# Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck Institut für Informationssysteme

Tanya Braun (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 (begrenztes Alphabet: A, C, G, T)



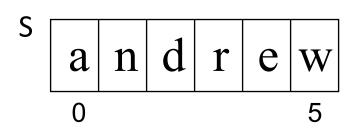
#### Teilzeichenkette, Präfix, Suffix

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



## Beispiele

• Teilzeichenkette S[1..3] = "ndr"

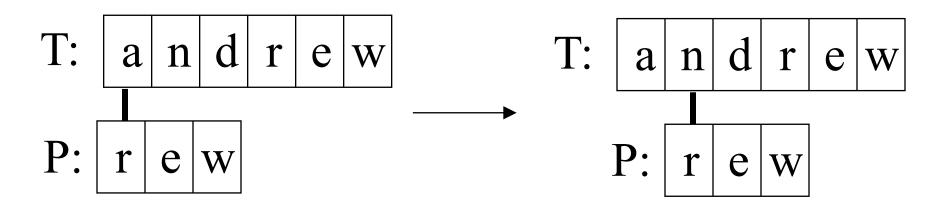


- Alle möglichen Präfixe von S:
  - "andrew", "andre", "and", "an", "a"
- Alle möglichen Suffixe von S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"



#### 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



**FOCUS DAS LEBEN** 

#### Brute-Force-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: "aaaaaaaaaaaaaaaaaaaaaa"

Muster: "aaaah"

- Das Muster wird an jeder Position im Text durchlaufen: O(n⋅m)
- Bester Fall für erfolglose Suche
  - Beispiel

• Text: "aaaaaaaaaaaaaaaaaa"

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)



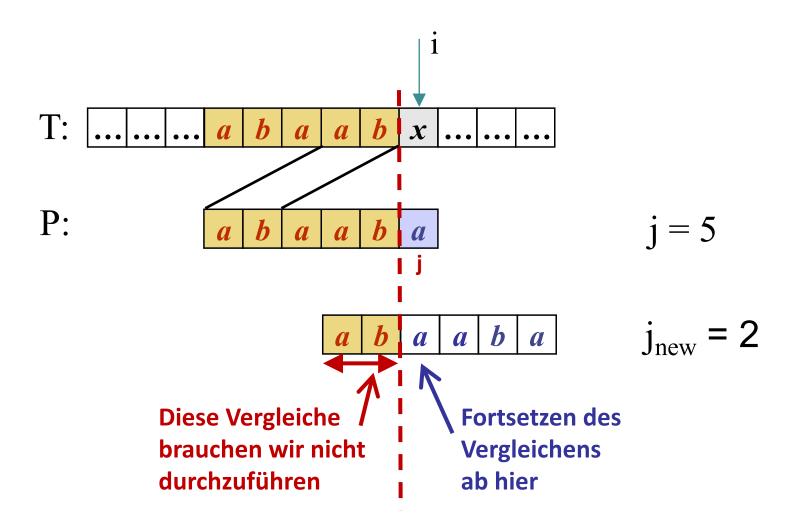
#### Weitere Analyse

- 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



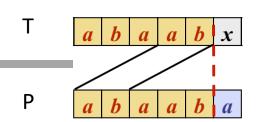


## Knuth-Morris-Pratt-Algorithmus (KMP)

- 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?



