Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren

Motivation Zeichenkettenabgleich

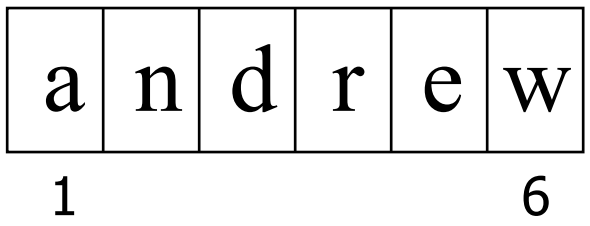
- Gegeben eine Folge von Zeichen (Text), in der eine Zeichenkette (Muster) gefunden werden soll
- Varianten
  - *Alle* Vorkommen des Musters im Text
  - Ein *beliebiges* Vorkommen im Text
  - *Erstes* Vorkommen im Text
- Anwendungen
  - Suchen von Mustern in DNA-Sequenzen (begrenztes Alphabet: A, C, 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 ($1 \leq i \leq j \leq m$)
- Ein Präfix ist eine Teilzeichenkette S[1..i] ($1 \leq i \leq m$)
- Eine Suffix ist eine Teilzeichenkette S[i..m] ($1 \leq i \leq m$)

Beispiele

- Teilzeichenkette S[2..4] = "ndr"

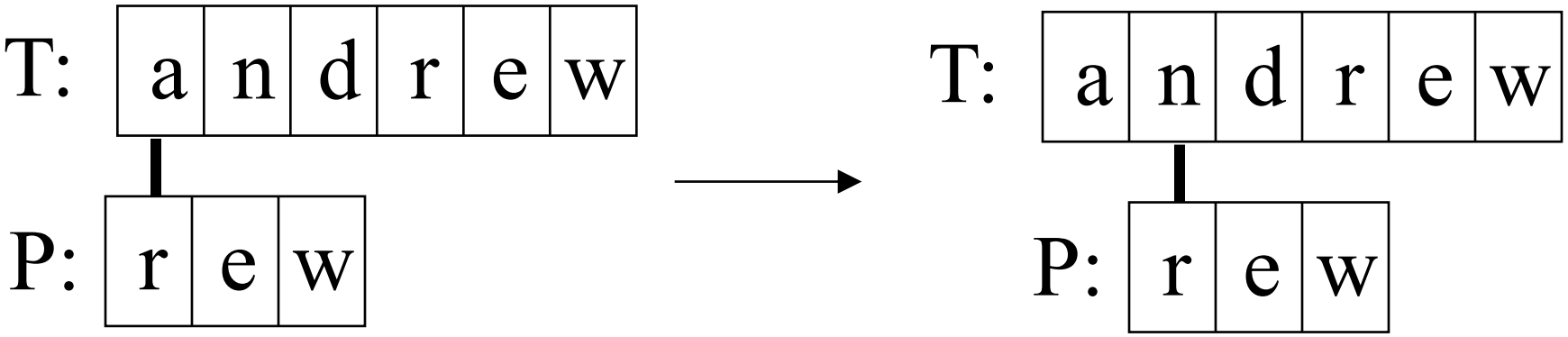


- Alle möglichen Präfixe von S:
  - "andrew", "andre", "andr", "and", "an”, "a"

- 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 daraufhin, ob das Muster dort startet



P bewegt sich jedes Mal um 1 Zeichen durch T

. . . .

Brute-Force-Suche

```
function bf_search(t :: String, p :: String) :: Int
  n = length(t); m = length(p)
  for i = 1:(n - m) + 1
    j = 0
    while j < m && t[i + j] == p[j + 1] # passende Teilkette
      j += 1
    end
    if j == m    return i end           # erfolgreiche Suche
  end
  return -1                            # erfolglose Suche
end
```

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

Analyse der Komplexität für Suche

- Schlechtester Fall für erfolglose Suche
  - Beispiel
    - Text: „aaaaaaaaaaaaaaaaaaaaa“
    - Muster: „aaaah“
  - Das Muster wird an jeder Position im Text durchlaufen: O(n·m)
- Bester Fall für erfolglose Suche
  - Beispiel
    - Text: „aaaaaaaaaaaaaaaaaaaa“
    - 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)

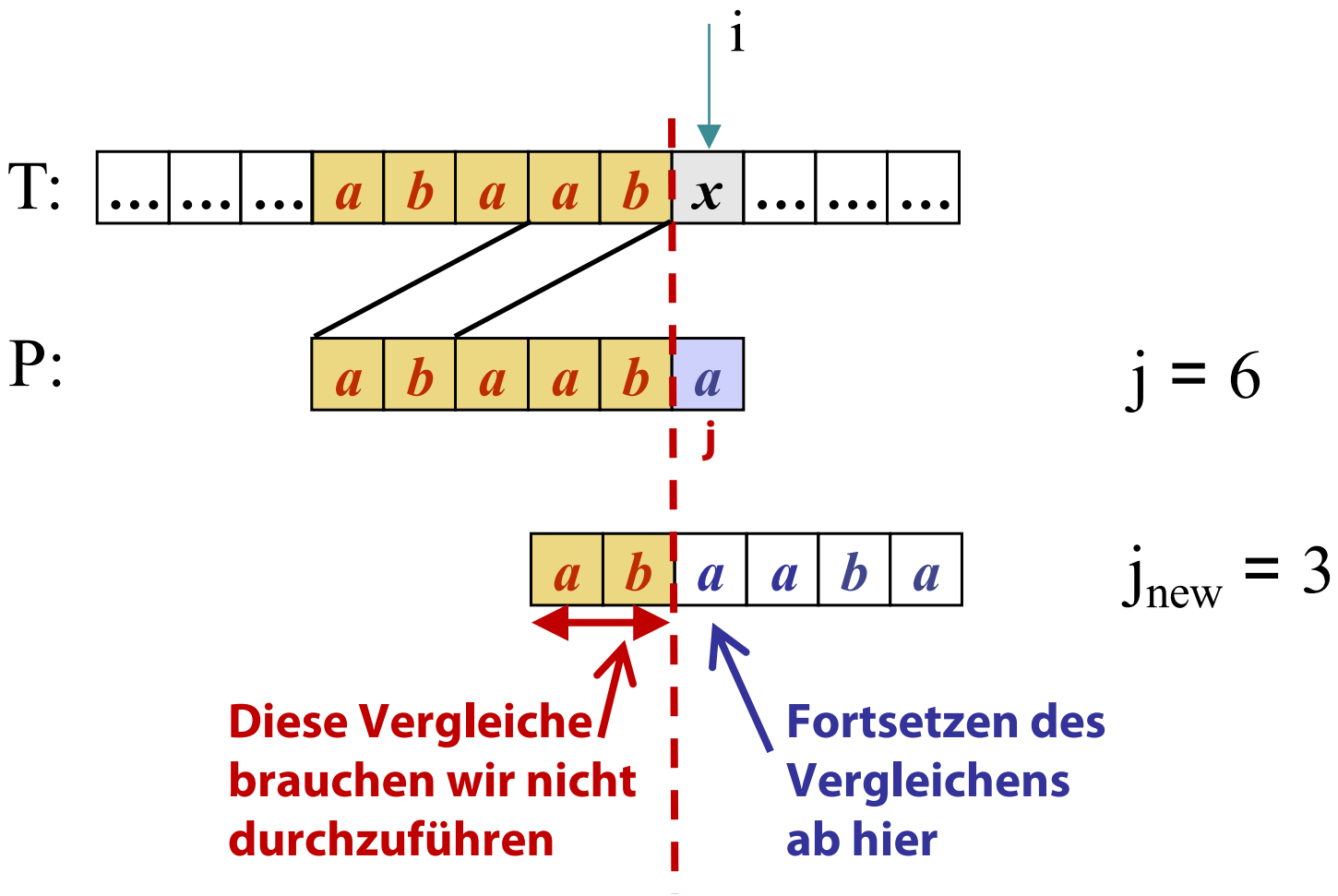
Weitere Analyse

- 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

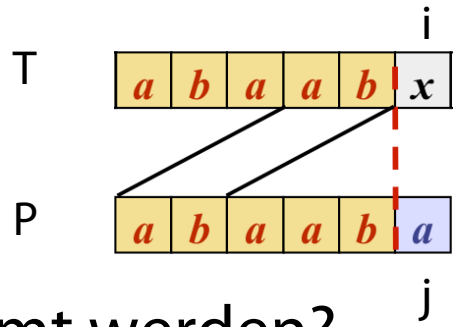
- Bessere Verschiebung des Musters zum Text als bei Brute-Force möglich?

