**Dynamic Programming**

We look at sequences of characters (strings)
e.g. \( x = \text{"ABCA"} \)

**Def:** A subsequence of \( x \) is an sequence obtained from \( x \) by possibly deleting some of its characters (but without changing their order)

**Examples:**
- “ABC”, “ACA”, “AA”, “ABCA”

**Def** A prefix of \( x \), denoted \( x[1..m] \), is the sequence of the first \( m \) characters of \( x \)

**Examples:**
- \( x[1..4] = \text{"ABCA"} \)
- \( x[1..3] = \text{"ABC"} \)
- \( x[1..2] = \text{"AB"} \)
- \( x[1..1] = \text{"A"} \)
- \( x[1..0] = \text{""} \)

**Longest Common Subsequence (LCS)**
- Given two sequences \( x[1..m] \) and \( y[1..n] \), find a longest subsequence common to them both.

\[
\begin{align*}
x: & \quad A \quad B \quad C \quad B \quad D \quad A \quad B \\
y: & \quad B \quad D \quad C \quad A \quad B \quad A \\
\end{align*}
\]

\[ \text{BCBA} = \text{LCS}(x, y) \]

Different phrasing: Find a set of a maximum number of segments, such that
- Each segment connects a character of \( x \) to an identical character of \( y \),
- Each character is used at most once
- Segments do not intersect.
Cs445 salute

Brute-force LCS algorithm

Checking every subsequence of $x$ whether it is also a subsequence of $y$.

Analysis

• Checking $= \Theta(m+n)$ time per subsequence.
• $2^m$ subsequences of $x$

Worst-case running time $= \Theta((m+n)2^m) = \text{exponential time.}$

Towards a better algorithm

Simplification:

1. Look at the length of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.
Towards a better algorithm

**Simplification:**
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

**Notation:** Denote the length of a sequence $s$ by $|s|$.

---

Towards a better algorithm

**Simplification:**
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

**Notation:** Denote the length of a sequence $s$ by $|s|$.

**Strategy:** Consider *prefixes* of $x$ and $y$.
- Define $c[i, j] = |\text{LCS}(x[1..i], y[1..j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

---

Recursive formulation

**Observation:** It is impossible that $x[m]$ is matched to an element in $y[1..n-1]$ and simultaneously $y[n]$ is matched to an element in $x[1..m-1]$ (since it must create a pair of crossing segments).

**Conclusion** – either $x[m]$ is matched to $y[n]$, or one at least of them is unmatched in $\text{OPT}$.

{**OPT** – the optimal solution}

---

Recursive formula

Let us just consider the last character of $x$ and $y$.

**Case (I):** $x[m] = y[n]$.  
Claim: $c[m, n] = c[m-1, n-1] + 1$.

**Proof:**
Recursive formula

Let's just consider the last character of $x$ and of $y$.

**Case (I):** $x[m] = y[n]$. Claim: $c[m, n] = c[m-1, n-1] + 1$.

**Proof:**

We claim that there is a max matching that matches $x[m]$ to $y[n]$.

Indeed, if $x[m]$ is matched to $y[k]$ (for $k < m$) then $y[n]$ is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by matching $x[m]$ to $y[n]$.

This implies that we can find an optimal matching of

$LCS(x[1..m-1], y[1..n-1])$, and add the segment $(x[m], y[n])$.

So $c[m, n] = c[m-1, n-1] + 1$.

---

Recursive formulation-cont

**Case (II):** $x[m] \neq y[n]$. Claim: $c[m, n] = \max\{c[m, n-1], c[m-1, n]\}$

Recall - in $LCS(x[1 \ldots m], y[1 \ldots n])$ it cannot be that both $x[m]$ and $y[n]$ are both matched.

If $x[m]$ is unmatched in OPT then

$LCS(x[1 \ldots m], y[1 \ldots n]) = LCS(x[1 \ldots m-1], y[1 \ldots n])$

If $y[n]$ is unmatched in OPT then

$LCS(x[1 \ldots m], y[1 \ldots n]) = LCS(x[1 \ldots m], y[1 \ldots n-1])$

So $c[m, n] = \max\{c[m-1, n], c[m, n-1]\}$.

