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

**Dynamic Programming**

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) problem:**
- Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to them both.
- (Sometimes in the slides we will use $n$ to denote the lengths of both sequences)

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, \( c[m, n] = |\text{LCS}(x, y)| \).

**Structure of LCS- recursive formula**

Consider a matching length with maximal length (\(|\text{LCS}(x,y)|\)).

Let's look at \( x[n] \) (the last character of \( x \)). Three cases:

1. \( x[m] \) is matched and \( x[m] = y[n] \) (e.g., they both contain the same char ‘B’)
2. \( x[m] \) matched and \( x[m] \neq y[n] \) (e.g., \( x[m] = 'B', y[n] = 'C' \))
3. \( x[m] \) is not matched

- Let's consider case 1.
- Claim: we could assume that \( x[m] \) is matched to \( y[n] \). If not, change the matching to include this pair.
- That is, replace the segment \( [m] \) with the segment \( [m], [n] \). The new segment cannot cross any previous segments.

- Conclusion: if \( x[m] = y[n] \) then \( |\text{LCS}(m, n)| = 1 + |\text{LCS}(m-1, n-1)| \)

**case 2 and 3 - \( x[m] \neq y[n] \)**

- 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).
  - That is, at most one of \( x[m], y[n] \) is matched in OPT.
- Conclusion: Deleting the unmatched character from its sequence will not hurt \( \text{LCS}(x,y) \).

\[ |\text{LCS}(x,y)| = \max \left\{ |\text{LCS}(x[1..m-1], y)|, |\text{LCS}(x, y[1..n-1])| \right\} \]

In the example,
\( x[1..m-1] = \text{CCB}, y[1..n-1] = \text{DBC} \)
\( \text{LCS}(x,y) = \text{CC} \)
Putting it together

- For every i,j
- If x[i] = y[j] then c[i, j] = 1 + c[i-1, j-1]
- If x[i] ≠ y[j] then c[i, j] = max{c[i, j-1], c[i-1, j]}

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

\[
\text{LCS}(x, y, i, j) \quad // \text{we start the method with } i=|x|, j=|y| \\
\hspace{1em} \text{if } (i==0 \text{ or } j==0) \text{ return } 0 \\
\hspace{1em} \text{if } x[i] = y[j] \text{ then return } \text{LCS}(x, y, i-1, j-1) + 1 \\
\hspace{1em} \text{else return } \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}
\]

Worst-case: x[i] ≠ 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 $m \times n$.

---

### Memoization algorithm

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

**LCS($x, y$)**

1. **Initialization**:
   - For $i = 0$ to $m$ set $c[i, 0] = 0$.
   - For $j = 0$ to $n$ set $c[0, j] = 0$.
2. **Recurrence**:
   - 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 = $\Theta(mn)$ = constant work per table entry.
Space = $\Theta(mn)$.

---

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

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

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

**LCS Reconstruction:**

1. Set $i = m$; $j = n$; $k = c[i,j]$.
2. 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 $c[i,j] = c[i,j-1]$)
     - If ($c[i,j] = c[i-1,j]$) $j--$;
     - Else $i--$;
3. \}

---

### 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>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>
</tr>
<tr>
<td>2D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>
</tr>
<tr>
<td>4A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</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>
</tr>
<tr>
<td>6A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
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 \( c[i,j] = c[i,j-1] \)

Recursive algorithm for LCS (just an example of what could go wrong)

\[
\text{LCS}(x, i, j) \quad \text{// we start the method with } i=|x|, j=|y| \\
\text{if } (i==0 \text{ or } j==0) \text{ return } 0 \\
\text{if } x[i] = y[j] \\
\text{then return } \text{LCS}(x, y, i-1, j-1) + 1 \\
\text{else return } \max\{\text{LCS}(x, y, i-1, j), \ \text{LCS}(x, y, i, j-1)\}
\]

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

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 \( mn \).
Dynamic Programming:
Example 2: Rectilinear Steiner tree for points on 2 lines

- Input: a set of points (called “terminals”)
- A Steiner tree is a network that connects all of them, and of minimal length.
- This tree (in contrast to Minimum Spanning Tree) could have junctions (called Steiner Points) at points that are not terminals.
- The tree is rectilinear if edges are either parallel or orthogonal to each other.
- Special case: All terminals are on one of two parallel lines.

