CS 341: ALGORITHMS

Lecture 11: dynamic programming III

Readings: see website

Trevor Brown (co-taught with Anna Lubiw)

https://www.student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
PROBLEM: LONGEST COMMON SUBSEQUENCE (LCS)

Problem 5.3

Longest Common Subsequence

Instance: Two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ over some finite alphabet $\Gamma$.

Find: A maximum length sequence $Z$ that is a subsequence of both $X$ and $Y$.

$Z = (z_1, \ldots, z_{\ell})$ is a subsequence of $X$ if there exist indices $1 \leq i_1 < \cdots < i_{\ell} \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, $Z$ is a subsequence of $Y$ if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_{\ell} \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Let’s first solve for the length of the LCS.
EXAMPLES

- X=aaaaa  Y=bbbbbb  Z=LCS(X,Y)=?
  - Z=ε (empty sequence)
- X=abcde  Y=bcd  Z=LCS(X,Y)=?
  - Z=bcd
- X=abcde  Y=labeef  Z=LCS(X,Y)=?
  - Z=abe
POSSIBLE GREEDY SOLUTIONS?

- Alg: for each $x_i \in X$, try to choose a matching $y_j \in Y$ that is to the right of all previously chosen $y_j$ values

- $X=\text{abcde}$, $Y=\text{labef}$
- $X=\text{abcde}$, $Y=\text{labef}$
- $X=\text{abcde}$, $Y=\text{labef}$ [no suitable $y_j$ found]
- $X=\text{abcde}$, $Y=\text{labef}$ [no suitable $y_j$ found]
- $X=\text{abcde}$, $Y=\text{labef}$
- $Z=\text{abe}$, Optimal?
POSSIBLE GREEDY SOLUTIONS?

- Alg: for each $x_i \in X$, try to choose a matching $y_j \in Y$ that is to the right of all previously chosen $y_j$ values
  - $X=\text{azbracadabra}$  $Y=\text{abracadabraz}$
  - $X=\text{azbracadabra}$  $Y=\text{abracadabrazz}$
  - $X=\text{zbracadabra}$  $Y=\text{abracadabraz}$ [no $y_j$ after $z$]
  - $X=\text{zbracadabra}$  $Y=\text{abracadabraz}$ [no $y_j$ after $z$]
  - $X=\text{azbracadabra}$  $Y=\text{abracadabraz}$ [no $y_j$ after $z$]

Blindly taking $z$ is bad. **How to decide** whether to take or leave $z$?

Try both possibilities! (Brute force / dynamic programming)

Similar greedy alg that goes right-to-left works for this input, but fails for other inputs.
DEFINING SUBPROBLEMS

- **Full problem:** return $|\text{LCS}(X, Y)|$ (i.e., length of LCS)
- Reduce size by taking **prefixes** of $X$ or $Y$
- Let $X_i = (x_1, ..., x_i)$ and $Y_i = (y_1, ..., y_i)$

\[
\begin{array}{ccccccc}
X_m & x_1 & x_2 & x_3 & x_4 & \ldots & x_{m-1} & x_m \\
X_4 & x_1 & x_2 & x_3 & x_4 \\
\end{array}
\]

Note $X = X_m$ and $Y = Y_n$

- **Subproblem:** return $|\text{LCS}(X_i, Y_j)|$
- **Idea for recurrence:** remove the last letter of $X$ or $Y$
Consider optimal solution: $Z = \text{LCS}(X, Y)$

Since $x_m, y_n \notin Z$ we know $Z = \text{LCS}(X_{m-1}, Y_{n-1})$
BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #2

Since $Z$ is a subsequence of $Y$, $z_\ell = a$ must appear in $Y_{n-1}$

$Y_{n-1}$

Case $x_m \notin Z, y_n \in Z$

$Z = LCS(X_{m-1}, Y)$

Since $y_n \notin Z$ we know $Z = LCS(X, Y_{n-1})$

Or maybe this is

This might be the final $a$ in $Z$

But this certainly is not :)

$x_1 x_2 x_3 x_4 \ldots x_{m-1} x_m$

$y_1 y_2 y_3 y_4 \ldots y_{n-1} y_n$

$z_1 z_2 z_3 \ldots z_{\ell-1} z_\ell$

$X$: a b r a c a z

$Y$: a z b r a c a d a b

$Z$: a b r a c a
BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #3

This might be the final a in Z

Or maybe this is...