---

$c[i, j]$ For general $i, j$

Since we only care for OPT matching the prefixes, then

**Case (I):** $x[i] = y[j]$.

Claim: if $x[i] = y[j]$ then $c[i, j] = c[i-1, j-1] + 1$.

---

$c[i, j]$ For general $i, j$

Since we only care for OPT matching the prefixes, then

**Case (I):** $x[i] = y[j]$.

Claim: if $x[i] = y[j]$ then $c[i, j] = c[i-1, j-1] + 1$. 

---
c[i,j] For general i,j
Since we only care for OPT matching the prefixes, then
Case (I): x[i] = y[j].
Claim: if x[i] = y[j] then c[i, j] = c[i-1, j-1] + 1.

We claim that there is a max matching that matches x[i] to y[j].

Indeed, if x[i] is matched to y[k] (for k<j) then y[j] is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by match x[i] to y[j].

This implies that we can match x[1..i-1] to y[1..j-1], and add the match (x[i],y[j]). So c[i, j] = c[i-1, j-1] + 1

Dynamic-programming hallmark #1

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Recursive formulation-cont

Case (II): if x[i] ≠ y[j] then c[i, j] = max{c[i-1, j], c[i, j-1]}

Recall - in LCS(x[1..i], y[1..j]) it cannot be that both x[i] and y[j] are both matched.

If x[i] is unmatched then
LCS(x[1..i], y[1..j]) = LCS(x[1..i-1], y[1..j])

If y[j] is unmatched then
LCS(x[1..i], y[1..j]) = LCS(x[1..i], y[1..j-1])

So c[i, j] = max{c[i-1, j], c[i, j-1]}

Dynamic-programming hallmark #1

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.
Recursive algorithm for LCS

\[
\text{LCS}(x, y, i, j) \\
\text{if } (i = 0 \text{ or } j = 0) \text{ return } 0 \\
\text{if } x[i] = y[j] \\
\text{ then return } \text{LCS}(x, y, i-1, j-1) + 1 \\
\text{else return } \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}
\]

To call the function \( \text{LCS}(x, y, m, n) \)

**Worst-case:** \( x[i] \neq y[j] \), for all \( i,j \) in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

\[
m = 3, n = 4: \\
\text{Height } = m + n \Rightarrow \text{work potentially } 2^{m+n} \text{ exponential.}
\]
**Recursion tree**

$m = 3, n = 4$:

- Height = $m + n = 3 + 4 = 7$.
- Work potentially $2^{m+n}$ exponential.
- But we’re solving subproblems already solved!

**Dynamic-programming hallmark #2**

- **Overlapping subproblems**
  
  A recursive solution contains a “small” number of distinct subproblems repeated many times.

**Memoization algorithm**

**Memoization**: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

The number of distinct LCS subproblems for two strings of lengths $m$ and $n$ is only $mn$. 

**Dynamic-programming hallmark #2**

- **Overlapping subproblems**
  
  A recursive solution contains a “small” number of distinct subproblems repeated many times.
**Memoization algorithm**

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

\[
\text{LCS}(x, y) \\
\text{for } i=0 \text{ to } m \quad c[i, 0] = 0 \\
\text{for } j=0 \text{ to } n \quad c[0, j] = 0 \\
\text{for } i=1 \text{ to } m \text{ for } j=1 \text{ to } n \\
\quad \text{if } (x[i] = y[j]) \quad \text{then } c[i, j] \leftarrow c[i-1, j-1] + 1 \\
\quad \text{else } c[i, j] \leftarrow \max\{c[i-1, j], c[i, j-1]\}
\]

Time = \(\Theta(mn)\) = constant work per table entry.
Space = \(\Theta(mn)\).

