Chapter 3
Dynamic Programming

Algorithm Theory
WS 2017/18

Fabian Kuhn

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 $W(1), W(2), W(3), \ldots$

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

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

Matrix-chain multiplication

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

$(A_1 A_2) \cdot (A_3 A_4)$

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

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

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

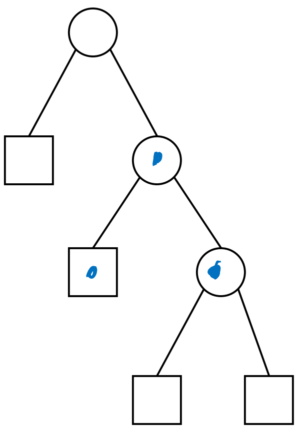
$$((A_1A_2)(A_3A_4))$$

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

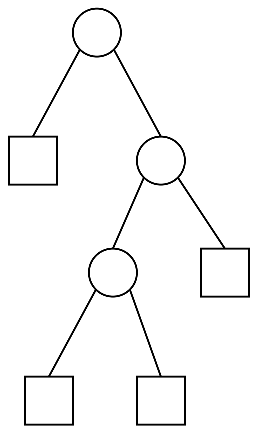
$$(((A_1A_2)A_3)A_4)$$

Different parenthesizations

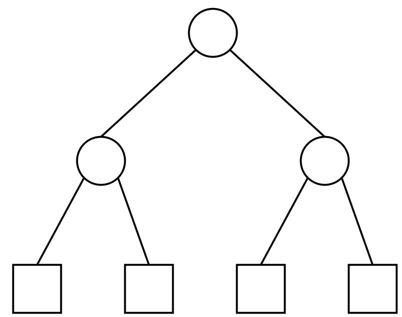
Different parenthesizations correspond to different trees:



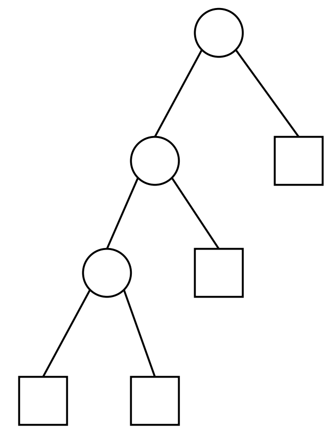
$$\left(A_1\left(A_2(A_3A_4)\right)\right)$$



$$\left(A_1\left((A_2A_3)A_4\right)\right)$$



$$\left((A_1A_2)(A_3A_4)\right)$$



$$\left(\left((A_1A_2)A_3\right)A_4\right)$$

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

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

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

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \left(\begin{array}{c|c} & \\ \hline & \end{array}\right)$$

