Example

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

```
2 1 7 1 4 3
```

- 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

- Avoid calculating repeating subproblems
- $fib(1)=fib(0)=1$;
- $fib(n) = fib(n-1)+fib(n-2)$
- Although natural to encode (and a useful task for novice programmers to learn about recursion) recursively, this is inefficient.

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- 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.
Structure within the problem

- The fact that it is not a tree indicates overlapping subproblems.

A dynamic-programming algorithm solves every subproblem 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.
Matrix multiplication

- for \( i = 1 \ldots n \)
  - for \( j = 1 \ldots k \)
    
    \[
    c_{ij} = \sum_{x=1}^{m} a_{ix} b_{xj}
    \]

\[
A \times B = C
\]

\[O(nmk)\]

Chain matrix multiplication

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.
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

\[
P(n) = \frac{1}{n+1} \binom{2n}{n} + \sum_{k=1}^{n-1} P(k) P(n-k)\]

if \(n = 1\),

\[
P(n) = \Omega(4^n / n^{3/2})\]

if \(n \geq 2\).

\[
A_1 A_2 A_3 A_4
\]

- \((A_1 (A_2 A_3)) A_4)
- \((A_1 A_2) (A_3 A_4))
- ((A_1 A_2) A_4) A_3
- (((A_1 A_2) A_3) A_4)

Let’s crack the problem

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

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

Recursion

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

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 \(A_{i,k} \cdots A_k\) becomes

\[
m[i,j] = \min_{0 \leq k \leq j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} \quad \text{if } i < j;
\]

\[
m[i,j] = 0 \quad \text{if } i = j.
\]

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 \(A_{i,k} \cdots A_k\) to obtain an optimal parenthesization. That is, \(s[i,j]\) equals a value \(k\) such that \(m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\).

\[
m[i,j] = \min_{0 \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} \quad \text{if } i < j.
\]
Recursion

• Checks all possibilities...

• But – there is only a few subproblems – choose i, j s.t. 1 ≤ i ≤ j ≤ 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.

Example

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 O(n^3) \( \Rightarrow \) \( \Theta(n^3) \)
• Space \( \Theta(n^2) \)

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_5A_6))
```

matrix dimensions:

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>30X35</td>
<td>35X15</td>
<td>15X20</td>
<td>20X10</td>
<td>10X20</td>
<td>20X25</td>
</tr>
</tbody>
</table>