**Reconstruction** \(z = \text{LCS}(x, y)\)

**IDEA:** Compute the table bottom-up. Fill \(z\) backward.

\[
\text{LCS}(x, y) \\
\text{for } i=0 \text{ to } m \quad c[i, 0] = 0 \\
\text{for } j=0 \text{ to } n \quad c[0, j] = 0 \\
\text{for } i=1 \text{ to } m \text{ for } j=1 \text{ to } n \\
\quad \text{if } (x[i] = y[j]) \quad \text{then } c[i, j] \leftarrow c[i-1, j-1] + 1 \\
\quad \text{else } c[i, j] \leftarrow \max\{c[i-1, j], c[i, j-1]\}
\]

\[
\text{LCS}(x, y) \\
\text{for } i=0 \text{ to } m \quad c[i, 0] = 0 \\
\text{for } j=0 \text{ to } n \quad c[0, j] = 0 \\
\text{for } i=1 \text{ to } m \text{ for } j=1 \text{ to } n \\
\quad \text{if } (x[i] = y[j]) \quad \text{then } c[i, j] \leftarrow c[i-1, j-1] + 1 \\
\quad \text{else } c[i, j] \leftarrow \max\{c[i-1, j], c[i, j-1]\}
\]

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

**LCS Reconstruction:**

Set \(i=m; \ j=n; \ k=c[i,j]\)

While(\(k>0\)){

\[
\text{if } (c[i,j]>c[i-1,j] \text{ and } c[i,j]>c[i,j-1]) \{ \\
\quad z[k]=x[i]; \\
\quad i--; j--; k--; \\
\} \text{else } // c[i,j]=c[i-1,j] \text{ or } c[i,j]=c[i,j-1] \\
\text{if } (c[i,j]==c[i,j-1]) \quad j--; \\
\text{else } i--; \\
\}
\]
### Reconstruction \( z = LCS(x, y) \)

**IDEA:** Compute the table bottom-up. Fill \( z \) backward. 

\( LCS(x,y)=“BCBA” \)

**Observation:** \( c[i][j] \geq c[i-1][j] \) and \( c[i][j] \geq c[i][j-1] \)

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

LCS Reconstruction:

Set \( i=m; j=n; k=c[i][j] \)

While \((k>0)\) {

- if \((c[i][j]>c[i-1][j] \text{ and } c[i][j]>c[i][j-1])\) {
  - \( z[k]=x[i] \);
  - \( i--; j--; k-- \);
}

- else // \( c[i][j]=c[i-1][j] \) or \( c[i][j]=c[i][j-1] \)
  - \( z[k]=c[i][j] \);
  - \( i--; j--; k-- \);

- \}

\( x=B \ D \ C \ A \ B \ A \)

\( y=A \ B \ C \ B \ D \ A \ B \)

<table>
<thead>
<tr>
<th></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>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>1B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### Reconstruction \( z = LCS(x, y) \)

**IDEA:** Compute the table bottom-up. Fill \( z \) backward. 

\( LCS(x,y)=“BCBA” \)

**Observation:** \( c[i][j] \geq c[i-1][j] \) and \( c[i][j] \geq c[i][j-1] \)

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

LCS Reconstruction:

Set \( i=m; j=n; k=c[i][j] \)

While \((k>0)\) {

- if \((c[i][j]>c[i-1][j] \text{ and } c[i][j]>c[i][j-1])\) {
  - \( z[k]=x[i] \);
  - \( i--; j--; k-- \);
}

- else // \( c[i][j]=c[i-1][j] \) or \( c[i][j]=c[i][j-1] \)
  - \( z[k]=c[i][j] \);
  - \( i--; j--; k-- \);

- \}

\( x=B \ D \ C \ A \ B \ A \)

\( y=A \ B \ C \ B \ D \ A \ B \)

<table>
<thead>
<tr>
<th></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>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>1B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**Reconstruction $z=LCS(x,y)$**

**IDEA:** Compute the table bottom-up. Fill $z$ backward.

LCS Reconstruction:
Set $i=m$; $j=n$; $k=c[i;j]$
While($k>0$)
  if ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i-1;j-1]$) {
    $z[k] = x[i]$ ;
    $i--; j--; k--;$
  } else if ($c[i;j]=c[i-1;j]$) or $c[i;j]=c[i-1;j-1]$
    $i--;$
  }

