Design and Analysis of Algorithms

CSE 5311

Lecture 15 Dynamic Programming

Song Jiang, Ph.D.
Department of Computer Science and Engineering
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”

  $x$: 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 $x[1 \ldots m]$ 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 $= O(n2^m) = \text{exponential time}$. 
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 \( c[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])| \).
- 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 \( x[i] = y[j] \):

- 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 \(z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j])\), 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

\[ \text{LCS}(x, y, i, j) \]

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

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

**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

$m = 3, n = 4$:

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.

\[
\text{LCS}(x, y, i, j) \begin{align*}
\text{if } & c[i, j] = \text{NIL} \\
\text{then if } & x[i] = y[j] \\
\text{then } & c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
\text{else } & c[i, j] \leftarrow \max \left\{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \right\}
\end{align*}
\]

*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.

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

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

*same as before*
**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>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>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>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</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 = $\Theta(mn)$. 

![Dynamic-programming algorithm table](image)
**Dynamic-programming algorithm**

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

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

<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>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</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.

Space = $\Theta(mn)$.

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