



# **Chapter 3**

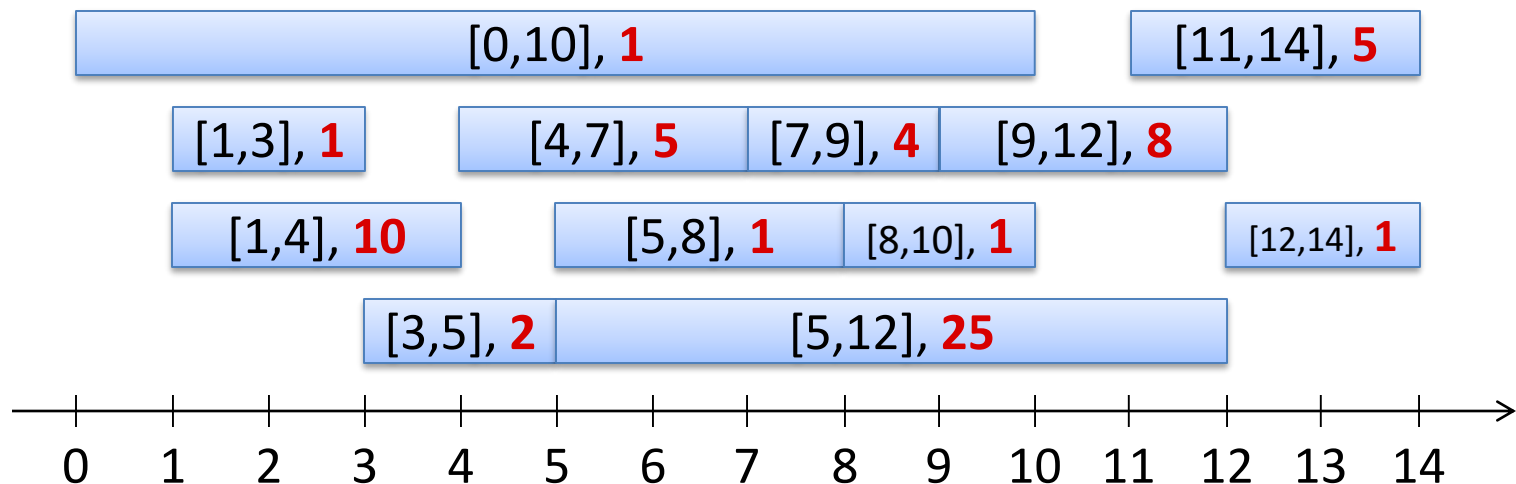
# **Dynamic Programming**

**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  $w$**



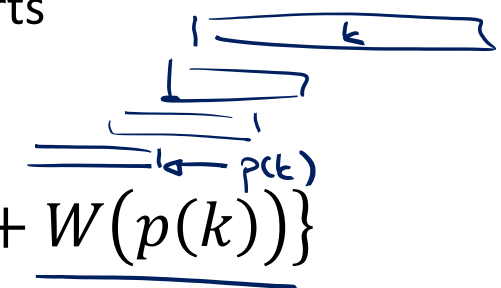
- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e.,  $[4,7]$  and  $[7,9]$  are non-overlapping
- **Example:** Intervals are room requests of different importance