Algorithm *Matrix-Mult*

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

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

```
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 $O(n^{2.373})$ 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 $\big(A_1(A_2A_3)\big)$ require:

50×100

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

5×10

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

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

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

Sum: $75'000$ $7'500$

Structure of an Optimal Parenthesization

- $(A_{\ell\ldots r})$: optimal parenthesization of $A_\ell \cdot \ldots \cdot A_r$ #subproblems: $\Theta(n^2)$

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

- Any optimal solution contains optimal solutions for sub-problems

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

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

- Then: $a < b$

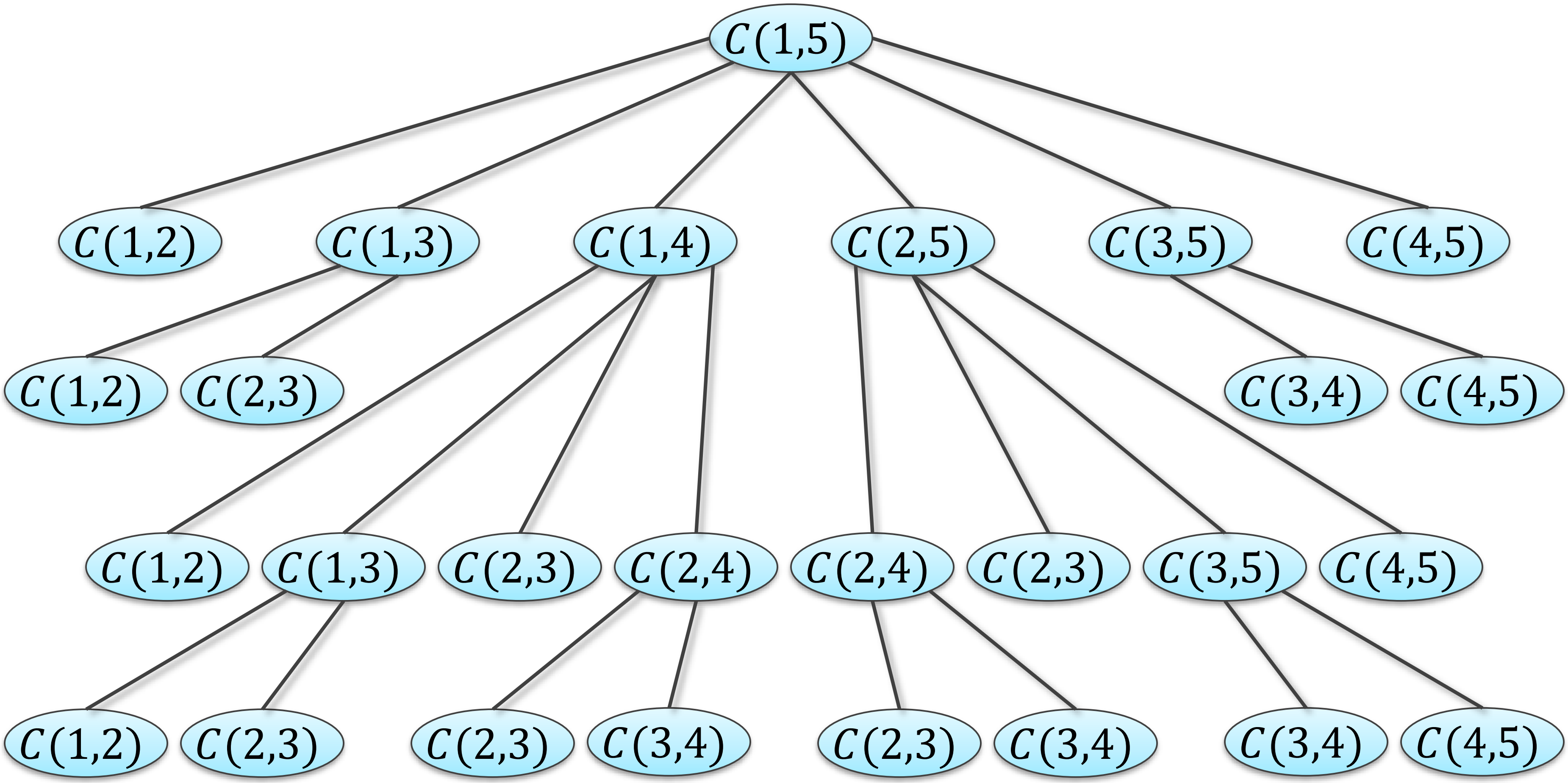
$$\boldsymbol{C(a,b) = \min_{a \leq k < b} C(a,k) + C(k+1,b) + \underbrace{d_{a-1} d_k\, d_b}_{\text{cost of last mult.}}}$$

