Dynamic Programming

Some of the slides are courtesy of Charles Leiserson with small changes by Carola Wenk
Example: All-Pairs Shortest Paths
Floyd-Warshall alg
(Spring 2021)

▪ Given a graph $G(V,E)$ with weights (positive and negative) assign to each edge. Assume $V=\{v_1 \ldots v_n\}$.

▪ Compute a matrix $D$ such that $D[i,j]$ contains the length of the shortest path $v_i \rightarrow v_j$

▪ Also compute a matrix $\Pi[1..n,1..n]$ such that $\Pi[i,j]$ is the vertex that precede $v_j$ along the shortest path $v_i \rightarrow v_j$

▪ Warshall-Floyd Algorithm computes these tables in $O(n^3)$

▪ Can you think about alternative approaches when the weights of all edges is positive?

In the figure to the right, $k = \Pi[i,j]$. Compare to $\Pi[v_i]$ in Dijkstra or Bellman-Ford
We look at sequences of characters (strings)

e.g. \( x = "ABCA" \)

**Def:** A *subsequence* of \( x \) is a 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:**
\( x[1..4] = "ABCA" \) \( x[1..3] = "ABC" \) \( x[1..2] = "AB" \) \( x[1..1] = "A" \) \( x[1..0] = "" \)
**Longest Common Subsequence (LCS)**

- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

$x$: A B C B D A B

$y$: B D C A B A

BCBA = 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.
**Longest Common Subsequence (LCS)**

- Given two sequences $x[1\ldots m]$ and $y[1\ldots n]$, find a longest subsequence common to them both.

  

  “a” not “the”

  $x$: A B C B D A B

  $y$: B D C A B A

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

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 $y$. 
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 = $\Theta((m+n)2^m)$ = 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 $c[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])|$.
• 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 \( OPT \).
\{OPT – the optimal solution\}
Recursive formula

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


Proof.

```
x: 1 2 m

  1 2 = n

y: 1 2 m
```
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

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

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

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

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

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

So $c[m,n] = \max\{c[m-1, n], c[m, n-1]\}$
For general $i,j$

Since we only care for OPT matching the prefixes, then


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


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

\[ x: \quad \begin{array}{ccc|c|c|c} 1 & 2 & \cdots & i & \cdots & m \\ \hline 1 & 2 & \cdots & & \cdots & n \end{array} \quad y: \quad \begin{array}{ccc|c|c|c} 1 & 2 & \cdots & i & \cdots & m \\ \hline 1 & 2 & \cdots & & \cdots & n \end{array} \]
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.
Recursive formulation-cont

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

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

If \( x[i] \) is unmatched then
\[
\text{LCS}(x[1 \ldots i], y[1 \ldots j]) = \text{LCS}(x[1 \ldots i-1], y[1 \ldots j])
\]

If \( y[j] \) is unmatched then
\[
\text{LCS}(x[1 \ldots i], y[1 \ldots j]) = \text{LCS}(x[1 \ldots i], y[1 \ldots j-1])
\]

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.
Dynamic-programming hallmark
#1

**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 $y$. 
Recursive algorithm for LCS

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

To call the function LCS(x, y, m,n )
Recursive algorithm for LCS

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

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

\( m = 3, n = 4: \)

\[
\begin{array}{c}
\text{3,4} \\
\text{2,4} \\
\text{1,4} \\
\text{1,3} \\
\text{2,2} \\
\end{array}
\begin{array}{c}
\text{3,3} \\
\text{2,3} \\
\text{1,3} \\
\text{2,2} \\
\end{array}
\begin{array}{c}
\text{3,2} \\
\text{2,3} \\
\text{1,3} \\
\text{2,2} \\
\end{array}
\]
Recursion tree

$m = 3, n = 4$:

\[ \begin{align*}
3,4 \\
2,4 \\
1,4 \\
2,3 \\
1,3 \\
1,3 \\
2,2 \\
2,2 \\
3,3 \\
3,2 \\
3,2
\end{align*} \]

Height = $m + n$ ⇒ work potentially $2^{m+n}$ exponential.
Recursion tree

$m = 3, n = 4$:

Height $= m + n \Rightarrow$ work potentially $2^{m+n}$ exponential.
but we’re solving subproblems already solved!

