
Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren



Danksagung

- Das folgende Präsentationsmaterial wurde von Sven Groppe für das Modul Algorithmen und Datenstrukturen erstellt und mit Änderungen hier übernommen (z.B. werden Algorithmen im Pseudocode präsentiert)

Motivation Zeichenkettenabgleich

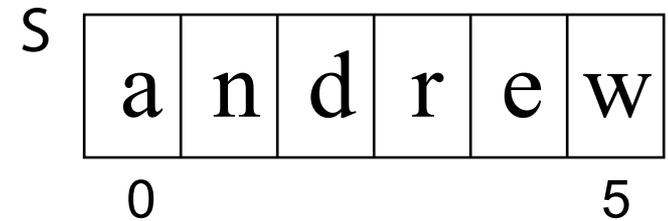
- Gegeben eine Folge von Zeichen (Text), in der eine Zeichenkette (Muster) gefunden werden soll
- Varianten
 - *Alle* Vorkommen des Musters im Text
 - Ein *beliebiges* Vorkommen im Text
 - *Erstes* Vorkommen im Text
- Anwendungen
 - Suchen von Mustern in DNA-Sequenzen (begrenzttes Alphabet: A, C, G, T)

Teilzeichenkette, Präfix, Suffix

- S sei eine **Zeichenkette** der Länge m
- $S[i..j]$ ist dann eine **Teilzeichenkette** von S zwischen den Indizes i und j ($0 \leq i \leq j \leq m-1$)
- Ein **Präfix** ist eine Teilzeichenkette $S[0..i]$ ($0 \leq i \leq m-1$)
- Eine **Suffix** ist eine Teilzeichenkette $S[i..m-1]$ ($0 \leq i \leq m-1$)

Beispiele

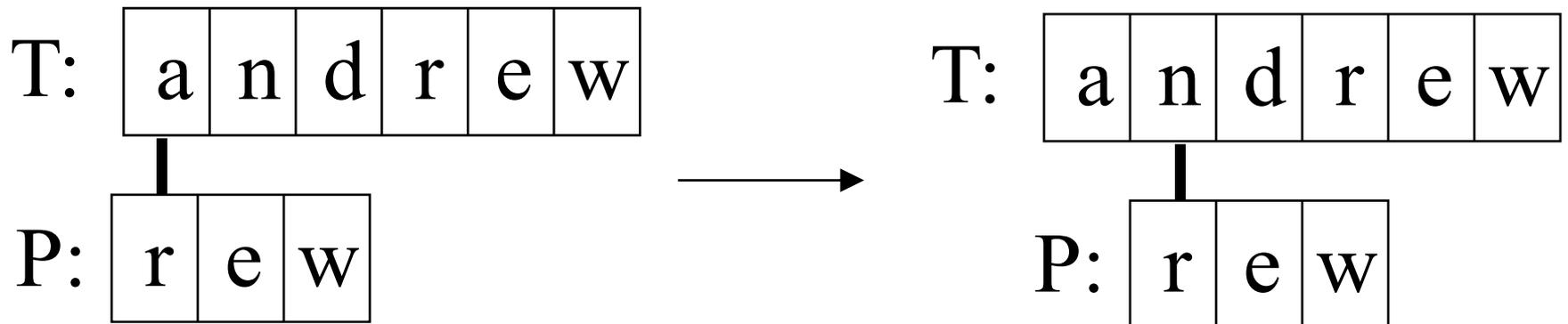
- Teilzeichenkette $S[1..3] = \text{"ndr"}$



- Alle möglichen Präfixe von S:
 - "andrew", "andre", "andr", "and", "an", "a"
- Alle möglichen Suffixe von S:
 - "andrew", "ndrew", "drew", "rew", "ew", "w"

Der Brute-Force-Algorithmus

- **Problem:** Bestimme Position des ersten Vorkommens von Muster **P** in Text **T** oder liefere **-1**, falls **P** nicht in **T** vorkommt
- **Idee:** Überprüfe jede Position im Text, ob das Muster dort startet



P bewegt sich jedes Mal um 1 Zeichen durch T



...

Brute-Force-Suche

```
function BFsearch(text, pattern: String): Integer
  n := length(text); m := length(pattern)
  for i from 0 to (n-m) do
    j := 0
    while j < m and text[i+j] = pattern[j] do // passende Teilkette
      j := j + 1
    if j = m then
      return i // erfolgreiche Suche
  return -1 // erfolglose Suche
```

- Datentyp `String` entspricht `Array [0..length-1] of Character`
- `Character` umfasse hier 128 Zeichen (ASCII)

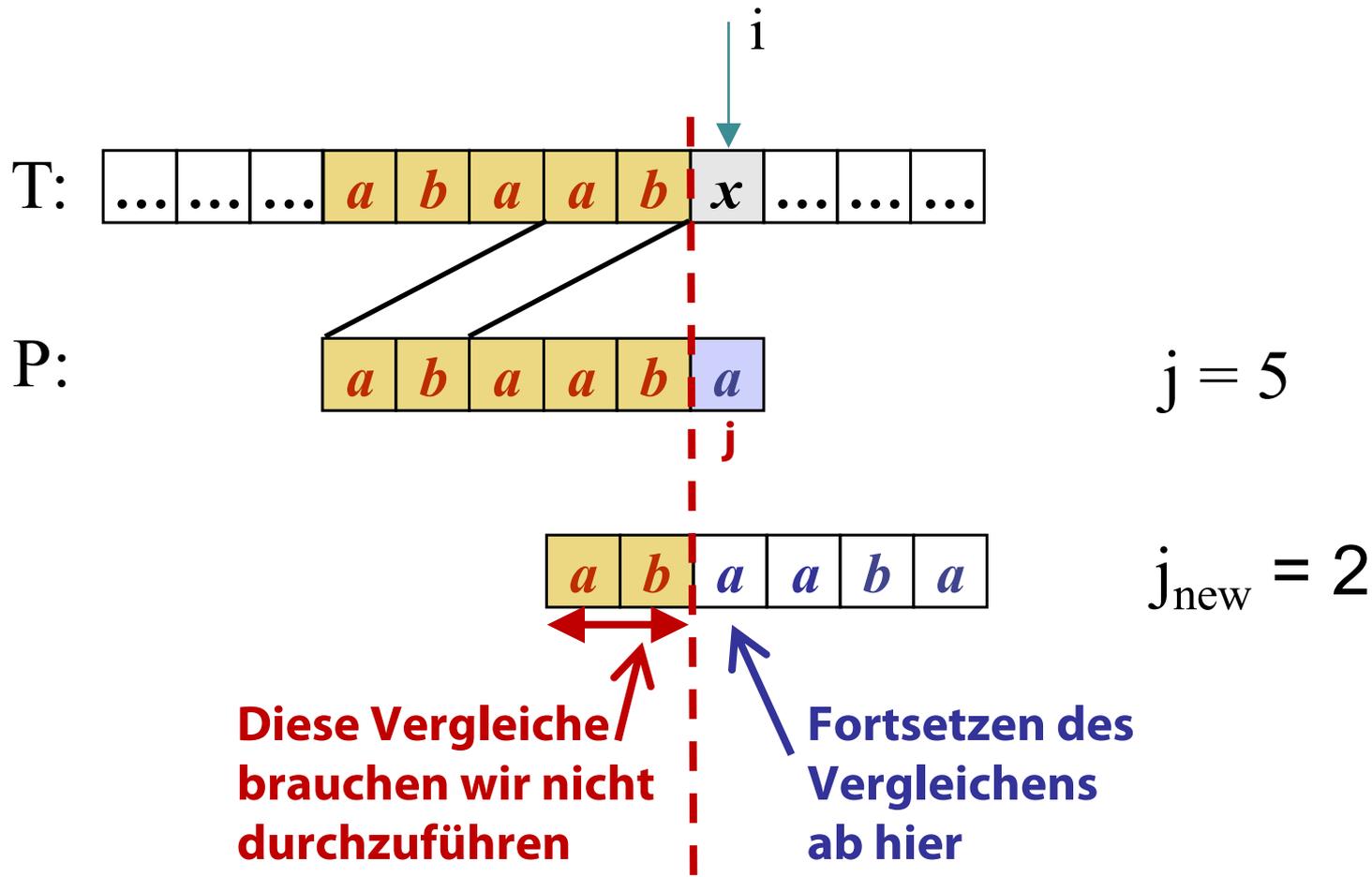
Analyse der Komplexität für Suche

- Schlechtester Fall für erfolglose Suche
 - Beispiel
 - Text: „aaaaaaaaaaaaaaaaaaaaaaaa“
 - Muster: „aaaah“
 - Das Muster wird an jeder Position im Text durchlaufen: $O(n \cdot m)$
- Bester Fall für erfolglose Suche
 - Beispiel
 - Text: „aaaaaaaaaaaaaaaaaaaaaaaa“
 - Muster: „bbbbbb“
 - Das Muster kann an jeder Position im Text bereits am ersten Zeichen des Musters falsifiziert werden: $O(n)$
- Komplexität erfolgreiche Suche im Durchschnitt
 - Meist kann das Muster bereits an der ersten Stelle des Musters falsifiziert werden und in der Mitte des Textes wird das Muster gefunden: $O(n+m)$