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

$C(1,n)$?

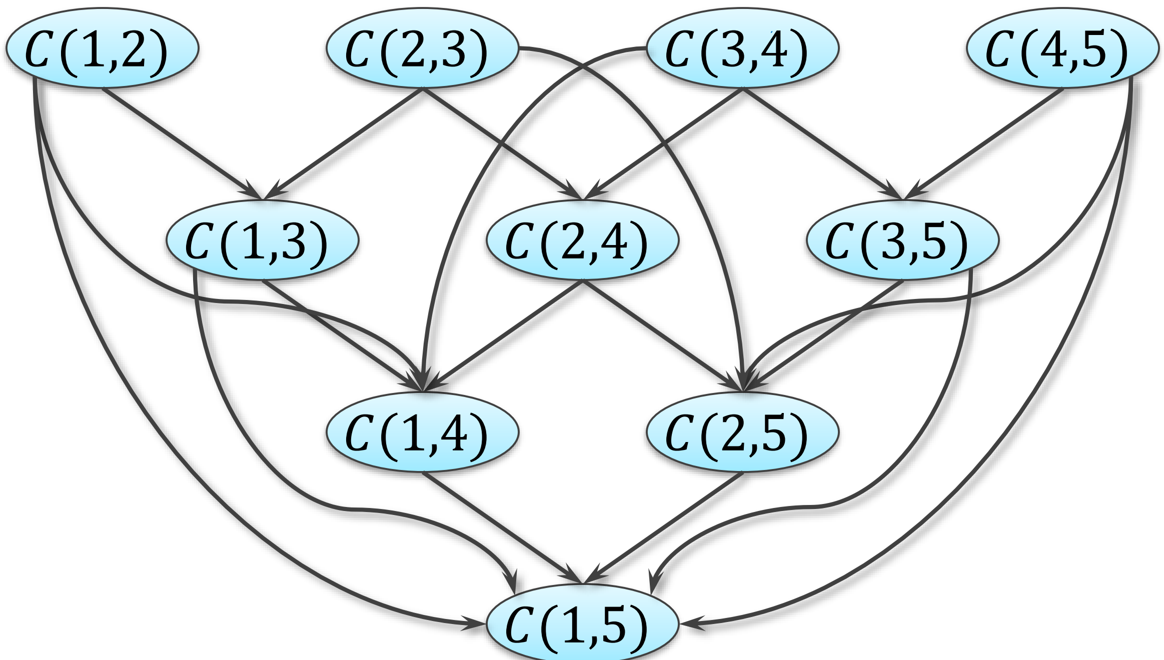
Recursive Computation of Opt. Solution

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



Using 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.

Knapsack

NP-hard

- 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:

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

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

- 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(n)

- 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!

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

A More Complicated Recursion

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

$\mathrm{OPT}(n, W)$

Recursion:

only makes sense if ≥ 0

$$\mathrm{OPT}(k,x) = \max\{\underbrace{\mathrm{OPT}(k-1,x)}_{\text{opt. sol. when not using item } k},\ v_k + \mathrm{OPT}(k-1, \underbrace{x - w_k}_{\text{remaining cap.}})\}$$

Initialization

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

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

