Week 7.1, Monday, Sept 30

Tentative Date for Final Exam: Thursday, December 12
(7-9PM STEW 130)
Announcements

- Midterm grading in progress...
Dynamic Programming

- Longest Common Subsequence
- Optimal substructure
- Overlapping subproblems
- Sequence alignment (Edit Dist.)
Problem 4: *Longest Common Subsequence (LCS)*

*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

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

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.

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

“a” not “the”
Dynamic programming

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.

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

"a" not "the"

$BCBA = \text{LCS}(x, y)$

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 $|s|$. 
Clicker Question

Longest Common Subsequence (LCS)
• Input: two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$
• Output: length of longest common subsequence

Let $OPT(k)$ be longest subsequence common to both $x[1 \ldots k]$ and $y[1 \ldots n]$,

Which statement is true?

A. $OPT(k) = OPT(k - 1) + 1$ if $x[k] = y[n]$
B. $OPT(k) = OPT(k - 1)$ if $x[k] \neq y[n]$
C. Both A and B are true
D. Neither claim is true
E. Don’t pick this choice it is incorrect! 😊
Let $OPT(k)$ be the (length of) longest subsequence common to both $x[1 \ldots k]$ and $y[1 \ldots n]$,

Which statement is true?

A. $OPT(k) = OPT(k - 1) + 1$ if $x[k] = y[n]$

Counterexample: $x=$“aa”, $y=$“bbba” $\Rightarrow$ Opt(2) = 1 and Opt(1) = 1

Opt(2) $\neq$ Opt(1) + 1

B. $OPT(k) = OPT(k - 1)$ if $x[k] \neq y[n]$

Counterexample: $x=$“bb”, $y=$“bba” $\Rightarrow$ Opt(2) = 2 and Opt(1) = 1

Opt(2) $\neq$ Opt(1)

C. Both A and B are true

D. Neither claim is true

E. Don’t pick this choice it is incorrect! 😊
Stuck?

Let OPT(k) be (length of) longest subsequence common to both $x[1 \ldots k]$ and $y[1 \ldots n]$

We can try to develop a recurrence for Opt(k) but we will fail!

Why? There are not enough sub-problems to develop a recurrence…

(Sub-problems always try to match $x[1 \ldots i]$ with all of $y[1 \ldots n]$)

Solution: Introduce more sub-problems!

Let OPT(i,j) be (length of) longest subsequence common to both $x[1 \ldots i]$ and $y[1 \ldots j]$
Let \( \text{OPT}(i,j) \) be (length of) longest subsequence common to both 
\[ x[1 \ldots i] \] and \[ y[1 \ldots j] \]

**Case 1:** \( x[i]=y[j] \)

\[ \Rightarrow \text{OPT}(i,j) = 1 + \text{OPT}(i-1,j-1) \]

**Case 2:** \( x[i] \neq y[j] \)

\[ \Rightarrow \text{OPT}(i,j) = \max\{\text{OPT}(i-1,j), \text{OPT}(i,j-1)\} \]

**Base Cases:** \( \text{OPT}(0,j) = 0 \) and \( \text{OPT}(i,0) = 0 \)
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 $\text{OPT}[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])|$. 
• Then, $\text{OPT}[m, n] = |\text{LCS}(x, y)|$. 
Recursive formulation

Theorem.

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

**Theorem.**

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

**Proof.** Case \( x[i] = y[j] \):

\[ \begin{align*}
\text{x:} & \quad 1 \quad 2 \quad \cdots & \quad i \\
\text{y:} & \quad 1 \quad 2 \quad \cdots & \quad j
\end{align*} \]
Recursive formulation

Theorem.

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

Proof. Case $x[i] = y[j]$: Let $z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j])$. Then, $z[k] = x[i]$, or else $z$ could be extended. Thus, $z[1 \ldots k-1]$ is Common Substring of $x[1 \ldots i-1]$ and $y[1 \ldots j-1]$. 

