Today: Dynamic Programming
- Longest Common Subsequence Problem
- Optimal Substructure Property
- Memoization.

Dynamic Programming:
- Very general paradigm for design of algorithms.
- Will see more examples later in course.

Today's Example: Longest Common Subsequence.

Given: Two (character) sequences
\[ X[1..m] \text{ and } Y[1..n] \]

Find: Longest common subsequence.
**Example**

\[ X = A \ B \ C \ B \ D \ A \ B \]

\[ Y = B \ D \ C \ A \ B \ B \ A \]

Common Subsequence: \( Z[l..k] \)

if \( \exists 1 \leq i_1 < i_2 \ldots < i_k \leq m \) 

\& \( 1 \leq j_1 < j_2 \ldots < j_k \leq n \)

\[ X[i_j] = Y[j_j] = Z[l] \quad \forall l \in [1..k] \]

So we need to find largest such sequences for which this holds.

\[ \text{[In example} \quad k = 4 \quad Z = B \ C \ B \ A \]

\[ l_1 = 2 \quad i_2 = 3 \quad i_3 = 4 \quad i_4 = 6 \]

\[ j_1 = 1 \quad j_2 = 3 \quad j_3 = 5 \quad j_4 = 7 \]

\[ \text{But is this the largest } k ? \text{[} \]

**Brute Force:**

1. Enumerate all possible subsequences of $X$.
2. For each subsequence, check if it appears as a subsequence of $Y$.
3. Keep track of the longest such sequence found.

---

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

```python
if Z[i] = Y[j] then i++; j++
else j++;
```

**Bad News:** # 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 alg.
  - "Memo"ize the recursive algorithm to get the efficiency

To apply the steps, need to ask

- Structure in optimum of LCS?

- Recursive alg.?

- Memoize what?
Longest Common Subsequence: Structure

- Suppose we are told \( k \) and \( i_k \).

Does this simplify our task?

- Let \( l \) be largest index such that \( Y[l] = X[i_k] \). Then

0 \( \rightarrow \) Can assume w.l.o.g. \( i_k = l \).

But can we say anything about

\( i_1, i_2, \ldots, i_{k-1} \)

\( j_1, \ldots, j_{k-1} \) ?

\( \text{LCS} \left( X[1 \ldots i_{k-1}], Y[1 \ldots j_{k-1}] \right) \)

\( \text{STRUCTURE} !!! \)
Main Insight: \( \text{LCS} \left( X[1..m], Y[1..n] \right) \)

obtained by adding to

\[ \text{LCS} \left( X[1..i], Y[1..j] \right) \]

---

"Optimal Substructure" Property

(Warning: Not formal, only intuitive)

Optimal solution to problem also yields

\[ \text{Optimal solution to Some subproblem} \]

\[ \uparrow \]

which ones?

\[ \uparrow \]

identifying this is crux of algorithm
Back To LCS

\[ \text{LCS}(i, j) = \text{LCS}(X[i \ldots i], Y[j \ldots j]) \]

\[ l(i, j) = |\text{LCS}(i, j)| \]

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

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

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

\textbf{Case 1:} \( X[i] = Y[j] \)

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

\textbf{Case 2:} \( X[i] \neq Y[j] \)

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

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

\textbf{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 $LCS[i,j]$.

Then $l(i,j) = l(i-1,j-1)$

But how do we know which case we are in? We don’t. But we can still say

$$l(i,j) = \max \left\{ l(i-1,j), l(i, j-1), l(i-1, j-1) \right\}$$

$$= \max \left\{ l(i-1,j), l(i, j-1) \right\}$$

---

Recursive Algorithm. For $l(i,j)$

$$l(x[1..i], y[1..j]) = l(i,j)$$

if $x[i] = y[j]$ then return $(l(i-1,j-1) + 1)$;
else return $\max \{ l(i-1,j), l(i,j-1) \}$.
How efficient is this algorithm?

\[ T(i, j) = \max \left\{ \begin{array}{c} T(i-1, j-1) \\ T(i, j-1) + T(i-1, j) \end{array} \right\} \]

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

\[ T(i, j) = \left( \frac{i + j}{2} \right) = 2^{\min \{ i, j \}}. \]

No progress?

Let's look carefully

\[ \text{Same!} \]

\[ \text{Should not open up tree twice!} \]
Memoization

In recursive programs, should not compute same program on same input twice, instead should remember (memorize) answer and return it.

Pseudo-code for Memoized $L[i,j]$

\[
\text{INITIALIZE}(); \\
\text{for } i = 1 \rightarrow m \\
\quad \text{for } j = 1 \rightarrow n \\
\quad \quad L[i,j] \leftarrow "?"
\]

\[
\text{COMPUTE-L}(i,j) \\
\quad \text{if } L[i,j] = "?" \text{ then return } L[i,j] \\
\quad \text{else if } X[i] = Y[j] \text{ return } \\
\quad \quad \text{COMPUTE-L}[i-1,j-1] + 1; \\
\quad \text{else return } \\
\quad \quad \max \left\{ \text{COMPUTE-L}[i,j-1], \text{COMPUTE-L}[i-1,j] \right\}
\]
\[ X = A \quad \quad \quad \quad \quad \quad \quad \quad B \quad D \quad A \quad B \]
\[ Y = B \quad D \quad C \quad A \quad B \quad B \quad A \]

Efficiency = ?

First an iterative algorithm:
- Initialize: \[ l(0, i) = l(i, 0) = 0 \]
- Iteration: for \( i = 1 \) to \( m \) do
  - for \( j = 1 \) to \( n \) do
    - if \( x[i] = y[j] \)
      \[ l(i, j) = l(i-1, j-1) + 1 \]
    - else \[ l(i, j) = \max \{ \ldots \} \]
Clearly: Iterative algorithm runs in time $O(mn)$.

What about recursive algorithm?

**Charging scheme:**

- Charge "if $l[i,i] = ?" \ldots"

  step to calling program &

  rest to "compute $l[i,j]$".

- Each compute $l[i,j]$ charged $\Theta(1)$.

- Total cost $\leq \sum \text{charge (compute } l(i,j))$

  $i,j$

  $= \Theta(mn)$.

**Food for thought:**

- How to compute $LCS(i,j)$?

  - Idea 1: modify code ... works
  - Idea 2: use $l(i,j)_t$ table ... How? ... also works.