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

---

**Reconstruction $z=LCS(x,y)$**

**IDEA:** Compute the table bottom-up. Fill $z$ backward.

LCS Reconstruction:
Set $i=m$; $j=n$; $k=c[i;j]$
While($k>0$)
  if ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i-1;j-1]$) {
    $z[k] = x[i]$ ;
    $i--; j--; k--;$
  } else if ($c[i;j]=c[i-1;j]$) or $c[i;j]=c[i-1;j-1]$
    $i--;$
  }

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

---

**Reconstruction $z=LCS(x,y)$**

**IDEA:** Compute the table bottom-up. Fill $z$ backward.

LCS Reconstruction:
Set $i=m$; $j=n$; $k=c[i;j]$
While($k>0$)
  if ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i-1;j-1]$) {
    $z[k] = x[i]$ ;
    $i--; j--; k--;$
  } else if ($c[i;j]=c[i-1;j]$) or $c[i;j]=c[i-1;j-1]$
    $i--;$
  }

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.

---

**Reconstruction $z=LCS(x,y)$**

**IDEA:** Compute the table bottom-up. Fill $z$ backward.

LCS Reconstruction:
Set $i=m$; $j=n$; $k=c[i;j]$
While($k>0$)
  if ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i-1;j-1]$) {
    $z[k] = x[i]$ ;
    $i--; j--; k--;$
  } else if ($c[i;j]=c[i-1;j]$) or $c[i;j]=c[i-1;j-1]$
    $i--;$
  }

**Proof Sketch:** We use a longer prefix, so there are more chars to be match.
Reconstructing \( z = \text{LCS}(X,Y) \)

Another idea – While filling \( c[i,j] \), add arrows to each cell \( c[i,j] \) specifying which neighboring cell \( c[i,j] \) it got its value.

- \( c[i,j].\text{flag} = \text{"\"} \) if \( c[i,j] = c[i-1,j-1]+1 \)
- \( c[i,j].\text{flag} = \text{"\"} \) if \( c[i,j] = c[i,j-1]+1 \)
- \( c[i,j].\text{flag} = \text{"\"} \) if \( c[i,j] = c[i-1,j] \)

Example 2: Edit distance

Given strings \( X,Y \), the edit distance \( \text{ed}(X,Y) \) between \( X \) and \( Y \) is defined as the minimum number of operations that we need to perform on \( X \), in order to obtain \( Y \).

**Definition:** An Operations (in this context) Insertion/Deletion/Replacement of a single character.

Examples:

\[
\begin{align*}
\text{ed}("aaba", "aaab") &= 0 \\
\text{ed}("aaa", "abaa") &= 1 \\
\text{ed}("aaaa", "abaa") &= 1 \\
\text{ed}("baaa", "aaab") &= 2
\end{align*}
\]

Note that the term “distance” is a bit misleading: We need both the value (how many operations) as well as knowing which operations.

Example 3’:

```
``Priced” Edit distance \( \text{ed}(X,Y) \)

Assume also given

- \( \text{InsCost} \) - the cost of a single insertion into \( x \).
- \( \text{DelCost} \) - the cost of a single deletion from \( x \), and
- \( \text{RepCost} \) - the cost of replacing one character of \( x \) by a different character.

**Definition:** Given strings \( X,Y \), the edit distance \( \text{ed}(X,Y) \) between \( X \) and \( Y \) is the cheapest sequence of operations, starting on \( X \) and ending at \( Y \).

**Problem:** Compute \( \text{ed}(X,Y) \), (both the value and the optimal sequence of operations.)

**Definition:** \( \text{c[i,j]} = \text{Cost}(\text{ed}(X[1..i], Y[1..j])) \).

Will first compute \( \text{Cost}(c[m,n]) \). Then will recover the sequence.

Thm:

Let \( \text{c[i,j]} = \text{ed}(x[1..i], y[1..j]) \).
Assume \( \text{c[i-1,j-1]}, \text{c[i-1,j]}, \text{c[i,j-1]} \) are already computed.

If \( X[i] = Y[j] \) then \( \text{c[i,j]} = \text{c[i-1,j-1]} \)
Else // \( X[i] \neq Y[j] \)
\[
\text{c[i,j]} = \min\{ \\
\text{c[i-1,j-1]} + \text{RepCost}, \text{convert } X[1..i-1] \rightarrow Y[1..j-1], \text{ and replace } y[j] \text{ by } x[i] \\
\text{c[i-1,j]} + \text{DelCost}, \text{ delete } X[i] \text{ and convert } X[1..i-1] \rightarrow Y[1..j] \\
\text{c[i,j-1]} + \text{InsCost}, \text{ convert } X[1..i] \rightarrow Y[1..j-1], \text{ and insert } Y[i] \}
\}
Algorithm

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```plaintext
ed(X, Y)
for i=0 to m  c[i, 0] = i DelCost
for j=0 to n  c[0, j] = j InsCost

for i=1 to m
  for j=1 to n
    if (X[i] == Y[j])
      then c[i, j] ← c[i-1, j-1] + DelCost,
    else c[i, j] ← min{
      c[i-1, j] + DelCost,
      c[i-1, j-1] + RepCost,
      c[i, j-1]    + InsCost
    }
```