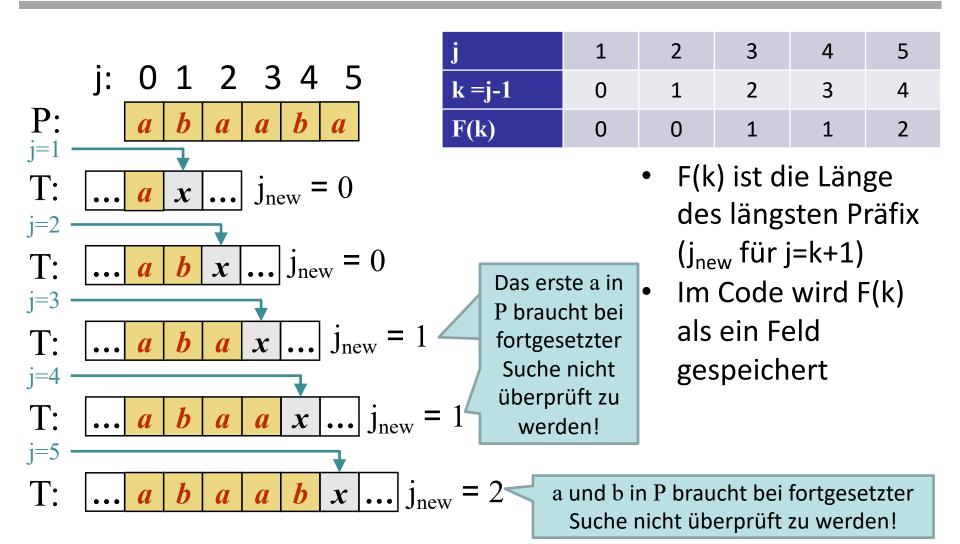
#### KMP Fehlerfunktion



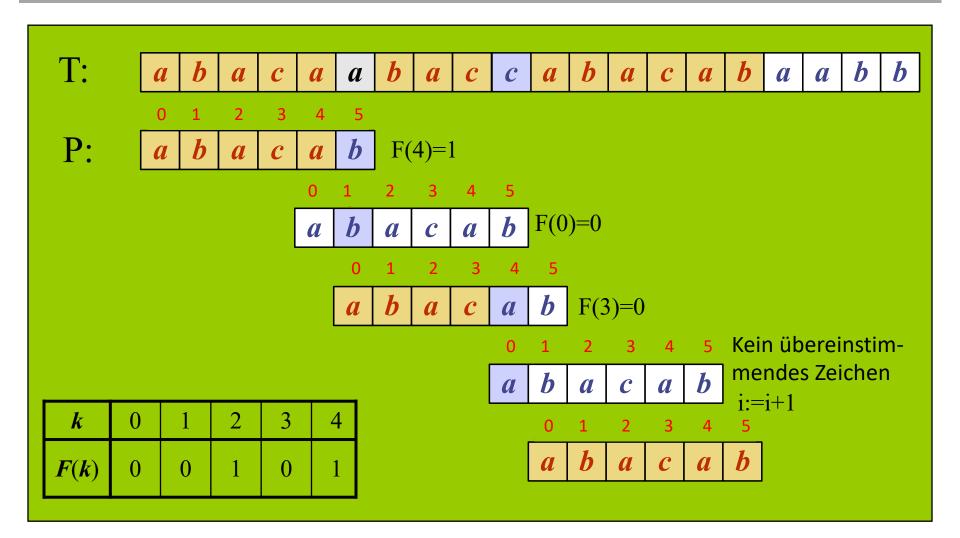
- KMP verarbeitet das Muster vor, um Übereinstimmungen zwischen den Präfixen des Musters mit sich selbst zu finden
- j = Position der Ungleichheit in P
- k = Position vor der Ungleichheit (k = j-1).
- Die sog. Fehlerfunktion F(k) ist definiert als die Länge des längsten Präfixes von P[0..k], welcher 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



## Beispiel





#### Unterschiede im Code zum Brute-Force-Algorithmus

- Knuth-Morris-Pratt Ansatz modifiziert den Brute-Force-Algorithmus
- Falls keine Übereinstimmung bei P[j]
   (d.h. P[j] ≠ T[i]), dann
   k := j-1;
   j := F(k); // berechne neues j