# 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:

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

$$\underline{W(1)} = \underline{w(1)} \quad \text{with } w(0) = 0$$


Immediately gives a simple, recursive algorithm

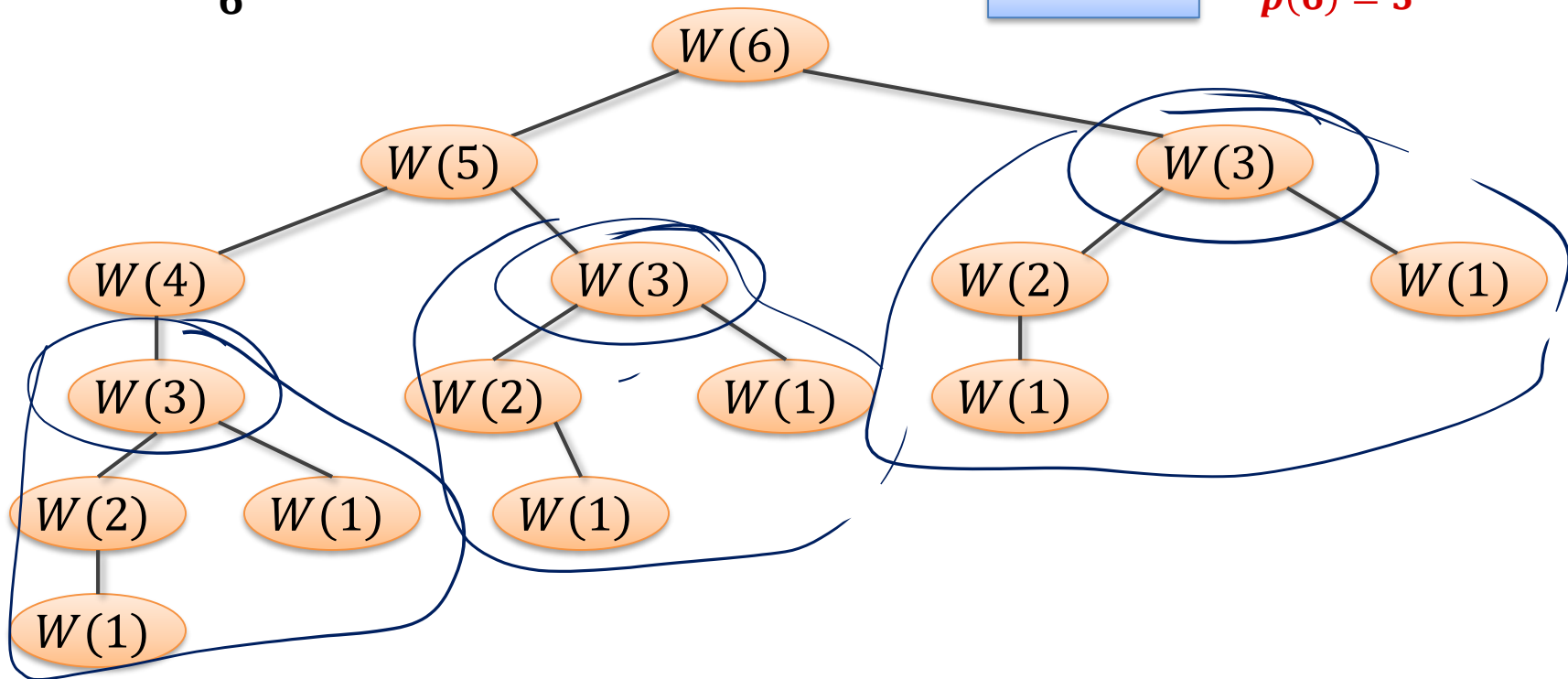
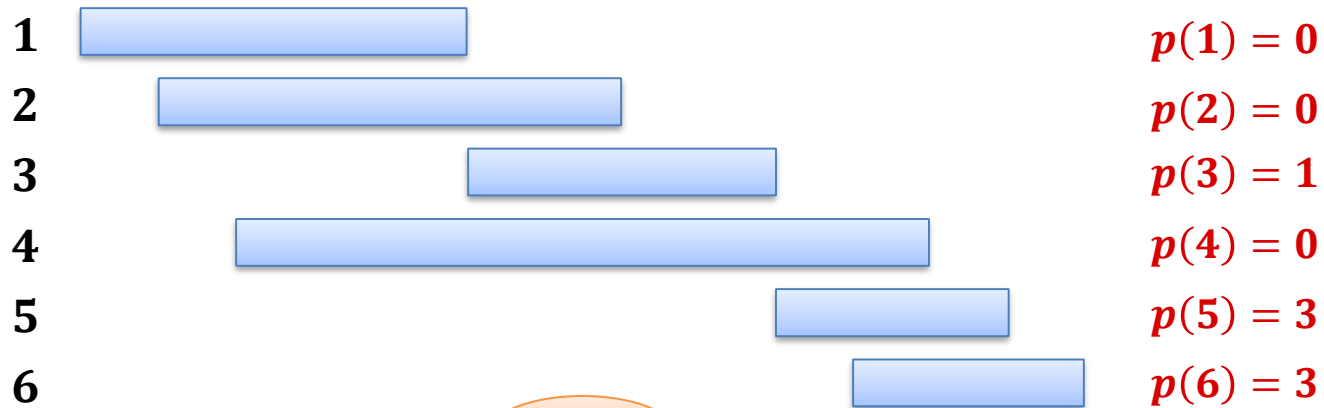
Compute  $p(k)$  values for all  $k$

$W(k)$ :

```

if k == 0:
    x = w(1) 0
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  $k$

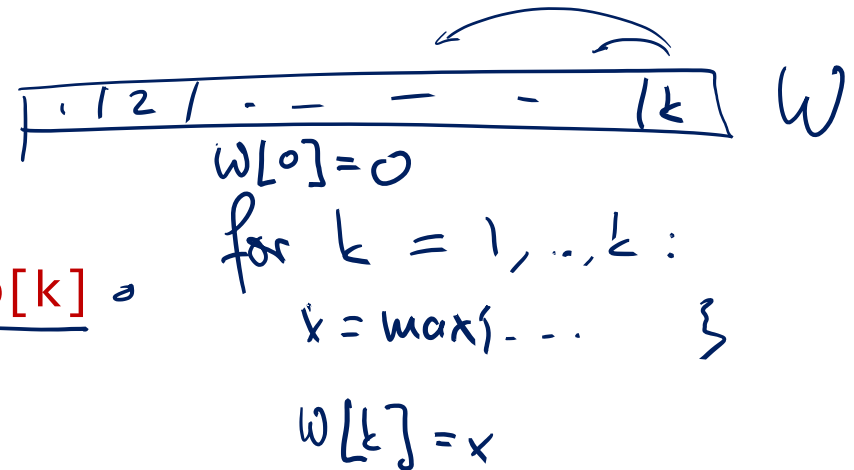
$\text{memo} = \{\};$

$W(k)$ :

```

    if  $k$  in  $\text{memo}$ : return  $\text{memo}[k]$ 
    if  $k == 1$ :
         $x = w(1)$ 
    else:
         $x = \max\{W(k-1), w(k) + W(p(k))\}$ 
     $\text{memo}[k] = x$ 
    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