This might be the final a in Z

Consume $x_m$ and $y_n$ by matching with $z_\ell$

Then we have $Z = \text{LCS}(X_{m-1}, Y_{n-1}) + z_\ell$
SUMMARIZING CASES

- $z_{\ell}$ matches **neither** $x_m$ nor $y_n$: $Z = \text{LCS}(X_{m-1}, Y_{n-1})$
- $z_{\ell}$ matches $x_m$ but not $y_n$: $Z = \text{LCS}(X_m, Y_{n-1})$
- $z_{\ell}$ matches $y_n$ but not $x_m$: $Z = \text{LCS}(X_{m-1}, Y_n)$
- $z_{\ell}$ matches **both**: $Z = \text{LCS}(X_{m-1}, Y_{n-1}) + z_{\ell}$

- ... but we don’t know $z_{\ell}$
  - Try all cases and maximize
  - **Careful:** last case is only valid if $x_m = y_n$
  - Also note $x_m = y_n$ only holds in the last case
  - Cases 2&3: trivial
  - Case 1: if $x_m = y_n \neq z_{\ell}$ then we can improve $Z$ (contra)
DERIVING A RECURRENCE

- \( z_\ell \) matches **neither** \( x_m \) nor \( y_n \) \((x_m \neq y_n)\) \( Z = \text{LCS}(X_{m-1}, Y_{n-1}) \)
- \( z_\ell \) matches \( x_m \) but not \( y_n \) \((x_m \neq y_n)\) \( Z = \text{LCS}(X_m, Y_{n-1}) \)
- \( z_\ell \) matches \( y_n \) but not \( x_m \) \((x_m \neq y_n)\) \( Z = \text{LCS}(X_{m-1}, Y_n) \)
- \( z_\ell \) matches **both** \((x_m = y_n)\) \( Z = \text{LCS}(X_{m-1}, Y_{n-1}) + z_\ell \)

Let \( c(i, j) = |\text{LCS}(X_i, Y_j)| \)

- **Brainstorming sensible base cases**
  - \( i = 0 \) one string is empty, so \( c(0, j) = 0 \) (similarly for \( j = 0 \))
- **General cases**

\[
\begin{align*}
  c(i, j) &= c(i - 1, j - 1) + 1 & \text{if } x_m = y_n \\
  c(i, j) &= \max\{c(i - 1, j - 1), c(i, j - 1), c(i - 1, j)\} & \text{if } x_m \neq y_n
\end{align*}
\]
RECURSION

Combining expressions

\[ c(i, j) = \begin{cases} 
0 & \quad \text{if } i = 0 \text{ or } j = 0 \\
 c(i - 1, j - 1) + 1 & \quad \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
 \max\{c(i, j - 1), c(i - 1, j), c(i - 1, j - 1)\} & \quad \text{if } i, j \geq 1 \text{ and } x_i \neq y_j 
\end{cases} \]

Can simplify!

Observe \( c(i - 1, j - 1) \leq c(i - 1, j) \)  
(former is a subproblem of the latter)

\[ c(i, j) = \begin{cases} 
0 & \quad \text{if } i = 0 \text{ or } j = 0 \\
 c(i - 1, j - 1) + 1 & \quad \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
 \max\{c(i, j - 1), c(i - 1, j)\} & \quad \text{if } i, j \geq 1 \text{ and } x_i \neq y_j 
\end{cases} \]
Suppose $X = \text{gdvegta}$ and $Y = \text{gvcekst}$

$$c(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 c(i - 1, j - 1) + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
 \max\{c(i, j - 1), c(i - 1, j)\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j 
\end{cases}$$
Suppose $X = \text{gdvegta}$ and $Y = \text{gvcekst}$

$$c(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
c(i - 1, j - 1) + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
\max\{c(i, j - 1), c(i - 1, j)\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j
\end{cases}$$
PSEUDOCODE

Algorithm: \textit{LCS1}(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))

\begin{align*}
\text{for } i \leftarrow 0 \text{ to } m \\
& \quad c[i, 0] \leftarrow 0 \\
\text{for } j \leftarrow 0 \text{ to } n \\
& \quad c[0, j] \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } m \\
& \quad \text{for } j \leftarrow 1 \text{ to } n \\
& \quad \quad \text{if } x_i = y_j \\
& \quad \quad \quad \text{then } c[i, j] \leftarrow c[i - 1, j - 1] + 1 \\
& \quad \quad \quad \text{else } c[i, j] \leftarrow \max\{c[i, j - 1], c[i - 1, j]\} \\
\text{return } (c[m, n]);
\end{align*}

