CS 545

Dynamic Programming

Some of the slides are courtesy of Charles Leiserson with small changes by Carola Wenk

Example: Floyd Warshall Algorithm: Computing all pairs shortest paths

Further reading: CLRS

- Given $G(V,E)$, with weight $w(v_i,v_j)$ given on each of its edges (positive or negative), the output is a matrix $D[1..n, 1..n]$ such that (for every $i,j$)
  
  $D[i,j]$ is the length of the shortest path from $v_i$ to $v_j$

- How to find the shortest paths (and not only their costs) will be discussed in the homeworks.
  (analogous to Dijkstra)

- Assume no negative cycles exist in $G(V,E)$.

- In the homework: Finding such cycles.

Assume $V = \{v_1, v_2, \ldots, v_n\}$

**Def** $P_0(i,j)$ is the shortest path $v_i$ to $v_j$ avoiding any vertex from $\{v_{k+1}, \ldots, v_n\}$ as intermediate vertex.

Example: $P_0(i,j)$ could not go through any vertex of $V$.

**Def** $D_0[i,j]$ is its length of $P_0(i,j)$

So if the edge $(v_i, v_j)$ is in $G$ then

$P_0(i,j)=\{(v_i, v_j)\}$

$D_0(i,j)=w(v_i, v_j)$

If the edge $(v_i, v_j)$ is not in $E$, then $D_0(i,j)=+\infty$ (since any path connecting them must use a vertex from $V=\{v_1, v_n\}$)

$v_1 \quad 2 \quad v_2 \quad 4 \quad v_3$

**Def** $P_k(i,j)$ is the shortest path from $v_i$ to $v_j$ avoiding any vertex from $\{v_{k+1}, \ldots, v_n\}$ as an intermediate vertex. (the sets $\{v_{k+1}, \ldots, v_n\}$ is forbidden)

**Def** $D_k[i,j]$ is its length of $P_k(i,j)$

- Assume $D_{k-1}[i,j]$ has been computed ($1 < i, j < n$).
- We now want to compute the matrix $D_k[i,j]$.
- Now we could (but don’t have to) go through $v_k$ along the shortest path $v_i \rightarrow v_j$.

- Two option:
  1. Going through $v_k$ is longer, and we better stick to $P_{k-1}(i,j)$.
  2. Use $P_{k-1}(i,k)$, the shortest path $v_i \rightarrow v_k$, to reach $v_k$, and continue $P_{k-1}(k,j)$ along to $v_j$.

- Conclusion: $D_k[i,j] = \min(D_{k-1}[i,k], D_{k-1}[i,k] + D_{k-1}[k,j])$
Floyd Warshall-Pairs Shortest Paths
Computing \( D_k[i,j] \) for every \( i,j,k \).

Algorithm \( \text{AllPair}(G) \) for all vertex pairs \((i,j)\)

Use \( n \) tables \( D_0, D_n \). Each is an \( n \times n \)
if \( i = j \) then \( D_0[i,i] \leftarrow 0 \)
else if \((v_i,v_j)\) is an edge in \( G \)
\( D_0[i,j] \leftarrow w(v_i,v_j) \)
else
\( D_0[i,j] \leftarrow +\infty \)
for \( k \leftarrow 1 \) to \( n \) do
for \( i \leftarrow 1 \) to \( n \) do
for \( j \leftarrow 1 \) to \( n \) do
\( D_k[i,j] = \min \{ D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j] \} \)
return \( D_n \)

Floyd Warshall-Pairs Shortest Paths
Computing \( D_k[i,j] \) for every \( i,j,k \).

Algorithm \( \text{AllPair}(G) \) for all vertex pairs \((i,j)\)

Use \( n \) tables \( D_0, D_n \). Each is an \( n \times n \)
if \( i = j \) then \( D_0[i,i] \leftarrow 0 \)
else if \((v_i,v_j)\) is an edge in \( G \)
\( D_0[i,j] \leftarrow w(v_i,v_j) \)
else
\( D_0[i,j] \leftarrow +\infty \)
for \( k \leftarrow 1 \) to \( n \) do
for \( i \leftarrow 1 \) to \( n \) do
for \( j \leftarrow 1 \) to \( n \) do
\( D_k[i,j] = \min \{ D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j] \} \)
return \( D_n \)

Floyd’s algorithm: example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Dynamic Programming:
Example 1: Longest Common Subsequence

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.

```
<table>
<thead>
<tr>
<th>x:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
```

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.

---

**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.

**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 $OPT$.

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


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 $k < n$) 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 $LCS(x[1..m-1] to y[1..n-1])$, and add the segment $(x[m],y[n])$. So $c[m,n] = c[m-1,n-1] + 1$.