Beispiel


i
T:
… … … a b a a b x … … …
P:
a b a a b a
j
j = 6
a b a a b a
j_{new} = 3
Diese Vergleiche brauchen wir nicht durchzuführen
Fortsetzen des Vergleichens ab hier

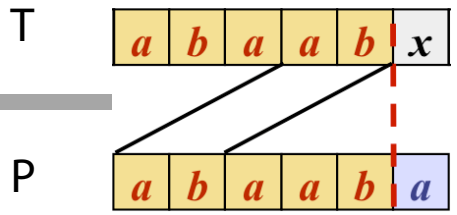
Knuth-Morris-Pratt-Algorithmus (KMP)

- 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[1..j], der ein Suffix von T[i-j+1..i] ist



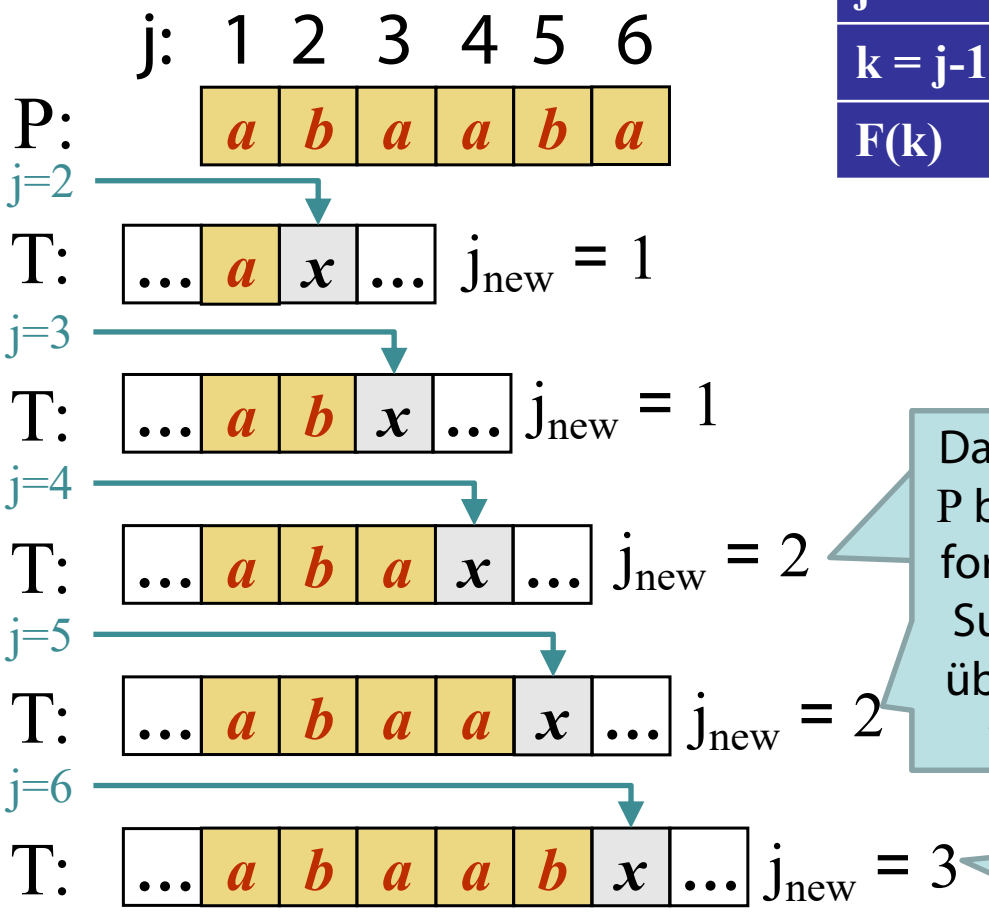
- Wie können solche Präfixe mit vertretbarem Aufwand bestimmt werden?

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[1..k+1], welcher auch ein Suffix von T[i-j+1..i] ist
- Oder welcher auch ein Suffix von P[1..k+1] ist!
  - Wenn das nicht so wäre, käme man nicht an die Pos. j

Beispiel Fehlerfunktion



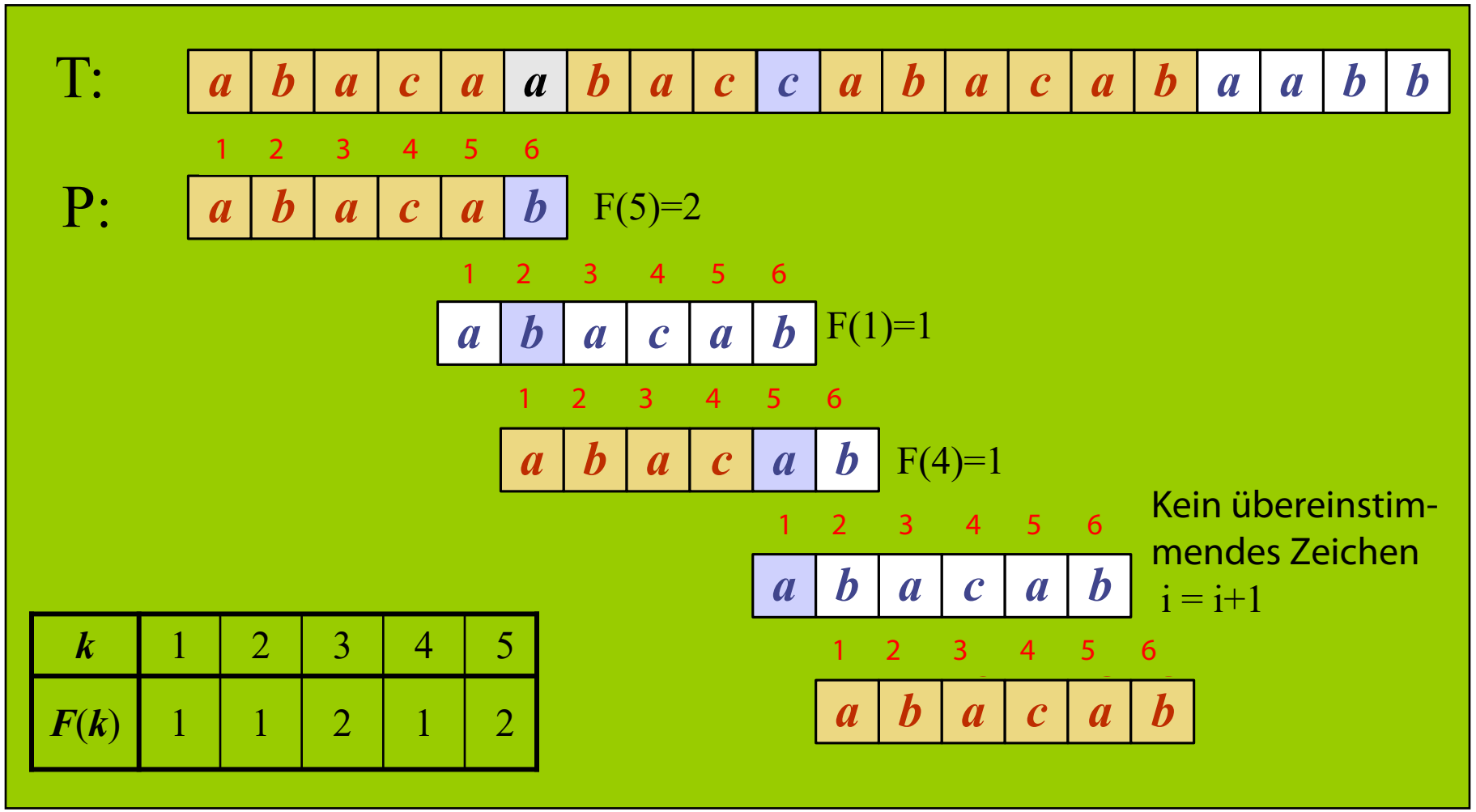
j	2	3	4	5	6
k = j-1	1	2	3	4	5
F(k)	1	1	2	2	3

- 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



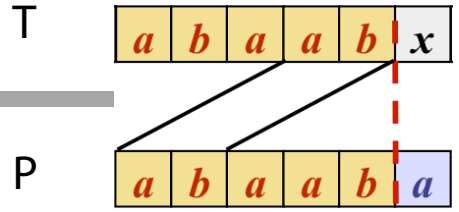
k	1	2	3	4	5
$F(k)$	1	1	2	1	2

Verfahren Knuth-Morris-Pratt

```
function kmp_search(t :: String, p :: String) :: Int
  n = length(t); m = length(p); f = compute_f(p)
  i = 1; j = 1
  while i <= n # passende Teilkette
    if p[j] == t[i]
      if j == m
        return i - m + 1 # erfolgreiche Suche
      end
      i += 1; j += 1
    elseif j > 1
      j = f[j-1]
    else
      i += 1
    end end
  return -1 # erfolglose Suche
end
```

