CMP461: Algorithms

Lecture 08: Dynamic Programming

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

Agenda

- Fibonacci Numbers
- Longest Common Subsequence
- Properties of Dynamic Programming
  - Optimal Substructure
  - Overlapping Subproblems

Acknowledgment
A lot of slides adapted from the slides of Erik Demaine and Charles Leiserson

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

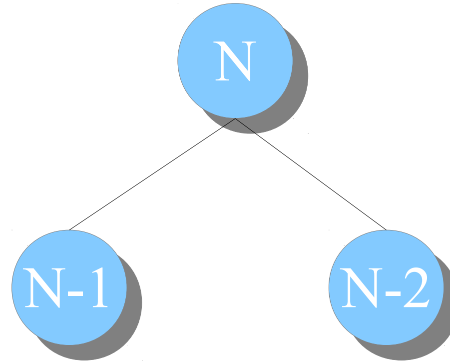
Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Recursive Solution

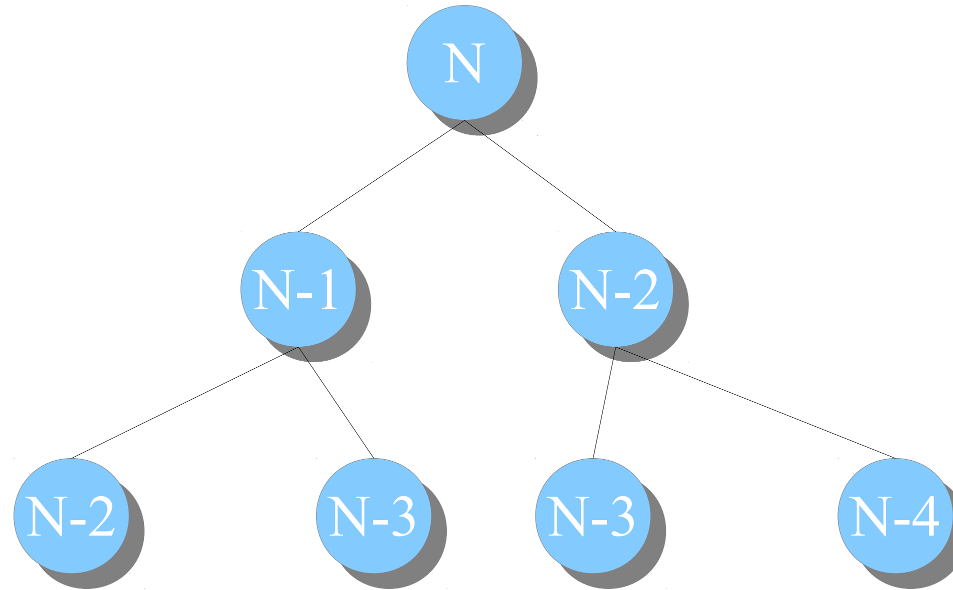
```
int fibonacci(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Recurrence: $T(n) = T(n-1) + T(n-2)$

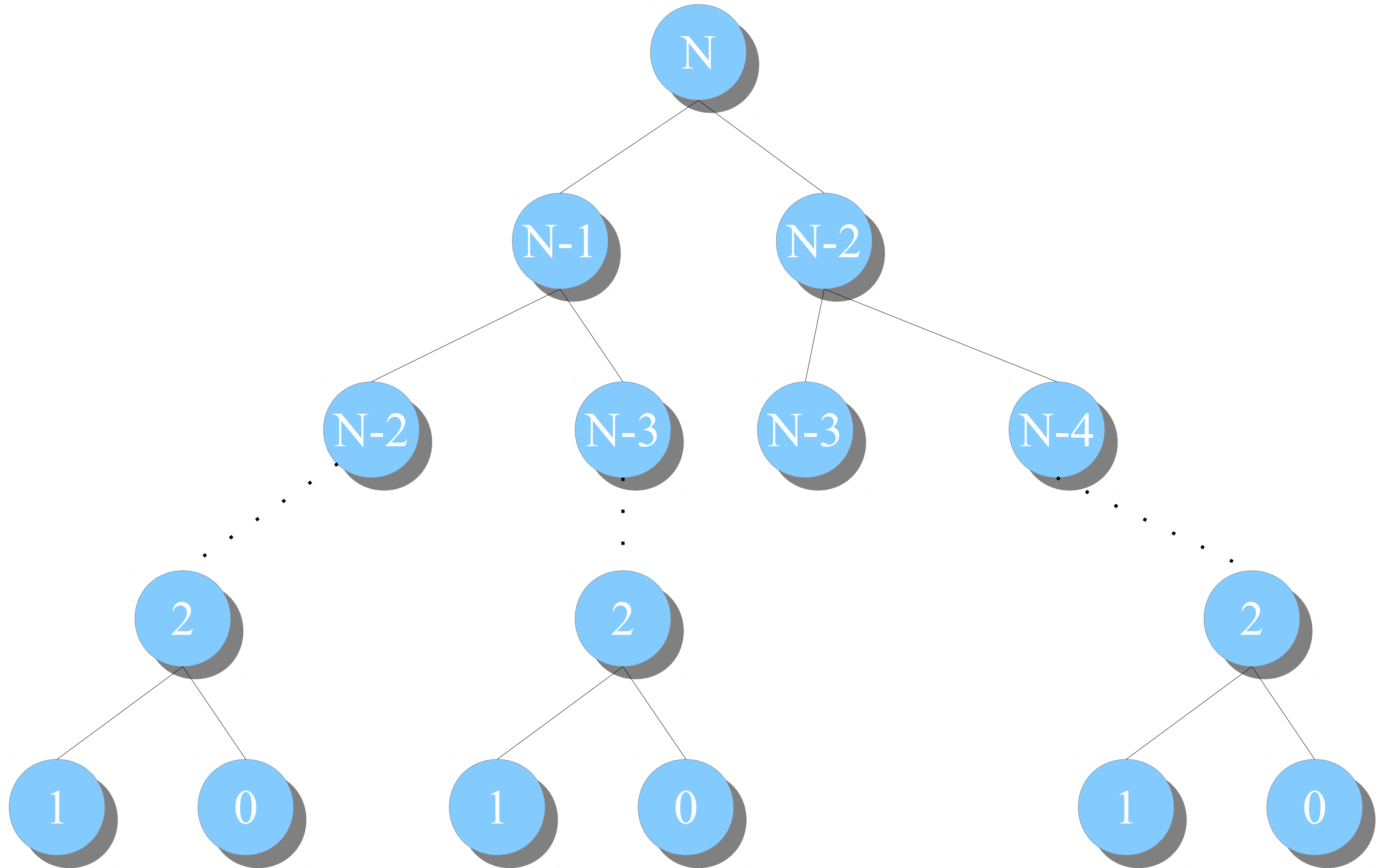
Fibonacci Numbers



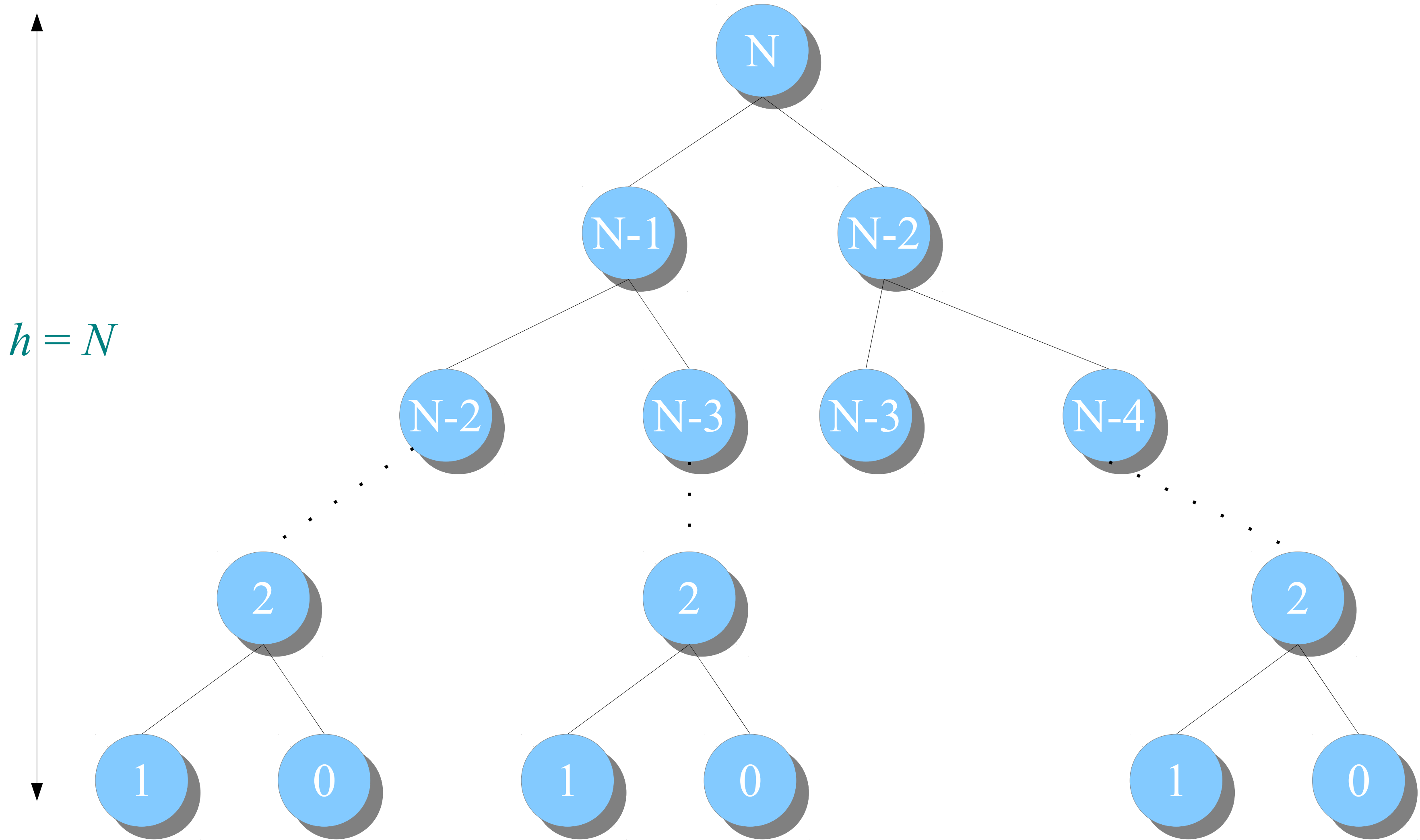
Fibonacci Numbers



Fibonacci Numbers



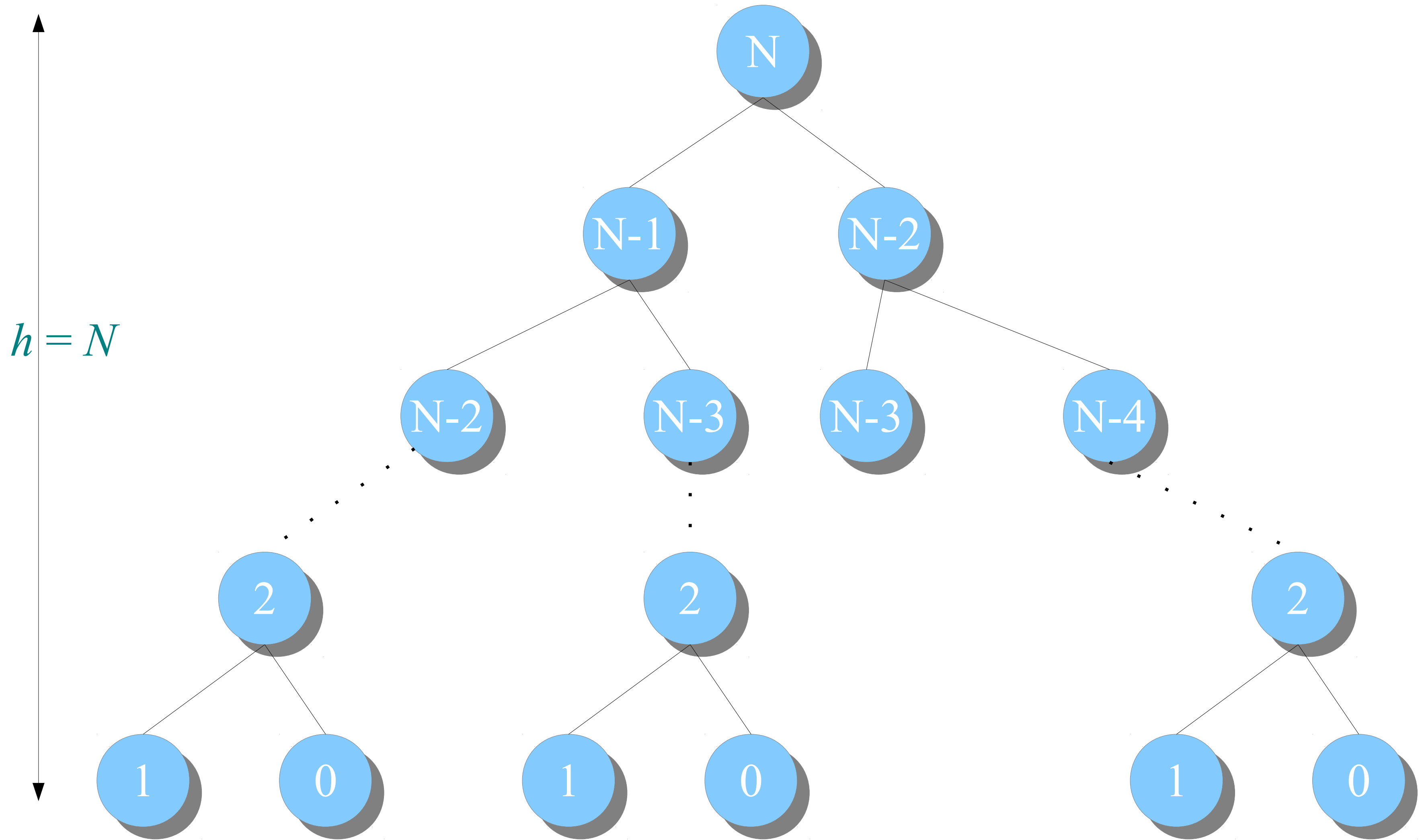
Fibonacci Numbers



Number of leaves $\leq 2^h$

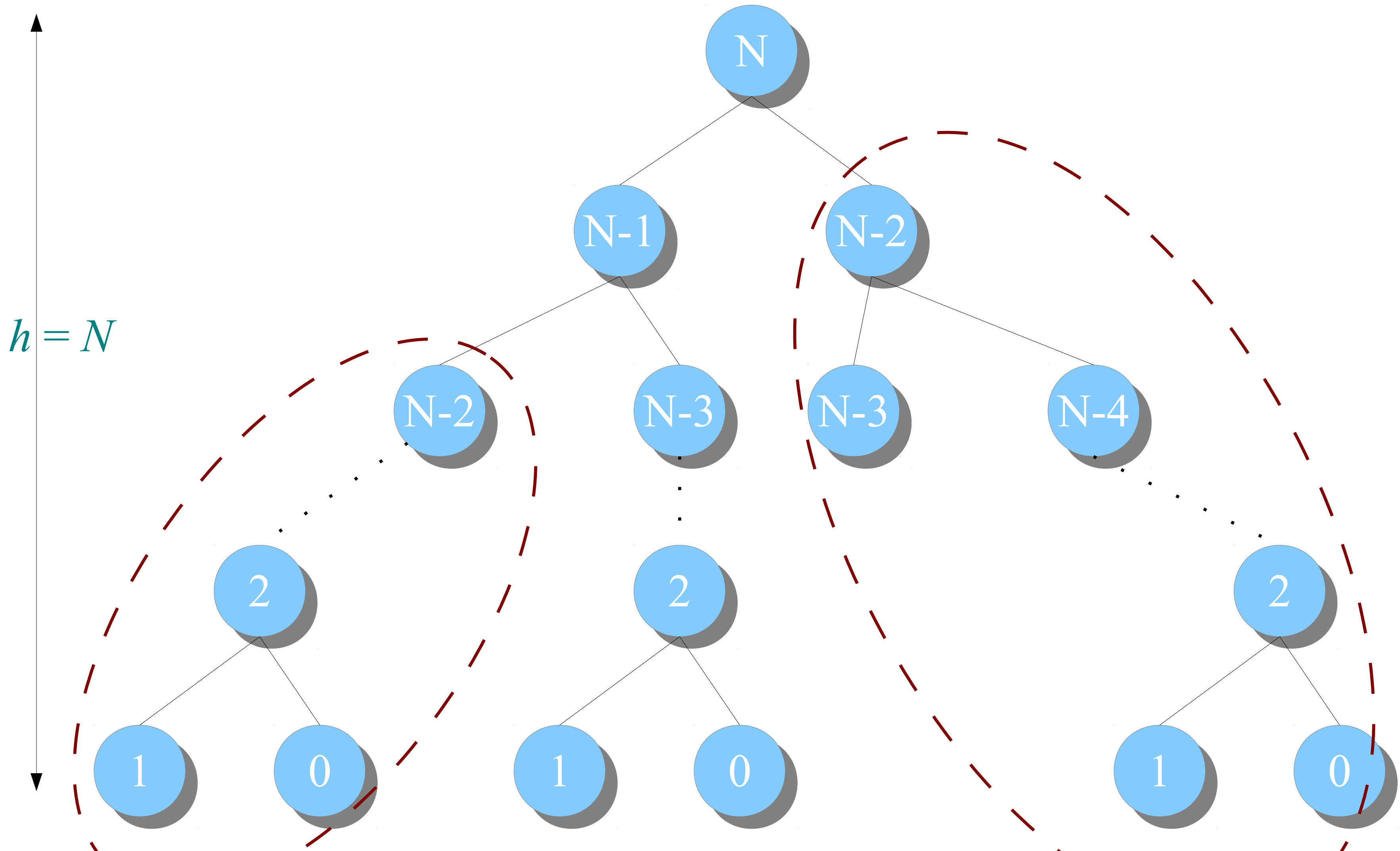
Number of nodes $\leq 2^{h+1} - 1$

Fibonacci Numbers



$T(n) = O(2^n)$ Exponential time !!

Fibonacci Numbers



What's the problem? Too many repeated subproblems!

Idea: solve subproblems once and store results.

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

Iterative Solution

