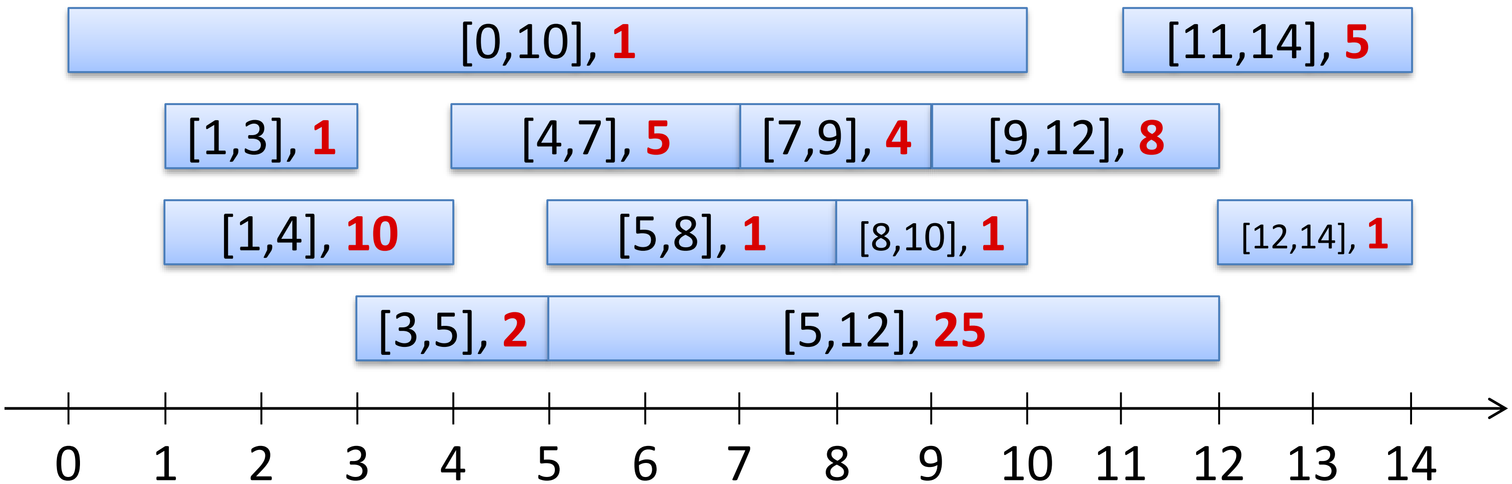
Chapter 3
Dynamic Programming

Algorithm Theory
WS 2016/17

Fabian Kuhn

Weighted Interval Scheduling

- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]
- Each interval has a **weight *w***



- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e., [4,7] and [7,9] are non-overlapping
- **Example:** Intervals are room requests of different importance

Recursive Definition of Optimal Solution

- Recall:
  - $W(k)$: weight of optimal solution with intervals $1, \dots, k$
  - $p(k)$: last interval to finish before interval k starts