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

Examples:
- ed(“aaba”, “aaaba”) = 0
- ed(“aaa”, “aaba”) = 1
- ed(“aaaa”, “abaa”) = 1
- ed(“baaa”, “”) = 4
- ed(“baaa”, “aaab”) = 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’:
```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
**Thm:**

Let \( c[i,j] = \text{ed}(x[1..i], y[1..j]) \).

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 // \( X[i] \neq Y[j] \)

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

**Algorithm**

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

**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 \( y \).

**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.

**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**

Frechet($P$, $Q$, $r$)

Definition of 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 $\rightarrow$ 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 need.

So we could use binary search.

### Computing Frechet($P$, $Q$, $r$)

Frechet($P$, $Q$, $r$)

// $c[1..n, 1..n]$ – boolean array

Init:

$c[1,1] = (||p_1-q_1|| \leq r)$ (YES/NO)

For $i = 2$ to $n$

$c[i,1] = (||p_i-q_1|| \leq r)$ AND $c[i-1,1]$ (YES/NO)

For $j = 2$ to $n$

$c[1,j] = (||p_1-q_j|| \leq r)$ AND $c[1,j-1]$.

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.

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

Init- previous slide

For $i = 2$ to $n$

For $j = 2$ to $n$

$c[i,j] = (||p_i-q_j|| \leq r)$ AND

$\{$

$c[i-1,j-1]$, // 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 $c[n,n]$

Another way to think about the Frechet($P$, $Q$, $r$) problem

Create a graph $G(V, E)$ where

$V = \{v_{i,j} \mid p_i, q_j$ are vertices of $P$ and $Q\}$

That is, create a node for every pair of positions of (person, dog).

Edges connect two vertices that are legal move (person moves, dog moves, or both).

$v_{i,j} \rightarrow v_{i+1,j}$, $v_{i,j} \rightarrow v_{i,j+1}$, $v_{i,j} \rightarrow v_{i+1,j+1}$.

Note $|V|$, $|E|$ are both $O(n^2)$.

Color each vertex are blue/red.

$v_{i,j}$ is blue iff $||p_i-q_j|| \leq r$.

$v_{i,j}$ is red iff $||p_i-q_j|| > r$.

Find a path from $v_{1,1}$ to $v_{n,n}$, that uses only blue vertices. (BSF)
Continuous Frechet \((P, Q, \varepsilon)\). Free Space Diagram.

(person, dog) could be anywhere along their paths, as long as distance between them \(\leq \varepsilon\).

- The person and the dog move continuously along their trajectories. However, the distance between them must always be \(\leq \varepsilon\) (given max leash distance).
- A “continuous” graph, which \(\infty\) vertices.
- Each point in the \((x, y)\) in the plane is associated with a pair person/dog location
- How ? Person walks distance of \(x\) (starting at beginning of \(P\))
- Dogs walks distance of \(y\) from (starting at beginning of \(Q\))
- For every segment \(s_i\) of \(P\), and every segment \(e_j\) of \(Q\) compute the (boundary of) all pts representing acceptable locations (distance \(\leq \varepsilon\)). This takes \(O(1)\) per a pair of segments.
- A point is black/white iff it represents a locations where distance acceptable. The resulting shape is called the Free-Space Diagram (term borrowed from robotics)
- Find a path inside the white region of the Free-Space diagram \(O(n^2)\) time

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 \(Q\)

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