Weitere Analyse

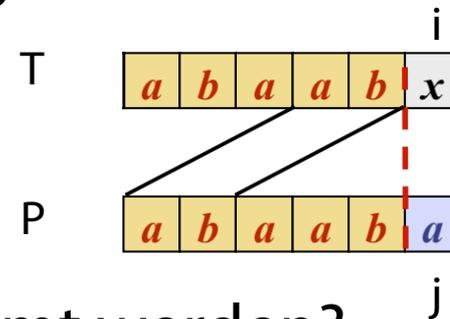
- Brute-Force-Algorithmus ist um so schneller, je größer das Alphabet ist
 - Größere Häufigkeit, dass das Muster bereits in den ersten Zeichen falsifiziert werden kann
- Für kleine Alphabete (z.B. binär 0,1) ungeeigneter
- Bessere Verschiebung des Musters zum Text als bei Brute-Force möglich?

Beispiel



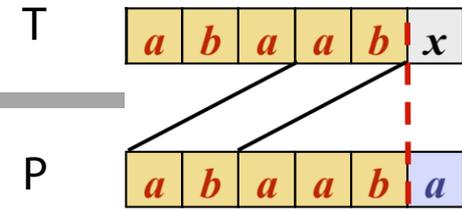
Knuth-Morris-Pratt-Algorithmus (KMP)

- Vergleich des Musters im Text von links nach rechts
 - Wie im Brute-Force-Ansatz
- Bessere Verschiebung des Musters zum Text als bei Brute-Force
 - **Frage:** Falls das Muster an der Stelle j falsifiziert wird, was ist die größtmögliche Verschiebung, um unnötige Vergleiche zu sparen?
 - **Antwort:** Verschiebe um den längsten Präfix von $P[0..j-1]$, der ein Suffix von $T[i-j..i-1]$ ist



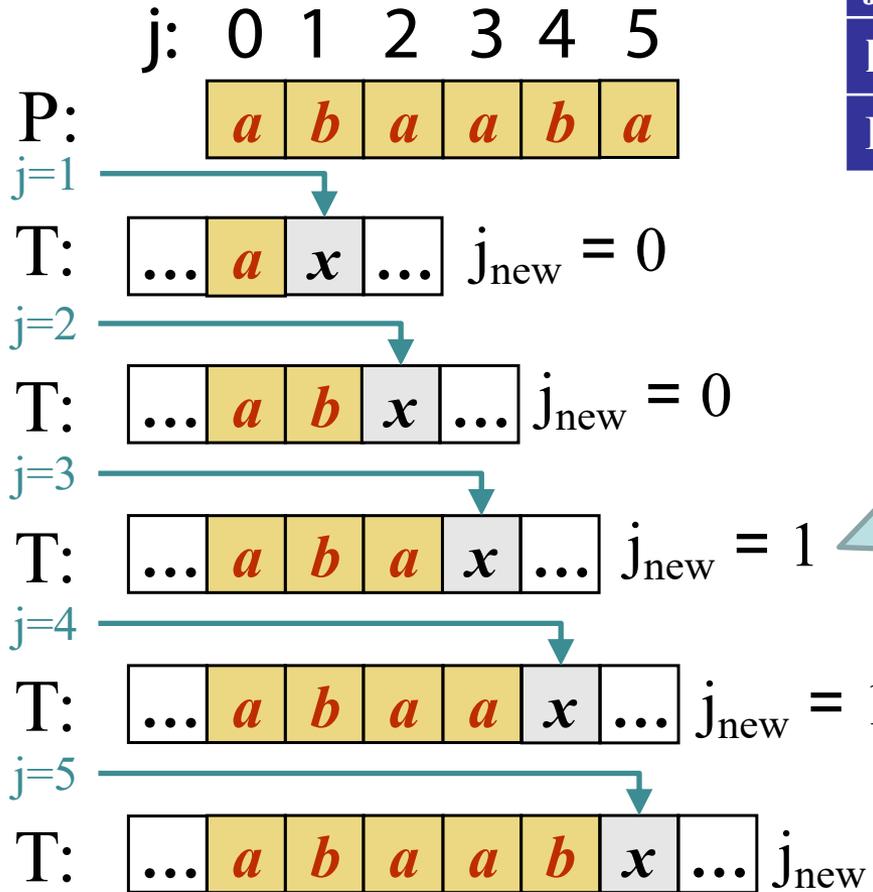
- Wie können solche Präfixe mit vertretbarem Aufwand bestimmt werden?

KMP Fehlerfunktion



- KMP verarbeitet das Muster vor, um Übereinstimmungen zwischen den Präfixen des Musters mit sich selbst zu finden
- j = Position der Ungleichheit in P
- k = Position vor der Ungleichheit ($k = j-1$).
- Die sog. Fehlerfunktion $F(k)$ ist definiert als die Länge des **längsten Präfixes von $P[0..k]$** , welcher auch ein Suffix von $T[i-j..i-1]$ ist
- Oder welcher **auch ein Suffix von $P[0..k]$** ist!
 - Wenn das nicht so wäre, käme man nicht an die Pos. j

Beispiel Fehlerfunktion



j	1	2	3	4	5
k = j-1	0	1	2	3	4
F(k)	0	0	1	1	2

- F(k) ist die Länge des längsten Präfix (j_{new} für $j=k+1$)
- Im Code wird F(k) als ein Feld gespeichert

Das erste a in P braucht bei fortgesetzter Suche nicht überprüft zu werden!

a und b in P braucht bei fortgesetzter Suche nicht überprüft zu werden!

Beispiel

T:

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

0 1 2 3 4 5

P:

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

 $F(4)=1$

0 1 2 3 4 5

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

 $F(0)=0$

0 1 2 3 4 5

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

 $F(3)=0$

0 1 2 3 4 5

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

Kein übereinstimmendes Zeichen
 $i:=i+1$

0 1 2 3 4 5

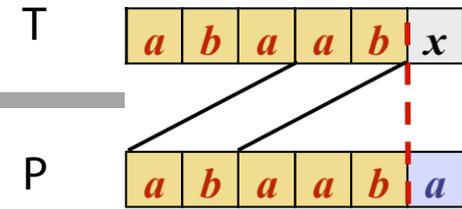
<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

k	0	1	2	3	4
$F(k)$	0	0	1	0	1

Verfahren Knuth-Morris-Pratt

```
function KMPsearch(text, pattern: String): Integer
  n := length(text); m := length(pattern)
  F := computeF(pattern)
  i := 0; j := 0
  while i < n do // passende Teilkette
    if pattern[j] = text[i] then
      if j = m - 1 then
        return i - m + 1 // erfolgreiche Suche
      i := i + 1
      j := j + 1
    else if j > 0 then
      j := F[j - 1]
    else i := i + 1
  return -1 // erfolglose Suche
```

