Advanced Algorithmics (6EAP)
Dynamic Programming

Jaak Vilo
2013 Spring
Example

• Wind has blown away the +, *, (, ) signs
• What’s the maximal value?
• Minimal?

2 1 7 1 4 3
• 2 1 7 1 4 3

• $(2+1)*7*(1+4)*3 = 21*15 = 315$

• $2*1 + 7 + 1*4 + 3 = 16$
• Q: How to maximize the value of any expression?

2 4 5 1 9 8 12 1 9 8 7 2 4 4 1 1 2 3 = ?
• Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
  – Dünaamiline planeerimine.

• Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

• In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.
Dynamic programming

- Avoid calculating repeating subproblems

- \( \text{fib}(1)=\text{fib}(0)=1; \)
- \( \text{fib}(n) = \text{fib}(n-1)+\text{fib}(n-2) \)

- Although natural to encode (and a useful task for novice programmers to learn about recursion) recursively, this is inefficient.
Structure within the problem

• The fact that it is not a tree indicates overlapping subproblems.
• A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.
Topp-down (recursive, memoized)

- **Top-down approach**: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily memoize or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.
Bottom-up

• *Bottom-up approach*: This is the more interesting case. Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. For example, if we already know the values of $F_{41}$ and $F_{40}$, we can directly calculate the value of $F_{42}$. 
• Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

• We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.
The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.
Edit (Levenshtein) distance

- **Definition** The edit distance $D(A,B)$ between strings $A$ and $B$ is the minimal number of edit operations to change $A$ into $B$. Allowed edit operations are deletion of a single letter, insertion of a letter, or replacing one letter with another.

- Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_m$.
  - $E1$: **Deletion** $a_i \rightarrow \epsilon$
  - $E2$: **Insertion** $\epsilon \rightarrow b_i$
  - $E3$: **Substitution** $a_i \rightarrow b_j$ (if $a_i \neq b_j$)

- Other possible variants:
  - $E4$: **Transposition** $a_ia_{i+1} \rightarrow b_jb_{j+1}$ and $a_i=b_{j+1} \land a_{i+1}=b_j$
    (e.g. lecture $\rightarrow$ letcure)
Recursion?

\[ d(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ d(i-1,j-1) + d(i-1,j) + d(i,j-1) & \text{otherwise} \end{cases} \]
Dynamic Programming

\[
d(i,j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
d(i-1,j-1) + c(i,j) & \text{if } i > 0 \text{ and } j > 0 
\end{cases}
\]

where \(c(i,j)\) represents the cost associated with moving from \(i\) to \(j\).
Algorithm Edit distance D(A,B) using Dynamic Programming (DP)

Input: A=a_1a_2...a_n, B=b_1b_2...b_m
Output: Value d_{mn} in matrix (d_{ij}), 0 \leq i \leq m, 0 \leq j \leq n.

for i=0 to m do d_{i0}=i ;
for j=0 to n do d_{0j}=j ;
for j=1 to n do
  for i=1 to m do
    d_{ij} = \min( d_{i-1,j-1} + (if a_i==b_j then 0 else 1),
                     d_{i-1,j} + 1, d_{i,j-1} + 1 )
return d_{mn}
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 6 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 7 | A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 8 | A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 9 | A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 10| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 11| B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 12| B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 13| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 14| A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 15| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 16| A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 17| B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 18| B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 19| A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 20| A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 21| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 22| A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 23| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 24| C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 25| B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Edit distance**

The edit distance is the shortest path in the matrix.
Matrix multiplication

- for $i=1..n$
  - for $j = 1..k$
    - $c_{ij} = \sum_{x=1..m} a_{ix} b_{xj}$

$O(nmk)$
MATRIX-MULTIPLY($A,B$)

1. if columns $[A] \neq$ rows $[B]$

2. then error "incompatible dimensions"

