Let's first solve for the length of the 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 \( Y \).

\( Z = (z_1, \ldots, z_t) \) is a subsequence of \( X \) if there exist indices

\[ 1 \leq i_1 < \cdots < i_t \leq m \quad \text{such that} \quad x_{i_j} = z_{j}, \quad 1 \leq j \leq t. \]

Similarly, \( Z \) is a subsequence of \( Y \) if there exist (possibly different) indices

\[ 1 \leq h_1 < \cdots < h_t \leq n \quad \text{such that} \quad y_{h_j} = z_{j}, \quad 1 \leq j \leq t. \]

**Examples**

- \( X = \text{aaaaa} \) \( Y = \text{bbbb} \)
  \( Z = \text{LCS}(X, Y) = \text{a} \)
- \( X = \text{abaced} \) \( Y = \text{bcd} \)
  \( Z = \text{LCS}(X, Y) = \text{bc} \)
- \( X = \text{abcde} \) \( Y = \text{labef} \)
  \( Z = \text{LCS}(X, Y) = \text{ab} \)
- \( X = \text{abcde} \) \( Y = \text{labef} \)
  \( Z = \text{LCS}(X, Y) = \text{ab} \)
- \( Z = \text{abe} \)

**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
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabraz} \)
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabraz} \) [no \( y_j \) after \( z \)]
  - \( X = \text{bracadabra} \) \( Y = \text{bracadabraz} \) [no \( y_j \) after \( z \)]

**Defining Subproblems**

- Full problem: return [LCS(\( X, Y \))] (i.e., length of LCS)
- Reduce size by taking prefixes of \( X \) or \( Y \)
  - Let \( X_i = (x_1, \ldots, x_i) \) and \( Y_i = (y_1, \ldots, y_i) \)
    \[
    \begin{array}{cccccccc}
    x_1 & x_2 & x_3 & x_4 & \ldots & x_m \\
    y_1 & y_2 & y_3 & y_4 & \ldots & y_n \\
    x_1 & x_2 & x_3 & x_4 & \ldots & x_m \\
    y_1 & y_2 & y_3 & y_4 & \ldots & y_n \\
    \end{array}
    \]

- Note \( X = X_m \) and \( Y = Y_n \)
- Subproblem: return [LCS(\( X_i, Y_i \))]

**Idea for recurrence:** remove the last letter of \( X \) or \( Y \)
BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #1 TO BUILD INTUITION

This cannot be the final $z$ in $Z$.

Nor of these is a part of $Z$.

The correct is the final $z$ in $Z$.

Since $z$ is a subsequence of $x$, $z_r$ must appear in $x_{i+1}$.

Consider optimal solution $Z = LCS(x, y)$.

Since $x_m, y$ is $z$ we know $Z = LCS(x_m, y)$.

EXAMPLE #2

Or maybe this is.

But this certainly is not.

Case $x_k \notin Z$, $y \in Z$).

Since $y_k$, $z$ we know $Z = LCS(x, y)$.

SUMMARIZING CASES

- $x_r$ matches neither $x_m$ nor $y_n$.
- $x_r$ matches $x_m$ but not $y_n$.
- $x_r$ matches $y_n$ but not $x_m$.
- $x_r$ matches both $y_n$.

- \ldots but we don’t know $x_r$.

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 $x_m = y_n$ then we can improve $Z$ (contra).

DERIVING A RECURRENCE

Recall $Z = LCS(x_m, y_n)$.

- $x_r$ matches neither $x_m$ nor $y_n$.
- $x_r$ matches $x_m$ but not $y_n$.
- $x_r$ matches $y_n$ but not $x_m$.
- $x_r$ matches both $y_n$.

Let $c(i, j) = |LCS(x_i, y_j)|$.

- Brainstorming sensible base cases:
  - $i = 0$: one string is empty, so $c(0, j) = 0$ (similarly for $j = 0$).
- General cases
  - $c(i, j) = c(i-1, j-1) + 1$ if $x_i = y_j$.
  - $c(i, j) = \max(c(i-1, j-1), c(i-1, j))$ if $x_i \neq y_j$.