same subproblem
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 $mn$. 
Memoization algorithm

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

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

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

for \( i=0 \) to \( m \) \( \quad c[i, 0] = 0 \)
for \( j=0 \) to \( n \) \( \quad c[0, j] = 0 \)

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

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

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

\[
\text{for } i=0 \text{ to } m \quad c[i, 0] = 0
\]

\[
\text{for } j=0 \text{ to } n \quad c[0, j] = 0
\]

\[
\text{for } i=1 \text{ to } m
\]

\[
\text{for } j=1 \text{ to } n
\]

\[
\text{if } (x[i] = y[j]) \\
\quad \text{then } c[i, j] \leftarrow c[i-1, j-1] + 1
\]

\[
\text{else } c[i, j] \leftarrow \max\{ c[i-1, j], c[i, j-1] \}
\]

Time = \(\Theta(mn)\) = constant work per table entry.
Space = \(\Theta(mn)\).
LCS: Dynamic-programming algorithm

LCS(X,Y)="BCBA"

X=B D C A B A

Y=A B C B D A B

Y=  
 1 2 3 4 5 6 7

A 0 0 0 0 0 0 0

B 0 0 1 1 1 1 1

C 0 0 1 1 1 2 2

D 0 0 1 2 2 2 2

E 0 1 1 2 2 2 3

F 0 1 2 2 3 3 3

G 0 1 2 2 3 3 4

H 0 1 2 2 3 3 4

LCS(X,Y)="BCBA"
Reconstruction $z=LCS(x,y)$

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

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

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

**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 \( c[i,j]=c[i-1,j] \)  
  if \((c[i,j]==c[i,j-1]) \) \( j--; \)  
  else \( i--; \) 
}

\[ LCS(x,y)=“BCBA” \]

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

\( y=A\ B\ C\ B\ D\ A\ B \)
Reconstruction $z=LCS(x,y)$

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

**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 ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i;j-1]$) {
    $z[k]=x[i]$;
    $i--; j--; k--;$
  } else if ($c[i;j]=c[i-1;j]$ or $c[i;j]=c[i-1;j]$) {
    if ($c[i;j]==c[i;j-1]$) $j--$;
    else $i--$;
  }
}
**Reconstruction** $z=LCS(x,y)$

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

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 ($c[i;j]>c[i-1;j]$ and $c[i;j]>c[i;j-1]$ ) {
    $z[k] = x[i]$ ;
    $i--; j--; k--;$
  } else // $c[i;j]=c[i-1;j]$ or $c[i;j]=c[i-1;j]$
  if ($c[i;j]==c[i;j-1]$)  $j--;$
  else  $i--;$
}

$LCS(x,y) = \text{“BCBA”}$

$x=B \ D \ C \ A \ B \ A$

$y=A \ B \ C \ B \ D \ A \ B$
Reconstruction \( z=LCS(x,y) \)

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

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

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

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

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

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

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

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

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

\[ x = \text{B D C A B A} \]

\[ y = \text{A B C B D A B} \]
Reconstruction $z = \text{LCS}(x, y)$

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

**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 ($c[i, j] > c[i-1, j]$ and $c[i, j] > c[i, j-1]$) {
    $z[k] = x[i]$; 
    $i$--; $j$--; $k$--; 
  } else if ($c[i, j] = c[i-1, j]$ or $c[i, j] = c[i-1, j]$)
    if ($c[i, j] == c[i, j-1]$) $j$--; 
  else $i$--; 
}
Reconstructing $z = \text{LCS}(X,Y)$

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

- $c[i,j].\text{flag} = "\ \"$ if $c[i,,j]=c[i-1;j-1]+1$
- $c[i,j].\text{flag} = "↑"$ if $c[i,,j]=c[i-1;j ]$
- $c[i,j].\text{flag} = "←"$ if $c[i,,j]=c[i-1;j ]$
- $c[i,j].\text{flag} = "←"$ 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:

\[
\begin{align*}
\text{ed}(\text{“aaba”}, \text{“aaba”}) & = 0 \\
\text{ed}(\text{“aaa”}, \text{“aaba”}) & = 1 \\
\text{ed}(\text{“aaaa”, “abaa”}) & = 1 \\
\text{ed}(\text{“baaa”, “”}) & = 4 \\
\text{ed}(\text{“baaa”, “aaab”}) & = 2
\end{align*}
\]

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 $ed(X,Y)$

Assume also given

- $InsCost$, - the cost of a single **insertion** into $x$.
- $DelCost$ - the cost of a single **deletion** from $x$, and
- $RepCost$ - the cost of **replacing** one character of $x$ by a different character.