```
int fibonacci(int n) {

  vector<int> fs(n+1);
  fs[0] = 0;
  fs[1] = 1;

  for (int i = 2; i <= n; ++i)
    fs[i] = fs[i-1] + fs[i-2];

  return fs[n];
}
```

Linear time !! $T(n) = \Theta(n)$ Trade off space for time.

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

Memoization

Compute values once and use many times

Recursive with time $\Theta(n)$

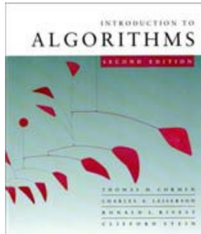
```
vector<int> fibs(MAX_N);
fill(fibs, -1);

int fibonacci(int n) {

  if (fibs[n] == -1) {
    if (n == 0) {
      fibs[0] = 0;
    } else if (n == 1) {
      fibs[1] = 1;
    } else {
      fibs[n] = fibonacci(n-2) + fibonacci(n-1);
    }
  }

  return fibs[n];
}
```

Fibonacci Numbers

- Key Observation:
  - Lots of small subproblems
  - Subproblems are repeated over and over
- Key Idea:
  - Solve each subproblem only once
  - Store solution and reuse many times

Dynamic Programming

- Design technique
- Can be used to:
  - Make recursive solutions more *efficient* by *memoization* or conversion to *iterative* solutions
  - Solve optimization problems *efficiently* e.g. find the shortest path, the best matrix parenthesization, the longest common subsequence, … etc.



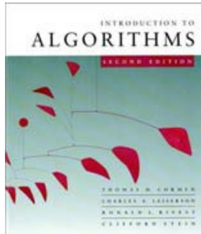
Dynamic programming

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.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



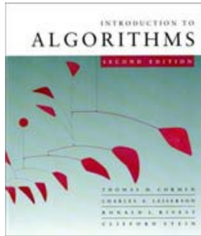
Dynamic programming

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"



Dynamic programming

Design technique, like divide-and-conquer.

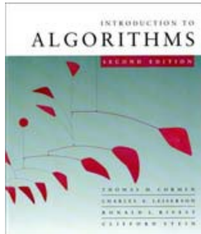
Example: ***Longest Common Subsequence (LCS)***

- 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



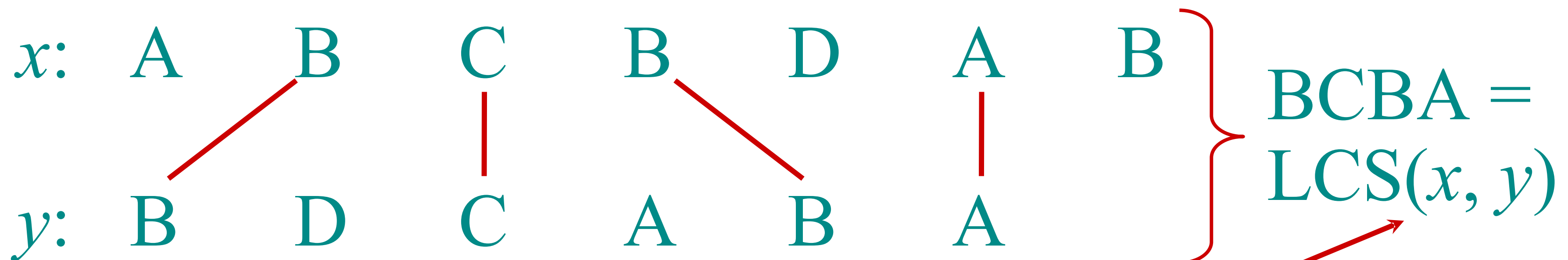
Dynamic programming

Design technique, like divide-and-conquer.

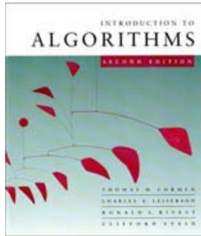
Example: *Longest Common Subsequence (LCS)*

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

"a" *not* "the"



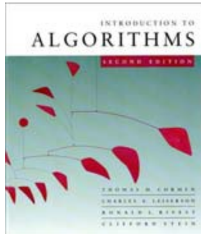
functional notation, but not a function



Brute-force LCS algorithm

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Brute-force LCS algorithm

