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 = 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.

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>TARTU</th>
<th>STARTUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>STARTUP</td>
<td>STAARDUST</td>
</tr>
</tbody>
</table>

How can we calculate this?

Recursive view

\[
D(a, b) = \min \begin{cases} 
D(a, b) & \text{if } a = b \\
D(a, b) + 1 & \text{if } a \neq b \\
D(a-1, b) & \text{if } b \neq \epsilon \\
D(a, b-1) & \text{if } a \neq \epsilon \\
D(a-1, b-1) & \text{if } a = b \\
\end{cases}
\]

A bit like proof by induction

How can we calculate this efficiently?

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

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

\[
d(i, j) = \min \begin{cases} 
d(i-1, j) + (S[i]=T[j]) \cdot 0 : 1 \\
d(i, j-1) + 1 \\
d(i-1, j-1) + 1 \\
\end{cases}
\]
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 $(d_{ij})$, $0\leq i\leq m$, $0\leq j\leq n$.

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

\[ d_{ij} = \min \left( d_{i-1,j-1} + (\text{if } a_i = b_j \text{ then } 0 \text{ else } 1), \right. \]
\[ \left. d_{i-1,j} + 1, \quad d_{i,j-1} + 1 \right) \]

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

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

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

Three possible minimizing paths

- Add into \( \text{pred}[i,j] \)
  - \( (i-1,j-1) \) if \( a_i = b_j \) then 0 else 1
  - \( (i-1,j) \) if \( a_i \neq b_j \) then 1
  - \( (i,j-1) \) if \( a_i \neq b_j \) then 1
The path (in reverse order) ε → c₆ → b₅ → c₄ → a₃ → a₂ → b₂ → a₁.

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: $A=a₁a₂⋯aₘ$, $B=b₁b₂⋯bₙ$ (choose $m\leqslant n$)
Output: $d_{∞} = D(A, B)$

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
    $d = \min(C + (\text{if } aᵢ = bₖ \text{ then } 0 \text{ else } 1), C[i-1] + 1, C[i] + 1)$
    $C[i] = d$  // 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
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 lemma

Lemma: For each \(d_{ij}\), 1≤i≤m, 1≤j≤n holds: \(d_{ij} = d_{i-1,j} + 1 \) or \(d_{ij} = d_{i,j-1} + 1\). (notice that \(d_{ii}\) and \(d_{i+1,i}\) are on the same diagonal)

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

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

Induction on \(i+j\):

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

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

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 \(d_{ij} \leq d_{\text{max}}\) 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 \(i\) (\(d_{ij} = p\) and \(j-i=k\)).
  - Initialization: \(f_{k0} = 0\) and \(f_{p,\text{max}} = \infty\) when \(p \leq |k| - 1\)
  - \(d_{\text{max}} = p\), such that \(f_{k0} = \text{m}\)
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 \(t=\max(f_{k,p-1}+1, f_{k-1,p}+1, f_{k+1,p}+1)\) if the diagonal \(k\) reaches so far.
- If on row \(t+1\) additionally \(a_t = b_{t+k}\) on the same diagonal, then \(d_t\) cannot increase, and value \(p\) reaches row \(t+1\).
- Repeat previous step until \(a_t \neq b_{t+k}\) on diagonal \(k\).

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

\[A(k,p)\]

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

**Algorithm: Diagonal method by columns**

\(p=-1\)
\nwhile \(f_{n,m,p} \neq m\)
\n\(p=p+1\)
\nfor \(k=-p\) to \(p\) do // \(f_{kp} = A(k,p)\)
\n\(t = \max(f_{k,p-1}+1, f_{k-1,p}+1, f_{k+1,p}+1)\)
\nwhile \(a_{t+1} = b_{t+1+k}\) do \(t = t+1\)
\n\(f_{kp} =\) if \(t>m\) or \(t+k >n\) then undefined else \(t\)
Extensions to basic edit distance

- New operations
- Variable costs
- Generalization

Transposition (ab → 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 → letcure)

Generalized edit distance

- Use more operations E1...En, 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 \( A = uvx \) 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 \( w(x \rightarrow y) \geq 0 \).

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

- 1:11000 aerial
- 3:1350000 aerial
- 1:50000 sketch
- 1:5000000 sketch
- 1:3000000 cart
- 1:15000000 cart

Links
- Est-Eng; Old Estonian; Est-Rus transliteration
- Pronunciation
  - https://bit.devut.ugp/~orasmaa/ing_ligikaudne/
- Github (Reina Uiba; Siim Orasmaa)
  - https://github.com/oras/geneEditDist

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

näituseks – näiteks
Ahwrika - Afrika
weikes - väikes
materjaali - materjali

tuseks - tekks
a -> aa , hw -> f
w -> v , e -> å
aa -> a

“kavalam” otsimine
Dush, dušš, dushšsh?
Gorbatšov, Gorbatshov, Горбачов,
Gorbachev
režim, režim, riim
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...