3. else for $i = 1$ to rows $[A]$

4. do for $j = 1$ to columns $[B]$

5. do $C[i, j] = 0$

6. for $k = 1$ to columns $[A]$

7. do $C[i, j] = C[i, j] + A[i, k] \times B[k, j]$

8. return $C$
Chain matrix multiplication

**Figure 6.6** $A \times B \times C \times D = (A \times (B \times C)) \times D.$

(a)
\[
\begin{array}{c}
\times \\
A \\
50 \times 20
\end{array}
\quad
\begin{array}{c}
\times \\
B \\
20 \times 1
\end{array}
\quad
\begin{array}{c}
\times \\
C \\
1 \times 10
\end{array}
\quad
\begin{array}{c}
D \\
10 \times 100
\end{array}
\]

(b)
\[
\begin{array}{c}
\times \\
A \\
50 \times 20
\end{array}
\quad
\begin{array}{c}
\times \\
B \times C \\
20 \times 10
\end{array}
\quad
\begin{array}{c}
\times \\
D \\
10 \times 100
\end{array}
\]

(c)
\[
\begin{array}{c}
\times \\
A \times (B \times C) \\
50 \times 10
\end{array}
\quad
\begin{array}{c}
D \\
10 \times 100
\end{array}
\]

(d)
\[
\begin{array}{c}
\times \\
(A \times (B \times C)) \times D \\
50 \times 100
\end{array}
\]

6.5 Chain matrix multiplication
Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $mnp$ multiplications, to a good enough approximation. Using this formula, let’s compare several different ways of evaluating $A \times B \times C \times D$:

<table>
<thead>
<tr>
<th>Parenthesization</th>
<th>Cost computation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times ((B \times C) \times D)$</td>
<td>$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$</td>
<td>120,200</td>
</tr>
<tr>
<td>$(A \times (B \times C)) \times D$</td>
<td>$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$</td>
<td>60,200</td>
</tr>
<tr>
<td>$(A \times B) \times (C \times D)$</td>
<td>$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$</td>
<td>7,000</td>
</tr>
</tbody>
</table>

As you can see, the order of multiplications makes a big difference in the final running time! Moreover, the natural greedy approach, to always perform the cheapest matrix multiplication available, leads to the second parenthesization shown here and is therefore a failure.
The diagram illustrates the multiplication of matrices $A$, $B$, $C$, and $D$ with their dimensions indicated:

- $A$ is a $50 \times 20$ matrix,
- $B$ is a $20 \times 1$ matrix,
- $C$ is a $1 \times 10$ matrix,
- $D$ is a $10 \times 100$ matrix.

Below is a table showing different parenthesizations and their corresponding cost computations and costs:

<table>
<thead>
<tr>
<th>Parenthesization</th>
<th>Cost computation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times ((B \times C) \times D)$</td>
<td>$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$</td>
<td>120,200</td>
</tr>
<tr>
<td>$(A \times (B \times C)) \times D$</td>
<td>$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$</td>
<td>60,200</td>
</tr>
<tr>
<td>$(A \times B) \times (C \times D)$</td>
<td>$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$</td>
<td>7,000</td>
</tr>
</tbody>
</table>
• The *matrix-chain multiplication problem* can be stated as follows: given a chain \(<A_1, A_2, \ldots, A_n>\) of \(n\) matrices
• matrix \(A_i\) has dimension \(p_{i-1} \times p_i\)
• fully parenthesize the product \(A_1 A_2 \ldots A_n\) in a way that minimizes the number of scalar multiplications.
$A_1 A_2 A_3 A_4$

- $(A_1 (A_2 (A_3 A_4)))$
- $(A_1 ((A_2 A_3) A_4))$
- $((A_1 A_2)(A_3 A_4))$
- $((A_1 (A_2 A_3)) A_4)$
- $(((A_1 A_2) A_3) A_4)$
• Denote the number of alternative parenthesizations of a sequence of $n$ matrices by $P(n)$.