Fehlerfunktion compute_f

```
function compute_f(p :: String) :: Array{Int}
  f = fill(1, length(p) - 1)
  m = length(f); i = 2; j = 1
  while i <= m
    if p[j] == p[i]
    # j Zeichen stimmen überein
      f[i] = j + 1
      i += 1; j += 1
    elseif j > 1
    # j folgt dem übereinstim. Präfix
      j = f[j]
    else # keine Übereinstimmung
      f[i] = 1; i += 1
  end end
  return f
end
```



j	i
1	2
1	3
2	4
3	5
1	6
2	7
3	8
4	9
5	10

1 ~~2~~ ~~3~~

Position	1	2	3	4	5	6	7	8	9	Länge	
Muster	a	b	a	b	c	a	b	a	b	af[k]	k
Präfix [1..1]										1	1
Präfix [1..2]										1	2
Präfix [1..3]	a		a							2	3
Präfix [1..4]	a	b	a	b						3	4
Präfix [1..5]										1	5
Präfix [1..6]	a					a				2	6
Präfix [1..7]	a	b				a	b			3	7
Präfix [1..8]	a	b	a			a	b	a		4	8
Präfix [1..9]	a	b	a	b		a	b	a	b	5	9

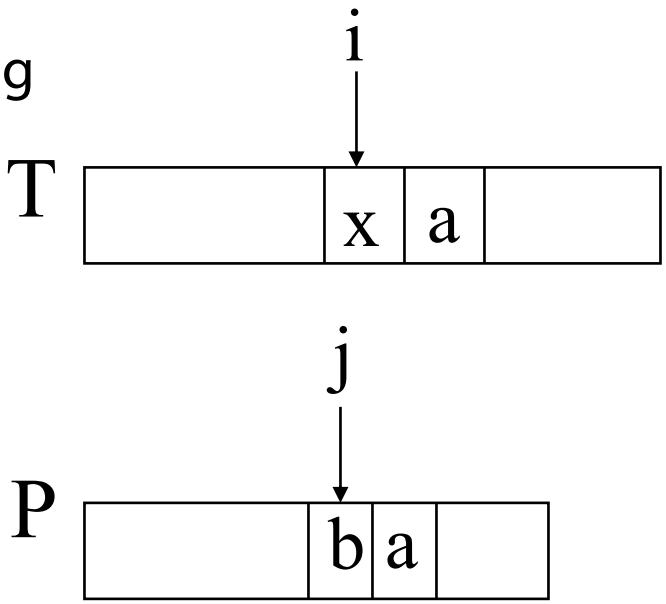
Analyse des Knuth-Morris-Pratt-Algorithmus

- 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

D. E. Knuth, J. H. Morris, V. R. Pratt: Fast Pattern Matching in Strings.
In: SIAM Journal of Computing. 6 (2): 323–350, **1977**

Boyer-Moore-Algorithmus

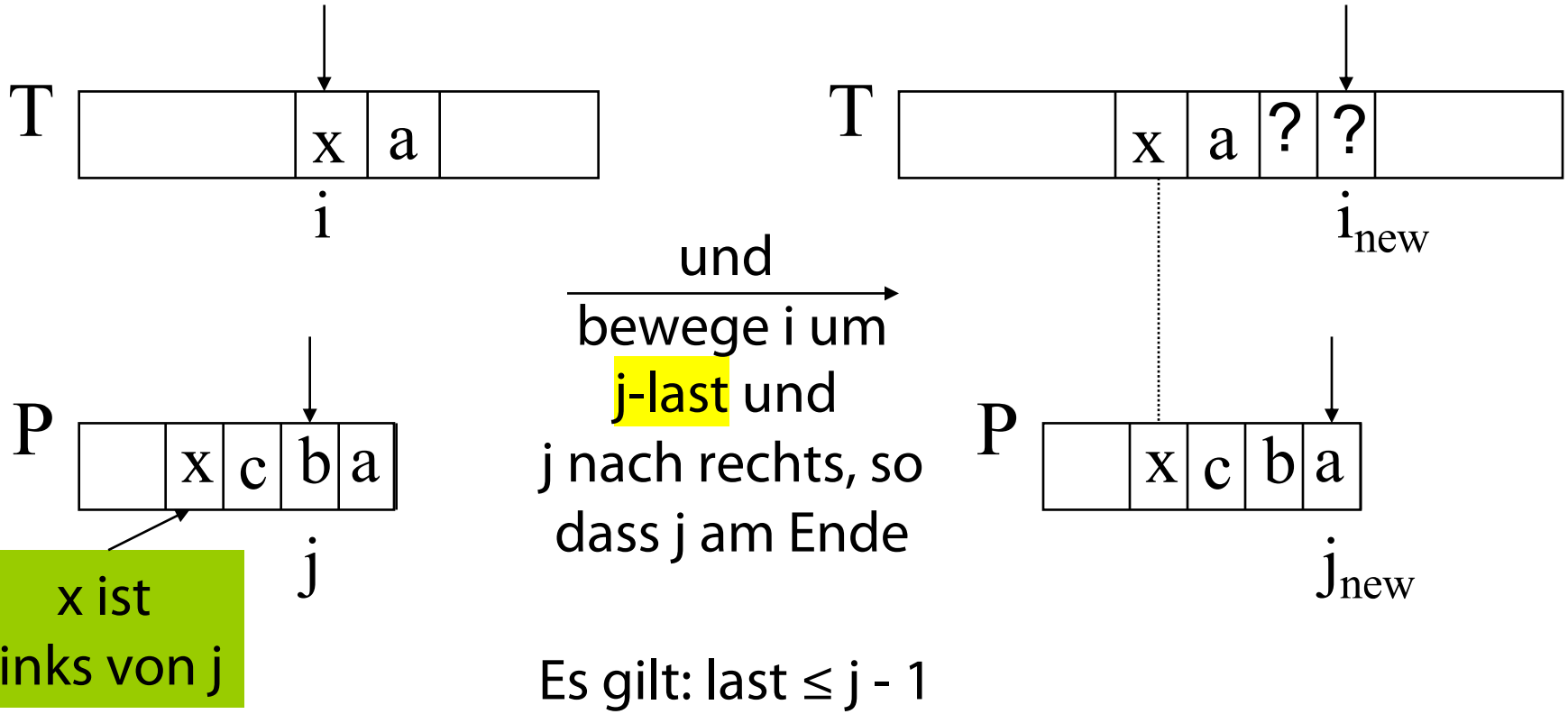
- Basiert auf 2 Techniken
  - Spiegeltechnik
    - Finde P in T durch Rückwärtslaufen durch P, am Ende beginnend
  - Zeichensprung:
    - Im Falle von Nichtübereinstimmung des Textes an der i-ten Position (T[i]=x) und des Musters an der j-ten Position (P[j]≠T[i])
    - 3 Fälle…



Robert S. Boyer, J. Strother Moore: A fast string searching algorithm. In: Communications of the ACM. Bd. 20, Nr. 10, S. 762–772, **1977**