Fehlerfunktion computeF



```
function computeF(pattern: String): Array[] of IN
  F := <0,...,0>: Array [0..length(pattern)-1 - 1] of IN
```

```
  m := length(F)
```

```
  i := 1; j := 0
```

```
  while i < m do
```

```
    if pattern[j] = pattern[i] then
      // j+1 Zeichen stimmen überein
```

```
      F[i] := j + 1
```

```
      i := i + 1; j := j + 1
```

```
    else if j > 0 then
```

```
      // j folgt dem übereinstim-
      // menden Präfix
```

```
      j := F[j - 1]
```

```
    else //keine Übereinstimmung
```

```
      F[i] := 0
```

```
      i := i + 1
```

```
  return F
```

j | i
 0 1
 0 2
 1 3
 0 4
 0 5
 1 6
 2 7
 3 8
 2 9

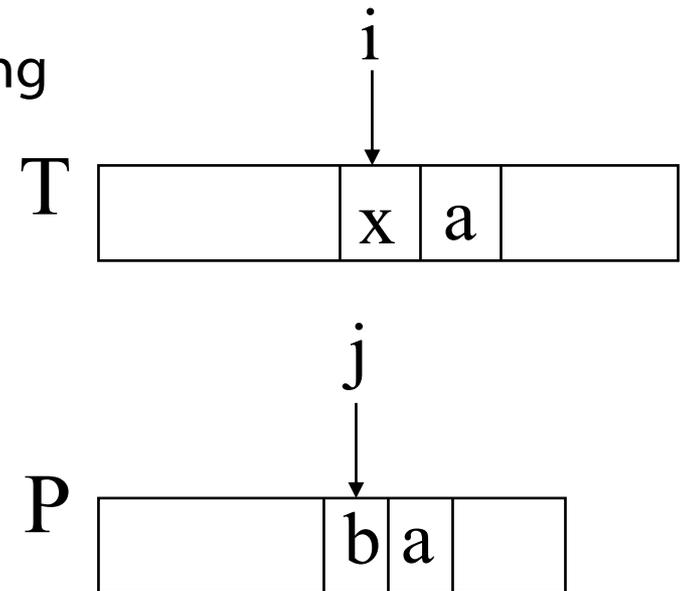
											Länge
Position:	0	1	2	3	4	5	6	7	8		
Muster:	a	b	a	b	c	a	b	a	b	a	9
Präfix (0..0):											0
Präfix (0..1):											0
Präfix (0..2):	a		a								1
Präfix (0..3):	a	b	a	b							2
Präfix (0..4):											0
Präfix (0..5):	a					a					1
Präfix (0..6):	a	b				a	b				2
Präfix (0..7):	a	b	a			a	b	a			3
Präfix (0..8):	a	b	a	b		a	b	a	b		4
	a b a			a b a							

Analyse des Knuth-Morris-Pratt-Algorithmus

- Der Algorithmus springt niemals zurück im Text
 - Geeignet für Datenströme
- Komplexität $O(m+n)$
- Algorithmus wird i.a. langsamer, wenn das Alphabet größer ist
 - Werte der Fehlerfunktion werden tendenziell kleiner

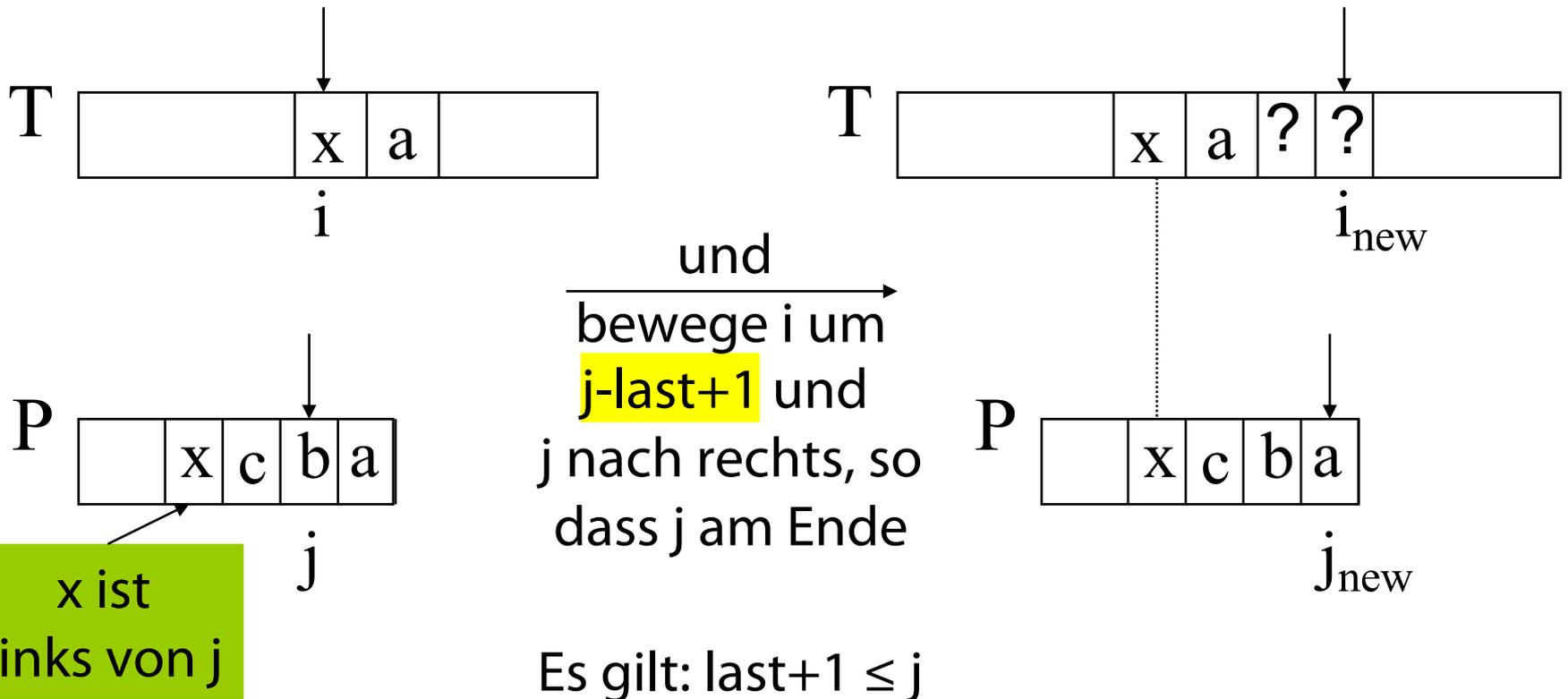
Boyer-Moore-Algorithmus

- Basiert auf 2 Techniken
 - Spiegeltechnik
 - Finde P in T durch Rückwärtslaufen durch P , am Ende beginnend
 - Zeichensprung:
 - Im Falle von Nichtübereinstimmung des Textes an der i -ten Position ($T[i]=x$) und des Musters an der j -ten Position ($P[j] \neq T[i]$)
 - 3 Fälle...