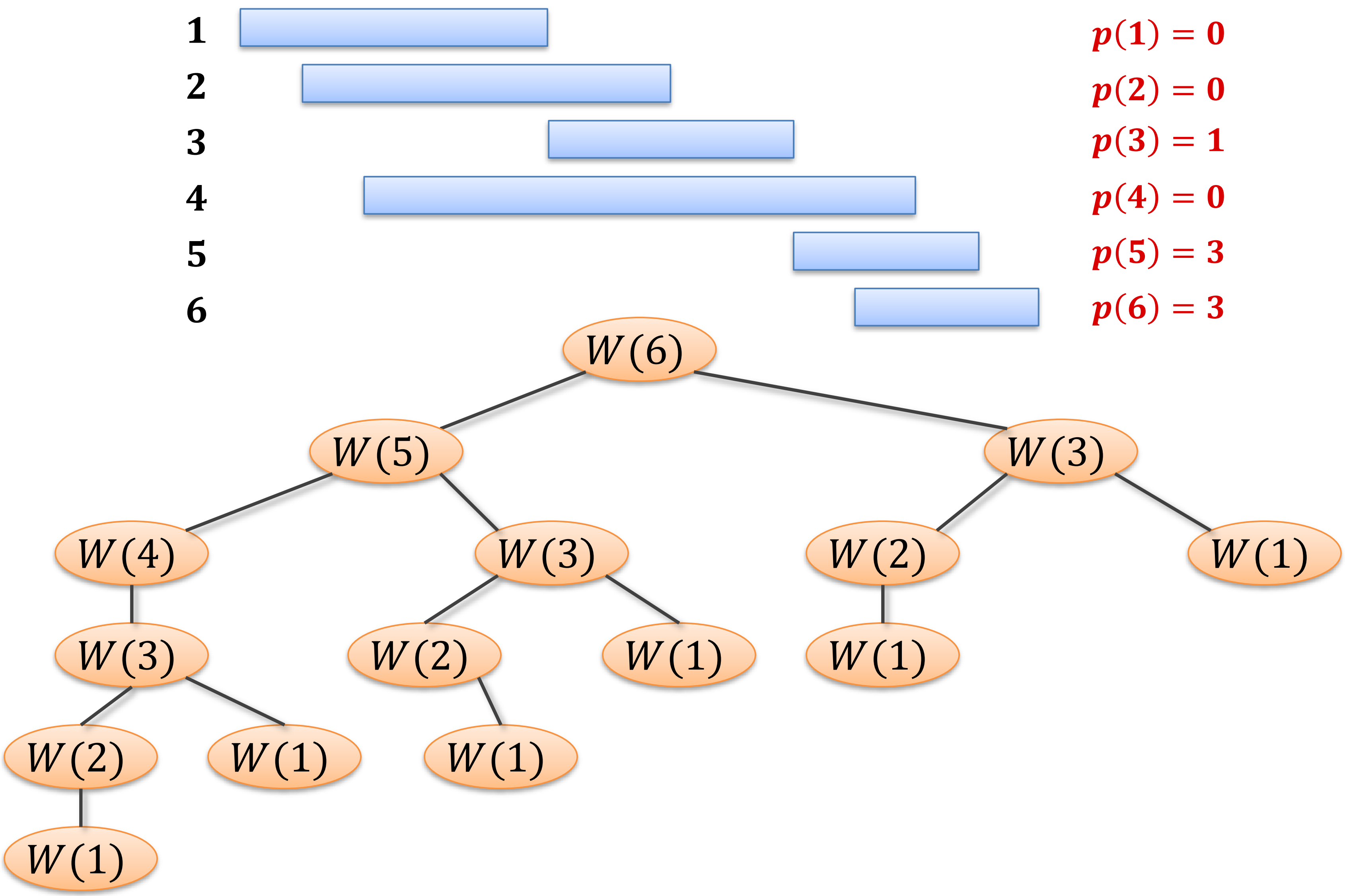
- Recursive definition of optimal weight:

$$\forall k > 1: \; W(k) = \max\{W(k-1), w(k) + W\big(p(k)\big)\}$$
$$W(1) = w(1)$$

Immediately gives a simple, recursive algorithm

```
Compute p(k) values for all k
W(k):
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    return x
```

Running Time of Recursive Algorithm



Memoizing the Recursion

- Running time of recursive algorithm: exponential!
- But, alg. only solves n different sub-problems: $W(1), \dots, W(n)$
- There is no need to compute them multiple times

Memoization: Store already computed values for future rec. calls

```
Compute p(k) for all k
memo = {};
W(k):
    if k in memo: return memo[k]
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    memo[k] = x
    return x
```

Dynamic Programming (DP)

DP $\approx$ Recursion + Memoization

Recursion: Express problem *recursively* in terms of
(a 'small' number of) *subproblems* (of the same kind)

Memoize: *Store* solutions for *subproblems*
reuse the stored solutions if the same subproblems
has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3)$, ...

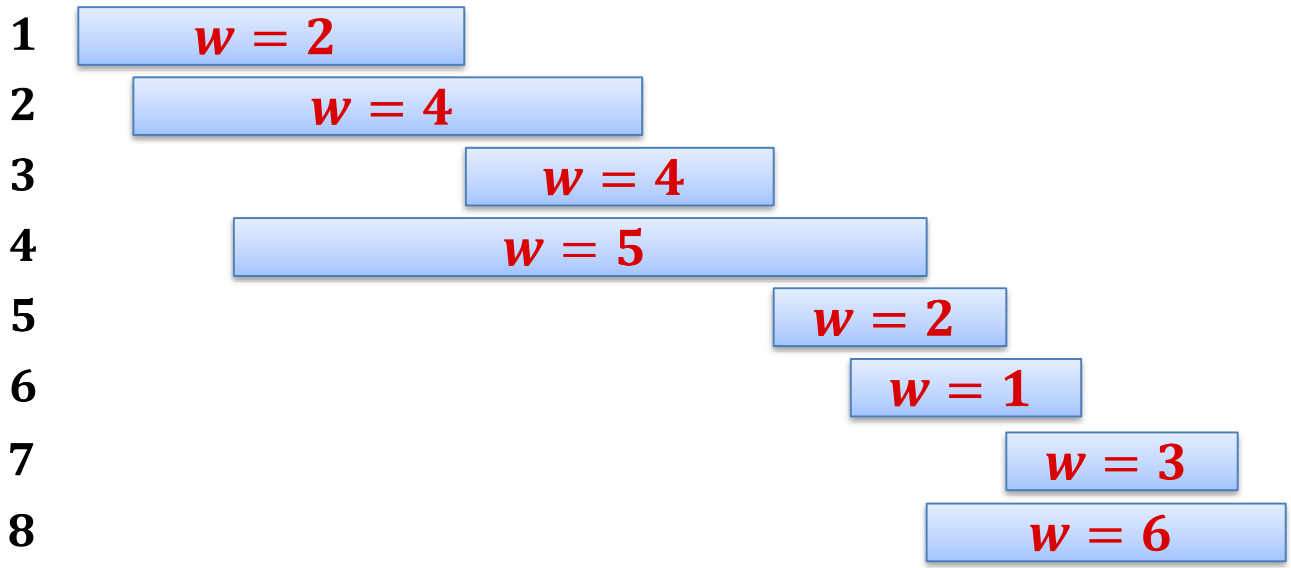
runtime $=$ #subproblems $\cdot$ time per subproblem

