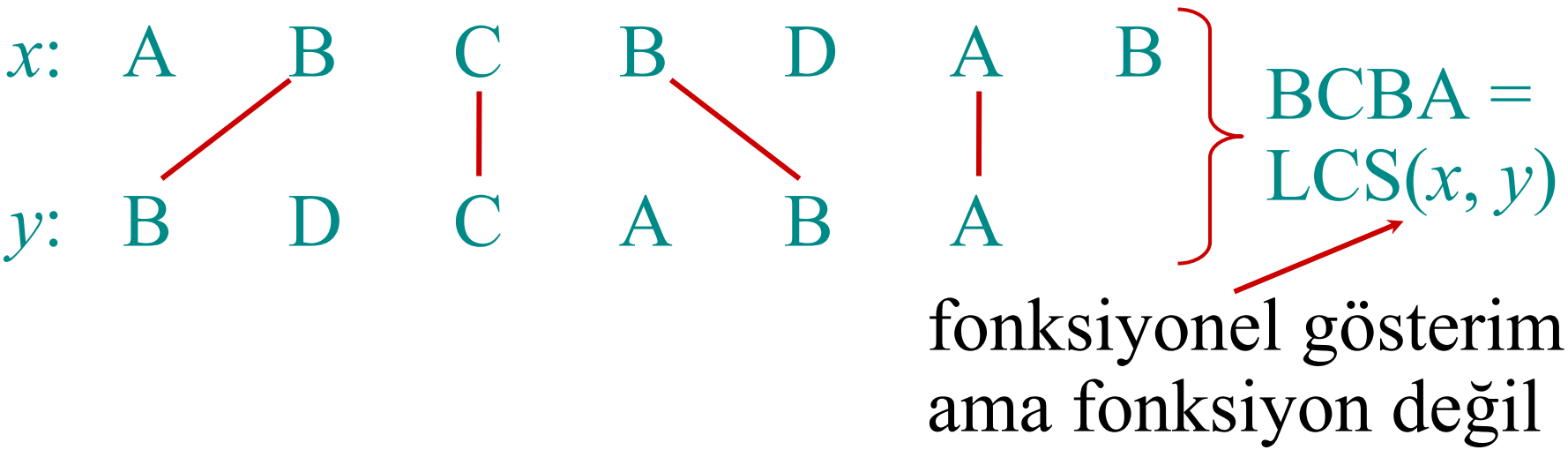
Dinamik Programlama

Böl ve fethet gibi bir teknik tasarlayın.

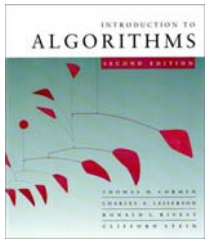
Örnek:En uzun ortak altdizi (LCS)

- İki tane, x[1 . . m]ve y[1 . . n] dizisi verilmiş, ikisinde de ortak olan en uzun altdiziyi bulun.

En uzun altdizi tek değildir.

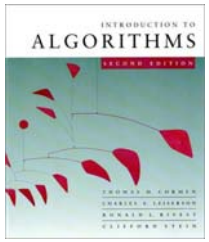


fonksiyonel gösterim, ama fonksiyon değil



Kaba-kuvvet algoritması

$x[1 \,..\, m]$'nin altdizisi mi yoksa aynı zamanda $y[1 \,..\, n]$'nin de altdizisi mi diye kontrol edelim.



Kaba-kuvvet algoritması

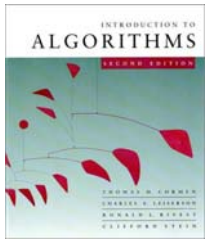
$x[1\ .\ .\ m]$' nin altdizisi mi yoksa aynı zamanda $y[1\ .\ .\ n]$' nin de altdizisi mi diye kontrol edelim.