#### Verfahren Knuth-Morris-Pratt

```
Procedure 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

```
Р
function computeF(pattern: String): Array[] of IN
    F:= <0,...,0>: Array [0..length(pattern)-1 - 1] of IN
    F[0] := 0
                                                                                              Länge
    m := length(F)
    i := 1; j := 0
                                                       Position:
                                                                   0
                                                                         2
                                                                            3
                                                                                  5
                                                                      1
                                                                               4
                                                                                     6
                                                                                        7
                                                                                           8
    while i < m do
                                                       Muster:
                                                                            b
                                                                                             a 9
                                                                                     b
                                                                                           b
                                                                   а
                                                                         а
                                                                               C
                                                                                  а
                                                                                        а
      if pattern[j] = pattern[i] then
                                                     Präfix (0..0):
                                                                                                 0
                                                                                                      0
        // j+1 Zeichen stimmen überein
                                                     Präfix (0..1):
                                                                                                 0
        F[i] := i + 1
                                                     Präfix (0..2):
        i := i + 1; j := j + 1
                                                                                                 1
                                                                         а
      else if j > 0 then
                                                                                                      3
                                                     Präfix (0..3):
                                                                                                 2
                                                                   а
              // j folgt dem übereinstim- 0 14
                                                     Präfix (0..4):
                                                                                                 0
                                                                                                      4