\(dtw(P,Q)\) is the smallest sum (over all possible sequences)

Every instance they stop, we measure the distance (the length of the leash) person\(\rightarrow\)dog.
We sum the lengths of all leashes.

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 $c[i,j] = dtw(P[1..i], Q[1..j])$.

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

For every $i>1, j>1$
\[
\begin{align*}
  c[1,1] &= \|p_1 - q_1\| \\
  c[i,1] &= c[i-1,1] + \|p_i - q_1\| \\
  c[i,j] &= c[i,j-1] + \|p_i - q_j\|
\end{align*}
\]

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 $q_j$.

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

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

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.

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.
Ideas:
- Consider sub-polygons
- Assemble solutions to smaller problems to form solutions to larger polygons
- Let $k$, the run-length, be an integer $1 \leq k \leq n$
- A run of $k$ vertices:
  - Start at $p_i$, walks $k$ vertices CCW ($p_i, p_{i+1}, p_{i+2}, \ldots, p_{i+k-1}$)
  - Indexes are computed modulo $n$, so a run of $k=d$ starting at $p_{n-1}$ is ($p_{n-1}, p_0, p_1, \ldots, p_k$)
  - We should have written ($p_i, p_{i+1}, \ldots, p_{i+(k-1)}$) mod $n$.
  - We omit the mod notation when there is no risk of confusion.
- ($p_0, \ldots, p_{n-1}$) is the whole $P$ (a run of length $n$)

Another Example of DP:
Polygon Triangulation
This is an interesting example because of the invariants - which cells are already calculated, and at which order to visit new cells so we only use already-computed values
- Given: a polygon $P$. It is given by specifying its vertices in the order they appear along the boundary CounterClock Wise (CCW).
- 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 each other)
- Goal: Given $P$, find a triangulation $T$ that minimizes the cost

\[
\text{Cost: } \sum_{\Delta \in T} (\text{Area}(\Delta))^2
\]

Cost of triangulation = sum of squares of areas of its triangles

\[
\text{(note that without the square, all triangulations are equally good)}
\]
- To present the algorithm, we could assume that $P$ is convex (Otherwise we just ignore some chords.)
- Fact: The number of triangles is always $n - 2$, and number of chords is $n - 3$
Remember
\( \Pi[i,j] \) – the cost of opt triangulation of the sub polygon defined by the vertices \( \{ p_i, p_{i+1}, p_j \} \)
\( \Pi[i,j] \) contains the third vertex of the triangle that uses the edge \( p_i p_j \).

**Init part of the algorithm:**
// The init handles runs of length 0, 1 and 2.
For \( i=1 \) to \( n \)
   For \( j=1 \) to \( n \) { \( c[i,j]=\infty \); \( \Pi[i,j]=\text{NULL} \}
For \( i=1 \) to \( n \) { 
  \( c[i,i]=0 \); \( \Pi[i,i]=p_i \); 
  \( c[i,i+1]=0 \); \( \Pi[i,i+1]=p_i \); 
// polygon that is a point or an edge has zero area.
  \( c[i,i+2]=\text{Area}^2(\Delta p_i p_{i+1} p_{i+2}) \);
  \( \Pi[i,i+2]=p_{i+1} \);
// Every run of length 3 defines a triangle 
}
https://www.geogebra.org/m/wq6snzgc

**Example 6: Dynamic programming for TSP**
- **Input**: A graph \( G(V,E) \), where \( w(v_i,v_j) \) is the cost of edge \( w(v_i,v_j) \).
- **Problem**: find a shortest path starting at vertex \( v_1 \) and visits each node exactly once. Naive solution takes \( O(|V|^2) \).
- Given \( S \subseteq V \) let \( C(S,v_j) \) be the cost of shortest path starting at node \( v_1 \), visits all nodes in \( S \) and ending at vertex \( v_k \).