• Since we can split a sequence of $n$ matrices between the $k$th and $(k + 1)$st matrices for any $k = 1, 2, \ldots, n - 1$ and then parenthesize the two resulting subsequences independently, we obtain the recurrence

$$P(n) = \begin{cases} 1, & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{if } n \geq 2. \end{cases}$$
Problem 13-4 asked you to show that the solution to this recurrence is the sequence of Catalan numbers:

\[ C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(4^n/n^{3/2}) \]

- \( P(n) = C(n - 1) \), where
- The number of solutions is thus exponential in \( n \), and the brute-force method of exhaustive search is therefore a poor strategy for determining the optimal parenthesization of a matrix chain.
Let’s crack the problem

\[ A_{i..j} = A_i \cdot A_{i+1} \cdot \cdots \cdot A_j \]

- Optimal parenthesesization of \( A_1 \cdot A_2 \cdot \cdots \cdot A_n \) splits at some \( k, k+1 \).
- Optimal = \( A_{1..k} \cdot A_{k+1..n} \)

\[ T(A_{1..n}) = T(A_{1..k}) + T(A_{k+1..n}) + T(A_{1..k} \cdot A_{k+1..n}) \]

- \( T(A_{1..k}) \) must be optimal for \( A_1 \cdot A_2 \cdot \cdots \cdot A_k \)
Recursion

• $m[i, j]$ - minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$;
• $m[i,i] = 0$
• $cost(A_{i..k} \cdot A_{k+1..j}) = p_{i-1} p_k p_j$
• $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

• This recursive equation assumes that we know the value of $k$, which we don't. There are only $j - i$ possible values for $k$, however, namely $k = i, i + 1, \ldots, j - 1$. 
Since the optimal parenthesization must use one of these values for $k$, we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \ldots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{r \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$ (16.2)$$

To help us keep track of how to construct an optimal solution, let us define $s[i, j]$ to be a value of $k$ at which we can split the product $A_i A_{i+1} \ldots A_j$ to obtain an optimal parenthesization. That is, $s[i, j]$ equals a value $k$ such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$. 
Recursion

• Checks all possibilities...

• But – there is only a few subproblems – choose \( i, j \) s.t. \( 1 \leq i \leq j \leq n \) – \( O(n^2) \)

• A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of the applicability of dynamic programming.
MATRIX-CHAIN-ORDER\( (p) \)

1 \( n \leftarrow length[p] \ - \ 1 \)

2 \textbf{for} \( i \leftarrow 1 \ \textbf{to} \ n \)

3 \hspace{1em} \textbf{do} \( m[i, i] \leftarrow 0 \)

4 \textbf{for} \( l \leftarrow 2 \ \textbf{to} \ n \) \hspace{1em} // foreach length from 2 to n

5 \hspace{1em} \textbf{do} \( i \leftarrow 1 \ \textbf{to} \ n \ - \ l \ + \ 1 \) \hspace{1em} // foreach start index i

6 \hspace{2em} \textbf{do} \( j \leftarrow i \ + \ l \ - \ 1 \)

7 \hspace{3em} \hspace{1em} m[i, j] \leftarrow \infty \)

8 \hspace{3em} \hspace{1em} \hspace{1em} \textbf{for} \( k \leftarrow i \ \textbf{to} \ j \ - \ 1 \) \hspace{1em} // check all mid-points for optimality

9 \hspace{4em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{do} \( q \leftarrow m[i, k] \ + \ m[k \ + \ 1, j] \ + \ p[i \ - \ 1] \cdot p[k] \cdot p[j] \)

10 \hspace{4em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{if} \( q < m[i, j] \)

11 \hspace{4em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{then} \( m[i, j] \leftarrow q \) \hspace{1em} // new best value q

12 \hspace{4em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{then} \( s[i, j] \leftarrow k \) \hspace{1em} // achieved at mid point k