Analizler

- Checking(kontrol) = her altdizi için $O(n)$zamanı.
- x' in altdizileri 2^m kadardır.(m' uzunluğunun her bir bit-vektörü, x'in farklı bir altdizisini belirler.)

  En kötü durum çalışma süresi = $O(n2^m)$

  = artan zaman.

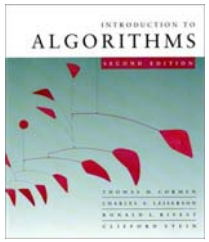


Daha iyi bir algoritma

Basitleştirme:

1. En uzun ortak altdizinin uzunluğuna bakalım.

2. Algoritmayı LCS' yi kendisi bulacak şekilde genişletin.



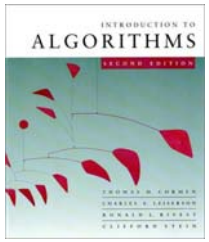
Daha iyi bir algoritma

Basitleştirme:

1.En uzun ortak altdizinin *uzunluğuna* bakalım.

2.Algoritmayı LCS' yi kendisi bulacak şekilde genişletin.

Gösterim: s dizisinin uzunluğunu $|s|$ ile belirtin.



Daha iyi bir algoritma

Basitleştirme:

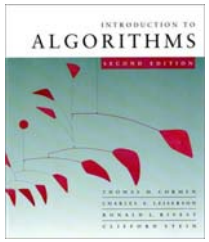
1. En uzun ortak altdizinin uzunluğuna bakalım.

2. Algoritmayı LCS' yi kendisi bulacak şekilde genişletin.

Gösterim: s dizisinin uzunluğunu $|s|$ ile belirtin.

Strateji: x ve y'nin öneklerini düşünün.

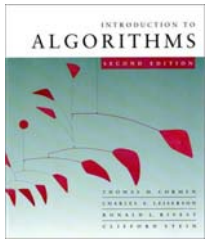
- Define(tanımlama) $c[i, j] = |\mathrm{LCS}(x[1\,..\,i], y[1\,..\,j])|$.
- Then(sonra), $c[m, n] = |\mathrm{LCS}(x, y)|$.



Özyinelemeli formülleme

Teorem.

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

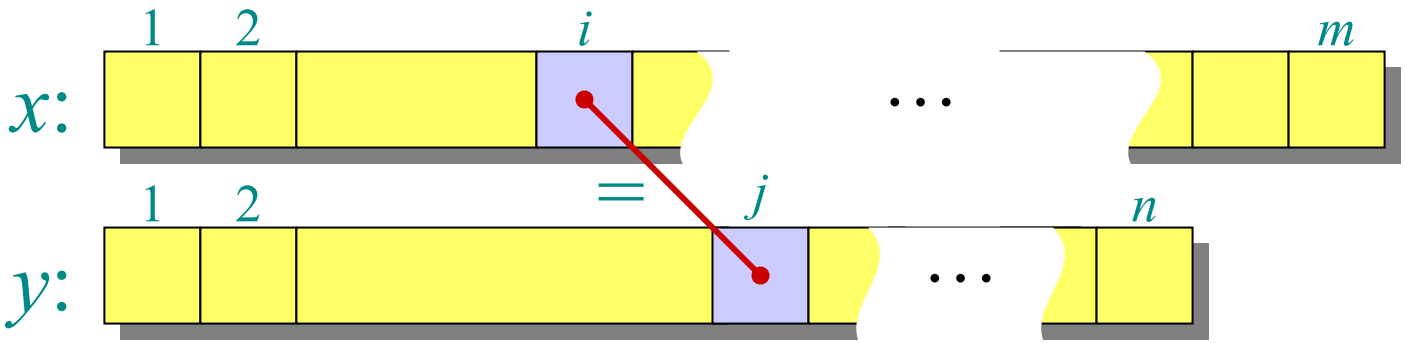


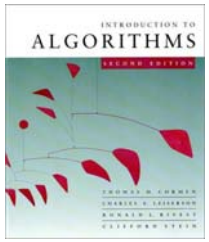
Özyinelemeli formülleme

Teorem.