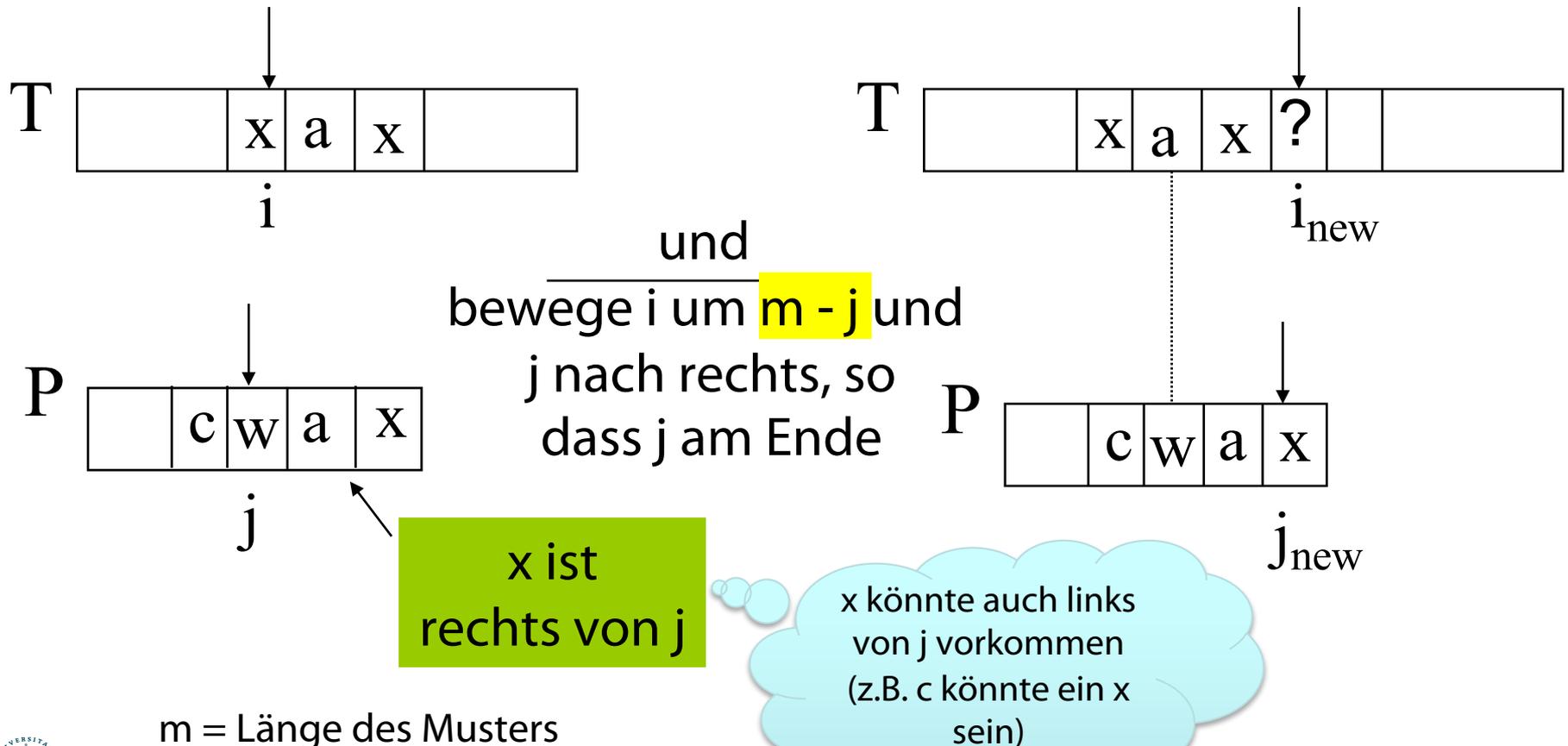
Fall 1: P enthält x nur links von j

- Bewege P nach rechts, um das letzte Vorkommen von x in P mit $T[i]$ abzugleichen



Fall 2: P enthält x rechts von j

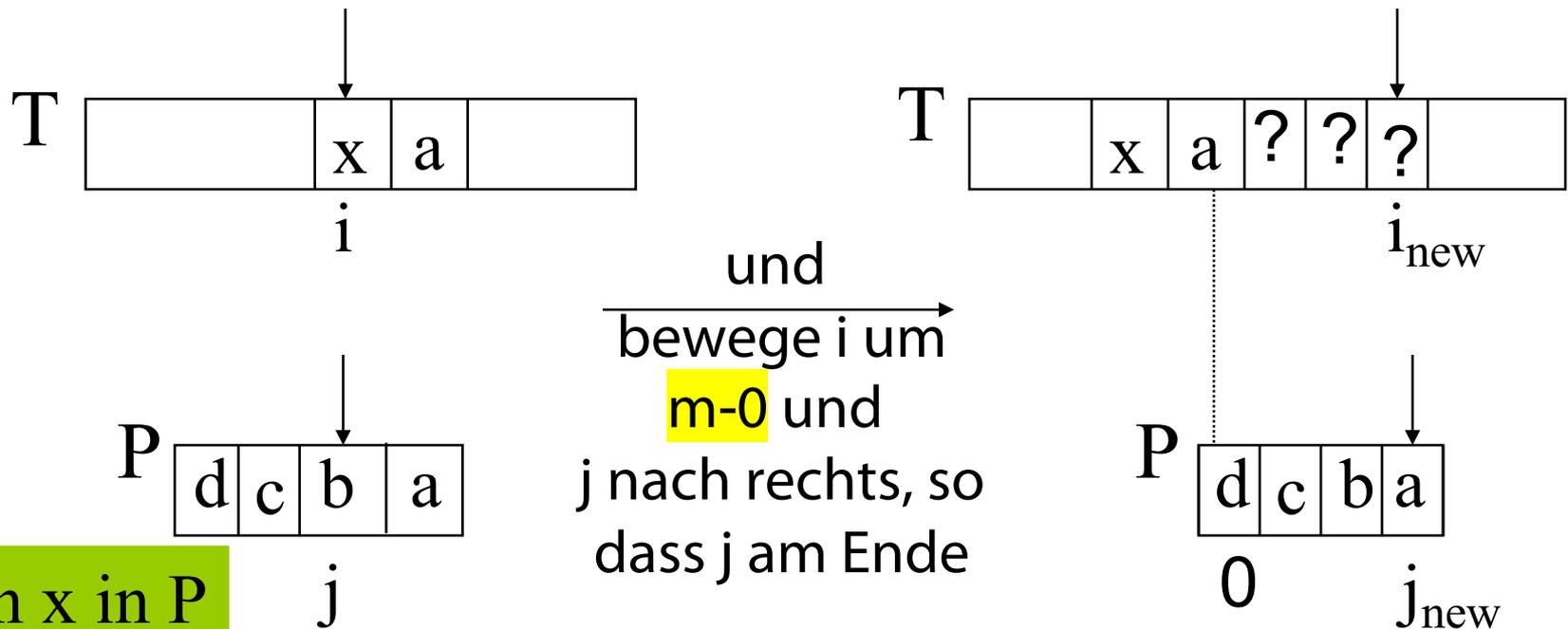
- Bewege P um 1 Zeichen nach T[i+1]



$m =$ Länge des Musters

Fall 3: Falls Fall 1 und 2 nicht anzuwenden sind (x ist *nicht* in P enthalten)

- Bewege P nach rechts, um $P[0]$ und $T[i+1]$ abzugleichen



Wenn kein x in P sollte $last=-1$ sein:

$$i_{new} = i + m - \min(j, last+1)$$

Beispiel (Boyer-Moore)

T:

a p a t t e r n m a t c h i n g a l g o r i t h m

1 3 5 11 10 9 8 7
r i t h m r i t h m r i t h m r i t h m

P:

2 4 6
r i t h m r i t h m r i t h m

Funktion des letzten Vorkommens

- Der Boyer-Moore Algorithmus verarbeitet das Muster P und das Alphabet A vor, so dass eine Funktion L des letzten Vorkommens berechnet wird
 - L bildet alle Zeichen des Alphabets auf ganzzahlige Werte ab
- $L(x)$ (mit x ist Zeichen aus A) ist definiert als
 - den größten Index i , so dass $P[i]=x$, oder
 - -1 , falls solch ein Index nicht existiert
- Implementierungen
 - L ist meist in einem Feld der Größe $|A|$ gespeichert

Beispiel L

$A = \{a, b, c, d\}$

$P = \text{„abacab“}$

P

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



x	a	b	c	d
$L(x)$	4	5	3	-1

L speichert Indexe von P

Funktion des letzten Vorkommens

```
function buildL(pattern: String): Array[0..127] of Integer
  L := <-1,...,-1>: Array [0..127] of Integer // nur ASCII unterstützt
  for i from 0 to length(pattern)-1 do
    L[pattern[i]] := i
  return L
```

