Example

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

\[
\begin{align*}
2 & \ 1 & \ 7 & \ 1 & \ 4 & \ 3 \\
\end{align*}
\]

\[
(2+1)\times7\times(1+4)\times3 = 21\times15 = 315 \\
2\times1 + 7 + 1\times4 + 3 = 16
\]

• Q: How to maximize the value of any expression?

\[
\begin{align*}
2 & \ 4 & 5 & 1 & 9 & 8 & 12 & 1 & 9 & 8 & 7 & 2 & 4 & 4 & 1 & 1 & 2 & 3 = ? \\
\end{align*}
\]


• 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
  
  \[
  \begin{align*}
  \text{fib}(1) &= \text{fib}(0) = 1; \\
  \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2)
  \end{align*}
  \]

- 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 \( A = a_1, a_2, ..., a_n \) and \( B = b_1, b_2, ..., b_m \)
  - \( E_1: \) Deletion \( a_i \rightarrow \varepsilon \)
  - \( E_2: \) Insertion \( \varepsilon \rightarrow b_i \)
  - \( E_3: \) Substitution \( a_i \rightarrow b_i \) (if \( a_i \neq b_i \))

- Other possible variants:
  - \( E_4: \) Transposition \( a_i a_{i+1} \rightarrow b_j b_{j+1} \) and \( a_i = b_{j+1} \) and \( a_{i+1} = b_j \) (e.g., lecture → letcure)

- How can we calculate this?

  \[
  D(a, b) = \begin{cases} 
  0 & \text{if } a = b \leq |a| = |b| \\
  D(a[1..|a|-1], b[1..|b|-1]) + 1 & \text{if } a < b \\
  D(a[1..|a|], b[1..|b|-1]) + 1 & \text{if } a > b \\
  \end{cases}
  \]

- How can we calculate this efficiently?

  \[
  D(S, T) = \min \begin{cases} 
  D(S[1..n-1], T[1..m-1]) + (S[n] = T[m]) \text{ if } n, m > 0 \\
  D(S[1..n], T[1..m-1]) + 1 & \text{if } m = 0 \\
  D(S[1..n-1], T[1..m]) + 1 & \text{if } n = 0 \\
  \end{cases}
  \]

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

  \[
  d(i, j) = \min \begin{cases} 
  d(i-1, j) + (S[i] = T[j]) \text{ if } i, j > 0 \\
  d(i-1, j) + 1 & \text{if } j = 0 \\
  d(i, j-1) + 1 & \text{if } i = 0 \\
  \end{cases}
  \]

**Edit distance (Levenshtein distance)**

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

**Recursion?**
Recursion?

\[
d(i,j) = \begin{cases} 
    0 & \text{if } i = 0 \text{ or } j = 0 \\
    d(i-1,j-1) + 1 & \text{if } a_i = b_j \\
    d(i-1,j) + 1 & \text{if } a_i \neq b_j \\
    d(i,j-1) + 1 & \text{if } a_i = b_j \\
    d(i-1,j-1) & \text{if } a_i \neq b_j
\end{cases}
\]

Example

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \mathbf{n} \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \mathbf{m} \\
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \mathbf{m} \\
3 & 4 & 5 & 6 & 7 & 8 & 9 & \mathbf{m} \\
4 & 5 & 6 & 7 & 8 & 9 & \mathbf{m} \\
5 & 6 & 7 & 8 & 9 & \mathbf{m} \\
6 & 7 & 8 & 9 & \mathbf{m} \\
7 & 8 & 9 & \mathbf{m} \\
8 & 9 & \mathbf{m} \\
9 & \mathbf{m} \\
\end{array}
\]

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} + (if a_i==b_j then 0 else 1), 
                  d_{i-1,j} + 1, 
                  d_{i,j-1} + 1 )
return d_{mn}
Dynamic programming

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

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
  
  ![Diagram of Edit Distance](image)