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

Kanıt. Durum $x[i] = y[j]$:



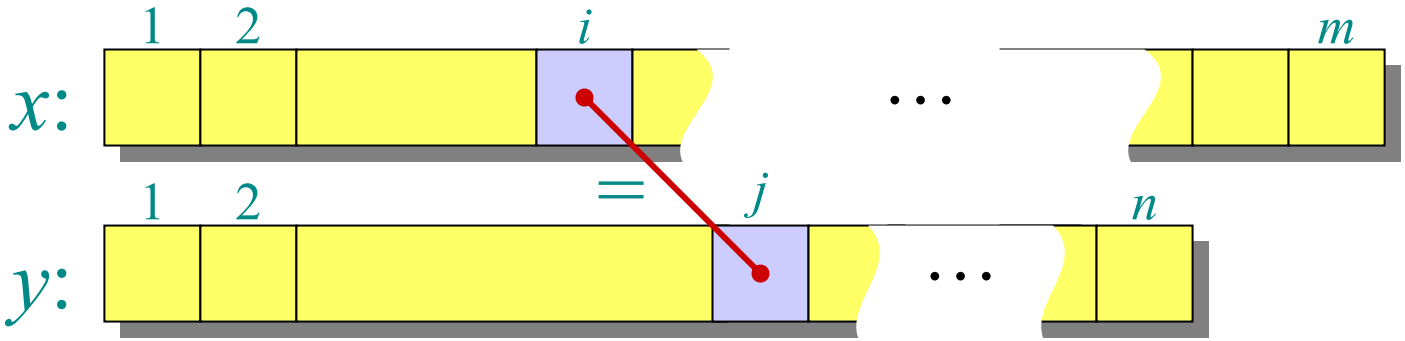


Recursive formulation

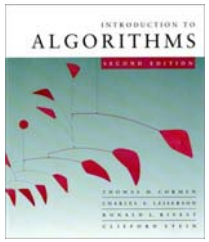
Theorem.

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

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



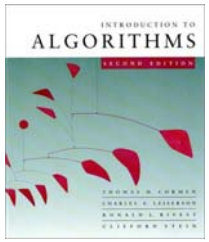
Let $z[1\,..\,k] = \text{LCS}(x[1\,..\,i], y[1\,..\,j])$, where $c[i,j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1\,..\,k-1]$ is CS of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$.



Proof (continued)

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

Suppose w is a longer CS of $x[1\,..\,i\text{–}1]$ and $y[1\,..\,j\text{–}1]$, that is, $|w| > k\text{–}1$. 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.



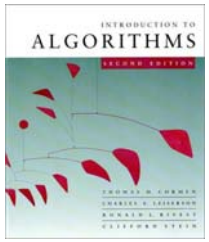
Proof (continued)

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

Suppose w is a longer CS of $x[1\,..\,i\text{–}1]$ and $y[1\,..\,j\text{–}1]$, that is, $|w| > k\text{–}1$. 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, $c[i\text{–}1, j\text{–}1] = k\text{–}1$, which implies that $c[i, j] = c[i\text{–}1, j\text{–}1] + 1$.

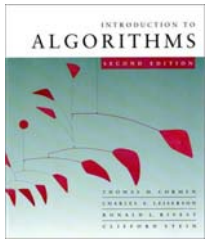
Other cases are similar. □



Dynamic-programming hallmark #1

Optimal substructure

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

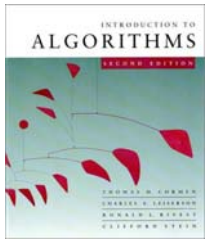


Dynamic-programming hallmark #1

Optimal substructure

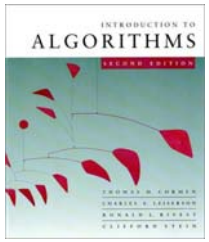
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.