Time = Θ(m n) = constant work per table entry. Space = Θ(m n).

Homework: Compute the sequence of operations. Compute which characters in x matches which chars in y.

---

Polygonal Path - definition

We define a polygonal path \( P=\{p_1...p_n\} \) where

- Each vertex \( p_i \) is a point in the plane,
- Vertex \( p_1 \) is the first vertex, \( p_n \) is the last,
- Vertex \( p_i \) is connected to the next vertex \( p_{i+1} \) by a straight segment.

\[
P_1 \quad p_2 \quad p_3 \quad p_4 \quad p_n
\]
**Good ways to measure distance between curves**

- Should not be effected by how curves are sampled
- Should reflect the “order” of the points along the curves.

\[ P[1..i] \] is the polygonal line with the first \( i \) vertices of \( P \)

\[ Q[1..j] \] is the polygonal line with the first \( j \) vertices of \( P \)

**Problem: Computing the Frechet Distance between polylines**

\[ \text{Frechet}(P, Q, r) \]

Definition of \( \text{Frechet}(P, Q, r) \)

Assume a person walks on \( P = \{p_1, \ldots, p_n\} \) while a dog walks on \( Q = \{q_1, q_2, \ldots, q_n\} \).

- \( r \) is the leash length (part of input).
- The person starts at \( p_1 \) and ends at \( p_n \).
- The dog starts at \( q_1 \) and ends at \( q_n \).

At each time stamp,
- either the person jumps to the next vertex
- or the dog jumps to the next vertex
- or both jumps to the next vertex

• Every instance they stop, we measure whether the distance between person ↔ dog (the length of the leash) \( \leq r \).

• \( \text{Frechet}(P, Q, r) = \text{YES} \) if the answer is positive for all time stamps.

• (if not, a longer leash is needed. If yes, maybe a shorter one is sufficient.)

• So we could use binary search.

- Every instance they stop, we measure whether the distance between person ↔ dog (the length of the leash) \( \leq r \).

- \( \text{Frechet}(P, Q, r) = \text{YES} \) if the answer is positive for all time stamps.

- (if not, a longer leash is needed. If yes, maybe a shorter one is sufficient.)

- So we could use binary search.
Computing Frechet(P,Q,r)

Frechet(P,Q,r)
// c[1..n, 1..n] – boolean array
// c[i][j]= Frechet(P[1..i],[Q[1..j], r )

Init:
c[1,1]= (|| p_1 - q_1 || ≤ r ) (YES/NO)
For i=2 to n
c[i,1]= (|| p_i - q_1 || ≤ r ) AND c[i-1,1] (YES/NO)
For j=2 to n
\[ c[1,j] = ( || p_1 - q_j || ≤ r ) AND c[1,j-1] \]

Comments

- This is actually the **Discrete** Frechet Distance (only distances between vertices counts). We do not discuss the **continuous** version.
- This is only the Decision problem – we actually want the shortest leash. We could use a binary search to approximate it. Exact algorithm outside the scope of this course.
- If person/dog could move backward, the problem is called the **weak** Frechet.

Maurice René Fréchet

Computing Frechet(P,Q,r) (cont.)