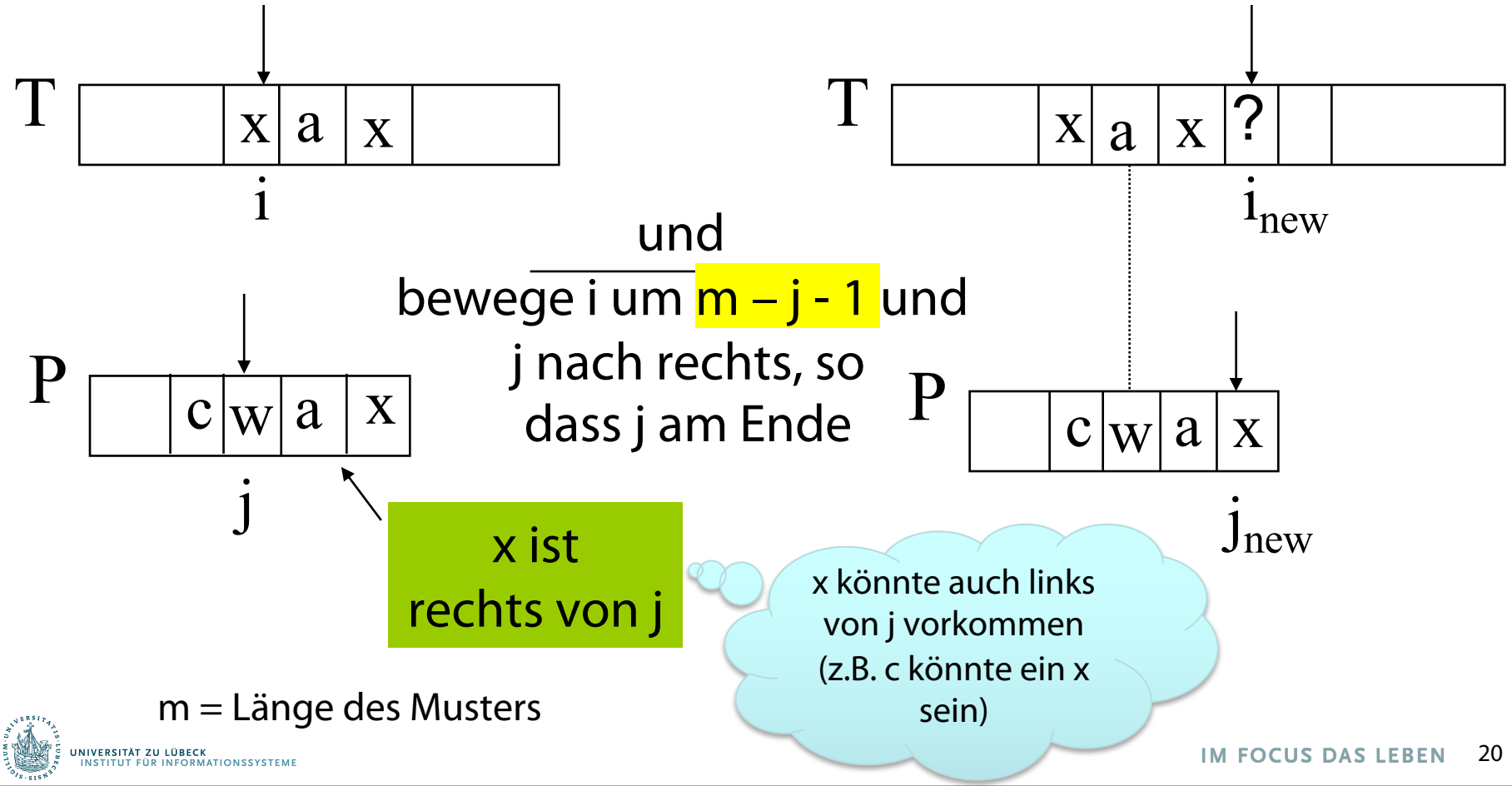
Fall 1: P enthält x nur links von j

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



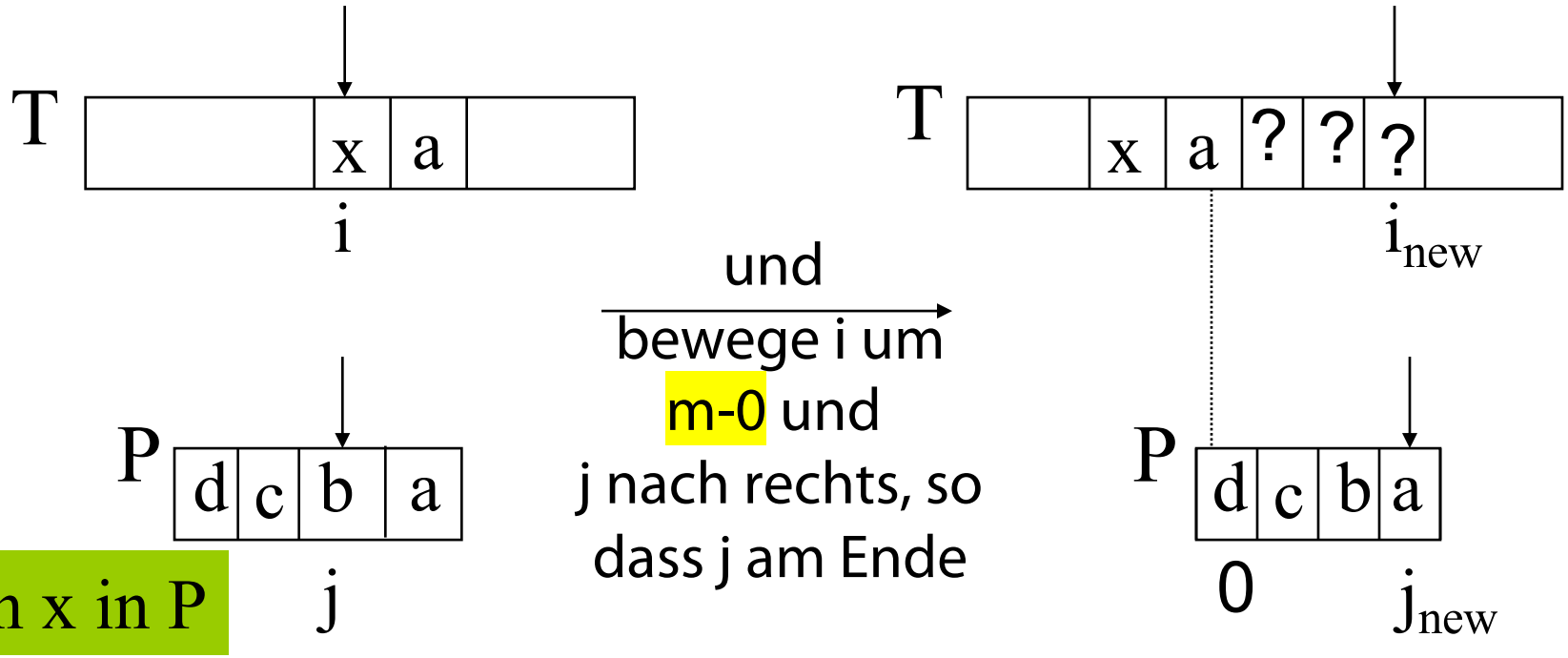
Fall 2: P enthält x rechts von j

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



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

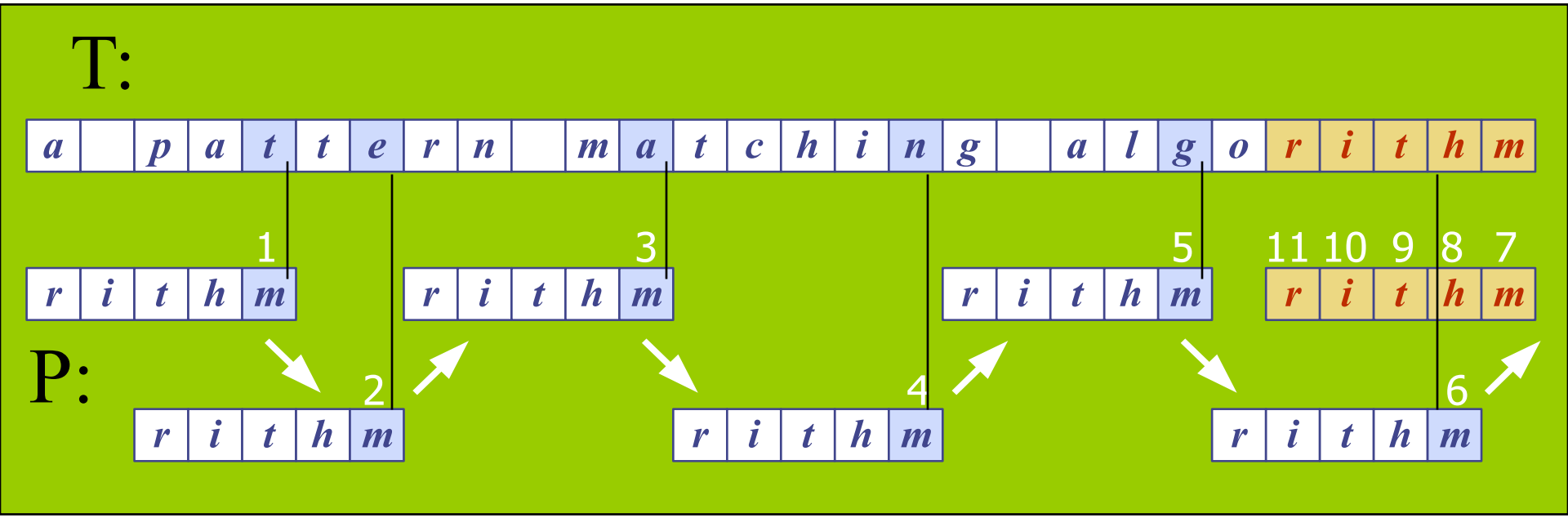
- Bewege P nach rechts, um P[1] und T[i+1] abzugleichen