Complexity: 
Space? Time? 
(word RAM model) 
\(\Theta(nm)\) for both
COMPUTING THE LCS
NOT JUST ITS LENGTH

To make it easy to find the actual LCS (not just its length),

Consider which table entry was used to calculate \( c[i, j] \)

\[
\begin{align*}
\begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max\{c(i - 1, j - 1), c(i - 1, j), c(i, j - 1)\} & \text{if } i, j \geq 1
\end{cases}
\end{align*}
\]

We store the direction to that entry in an array \( \pi[i, j] \)

Case 1: \( c(i, j) = c(i, j - 1) \)
We store “J” in \( \pi[i, j] \) to indicate decrementing \( j \) (to get \( i, j - 1 \))

Case 2: \( c(i, j) = c(i - 1, j) \)
We store “I” in \( \pi[i, j] \) to indicate decrementing \( i \) (to get \( i - 1, j \))

Case 3: \( c(i, j) = c(i - 1, j - 1) + 1 \)
We store “IJ” in \( \pi[i, j] \) to indicate decrementing both \( i \) and \( j \)

Recall in this case, \( x_i = y_j \) so we include \( x_i \) in the LCS

In our example table we just draw an arrow to the entry...
SAVING THE DIRECTION TO THE PREDECESSOR SUBPROBLEM $\pi$

LCS2($X[1..m], Y[1..n]$)

1. $c = \text{new array}[0..m][0..n]$
2. $\pi = \text{new array}[0..m][0..n]$
3. for $i = 0..m$ do $c[i][0] = 0$
4. for $j = 0..n$ do $c[0][j] = 0$
5. for $i = 1..m$
6.   for $j = 1..n$
7.     if $X[i] = Y[j]$
8.       $c[i][j] = c[i-1][j-1] + 1$
9.       $\pi[i][j] = \text{"IJ"}$
10.      else if $c[i][j-1] > c[i-1][j]$
11.       $c[i][j] = c[i][j-1]$
12.       $\pi[i][j] = \text{"J"}$
13.      else // $c[i][j-1] <= c[i-1][j]$
14.       $c[i][j] = c[i-1][j]$
15.       $\pi[i][j] = \text{"I"}$
16. return $c, \pi$

Case: $c(i, j) = c(i - 1, j - 1) + 1$
- We store “IJ” in $\pi[i, j]$ to indicate decrementing both $i$ and $j$.

Case: $c(i, j) = c(i, j - 1)$
- We store “J” in $\pi[i, j]$ to indicate decrementing $j$ (to get $i, j - 1$).

Case: $c(i, j) = c(i - 1, j)$
- We store “I” in $\pi[i, j]$ to indicate decrementing $i$ (to get $i - 1, j$).

Recall in this case, $x_i = y_j$ so we include $x_i$ in the LCS.
Suppose $X = gdvegta$ and $Y = gvcekst$.

### Example

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>g</th>
<th>d</th>
<th>v</th>
<th>e</th>
<th>g</th>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th></th>
<th>i = 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>0</td>
<td>0</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>e</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Done:**

`seq=gvet`

**seq=et**

`this is.`

`seq=t`

`this “a” is not in`
FOLLOWING PREDECESSORS TO COMPUTE THE LCS

```java
FindLCS(c[0..m][0..n], π[0..m][0..n], X[0..m])
    lcs = new string
    i = m
    j = n

    while i>0 and j>0
        if π[i][j] == "IJ"
            lcs.append(X[i])
            i--
            j--
        else if π[i][j] == "J"
            j--
        else // π[i][j] == "I"
            i--

    return reverse(lcs)
```

Complexities of this trace-back algo:
Space? Time?
(word RAM model)

- space: $O(n+m)$ words
- time: $O(n+m)$
PROBLEM: MINIMUM LENGTH TRIANGULATION

- **Input:** \( n \) points \( q_1, \ldots, q_n \) in 2D space that form a **convex** \( n \)-gon \( P \)
  - Assume points are **sorted clockwise** around the center of \( P \)

- **Find:** a triangulation of \( P \) such that the sum of the perimeters of the \( n - 2 \) triangles is minimized

- **Output:** the sum of the **perimeters** of the triangles in \( P \)
How hard is this problem?

How many triangulations are there?