// c[1..n, 1..n] – boolean array

Init- previous slide

For i=2 to n
For j=2 to n
\[ c[i,j] = ( || p_i - q_j || ≤ r ) \]
\{ c[i-1,j-1] // both jumps
OR c[i-1,j] // person jumped from p_{i-1} to p_i, dog stays at q_j
OR c[i,j-1] // person stayed at p_i, dog jumped from q_{j-1} to q_j
\}

Return c[n,n]

Note – this is only the cost (that is the distance itself. We still need to find what is the series of steps that yield this cost

Problem: Computing Dynamic Time Warping \( dtw(P,Q) \) between polylines

Given 2 polygonal curves 
\[ P=\{p_1,\ldots,p_n\} \quad \text{and} \quad Q=\{q_1,\ldots,q_m\}, \]
The input is the locations of their vertices (e.g. GIS coordinates)

How similar are \( P \) to \( Q \)?

Need to come up with a number \( dtw(P,Q) \)?
So if \( dtw(P,Q)< dtw(P,Q') \), then \( P \) is more similar to \( Q \)
Problem: Computing Dynamic Time Warping $dtw(P, Q)$ between polylines

Given 2 polygonal curves $P=\{p_1, \ldots, p_n\}$ and $Q=\{q_1, \ldots, q_m\}$, the input is the locations of their vertices (e.g. GIS coordinates).

How similar are $P$ to $Q$?

Need to come up with a number $dtw(P, Q)$?

So if $dtw(P, Q) < dtw(P, Q')$, then $P$ is more similar to $Q$.

Definition of $dtw(P, Q)$

Assume a person walks on $P=\{p_1, \ldots, p_n\}$ while a dog walks on $Q=\{q_1, \ldots, q_m\}$.

They person starts at $p_1$ and ends at $p_n$.

They dog starts at $q_1$ and ends at $q_n$.

At each time stamp,
- either the person jumps to the next vertex
- or the dog jumps to the next vertex
- or both jumps to the next vertex

- Every instance they stop, we measure the distance (the length of the leash) person→dog.
- We sum the lengths of all leashes.
- $dtw(P, Q)$ is the smallest sum (over all possible sequences).

Motivation:

Distance between trajectories enables finding nearest neighbor, and clustering.

But two very similar trajectories might have vertices in very different places.

DTW is used in
- Signal processing (speech reco)
- Signature verification
- Analysis of vehicles trajectories for roads networks
- Improving locations-based services
- Animals migrations patterns
- Stocks analysis
**Thm 1:**
Let \( c[i,j] = \text{dtw}( P[1..i], Q[1..j] ) \).

Let \( || p_i - q_j || \) be the between the points \( p_i \) and \( q_j \).
That is, the length of the leash.

For every \( i>1 \), \( j>1 \)
\[
c[1,1] = || p_1 - q_1 ||
\]
\[
c[i,j] = c[i,j-1] + || p_i - q_j ||
\]
\[
c[i,1] = c[i-1,1] + || p_i - q_1 ||
\]

**Thm 2:**
Assume at some time, the person is at \( p_i \) while dog at \( q_j \).
Assume \( i>1 \) and \( j>1 \).

What (might have) happened one step ago?

Three possibilities

- Both person and the dog jumped (from \( p_{i-1} \) and from \( q_j \)) OR
- Person jumped from \( p_{i-1} \) to \( p_i \), dog stays at \( q_j \) OR
- Person stayed at \( p_i \), dog jumped from \( q_{j-1} \) to \( q_j \).

**Thm 2 cont:**
Let \( c[i,j] = \text{dtw}( P[1..i], Q[1..j] ) \).

If \( i>1 \) and \( j>1 \) then
\[
c[i,j] = || p_i - q_j || + \min\{ \\
\quad c[i-1,j-1], \quad \text{// both jumps} \\
\quad c[i-1,j], \quad \text{// person jumped from } p_{i-1} \text{ to } p_i \text{, dog stays at } q_j \\
\quad c[i,j-1], \quad \text{// person stayed at } p_i \text{, dog jumped from } q_{j-1} \text{ to } q_j \\
\}
\]

Since we are not sure that when the person is at \( p_i \) the dog is at \( q_j \) we will compute all such pairs \( i,j \) – one of them must happened.