Wenn kein x in P sollte last=0 sein:

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

Beispiel (Boyer-Moore)



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
  - 0, falls solch ein Index nicht existiert
- Implementationen
  - L ist meist in einem Feld der Größe |A| gespeichert

Beispiel L

A = {a, b, c, d}
P = „abacab“

P

a	b	a	c	a	b
1	2	3	4	5	6

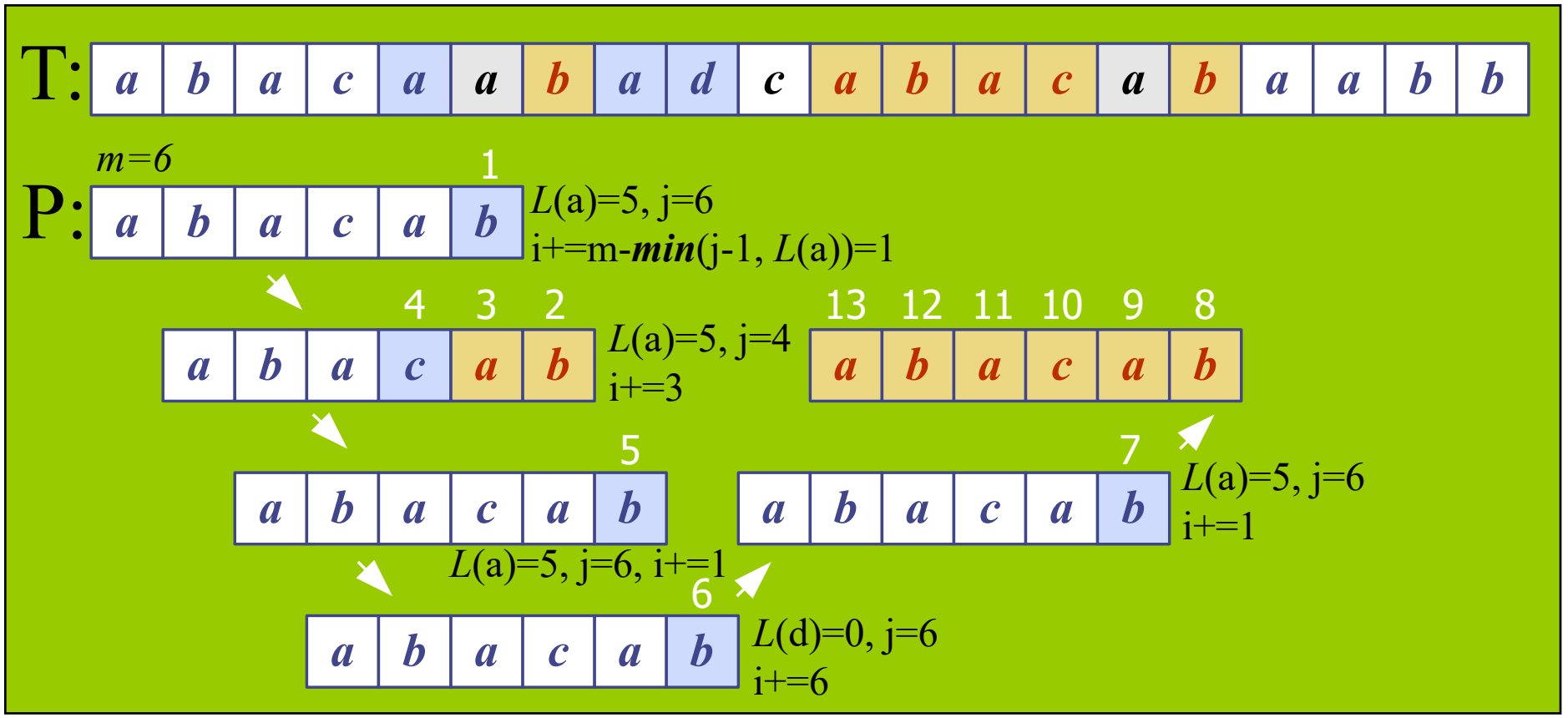
x	***a***	***b***	***c***	***d***
L(x)	5	6	4	0

L speichert Indexe von P

Funktion des letzten Vorkommens

```
function build_l(p :: String) :: Array{Int}
  l = fill(0, 128) # nur ASCII unterstützt
  for i = 1:length(p)
    l[Int(p[i])] = i
  end
  return l
end
```

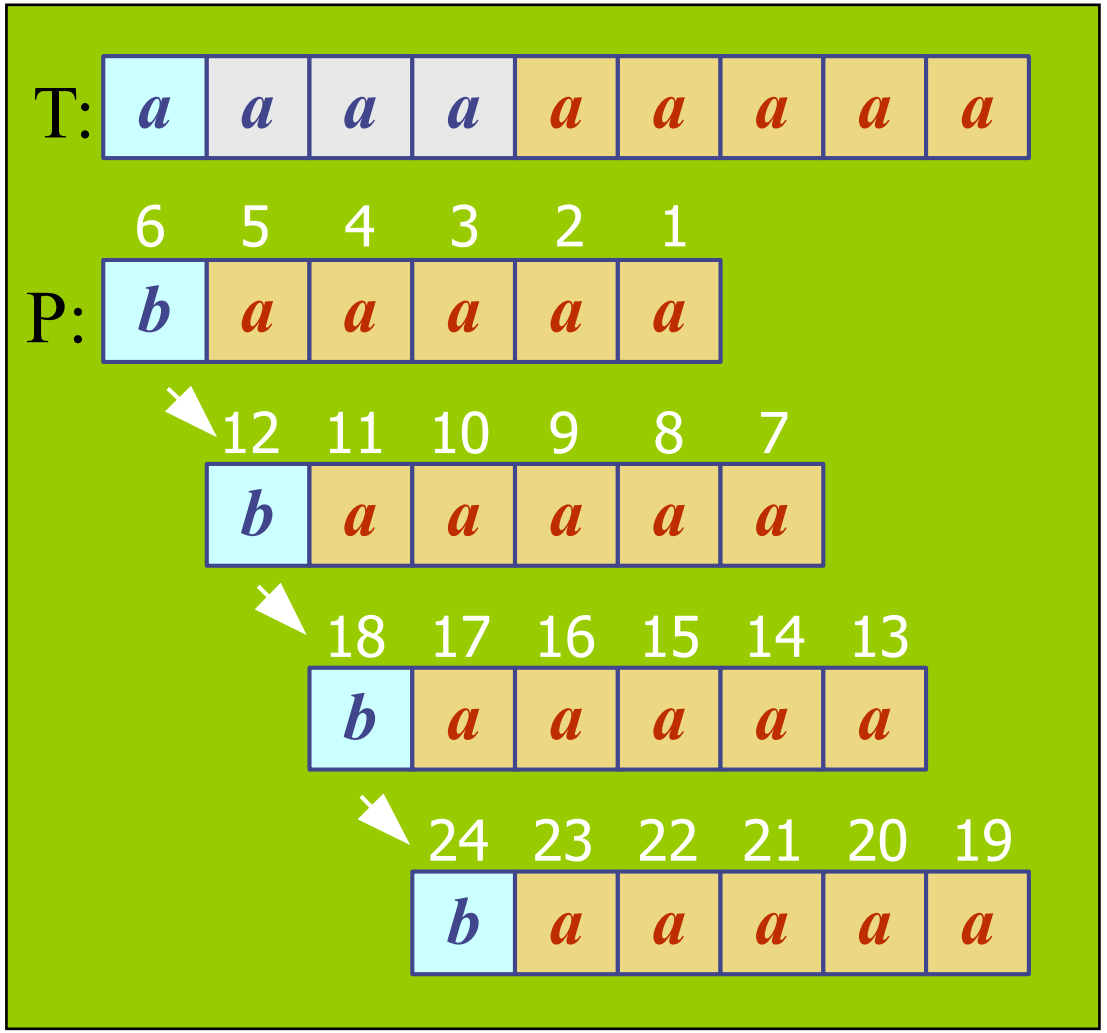
Zweites Boyer-Moore-Beispiel



x	*a*	*b*	*c*	*d*
L(*x*)	5	6	4	0

Analyse der Komplexität

- Schlechtester Fall
  - Beispiel
    - T: „aaaaaaaaaa…a“
    - P: „baa…a“
  - O(m·n + |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

Robert S. Boyer, J. Strother Moore: A fast string searching algorithm.
In: Communications of the ACM. Bd. 20, Nr. 10, S. 762–772, **1977**