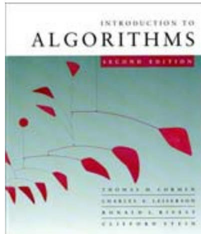
Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

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 $= O(n2^m)$
$= \text{exponential time.}$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

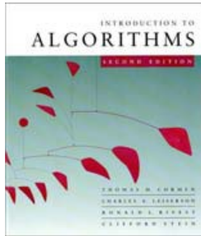


Towards a better algorithm

Simplification:

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



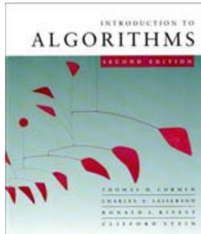
Towards a better algorithm

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 $|s|$.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Towards a better algorithm

Simplification:

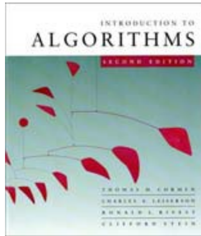
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

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

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

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

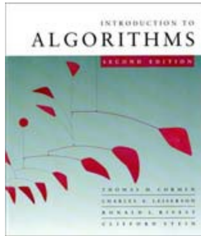


Recursive formulation

Theorem.

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

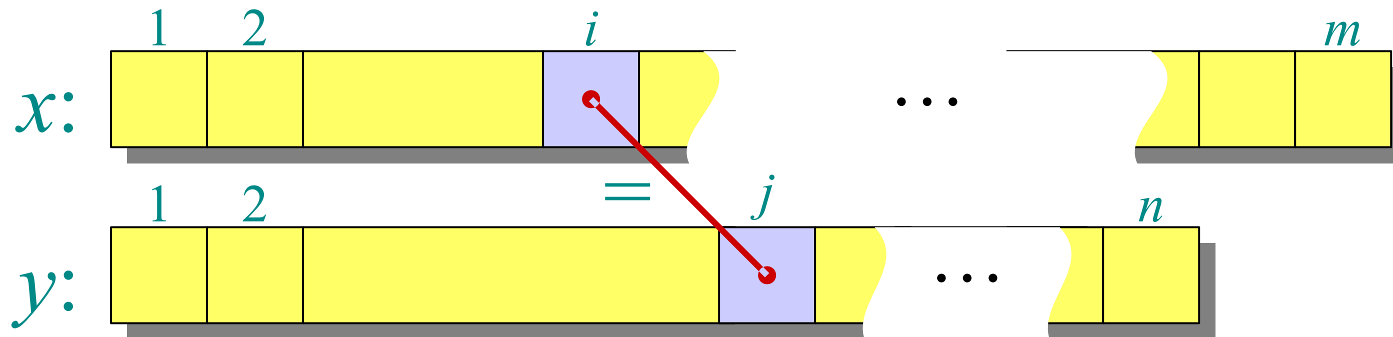


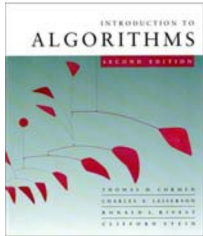
Recursive formulation

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



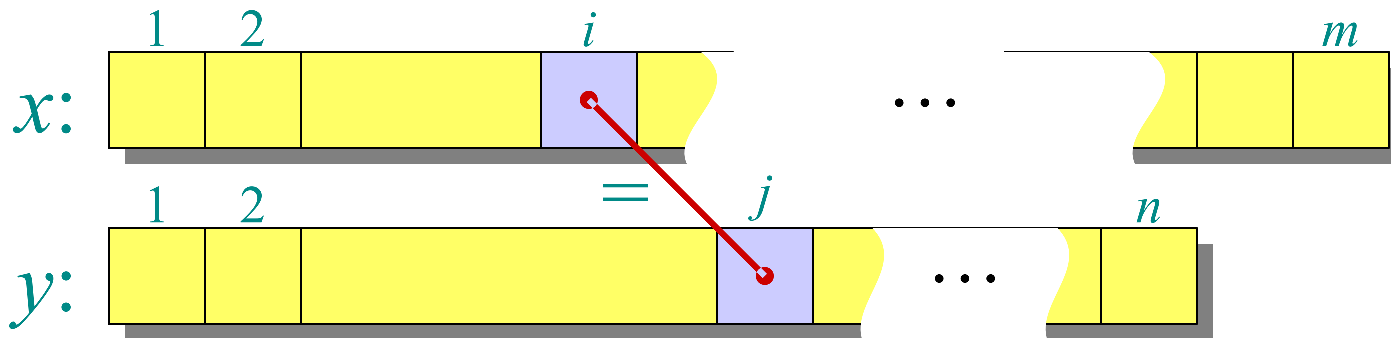


Recursive formulation

Theorem.

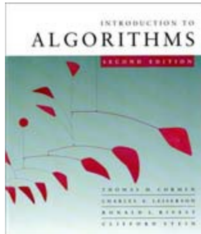
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

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 $y[1 \,..\, j-1]$.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

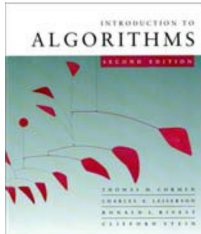


Proof (continued)

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

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



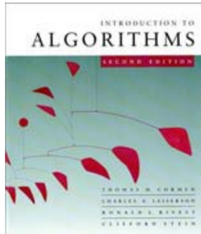
Proof (continued)

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

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. □

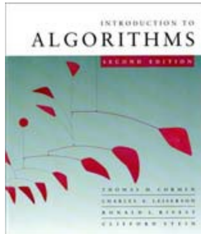


Dynamic-programming hallmark #1

Optimal substructure

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



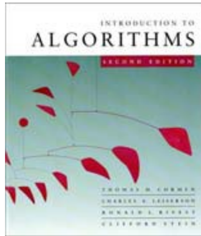
Dynamic-programming hallmark #1

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.