DP: Some History …

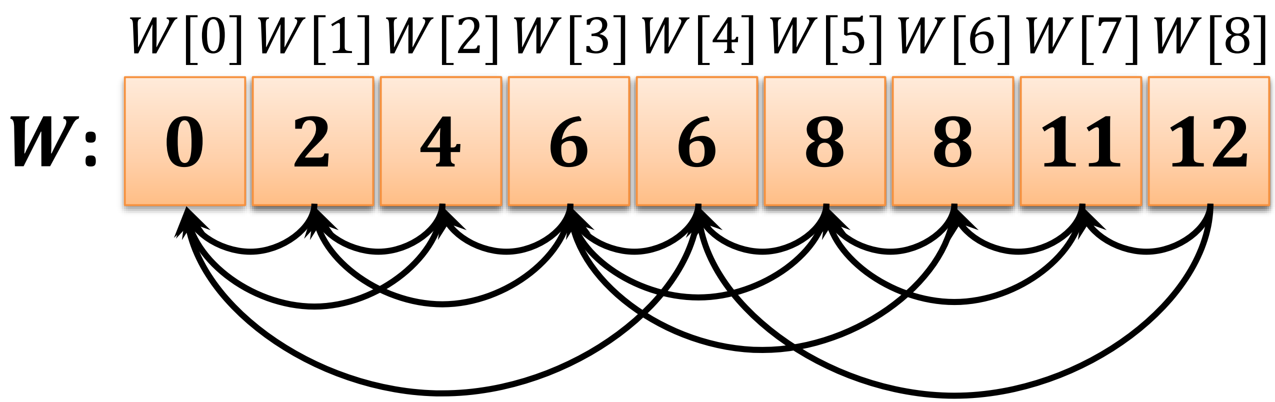
- Where das does the name come from?
- DP was developed by Richard E. Bellman in 1940s/1950s.
- In his autobiography, it says:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Example



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$
$p(7) = 5$
$p(8) = 4$



Computing the schedule: **store where you come from!**

Matrix-chain multiplication

Given: sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of matrices

Goal: compute the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$

Problem: Parenthesize the product in a way that minimizes the number of scalar multiplications.

Definition: A product of matrices is *fully parenthesized* if it is

- a single matrix
- or the product of two fully parenthesized matrix products, surrounded by parentheses.

Example

All possible fully parenthesized matrix products of the chain $\langle A_1, A_2, A_3, A_4 \rangle$:

$$(A_1(A_2(A_3A_4)))$$

$$(A_1((A_2A_3)A_4))$$

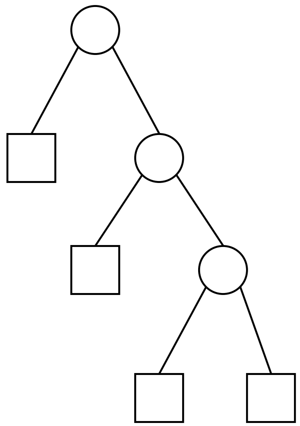
$$((A_1A_2)(A_3A_4))$$

$$((A_1(A_2A_3))A_4)$$

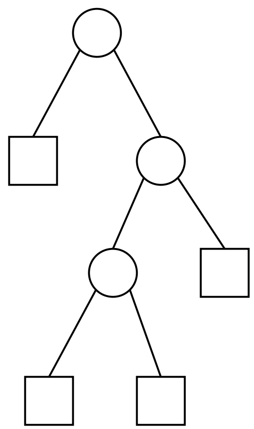
$$(((A_1A_2)A_3)A_4)$$

Different parentheses

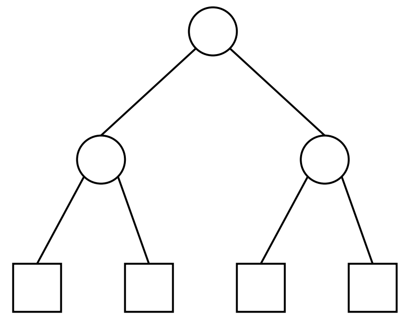
Different parenthesizations correspond to different trees:



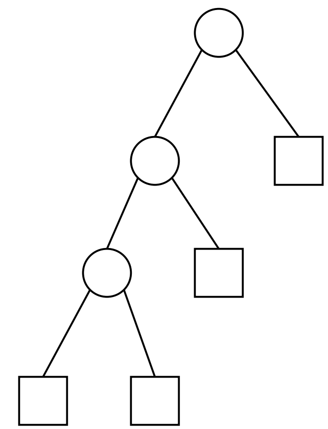
$$\big(A_1\big(A_2(A_3A_4)\big)\big)$$



$$\big(A_1\big((A_2A_3)A_4\big)\big)$$



$$\big((A_1A_2)(A_3A_4)\big)$$



$$\big(\big((A_1A_2)A_3\big)A_4\big)$$

Number of different parenthesizations

- Let $P(n)$ be the number of alternative parenthesizations of the product $A_1 \cdot \ldots \cdot A_n$:

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k), \qquad \text{for } n \geq 2$$

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

$$P(n+1) = C_n \qquad (n^{th} \text{ Catalan number})$$

- Thus: Exhaustive search needs exponential time!

Multiplying Two Matrices

$$A = \left(a_{ij}\right)_{p\times q}, \qquad B = \left(b_{ij}\right)_{q\times r}, \qquad A\cdot B = C = \left(c_{ij}\right)_{p\times r}$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Algorithm *Matrix-Mult*

Input: $(p \times q)$ matrix A, $(q \times r)$ matrix B

Output: $(p \times r)$ matrix $C = A \cdot B$

```
for i := 1 to p do
    for j := 1 to r do
        C[i,j] := 0;
        for k := 1 to q do
            C[i,j] := C[i,j] + A[i,k] · B[k,j]
```

Remark:

Using this algorithm, multiplying two $(n \times n)$ matrices requires n^3 multiplications. This can also be done using $O(n^{2.376})$ multiplications.

Number of multiplications and additions: $\boldsymbol{p \cdot q \cdot r}$

Matrix-chain multiplication: Example

Computation of the product $A_1\, A_2\, A_3$, where

A_1 : (50×5) matrix

A_2 : (5×100) matrix

A_3 : (100×10) matrix

a) Parenthesization $((A_1\, A_2)A_3)$ and $\big(A_1(A_2 A_3)\big)$ require:

$A' = \ (A_1\, A_2)$:

$A'' = (A_2 A_3)$:

$A' A_3$:

$A_1 A''$:

Sum:

Structure of an Optimal Parenthesization

- $(A_{\ell \ldots r})$: optimal parenthesization of $A_\ell \cdot \ldots \cdot A_r$

  For some $1 \leq k < n$: $(\boldsymbol{A_{1\ldots n}}) = \big((\boldsymbol{A_{1\ldots k}}) \cdot (\boldsymbol{A_{k+1\ldots n}})\big)$

- Any optimal solution contains optimal solutions for sub-problems

- Assume matrix A_i is a $(d_{i-1} \times d_i)$-matrix

- Cost to solve sub-problem $A_\ell \cdot \ldots \cdot A_r, \ell \leq r$ optimally: $C(\ell, r)$