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

Karp, Richard M.; Rabin, Michael O.: „Efficient randomized pattern-matching algorithms“.
IBM Journal of Research and Development 31 (2), 249–260, **1987**

Hash-Funktion für Rabin-Karp-Algorithmus

- Für ein Zeichen
  - $h(k)=code(k)\cdot q$ mit k z.B ASCII-Code des betrachteten Zeichens und q eine Primzahl
- Für eine Zeichenkette
  - $h'(k_1..k_m) = h(k_1) + ... + h(k_m)$
- 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:
  - $h'(1234) = 1\cdot5 + 2\cdot5 + 3\cdot5 + 4\cdot5 = 50$

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:

  $h'(k_2..k_m\ k_{m+1}) = h'(k_1\ k_2\ ...\ k_m) - k_1 \cdot q + k_{m+1} \cdot q$

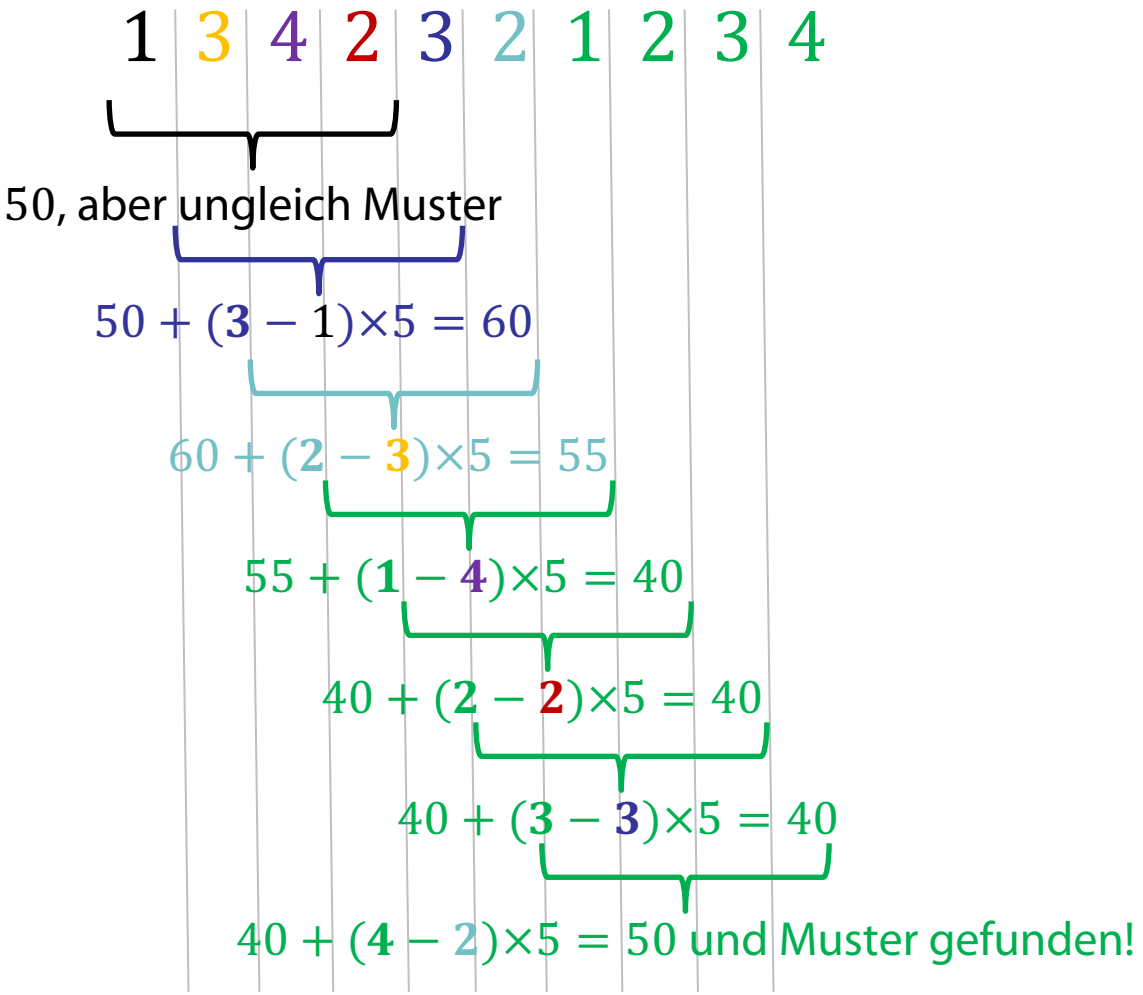
  $= h'(k_1\ k_2\ ...\ k_m) + (k_{m+1} - k_1) \cdot q$
- **Man beachte:** Konstanter Aufwand pro Zeichen

Beispiel: Suche nach der Hash-Signatur

Muster 1234

q=5

h(1234) = 50



Rabin-Karp-Algorithmus

```
function rk_search(t, p, hf :: Function = hash)
  n = length(t)
  m = length(p)
  h_p = hf(p)
  for i = 1:(n - m) + 1
    h = hf(t[i:i + m - 1])
    if h == h_p
      if t[i:i + m - 1] == p
        return i                    # erfolgreiche Suche
      end
    end
  end
  return -1                         # erfolglose Suche
end
```

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

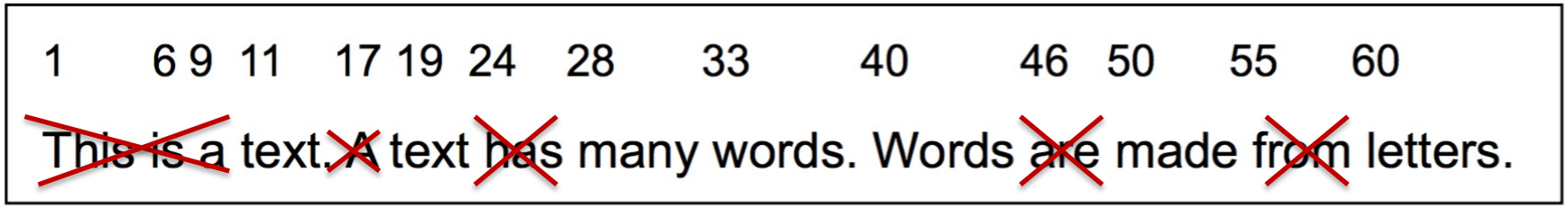
Karp, Richard M.; Rabin, Michael O.: „Efficient randomized pattern-matching algorithms“. IBM Journal of Research and Development 31 (2), 249–260, **1987**

Zusammenfassung

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

Indexstrukturen für eindimensionale Daten

Wiederholung: Tries



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

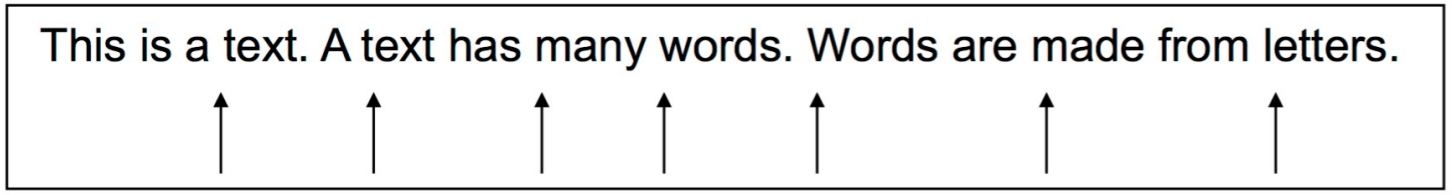
1	2	3	4
This is a text.	A text has many	words. Words are	made from letters.

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.