\# subproblems?

arbitrary weights $\approx 2^n$

integer weights : $n \cdot W$

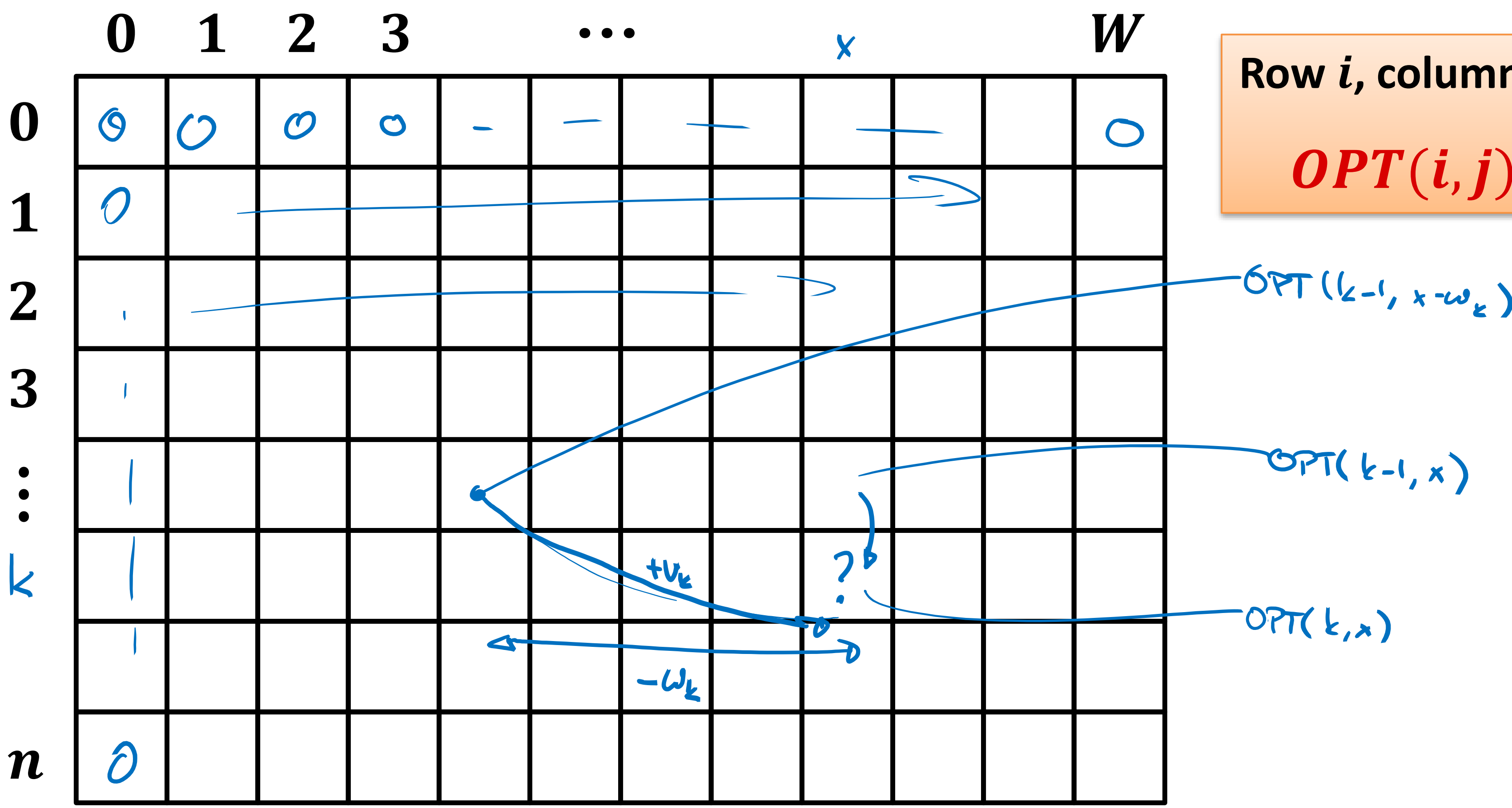
assume that weights are integers

$\Longrightarrow$ time: $O(n \cdot W)$

Dynamic Programming Algorithm

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

- Assume that all weights w_i are integers!



Row i, column j:

$OPT(i, j)$

Example

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12 **weight** **value**
- $OPT(k, x) = \max\{OPT(k-1, x), OPT(k-1, x - w_k) + v_k\}$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	2	2	2	–	–	–	–	–	–	2
2	0	4	4	4	6	6	–	–	–	–		6
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: $O(nW)$**
- 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.

still possible if weights are rational

another special case : values are integers

$\boxed{OPT(k, v)}$

- general case : NP-hard

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:

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

l　　　　i c

10 ops

Edit Distance

Given: Two strings $A = a_1 a_2 \dots a_m$ 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
```