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

Three possible minimizing paths

- Add into \( \text{pred}[i,j] \)
  - \((i-1,j-1)\) if \( d_{ij} = d_{i-1,j-1} + (\text{if } a_i=b_j \text{ then } 0 \text{ else } 1) \)
  - \((i-1,j)\) if \( d_{ij} = d_{i-1,j} + 1 \)
  - \((i,j-1)\) if \( d_{ij} = d_{i,j-1} + 1 \)

![Matrix](image)

The path (in reverse order) \( a \rightarrow c_6, b_5 \rightarrow b_5, c_4 \rightarrow c_4, a_3 \rightarrow a_3, a_2 \rightarrow b_2, b_1 \rightarrow a_1 \).
Multiple paths possible

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

Space can be reduced

![Matrix example](image)

Calculation of $D(A,B)$ in space $\Theta(m)$

**Input:** $A=a_1, a_2, ..., a_m$, $B=b_1, b_2, ..., b_n$  
(choose $m \leq n$)

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

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

$d = \min(C + (\text{if } a_i == b_j \text{ then } 0 \text{ else } 1), C[i-1] + 1, C[i] + 1)$

$C[i] = \text{// memorize new "diagonal" value}$

$C[i] = d$

write $C[m]$

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

Shortest path in the graph

![Graph example](image)

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)

Diagonal

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

Diagonal lemma

Lemma: For each \(d_{ij}\), 1≤i≤m, 1≤j≤n holds: \(d_{ij} = d_{i-1,j-1} \) or \(d_{ij} = d_{i-1,j-1} + 1\)

Proof: Since \(d_{ij}\) is an integer, show:
1. \(d_{ij} \leq d_{i-1,j-1} + 1\)
2. \(d_{ij} \geq d_{i-1,j-1}\)

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 \(d_{ij} \leq d_{mn}\) 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 \(p\) from \(d_{kp}\) such that on diagonal \(k\) the value \(p\) reaches row \(i\) (\(d_{ij}\) and \(j = k\)).
  - Initialization: \(f_{0,-1} = 1\) and \(f_{-1,-\infty}\) when \(p \leq |k| - 1\)
  - \(d_{mn} = p\), such that \(f_{m,n} = m\)
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 \(t = \max( f_{kp}+1, f_{k+1,p-1}, f_{k+1,p+1} ) \) 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 \(f_{kp}\)

\[
A(k,p) = \begin{cases} 
  t = \max( f_{k,p-1}+1, f_{k+1,p-1} ) & \text{if } t > m \text{ or } t+k > n \text{ then undefined} \\
  f_{kp} & \text{else}
\end{cases}
\]

Algorithm: Diagonal method by columns

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

- \(f_{kp}+1\) - same diagonal
- \(f_{k-1,p}\) - diagonal below
- \(f_{k+1,p}+1\) - diagonal above

\[
\begin{array}{cccccccc}
  a & b & a & c & b & c \\
  0 & 1 & 2 & 3 & 4 & 5 & 6 \\
  b & 1 & 2 & 3 & 4 & 5 & 6 \\
  a & 2 & 2 & 2 & 2 & 2 & 2 \\
  c & 3 & 3 & 3 & 3 & 3 & 3 \\
  b & 4 & 4 & 4 & 4 & 4 & 4 \\
  a & 5 & 4 & 4 & 4 & 4 & 4 \\
  t & 0,2, \ldots \text{ t is } \max(3,2,3) = 2 \\
  f(0,2) \Rightarrow f(0,2) = 5 \\
  \end{array}
\]

- \(p\) 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 \(k\)
    - if \(-m \leq k \leq 0\) then \(|k| \leq d_{ij} \leq m\)
    - if \(1 \leq k \leq n\) then \(k \leq d_{ij} \leq k+m\)
    - Hence, \(-m \leq k \leq m\) if \(p \leq m\) and \(-p \leq k \leq p\) if \(p \leq m\)
Extensions to basic edit distance

- New operations
- Variable costs
- Time Warping

Dynamic Time Warp (simplest)

```java
int DTWDistance(s: array [1..n], t: array [1..m]) {
    DTW := array [0..n, 0..m]
    for i := 1 to n DTW[i, 0] := infinity
    for i := 1 to m DTW[0, i] := infinity
    DTW[0, 0] := 0
    for i := 1 to n
        for j := 1 to m
            DTW[i, j] := dist(s[i], t[j]) +
                minimum: DTW[i-1, j], // insertion
                DTW[i , j-1], // deletion
                DTW[i-1, j-1]) // match
    return DTW[n, m]
}
```