RECURRENCE

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

- Can simplify:
  - Observe $c(i-1, j-1) \leq c(i-1, j)$ (former is a subproblem of the latter).
  - $c(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\ \max(c(i-1, j-1), c(i-1, j)) & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j \end{cases}$
Suppose $X = \text{gdvegta}$ and $Y = \text{gyveckst}$

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

_Pseudocode_

**Algorithm:** LCS($X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n)$)

1. for $i = 1$ to $m$
   - $c[i,0] = 0$
2. for $j = 1$ to $n$
   - $c[0,j] = 0$
3. for $i = 1$ to $m$
   - for $j = 1$ to $n$
     - if $x_i = y_j$
       - $c[i,j] = c[i-1,j-1] + 1$
     - else
       - $c[i,j] = \max(c[i-1,j], c[i,j-1])$
4. return $c[m,n]$.

**Complexity:** Space? Time?

**Pseudo-code for time:**

Case 1: $c(i,j) = c(i-1,j-1) + 1$
- We store "1" in $c[i,j]$ to indicate an entry in the LCS.

Case 2: $c(i,j) = c(i-1,j)$
- We store "1" in $c[i,j]$ to indicate an entry in the LCS.

Case 3: $c(i,j) = c(i,j-1)$
- We store "1" in $c[i,j]$ to indicate an entry in the LCS.

**Computing the LCS not just its length**

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

**Example**

Suppose $X = \text{gdvegta}$ and $Y = \text{gyveckst}$.

How to obtain LCS? from this table?

**Saving the direction to the predecessor subproblem π**

Case: $c(i,j) = c(i-1,j-1) + 1$
- We store "1" in $c[i,j]$ to indicate an entry in the LCS.

Recall in this case, $x_i = y_j$ so we include $x_i$ in the LCS.

Case: $c(i,j) = c(i-1,j)$
- We store "0" in $c[i,j]$ to indicate an entry not in the LCS.

Case: $c(i,j) = c(i,j-1)$
- We store "0" in $c[i,j]$ to indicate an entry not in the LCS.

Recall in this case, $x_i \neq y_j$ so we include $y_j$ in the 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 $n-2$nd Catalan number

This is $\mathcal{C}_{n-2} = \frac{1}{(n-2)} \binom{2n-4}{n-2}$

It can be shown that $\mathcal{C}_{n-2} \in \Theta\left(\frac{4^n}{n!}\right)$
**PROBLEM DECOMPOSITION**

The edge \( e_{i,j} \) 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 \( e_{i,j}q_k \) \( (1) \)
- the polygon with vertices \( q_j, \ldots, q_i \) \( (2) \)
- the polygon with vertices \( q_k, \ldots, q_i \) \( (3) \)

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

\[
S(i,j) = S(i,k) + \Delta_{i,j} + S(k,j)
\]

**FILLING IN THE TABLE**

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

\[
S(i,j) = \begin{cases} 
0 & \text{if } j \leq i + 2 \\
\min_{k \in (i+1, j-1)} \left[ S(i,k) + \Delta_{i,k} + S(k,j) \right] & \text{otherwise}
\end{cases}
\]

**Dependencies**:
- \( S(i,k) \) and \( S(k,j) \)
- For \( k = i+1, \ldots, j-1 \)
**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 $O(n^2)$
- More effort needed to show $\Omega(n^2)$, 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

**EXAMPLE: USING MEMOIZATION TO COMPUTE FIBONACCI NUMBERS EFFICIENTLY**

```python
def RecFib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        memo[n] = RecFib(n-1) + RecFib(n-2)
        return memo[n]

# Initialize memoization table
memo = {0: 0, 1: 1}

# Compute Fibonacci numbers up to n
for i in range(2, n+1):
    memo[i] = memo[i-1] + memo[i-2]

# Return the nth Fibonacci number
RecFib(n)
```

**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 some 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 initialized a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.

**UNIT COST MODEL (NOT USED ANYMORE)**

- Words have unlimited size
- But still $O(1)$ access time

**MODEL COMPARISON**

<table>
<thead>
<tr>
<th>RAM Model</th>
<th>Word RAM</th>
<th>BF Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each variable is a word entry, etc.</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>All words have the same size</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Run time is the number of operations on words</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Space complexity is the number of words used (excluding the input)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
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 \(w = \#\) words in the input.
  - So in the case, \(\Theta((n + m)\log(n + m))\) bits per word.
  - So \(S \in \Theta((n + m)\log(n + m))\).
  - 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 \(O(nm) \leq O(W^2)\).
  - 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 \(O(nm) \leq O(W^2) \implies O(nm) \leq O(S^2)\).