alignment

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: $\boldsymbol{c(\varepsilon,b)}$

$c(a,a) = 0$

- **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
abb-adfl   and  rgikacc-
```

$A_{i,j}$

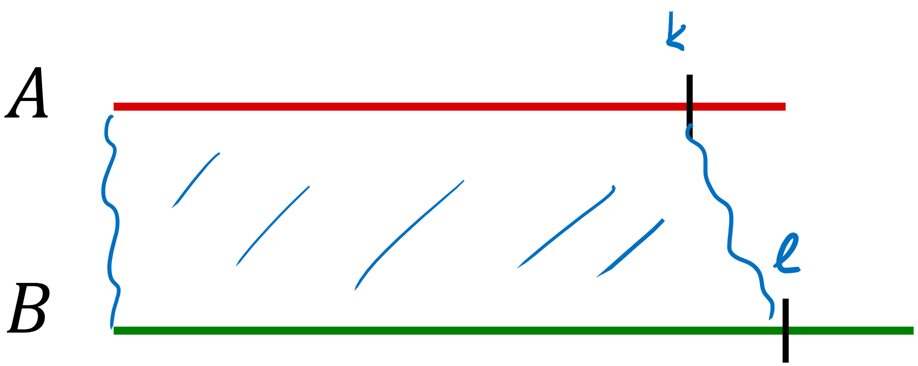
- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

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

Computation of the Edit Distance $A_{i,j}$ $A_{i,j} = A_i$

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

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

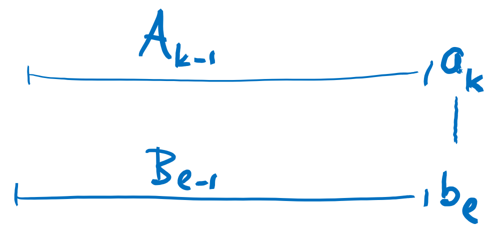


Computation of the Edit Distance $D_{k,\ell}$

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

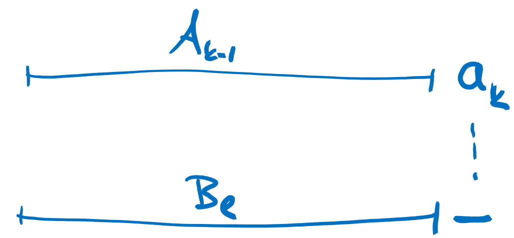
1. a_k is replaced by b_ℓ:

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



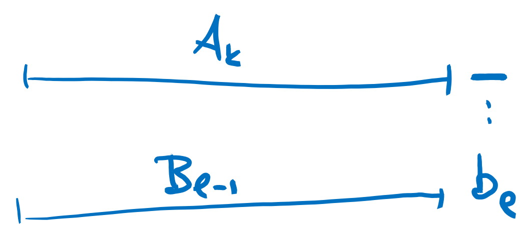
2. a_k is deleted:

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



3. b_ℓ is inserted:

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



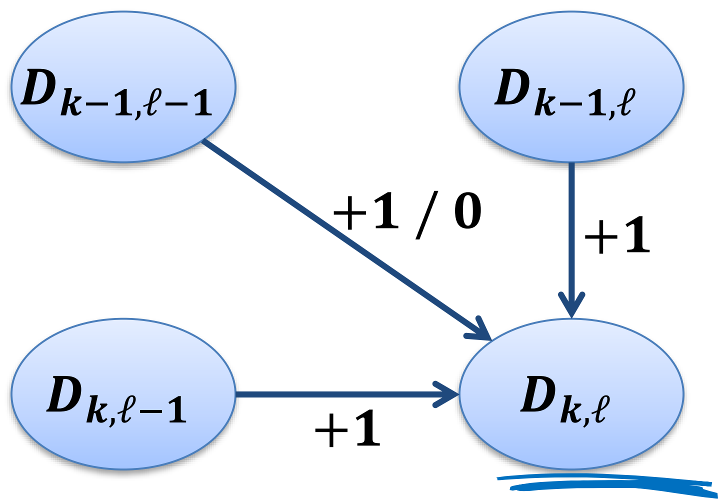
Computing the Edit Distance