              // menden Präfix
                                                                                                      5
                                                     Präfix (0..5):
                                                                                                 1
                                                                                  а
               i := F[i - 1]
                                                                                                      6
            else //keine Übereinstimmung \frac{1}{2}
                                                     Präfix (0..6):
                                                                      b
                                                                   а
                                                                                     b
                                                                                  а
                                                     Präfix (0..7):
                                                                   а
                                                                      b
                                                                         а
                                                                                                 3
                                                                                     b
                                                                                  а
               F[i] := 0
                                                                                                      8
                                                     Präfix (0..8):
                                                                      b
                                                                         а
                                                                            b
                                                                                     b
                                                                                           b
                                                                                                 4
                                                                                        а
                                                                                  а
               i := i + 1
    return F
                                                                                                     F[]
```

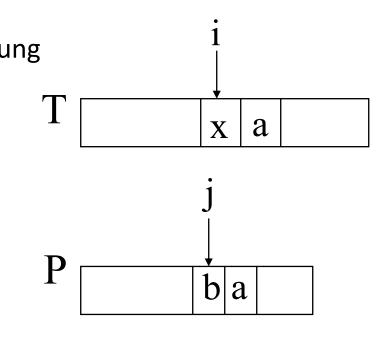
#### Analyse des Knuth-Morris-Pratt-Algorithmus

- 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]≠T[i])
    - 3 Fälle...

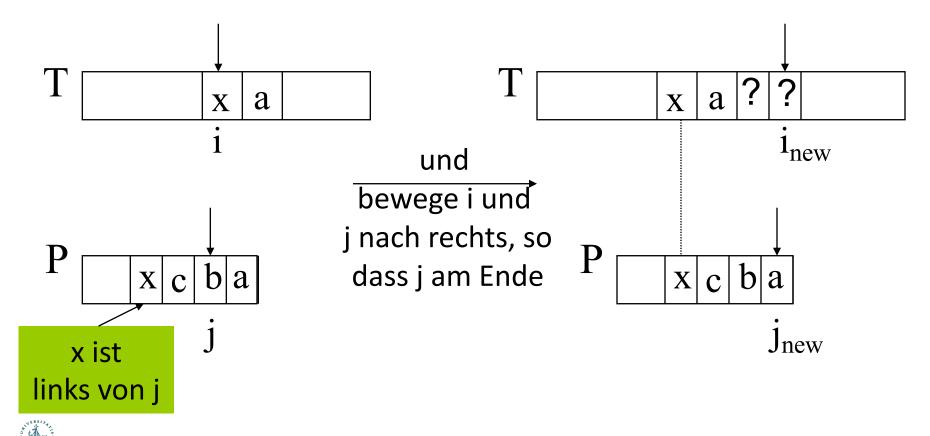




#### Fall 1: P enthält x nur links von j

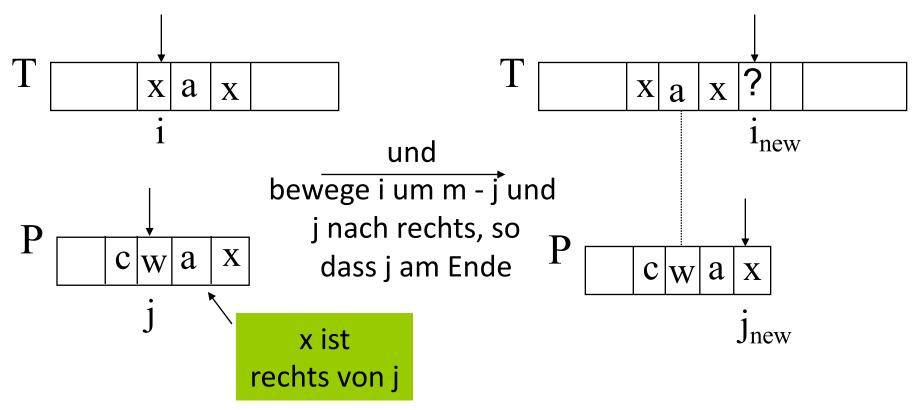
JNIVERSITÄT ZU LÜBECK INSTITUT FÜR INFORMATIONSSYSTEME

 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]

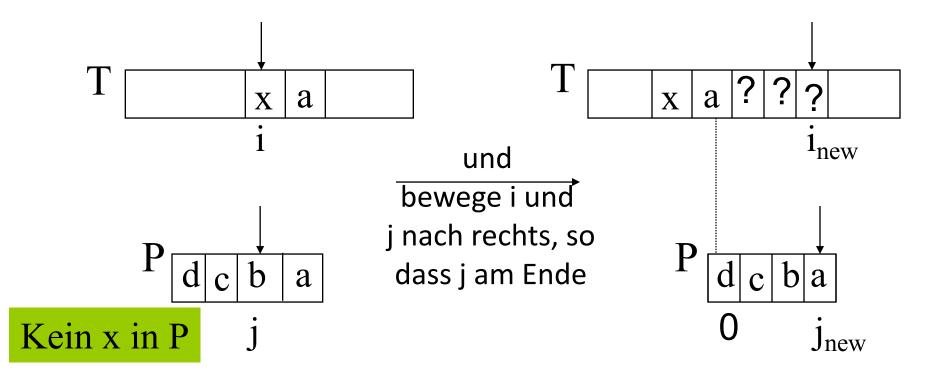


m = Länge des Musters



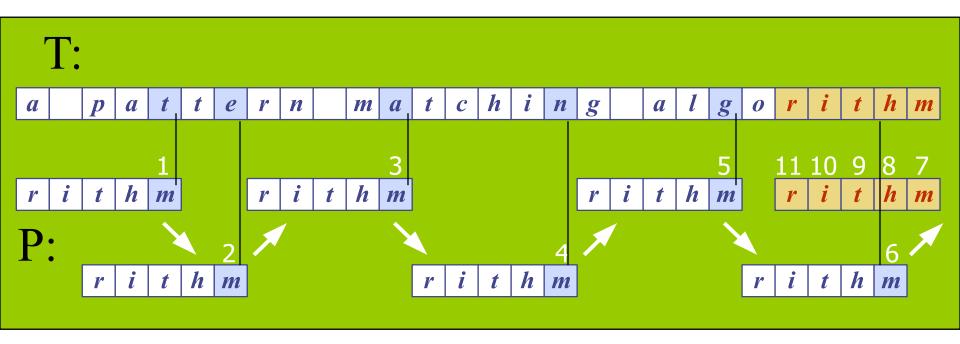
# Fall 3: Falls Fall 1 und 2 nicht anzuwenden sind (x ist *nicht* in P enthalten)

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





# 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
  - -1, falls solch ein Index nicht existiert
- Implementationen
  - L ist meist in einem Feld der Größe | A | gespeichert



# Beispiel L



X	a	b	C	d
L(x)	4	5	3	-1

L speichert Indexe von P

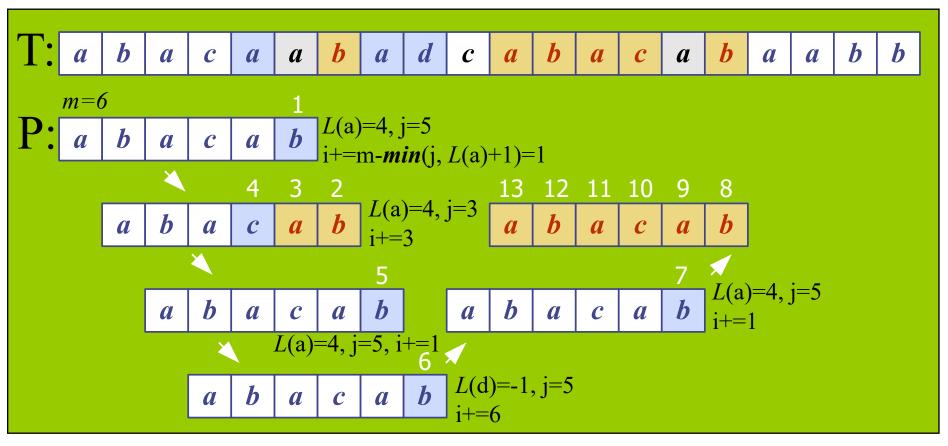


#### Funktion des letzten Vorkommens