Longest Common Subsequence (LCS)

![Longest Common Subsequence](image)

RNA structure

![RNA Structure](image)

Advanced Dynamic Programming

- Robert Giegerich:
  - [http://www.techfak.uni-bielefeld.de/ags/pa/seho/ADP/](http://www.techfak.uni-bielefeld.de/ags/pa/seho/ADP/)
- Algebraic dynamic programming
  - Functional style
  - Haskell compiles into C

Algorithmics (6EAP)

Time Warping

Jaak Vilo
2020 Fall
Dynamic Time Warp (simplest)

```c
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]
}
```

between the two compared sequences, i.e., physiological temporal patterns. The smaller the DTW
dissimilarity between two sequences, the similar are the two compared sequences.

\[
D(\alpha, \beta) = D(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} \left( \alpha_i - \beta_j \right)^2
\]

where \(\alpha_i, \beta_j\) \(\alpha_i, \beta_j\) and \(\alpha_i, \beta_j\) are the upper left, upper and left neighboring elements of the \(\alpha_i, \beta_j\) element of the DTW matrix.

- https://medium.com/datadriveninvestor/dynamic-time-warping-dtw-d51d1a1e4afc
Matrix multiplication

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

\[ A \times B = C \]

\[ O(nmk) \]

Chain matrix multiplication

- 6.5 Chain matrix multiplication

MATRIS-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 \([A]\)
7) do \( C[i, j] = C[i, j] + A[i, k] \times B[k, j] \)
8) return \( C \)

Multiplying an \( m \times n \) matrix by an \( n \times p \) matrix takes \( mn \times p \) multiplications, so a good
approximation. Using this formula, let's compare several different ways of evaluating
\( A \times B \times C = D \)

<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 \times 10 \times 20 \times 10 \times 30 \times 10 \times 30</td>
<td>7,200</td>
</tr>
<tr>
<td>((A \times (B \times C)) \times D)</td>
<td>10 \times 20 \times 1 \times 10 \times 10 \times 10 \times 30 \times 100 \times 4 \times 10</td>
<td>7,200</td>
</tr>
</tbody>
</table>

As you can see, the order of multiplications makes a big difference in the final running time. However, the natural greedy approach, to always perform the cheapest matrix multiplication available, leads to the second parentheses shown here and is therefore a failure.
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.

<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 + 20 \cdot 10 \cdot 100 + 20 \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 + 10 \cdot 1 \cdot 100</td>
<td>7,000</td>
</tr>
</tbody>
</table>

### Problem 13-4

- 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

\[
P(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 
\end{cases}
\]

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 = \(A_{1,k} \bullet A_{k+1,n}\)
- \(T(A_{1,k}) = T(A_{1,k}) + T(A_{k+1,n}) + T(A_{1,k} \bullet A_{k+1,n})\)
- \(T(A_{1,k})\) 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 \( A_{i,j} \)
- \( m[i, i] = 0 \)
- \( \text{cost}(A_{i,k} \cdot A_{k+1,j}) = p_{i-k} \cdot p_k \cdot p_j \)
- \( m[i, j] = m[i, k] + m[k+1, j] + p_{i-k} \cdot p_k \cdot p_j \).

This recursive equation assumes that we know the value of \( k \), which we don't. There are only \( j - i \) possible values for \( k \), 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 \( A_A \ldots A \) becomes

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j, \\
\min_{k<i<j} \{ m[i, k] + m[k+1, j] + p_{i-k} \cdot p_k \cdot p_j \} & \text{if } i < j. 
\end{cases}
\]

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

Example

\[
\begin{align*}
(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))) \\
\begin{array}{c}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
3 & 4 & 5 & 6 \\
5 & 6 & 7 & 8
\end{array}
\end{align*}
\]

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 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 \( \Omega(n^3) \Rightarrow \Theta(n^3) \)
- 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.

<table>
<thead>
<tr>
<th>Multiply using S table</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATRIX-CHAIN-MULTIPLY($A$, $s$, $i$, $j$)</td>
</tr>
<tr>
<td>1 if $j &gt; i$</td>
</tr>
<tr>
<td>2 then $X = MATRIX-CHAIN-MULTIPLY(A, s, i, s[j])$</td>
</tr>
<tr>
<td>3 $Y = MATRIX-CHAIN-MULTIPLY(A, s, s[i] + 1, j)$</td>
</tr>
<tr>
<td>4 return $MATRIX-MULTIPLY(X, Y)$</td>
</tr>
<tr>
<td>5 else return $A_i$</td>
</tr>
<tr>
<td>$((A_iA_jA_k)(A_{i+1}A_{i+2}A_{i+3}))$</td>
</tr>
</tbody>
</table>

91

92

93

94

95

96

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 $7 \cdot 3 = 21$ chords and divides the polygon into $7 \cdot 2 = 14$ triangles.

Parse tree

Parse trees. (a) The parse tree for the parenthesized product $((A_1(A_2(A_3(A_4(A_5A_6)))))$ 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 $v_i$ for $i = 1, 2, \ldots, 6$.

Optimal triangulation

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.