Recursive algorithm for LCS

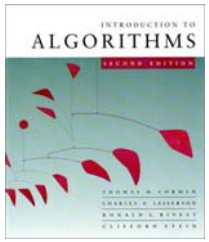
LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i–1, j–1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i–1, j)$,
                LCS$(x, y, i, j–1)\}$



Recursive algorithm for LCS

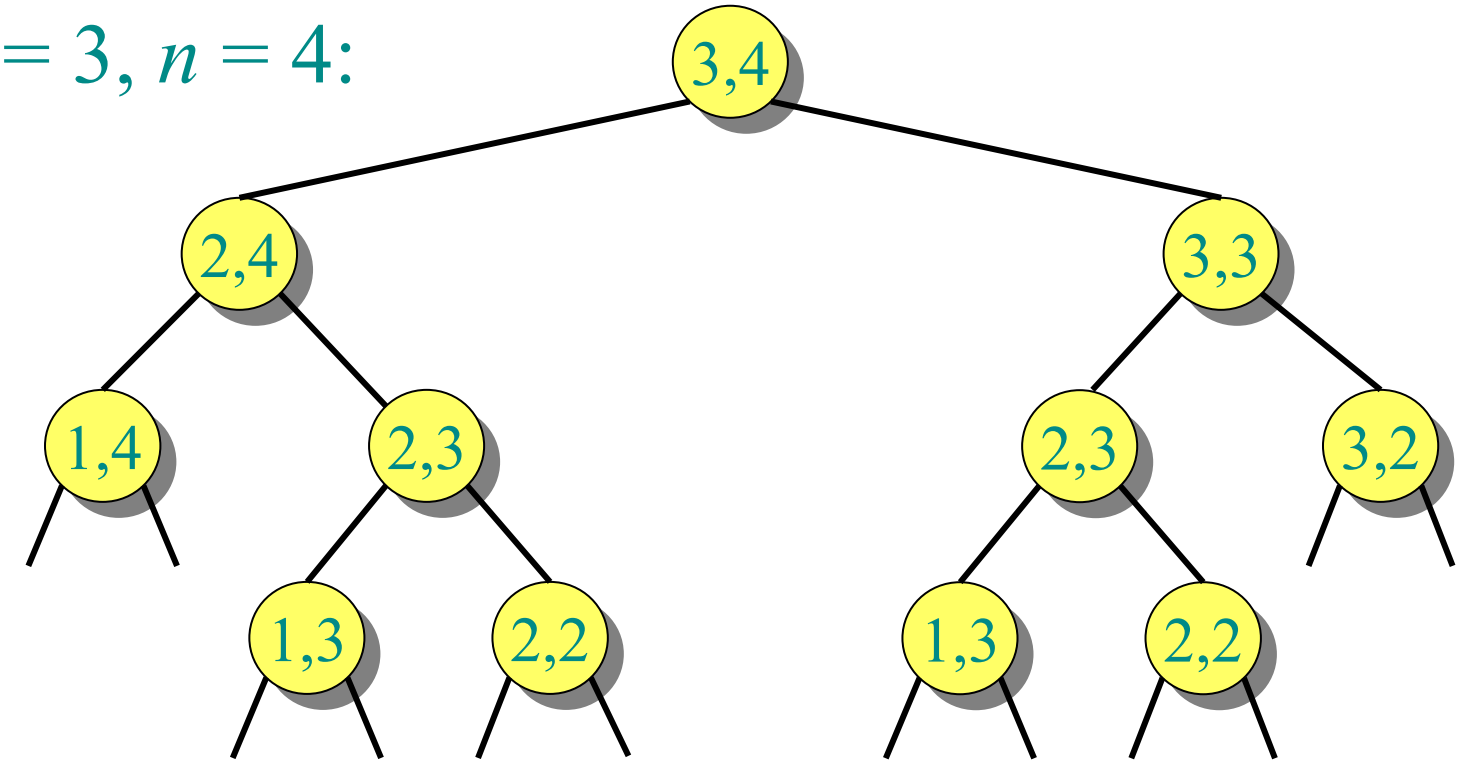
$$\begin{aligned}
&\text{LCS}(x, y, i, j) \\
&\quad \textbf{if } x[i] = y[j] \\
&\quad\quad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i{-}1, j{-}1) + 1 \\
&\quad\quad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i{-}1, j), \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{LCS}(x,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