Number of triangulations of a convex $n$-gon $= \text{the (n - 2)nd Catalan number}$

This is $C_{n-2} = \frac{1}{n-1} \binom{2n-4}{n-2}$

It can be shown that $C_{n-2} \in \Theta(4^n / (n - 2)^{3/2})$
PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n-1\}$. 
PROBLEM DECOMPOSITION

The edge \( q_n q_1 \) is in a triangle with a third vertex \( q_k \), where \( k \in \{2, \ldots, n - 1\} \).

For a given \( k \), we have:

the triangle \( q_1 q_k q_n \), \hfill (1)

\[
\begin{align*}
q_1 & \quad q_2 \\
q_2 & \quad q_3 \\
q_3 & \quad q_k \\
q_k & \quad q_n \\
q_n & \quad q_{n-1}
\end{align*}
\]
PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n - 1\}$.

For a given $k$, we have:

the triangle $q_1 q_k q_n$, \hspace{1cm} (1)

the polygon with vertices $q_1, \ldots, q_k$, \hspace{1cm} (2)
PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n - 1\}$.

For a given $k$, we have:

- the triangle $q_1 q_k q_n$, \hspace{1cm} (1)
- the polygon with vertices $q_1, \ldots, q_k$, \hspace{1cm} (2)
- the polygon with vertices $q_k, \ldots, q_n$. \hspace{1cm} (3)
PROBLEM DECOMPOSITION

The edge $q_nq_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n-1\}$.

For a given $k$, we have:

- the triangle $q_1q_kq_n$, (1)
- the polygon with vertices $q_1, \ldots, q_k$, (2)
- the polygon with vertices $q_k, \ldots, q_n$. (3)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).
PROBLEM DECOMPOSITION

The edge $q_nq_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n-1\}$.

For a given $k$, we have:

- the triangle $q_1q_kq_n$, \hspace{1cm} (1)
- the polygon with vertices $q_1, \ldots, q_k$, \hspace{1cm} (2)
- the polygon with vertices $q_k, \ldots, q_n$. \hspace{1cm} (3)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).
RECURRENCE RELATION

- Let $S(i, j) =$ optimal solution to the subproblem consisting of the polygon with vertices $q_i \ldots q_j$

- Let $\Delta_{ijk}$ denote $\text{perimeter}(q_i \ldots q_j)$

- If a given point $q_k$ is in the optimal solution, then $S(i, j) = S(i, k) + \Delta_{ijk} + S(k, j)$
**RECURRENCE RELATION**

- But we don’t know the optimal $k$
  - Minimize over all $k$ strictly between $i$ and $j$

\[
S(i, j) = \begin{cases} 
\min_{i < k < j} \{S(i, k) + \Delta_{ijk} + S(k, j)\} & \text{if } j \geq i + 2 \\
0 & \text{otherwise}
\end{cases}
\]
FILLING IN THE TABLE

- Table $S[1..n, 1..n]$ of solutions to $S(i, j)$ for all $i, j \in \{1..n\}$

$$S(i, j) = \begin{cases} 
\min_{i < k < j} \{S(i, k) + \Delta_{ijk} + S(k, j)\} & \text{if } j \geq i + 2 \\
0 & \text{otherwise}
\end{cases}$$

Dependencies:
$S[i, k]$ and $S[k, j]$

For $k = (i + 1) \ldots (j - 1)$

We depend on larger $i$
And same $i$ but smaller $j$

What’s a correct fill order?
For $i = n..1$, for $j = 1..n$
RUNTIME
WORD RAM MODEL

- Number of subproblems: $n^2$
- Time to solve subproblem $S(i, j): O(j - i) \subseteq O(n)$
- So total runtime is in $O(n^3)$
  - More effort needed to show $\Omega(n^3)$, since so many subproblems are base cases, which take $\Theta(1)$ steps
  - **Incidentally**, this is polynomial time (in the input size)
  - But basic runtime analysis does **not** require such an argument

$$S(i, j) = \begin{cases} 
\min_{i < k < j} \{S(i, k) + \Delta_{ijk} + S(k, j)\} & \text{if } j \geq i + 2 \\
0 & \text{otherwise}
\end{cases}$$
MEMOIZATION: AN ALTERNATIVE TO DP

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal.