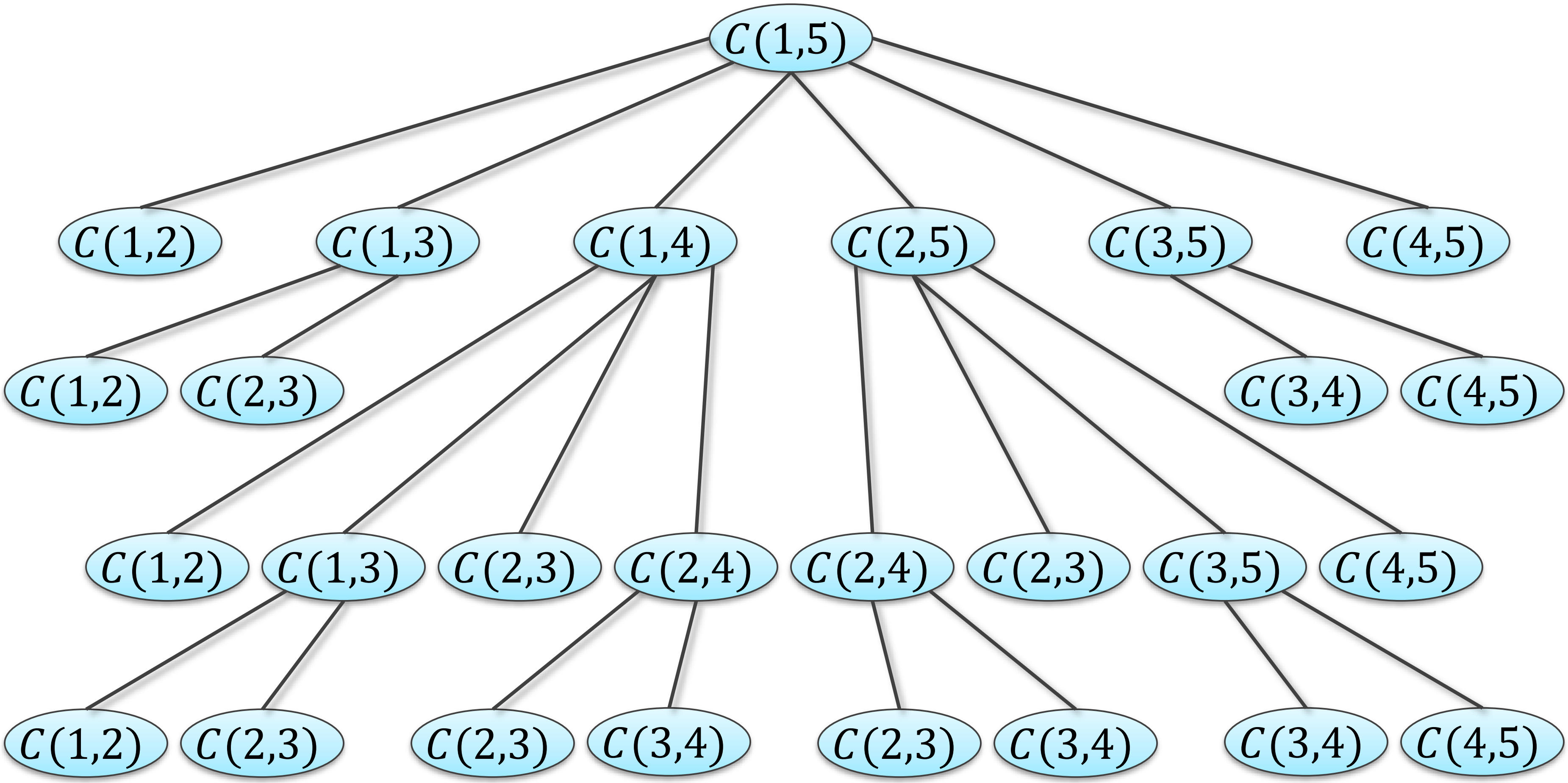
- Then:

$$\boldsymbol{C(a,b) = \min_{a \leq k < b} C(a,k) + C(k+1,b) + d_{a-1} d_k d_b}$$

$$\boldsymbol{C(a,a) = 0}$$

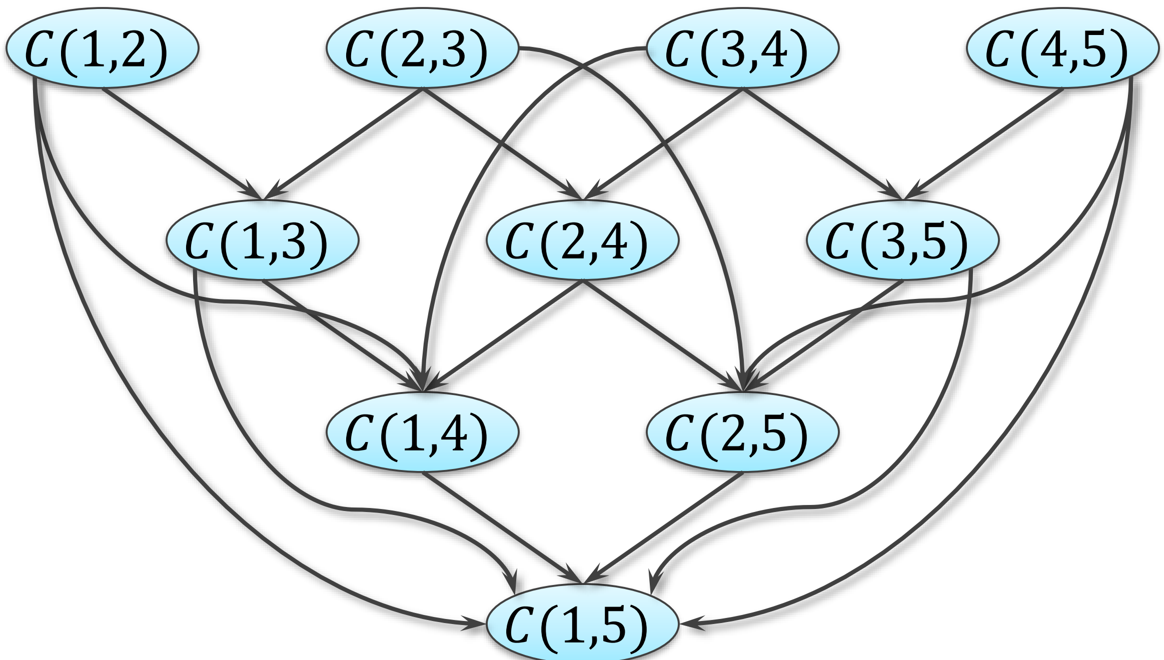
Recursive Computation of Opt. Solution

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



Using Meomization

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



Compute $A_1 \cdot \ldots \cdot A_n$:

- Each $C(i,j), i < j$ is computed exactly once → $O(n^2)$ values
- Each $C(i,j)$ dir. depends on $C(i,k), C(k,j)$ for $i < k < j$

Cost for each $C(i,j)$: $O(n)$ → overall time: $\boldsymbol{O(n^3)}$

Remarks about matrix-chain multiplication

1. There is an algorithm that determines an optimal parenthesization in time

$$O(n \cdot \log n).$$

2. There is a linear time algorithm that determines a parenthesization using at most

$$1.155 \cdot C(1, n)$$

multiplications.

Dynamic Programming

„Memoization“ for increasing the efficiency of a recursive solution:

- Only the *first time* a sub-problem is encountered, its solution is computed and then stored in a table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned

  (without repeated computation!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)

- Number of sub-problems that need to be considered is small

Knapsack

- n items $1, \ldots, n$, each item has weight w_i and value v_i
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t.} \quad S \subseteq \{1, \ldots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Recursive Structure?

- Optimal solution: $\mathcal{O}$
- If $n \notin \mathcal{O}$: $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$
- What if $n \in \mathcal{O}$?
  - Taking n gives value v_n
  - But, n also occupies space w_n in the bag (knapsack)
  - There is space for $W - w_n$ total weight left!

$$\mathrm{OPT}(n) = w_n + \text{optimal solution with first } n-1 \text{ items}$$
$$\text{and knapsack of capacity } W - w_n$$

A More Complicated Recursion

$\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$: value of optimal solution with items $1, \dots, k$
and knapsack of capacity x

Recursion:

Dynamic Programming Algorithm

Set up table for all possible $\mathrm{OPT}(k, x)$-values

- Assume that all weights w_i are integers!

	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$		$\cdots$						$\boldsymbol{W}$
$\mathbf{0}$												
$\mathbf{1}$												
$\mathbf{2}$												
$\mathbf{3}$												
$\vdots$												
$\boldsymbol{n}$												

Row $\boldsymbol{i}$, column $\boldsymbol{j}$:

$\boldsymbol{OPT(i, j)}$

Example

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12

  weight **value**

- $\boldsymbol{OPT(k,x) = \max\{OPT(k-1,x), OPT(k-1,x-w_k)+v_k\}}$

	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$
- Time per table entry: $O(1)$ → **overall time:** $\boldsymbol{O(nW)}$
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)
- Note: Time depends on W → can be exponential in n…
- And it is problematic if weights are not integers.

String Matching Problems

Edit distance:

- For two given strings A and B, efficiently compute the **edit distance $D(A, B)$** (# edit operations to transform A into B) as well as a minimum sequence of edit operations that transform A into B.

- **Example:** mathematician → multiplication:

$$\mathtt{m\ u}\underbrace{\ \ }_{\mathtt{l}}\mathtt{t\ i\ p\ l}\underbrace{\ \ }_{\mathtt{i\,c}}\mathtt{a\ t\ i\ o}\ \cancel{\mathtt{i}}\ \cancel{\mathtt{a}}\ \mathtt{n}$$

Edit Distance

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

```
m a - t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Edit Distance – Cost Model

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$
- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$
- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Recursive Structure

- Optimal "alignment" of strings (unit cost model)

  `bbcadfagikccm` and `abbagflrgikacc`:

```
- b b c a g f a - g i k - c c m
a b b - a d f l r g i k a c c -
```

- Consists of optimal "alignments" of sub-strings, e.g.:

```
-bbcagfa        -gik-ccm
            and
abb-adfl        rgikacc-
```

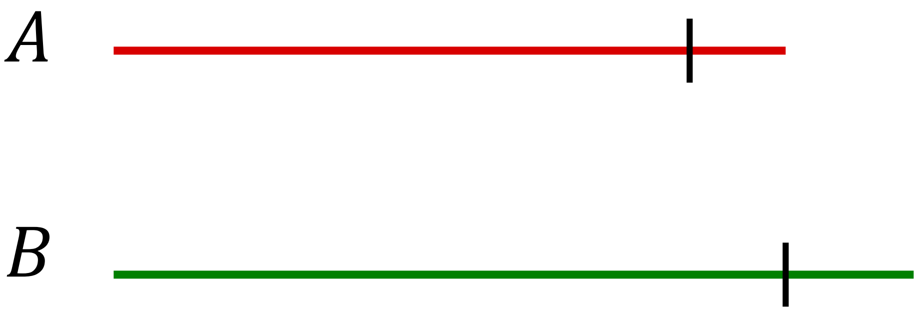
- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

$$D(A,B) = \min_{k,\ell}\{D(A_{1,k}, B_{1,\ell}) + D(A_{k+1,m}, B_{\ell+1,n})\}$$

Computation of the Edit Distance

Let $A_k := a_1 \dots a_k$, $B_\ell := b_1 \dots b_\ell$, and

$$D_{k,\ell} := D(A_k, B_\ell)$$



Computation of the Edit Distance

Three ways of ending an "alignment" between A_k and B_ℓ:

1. a_k is replaced by b_ℓ:

   $$D_{k,\ell} = D_{k-1,\ell-1} + c(a_k, b_\ell)$$

2. a_k is deleted:

   $$D_{k,\ell} = D_{k-1,\ell} + c(a_k, \varepsilon)$$

3. b_ℓ is inserted:

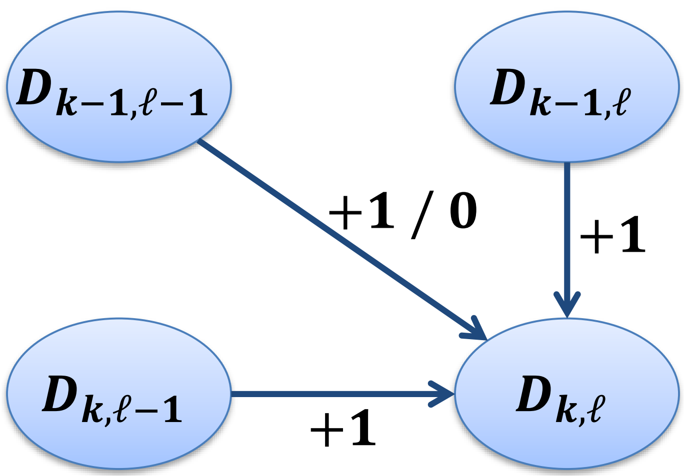
   $$D_{k,\ell} = D_{k,\ell-1} + c(\varepsilon, b_\ell)$$

Computing the Edit Distance

- Recurrence relation (for $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix} = \underbrace{\min \begin{Bmatrix} D_{k-1,\ell-1} & + 1 \;/\; 0 \\ D_{k-1,\ell} & + 1 \\ D_{k,\ell-1} & + 1 \end{Bmatrix}}_{\textbf{unit cost model}}$$

- Need to compute $D_{i,j}$ for all $0 \leq i \leq k$, $0 \leq j \leq \ell$:



Recurrence Relation for the Edit Distance

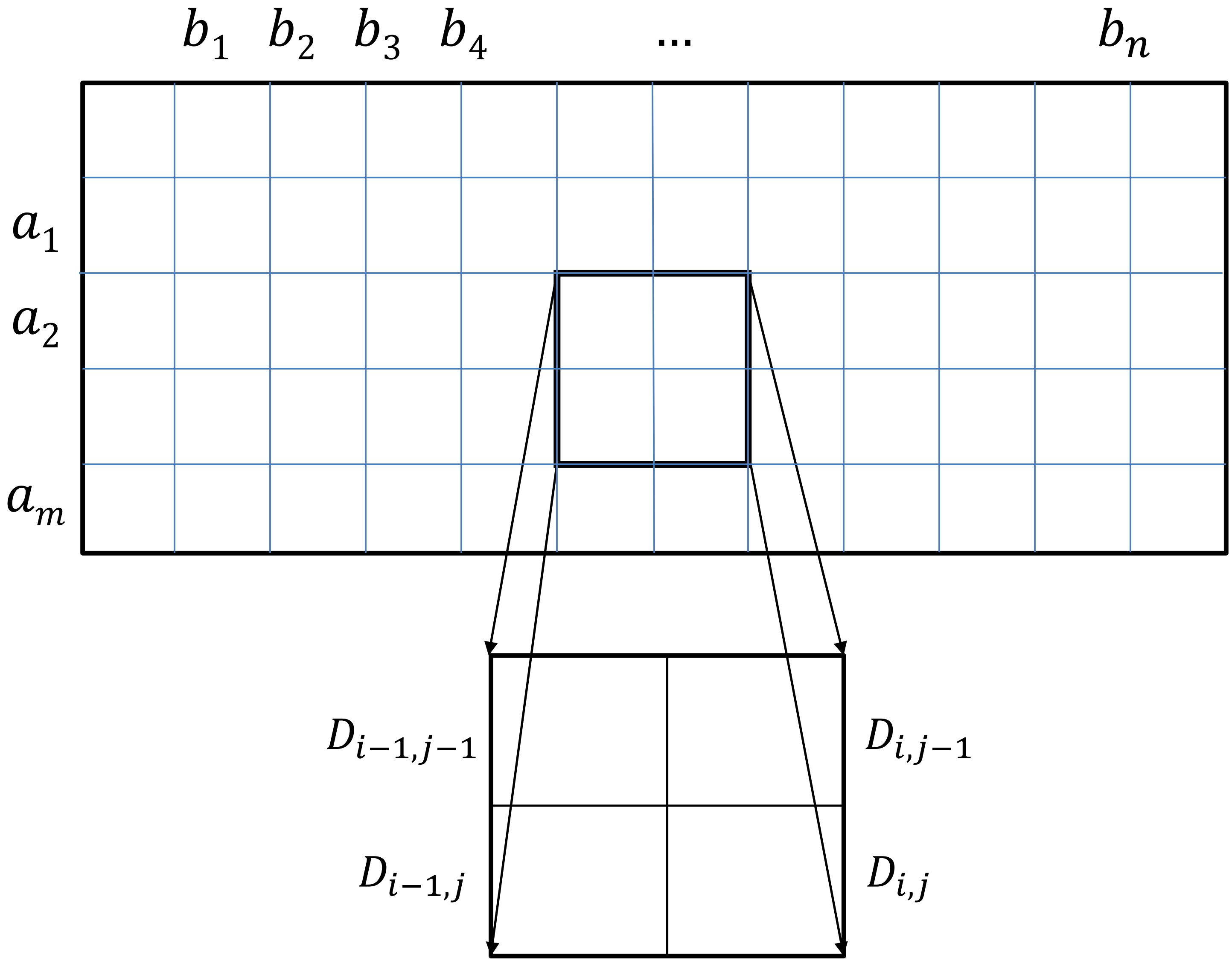
Base cases:

$$
\begin{aligned}
D_{0,0} &= D(\varepsilon, \varepsilon) = 0 \\
D_{0,j} &= D(\varepsilon, B_j) = D_{0,j-1} + c(\varepsilon, b_j) \\
D_{i,0} &= D(A_i, \varepsilon) = D_{i-1,0} + c(a_i, \varepsilon)
\end{aligned}
$$

Recurrence relation:

$$
D_{i,j} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix}
$$

Order of solving the subproblems



Algorithm for Computing the Edit Distance

Algorithm *Edit-Distance*

Input: 2 strings $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$