- Properties
  - if \( S=\{1,v_j\} \), then \( C(S,v_j) = w(1,v_j) \) (for \( k=2,3,...n \))
  - for large sets \( S \) and any node \( v_k \in S \), lets analyze the path \( \pi \) of \( C(S,v_k) \). This is the shortest path that starts at \( 1 \), ends at \( v_k \), and visits all nodes of \( S \). Let \( v_i \) be the node of \( \pi \) just before node \( k \). Let \( \pi' \) denote the prefix of \( \pi \), from 1 till \( v_k \).
  - The cut-N-paste lemma implies that this \( \pi' \) is the shortest path that starts at 1, ends at \( v_k \), and visits all vertices of \( S-v_k \). So \( \pi' \) is the solution to \( C(S-v_k,v_k) \).

**Problem**: Need to find \( v_i \). So we check all options.

\[ C(S,v_j) = \min_{v_i \in S-v_k} \{ C(S-v_k,v_i) + w(v_i,v_j) \} \]

**Algorithm**

**Init (previous slide)**

For \( k=4..n-1 \) // length of run 
   for \( t=0..n-1 \)
   \( j=i+k-1 \) // note – not a loop
   for \( t=i+1 \) to \( j-1 \)
   \[ A=c[i,j]+c[i,j]+\text{area}^2 \Delta(p_i p_j p_k) \]
   if \( (A<c[i,j]) \) then 
   \[ c[i,j]=A \]
   \[ \Pi[i,j]=t \]

**Return \( c[1,n] \)**

**Homework**: Use \( \Pi \) to reconstruct the triangulation

https://www.geogebra.org/m/wq6snzgc

Let \( C(S,k) \) denote the shortest path that starts at node 1, ends at node \( k \), and visits every node of \( S \)

for \( k=2\ldots n \) do \( C(\{1,k\},k)=w(1,k) \);

for \( t=2\ldots n \) do 
   for all subsets \( S \subseteq V \). \( |S|=t \), \( v_1 \in S \)
   \( C(S,v_k) = \min_{v_{k-1} \in S} \{ C(S-v_k,v_{k-1}) + w(v_k,v_{k-1}) \} \) % takes \( |S|-2 \) checks.

Every subset \( S \) of \( V \) is evaluated \( (|S|-1)(|S|-2) \) times, once per every vertex of \( S \), and every vertex \( v_k \in S \) for every \( v_k \in S \), \( v_k \neq 1 \)

There are \( 2^n - 1 \) subsets that contains vertex 1.
Total time \( \Theta(n^2 \cdot 2^n) \). Space: \( \Theta(n \cdot 2^n) \)
Does it worth the effort?

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

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

Clustering Problem

Given a point set \(P = \{p_1, p_2, \ldots, p_n\}\) sorted from left to right, and a cluster penalty \(R > 0\).

Problem: Find a partition of \(P\) into runs (clusters)

\[ R + \text{Err}(\ell_1, (p_1, p_2, \ldots, p_i)) + R + \text{Err}(\ell_2, (p_{i+1}, p_{i+2}, \ldots, p_j)) + \ldots + R + \text{Err}(\ell_p, (p_{n-1}, p_n)) \]

and lines \(\ell_1, \ldots, \ell_p\) such that the total clustering cost, \(t\text{ct}([p_1, p_2, \ldots, p_n])\) is as small as possible. We define the total clustering cost, \(t\text{ct}([p_1, p_2, \ldots, p_n])\) as the sum of \(k\) penalties \((k \cdot R)\), plus the sum of the fitting errors between the points in each cluster and the line the fit them best.

\[ t\text{ct}([p_1, \ldots, p_n]) \text{ is defined as the value} \]

\[ R + \text{Err}(\ell_1, (p_1, p_2, \ldots, p_i)) + R + \text{Err}(\ell_2, (p_{i+1}, p_{i+2}, \ldots, p_j)) + \ldots + R + \text{Err}(\ell_p, (p_{n-1}, p_n)) \]

Note that if \(R = 0\) (no penalty on new clusters) then the optimum clustering uses \(\frac{n}{2}\) runs:

\((p_1, p_2), (p_3, p_4), \ldots, (p_{n-1}, p_n)\). If \(R\) is huge, then the opt uses only one cluster, containing all the points.

In the example on top, \(k = 3\), \(i_1 = 5\), \(i_2 = 8\)

Clustering a set of points