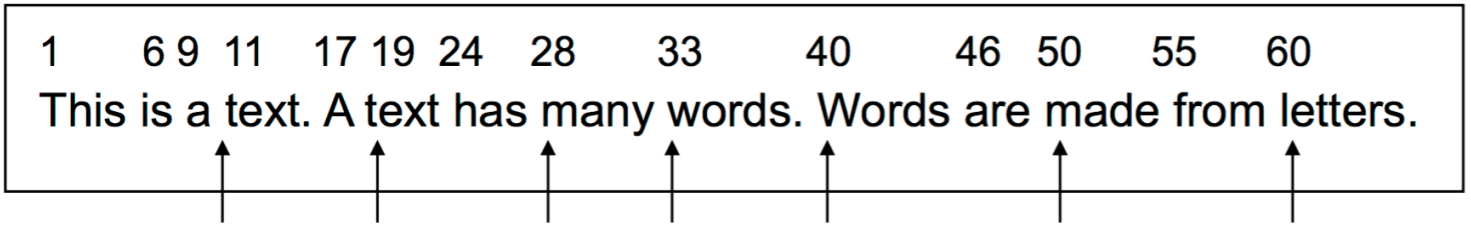


Suffixe:

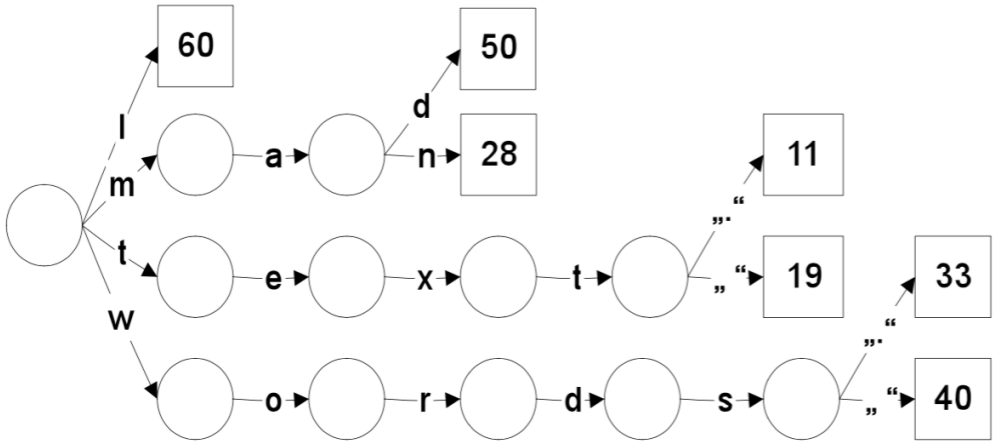
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
letters.

Suffix-Tries

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



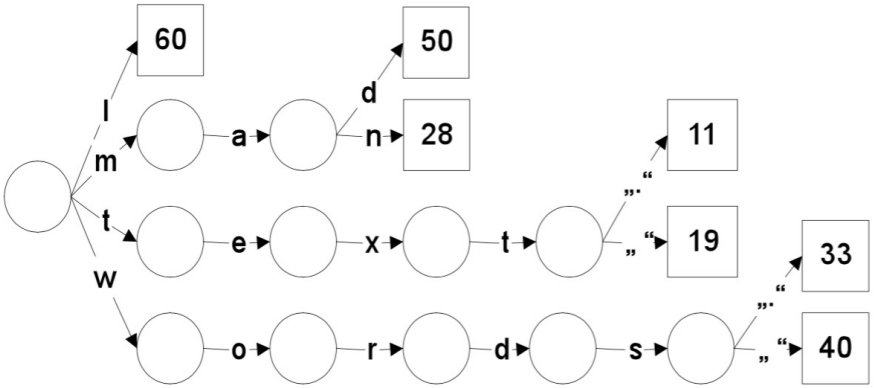
Suffix-Trie:



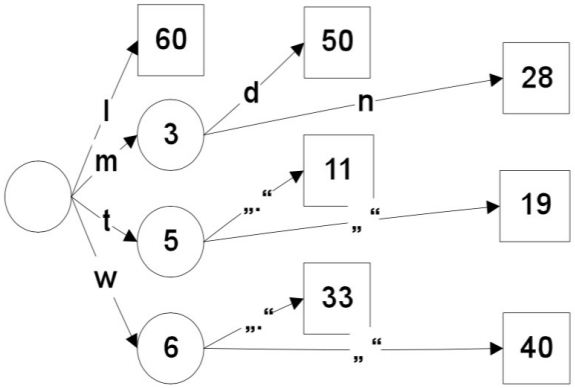
Patricia-Tries

- 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, prüfe Text bis dahin
3. Folge der Kante, die den Buchstaben an der aktuellen Position akzeptiert

Suffix-Bäume

- Konstruktion: O(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

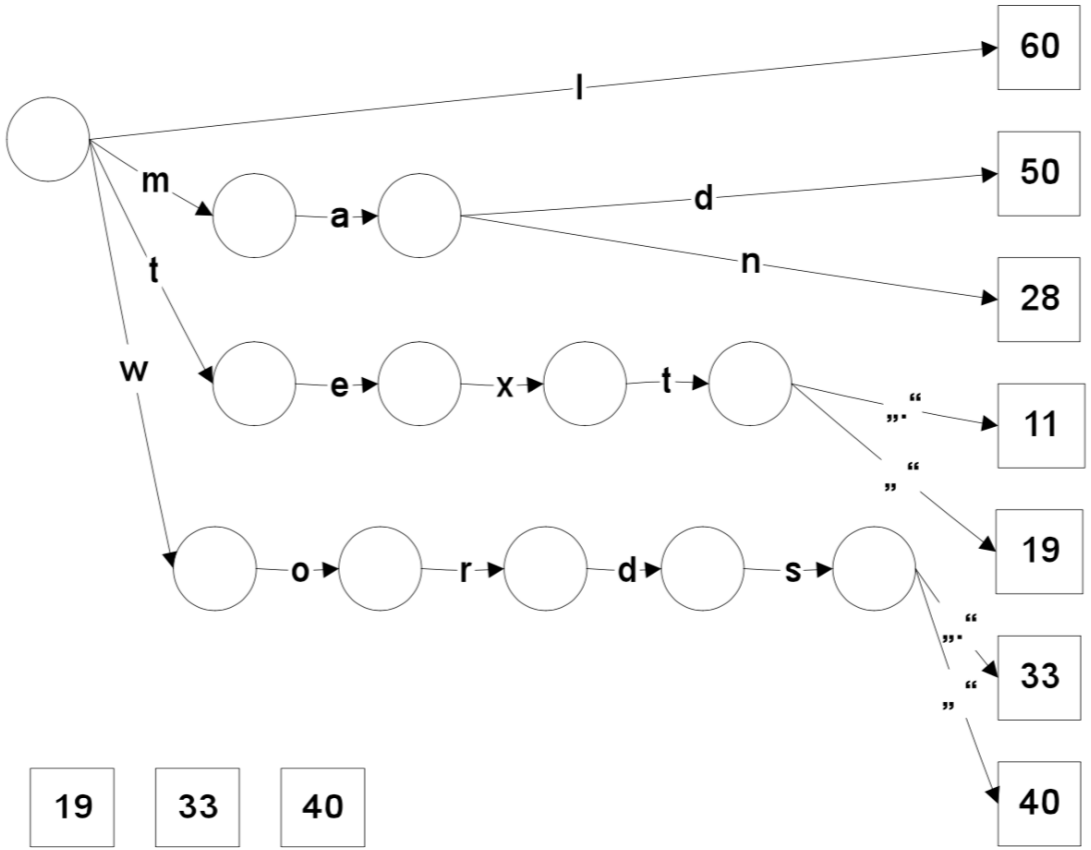
Suffix-Felder (Aufbau naiv)

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

Suffix-Trie,

Lexikographisch

sortiert:



Suffix-Array:

60	50	28	11	19	33	40

Beispiel

Suffixsortierung → Binäre Suche

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

lenght(SA) = 11

Beispiel: Suche „sip“ in „mississippi“

Beispiel von Karsten Klein, Uni Dortmund,
Vorlesung Algorithmen und Datenstrukturen

pos	L	7	4	1	0	M	8	6	3	5	R
A	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
sip < ssi				p	i				i		p
sip > i				p	p						i
sip > pi				i	p						
					i						

Suche: sip

pos	10	7	4	1	0	L	8	6	M	5	R
A	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

pos	10	7	4	1	0	L	8	M	R	5	2
A	i	i	i	i	m	p	p	s	s	s	s
		p	s	s	i	i	p	i	s	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		i
				p	i				i		p
				p	p						p
				i	p						i
					i						

1 Vorkommen

Mehrere Vorkommen?