Output: matrix $D = \left(D_{ij}\right)$

$D[0,0] \coloneqq 0;$

for $i \coloneqq 1$ **to** m **do** $D[i,0] \coloneqq i;$

for $j \coloneqq 1$ **to** n **do** $D[0,j] \coloneqq j;$

for $i \coloneqq 1$ **to** m **do**

    for $j \coloneqq 1$ **to** n **do**

$$D[i,j] \coloneqq \min \left\{ \begin{array}{ll} D[i-1,j] & +1 \\ D[i,j-1] & +1 \\ D[i-1,j-1] & + c\left(a_i, b_j\right) \end{array} \right\};$$

Example

		a	b	c	c	a
b						
a						
b						
d						
a						

Computing the Edit Operations

Algorithm *Edit-Operations*(i, j)
Input: matrix D (already computed)
Output: list of edit operations

if $i = 0$ **and** $j = 0$ **then return** empty list

if $i \neq 0$ **and** $D[i, j] = D[i-1, j] + 1$ **then**
    return *Edit-Operations*$(i-1, j) \circ$ „delete a_i“

else if $j \neq 0$ **and** $D[i, j] = D[i, j-1] + 1$ **then**
    return *Edit-Operations*$(i, j-1) \circ$ „insert b_j“

else // $D[i, j] = D[i-1, j-1] + c(a_i, b_j)$
    if $a_i = b_i$ **then return** *Edit-Operations*$(i-1, j-1)$
    else return *Edit-Operations*$(i-1, j-1) \circ$ „replace a_i by b_j“

Initial call: *Edit-Operations*(*m,n*)

Edit Operations

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2	1	2	2	3	3
b	3	2	1	2	3	4
d	4	3	2	2	3	4
a	5	4	3	3	3	3

Edit Distance: Summary

- Edit distance between two strings of length m and n can be computed in $O(mn)$ time.

- Obtain the edit operations:
  - for each cell, store which rule(s) apply to fill the cell
  - track path backwards from cell (m, n)
  - can also be used to get all optimal "alignments"

- Unit cost model:
  - interesting special case
  - each edit operation costs 1