$W(L)$

**Weighted interval scheduling:** subproblems  $W(1), W(2), W(3), \dots$

**runtime = #subproblems · time per subproblem**

# DP: Some History ...

- Where 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. ..."*





# Matrix-chain multiplication $((\ )(\ )) \times (\ \ \ \ )(\ )$

**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  $\langle A_1, A_2, A_3, A_4 \rangle$ :

$$(A_1 (A_2 (A_3 A_4)))$$

$$(A_1 ((A_2 A_3) A_4))$$

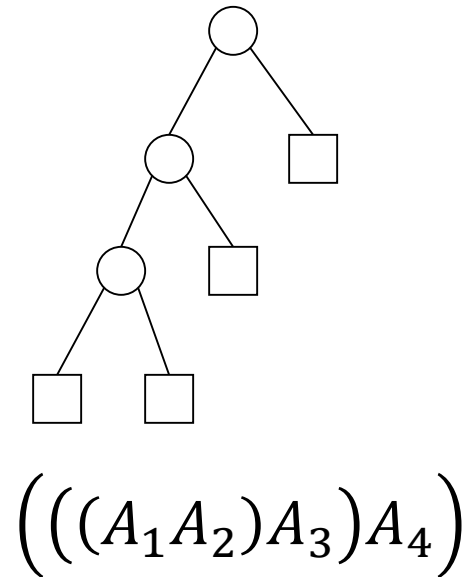
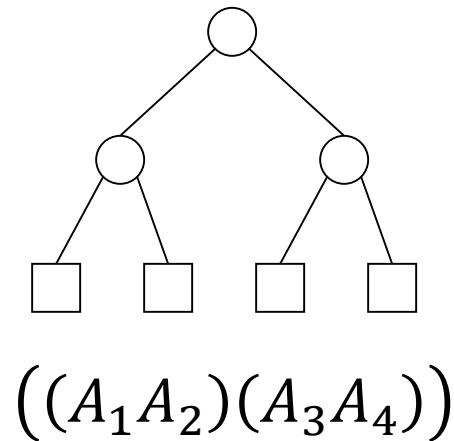
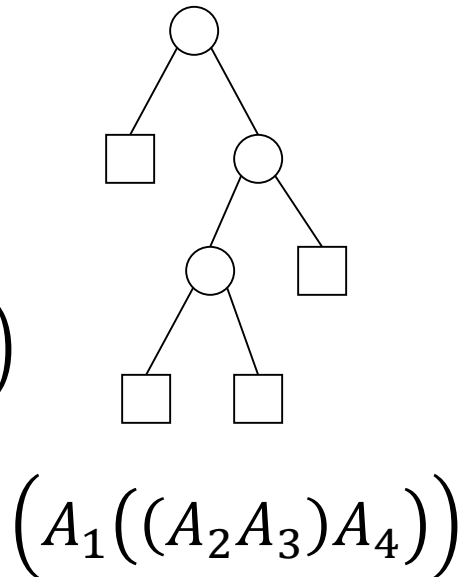
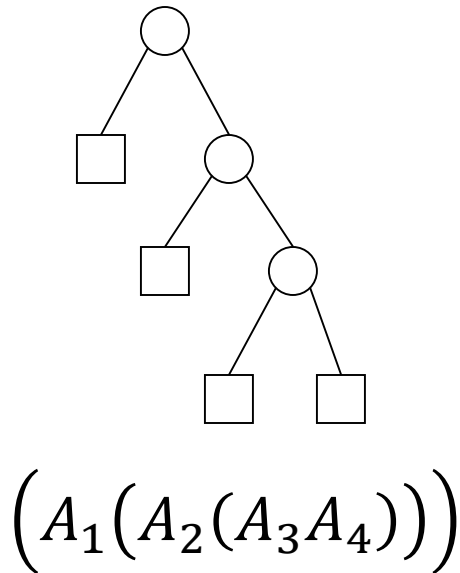
$$((A_1 A_2)(A_3 A_4))$$

$$((A_1 (A_2 A_3)) A_4)$$

$$(((A_1 A_2) A_3) A_4)$$

# Different parenthesizations

Different parenthesizations correspond to different trees:



# Number of different parenthesizations

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

$$P(1) = 1$$

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

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

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

- Thus: Exhaustive search needs exponential time!

# Multiplying Two Matrices

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

$$p \cdot \left( \overbrace{\quad}^q \right) \cdot \left( \underbrace{\quad}_r \right) = \left( \cdot \right) c_{ij} = \sum_{k=1}^q a_{ik} b_{kj} \quad \left( \begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array} \right) \cdot \left( \begin{array}{c} + \\ + \\ + \end{array} \right)$$

## Algorithm Matrix-Mult

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

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

```

1 for  $i := 1$  to  $p$  do
2   for  $j := 1$  to  $r$  do
3      $C[i, j] := 0$ ;
4     for  $k := 1$  to  $q$  do
5        $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$ 

```

## Remark:

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

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

# 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 ( $A_1$  $(A_2 A_3)$ ) require:

$50 \times 100$

$$A' = (A_1 A_2): 50 \cdot 5 \cdot 100 = 25'000$$

$5 \times 10$

$$A'' = (A_2 A_3): 5 \cdot 100 \cdot 10 = 5'000$$

$$A' A_3: 50 \cdot 100 \cdot 10 = 50'000$$

$$A_1 A'': 50 \cdot 5 \cdot 10 = 2'500$$

---

Sum: 75'000

7'500

# Structure of an Optimal Parenthesization

- $(A_{\ell \dots r})$ : optimal parenthesization of  $A_{\ell} \cdot \dots \cdot A_r$   $O(n^2)$  subproblems  
 $\uparrow C(\ell, r)$