Vorkommen benachbart!

sip = sip

pos	10	7	4	1	0	9	8	6	L	M	R
A	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						

Suche nach ssi

2 Vorkommen

Suche in Suffix-Feldern: Binärsuche

```
function find(p, A, pos)
  # Suchzeichenkette p, Suffix-Feld A, Positionen pos
  # liefert Position von p oder -1, falls p nicht gefunden
  l = 1; r = length(A)
  n = length(p)
  while l < r
    m = ceil(Int, (l+r) / 2)
    if p == A[m][begin:minimum((n, end))]
      return pos[m]
    elseif p < A[m][begin:minimum((n, end))]
      r = m
    else
      l = m
  end end
  return -1
end
```

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(Länge der Suchzeichenkette)
- Suffix-Felder: O(log(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**

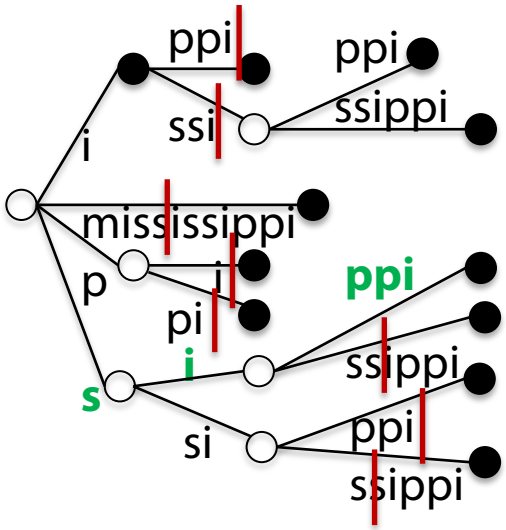
Approximativer Zeichenkettenabgleich

- *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≤2:



Abbruch wegen #Edits > 2

Berechnungen für gemeinsame Präfixe nur einmal!

H. Shang and T. H. Merrettal. "Tries for approximate string matching." *IEEE TKDE* 8.4: 540-547, 1996.

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]

Hier gibt es wieder einen Index 0 analog zum Kapitel dynamische Programmierung.

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

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

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

Buchstabe hinzufügen löschen ersetzen

$$O(nm)$$

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j] — Buchstabe stimmt überein!
  { 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

Buchstabe: hinzufügen löschen ersetzen

j

i

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

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j] Buchstabe stimmt überein!
  { 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

Buchstabe: hinzufügen löschen ersetzen

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							
b	2									
a	3									
b	4									
a	5									
c	6									

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j] Buchstabe stimmt überein!
  { 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

Buchstabe: hinzufügen löschen ersetzen

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

Beispiel

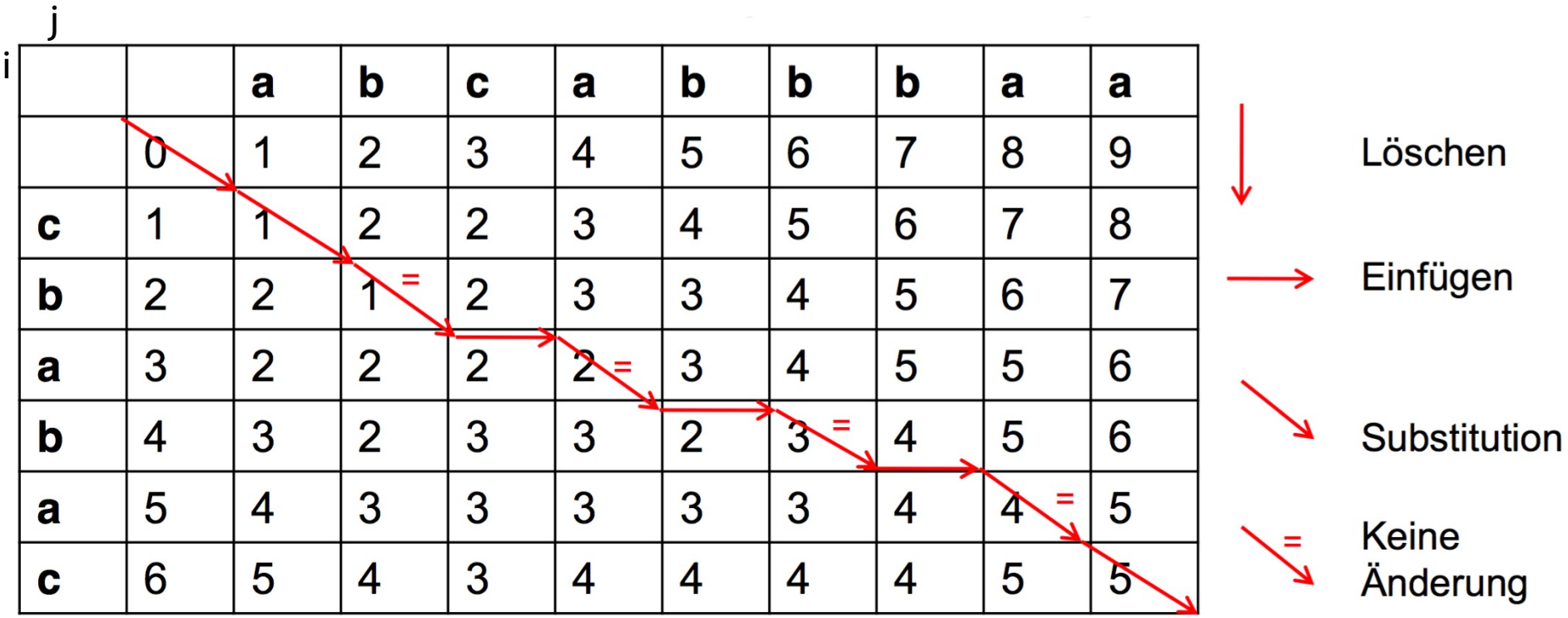
- C[0, j] = j
- C[i, 0] = i
- $$C[i,j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min\{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$$

Buchstabe stimmt überein!

Buchstabe: hinzufügen löschen ersetzen

i \ j		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

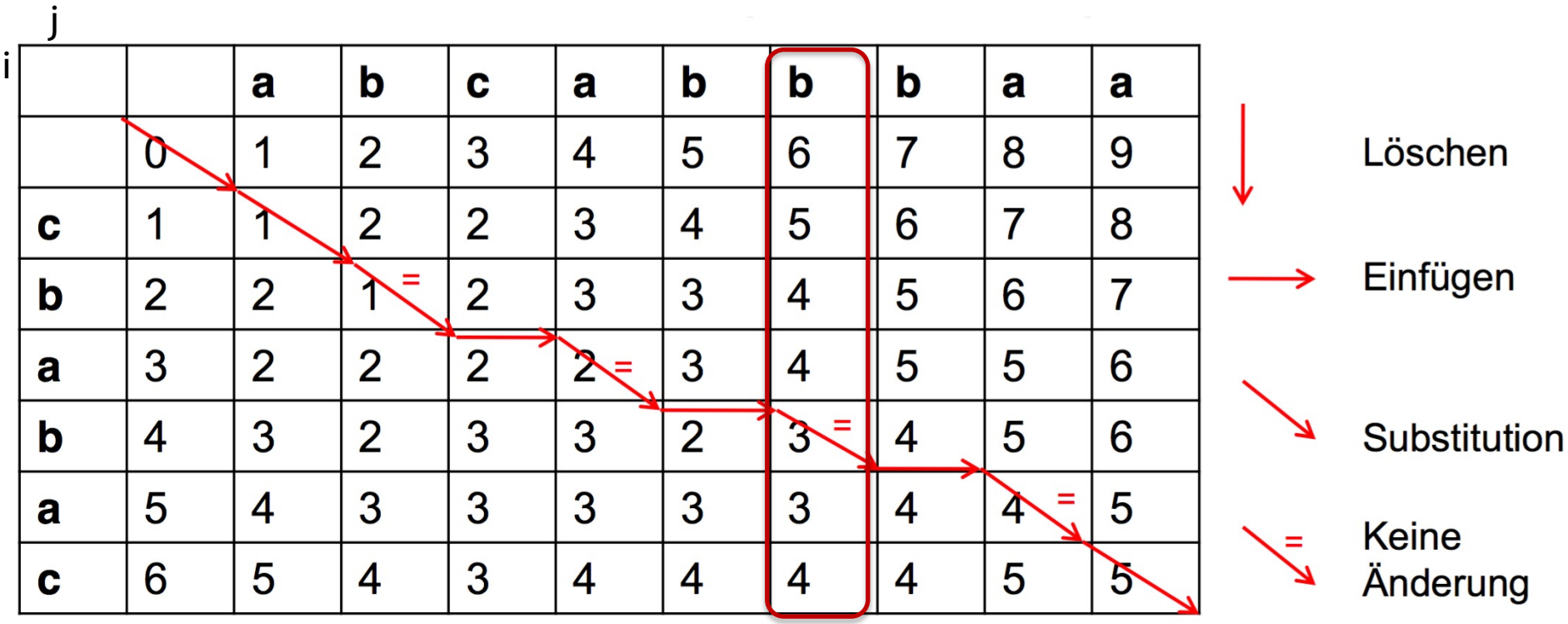
Beispiel



		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

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

Editierabstand immer >k, falls ganze Spalte>k



		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

Ganze Spalte >2
=> Abbruch falls k≤2 gefordert!
(Vgl. Approx.-suche in Trie)

Nicht jede Zelle der Matrix braucht berechnet zu werden (→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