Example: Floyd Warshall Algorithm:
Computing all pairs shortest paths

- 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 $D[i..n, j..n]$ 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 $P_k(i,j)$** is the shortest path $v_i$ to $v_j$ avoiding any vertex from $\{v_{k+1} \ldots v_n\}$ as intermediate vertex.
Example: $P_k(i,j)$ could not go through any vertex of $V$.

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

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

$$P_k(i,j) = (v_i, v_j)$$
$$D_k(i,j) = w(v_i, v_j)$$

If the edge $(v_i, v_j)$ is not in $E$, then $D_k(i,j) = +\infty$ (since any path connecting them must use a vertex from $V = \{v_1, v_n\}$)
Def $P_{i,j}(k)$ 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 $\{v_k \ldots v_n\}$ is forbidden)
Def $D_{i,j}(k)$ is its length of $P_{i,j}(k)$

Assume $D_{i,j}(k-1)$ has been computed ($1 < i,j < n$).

We now want to compute the matrix $D_{i,j}(k)$.

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 $P_{i,j}(k-1)$, the previous found shortest path $v_i \rightarrow v_j$.
2. Use $P_{i,k}(k)$, the shortest path $v_i \rightarrow v_k$ to reach $v_k$, and continue $P_{k,j}(k)$ along to $v_j$.

Conclusion:
$D_{i,j}(k) = \min(D_{i,j}(k-1), D_{i,k}(k-1) + D_{k,j}(k-1))$

Floyd Warshall-Pairs Shortest Paths
Computing $D_{i,j}(k)$ for every $i,j,k$.

Algorithm AllPair($G$) for all vertex pairs ($i,j$)
Use $n$ tables $D_0 \ldots D_n$. Each is an $n \times n$
if $i = j$ then $D_0[i,j] := 0$
else if ($v_i, v_j$) is an edge in $G$
$D_0[i,j] := w(v_i, v_j)$
else
$D_0[i,j] := +\infty$
for $k$ from 1 to $n$
do
for $i$ from 1 to $n$
do
for $j$ from 1 to $n$
do
$D_k[i,j] = \min(D_k[i,j], D_k[i,k] + D_k[k,j])$
return $D_n$

Floyd’s algorithm: example
Floyd Warshall-Pairs Shortest Paths

Computing $D_{ij}$ for every $i,j,k$.

Algorithm AllPair($G$) for all vertex pairs $(i,j)$
Use $n$ tables $D_0, D_1, \ldots$, each an $n \times n$ table.
- If $i = j$ then $D_0[i,j] \leftarrow 0$
- 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 $D_n$

Running time $O(n^3)$

Space ???

Dynamic Programming:
Example 2: Longest Common Subsequence

We look at sequences of characters (strings)

e.g. $x = \text{"ABCA"}$

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", "ABC"

Def A prefix of $x$, denoted $x[1..m]$, is the sequence of the first $m$ characters of $x$

Examples:

$x[1..4] = \text{"ABC"}$
$x[1..3] = \text{"ABC"}$
$x[1..2] = \text{"AB"}$
$x[1..1] = \text{"A"}$
$x[1..0] = \text{""}$

Example 1: Longest Common Subsequence (LCS)

- Given two sequences $x[1..m]$ and $y[1..n]$, 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 = 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 \( c[i, j] = |\text{lcs}(x[1..i], y[1..j])| \).
• Then, \( c[m, n] = |\text{lcs}(x, y)| \).

Recursive formulation

Theorem.

\[
c[i, j] = \begin{cases} 
  c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\
  \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise}.
\end{cases}
\]

Proof: It is impossible that \( x[i] \) is matched to an element in \( y[1..j-1] \) and in addition \( y[j] \) is matched to an element in \( x[1..i-1] \).
Recursive formulation-cont

Case (I): \( x[i] = y[j] \). Claim: \( c[i, j] = c[i-1, j-1] + 1 \).

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 \( x[1..i-1] \) 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.