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

- 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  $A_{\ell} \cdot \dots \cdot A_r$ ,  $\ell \leq r$  optimally:  $C(\ell, r)$
- Then:

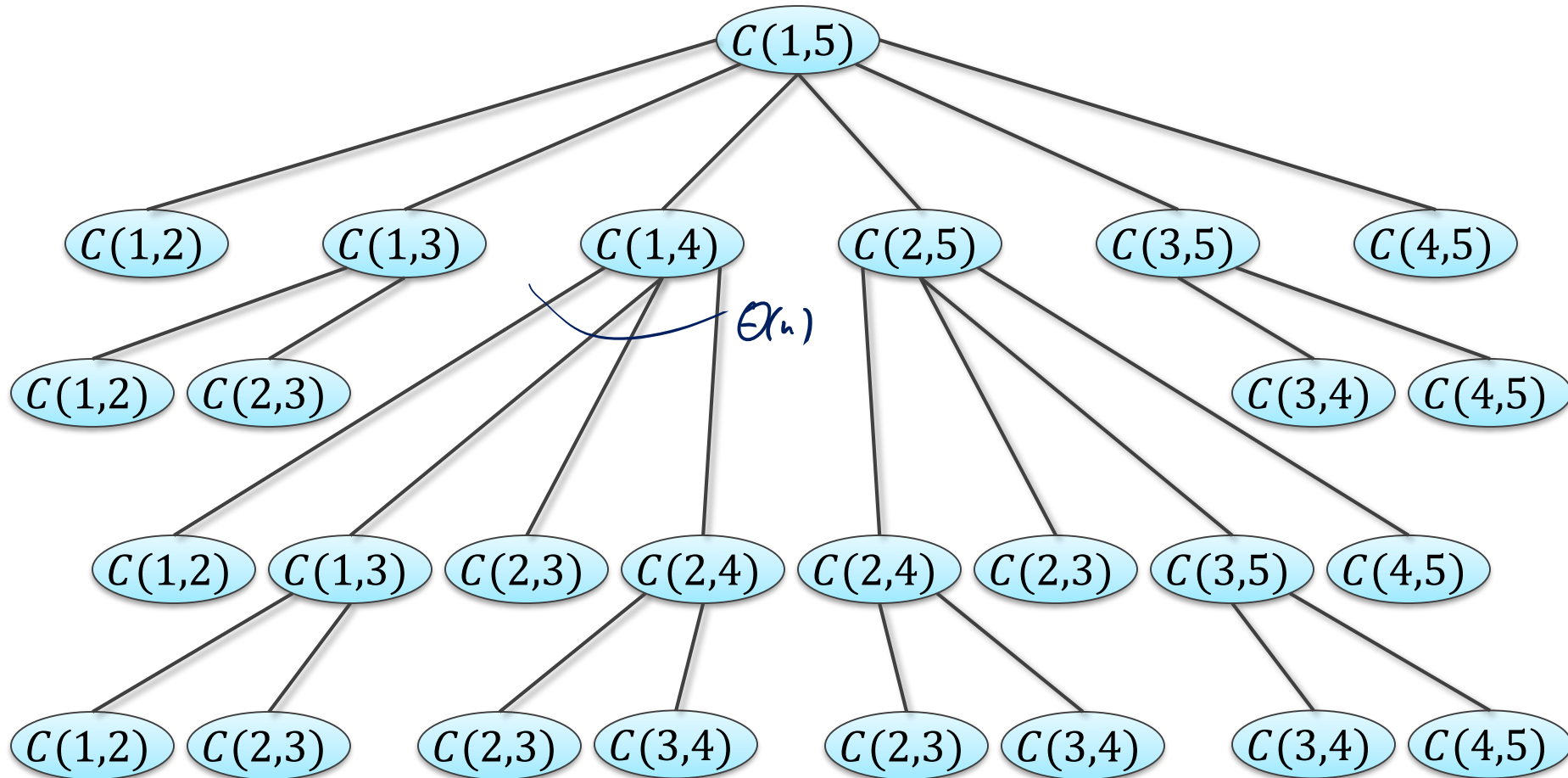
$$C(a, b) = \min_{a \leq k < b} C(a, k) + C(k+1, b) + \underbrace{d_{a-1} d_k d_b}_{\text{cost of "last" mult.}}$$