Memoization is a recursive algorithm based on the same recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.
EXAMPLE: USING MEMOIZATION TO COMPUTE FIBONACCI NUMBERS EFFICIENTLY

main

for $i \leftarrow 2$ to $n$
do $M[i] \leftarrow -1$
return $(RecFib(n))$

procedure $RecFib(n)$
if $n = 0$ then $f \leftarrow 0$
else if $n = 1$ then $f \leftarrow 1$
else if $M[n] \neq -1$ then $f \leftarrow M[n]$
\hspace{1em}$\left\{ \begin{align*}
&f_1 \leftarrow RecFib(n - 1) \\
&f_2 \leftarrow RecFib(n - 2)
\end{align*} \right.$
\hspace{1em}else
\hspace{2em}$\left\{ \begin{align*}
&f \leftarrow f_1 + f_2 \\
&M[n] \leftarrow f
\end{align*} \right.$
\hspace{2em}return $(f)$;

If $M[n]$ is already computed, don’t recurse!
If $M[n]$ is already computed, don’t recurse!

**VISUALIZING MEMOIZATION**

```plaintext
procedure RecFib(n)
    if $n = 0$ then $f \leftarrow 0$
    else if $n = 1$ then $f \leftarrow 1$
    else if $M[n] \neq -1$ then $f \leftarrow M[n]$
        $f_1 \leftarrow \text{RecFib}(n - 1)$
        $f_2 \leftarrow \text{RecFib}(n - 2)$
    else
        $f \leftarrow f_1 + f_2$
        $M[n] \leftarrow f$
    return $(f)$;
```

Already done!

Done! $f_5 = 5$

Done! $f_4 = 3$

Done! $f_3 = 2$

Done! $f_2 = 1$

Already done! $f_1 = 1$

Done! $f_1 = 1$

Done! $f_0 = 0$
BONUS CLARIFICATION MATERIAL
MODEL COMPARISON

Word RAM
- Each variable (or array entry, etc.) \( x \) is a word
- All words have the same size
  - \( O(\log W) \) bits where \( W = \) # of words in the input (not realistic if variables contain big numbers!)
- Runtime is the number of operations on words (where each word operation takes \( O(1) \) time)
  - Read/write in \( O(1) \)
  - Add in \( O(1) \)
  - Multiply in \( O(1) \)
- Space complexity is the number of words used (excluding the input)

Bit complexity
- Each variable \( x \) is a bit string
- Variables can have different numbers of bits
  - \( x \) is encoded in \( O(\log x) \) bits
- Runtime is the number of operations on bits (where each bit operation takes \( O(1) \) time)
  - Read/write in \( O(\log x) \) (\( \log x = \) # bits in \( x \))
  - Add \( x + y \) in \( O(\log x + \log y) \)
  - Multiply in \( O(\log x \times \log y) \)
- Space complexity is the number of bits used (excluding the input)

Unit cost model (not used anymore)
- Words have unlimited size
- But still \( O(1) \) access time
CALCULATIONS USING INPUT SIZE

- Clarification: you can compute space/time complexity without calculating the input size
- Input size calculations are typically only needed if we ask you to show an algorithm runs in polytime
  - “runs in polytime” means the runtime is at most a polynomial in the number of bits in the input
  - Lots of this in tractability / NP completeness
- Just trying to expose you to these ideas ahead of time...
SO... IS DP LCS A POLYTIME ALGORITHM?

- Is $nm$ polynomial in the input size (# of bits in the input)?
- Word RAM model
  - Input contains $\Theta(n + m)$ words
  - Word RAM model says each word stores $\Theta(\log w)$ bits
    where $w = \# \text{ words in the input}$
    - So in this case, $\Theta(\log(n + m))$ bits per word
  - $S \in \Theta((n + m) \log(n + m))$
    - Want a term that looks like $nm$
SO... IS DP LCS A POLYTIME ALGORITHM?

- Try squaring: $S^2 \in \Theta((n + m) \log(n + m)^2)$
- $\Theta(n^2 \log^2(n + m) + nm \log^2(n + m) + m^2 \log^2(n + m))$
- Of course, $nm \in O(nm \log^2(n + m))$
  - ... which is just one of the terms of $S^2$
  - So $nm \in O(S^2)$
- So the runtime is polynomial in the input size (in bits)
  - But this was ugly. Is there a simpler approach?
SO... IS DP LCS A POLYTIME ALGORITHM?

- Calculation using words would be simpler...
- **Words in the input** $W \in O(n + m)$
- $W^2 \in O((n + m)^2) = O(n^2 + nm + m^2)$
  - So $O(nm) \subseteq O(W^2)$
  - I.e., polynomial in the # of **words** in the input

- How does this help us?
  - # **words** $W$ in the input $\leq$ # **bits** $S$ in the input
  - So $O(nm) \subseteq O(W^2)$ **implies** $O(nm) \subseteq O(S^2)$

So, for showing **polytime**, you can calculate the input size using **words** (and justify like this)