Let's first solve for the length of the LCS

**Possibility 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
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabra} \)
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabra} \)
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabra} \) (no \( y_j \) after \( z \))
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabra} \) (no \( y_j \) after \( z \))
  - \( Z = \text{abe} \)
  - \( Z = \text{abe} \)
  - \( Z = \text{abe} \)

Blindly taking is bad. How to decide whether to take or leave if...

Similar greedy alg that goes right-to-left works for this input...

Try both possibilities (brute force / dynamic programming)

**Defining subproblems**
- Full problem: return \([\text{LCS}(X, Y)]\) (i.e. length of LCS)
- Reduce size by taking prefixes of \( X \) or \( Y \)
  - Let \( X_1 = (x_1, \ldots, x_t) \) and \( Y_1 = (y_1, \ldots, y_t) \)
  - Note \( X = X_m \) and \( Y = Y_n \)
  - Subproblem: return \( [\text{LCS}(X_1, Y_1)] \)
  - Idea for recurrence: remove the last letter of \( X \) or \( Y \)
**Example #1 to Build Intuition**

- $x = bba$, $y = abab$, $z = abba$
- $LCS(x, y) = abc$
- $LCS(y, z) = cba$
- $c = 1$

- **Finding $y_i$**
  - $y_i = c$ must appear in $y_{i+1}$
  - $y_i$ cannot be the first $a$ in $y$

- **Finding $x_i$**
  - $x_i$ cannot be the first $a$ in $x$
  - If $y_i$ is a part of $z$ then $y_{i+1}$ must appear in $x_{i+1}$

**Example #2**

- **Finding $y_i$**
  - $y_i = a$ must appear in $y_{i+1}$
  - $y_i$ cannot be the first $a$ in $y$

- **Finding $x_i$**
  - $x_i$ cannot be the first $a$ in $x$
  - The $y_{i+1}$ cannot be the first $a$ in $y$

**Example #3**

- **Finding $y_i$**
  - $y_i = c$ must appear in $y_{i+1}$
  - $y_i$ cannot be the first $a$ in $y$

- **Finding $x_i$**
  - $x_i$ cannot be the first $a$ in $x$
  - The $y_{i+1}$ cannot be the first $a$ in $y$

**Summarizing Cases**

- $x_i$ matches neither $x_m$ nor $y_k$ ($x_m = y_k$)
  - $Z = LCS(x_m, y_{m+1})$
- $x_i$ matches $x_m$ but not $y_k$ ($x_m = y_k$)
  - $Z = LCS(x_m, y_{m+1})$
- $x_i$ matches $y_k$ but not $x_m$ ($x_m = y_k$)
  - $Z = LCS(x_{m+1}, y_k)$
- $x_i$ matches both ($x_m = y_k$)
  - $Z = LCS(x_{m+1}, y_k) + x_i$

**But we don’t know $x_i$**

- Try all cases and maximize
- **Careful! last case is only valid if $x_m = y_k$**
- Also note $x_m = y_k$ only holds in the last case
- Cases 2 & 3: trivial
- Case 1: if $x_m = y_k \neq x_i$ then we can improve $Z$ (contra)

**Deriving a Recurrence**

- Recall $Z = LCS(x_m, y_k)$
- $c(i, j) = |LCS(x_i, y_j)|$

- **Finding $y_i$**
  - $i = 0$, one string is empty, so $c(0, j) = 0$ (similarly for $j = 0$)

- **General cases**

0. $c(i, j) = c(i - 1, j - 1) + 1$ if $x_m \neq y_k$
1. $c(i, j) = \max(c(i - 1, j), c(i, j - 1))$ if $x_m \neq y_k$

**Recurrent**

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

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

<table>
<thead>
<tr>
<th>( Y )</th>
<th>g</th>
<th>v</th>
<th>e</th>
<th>g</th>
<th>v</th>
<th>c</th>
<th>t</th>
<th>s</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X )</td>
<td>o</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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

**PSEUDOCODE**

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

for \( i = 1 \) to \( m \)

\( e(0,0) = 0 \)

for \( j = 1 \) to \( n \)

\( e(0,j) = 0 \)

for \( i = 1 \) to \( m \)

for \( j = 1 \) to \( n \)

if \( x_i = y_j \)

\( e(i,j) = e(i-1,j-1) + 1 \)

else

\( e(i,j) = \max(e(i-1,j),e(i,j-1)) \)

return \( e(m,n) \).

**COMPUTING THE LCS**

Not Just Its Length

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

Consider each table entry used to calculate \( e(i,j) \)

<table>
<thead>
<tr>
<th>Case 1: ( e(i,j) = e(i-1,j) )</th>
<th>Case 2: ( e(i,j) = e(i,j-1) )</th>
<th>Case 3: ( e(i,j) = e(i-1,j-1) + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>We store &quot;I&quot; in ( e(i,j) ) to indicate that ( x_i ) is in the LCS.</td>
<td>We store &quot;V&quot; in ( e(i,j) ) to indicate that ( y_j ) is in the LCS.</td>
<td>We store &quot;D&quot; in ( e(i,j) ) to indicate that ( x_i = y_j ) and ( x_i ) and ( y_j ) are both included in the LCS.</td>
</tr>
</tbody>
</table>

In our example table we just draw an arrow to the entry...

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

**SAVING THE DIRECTION TO THE PREDECESSOR SUBPROBLEM π**

Case: \( e(i,j) = e(i-1,j) + 1 \)

We store "D" in \( e(i,j) \) to indicate that \( x_i \) is in the LCS.