$$C(a, a) = 0$$

$$C(1, n)$$

# Recursive Computation of Opt. Solution

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

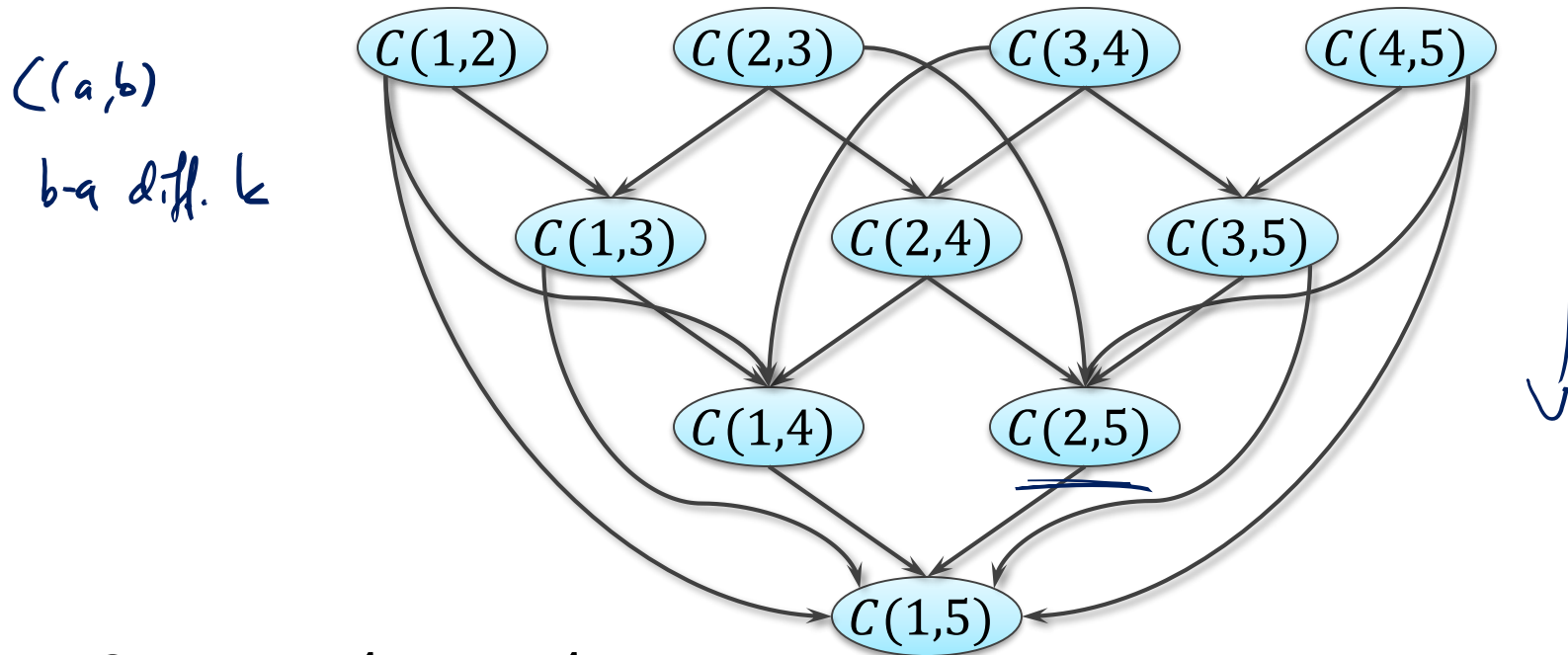




# Using Memoization

#subproblems · time per subpr.  
 $O(n^2) \cdot O(n) = O(n^3)$

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



Compute  $A_1 \cdot \dots \cdot A_n$ :

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

Cost for each  $C(i,j)$ :  $O(n) \rightarrow$  overall time:  $O(n^3)$

# Remarks about matrix-chain multiplication

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

$$\underline{O(n \cdot \log n)}.$$

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

$$\underline{1.155 \cdot C(1, n)}$$

multiplications.

„*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  $1, \dots, n$ , 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**:

$$\left[ \begin{array}{l} \max \sum_{i \in S} v_i \\ \text{s. t. } S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W \end{array} \right.$$

*feasible*

- 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?

OPT(k)



- Optimal solution:  $\mathcal{O}$
  - If  $n \notin \mathcal{O}$ : OPT(n) = 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!
- OPT(n) =  <sup>$v_n$</sup>  ~~$w_n$~~  + optimal solution with first  $n - 1$  items and knapsack of capacity  $W - w_n$

# A More Complicated Recursion

$\text{OPT}(k, x)$ : value of **optimal solution** with items 1, ..., k  
and knapsack of capacity x

Recursion:

$$\text{OPT}(k, x) = \max \left\{ \underbrace{\text{OPT}(k-1, x)}_{\text{opt. sol. if we don't take item } k}, \quad v_k + \underbrace{\text{OPT}(k-1, x - w_k)}_{\substack{\text{rem. cap. after} \\ \text{taking item } k}} \right\}$$