13 \textbf{return} \( m \) and \( s \)
Example

\[(A_1(A_2A_3))((A_4A_5)A_6)\]

\[
\begin{align*}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
15,750 & 7,875 & 9,375 & 11,875 & 15,125 & 6 \\
2,625 & 4,375 & 7,125 & 10,500 & 15,125 & 1 \\
750 & 2,500 & 5,375 & 11,875 & 15,125 & 2 \\
1,000 & 3,500 & 5,375 & 11,875 & 15,125 & 3 \\
5,000 & 3,500 & 5,375 & 11,875 & 15,125 & 4 \\
0 & 0 & 0 & 0 & 0 & 5
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
m[2, 5] &= \text{min} \left\{ m[2, 2] + m[3, 5] + \pi_1 p_2 p_3, m[2, 3] + m[4, 5] + \pi_1 p_3 p_5, m[2, 4] + m[5, 5] + \pi_1 p_4 p_5 \right\} \\
&= \min \left\{ 15,750 + 7,875 + 9,375 + 11,875 + 15,125 + 6, 7,875 + 4,375 + 7,125 + 10,500 + 15,125 + 1, 5,000 + 3,500 + 5,375 + 11,875 + 15,125 + 4 \right\} \\
&= \min \{ 43,750, 43,750, 43,750 \} \\
&= 43,750.
\end{align*}
\]

\[
\begin{align*}
m[2, 2] + m[3, 5] + \pi_1 p_2 p_3 &= 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\
m[2, 3] + m[4, 5] + \pi_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\
m[2, 4] + m[5, 5] + \pi_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375.
\end{align*}
\]

Matrix dimensions:
- \(A_1\) 30 X 35
- \(A_2\) 35 X 15
- \(A_3\) 15 X 5
- \(A_4\) 5 X 10
- \(A_5\) 10 X 20
- \(A_6\) 20 X 25
• A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index ($l$, $i$, and $k$) takes on at most $n$ values.
• Time $\Omega(n^3) \Rightarrow \Theta(n^3)$
• Space $\Theta(n^2)$
• Step 4 of the dynamic-programming paradigm is to construct an optimal solution from computed information.

• Use the table $s[1 \ldots n, 1 \ldots n]$ to determine the best way to multiply the matrices.
Multiply using S table

MATRIX-CHAIN-MULTIPLY(A, s, i, j)

1 if j > i
2 then X = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
3 Y = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j]+1, j)
4 return MATRIX-MULTIPLY(X, Y)
5 else return A_i

(((A_1(A_2A_3))((A_4A_5)A_6)))
Elements of dynamic programming

• **Optimal substructure** within an optimal solution

• **Overlapping subproblems**

• **Memoization**
• A **memoized recursive algorithm** maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned. (tabulated)

• This approach presupposes that the set of all possible subproblem parameters is known and that the relation between table positions and subproblems is established. Another approach is to memoize by using hashing with the subproblem parameters as keys.
Overlapping subproblems
Longest Common Subsequence (LCS)

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Optimal triangulation

The problem is to find a triangulation that minimizes the sum of the weights of the triangles in the triangulation.

Two ways of triangulating a convex polygon. Every triangulation of this 7-sided polygon has 7 - 3 = 4 chords and divides the polygon into 7 - 2 = 5 triangles.
Parse trees. (a) The parse tree for the parenthesized product \(((A_1(A_2A_3))(A_4(A_5A_6)))\) and for the triangulation of the 7-sided polygon (b) The triangulation of the polygon with the parse tree overlaid. Each matrix \(A_i\) corresponds to the side \(v_{i-1}v_i\) for \(i = 1, 2, \ldots, 6\).
Optimal triangulation

\[ t[i, j] = \begin{cases} 
0 & \text{if } i = j, \\
\min_{i \leq k \leq j-1} \{t[i, k] + t[k + 1, j] + w(\Delta v_{i-1} v_k v_j)\} & \text{if } i < j.
\end{cases} \]  

(16.7)
Edit distance (Levenshtein distance)

- Smallest nr of edit operations to convert one string into the other

  INDUSTRY

  INTEREST

  INDUSTRY

  INTEREST
• **Definition** The edit distance $D(A,B)$ between strings $A$ and $B$ is the minimal number of edit operations to change $A$ into $B$. Allowed edit operations are deletion of a single letter, insertion of a letter, or replacing one letter with another.

• Let $A = a_1 \ a_2 \ ... \ \ a_m$ and $B = b_1 \ b_2 \ ... \ \ b_m$.
  