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

Case: \( e(i,j) = e(i,j-1) \)

We store "I" in \( e(i,j) \) to indicate that \( x_i \) is in the LCS.

Case: \( e(i,j) = e(i-1,j-1) + 1 \)

We store "V" in \( e(i,j) \) to indicate that \( y_j \) is in the LCS.

Case: \( e(i,j) = e(i-1,j-1) \)

We store "D" in \( e(i,j) \) to indicate that \( x_i = y_j \) and \( x_i \) and \( y_j \) are both included in the LCS.

Example

Suppose \( X = \text{gvegta} \) and \( Y = \text{gvckst} \). How to obtain LCS?
Complexities of this trace-back algo: Space? Time? (word RAM model)

- Space: $O(n+m)$ words
- Time: $O(n+m)$

How many triangulations are there?

Number of triangulations of a convex $n$-gon = the $n$-th Catalan number

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

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

Problem: Minimum length triangulation

- Input: $n$ points $q_1, ..., q_n$ 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 $P$

Problem decomposition

The edge $q_kq_1$ is in a triangle with a third vertex $q_{n-1}$, where $k \in \{2, ..., n-3\}$.

For a given $k$, we have:
- the triangle $q_kq_{n-1}q_{n}$ (1)
- the polygon with vertices $q_1, ..., q_{n-1}$ (2)
**PROBLEM DECOMPOSITION**

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

For a given $k$, we have:
- the triangle $q_i q_k q_j$ (1)
- the polygon with vertices $q_1, \ldots, q_{k-1}$ (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).

**RECURSION RELATION**

- Let $S(i,j)$ be the optimal solution to the subproblem consisting of the polygon with vertices $q_i, \ldots, q_j$.
- Let $\Delta_{ij}$ denote $\text{perimeter}(q_i q_k q_j)$.
- If a given point $q_k$ is in the optimal solution, then $S(i,j) = S(i,k) + \Delta_{ij} + S(k,j)$.

**FILLING IN THE TABLE**

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

- Dependencies:
  - $S(i, j)$ and $S(j, i)$ for $k = i+1, \ldots, j-1$.
  - $S(i, j)$ and $S(i, k)$ for $k = i+1, \ldots, j-1$.
  - $S(i, j)$ and $S(k, j)$ for $k = i+1, \ldots, j-1$.

- We depend on larger $i$ and same (but smaller) $j$.

What's a correct fill order? For $i = n$, for $j = 1$.
**RUNTIME**

**WORD RAM MODEL**

- Number of subproblems: \( n^2 \)
- Time to solve subproblem \( S(i,j): O(j - i) \leq O(n) \)
- 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 (\( n \) the input size)
- But basic runtime analysis does not require such an argument

**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 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-computed.

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**

```plaintext
main
for i ← 2 to n
    do M[i] ← −1
return (RecFib(n))

procedure RecFib(n)
if n = 0 then f ← 0
else if n = 1 then f ← 1
else if M[n] ≠ −1 then f ← M[n]
else
f1 ← RecFib(n − 1)
f2 ← RecFib(n − 2)
f ← f1 + f2
M[n] ← f
return (f)
```

**VISUALIZING MEMOIZATION**

**MODEL COMPARISON**

<table>
<thead>
<tr>
<th>Model</th>
<th>Word RAM</th>
<th>BF complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Words have unlimited size</td>
<td>- Variables can have different numbers of bits</td>
</tr>
<tr>
<td></td>
<td>Each word is a single word</td>
<td>- Variables are 1 bit each</td>
</tr>
<tr>
<td></td>
<td>Each variable is a bit string</td>
<td>- Space complexity is the number of bits used (excluding the input)</td>
</tr>
<tr>
<td></td>
<td>All words have the same size</td>
<td>- Time complexity is ( O(</td>
</tr>
<tr>
<td></td>
<td>Runtime is the number of operations on words (where each word operation takes ( O(1) ) time)</td>
<td>- Size is ( 1 ) or ( 0 ) in ( O(</td>
</tr>
<tr>
<td></td>
<td>Move((\text{write} \rightarrow \text{read}))</td>
<td>- Add is ( x + y ) in ( O(</td>
</tr>
<tr>
<td></td>
<td>Add((x \rightarrow x + y))</td>
<td>- And is ( x &amp; y ) in ( O(</td>
</tr>
<tr>
<td></td>
<td>And((x &amp; y \rightarrow \text{nop}))</td>
<td>- Space complexity is the number of bits used (excluding the input)</td>
</tr>
</tbody>
</table>

**BONUS CLARIFICATION MATERIAL**
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 Θ(n + m) words
  • Word RAM model says each word stores Θ(log w) bits
  where w = # words in the input
  • So in this case, Θ(log(n + m)) bits per word
  • So θ ∈ Θ((n + m) log(n + m))
  • Want a term that looks like \( \text{time} \)

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...
  • \textbf{Words in the input} \( W \in O(n + m) \)
  • \( W^2 \in O((n + m)^2) = O(n^2 + nm + m^2) \)
    • So \( \Theta(n) \leq \Theta(W^2) \)
    • I.e., polynomial in the # of words in the input
  • How does this help us?
    • \# words \( W \) in the input \( \leq \# \text{ bits } S \) in the input
    • So \( O(nm) \leq O(W^2) \) \textbf{implies} \( O(nm) \leq O(S^2) \)