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

\[
LCS(x, y, i, j) = \\
\begin{cases} 
0 & \text{if } (i == 0 \text{ or } j == 0) \\
LCS(x, y, i-1, j-1) + 1 & \text{if } x[i] = y[j] \\
\max \{ LCS(x, y, i-1, j), LCS(x, y, i, j-1) \} & \text{else}
\end{cases}
\]

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 \( m + n \) ⇒ 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.

\[
\text{LCS}(x, y) = \begin{cases} 
0 & \text{if } x = \emptyset \text{ or } y = \emptyset \\
\text{LCS}(x[1:], y[1:]) + 1 & \text{if } x[i] = y[j] \\
\max\{ \text{LCS}(x[1:], y[j]), \text{LCS}(x[i], y[1:]) \} & \text{otherwise}
\end{cases}
\]

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

LCS: Dynamic-programming algorithm

LCS(X,Y)=”BCBA”

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

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 matched.
Reconstructing $z = LCS(X,Y)$

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].flag = \) if $c[i,j] = c[i-1,j-1]+1$
- $c[i,j].flag = \uparrow$ if $c[i,j] = c[i-1,j]$
- $c[i,j].flag = \leftarrow$ if $c[i,j] = c[i-1,j]$

**Example 3: Edit distance**

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 operation (in this context) is Insertion/Deletion/Replacement of a single character.

Examples:
- $ed("aaba", "aaba") = 0$
- $ed("aaa", "aaba") = 1$
- $ed("aaa", "abaa") = 1$
- $ed("baba", "") = 4$
- $ed("baba", "aaab") = 2$

**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 $ed(x,y)$, and compute the sequence of operations.
Thm:

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

If $x[i] = y[j]$ then $c[i,j] = c[i-1,j-1]$
Else if $x[i] \neq y[j]$ $c[i,j] = \min$
  $c[i-1,j-1] + \text{RepCost}$, // convert $x[1..i-1] \Rightarrow y[1..j-1]$, and replace $y[j]$ by $x[i]
  c[i-1,j] + \text{DelCost}$, // delete $x[i]$ and convert $x[1..i-1] \Rightarrow y[1..j]
  c[i,j-1] + \text{InsCost}$ // convert $x[1..i] \Rightarrow y[1..j-1]$, and insert $y[j]$

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] \leftarrow c[i-1,j-1]
     else $c[i,j] \leftarrow \min$
         $c[i-1,j] + \text{DelCost}$,
         $c[i,j-1] + \text{RepCost}$,
         $c[i-1,j-1] + \text{InsCost}$

Time = $O(mn)$ = constant work per table entry. Space = $O(mn)$.
```

Dynamic Time Wrapping

- On whiteboard
Dynamic programming for TSP

- **Input:** A graph $(V,E)$, 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. Naive solution takes $O(|V|!)$.
- 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 $k$.
- Properties
  - if $S=\{1,k\}$, then $C(S,k)=d(1,k)$ (for $k=2,3,\ldots,n$)
  - if $|S|>2$, then $\exists m \in S-\{k\}$ such that $C(S,k)=d[m,k] + \text{cost of optimal tour that starts from node 1, ends at } m$, and visits all nodes of $S-\{k\}$. That is, $C(S,k)=C(S-\{k\},m)+d[m,k]$

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?

$O(n^2)$ vs $O(n!)$

<table>
<thead>
<tr>
<th>n</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1024</td>
<td>&gt; 3.6M</td>
</tr>
<tr>
<td>20</td>
<td>1M</td>
<td>&gt; 10^{18}</td>
</tr>
<tr>
<td>30</td>
<td>10^9</td>
<td>&gt; 10^{35}</td>
</tr>
</tbody>
</table>
Another application: Clustering

- 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:
  \[
  a = \frac{n \sum x y - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2} \\
  b = \frac{\sum y - a \sum x}{n}
  \]

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
- \((p_1, p_2, \ldots, p_i), (p_{i+1}, \ldots, p_{i+2}), \ldots, (p_{n-1}, \ldots, p_n)\)
- and lines \( \ell_1, \ldots, \ell_k \) (one per each run) such that the sum
  \[
  R + \sum_{i=1}^{k} Err(\ell_i, \{ p_i, p_{i+1}, \ldots, p_1 \})
  \]
  is as small as possible.
- Note if \( R = 0 \), we probably used \( n \) runs. If \( R \) is huge, we probably finds a single run. In the example probably \( k = 3 \), \( i_1 = 5 \), \( i_2 = 9 \) for most values of \( R \).

Algorithm:

- Preprocessing: For each \( j < i \), compute the line \( \ell \) minimizing the error for the set \( \{ p_j, p_{j+1}, \ldots, p_i \} \).
  Let \( c(j, i) = Err(\ell, \{ p_j, p_{j+1}, \ldots, p_i \}) \).
- 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:
  \[
  c[i] = \min \{ R + c[j] + c[j+1, i] \mid 0 \leq j < i \}
  \]
- Return \( c[n] \).
Summarizing

• The algorithm takes $O(n^4)$ and $O(n^3)$ space
• (for preprocessing $d[i,j]$)
• Note – we did not discuss how to reconstruct the solution itself. We only calculated its cost