- A toy problem: Given an ordered set \(S = \{p_1, \ldots, p_n\}\) sorted by \(x\)-value where \(p_i = (x_i, y_i)\), cluster the set into runs such that the errors of approximating each run is small, and the number of runs is small as well.
- A run \(C_i\) is a consecutive subset of points of \(S\). For example \(C_1 = \{p_1, p_2, \ldots, p_4\}, C_2 = \{p_5, p_6, \ldots, p_9\}, C_3 = \{p_{10}, \ldots\}\)
- The error is the highest-lowest in the cluster. Want to minimize max error among all clusters.
- Alternatively, we could also minimize the distance in each cluster to best-fit horizontal line.
- Problems: Find number of clusters, and find their boundaries.

Another application of DP: Clustering

(source: Kleinberg & Tardos 6.3)

DP meets in a clever way Divide & Conquer.

Given a set of points \(S = \{p_1, \ldots, p_n\}\) and a line \(\ell = ax + b\) are given.

We could measure how well \(\ell\) fits \(S\) by checking, for each \(p_i \in S\), the vertical distance from \(\ell\), and sum the squares of these distances.

\[ \text{Err}(\ell, S) := \sum_{p_i \in S} (ax_i + b - y_i)^2 \quad \text{when } p_i = (x_i, y_i) \]

Next, find the line \(y = a_{\text{opt}} x + b_{\text{opt}}\) that minimizes this error.

\[ 0 = \frac{\partial}{\partial a} \sum (y_i - ax_i - b)^2 = -2 \sum x_i (y_i - ax_i - b) \]
\[ a_{\text{opt}} = \frac{n \sum x_i y_i - \left(\sum x_i\right) \left(\sum y_i\right)}{n \sum x_i^2 - \left(\sum x_i\right)^2} \]
\[ 0 = \frac{\partial}{\partial b} \sum (y_i - ax_i - b)^2 = -2 \sum (y_i - ax_i - b) \]
\[ b_{\text{opt}} = \frac{\sum y_i - n \sum x_i y_i}{n} \]

Definition \(Err(S) = \sum_{p_i \in S} (ax_i + b - y_i)^2\) - fitting error.

Claim (not critical): We can compute \(Err(\{p_i, p_{i+1}, \ldots, p_j\})\) for \(i, j\) in time \(O(n^2)\).
Idea: The rightmost point of each cluster is the ‘cluster-head’, and this point has to pay the penalty $R$. Given a point set $P = \{p_1, p_2, \ldots, p_n\}$ sorted from left to right, and a cluster penalty $R > 0$.

Problem: Find a partition of $P$ into runs (clusters)

$$(p_1, p_2, \ldots, p_i), (p_{i+1}, p_{i+2}, \ldots, p_j), \ldots, (p_{k-1}, p_k, p_{k+1}, \ldots, p_n)$$

and lines $c_1, \ldots, c_5$ such that the total clustering cost, $tc[p_1, p_n]$, is as small as possible. We define the total clustering cost, $tc[p_1, p_n]$, as the sum of $k$ penalties $(k \cdot R)$, plus the sum of the fitting errors between the points in each cluster and the line that fit them best;

$$tc[p_1, p_n] = R + Err(c_1, [p_1, p_2, \ldots, p_i]) + R + Err(c_2, [p_{i+1}, p_{i+2}, \ldots, p_j]) + \ldots + R + Err(c_k, [p_{k-1}, p_k, p_{k+1}, \ldots, p_n])$$

Note that if $R = 0$ (no penalty on new clusters) then the optimum clustering uses $\frac{n}{2}$ runs:

$$\langle p_1, p_2 \rangle, \langle p_3, \ldots, p_5 \rangle, \ldots, \langle p_{n-1}, p_n \rangle.$$  If $R$ is huge, then the optimum uses only one cluster, containing all the points.

In the example on top, $k = 3$, $i_1 = 5$, $i_2 = 8$

Idea: The rightmost point of each cluster is the ‘cluster-head’, and this point has to pay the penalty $R$.

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

Algorithm

Preprocessing: for every pair of $i$ and $j$ (where $j < i$) compute the line $c_j$ that best fit the points $\{p_j, p_{j+1}, \ldots, p_i\}$. Calculate $Err(i, j) = Err(c_j, \{p_j, p_{j+1}, \ldots, p_i\})$