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

**Problem:** Compute $ed(X,Y)$, (both the value and the optimal sequence of operations.  )

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

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

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

If \( X[i] = Y[j] \) then \( c[i,j] = c[i-1,j-1] \)
Else

\[
\begin{align*}
  c[i,j] &= \min \{ \\
  & c[i-1,j-1] + \text{RepCost}, \quad \text{//convert } X[1..i-1] \Rightarrow Y[1..j-1], \text{ and replace } y[j] \\
  & c[i-1,j] + \text{DelCost}, \quad \text{//delete } X[i] \text{ and convert } X[1..i-1] \Rightarrow Y[1..j] \\
  & c[i,j-1] + \text{InsCost} \quad \text{//convert } X[1..i] \Rightarrow Y[1..j-1], \text{ and insert } Y[i] \\
  \} 
\end{align*}
\]
Algorithm

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

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

\[
\text{ed}(X, Y) \\
\text{for } i=0 \text{ to } m \quad c[i, 0] = i \text{ DelCost} \\
\text{for } j=0 \text{ to } n \quad c[0, j] = j \text{ InsCost} \\
\text{for } i=1 \text{ to } m \\
\quad \text{for } j=1 \text{ to } n \\
\quad \quad \text{if } (X[i] == Y[j] ) \\
\quad \quad \quad \text{then } c[i, j] \leftarrow c[i-1, j-1] \\
\quad \quad \text{else } c[i, j] \leftarrow \min \{ c[i-1, j] + \text{DelCost}, \\
\quad \quad \quad \quad c[i-1, j-1] + \text{RepCost}, \\
\quad \quad \quad \quad c[i, j-1] + \text{InsCost} \} \\
\]
Algorithm

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

\[
ed(X, Y)\
\begin{align*}
\text{for } i &= 0 \text{ to } m & c[i, 0] &= i \text{ DelCost} \\
\text{for } j &= 0 \text{ to } n & c[0, j] &= j \text{ InsCost} \\
\text{for } i &= 1 \text{ to } m \\
\text{for } j &= 1 \text{ to } n \\
\text{if } (X[i] == Y[j]) \\
\text{then } c[i, j] &= c[i-1, j-1] \\
\text{else } c[i, j] &= \min\{ c[i-1, j] + \text{DelCost}, \\
&\quad c[i-1, j-1] + \text{RepCost}, \\
&\quad c[i, j-1] + \text{InsCost}\} 
\end{align*}
\]

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

Homework: Compute the sequence of operations. Compute which characters in \(x\) matches which chars in \(y\).
We define a polygonal path $P=\{p_1 \ldots p_n\}$ 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.
Good ways to measure distance between curves

- Should not be affected 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 \( Q = \{q_1 \ldots q_n\} \).

\( r \) 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

At 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 need. If yes, maybe a shorter one is sufficient.
- So we could use binary search.
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 \( Q = \{q_1 \ldots q_n\} \). \( r \) 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.
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 \( Q = \{q_1 \ldots q_n\} \).

\( r \) 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\( \leftrightarrow \)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:
\[
c[1,1] = (|| p_1 - q_1 || \leq r ) \text{ (YES/NO)}
\]

For \( i = 2 \) to \( n \)
\[
c[i,1] = (|| p_i - q_1 || \leq r ) \text{ AND } c[i-1,1] \text{ (YES/NO)}
\]