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
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 1: $z[1 \ldots k-1] = LCS(x[1 \ldots i-1], y[1 \ldots j-1])$.
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, if $x[i] = y[j]$, we have $OPT[i-1, j-1] = k-1$, which implies that $OPT[i, j] = OPT[i-1, j-1] + 1$. 
Recursive formulation

**Theorem.**

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

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

Let $z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j])$. If $z[k] \neq y[j]$ then $z[1 \ldots k]$ is a Common Substring of $x[1 \ldots i]$ and $y[1 \ldots j-1]$. 
Proof (continued)

Claim 3: $z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j-1])$.
Suppose $w$ is a longer CS of $x[1 \ldots i]$ and $y[1 \ldots j-1]$, that is, $|w| > k$. Then $w$ is a common subsequence of $x[1 \ldots i]$ and $y[1 \ldots j]$ with $|w| > k$. Contradiction, proving the claim.

Thus, $\text{OPT}[i, j-1] = k$, which implies that $\text{OPT}[i, j] = \text{OPT}[i, j-1]$. Similarly, if $z[k] = y[j]$, then $z[k] \neq x[i]$ and $z = \text{LCS}(x[1 \ldots i-1], y[1 \ldots j])$. It follows that

$$\text{OPT}[i, j] = \max \{ \text{OPT}[i-1, j], \text{OPT}[i, j-1] \}$$
Finding the LCS

Base Case: \( \text{OPT}[0,j] = \text{OPT}[i,0] = 0 \)

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

\textbf{Step 1:} Fill in the DP table and compute \( \text{OPT}[i, j] \) for all \( i \leq m \) and \( j \leq n \) using above recurrence

\textbf{Step 2:} Backtrack to construct a common subsequence \( z \) of length \( k=\text{OPT}[m,n] \)

- If \( \text{OPT}[m,n]=\text{OPT}[m-1, n-1] + 1 \) then set \( z[k] = x[m] \) and recurse with \( x[1,\ldots,m-1], y[1,\ldots,n-1] \)
- If \( \text{OPT}[m,n]= \text{OPT}[m-1, n] \) then recurse with \( x[1,\ldots,m-1], y[1,\ldots,n] \)
- If \( \text{OPT}[m,n]= \text{OPT}[m, n-1] \) then recurse with \( x[1,\ldots,m], y[1,\ldots,n-1] \)
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 \( y \).
6.4 Knapsack Problem
Knapsack Problem

Knapsack problem.

- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Knapsack Problem

Knapsack problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

W = 11

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \implies \) greedy not optimal.
Knapsack Problem (Greedy)

Knapsack problem.
- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of $W$ kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \{ 3, 4 \} has value 40.

$$ W = 11 - 7 = 4 $$

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.67</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.
Ex: \{ 5, 2, 1 \} achieves only value = 35 $\Rightarrow$ greedy not optimal.
Knapsack problem.

- Given $n$ objects and a "knapsack."
- Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of $W$ kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: $\{3, 4\}$ has value 40.

$W = 11 - 7 = 4$

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.66..</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

**Greedy:** repeatedly add item with maximum ratio $v_i / w_i$.

Ex: $\{5, 2, 1\}$ achieves only value $= 35 \Rightarrow$ greedy not optimal.
Knapsack Problem (Greedy)

Knapsack problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

\( W = 4 - 2 = 2 \)

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Knapsack problem.
  - Given \( n \) objects and a "knapsack."
  - Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
  - Knapsack has capacity of \( W \) kilograms.
  - Goal: fill knapsack so as to maximize total value.

Ex: \( \{3, 4\} \) has value 40.

\[
\begin{array}{c|c|c|c}
# & value & weight & ratio \\
1 & 1 & 1 & 1 \\
2 & 6 & 2 & 3 \\
3 & 18 & 5 & 3.6 \\
4 & 22 & 6 & 3.66.. \\
5 & 28 & 7 & 4 \\
\end{array}
\]

\[
W = 4 - 2 = 2
\]

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{5, 2, 1\} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Dynamic Programming: False Start

Def. $OPT(i) = \text{max profit subset of items } 1, \ldots, i$.

- **Case 1**: $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$

- **Case 2**: $OPT$ selects item $i$.
  - accepting item $i$ does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before $i$, we don't even know if we have enough room for $i$

**Conclusion.** Need more sub-problems!