  – E1: **Deletion** $a_i \rightarrow \varepsilon$
  – E2: **Insertion** $\varepsilon \rightarrow b_i$
  – E3: **Substitution** $a_i \rightarrow b_j$ (if $a_i \neq b_j$)

• Other possible variants:
  
  – E4: **Transposition** $a_i a_{i+1} \rightarrow b_j b_{j+1}$ and $a_i=b_{j+1}$ ja $a_{i+1}=b_j$
    (e.g. lecture $\rightarrow$ letcure)
How can we calculate this?

\[
D(\alpha a, \beta b) =
\begin{cases}
1. \ D(\alpha, \beta) & \text{if } a=b \\
2. \ D(\alpha, \beta)+1 & \text{if } a\neq b
\end{cases}
\]
How can we calculate this efficiently?

\[ D(S, T) = \min \begin{cases} 
1. & D(S[1..n-1], T[1..m-1]) + (S[n]=T[m])? 0 : 1 \\
2. & D(S[1..n], T[1..m-1]) +1 \\
3. & D(S[1..n-1], T[1..m]) +1 
\end{cases} \]

Define: \( d(i,j) = D( S[1..i], T[1..j] ) \)

\[ d(i,j) = \min \begin{cases} 
1. & d(i-1,j-1) + (S[n]=T[m])? 0 : 1 \\
2. & d(i, j-1) +1 \\
3. & d(i-1, j) +1 
\end{cases} \]
Recursion?

\[ d(i, j) \]

\[ d(i-1, j-1) \quad d(i, j-1) \quad d(i-1, j) \]

\[ x \quad y \]
Recursion?
Algorithm Edit distance $D(A,B)$ using Dynamic Programming (DP)

Input: $A=a_1a_2...a_n$, $B=b_1b_2...b_m$

Output: Value $d_{mn}$ in matrix $(d_{ij})$, $0 \leq i \leq m$, $0 \leq j \leq n$.

for $i=0$ to $m$ do $d_{i0}=i$;

for $j=0$ to $n$ do $d_{0j}=j$;

for $j=1$ to $n$ do
  for $i=1$ to $m$ do
    $d_{ij} = \min( d_{i-1,j-1} + (\text{if } a_i == b_j \text{ then } 0 \text{ else } 1),
    d_{i-1,j} + 1, d_{i,j-1} + 1 )$

return $d_{mn}$
Dynamic Programming
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>S=</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>T=</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Edit distance is a metric

- It can be shown, that $D(A,B)$ is a metric
  - $D(A,B) \geq 0$, $D(A,B)=0$ iff $A=B$
  - $D(A,B) = D(B,A)$
  - $D(A,C) \leq D(A,B) + D(B,C)$
Path of edit operations

• Optimal solution can be calculated afterwards
  – Quite typical in dynamic programming

\[
\begin{align*}
d[i-1, j-1] & \quad d[i-1, j] \\
\downarrow & \quad \downarrow \\
d[i, j-1] & \quad d[i, j]
\end{align*}
\]

• Memorize sets \( \text{pred}[i,j] \) depending from where the \( d_{ij} \) was reached.
Three possible minimizing paths

• Add into pred[i,j]
  – (i-1,j-1) if $d_{ij} = d_{i-1,j-1} + (\text{if } a_i == b_j \text{ then } 0 \text{ else } 1)$
  – (i-1,j) if $d_{ij} = d_{i-1,j} + 1$
  – (i,j-1) if $d_{ij} = d_{i,j-1} + 1$
The path (in reverse order) $\varepsilon \rightarrow c_6, \ b_5 \rightarrow b_5, \ c_4 \rightarrow c_4, \ a_3 \rightarrow a_3, \ a_2 \rightarrow b_2, \ b_1 \rightarrow a_1$. 
Multiple paths possible

• All paths are correct

• There can be many (how many?) paths
Space can be reduced
Calculation of D(A,B) in space Θ(m)

Input: A=a₁a₂...aₘ, B=b₁b₂...bₙ (choose m<=n)