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

if $a_k = b_\ell$

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

- 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:

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

\# subproblems: $m \cdot n$

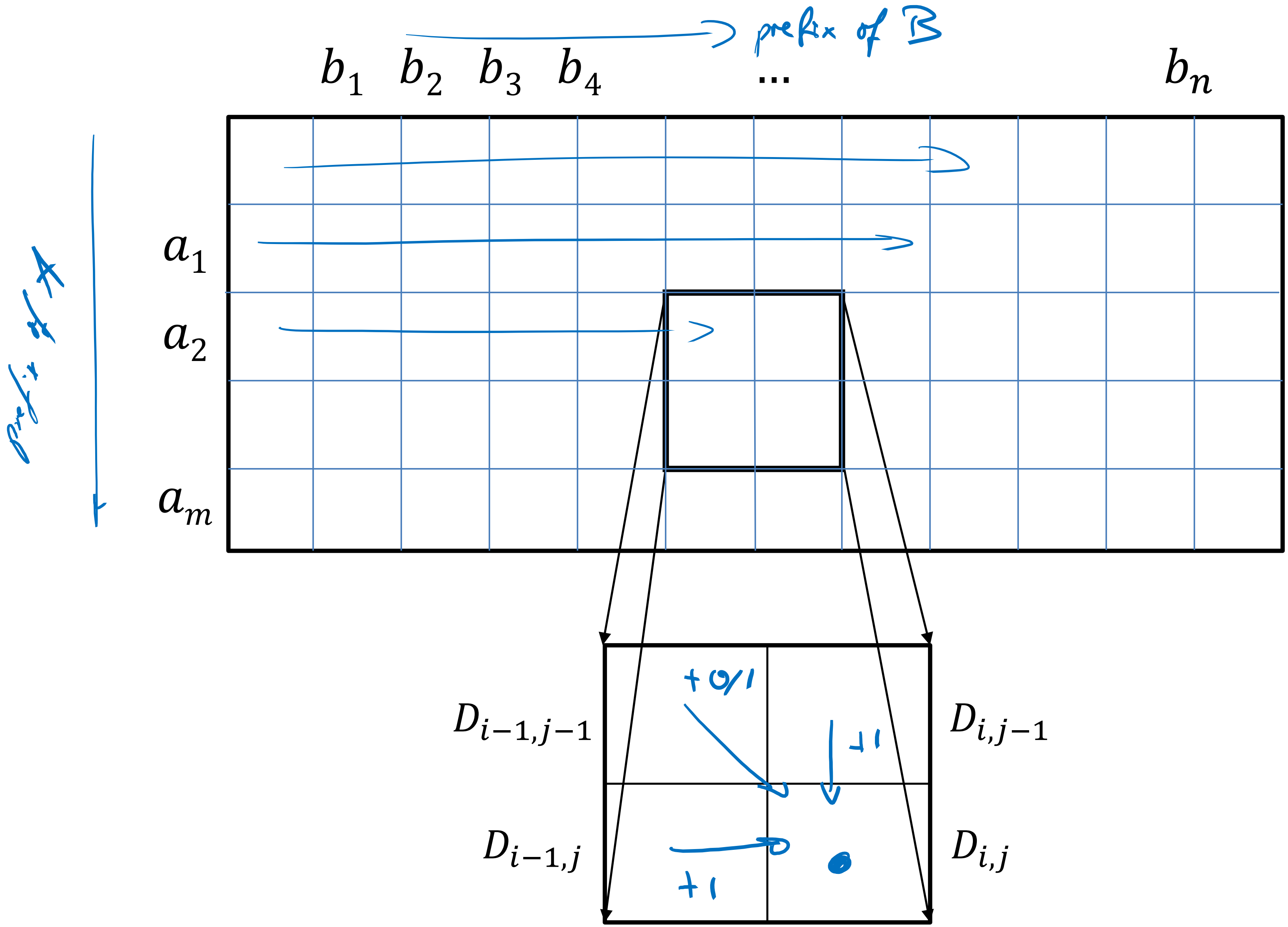
time per subpr.: $O(1)$

overall time: $O(m \cdot n)$

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**

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

Example

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

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 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

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 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"

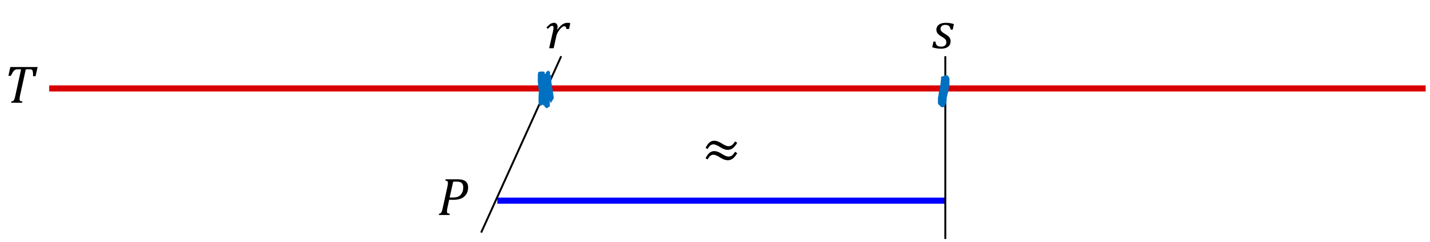
- Unit cost model:
  – interesting special case
  – each edit operation costs 1

Approximate String Matching

Given: strings $T = t_1 t_2 \dots t_n$ (text) and $P = p_1 p_2 \dots p_m$ (pattern).

Goal: Find an interval $[r, s]$, $1 \leq r \leq s \leq n$ such that the sub-string $T_{r,s} := t_r \dots t_s$ is the one with highest similarity to the pattern P:

$$\arg\min_{1 \leq r \leq s \leq n} D\left(T_{r,s}, P\right)$$



Approximate String Matching

Naive Solution:

for all $1 \le r \le s \le n$ **do**

compute $D(T_{r,s}, P)$

choose the minimum

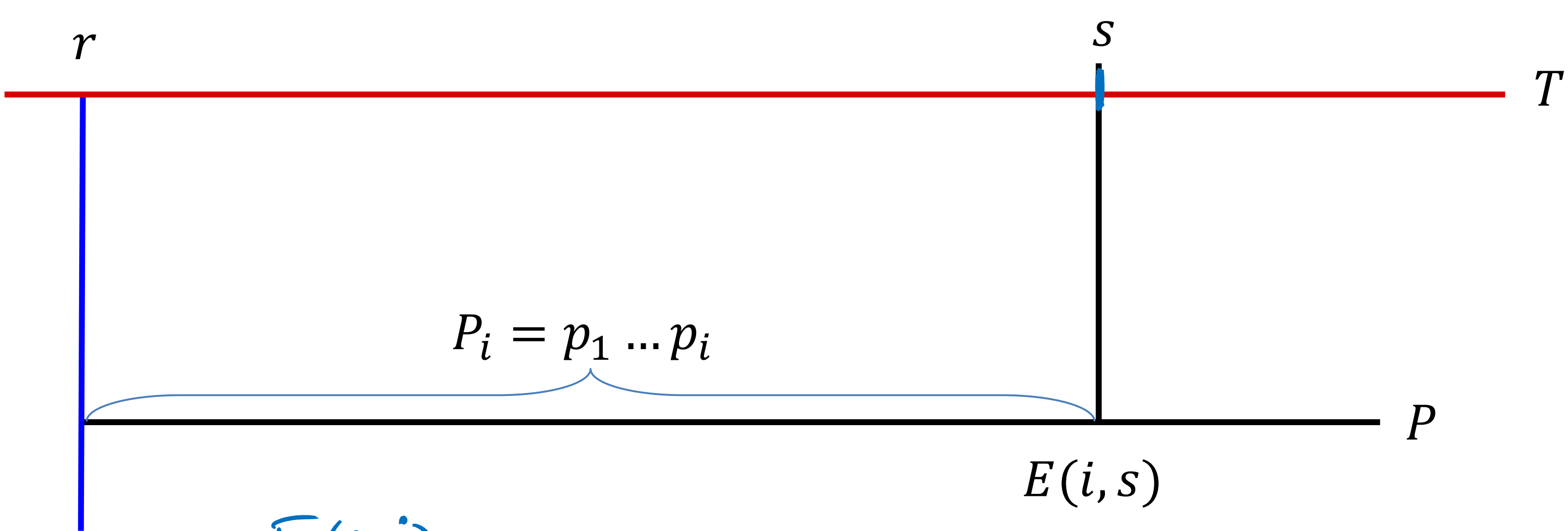
Overall: $O(m \cdot n^3)$

cost $((s-r) \cdot m) = O(m \cdot n)$

Approximate String Matching

A related problem:

- For each position s in the text and each position i in the pattern compute the minimum edit distance $E(i, s)$ between $P_i = p_1 \dots p_i$ and any substring $T_{r,s}$ of T that ends at position s.



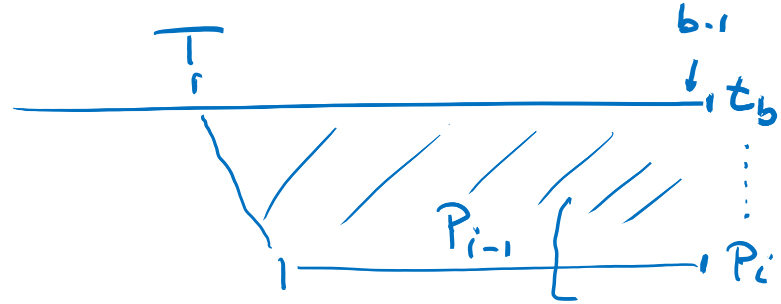
$$E(i, s) = \min_r D(T_{r,s}, P_i)$$

Approximate String Matching

Three ways of ending optimal alignment between T_b and P_i:

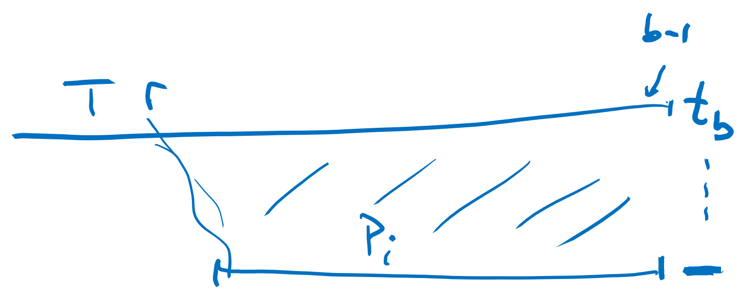
1. t_b is replaced by p_i:

   $$E_{b,i} = E_{b-1,i-1} + c(t_b, p_i)$$



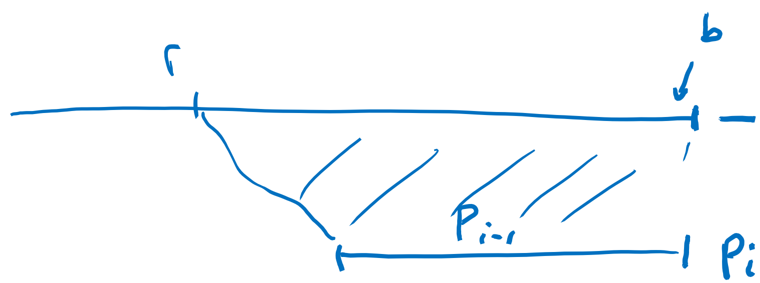
2. t_b is deleted:

   $$E_{b,i} = E_{b-1,i} + c(t_b, \varepsilon)$$



3. p_i is inserted:

   $$E_{b,i} = E_{b,i-1} + c(\varepsilon, p_i)$$