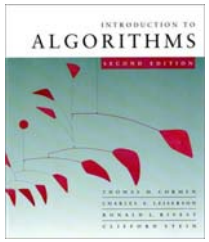
Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.



Recursion tree

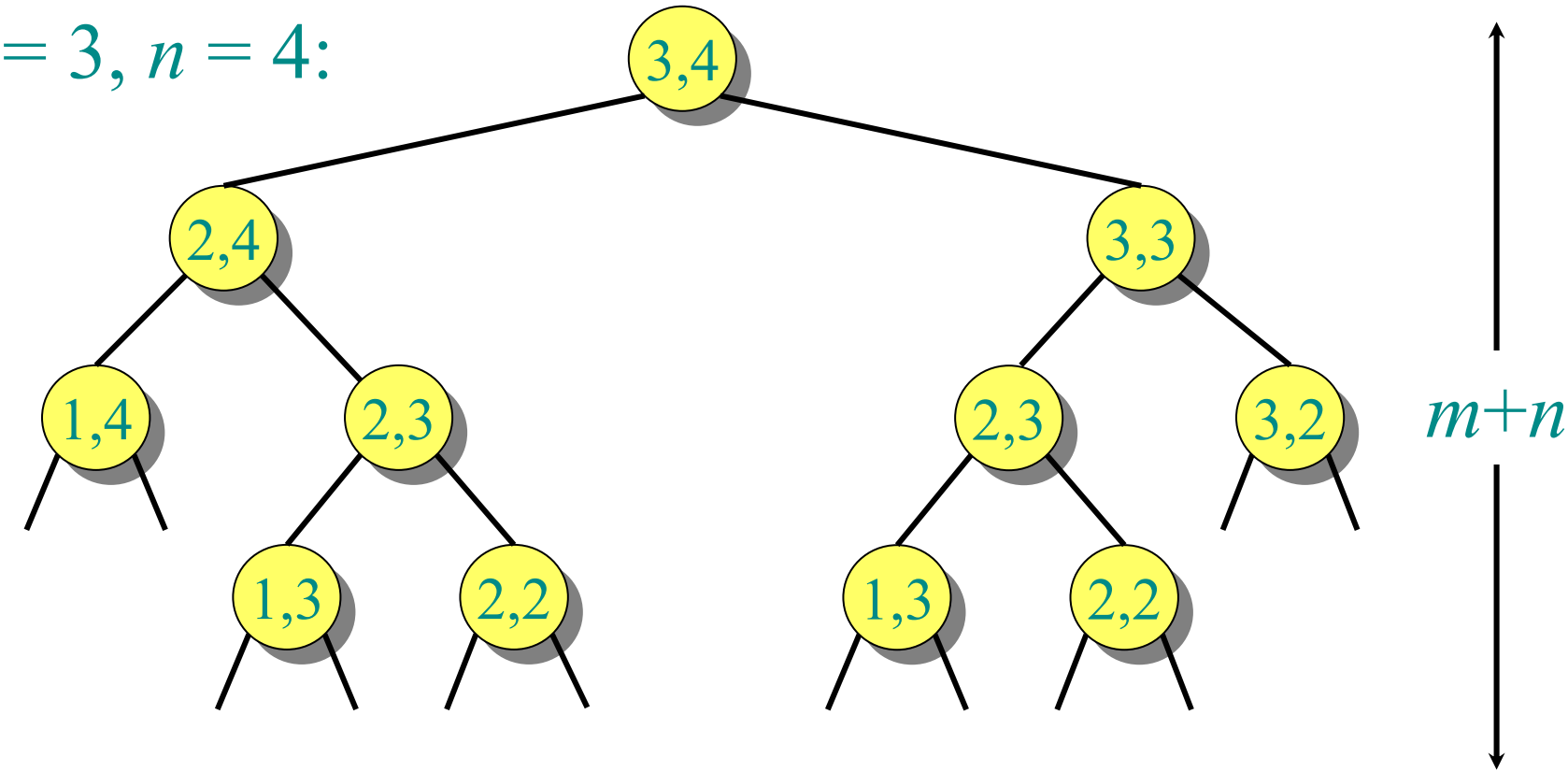
$m = 3$, $n = 4$:



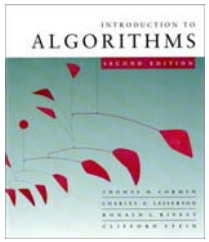


Recursion tree

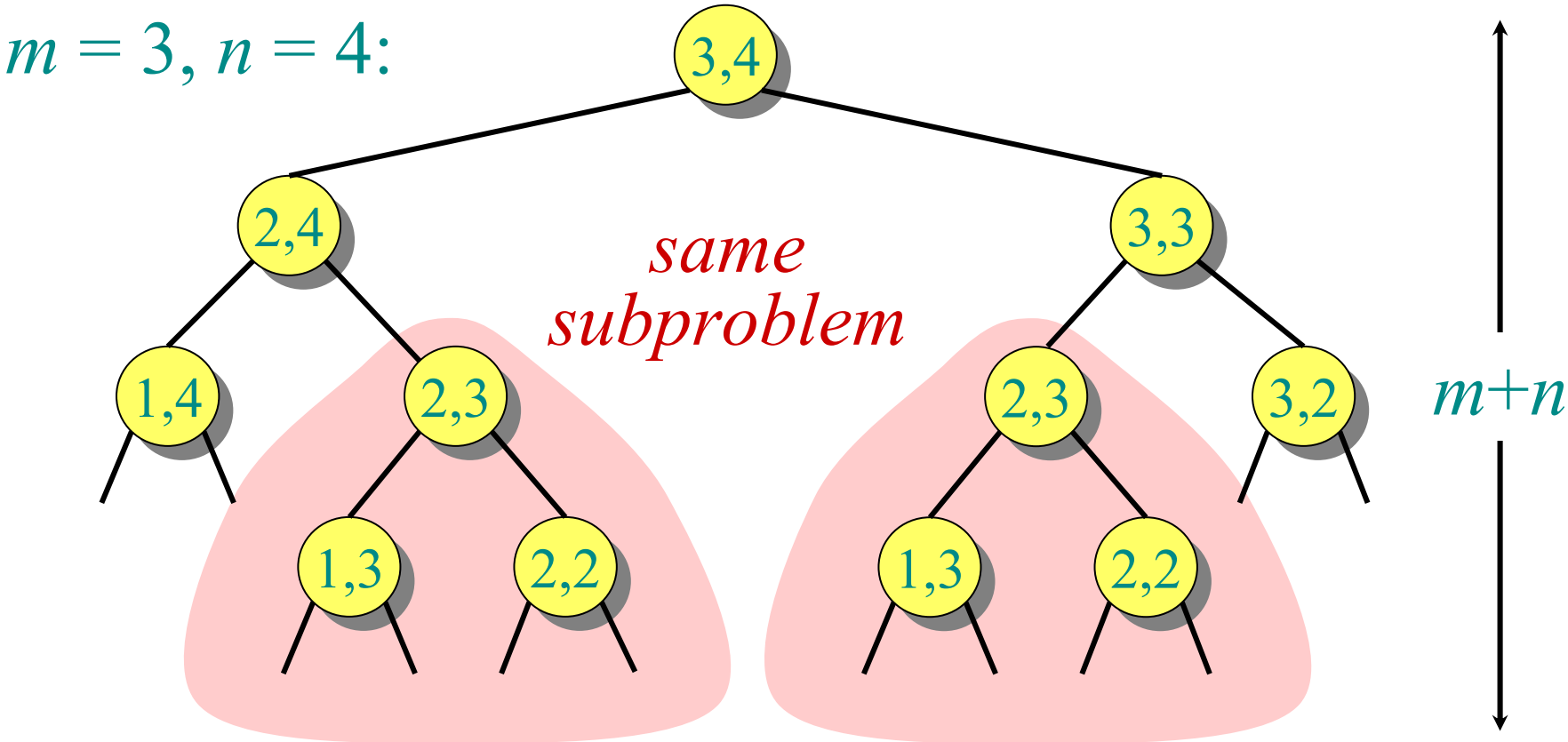
$m = 3, n = 4$:



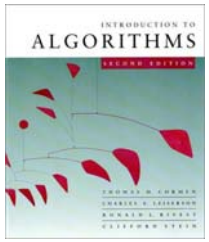
Height = $m + n$ $\Rightarrow$ work potentially exponential.



Recursion tree



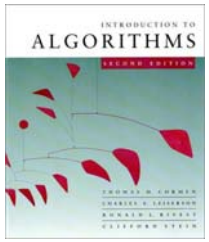
Height = $m + n$ ⇒ work potentially exponential, but we're solving subproblems already solved!



Dynamic-programming hallmark #2

Overlapping subproblems

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

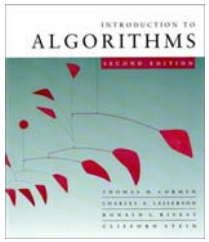


Dynamic-programming hallmark #2

Overlapping subproblems

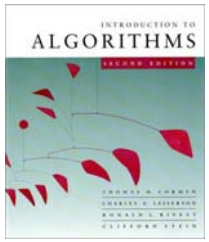
A recursive solution contains a "small" number of distinct subproblems repeated many times.

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.

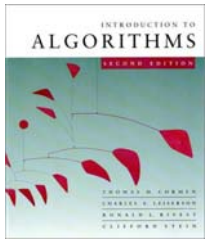


Memoization algorithm

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

LCS(x, y, i, j)
  if $c[i, j] = \text{NIL}$
    then if $x[i] = y[j]$
      then $c[i, j] \leftarrow \text{LCS}(x, y, i–1, j–1) + 1$
      else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i–1, j), \text{LCS}(x, y, i, j–1)\}$

} *same as before*



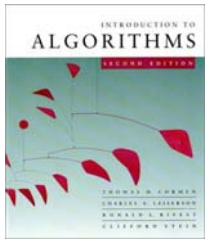
Memoization algorithm

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

```
LCS(x, y, i, j)
    if c[i, j] = NIL
        then if x[i] = y[j]
                then c[i, j] ← LCS(x, y, i–1, j–1) + 1
                else c[i, j] ← max{ LCS(x, y, i–1, j),
                                    LCS(x, y, i, j–1) }
```

same as before

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

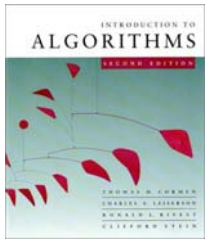


Dynamic-programming algorithm

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



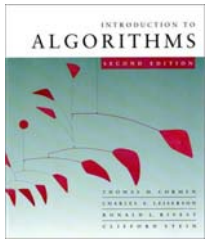
Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

		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

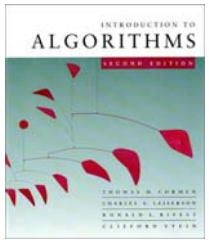
IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

		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