**Algorithm for computing dtw(P,Q)**
Init according to Thm 1.

For \( i=2 \) to \( n \)
For \( j=2 \) to \( n \)
\[
c[i,j] = || p_i - q_j || + \min\{ \\
\quad c[i-1,j-1], \quad \text{// both jumps} \\
\quad c[i-1,j], \quad \text{// person jumped from } p_{i-1} \text{ to } p_i \text{, dog stays at } q_j \\
\quad c[i,j-1], \quad \text{// person stayed at } p_i \text{, dog jumped from } q_{j-1} \text{ to } q_j \\
\}
\]

Return \( c[n,n] \)
Note – this is only the cost (that is the distance itself. We still need to find what is the series of steps that yield this cost.
Dynamic-programming hallmark #1
(we saw this slide already)

**Optimal substructure**
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1
(we saw this slide already)

**Optimal substructure**
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of $z$ is an LCS of a prefix of $x$ and a prefix of $y$.

Dynamic-programming hallmark #2

**Overlapping subproblems**
A recursive solution contains a “small” number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

**Overlapping subproblems**
A recursive solution contains a “small” number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths $m$ and $n$ is only $mn$. 
Another application of DP: **Clustering**
(source: Kleinberg & Tardos 6.3)

**Clustering Problem**
- Given points \( P = \{p_1, p_2, \ldots, p_n\} \) sorted from left to right, and a penalty \( R \), find optimal \( k \), and partition of \( P \) into \( k \) runs
  \((p_1, p_2, \ldots, p_3), (p_4, p_5, \ldots, p_6), \ldots, (p_{n-k+1}, \ldots, p_n)\)
  and lines \( \ell_1, \ldots, \ell_k \) (one per each run).
  So that the sum \( R + Err(\ell_1, \{p_1, p_2, \ldots, p_3\}) + \ldots + R + Err(\ell_k, \{p_{n-k+1}, \ldots, p_n\}) \) is as small as possible.

- Note that if \( R = 0 \), we will probably use \( n/2 \) runs \((p_1, p_2, \ldots, p_3), (p_4, p_5, \ldots, p_6), \ldots, (p_{n-k+1}, \ldots, p_n)\).
- If \( R \) is huge, we can afford only one penalty, so only one run \((p_1, \ldots, p_n)\).
- In the example, \( k = 3, i_1 = 5, i_2 = 8 \)

- Worth mentioning: There is no correct value of the penalty \( R \). Instead, think that the user could slowly increase \( R \) from 0 to \( \infty \), watch the number of clusters increases, and stop when the lines seems appropriate.
- The Geogabra applet link could help visualizing this process.

**Algorithm**

- Preprocessing: for every pair of \( i \) and \( j \) (where \( j < i \)) compute the line \( \ell_{j,i} \) that best fit the points \( \{p_j, p_{j+1}, p_{j+2}, \ldots, p_i\} \).
- Let \( c[i] = \text{cost of the cost of the opt clustering of the points } \{p_1, \ldots, p_i\} \). This term includes both the sum of errors and the sum of penalties. At the \( i \)th step of the algorithm, we assume that \( c[0], c[1], c[2], \ldots, c[i-1] \) are already computed, and using these values, we will compute \( c[i] \).
- Init: \( c[0] = 0 \)
- for \( i = 2 \) to \( n \) do { 
  - \( c[i] = \min \{ c[j] + R + e[j+1, i] \} \) such that \( j = 0, 1, 2, \ldots, i-1 \)
  - \( c[i] \) could also 'remember' for which value of \( j \) the minimum is obtained.
}

Idea: \( p_i \) must belong to a cluster. We pay \( R \) for this cluster. The inner loop finds what is the best point \( p_{j+1} \) to be the leftmost point of this cluster.

**Summarizing**

- The algorithm takes \( O(n^3) \) and \( O(n^2) \) space
- (for preprocessing \( d[j,i] \) )
- Note – we did not discuss how to reconstruct the solution itself. We only calculated its cost