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 1 1 2 3 = ?

Dynamic programming

• Avoid calculating repeating subproblems
• \(\text{fib}(0) = 1;\)
• \(\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)\)
• 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 distance (Levenshtein distance)

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

<table>
<thead>
<tr>
<th>INDUSTRY</th>
<th>INDUSTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST</td>
<td>INTEREST</td>
</tr>
</tbody>
</table>

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 $A = a_1 a_2 ... a_n$ and $B = b_1 b_2 ... b_m$:
  - E1: Deletion $a_i \rightarrow \epsilon$
  - E2: Insertion $\epsilon \rightarrow b_j$
  - 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 $a_{i+1} = b_j$ (e.g. lecture $\rightarrow$ letcure)

How can we calculate this?

Recursion?

$D(a, b) = \min\begin{cases} 
1. & D(a, b) \\
2. & D(a[1..n-1], b[1..m-1]) + 1 \\
3. & D(a[1..n-1], b[1..m]) + 1 \\
4. & D(a[1..n], b[1..m-1]) + 1 \\
\end{cases}$

How can we calculate this efficiently?

Recursion?

$D(i,j) = \min\begin{cases} 
1. & D(i-1,j-1) + (a_i = b_j) \ ? \ 0 \ : \ 1 \\
2. & D(i-1,j) + 1 \\
3. & D(i,j-1) + 1 \\
\end{cases}$

Define:

- $d(i, j) = D(S[1..i], T[1..j])$
- $d_0(j) = \begin{cases} 
0 & j = 0 \\
1 & j > 0 \\
\end{cases}$

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

$D(i,j) = \min_{i,j}
\begin{cases} 
1. & d(i,j) + (a_i = b_j) \ ? \ 0 \ : \ 1 \\
2. & d(i,j-1) + 1 \\
3. & d(i-1,j) + 1 \\
\end{cases}$
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≤i≤m, 0≤j≤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

---

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 $A$ ≠ 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 $[A]$
8 return $C$
The matrix-chain multiplication problem can be stated as follows: given a chain <A_1, A_2, ..., A_n> of n matrices

- matrix A_i has dimension p_{i-1} × p_i
- fully parenthesize the product A_1 A_2 ... A_n in a way that minimizes the number of scalar multiplications.

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 kth and (k + 1)st matrices for any k = 1, 2, ..., n

-1 and then parenthesize the two resulting subsequences independently, we obtain the recurrence

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

\[ C(n) = \frac{1}{n+1} \binom{2n}{n}, \]

\[ P(n) = C(n-1), \]

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**

\[ A_{i,j} = A_i \bullet A_{i+1} \bullet \cdots \bullet A_j \]

- Optimal parenthesization of \( A_1 \bullet A_2 \bullet \cdots \bullet A_n \) splits at some \( k, k+1 \).
- Optimal = \( A_{1,k} \bullet A_{k+1,n} \)
- \( T(A_{1,k}) = T(A_{1,k-1}) + T(A_{k+1,n}) + T(A_{1,k} \bullet A_{k+1,n}) \)
- \( T(A_{1,k}) \) must be optimal for \( A_1 \bullet A_2 \bullet \cdots \bullet A_k \)

**Recursion**

- \( m[i,j] \) - minimum number of scalar multiplications needed to compute the matrix \( A_{i,j} \)
- \( m[i,j] = 0 \)
- \( \text{cost}(A_{1,k} \bullet A_{k+1,j}) = P_{i-1} p_k p_j \)
- \( m[i,j] = m[i,k] + m[k+1,j] + P_{i-1} p_k p_j \)

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 \( k = i, i+1, \ldots, j-1 \).

**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.

---

// foreach length from 2 to n
// check all mid-points for optimality
// foreach start index i
// new best value q
// achieved at mid point k

---

- 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 \( A_1 \bullet A_2 \bullet \cdots \bullet A_n \) becomes

\[ m[i,j] = \begin{cases} 
0 & \text{if } i = j \\hat{\text{or}}\text{ the product is scalar} \\
\min (m[i,k] + m[k+1,j] + P_{i-1} p_k p_j) & \text{otherwise} 
\end{cases} \]

(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 \( A_1 \bullet A_2 \bullet \cdots \bullet A_n \) to obtain an optimal parenthesization. That is, \( s[i,j] \) equals a value \( k \) such that \( m[i,j] = m[i,k] + m[k+1,j] + P_{i-1} p_k p_j \).
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 $(i, j, k)$ takes on at most $n$ values.

- Time $O(n^3) \approx \Theta(n^3)$
- Space $\Theta(n^2)$

Elements of dynamic programming

- Optimal substructure within an optimal solution
- Overlapping subproblems
- Memoization

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.