Zweites Boyer-Moore-Beispiel

T:

a	b	a	c	a	a	b	a	d	c	a	b	a	c	a	b	a	a	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P: $m=6$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(a)=4, j=5$
 $i+=m-\min(j, L(a)+1)=1$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(a)=4, j=3$ $i+=3$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(a)=4, j=5$ $i+=1$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(a)=4, j=5, i+=1$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(d)=-1, j=5$ $i+=6$

a	b	a	c	a	b
---	---	---	---	---	---

 $L(d)=-1, j=5$ $i+=6$

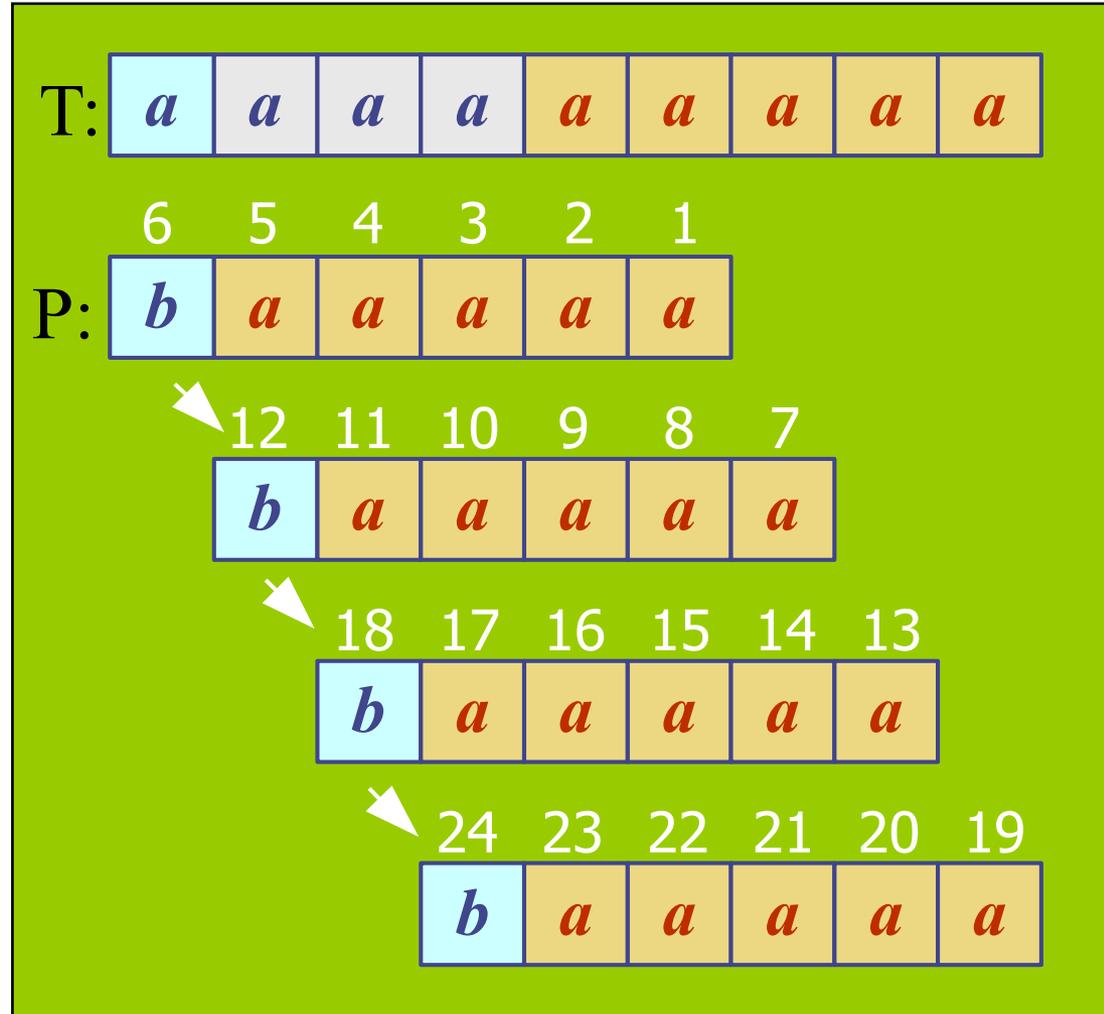
x	a	b	c	d
$L(x)$	4	5	3	-1

Boyer-Moore-Verfahren

```
function BMsearch(text, pattern: String): Integer
  L := buildL(pattern)
  n := length(text); m := length(pattern)
  i := m - 1
  if i > n - 1 then
    return -1 // Muster länger als Text, Suche erfolglos
  j := m - 1
  repeat
    if pattern[j] = text[i] then
      if j = 0 then
        return i // Suche erfolgreich
      else // Spiegeltechnik
        i := i - 1; j := j - 1
    else // Zeichensprungtechnik (last = last occurrence)
      last := L[text[i]]; i := i + m - min(j, last + 1); j := m - 1
  until i > n - 1
  return -1 // Suche erfolglos
```

Analyse der Komplexität

- Schlechtester Fall
 - Beispiel
 - T: „aaaaaaaaaa...a“
 - P: „baa...a“
 - $O(m \cdot n + |\text{Alphabet}|)$
 - Wahrscheinlichkeit hoch für schlechtesten Fall bei kleinem Alphabet



Analyse der Komplexität

- Bester Fall
 - Immer Fall 3, d.h. P wird jedes Mal um m nach rechts bewegt
 - Beispiel
 - T: „aaaaaaaa...a“
 - P: „bbb...b“
 - $O(n/m + |A|)$
 - Wahrscheinlichkeit für besten Fall höher bei großem Alphabet
- Durchschnittlicher Fall
 - nahe am besten Fall: $O(n/m + |A|)$

A = Alphabet

Rabin-Karp-Algorithmus

- Idee
 - Ermittle eine **Hash-Signatur** des Musters
 - Gehe durch den zu suchenden Text durch und vergleiche die jeweilige Hash-Signatur mit der des Musters
 - Mit geeigneten Hash-Funktionen ist es möglich, dass die **Hash-Signatur iterativ** mit konstantem Aufwand pro zu durchsuchenden Zeichen **berechnet** wird
 - Falls die Hash-Signaturen übereinstimmen, dann überprüfe noch einmal die Teilzeichenketten

Hash-Funktion für Rabin-Karp-Algorithmus

- Für ein Zeichen
 - $h(k) = \text{code}(k) \cdot q$ mit k z.B. ASCII-Code des betrachteten Zeichens und q eine Primzahl
- Für eine Zeichenkette
 - $h'(k_1..k_m) = h(k_1) + \dots + h(k_m)$
- Beispiel
 - $q=5$ (in der Praxis sollte allerdings eine möglichst große Primzahl gewählt werden)
 - $A = \{1, 2, 3, 4\}$
 - der Einfachheit halber sei hier der Code des Zeichens (der Ziffer) i wiederum i
- Berechnung der Hash-Signatur des Musters 1234:
 - $h'(1234) = 1 \cdot 5 + 2 \cdot 5 + 3 \cdot 5 + 4 \cdot 5 = 50$