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

Since we only care for $OPT$ matching the prefixes, then


Proof.

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 $(x[i..i-1],y[j])$, 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 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

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

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

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

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

Height = \( m + n \) \Rightarrow 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.

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.

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

for $i = 0$ to $m$
for $j = 0$ to $n$
c[$i, 0] = 0$
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 $c[i, j] \leftarrow \max\{c[i-1, j], c[i, j-1]\}$

Time = $\Theta(mn) = \text{constant work per table entry}$
Space = $\Theta(mn)$.

LCS: Dynamic-programming algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>Y=A B C B D A</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=B D C A B A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<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>1</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>2</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>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

LCS 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($i > 0$){
if ($c[i-1][j] > c[i][j]$ and $c[i][j] > c[i][j-1]$) {
$z[k] = x[i]$;
$i--; j--; k--$;
} else if ($c[i][j] = c[i][j-1]$ or $c[i][j] = c[i-1][j]$) {
if ($c[i][j] = c[i][j-1]$) $j--$;
else $i--$;
}
\[
\text{LCS}(x, y) = \text{BCBA}
\]

Y=A B C B D A B

X=B D C A B A

z = BCDABA
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 $c[i,j]=c[i-1,j]$ 
• $c[i,j].\text{flag} = \text{”←”}$ if $c[i,j]=c[i-1,j]$ 

Example 3: Edit distance

Further reading (for example): Wikipedia.

Given strings $X,Y$, the edit distance $ed(X,Y)$ 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:

$ed(\text{“aaba”}, \text{“aaba”}) = 0$
$ed(\text{“aaba”}, \text{“aaba”}) = 1$
$ed(\text{“aaba”}, \text{“aba”}) = 1$
$ed(\text{“aaba”}, \text{“aba”}) = 4$
$ed(\text{“aba”, “aba”}) = 2$

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 $x$.
- $RepCost$ - 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 $c[i,j] = ed(x[1..i], y[1..j])$.

Assume $c[i-1,j], c[i,j-1], c[i-1,j-1]$ are already computed.

If $X[i]=Y[j]$ then $c[i,j] = c[i-1,j-1]$
Else if $X[i] \neq Y[j]$

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

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

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

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

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

Polygonal Path - definition

We define a polygonal path \(P=[p_1 \ldots p_n]\) 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.

Example 2

Dynamic Time Warping \(d(P,Q)\)

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 \(d(P,Q)\)?
So if \(d(P,Q) < d(P,Q')\), then \(P\) is more similar to \(Q\)

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.
**Dynamic Time Warping** $\text{dtw}(P,Q)$

**Definition of $\text{dtw}(P,Q)$**

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) $p_i \leftrightarrow q_j$.

We sum the lengths of all leashes.

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

**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 $i>1, j>1$

\[
c[1,1] = \| p_1 - q_1 \|
\]

\[
c[i,j] = c[i-1,j] + \| p_i - q_j \|
\]

\[
c[i,1] = c[i-1,1] + \| p_i - q_1 \|
\]

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

Example 5: Frechet Distance between polylines

Frechet(P, Q,r)

Definition of Frechet(P,Q, r)
Assume a person walks on \( P=\{p_1, p_2, p_3, ..., p_n\} \)
while a dog walks on \( Q=\{q_1, q_2, q_3, ..., q_n\} \).
\( r \) is the leash length.
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 whether the distance between person--dog (the length of the leash) \( \leq r \).

Frechet(P,Q,r)=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.

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

Init:
\[
c[1,1]= (||p_1 - q_1|| \leq r ) \text{ YES} / \text{NO}
\]
For \( i=2 \) to \( n \)
\[
c[i,1]= (||p_i - q_1|| \leq r ) \text{ AND c[i-1,1] \text{ YES} / \text{NO}}
\]
For \( j=2 \) to \( n \)
\[
c[1,j]= (||p_1 - q_j|| \leq r ) \text{ AND c[1,j-1] \text{ YES} / \text{NO}}
\]
Computing Frechet (P,Q,r) (cont.)

\[
\begin{align*}
& \text{// } c[1..n, 1..n] = \text{ boolean array} \\
& \text{Init: previous slide} \\
& \text{For } j = 2 \text{ to } n \\
& \quad \text{For } i = 2 \text{ to } n \\
& \quad \quad c[i,j] = (\| p_i - q_j \| \leq r) \text{ AND} \\
& \quad \quad \text{// both jumps} \\
& \quad \quad \text{OR } c[i-1,j-1], \text{ // person jumped from } p_{i-1} \text{ to } p_i , \text{ dog stays at } q_j \\
& \quad \quad \text{OR } c[i,j-1], \text{ // person stayed at } p_i , \text{ dog jumped from } q_{j-1} \text{ to } q_j \\
& \text{Return } c[n,n] \\
\end{align*}
\]

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

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.