*Handwritten notes:*  
1, ..., W  
only makes sense if  $x - w_k \geq 0$

initialization

$$\text{OPT}(0, x) = 0$$

$$\text{OPT}(k, 0) = 0$$

# Subproblems?

# diff. capacities  $\leq 2^n$  (too much)

assumption: weights are integers

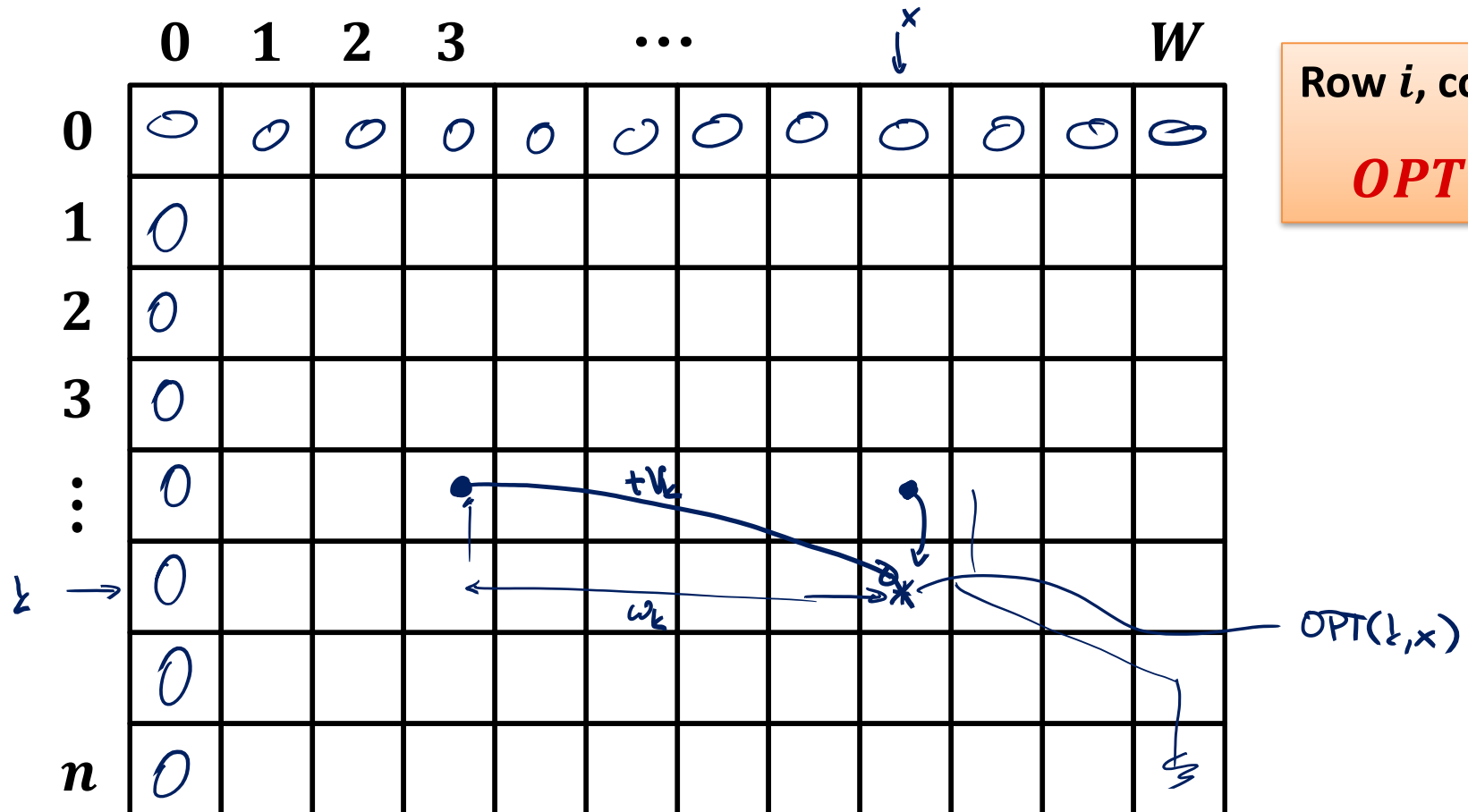
$\rightarrow W+1$  options for 2nd parameter

# subpr. :  $O(nW)$

# Dynamic Programming Algorithm

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

- Assume that all weights  $w_i$  are integers!





# Example

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

weight value

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

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	2	2	2	2	2	2	2	2	2	2
2	0	4	4	4	6	6	-	-		-		
3												
4												
5												
6												
7												
8												

$O(nw)$

# Running Time of Knapsack Algorithm

- **Size of table:**  $O(n \cdot W)$
- Time per table entry:  $O(1)$  → **overall time:**  $O(nW)$
- 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.

weights are rational numbers possible  
another special case : values are integers

- problem is NP-hard generally
- but, we can compute a solution which is arbitrarily close to opt. in poly. time

$1 + \epsilon$

$OPT(k, y)$

# 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  $\rightarrow$  multiplication:

m u t i p l a t i o ~~i~~ ~~a~~ n

          └─┬─┘           └─┬─┘

          l               i c

                                  10 ops.

# Edit Distance

**Given:** Two strings  $A = a_1 a_2 \dots a_{\underline{m}}$  and  $B = b_1 b_2 \dots b_{\underline{n}}$

**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	<u>i</u>	a	n
m	u	<u>l</u>	t	i	p	l	i	c	a	t	i	o	<u>,</u>	-	n

# Edit Distance – Cost Model

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

- **Triangle inequality:**

$$\underline{c(a, c) \leq c(a, b) + c(b, c)}$$

→ each character is changed at most once!

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

# 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	and	-gik-ccm
abb-adfl		rgikacc-

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

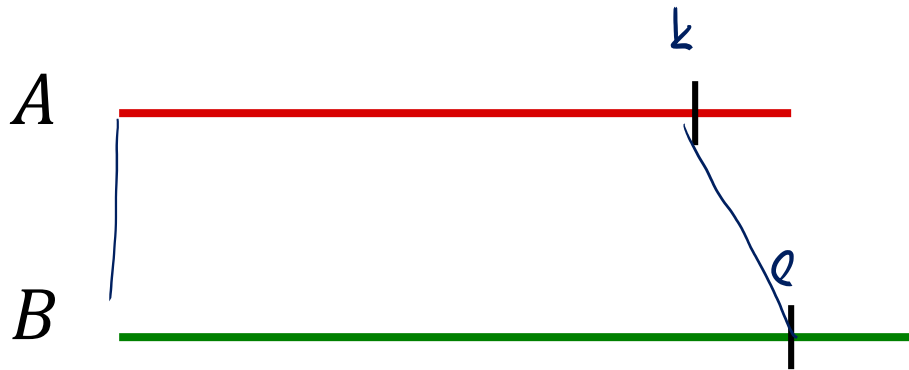
$$D(A, B) = \min_{\substack{k, \ell}} \{ D(A_{1,k}, B_{1,\ell}) + D(A_{k+1,m}, B_{\ell+1,n}) \}$$