Multiplying using the $S$ table

MATRIX-CHAIN-MULTIPLY($A$, $s$, $i$, $j$)
1 if $j \neq i$
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 $A_i$

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

A memoized recursive algorithm maintains an entry in a table for 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></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>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>F</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>G</td>
<td>H</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>I</td>
<td>J</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>K</td>
<td>L</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

OpYmal triangulaYon

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.

Optimal triangulation

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

Dynamic programming

- Avoid re-calculating same subproblems by
  - Characterising optimal solution
  - Clever ordering of calculations
Dynamic Programming

\[ d(i,j) \]

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

0 1 2 3 4 5 6 7 8 9 10

\[ A \]

\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 1 & 2 & 3 & 4 & 5 \\
a & 2 & 1 & 2 & 3 & 4 \\
a & 3 & 2 & 2 & 2 & 3 & 4 \\
c & 4 & 3 & 3 & 2 & 3 & 3 \\
b & 5 & 4 & 3 & 4 & 3 & 2 & 3 & 3
\end{array} \]

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

- Memorize sets \( \text{pred}[i, j] \) depending from where \( d_{ij} \) was reached.

Three possible minimizing paths

- Add into \( \text{pred}[i, j] \)
  - \((i-1, j-1)\) if \( d_{ij} = d_{i-1,j-1} + 1 \) (if \( a_i = b_j \) then 0 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) $\epsilon \rightarrow c_6, b_5 \rightarrow b_5, c_4 \rightarrow c_4, a_3 \rightarrow a_3, a_2 \rightarrow b_2, b_1 \rightarrow a_1$.

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 $\Theta(m)$**

**Input:** $A=a_1a_2...a_m, B=b_1b_2...b_n$ (choose $m \leq n$)

**Output:** $d_{mn}=D(A,B)$

- $C[0]=i$
- for $i=1$ to $n$ do
  - $C = C[0]; C[0]=j$
  - for $i=1$ to $m$ do
    - $d = \min(C + (\text{if } a_i = b_j \text{ then } 0 \text{ else } 1), C[i-1] + 1, C[i] + 1)$
    - $C = C[i]$ // memorize new "diagonal" value
    - $C[i] = d$
  - write $C[m]$

Time complexity is $\Theta(mn)$ since $C[0..m]$ is filled $n$ times

Shortest path in the graph

All nodes at distance 1 from source
15.10.15

**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**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0 123456</td>
</tr>
<tr>
<td>b</td>
<td>1 123456</td>
</tr>
<tr>
<td>a</td>
<td>2 222234</td>
</tr>
<tr>
<td>a</td>
<td>3 333323</td>
</tr>
<tr>
<td>b</td>
<td>4 433323</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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

- Diagonal \(k\), \(-m \leq k \leq n\), s.t. diagonal \(k\) contains only \(d_{ij}\) where \(j-i = k\).

**Diagonal lemma**

**Lemma:** For each \(d_{ij}\), 1st line, 1st column holds: \(d_{ij} = d_{i,j+1}\) or \(d_{ij} = d_{i+1,j} + 1\). (notice that \(d_{ij}\) and \(d_{i,j+1}\) are on the same diagonal)

**Proof:** Since \(d_{ij}\) is an integer, show:

1. \(d_{ij} \leq d_{i,j+1} + 1\)
2. \(d_{ij} \leq d_{i+1,j} + 1\)

From the definition of edit distance 1 holds since \(d_{ij} \leq d_{i,j+1} + 1\).

**Induction on \(i+j\):**

- Basis is trivial when \(i=0\) or \(j=0\) (if we agree that \(d_{-1,j} = d_{0,j}\)).
- Induction step: there are 3 possibilities:
  - On minimization the \(d_{ij}\) is calculated from entry \(d_{i-1,j}\), hence \(d_{ij} \geq d_{i-1,j+1}\)
  - On minimization the \(d_{ij}\) is calculated from entry \(d_{i,j-1}\), hence \(d_{ij} \leq d_{i-1,j} + 1\).
  - On minimization the \(d_{ij}\) is calculated from entry \(d_{i-1,j-1}\), hence \(d_{ij} \leq d_{i-1,j} + 1\). Analagous to 2.
- Hence, \(d_{i,j+1} \leq d_{ij}\)

**Transform the matrix into \(f_{kp}\)**

- For each diagonal only show the position (row index) where the value is increased by 1.
- Also, one can restrict the matrix \(d_{ij}\) to only this part where \(d_{ij} \leq d_{mn}\), since only those \(d_{ij}\) can be on the shortest path.
- We’ll use the matrix \(f_{kp}\) that represents the diagonals of \(d_{ij}\):
  - \(f_{kp}\) is a row index \(i\) from \(d_{ij}\), such that on diagonal \(k\) the value \(p\) reaches row \(i\) \((d_{ip} = p\) and \(i=\text{const}\)).
  - Initialization: \(f_{0j} = 1\) and \(f_{ip} = \infty\) when \(p \leq |k|-1\);
  - \(d_{mn} = p\), such that \(f_{mp} = m\)