The DP uses only one dimensional array $c[0 \ldots n]$, and another array $\Pi[1 \ldots n]$

- Let $c[i] =$ 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. This clustering contains at least one but possibly more clusters.
- 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 $c[i]$. All points $\{p_j, p_{j+1}\}$ have been ‘processed’. We are surprised by $p_i$, and need to update the clustering (pretending that $p_i$ is the last point).
- *Being the (globally) rightmost point, $p_i$ is the rightmost point of its own cluster. It needs to pay $R$, and also needs to decide what is the leftmost point of its cluster. For example, if leftmost is $p_1$, then we will end up with only one cluster.*

**Algorithm:**

- **Init:** $c[0] = 0$; for every $i > 1$ set $c[i] = \infty$ and $\Pi[i] = NULL$.
- For $i = 2$ to $n$
  - For $j = 1$ to $i$
    - If $j = i$
      - If $j = i$
        - $M = c[j-1] + R + \text{Err}[j, i]$ // total cost of clustering if the cluster of $p_j$ starts at $p_j$
      - If $M < c[i]$
        - $c[i] = M$
        - $\Pi[i] = j$ //used to reconstruct the actual clustering
  - Return $c[n]$ to $n$
Knapsack problem

- The problem is NP-hard. But easier to solve if we assume all length $w_i$ are integers.
- Let $S_i$ be the set of segments $S_i = \{s_1, \ldots, s_i\}$. $S_0 = \emptyset$.
- $c(i, j)$ is the opt revenue solution to the knapsack problem when segment are only picked from $S_i$, and the total length is at most $\sum w_i$ is at most $j$ (here $j \leq W$ is an int).
- In other words, we only could use segments from $S_i$ (instead of S) and we need to fit them into a narrow knapsack of width $j$ (instead of $W$)
- Fix knapsack width $j$. Assume $c(i, j)$ is already calculated for all $1 \leq i \leq n$.
- The outer loop of the algorithm checks wider and wider knapsacks.
- The inner loop checks $i=0,1,2,\ldots n$. That is, by increasing $i$, we could use more segments.
- Each segment is used at most once

![Diagram of knapsack problem](57.png)

- The problem is NP-hard. But easier to solve if we assume all length $w_i$ are integers.
- Let $S_i$ be the set of segments $S_i = \{s_1, \ldots, s_i\}$. $S_0 = \emptyset$.
- Notation: $[0..j]$ is a knapsack of length $j$. The original knapsack is $[0..W]$.
- $c(i, j)$ is the opt revenue for the knapsack problem, with two restriction:
  1. Segment are picked on from $S_i$.
  2. The total length is at most $\sum w_i$ is at most $j$ (here $j \leq W$ is an int). That is, fit into the knapsack $[0..j]$.
- Fix knapsack width $j$. Assume $c(i, j)$ is already calculated for all $1 \leq j' \leq j$ (narrower knapsacks).
- For $i = 0$ clearly $c(0, j) = 0$ (since $S_0 = \emptyset$, we cannot use any segment)
- Increasing $i$ from $i-1$ to $i$ implies that for the knapsack $[0..j]$ we could (allowed to) use $s_j$ - but should we use $s_j$ (for $[1..j]\}$?
  - If we do use $s_j$ at this point $(j,0)$, then we place $s_j$ so its right endpoint at $(j,0)$.
  - So its left endpoint is at $(j-w_j, 0)$, and the max revenue is $r_j + c(i-1, j-w_j)$
- Observation: $c(i, j) = \max \{c(i-1, j), r_i + c(i-1, j-w_j)\}$

![Diagram of knapsack problem](58.png)

- Algorithm
  1. Init $c(0, j) = 0$, $\forall j$
  2. For $j = 1, \ldots, W$
    2.1.1 $c(i, j) = \max \{c(i-1, j-1), c(i, j-1), r_i + c(i-1, j-w_j)\}$
    2.1.2 $%Of course, keep $\Pi[1..n,1..W]$ to construct the opt solution.$

![Diagram of knapsack problem](59.png)