Approximate String Matching

Recurrence relation (unit cost model):

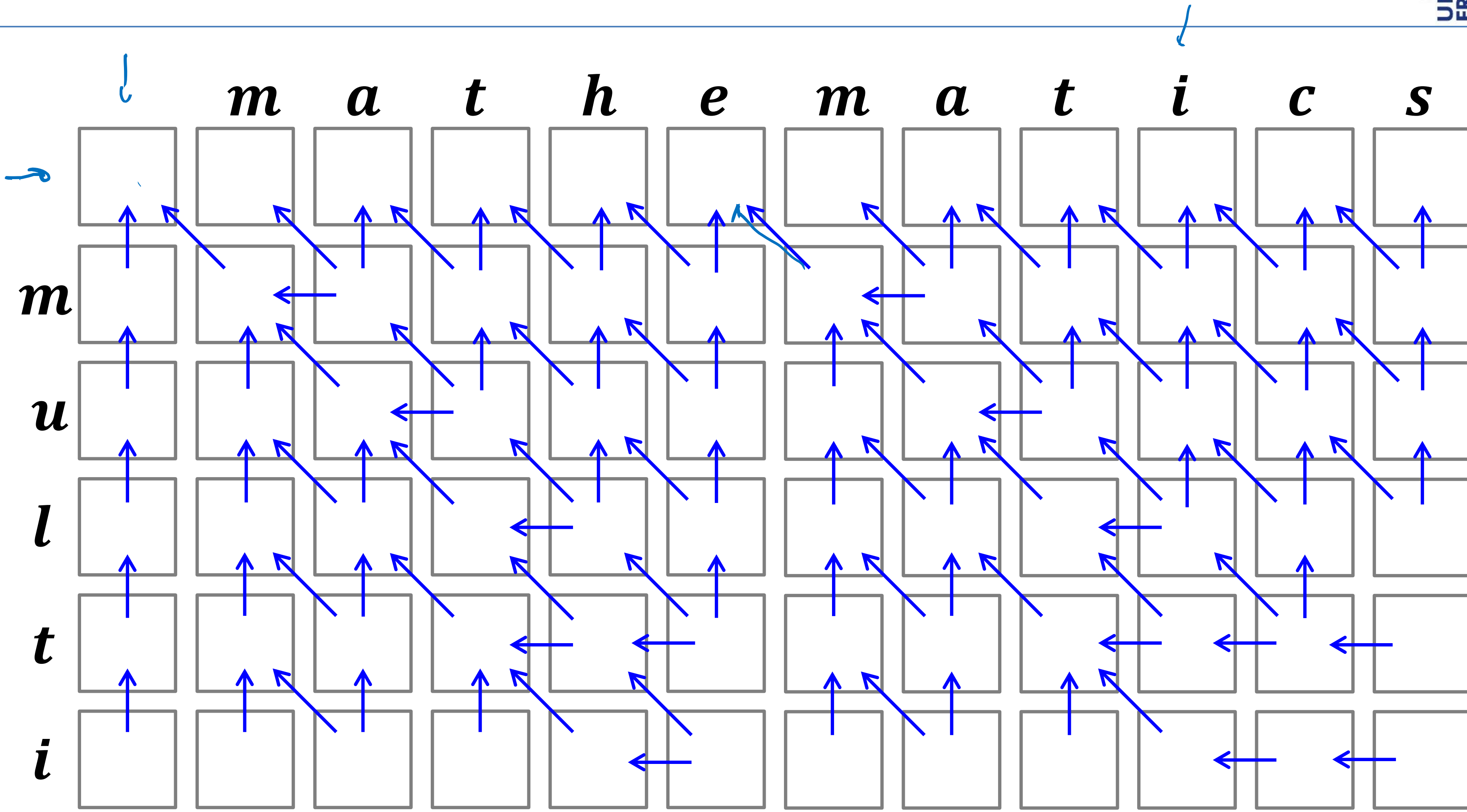
$/0 \leftarrow$ if $t_b = p_i$

$$E_{b,i} = \min \begin{Bmatrix} E_{b-1,i-1} & +1 \\ E_{b-1,i} & +1 \\ E_{b,i-1} & +1 \end{Bmatrix}$$

Base cases:

$$E_{0,0} = 0$$
$$E_{0,i} = i$$
$$\color{red}{E_{i,0} = 0}$$

Example



– e m a – t i .. c s
m u l t i

Approximate String Matching

- Optimal matching consists of optimal sub-matchings
- Optimal matching can be computed in $O(mn)$ time
- Get matching(s):
  - Start from minimum entry/entries in bottom row
  - Follow path(s) to top row
- Algorithm to compute $E(b, i)$ identical to edit distance algorithm, except for the initialization of $E(b, 0)$

Related Problems in Bioinformatics

Sequence Alignment:

Find optimal alignment of two given DNA, RNA, or amino acid sequences.

```
G A - C G G A T T A G
G A T C G G A A T - G
```

Global vs. Local Alignment:

- *Global alignment*: find optimal alignment of 2 sequences
- *Local alignment*: find optimal alignment of sequence 1 (pattern) with sub-sequence of sequence 2 (text)