The number of distinct LCS subproblems for two strings of lengths \( m \) and \( n \) is only \( mn \).
Example 5: Polygon Triangulation

- Given a Convex Polygon $P$ with $n$ vertices
- A chord is a segment fully inside $P$, and connecting two vertices of $P$.
- Triangulation of $P$ is a partition of $P$ into triangles using non-crossing chords. (add as many chords as possible, but don’t let them cross)
- Goal: Given $P$, find an optimal triangulation.
- Optimality could be measured in many ways.
- Let's pick one:
  - Cost of triangulation = sum of squares of areas of its triangles is minimum. $\sum \text{area}(\Delta_i)^2$ (smaller is better)

Algorithm – on whiteboard

```
Ideas:
- Consider sub-polygons
- Assemble solutions to smaller problems to form solutions to larger polygons
- A run of $k$ vertices: start at $p_i$, walks CCW to $p_{i+1}, p_{i+2}, \ldots$
- Stop after $k$ vertices are reached
- Don't stop at $p_n$. Consider $p_n+1 = p_1$.
- $\{p_i, p_{i+1}, \ldots p_j\}$ form a polygon with $j-i+1$ vertices.
- $\{p_i, p_{i+1}, \ldots p_n\}$ is the whole $P$.

Define $C[i,j]$ – the cost of opt triangulation of the sub polygon $\{p_i, p_{i+1}, \ldots p_j\}$
```

Init (previous slide)

```
For k=3..n-1 // length of run
    for i=1..n
        j=i+k-1 //Note – not a loop
        for t=i+1 to j-1
            A=c[i,t]+c[t,j]+area(Δi)
            if (A<c[i,j]) then
                c[i,j]=A; Π[i,j]=t

Return c[1,n]
```

Homework: Use Π to reconstruct the opt triangulation

Example 6: Dynamic programming for TSP

- Input: A graph $G(V,E)$, where $d[i,j]$ is the cost of edge $(i,j)$.
- Problem: find a shortest path starting at node 1 and visits each node exactly once. Naïve solution takes $O(|V|^2)$.
- Given $S \subseteq V$, let $C(S, k)$ be the cost of shortest path starting at node 1, visits all nodes in $S$ and ending at node $k$.
- Properties
  - if $S=\{1,k\}$, then $C(S, k) = d(1,k)$ (for $k = 2, 3, \ldots n$)
  - if $|S| > 2$, then $\exists m \in S \setminus \{k\}$ such that $C(S, k) = \min C(S \setminus \{k\}, m) + d[m,k]$

```
C(S, k) = \min C(S \setminus \{k\}, m) + d[m,k]
```

Problems: Need to find $m$. So we check all options.
Algorithm

- for \( k = 2 \) to \( n \) do \( C(\{1, k\}, \{k\}) = d[1, k] \)
- for \( t = 3 \) to \( n \) do
  - for all \( S \subseteq \{1, 2, \ldots, n\} \), \(|S| = t \) do
    - for all \( k \in S \) do
      \[ C(S, k) = \min\{C(S - \{k\}, m) + d[m, k] \mid m \neq k, m \in S\} \]

Every subset \( S \) of \( V \) is evaluated once, and we spend \( O(n) \) time for this subset, total \( O(n^2) \). Space: \( O(2^n) \).

Does it worth the effort?

\[ O(n^2) \text{ vs } O(n!) \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 2^n = 1024 )</th>
<th>( n! &gt; 3.6M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n=10 )</td>
<td>( 2^n = 1M )</td>
<td>( n! &gt; 10^{18} )</td>
</tr>
<tr>
<td>( n=20 )</td>
<td>( 2^n = 10^9 )</td>
<td>( n! &gt; 10^{35} )</td>
</tr>
</tbody>
</table>

Another application: Clustering

- Given \( P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) find a line minimizing \( \text{Err}(\ell, P) \)
- \( \text{Err}(\ell, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \)
  - 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

Further reading Kleinberg & Tardos

Input: Set of points (for example, the x-axis is time.)
- More to the right means later
- Break them into clusters where each cluster could be approximated well by a line

\[ P_1, P_2, P_3, P_4, P_5 \]
\[ \ell_1, \ell_2, \ell_3, \ell_4, \ell_5 \]
Summarizing

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