Lecture 10

Today: Dynamic Programming

- Paradigm for design of algorithms
- Example: Longest Common Subsequence (LCS)

Based on:
- Optimal Substructure Property
- Memo-ization
Today's example:

Longest Common Subsequence (LCS)

Given: Two (character) sequences

\[ X[1..m] \text{ and } Y[1..n] \]

Find: Longest common subsequence

Example:

\[ X = A B C \underbrace{B} D A B \]
\[ Y = B D C A B B A \]

Common Subsequence: \[ Z[1..K] \]

if exist \[ 1 \leq i_1 \leq i_2 \leq i_3 \leq \ldots \leq i_K \leq m \]

and \[ 1 \leq j_1 \leq j_2 \leq j_3 \leq \ldots \leq j_K \leq n \]

such that \[ X[i_l] = Y[j_l] \text{ for all } 1 \leq l \leq K \]

We need to find longest such common subsequence

Very Important Problem:

Similarity of web pages, DNA sequences, misspelled words, homework sets, ...
Back to example:

\[ k = 4 \]
\[ z = BCBA \]
\[ i_1 = 2 \quad i_2 = 3 \quad i_3 = 4 \quad i_4 = 6 \]
\[ j_1 = 1 \quad j_2 = 3 \quad j_3 = 5 \quad j_4 = 7 \]

but is this the largest?

An algorithm:

1. enumerate all possible subsequences of \( x \)
2. for each subsequence of \( x \), check if it appears as a subsequence of \( y \)
3. keep track of longest such sequence found

What is the runtime?
Some good news:

Step 2 can be done in linear time $O(m+n)$ using "two-finger" algorithm.

\[ \begin{align*}
& \text{If } z[i] = y[j], \text{ then } i^{++}, j^{++} \\
& \text{else } j^{++}
\end{align*} \]

Bad news:

number of possible $z$'s is $2^m$ 😞

Need to do better....
Dynamic Programming

- Used when optimizing and optimum solution shows obvious "structure"
called "optimal substructure property"

- Steps in designing algorithm:
  - try to capture structure mathematically
  - use above to design recursive algorithm
  - "Memo-ize" the recursive algorithm to get efficiency

Applying to LCS:

Need to ask:
- Structure in optimum of LCS?
- recursive algorithm?
- Memo-ize what?
LCS Structure

- Suppose we are told \( k \) and \( i_k \), does this simplify our task?

\[
x \ \\
y
\]

Let \( l \) be largest index such that \( y[l] = x[i_k] \) then

we can assume (without loss of generality) \( j_k = l \)

But can we say anything about

\[
i_1, i_2, \ldots, i_{k-1} \quad ? \quad j_1, \ldots, j_{k-1}
\]

\[
\text{LCS}(x[1, \ldots, i_{k-1}], y[1, \ldots, (j_k-1)])
\]

Recursive Structure! 😊
Main Insight \[ \text{LCS}(X[1..m], Y[1..n]) \]

obtained by adding to
\[ \text{LCS}(X[1..i], Y[1..j]) \]
for smaller \( i, j \)

"Optimal Substructure" Property

(Warning: not formal, only intuitive)

Optimal solution to problem also yields
optimal solution to some subproblems.

Which ones?

\[ \uparrow \]

Identifying this is crux of problem
(and algorithm)
Back to LCS

define some notation:
\[ \text{LCS}(i,j) = \text{LCS}(X[1..i], Y[1..j]) \]
\[ l(i,j) = |\text{LCS}(i,j)| \]

Our goal:
Compute \( \text{LCS}(m,n) \)

For simplicity of notation, will compute \( l(m,n) \)

General recurrence for \( l(i,j) \):

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

Then clearly \( l[i,j] = l(i-1,j-1) + 1 \)

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

**Case 2.1** \( X[i] \) is part of \( \text{LCS}(i,j) \)

then \( l(i,j) = l(i,j-1) \)

**Case 2.2** \( y[j] \) is part of \( \text{LCS}(i,j) \)

then \( l(i,j) = l(i-1,j) \)

**Case 2.3** neither \( X[i] \) nor \( y[j] \) in \( \text{LCS}(i,j) \)

then \( l(i,j) = l(i,j-1) \)
But how do we know which one we are in??

We don't.... but we can still say

\[ l(i, j) = \max \\{ 3, l(i-1, j), l(i, j-1), l(i-1, j-1) \} \]

\[ = \max 3, l(i-1, j), l(i, j-1) \]

This yields a recursive algorithm for \( l(i, j) \):

\[
\begin{align*}
\text{if } x[i] & = y[j] \text{ then return } l(i-1, j-1) + 1 \\
\text{else return } & \max 3, l(i-1, j), l(i, j-1) \end{align*}
\]

Running Time?
Efficient?

\[ T(i,j) \leq \max \sum \frac{T(i,j-1)}{T(i,j-1) + T(i-1,j)} \]

\[ = T(i,j-1) + T(i-1,j) \]

Solution turns out to be \( T(i,j) = (\min \{i,j\})^{*} \)

No progress?

Let's look carefully:

Should not open up twice!
Memoization

Instead of recomputing the answer on same input as before, write down the answer! (Memo-ize!)

Pseudocode for Memoized \( l[i,j] \)

Initialize ()

For \( i = 1 \ldots m \)
  For \( j = 1 \ldots n \)
    \( l[i,j] \leftarrow "?" \)

For \( i = 1 \ldots m \)
  \( l(i,0) \leftarrow 0 \)

For \( j = 1 \ldots n \)
  \( l(0,j) \leftarrow 0 \)

Compute \( l(i,j) \)

If \( l[i,j] \neq "?" \) then return \( l[i,j] \)

else if \( x[i]=y[i] \) return

  compute \( l[i-1,j-1] + 1 \)

else return

  \( \max \{ \text{compute-} l[i,j-1], \text{compute-} l[i-1,j] \} \)

What is the runtime?

Let's do an example first...
$X = \{ A, B, C, D, A, B \}$

$Y = \{ B, D, C, A, B, B, A \}$
Procrastinating a bit more on the runtime:

Iterative Algorithm:

Initialize:
\[ l(0,j) = l(i,0) = 0 \quad \text{for all } i,j \]

Iterations:
For \( j = 1 \) to \( m \) do
   For \( i = 1 \) to \( n \) do
      if \( x[i] = y[j] \)
         \[ l(i,j) = l(i-1,j-1) \]
      else \[ l(i,j) = \max \{ 0, \ldots, 3 \} \]

Runtime?
Clearly \( O(mn) \)
Runtime of recursive algorithm:

Charging scheme:

- Charge all but recursive calls to compute-\(l(i,j)\)
- each compute-\(l(i,j)\) performs \(\Theta(1)\) work per call called \(\leq 3\) times

\[ \therefore \Theta(1) \text{ total work} \]

- total cost = \(\sum_{i,j}\) charges to compute \(l(i,j)\)

\[ = \Theta(m,n) \]

Food for thought:

How do we find the actual longest common substring?
- modify code? \(\checkmark\) works
- use \(l(i,j)\) table? ... how? \(\checkmark\) works too