Proof (completed)

Q. How many \( h \)'s cause \( x \) and \( y \) to collide?

A. There are \( m \) choices for each of \( a_1, a_2, \ldots, a_r \), but once these are chosen, exactly one choice for \( a_0 \) causes \( x \) and \( y \) to collide, namely

\[
\begin{align*}
\sum_{i=1}^{r} (-1)^{i+1} & \cdot (x_i - y_i)^{-1} = \\
\left[ \begin{array}{c} x_0 - y_0 \\
\vdots \\
 x_r - y_r \\
\end{array} \right] \\
\left[ \begin{array}{c} a_1 \\
\vdots \\
 a_r \\
\end{array} \right] \\
\end{align*}
\]

Thus, the number of \( h \)'s that cause \( x \) and \( y \) to collide is

\[
 m^r = \frac{|H|}{m}.
\]

October 5, 2005

Copyright \( \odot 2001-5 \) by Erik D. Demaine and Charles E. Leiserson

L7.15
Dynamic programming

*Design technique, like divide-and-conquer.*

**Example: Longest Common Subsequence (LCS)**
- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.
Dynamic programming

*Design technique, like divide-and-conquer.*

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

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

  “a” not “the”

\[ x: \quad A \quad B \quad C \quad B \quad D \quad A \quad B \]

\[ y: \quad B \quad D \quad C \quad A \quad B \quad A \]
Dynamic programming

Design technique, like divide-and-conquer.

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

- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, 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 $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$. 
Brute-force LCS algorithm

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

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 = $O(n2^m)$

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

Theorem.

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

\[
\begin{array}{cccccc}
  & 1 & 2 & i & \ldots & m \\
  x: & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} \\
  & 1 & 2 & = & j & \ldots \\
  y: & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} & \boxed{\phantom{1}} \\
  & 1 & 2 & \ldots & \phantom{1} & \phantom{1}
\end{array}
\]
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 \( z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j]) \), where \( c[i, j] = k \). Then, \( z[k] = x[i] \), or else \( z \) could be extended. Thus, \( z[1 \ldots k-1] \) is CS of \( x[1 \ldots i-1] \) and \( y[1 \ldots j-1] \).
Proof (continued)

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

**Claim:** $z[1 \ldots k-1] = \text{LCS}(x[1 \ldots i-1], y[1 \ldots j-1])$. Suppose $w$ is a longer CS of $x[1 \ldots i-1]$ and $y[1 \ldots j-1]$, that is, $|w| > k-1$. Then, **cut and paste:** $w \| z[k]$ ($w$ concatenated with $z[k]$) is a common subsequence of $x[1 \ldots i]$ and $y[1 \ldots j]$ 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 $z = \text{LCS}(x, y)$, then any prefix of $z$ is an LCS of a prefix of $x$ and a prefix of $y$. 
Recursive algorithm for LCS

\[
\begin{align*}
\text{LCS}(x, y, i, j) \\
\quad \text{if } x[i] = y[j] \\
\qquad \text{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
\quad \text{else } c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \\
\qquad \text{LCS}(x, y, i, j-1) \}\n\end{align*}
\]
Recursive algorithm for LCS

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

\textbf{Worst-case: } x[i] \neq y[j], \text{ in which case the algorithm evaluates two subproblems, each with only one parameter decremented.}
Recursion tree

\[ m = 3, \quad n = 4: \]

```
3,4
/   \
2,4   3,3
|     |
1,4   2,3  3,2
|     |
1,3   2,2  2,2
```

\[ 1,3 \quad 2,2 \]

```
Recursion tree

$m = 3, n = 4$:

\[ \text{Height} = m + n \Rightarrow \text{work potentially exponential.} \]
Recursion tree

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

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.

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

*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.

\[
\text{LCS}(x, y, i, j) = \begin{cases} 
\text{NIL} & \text{if } c[i, j] = \text{NIL} \\
\text{same as before} & \text{if } x[i] = y[j] \\
\text{else } c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j-1) + 1, \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \} 
\end{cases}
\]

Time = $\Theta(mn)$ = constant work per table entry.
Space = $\Omega(mn)$. 
**Dynamic-programming algorithm**

**Idea:**
Compute the table bottom-up.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**Dynamic-programming algorithm**

**Idea:**
Compute the table bottom-up.

Time = $\Theta(mn)$. 

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**Dynamic-programming algorithm**

**Idea:**
Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.
**Dynamic-programming algorithm**

**IDEA:**
Compute the table bottom-up.

**Time =** $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

**Space =** $\Theta(mn)$.

**Exercise:**
$\mathcal{O}(\min\{m, n\})$