---

**Example**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0123456</td>
<td>a b c b c</td>
</tr>
<tr>
<td>b</td>
<td>1123456</td>
<td>0 12345</td>
</tr>
<tr>
<td>a</td>
<td>222234</td>
<td>a 22224</td>
</tr>
<tr>
<td>a</td>
<td>333323</td>
<td>a 33334</td>
</tr>
<tr>
<td>b</td>
<td>443323</td>
<td>a 44334</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

- In orig. matrix \(d_{ij}\) there are 42 values, and in diagonal matrix \(f_{kp}\) 12 values (exc. \(-\infty\))
- 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.
Calculating matrix \((f_{kp})\) by columns

- Assume the column \(p-1\) has been calculated in \((f_{kp})\), and we want to calculate \(f_{kp}\) (the region of \(d_{ij}=p\)).
- On diagonal \(k\) values \(p\) reach at least the row \(t=\max(f_{kp+1,p-1}, f_{k-1,p-1}, f_{k+1,p-1}+1)\) if the diagonal \(k\) reaches so far.
- If on row \(t+1\) additionally \(a_i=b_j\) on the same diagonal, then \(d_{ij}\) cannot increase, and value \(p\) reaches row \(t+1\).
- Repeat previous step until \(a_i\neq b_j\) on diagonal \(k\).

Algorithm \(A()\): calculate \(f_{kp}\)

\[
(A(k,p))
\]

1. \(t = \max(f_{k,p-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1)\)
2. while \(a_{t+1} = b_{t+1+k}\) do \(t = t+1\)
3. \(f_{kp} = \begin{cases} \text{undefined} & \text{if } t>m \text{ or } t+k>n \\ \text{else } t \end{cases}\)

Algorithm: Diagonal method by columns

\(p=-1\)

while \(f_{p,m,p} \neq m\)

\(p=p+1\)

for \(k=-p\) to \(p\) do

\(f_{kp} = A(k,p)\)

\(t = \max(f_{k,p-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1)\)

while \(a_{t+1} = b_{t+1+k}\) do \(t = t+1\)

\(f_{kp} = \begin{cases} \text{undefined} & \text{if } t>m \text{ or } t+k>n \\ \text{else } t \end{cases}\)
• $p$ can only occur on diagonals $-p \leq k \leq p$.
• Method can be improved since $k$ is often such that $f_{p}$ is undefined.
• We can decrease values of $k$:
  – $-m \leq k \leq n$ (diagonal numbers)
  – Let $m \leq n$ and $d_{i,j}$ on diagonal $k$.
    • if $-m \leq k \leq 0$ then $|k| \leq d_{i,j} \leq m$
    • if $1 \leq k \leq n$ then $k \leq d_{i,j} \leq k + m$
    • Hence, $-m \leq k \leq m$ if $p \leq m$ and $p - m \leq k \leq p$ if $p \geq m$

**Extensions to basic edit distance**

• New operations
• Variable costs
• Time Warping

**Transposition (ab → ba)**

• **E4: Transposition**
  
  $a_{i}a_{i+1} \rightarrow b_{j}b_{j+1}$, s.t. $a_{i}=b_{j+1}$ and $a_{i+1}=b_{j}$

• (e.g.: lecture → letcure)

**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 $A \xrightarrow{x} y$ then after the operation, $A \xrightarrow{y}$
• 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 $w(x \rightarrow y) \geq 0$.
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)
- ...

Examples

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Corrected</th>
</tr>
</thead>
<tbody>
<tr>
<td>näituseks</td>
<td>näiteks</td>
</tr>
<tr>
<td>Ahwrika</td>
<td>Aafrika</td>
</tr>
<tr>
<td>weikese</td>
<td>väikese</td>
</tr>
<tr>
<td>materjaal</td>
<td>materjal</td>
</tr>
</tbody>
</table>
```

```
tuseks -> teks
a -> aa, hw -> f
w -> v, e -> ä
aa -> a
```

Links

- Est-Eng; Old Estonian; Est-Rus transliteration
- Pronunciation
- Github (Reina Uba; Siim Orasmaa)
  - [https://github.com/oras/genEditDist](https://github.com/oras/genEditDist)
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/agp/lehre/ADP/
- Algebraic dynamic programming
  - Functional style
  - Haskell compiles into C