Suche nach der Hash-Signatur

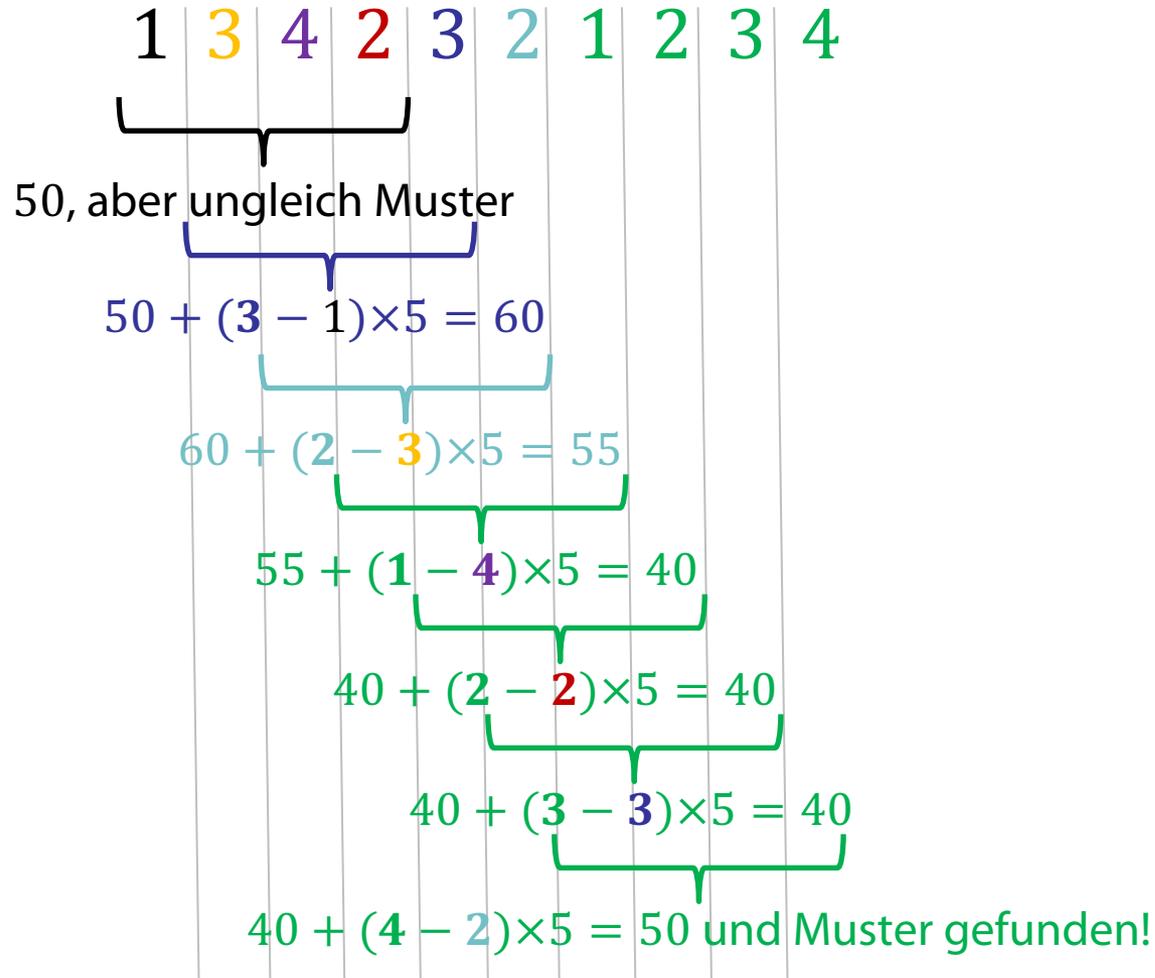
- Einmaliges Durchlaufen des zu durchsuchenden Textes und Vergleich der aktualisierten Hash-Signatur mit Hash-Signatur des Musters
- Hash-Signatur kann iterativ gebildet werden:
$$h'(k_2..k_m k_{m+1}) = h'(k_1 k_2 \dots k_m) - k_1 \cdot q + k_{m+1} \cdot q$$
$$= h'(k_1 k_2 \dots k_m) + (k_{m+1} - k_1) \cdot q$$
- **Man beachte:** Konstanter Aufwand pro Zeichen

Beispiel: Suche nach der Hash-Signatur

Muster 1234

$q:=5$

$h(1234) = 50$



Komplexitätsanalyse Rabin-Karp-Algorithmus

- Ermittle die Hash-Signatur des Musters: $O(m)$
- Gehe durch den zu suchenden Text durch und vergleiche die jeweilige Hash-Signatur mit der des Musters
 - Best (und Average) Case: $O(n)$
 - Hash-Signaturen stimmen nur bei einem Treffer überein
 - Worst Case: $O(n \cdot m)$
 - Hash-Signaturen stimmen immer überein, auch bei keinem Treffer
- Insgesamt $O(n + m)$ im Best/Average Case und $O(n \cdot m)$ im schlimmsten Fall

Zusammenfassung

- Textsuche
 - Brute-Force
 - Knuth-Morris-Pratt
 - Boyer-Moore
 - Rabin-Karp

Acknowledgements

- Präsentationen im nachfolgenden Teil sind entnommen aus dem Material zur Vorlesung Indexierung und Suchen von Tobias Scheffer, Univ. Potsdam

Indexstrukturen für eindimensionale Daten

Wiederholung: Tries

1	6	9	11	17	19	24	28	33	40	46	50	55	60
This is a text. A text has many words. Words are made from letters.													

Ausschließen von Füllwörtern/"Stop-Wörtern"...

Indexstrukturen für eindimensionale Daten

Invertierter Index

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.

Terme	Vorkommen
Letters	60
Made	50
Many	28
Text	11, 19
words	33, 40

Realisierung des Index:

Hashtabelle

Invertierter Index mit Blockadressierung

1	2	3	4
This is a text.	A text has many	words. Words are	made from letters.

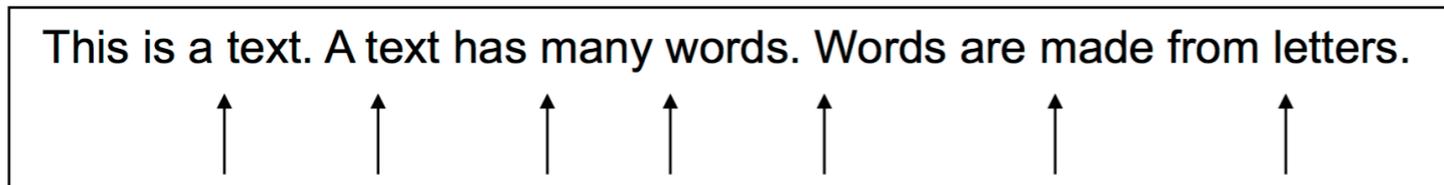
Terme	Vorkommen
Letters	4
Made	4
Many	2
Text	1, 2
words	3

Innerhalb eines Blocks
kann dann ein
Zeichenkettenabgleich
realisiert werden
(KMP, BM, RK, ...)

Suffix-Bäume

- Indexpunkte können Wortanfänge oder alle Zeichenkettenpositionen sein.
- Text ab Position: Suffix.

This is a text. A text has many words. Words are made from letters.

A rectangular box containing the text "This is a text. A text has many words. Words are made from letters." Below the text, seven vertical arrows point upwards to the start of the words: "This", "is", "a", "text.", "A", "text", and "has".

Suffixe:

text. A text has many words. Words are made from letters.

text has many words. Words are made from letters.

many words. Words are made from letters.

words. Words are made from letters.

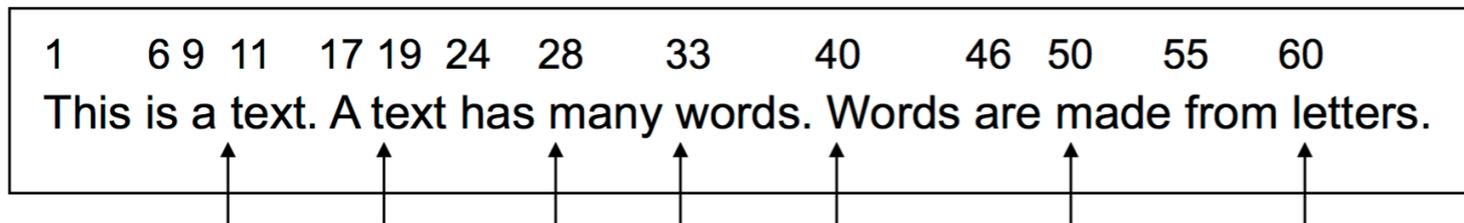
Words are made from letters.

made from letters.

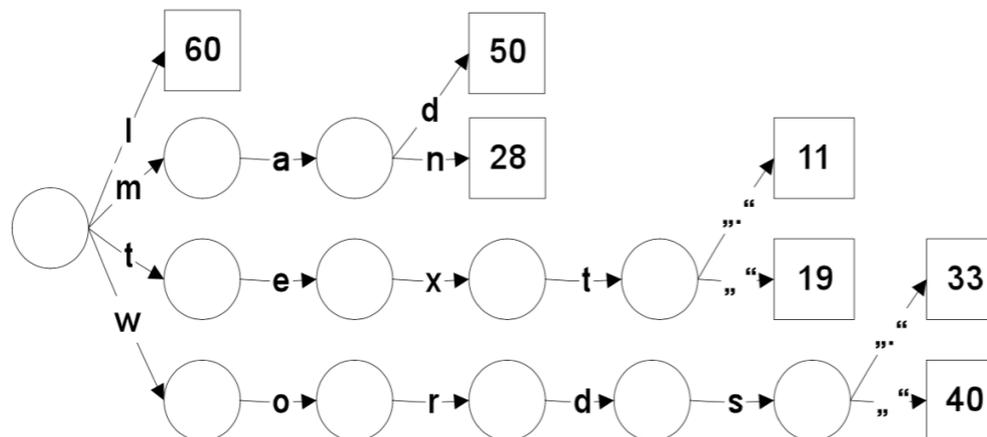
letters.