DTW example

Transposition (ab → ba)

- **E4: Transposition**
  - a_i-1 → b_j, s.t. a_i = b_j+1 and a_i+1 = b_j
  - (e.g., lecture → letcure)

Generalized edit distance

- Use more operations E1...En, and to provide different costs to each.
- **Definition.** Let x, y ∈ Σ*. Then every x → y is an edit operation. Edit operation replaces x by y.
  - If Axy then after the operation, Axyy
- We note by w(x → y) the cost or weight of the operation.
- Cost may depend on x and/or y. But we assume w(x → y) ≥ 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)
- ...

Examples

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

“kavalam” otsimine
Dush, dušš, dushsh ?
Gorbatšov, Gorbatschov, Горбачов,
Gorbachev
režim, režhim, riim

tuseks -> teks
a -> aa ,   hw -> f
w -> v , e -> ā
aa -> a
Links

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

Matrix multiplication

For i = 1..n
  for j = 1..k
    C[i,j] = Σ_{x=1..m} a_{i,x} b_{x,j}

O(nmk)

```
if columns [A] ≠ rows [B]
then error "incompatible dimensions"
else for i = 1 to rows [A]
  do for j = 1 to columns[B]
    do C[i,j] = 0
  for k = 1 to columns [A]
return C
```
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

\[ A_1 \times A_2 \times A_3 \times A_4 \]

\( A_1 \) has dimension \( p_{i-1} \times p_i \),

fully parenthesize the product \( A_1A_2...A_n \) 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 \( k \)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:

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

- \[ P(n) = C(n-1) \], where \( C(n) = \Omega(4^n/n^{3/2}) \).
- 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

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

- Optimal parenthesization of \( A_1 \cdot A_2 \cdot \ldots \cdot A_n \) 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 \( A_1 \cdot A_2 \cdot \ldots \cdot A_k \)

Recursion

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

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 \( A_{1,n} \) becomes

\[
m(i,j) = \min_{i \leq k \leq j} \left[ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right] 
\]

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

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.
Example

\[(A_1(A_2A_3))(A_4A_5A_6))\]

Matrix dimensions:
- \(A_1: 30 \times 35\)
- \(A_2: 35 \times 15\)
- \(A_3: 15 \times 5\)
- \(A_4: 5 \times 10\)
- \(A_5: 10 \times 20\)
- \(A_6: 20 \times 25\)

- A simple inspection of the nested loop structure of \textsc{Matrix-Chain-Order} yields a running time of \(O(n^3)\) for the algorithm. The loops are nested three deep, and each loop index \((l, i, j)\) 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 \(s[1..n, 1..n]\) to determine the best way to multiply the matrices.

Multiply using S table

\begin{align*}
\text{\textsc{Matrix-Chain-Multiply}}(A, s, i, j) \\
1 & \text{if } j > i \\
2 & \quad X = \text{\textsc{Matrix-Chain-Multiply}}(A, s, i, s[i, j]) \\
3 & \quad Y = \text{\textsc{Matrix-Chain-Multiply}}(A, s, s[i, j]+1, j) \\
4 & \quad \text{return } \text{\textsc{Matrix-Multiply}}(X, Y) \\
5 & \text{else return } A_i \tag{(tabulated)}
\end{align*}

Elements of dynamic programming

• \textbf{Optimal substructure} within an optimal solution
• \textbf{Overlapping subproblems}
• \textbf{Memoization}

• A \textbf{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 $((A_1(A_2A_3))(A_4(A_5A_6)))$ 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

$$\eta(i, j) = \begin{cases} 0 & \text{if } i = j, \\ \min_{k \in (i, j)} \{\eta(i, k) + \eta(k + 1, j) + w(v_{i-1} + v_{j+1})\} & \text{if } i < j. \end{cases} \quad (16.7)$$