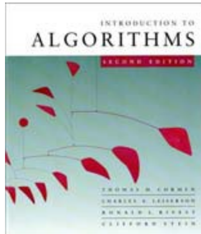
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recursive algorithm for LCS

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

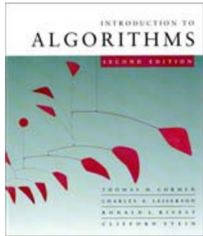


Recursive algorithm for LCS

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$(x, y, i, j-1)\}$

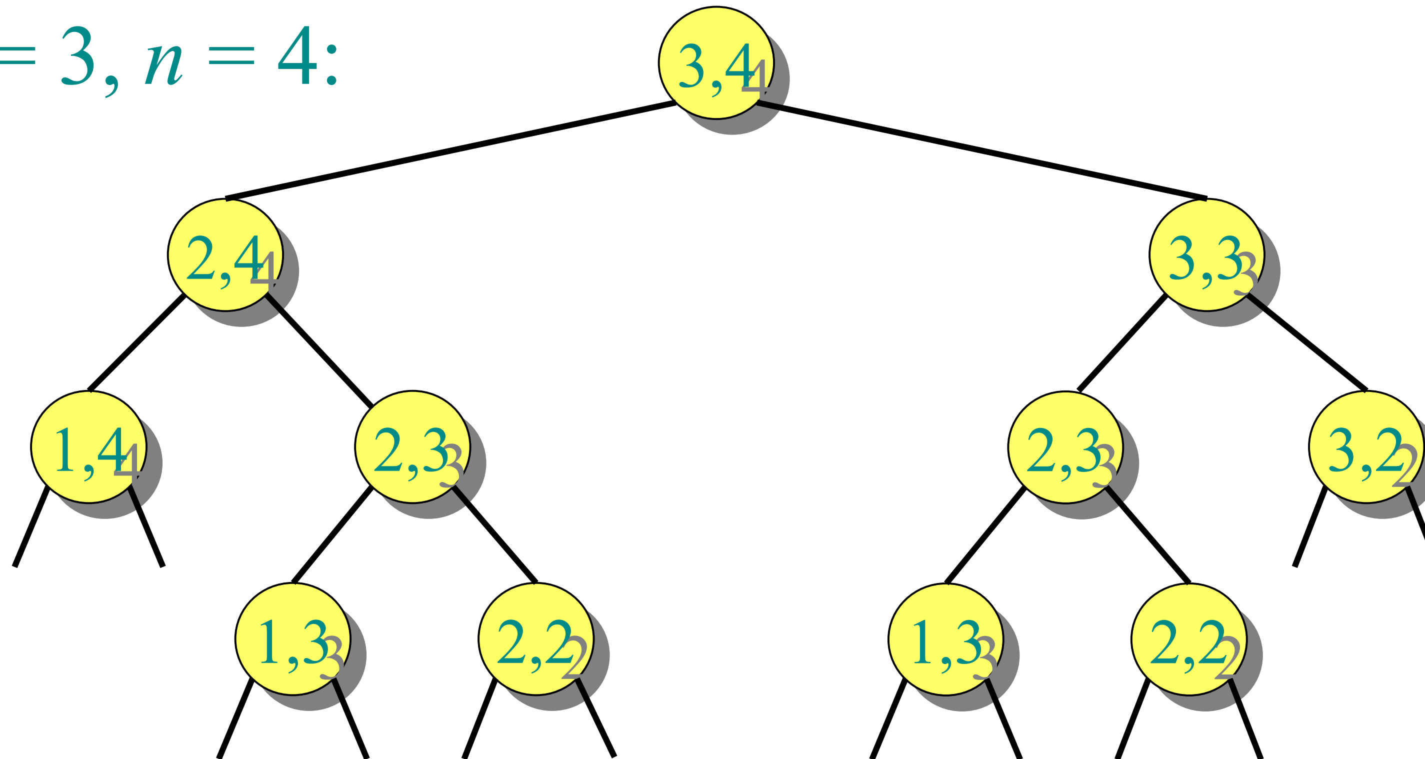
Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

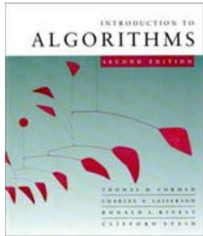
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recursion tree

$m = 3, n = 4$:





Recursion tree

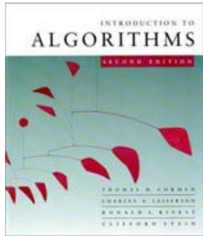
$m = 3, n = 4$:

3,4
2,4 3,3
1,4 2,3 2,3 3,2
1,3 2,2 1,3 2,2

$m+n$

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Recursion tree

$m = 3, n = 4$:

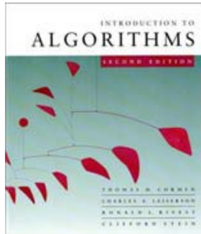
3,4
2,4
3,3
1,4
2,3
2,3
3,2
1,3
2,2
1,3
2,2

same subproblem

$m+n$

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

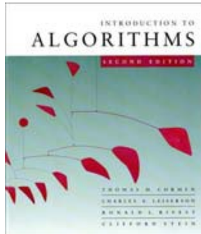


Dynamic-programming hallmark #2

Overlapping subproblems

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



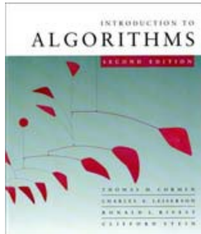
Dynamic-programming hallmark #2

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.

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Dynamic-programming hallmark #2

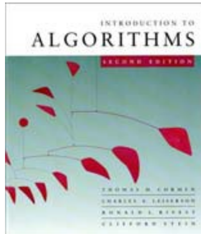
Overlapping subproblems

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.

All the possible combination of prefixes of x and y

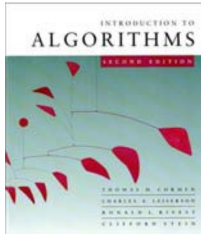
Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Memoization algorithm

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

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

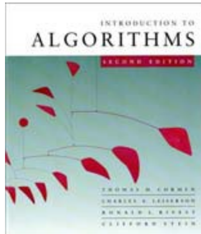


Memoization algorithm

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

same as before (applies to the lines from "then if" through the end)



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 (applies to the **then if** … **else** lines)

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

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Dynamic Programming Algorithm

Bottom up algorithm