Suffix-Tries

- Aufbau eines Suffix-Tries:
- Für alle Indexpunkte:
 - Füge Suffix ab Indexpunkt in den Trie ein.



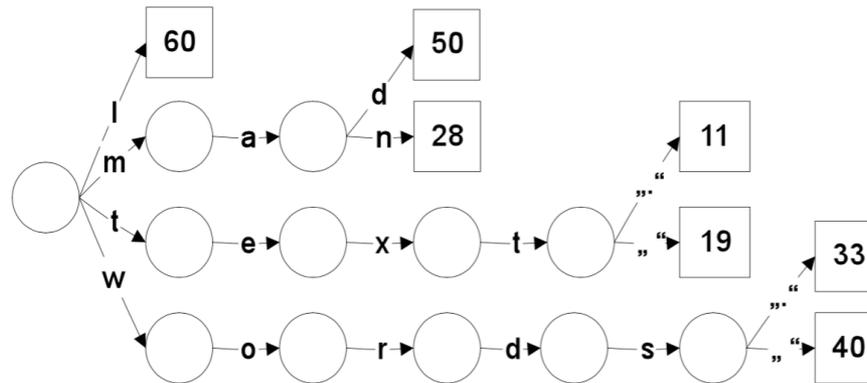
Suffix-Trie:



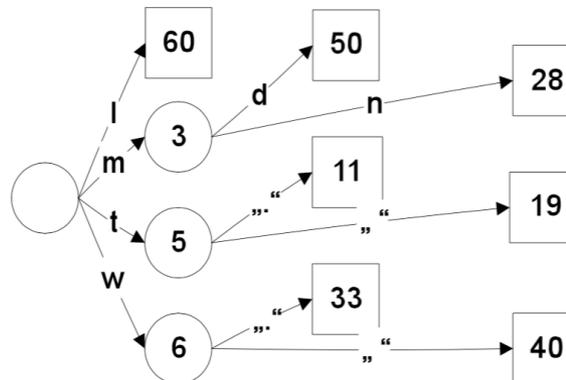
Patricia-Tries

- Ersetze interne Knoten mit nur einer ausgehenden Kante durch „Überleseknoten“, beschrifte sie mit der nächsten zu beachtenden Textposition.

Suffix-Trie:



PATRICIA-Trie
mit Überleseknoten



Suche im Suffix-Baum

Eingabe: Suchzeichenkette, Wurzelknoten.

Wiederhole

1. Wenn Terminalknoten, liefere Position zurück, überprüfe, ob Suchzeichenkette an dieser Position steht.
2. Wenn „Überleseknoten“, spring bis zur angegebenen Textposition weiter.
3. Folge der Kante, die den Buchstaben an der aktuellen Position akzeptiert.

Suffix-Bäume

- Konstruktion: $O(\text{Länge des Textes})$.
- Algorithmus funktioniert schlecht, wenn die Struktur nicht in den Hauptspeicher passt.
- Problem: Speicherstruktur wird ziemlich groß, ca. 120-240% der Textsammlung, selbst wenn nur Wortanfänge (Längenbegrenzung) indexiert werden.
- Suffix-Felder (Arrays): kompaktere Speicherung.

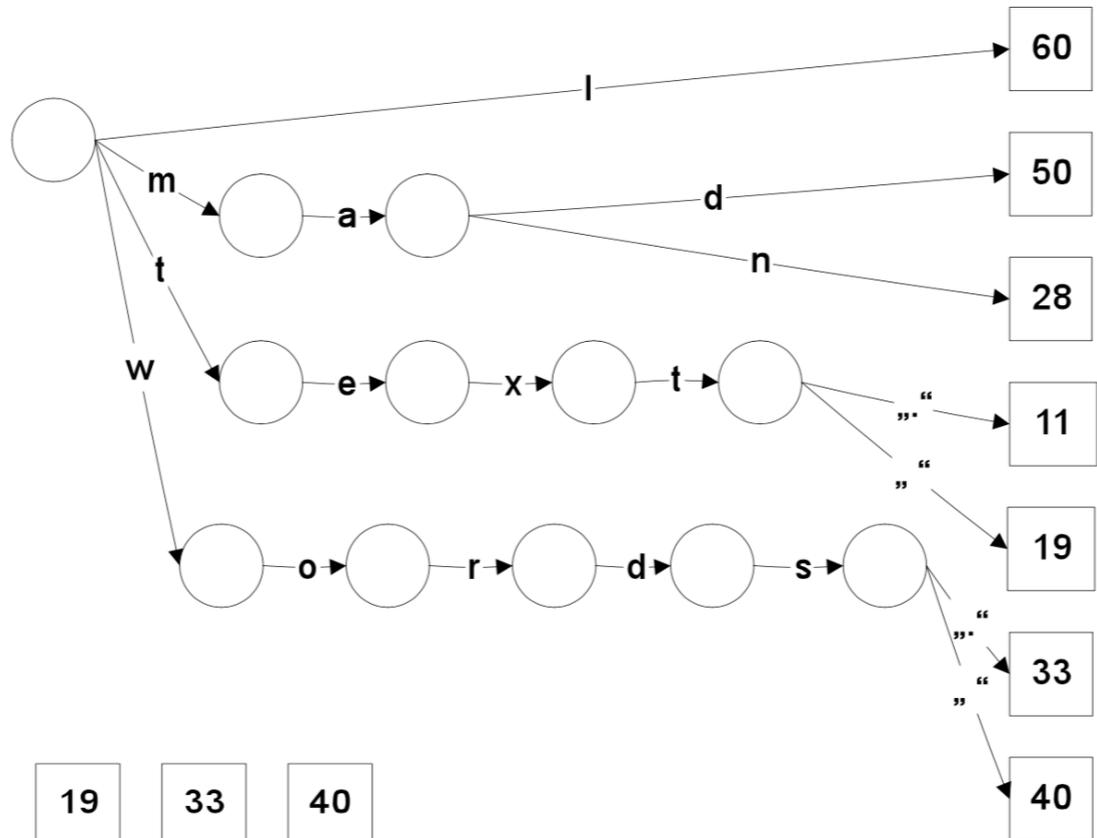
Suffix-Felder (Aufbau naiv)

- Suffix-Trie in lexikographische Reihenfolge bringen.
- Suffix-Feld= Folge der Indexpositionen.

Suffix-Trie,

Lexikographisch

sortiert:



Suffix-Array:



Beispiel

Suffixsortierung → Binäre Suche

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

length(SA) = 11

Beispiel: Suche „sip“ in „mississippi“

SA	L	7	4	1	0	M	8	6	3	5	R
----	---	---	---	---	---	---	---	---	---	---	---

i

i i i m

p p s s s

s

p s s i

i p i i s

s

p s s s

i p s i

i

i i i s

p s p

s

p s i

i i p

s

p s s

p i

i

i i s

p

p

p i

i

p

p p

i

i p

i

sip < ssi

sip > i

sip > pi

Suche: sip

SA	10	7	4	1	0	L	8	6	M	5	R
----	----	---	---	---	---	---	---	---	---	---	---

i	i	i	i	m	p	p	s	s	s	s
	p	s	s	i	i	p	i	i	s	s
	p	s	s	s		i	p	s	i	i
	i	i	i	s			p	s	p	s
		p	s	i			i	i	p	s
		p	s	s				p	i	i
		i	i	s				p		p
			p	i				i		p
			p	p						i
			i	p						
				i						

sip < sis

SA	10	7	4	1	0	L	8	M	R	5	2
----	----	---	---	---	---	---	---	---	---	---	---

i	i	i	i	m	p	p	s	s	s	s
	p	s	s	i	i	p	i	i	s	s
	p	s	s	s		i	p	s	i	i
	i	i	i	s			p	s	p	s
		p	s	i			i	i	p	s
		p	s	s				p	i	i
		i	i	s				p		p
			p	i				i		p
			p	p						i
			i	p						
				i						



1 Vorkommen

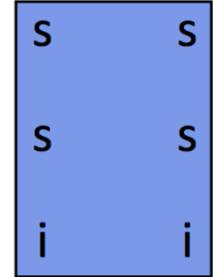
Mehrere Vorkommen?