For \( j = 2 \) to \( n \)
\[
c[1,j] = (|| p_1 - q_j || \leq r ) \text{ AND } c[1,j-1]
\]
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 $n$
    $c[i,j] = (||p_i - q_j|| \leq r) \text{ AND}$
    \{
      $c[i-1,j-1]$, \textit{both jumps}
      OR $c[i-1, j]$, \textit{person jumped from } $p_{i-1}$ \textit{to } $p_i$, \textit{dog stays at } $q_j$
      OR $c[ i,j-1]$, \textit{person stayed at } $p_i$, \textit{dog jumped from } $q_{j-1}$ \textit{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
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
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 $Q$
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 \( Q \)
Dynamic Time Warping \( dtw(P,Q) \)

**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\(\leftrightarrow\)dog.
- We sum the lengths of all leashes.
- \( dtw(P,Q) \) is the smallest sum (over all possible sequences)
Dynamic Time Warping \( dtw(P,Q) \)

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 \textbf{person} starts at \( p_1 \) and ends at \( p_n \)
They \textbf{dog} starts at \( q_1 \) and ends at \( q_n \)

At each time stamp,
- either the \textbf{person} jumps to the next vertex
- Or the \textbf{dog} jumps to the next vertex
- Or \textbf{both} jumps to the next vertex

\begin{itemize}
  \item Every instance they stop, we measure the distance (the length of the \textbf{leash}) person\(\leftrightarrow\)dog.
  \item We sum the lengths of all leashes.
  \item \( dtw(P,Q) \) is the smallest sum (over all possible sequences)
\end{itemize}
Motivation:

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

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 \( i > 1 \), \( j > 1 \)
\[
c[1,1] = \| p_1 - q_1 \|
\]
\[
c[1,j] = c[1,j-1] + \| p_1 - q_j \|
\]
\[
c[i,1] = c[i-1,1] + \| p_i - q_1 \|
\]
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 $c[i,j] = dtw(P[1..i], Q[1..j])$.

If $i > 1$ and $j > 1$ then

$$c[i,j] = \|p_i - q_j\| + \min\{c[i-1,j-1], // both jumps\ c[i-1,j] , // person jumped from $p_{i-1}$ to $p_i$, dog stays at $q_j$\ c[i,j-1] . // person stayed at $p_i$, dog jumped from $q_{j-1}$ to $q_j$.\}$$

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 $\text{dtw}(P,Q)$

Init according to Thm 1.

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

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)

Given points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ find a line minimizing $Err(\ell, P)$

$Err(\ell, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$

that is, the sum of squares of vertical distances from each $(x_i, y_i)$ to $\ell$.

Solution

$$a = \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}$$

$$b = \frac{\sum y_i - a \sum x_i}{n}.$$
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
  
  \[(p_1, p_2 \ldots p_{i_1})(p_{i_1+1}, p_{i_1+2} \ldots p_{i_2}), (p_{i_2+1}, \ldots p_{i_3}) \ldots (p_{i_{k-1}+1} \ldots p_n)\]
  
  and lines \( \ell_1 \ldots \ell_k \) (one per each run) So that the sum
  
  \[R + \text{Err}(\ell_1, \{p_1, p_2 \ldots p_{i_1}\}) + \]
  
  \[R + \text{Err}(\ell_2, \{p_{i_1+2} \ldots p_{i_2}\}) + \]
  
  \[\vdots \]
  
  \[R + \text{Err}(\ell_k, \{p_{i_{k-1}+1} \ldots p_n\})\]

  is as small as possible

Note that if \( R=0 \), we will probably use \( n/2 \) runs \((p_1, p_2), (p_3, p_4), \ldots (p_{n-1}, p_n)\).

If \( R \) is huge, we can afford only one penalty, so only one run \((p_1, \ldots p_n)\).

In the example, \( k=3, \ i_1=5, \ i_2=8 \)
• Algorithm:
• Preprocessing: $\forall j < i$: compute the line $\ell$ minimizing the error for the set $\{p_j, p_{j+1} \ldots p_i\}$.
  
  Let $e[j, i] = \text{Err}(\ell, \{p_j, p_{j+1} \ldots p_i\})$

• Idea: Let $c[i] =$ cost of the opt clustering problem for the set $\{p_1 \ldots p_i\}$.

• Init: $c[0] = 0$.

• for $i = 2$ to $n$ do {
  
  $c[i] = \min\{R + c[j] + e[j+1, i] \mid 0 \leq j < i\}$

  }

• return $c[n]$
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