```
LCS-Length(X, Y, m, n)
  for i = 1 to m
    c[i, 0] = 0
  for j = 1 to n
    c[0, j] = 0

  for i = 1 to m
    for j = 1 to n
      if X[i] == Y[j]
        c[i, j] = c[i-1, j-1] + 1
      elseif c[i-1, j] > c[i, j-1]
        c[i, j] = c[i-1, j]
      else
        c[i, j] = c[i, j-1]

  return c[m, n]
```

Initialize base cases

Loop in row-major order

Case 1

Case 2 & 3

Dynamic Programming Algorithm

```
for i = 1 to m
  c[i, 0] = 0

for j = 1 to n
  c[0, j] = 0
```

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

Dynamic Programming Algorithm

```
for i = 1 to m
  for j = 1 to n
    if X[i] == Y[j]
      c[i, j] = c[i-1, j-1] + 1
    elseif c[i-1, j] > c[i, j-1]
      c[i, j] = c[i-1, j]
    else
      c[i, j] = c[i, j-1]
```

		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							
C	0							
A	0							
B	0							
A	0							

Dynamic Programming Algorithm

```
for i = 1 to m
  for j = 1 to n
    if X[i] == Y[j]
      c[i, j] = c[i-1, j-1] + 1
    elseif c[i-1, j] > c[i, j-1]
      c[i, j] = c[i-1, j]
    else
      c[i, j] = c[i, j-1]
```

		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							
A	0							
B	0							
A	0							

Dynamic Programming Algorithm

```
for i = 1 to m
  for j = 1 to n
    if X[i] == Y[j]
      c[i, j] = c[i-1, j-1] + 1
    elseif c[i-1, j] > c[i, j-1]
      c[i, j] = c[i-1, j]
    else
      c[i, j] = c[i, j-1]
```

		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

```
for i = 1 to m
  for j = 1 to n
    if X[i] == Y[j]
      c[i, j] = c[i-1, j-1] + 1
    elseif c[i-1, j] > c[i, j-1]
      c[i, j] = c[i-1, j]
    else
      c[i, j] = c[i, j-1]
```

Time = $\Theta(mn)$

Space = $\Theta(mn)$

		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

Reconstruct the LCS by tracing backwards through the table

		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

Reconstruct the LCS by tracing backwards through the table

		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

Reconstruct the LCS by tracing backwards through the table

		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

Reconstruct the LCS by tracing backwards through the table

		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

Reconstruct the LCS by tracing backwards through the table

LCS = "BCBA"

		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

Edit Distance

The algorithm can be used to compute the Edit Distance between two strings i.e. number of insertions, deletions, and substitutions.

Alignment

AB*C*BDAB

*BDCAB*A*

Deletion
Insertion
Match

		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

Properties of Dynamic Programming

- Optimal Substructure
  - A solution to the problem is composed of solutions to smaller subproblems
  - At each step we need to examine more than one smaller subproblem (e.g. F_{n-1} and F_{n-2} for Fibonacci numbers or $c[i-1,j-1]$, $c[i-1,j]$ and $c[i,j-1]$ for LCS)
  - We need to combine their solutions to arrive at the solution to the current problem (e.g. addition or maximum)

Properties of Dynamic Programming

- Overlapping Subproblems
  - Usually there are exponentially many subproblems to solve in the recursive solution (e.g. Fibonacci or LCS)
  - However, these subproblems are not distinct. They keep repeating
  - Two ways to utilize this:
    - Memoization: modify the recursive algorithm by first looking up the value before making a recursive call (i.e. top-down strategy)
    - Dynamic Programming: by computing solutions to the small subproblems first and then combine them to arrive at a solution to the main problem (i.e. bottom-up strategy)

Recap

- Fibonacci Numbers
- Longest Common Subsequence
- Properties of Dynamic Programming
  - Optimal Substructure
  - Overlapping Subproblems
- Read Ch. 15 of **CLRS** for more examples (rod cutting, matrix chain multiplication)
- Next:
  - Greedy Algorithms