Vorkommen benachbart!

sip = sip

SA	10	7	4	1	0	9	8	6	L	M	R
----	----	---	---	---	---	---	---	---	---	---	---

i	i	i	i	m	p	p	s	s	s	s
	p	s	s	i	i	p	i	i	s	s
	p	s	s	s		i	p	s	i	i
	i	i	i	s			p	s	p	s
		p	s	i			i	i	p	s
		p	s	s				p	i	i
		i	i	s				p		p
			p	i				i		p
			p	p						i
			i	p						
				i						



2 Vorkommen

Suche nach ssi

Suche in Suffix-Feldern: Binärsuche

```
function find(p, A): Integer
  // Suchzeichenkette p, Suffix-Feld A
  // liefert Position von p in A oder -1, falls p nicht in A
  l := 1; r := length(A)
  n := length(p)
  while s < e do
    m :=  $\lceil (l + r) / 2 \rceil$  // nach oben runden
    if p = A[m..m+n-1] then
      return m
    if p < A[m..m+n-1]
    then r := m
    else l := m
  return -1
```

U. Manber and G.W. Myers "Suffix arrays: A new method for on-line string searches". In Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, 1990

Analyse der Aufwände: Zeitbedarf

- Suffix-Bäume: $O(\text{Länge der Suchzeichenkette})$
- Suffix-Felder: $O(\log(\text{Länge der Textsammlung}))$

Aufbau von Suffix-Feldern

- Ukkonens Algorithmus (1995, Hauptspeicher)
 - $O(n)$ für den Aufbau von Suffix-Bäumen
 - Traversierung durch Suffix-Baum und Transformation zu Suffix-Feld in $O(n)$
- Später: Datenbank-basierte Verfahren (ab 2001)
z.B. für Bioinformatik-Anwendungen

Weiner, Peter (1973). Linear pattern matching algorithms, 14th Annual Symposium on Switching and Automata Theory, pp. 1–11, **1973**

McCreight, Edward Meyers, A Space-Economical Suffix Tree Construction Algorithm". Journal of the ACM 23 (2): 262–272, **1976**

Ukkonen, E., On-line construction of suffix trees, Algorithmica 14 (3): 249–260, **1995**

Hunt, E., Atkinson, M. and Irving, R. W. "A Database Index to Large Biological Sequences". VLDB **2001**

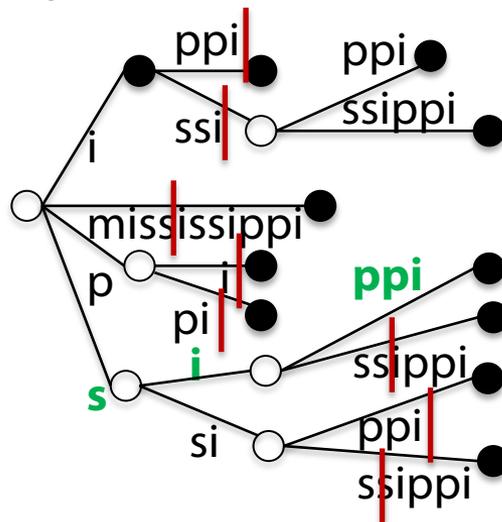
Tata, Hankins, Patel: „Practical Suffix Tree Construction“, VLDB **2004**

Approximativer Zeichenkettenabgleich

- *Editierabstand* (auch *Levenshtein-Distanz* genannt) von 2 Zeichenketten als *Ähnlichkeitsmaß*
 - minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um eine in eine andere (gegebene) Zeichenkette umzuwandeln
 - Bsp für Editierabstand 3:
 - Algo *ersetze l durch u*
 - Augo *ersetze g durch D*
 - AuDo *lösche o*
 - AuD

Approximativer Zeichenkettenabgleich

- Suche in y Unterzeichenketten, die Editierabstand von höchstens k von x haben
 - Effizient in Suffix-Tries möglich
 - „Grobe“ Idee:
Falls Editierabstand ab einer Teilzeichenkette „immer“ $>k$ ist, dann beende die Suche dort
Bsp.: Suffix-Trie von „mississippi“, Suche nach **suppe** mit $k \leq 2$:



Abbruch wegen #Edits > 2

Berechnungen für
gemeinsame
Präfixe nur einmal!

Approximativer Zeichenkettenabgleich

- Dynamische Programmierung zum Bestimmen des Edit-Abstandes zweier Zeichenketten

Berechnung von $C[0..m, 0..n]$; $C[i,j]$ = minimale # Fehler beim Abgleich von $x[1..i]$ mit $y[1..j]$

$$C[0, j] = j$$

$$C[i, 0] = i$$

$$C[i, j] = \begin{cases} C[i-1, j-1], & \text{wenn } x[i] = y[j] & \text{Buchstabe stimmt überein!} \\ 1 + \min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, & \text{sonst} \end{cases}$$

Buchstabe hinzufügen löschen ersetzen

$$O(nm)$$

Beispiel

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1], & \text{wenn } x[i] = y[j] \text{ Buchstabe stimmt überein!} \\ 1 + \min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, & \text{sonst} \end{cases}$
Buchstabe: hinzufügen löschen ersetzen

		a	b	c	a	b	b	b	a	a
	0	1	2	3	4	5	6	7	8	9
c	1	1	2	2	3	4	5	6	7	8
b	2	2	1	2	3	3	4	5	6	7
a	3	2	2	2	2	3	4	5	5	6
b	4	3	2	3	3	2	3	4	5	6
a	5	4	3	3	3	3	3	4	4	5
c	6	5	4	3	4	4	4	4	5	5

Beispiel

		j									
			a	b	c	a	b	b	b	a	a
i		0	1	2	3	4	5	6	7	8	9
	c	1	1	2	2	3	4	5	6	7	8
	b	2	2	1 =	2	3	3	4	5	6	7
	a	3	2	2	2	2 =	3	4	5	5	6
	b	4	3	2	3	3	2	3 =	4	5	6
	a	5	4	3	3	3	3	3	4	4 =	5
	c	6	5	4	3	4	4	4	4	5	5









Löschen

Einfügen

Substitution

Keine Änderung

cbabac -> ababac -> abcabac -> abcabbac ->
abcabbbac -> abcabbbaa

Editierabstand immer $>k$, falls ganze Spalte $>k$

		j									
i		a	b	c	a	b	b	b	a	a	
	0	1	2	3	4	5	6	7	8	9	
c	1	1	2	2	3	4	5	6	7	8	↓ Löschten
b	2	2	1 =	2	3	3	4	5	6	7	→ Einfügen
a	3	2	2	2	2 =	3	4	5	5	6	↘ Substitution
b	4	3	2	3	3	2	3 =	4	5	6	→ = Keine Änderung
a	5	4	3	3	3	3	3	4	4 =	5	
c	6	5	4	3	4	4	4	4	5	5	

**Ganze Spalte >2
 \Rightarrow Abbruch falls $k \leq 2$ gefordert!
 (Vgl. Approx.-suche in Trie)**

Nicht jede Zelle der Matrix braucht berechnet zu werden (\rightarrow Performanzsteigerung)

Wang, Jiannan, Jianhua Feng, and Guoliang Li. "Trie-join: Efficient trie-based string similarity joins with edit-distance constraints." VLDB 2010

Zusammenfassung

- Exakte Zeichenkettensuche
 - Knuth-Morris-Pratt
 - Boyer-Moore
 - Rabin-Karp
- Suffix-Tries
- Suffix-Bäume
- Approximativer Zeichenkettenabgleich
 - Levenshtein-Distanz
 - Suchraumbeschneidung im Suffix-Trie
 - Dynamische Programmierung