Output: dₘₙ=D(A,B)

for i=0 to m do C[i]=i

for j=1 to n do
  C=C[0]; C[0]=j;
  for i=1 to m do
    d = min( C + (if aᵢ==bⱼ then 0 else 1), C[i-1] + 1, C[i] + 1 )
    C = C[i] // memorize new “diagonal” value
    C[i] = d

write C[m]

Time complexity is Θ(mn) since C[0..m] is filled n times
Shortest path in the graph

Shortest path in the graph


All nodes at distance 1 from source
The image shows a Microsoft Excel spreadsheet with a table that appears to represent a dynamic programming algorithm, likely for calculating the edit distance between two sequences, S and T.

The table includes a color-coded grid, with values indicating the cost of transforming S into T. The cost increases as the transformation becomes more complex, with higher costs represented in darker shades.

The algorithm likely uses a matrix where each cell [i][j] represents the minimum cost of transforming the first i characters of S into the first j characters of T. The cost is calculated based on the cost of insertions, deletions, and substitutions.

The Excel sheet contains columns labeled A to U and rows labeled 1 to 25, with the sequences S and T aligned in the first row of the table. The color gradient helps visualize the cost matrix, with darker cells indicating higher costs.
Observations?

• Shortest path is close to the diagonal
  – If a short distance path exists

• Values along any diagonal can only increase (by at most 1)
Diagonal

Property of any diagonal: The values of matrix \(d_{ij}\) can on any specific diagonal either increase by 1 or stay the same.

Diagonal nr. 2, \(d_{02}, d_{13}, d_{24}, d_{35}, d_{46}\)

Diagonal \(k\), \(-m \leq k \leq n\),

s.t. diagonal \(k\) contains only \(d_{ij}\) where \(j-i = k\).
Example

- Example (compare to prev. ex)

Diagonals x edit distances

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>5</td>
</tr>
<tr>
<td>-4</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-∞</td>
<td>-∞</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-∞</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>0</td>
</tr>
</tbody>
</table>

In orig. matrix ($d_{ij}$) there are 42 values, and in diagonal matrix ($f_{kp}$) 22 values (excl. -∞).

- **Note:** The values in ($f_{kp}$) are not comfortably organized. With appropriate index transformations it is possible to give the ($f_{kp}$) more regular shape.
Extensions to basic edit distance

• New operations

• Variable costs

• ...
Transposition \((ab \rightarrow ba)\)

- **E4: Transposition**
  \[ a_i a_{i+1} \rightarrow b_j b_{j+1} \text{, s.t. } a_i = b_{j+1} \text{ and } a_{i+1} = b_j \]
  
  - (e.g.: lecture \(\rightarrow\) letcure)

\[ d(i,j) = \min \left\{ \begin{array}{l} 1. \ d(i-1,j-1) + (S[n]=T[m])? 0 : 1 \\ 2. \ d(i, j-1) +1 \\ 3. \ d(i-1, j) +1 \\ 4. \ d(i-2,j-2) + ( \text{ if } S[i-1,i] = T[j,j-1] \text{ then } 1 \text{ else } \infty) \end{array} \right\} \]
Generalized edit distance

• Use more operations $E_1...E_n$, and to provide different costs to each.

• **Definition.** Let $x, y \in \Sigma^*$. Then every $x \rightarrow y$ is an edit operation. Edit operation replaces $x$ by $y$.
  - If $A=uxv$ then after the operation, $A=uyv$

• We note by $w(x \rightarrow y)$ the cost or weight of the operation.

• Cost may depend on $x$ and/or $y$. But we assume $w(x \rightarrow y) \geq 0$. 
Generalized edit distance

- If operations can only be applied in parallel, i.e. the part already changed cannot be modified again, then we can use the dynamic programming.
- Otherwise it is an algorithmically unsolvable problem, since question - can A be transformed into B using operations of G, is unsolvable.
- The diagonal method in general may not be applicable.
- But, since each diversion from diagonal, the cost slightly increases, one can stay within the narrow region around the diagonal.
Applications of generalized edit distance

- Historic documents, names
- Human language and dialects
- Transliteration rules from one alphabet to another
  e.g. Tõugu => Tyugu (via Russian)
- ...