# Computation of the Edit Distance

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

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

$\mathcal{D}_{n,n}$

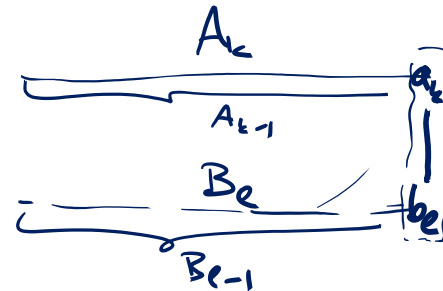


# Computation of the Edit Distance

Three ways of ending an “alignment” between  $A_k$  and  $B_\ell$ :

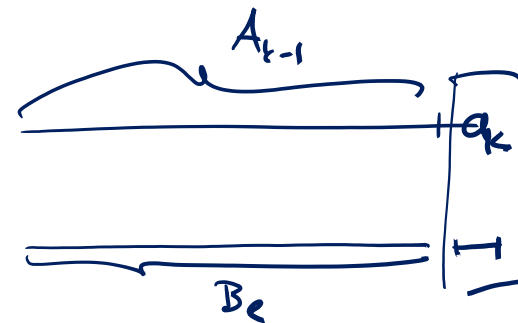
1.  $a_k$  is replaced by  $b_\ell$ :

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



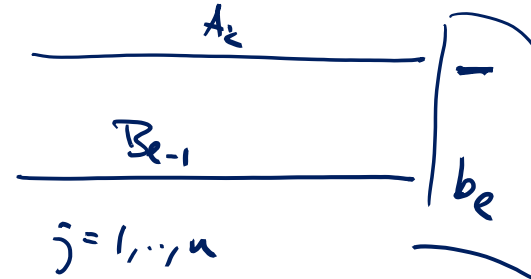
2.  $a_k$  is deleted:

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



3.  $b_\ell$  is inserted:

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



$D_{i,j}$

$i=1, \dots, m \quad j=1, \dots, n$

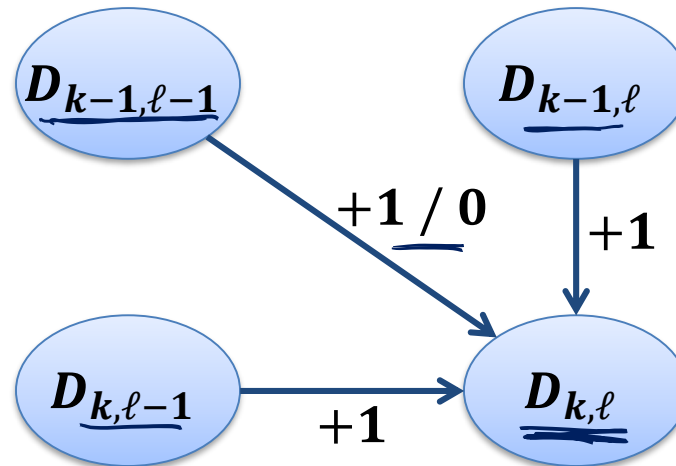


# Computing the Edit Distance

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

$$D_{k,\ell} = \min \left\{ \begin{array}{l} D_{k-1,\ell-1} + c(a_k, b_\ell) \\ D_{k-1,\ell} + c(a_k, \varepsilon) \\ D_{k,\ell-1} + c(\varepsilon, b_\ell) \end{array} \right\} = \min \underbrace{\left\{ \begin{array}{l} D_{k-1,\ell-1} + 1 / 0 \\ D_{k-1,\ell} + 1 \\ D_{k,\ell-1} + 1 \end{array} \right\}}_{\text{unit cost model}}$$

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



Base cases:

$$D_{0,0} = D(\varepsilon, \varepsilon) = \underline{0}$$

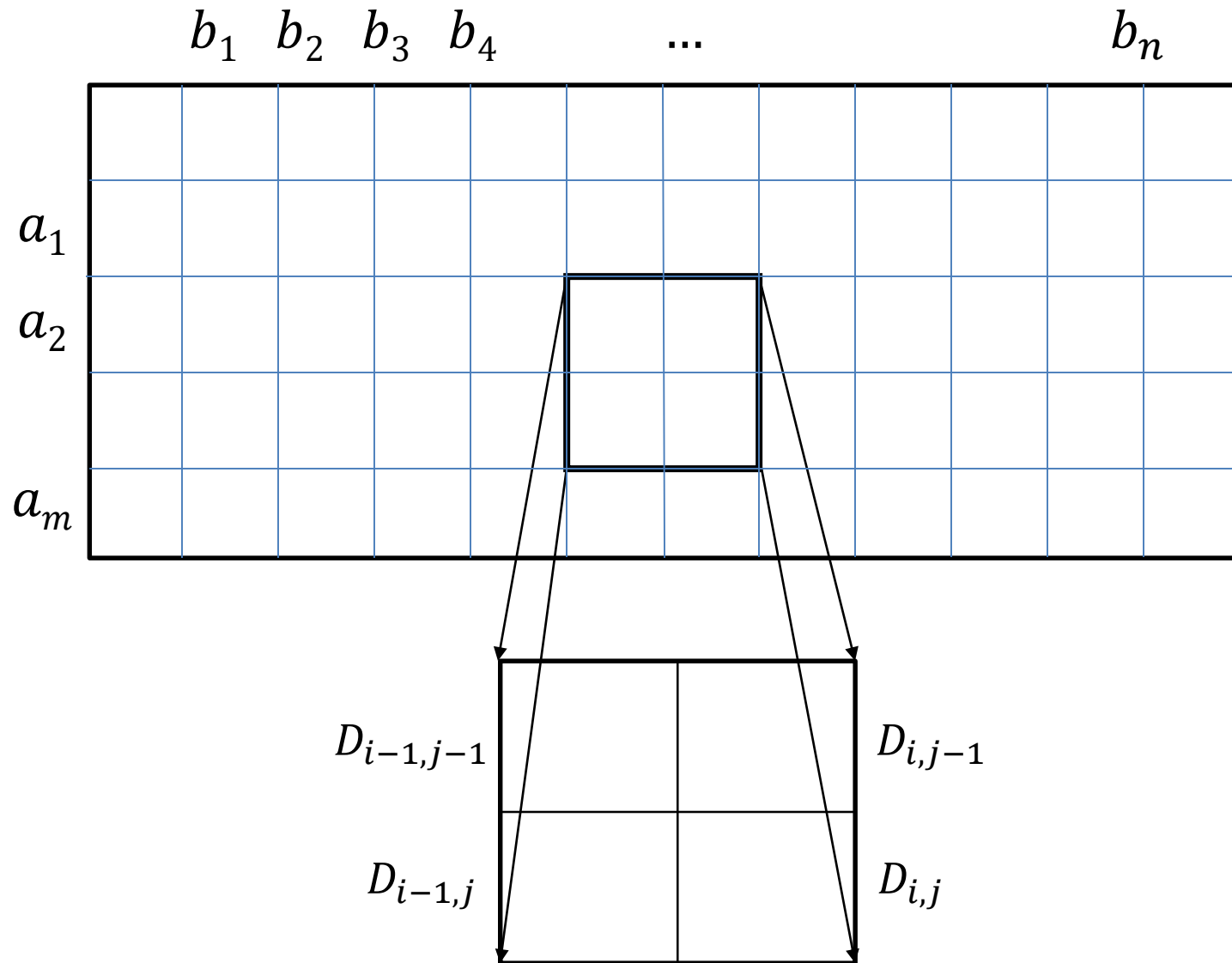
$$\underline{D}_{0,j} = D(\varepsilon, B_j) = \underline{D}_{0,j-1} + c(\varepsilon, b_j)$$

$$\underline{D}_{i,0} = D(A_i, \varepsilon) = \underline{D}_{i-1,0} + c(a_i, \varepsilon)$$

Recurrence relation:

$$D_{i,j} = \min \left\{ \begin{array}{ll} D_{k-1,\ell-1} + c(a_k, b_\ell) \\ D_{k-1,\ell} + c(a_k, \varepsilon) \\ D_{k,\ell-1} + c(\varepsilon, b_\ell) \end{array} \right\}$$

# 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  $D = (D_{ij})$

1  $D[0,0] := 0;$

2 **for**  $i := 1$  **to**  $m$  **do**  $D[i, 0] := i;$

3 **for**  $j := 1$  **to**  $n$  **do**  $D[0, j] := j;$

4 **for**  $i := 1$  **to**  $m$  **do**

5   **for**  $j := 1$  **to**  $n$  **do**

6      $D[i, j] := \min \left\{ \begin{array}{l} D[i-1, j] + 1 \\ D[i, j-1] + 1 \\ D[i-1, j-1] + c(a_i, b_j) \end{array} \right\};$

# Example

unit cost model

		<i>a</i>	<i>b</i>	<i>c</i>	<i>c</i>	<i>a</i>
	0	1	2	3	4	5
<i>b</i>	1	1	1	2	3	4
<i>a</i>	2	1				
<i>b</i>	3					
<i>d</i>	4					
<i>a</i>	5					

# Computing the Edit Operations

**Algorithm** *Edit-Operations*( $i, j$ )

**Input:** matrix  $D$  (already computed)

**Output:** list of edit operations

- 1 **if**  $i = 0$  **and**  $j = 0$  **then return** empty list
- 2 **if**  $i \neq 0$  **and**  $D[i, j] = D[i - 1, j] + 1$  **then**
- 3     **return** *Edit-Operations*( $i - 1, j$ )  $\circ$  „delete  $a_i$ “
- 4 **else if**  $j \neq 0$  **and**  $D[i, j] = D[i, j - 1] + 1$  **then**
- 5     **return** *Edit-Operations*( $i, j - 1$ )  $\circ$  „insert  $b_j$ “
- 6 **else** //  $D[i, j] = D[i - 1, j - 1] + c(a_i, b_j)$
- 7     **if**  $a_i = b_i$  **then return** *Edit-Operations*( $i - 1, j - 1$ )
- 8     **else return** *Edit-Operations*( $i - 1, j - 1$ )  $\circ$  „replace  $a_i$  by  $b_j$ “

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

# Edit Operations

$a \rightarrow b$   $d \rightarrow c$   $d \rightarrow a$

		<i>a</i>	<i>b</i>	<i>c</i>	<i>c</i>	<i>a</i>
	0	1	2	3	4	5
<i>b</i>	1	1	1	2	3	4
<i>a</i>	2	1	2	2	3	3
<i>b</i>	3	2	1	2	3	4
<i>d</i>	4	3	2	2	3	4
<i>a</i>	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