```c
// 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
```

```c
#define MATRIX-CHAIN-ORDER(p)
1 n = length[p] - 1
2 for i = 1 to n
3 do s[i, i] = 0
4 for j = 2 to n
5 do for i = 1 to n - j + 1
6 do j = i + 1
7 m[i, j] = =
8 for k = i to j - 1
9 do q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]
10 if q < m[i, j]
11 then m[i, j] = q
12 q[i, j] = k
13 return s and q
```
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)

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 tree

Parse trees. (a) The parse tree for the parenthesized product ($A_1(A_2A_3)(A_4A_5A_6)$) 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 $i = 1, 2, \ldots, 6$. 

Similarity

• How can we measure the similarity of two strings?

• When are the two things “almost” the same?

Edit distance (Levenshtein distance)

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

How can we calculate this?

\[
D(a, b) =
\begin{bmatrix}
\min & a & \downarrow \\
\beta & \begin{array}{c}
\text{1. } D(a, \beta)
\text{2. } D(a, b) + 1
\end{array} & b
\end{bmatrix}
\]
How can we calculate this efficiently?

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

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

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

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 \( (d_{ij}) \).
for \( i=0 \) to \( m \) do \( d_{0j} = i \);
for \( j=0 \) to \( n \) do \( d_{ij} = j \);
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 \( d_{mn} \)
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 $A=B$
  - $D(A,B) = D(B,A)$
  - $D(A,C) \leq D(A,B) + D(B,C)$

Alignment

Path of edit operations

- Optimal solution can be calculated afterwards
  - Quite typical in dynamic programming

Three possible minimizing paths

- Add into $\text{pred}[i,j]$
  - $(i-1,j-1)$ 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 $d_{ij} = d_{i,j-1} + 1$
Multiple paths possible

- All paths are correct
- There can be many (how many?) paths

What are the other questions in edit distance calculations?

- Space complexity
- Time complexity
- Other ways to look at the algorithm(s)
- Applications
- More complex notions of similarity
- ...

Space can be reduced

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>b</th>
<th>c</th>
<th>C[m]</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>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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: $d_{nm}=D(A,B)$
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
    $d = \min(C + (\text{if } a_i=b_j \text{ then } 0 \text{ else } 1), C[i-1] + 1, C[i] + 1)$
    $C = C[i]$  // 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


Shortest path in the graph


All nodes at distance 1 from source
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)

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 $d_{ij} \leq d_{max}$ 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 $i$ ($d_{ip}$ and $j-i=k$).
  - Initialization: $f_{kp} = -1$ and $f_{kp} = \infty$ when $p \leq |k|-1$;
  - $d_{max} = p$, such that $f_{n,p} = m$
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 \(t=\max(f_{kp-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1)\) 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 \(f_{kp}\)**

\[
A(k,p) \\
1. \quad t = \max(f_{kp-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1) \\
2. \quad \text{while } a_{t+1} = b_{t+1+k} \text{ do } t = t+1 \\
3. \quad f_{kp} = \text{if } t>m \text{ or } t+k>n \text{ then undefined else } t
\]

**Algorithm: Diagonal method by columns**

\[
p = -1 \\
\text{while } f_{n-m,p} \neq m \\
p = p+1 \\
\text{for } k = -p \text{ to } p \text{ do } // \quad f_{kp} = A(k,p) \\
\quad t = \max(f_{kp-1}+1, f_{k-1,p-1}, f_{k+1,p-1}+1) \\
\quad \text{while } a_{t+1} = b_{t+1+k} \text{ do } t = t+1 \\
\quad f_{kp} = \text{if } t>m \text{ or } t+k>n \text{ then undefined else } t
\]

- \(p\) can only occur on diagonals \(-p \leq k \leq p\).
- Method can be improved since \(k\) is often such that \(f_{kp}\) is undefined.
- We can decrease values of \(k\):
  - \(-m \leq k \leq n\) (diagonal numbers)
  - Let \(m \leq n\) and \(d_{ij}\) on diagonal \(k\).
    - \(-m \leq k \leq 0\) then \(|k| \leq d_i, d_j \leq m\)
    - \(1 \leq k \leq n\) then \(k, d_i, d_j \leq k+m\)
    - Hence, \(-m \leq k \leq m\) if \(p \leq m\) and \(p-m \leq k \leq p\) if \(p \geq m\)
Some notes

- In applications small $D(A,B)$ are most interesting.
- Can modify the algorithm by providing maximum $t$
- Hence, $O(tm)$ - the smaller the $t$, the faster the algorithm.
- Space can be reduced by keeping only previous column
- How to output the shortest path?
  - Relatively simple, in time $O(s)$, outputs a single path.

Extensions to basic edit distance

- New operations
- Variable costs
- ...  

Transposition (ab → ba)

- E4: Transposition $a_{i+1} \rightarrow b_{j+1}$, s.t. $a_{i} = b_{j}$ and $a_{i+1} = b_{j+1}$
- (e.g.: lecture → letcure)

$$d(i,j) = \begin{cases} 
\min(d(i-1,j-1), d(i-1,j) + 1, d(i,j-1) + 1, d(i-2,j-2) + (\text{if } S[i-1,i] = T[j,j-1] \text{ then } 1 \text{ else } \infty) \end{cases}$$

Longest common subsequences

- Definition. String $C=c_1c_2...c_r$ is a subsequence (alamjada) of $A=a_1a_2...a_m$ if by removing from $A$ null or more characters, one can get $C$.
- String $C=c_1c_2...c_r$ is the longest common subsequence, LCS (pikim ühiné alamjada) of $A=a_1a_2...a_m$ and $B=b_1b_2...b_n$, if $C$ is the longest string that is both the subsequence of $A$ and $B$.
- $C=LCS(A,B)$
- The length of $LCS(A,B)$, $|C|$, can be used as the similarity measure for $A$ and $B$.
- $LCS(A,B)$ can be calculated similarly to edit distance
\[ |\text{LCS}(A,B)| = \left( |A| + |B| - D'(A,B) \right)/2 \]

- Let \( D'(A,B) \) the edit distance where the only allowed operations are insertion and deletion (no replace).

**Theorem**

a) \( |\text{LCS}(A,B)| = (|A| + |B| - D'(A,B))/2 \)

b) Let have two sets \( D'(A,B) \) with optimal nr of changes:

1. \( \alpha_1 \rightarrow \epsilon, \alpha_2 \rightarrow \epsilon, \ldots, \alpha_p \rightarrow \epsilon \) deletions from \( A \)
2. \( \epsilon \rightarrow \beta_1, \epsilon \rightarrow \beta_2, \ldots, \epsilon \rightarrow \beta_q \) insertions into \( B \)

Then \( \text{LCS}(A,B) = C \) can be constructed such that, \( C \) is \( A \) after deletions of 1. and \( C \) is \( B \) after deletion of all insertions 2. (insertions in 2. are reversely deletions from \( B \)).

**Proof b)**

- According to construction, \( C \) is uniquely defined
- \( C \) is a subsequence of \( A \) as well as \( B \).
- If \( C \) was not the longest, then there would be \( C' \), \( |C| < |C'| \) s.t. \( C' = \text{LCS}(A,B) \).
- But then \( D'(A,B) \) is \( |A|-|C'|+|B|-|C'| < |A|-|C|+|B|-|C| = \)
- \( D'(A,B) \), which is a contradiction.
- Hence, \( C = \text{LCS}(A,B) \).

**Proof a)**

- According to b) \( |\text{LCS}(A,B)| = |A| - p \) and \( |\text{LCS}(A,B)| = |B| - r \), or 2. \( |\text{LCS}(A,B)| = |A| + |B| - p - r = |A| + |B| - D'(A,B) \).

- **Example.** \( \text{LCS(england,inglismaa)=ngla.} \)
  \( D'(\text{england,inglismaa})=8, |\text{ngla}|=4=(7+9-8)/2 \).
- Diagonal lemma holds, but the increase always occurs by two.
- Time complexity \( O(mn) \), with diag. method \( O(\min(s,m) \cdot s) \) where \( s = D'(A,B), m = |A|, n = |B| \).
- There exists other algorithms for LCS (e.g. Hunt-Szymanski)

### Generalized edit distance

- Use more operations \( E_1 \ldots E_n \), 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 \( x \rightarrow y \) then \( x \rightarrow y \).

- 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 \( w(x \rightarrow y) \geq 0 \).

### Unix command diff

- Compares files row by row and searches the deviations from the LCS of the two files.

**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. Töugu => Tyugu (via Russian)
- ...

Examples

- 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

Examples

- näituseks → näiteks
- Ahwrika → Aafrika
- weikese → väikese
- materjaali → materjali

“kavalam” otsimine
  Dush, duš, dush?  
  Gorbatšov, Gorbatov, Горбачов, Gorbachev
  režim, rozhim, röim
**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/](http://www.techfak.uni-bielefeld.de/ags/pi/lehre/ADP/)
- Algebraic dynamic programming
  - Functional style
  - Haskell compiles into C