ApplicaMons
t
  of
generalized	edit
distance
Examples

- (0.150000) search
- (0.500000) shorts
- (0.650000) serge
- (0.650000) surge
- (1.100000) cert
- (1.100000) certain

Päringuks kulus: ~2 sekund(it);
**Otsisõna:** jazhelo

Palun vali sönadetaid, milles otsing sooritatakse:
- Kohanimed
- Venekeelsed sõnad
- Rossinuse sõnad
- Stahl sõnad
- Müller sõnad
- ET-Eesti-Englise
- EN-Englise-Eesti

Milliseid vasteid otsida?
- Täpsed vasteid
- Algusosa vasteid
- Keskosa vasteid
- Lõpuosa vasteid

Täendavad otsingutingimused
- Tõstutundlik otsing
  - Maksimaalne teiseduskaugus: 1.5
  - Vastete maksimaalne arv: 10

**Täpsed vasted**

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.106000</td>
<td>1.106000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.108000</td>
<td>1.108000</td>
</tr>
</tbody>
</table>

**Keskosa vasted**

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.106000</td>
<td>1.106000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.108000</td>
<td>1.108000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.108000</td>
<td>1.108000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Venekeelsed sõnad</th>
<th>Venekeelsed sõnad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.108000</td>
<td>1.108000</td>
</tr>
</tbody>
</table>
Otsisõna: ratham

Palun vali sönadeesk(t), milles otsing sooritatakse:

- Kohanimed [Valik Eesti kohanimed]
- Venekeelseid sönd [Valik venekeelseid sönu]
- Rossiniuse sönad [Sõnad Rossiniuse (~1792-1868) teatristest]
- Stahl sönad [Sõnad Stahli (~1690-1767) teatristest]
- Müller sönad [Sõnad Mülleri (~1805-1868) teatristest]
- ET-Eesti-Inglise [Eestikeelseid vasted Eesti-Inglise sõnastikust]
- EN-Inglise-Eesti [Inglisekeelseid vasted Inglise-Eesti sõnastikust]

Milliseid vasteid otsida?

- Täpsed vasteid
- Algusosa vasteid
- Keskosa vasteid
- Lõpusosa vasteid

Täiedavat otsingu tingimused

- Tööstutudlik otsing

Maksimaalne tundlik otsing: 1.5

Vastete maksimaalne arv (ühes tulbas):

<table>
<thead>
<tr>
<th>Täpsed vasteid</th>
<th>Algusosa vasteid</th>
<th>Keskosa vasteid</th>
<th>Lõpusosa vasteid</th>
</tr>
</thead>
<tbody>
<tr>
<td>raghwas</td>
<td>1.084947</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rachwas</td>
<td>1.087829</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ragwas</td>
<td>1.260298</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rachwas</td>
<td>1.087829</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rachwas</td>
<td>1.087829</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rachwas</td>
<td>1.263404</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parrwas
näituseks – näiteks
tuseks -> teks
Ahwrika - Aafrika
weikese - väikese
materjaali - materjali

“kavalam” otsimine
tuseks -> teks
a -> aa , hw -> f
w -> v , e -> ä
aa -> a

Dush, dušš, dushsh ?
Gorbatšov, Gorbatshov, Горбачов,
Gorbachev
režiim, rezhiim, riim
How?

• Apply Aho-Corasick to match for all possible edit operations

• Use minimum over all possible such operations and costs

• Implementation: Reina Käärik, Siim Orasmaa
Possible problems/tasks

• Manually create sensible lists of operations
  – For English, Russian, etc…
  – Old language,

• Improve the speed of the algorithm (testing)

• Train for automatic extraction of edit operations and respective costs from examples of matching words…
Advanced Dynamic Programming

• Robert Giegerich:
  – http://www.techfak.uni-bielefeld.de/ags/pi/lehre/ADP/

• Algebraic dynamic programming
  – Functional style
  – Haskell compiles into C