```
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



x	а	b	c	d
L(x)	4	5	3	-1

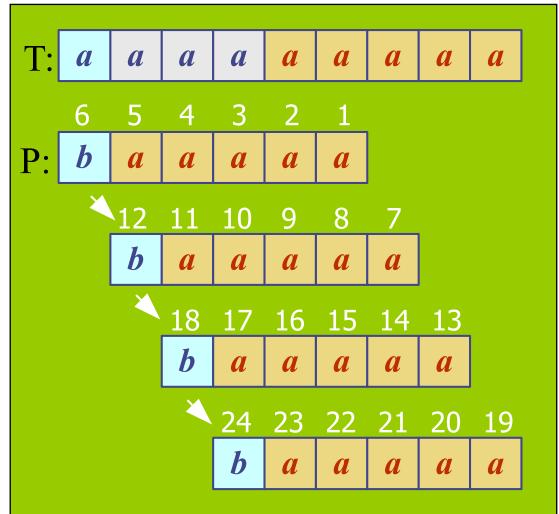


#### Boyer-Moore-Verfahren

```
procedure 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
   i := m - 1
   repeat
     if pattern[j] = text[i] then
       if i = 0 then
          return i // Suche erfolgreich
       else // Spiegeltechnik
         i := i -1; j := j -1
     else // Zeichensprungtechnik
       lo := L[text[i]]; i := i + m - min(j, lo+1); j := m - 1
   until j \le n - 1
   return -1 // Suche erfolglos
```

## Analyse der Komplexität

- Schlechtester Fall
  - Beispiel
    - T: "aaaaaaaaaa...a"
    - P: "baa...a"
  - $O(m \cdot n + |A|)$
  - 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|)



#### 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)=code(k)·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) = 1.5 + 2.5 + 3.5 + 4.5 = 50



## Suche nach der Hash-Signatur

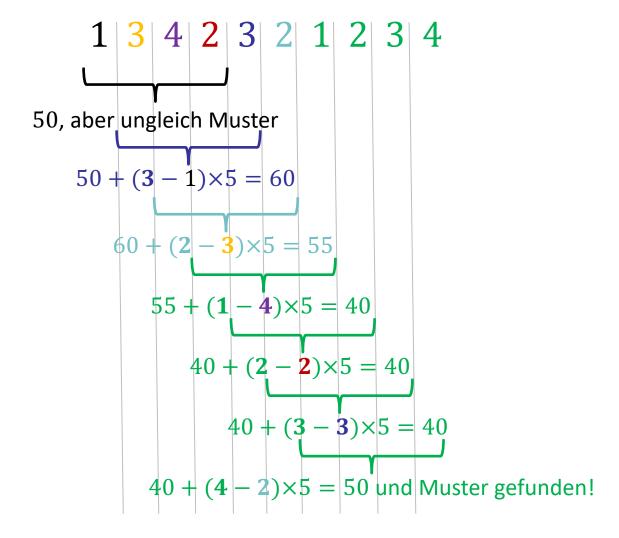
- Einmaliges Durchlaufen des zu durchsuchenden Textes und Vergleich der aktualisierten Hash-Signatur mit Hash-Signatur des Musters
- Hash-Signatur kann iterativ gebildet werden:

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

• Man beachte: Konstanter Aufwand pro Zeichen

# Beispiel: Suche nach der Hash-Signatur

Muster 1234





#### 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 · m)
    - Hash-Signaturen stimmen immer überein, auch bei keinem Treffer
- Insgesamt O(n + m) im Best/Average Case und O(n · m) im schlimmsten Fall



# Variante des Rabin-Karp-Algorithmus für Suche nach vielen Mustern

- Alle Hash-Signaturen (für Muster bis zu einer gegebenen maximalen Länge) können mit einem Durchlauf durch den Text in eine geeignete Datenstruktur z.B. eine Hashtabelle abgelegt werden
- Anschließende Suche nach vielen Mustern schnell möglich
  - im durchschnittlichen Fall in konstanter Zeit pro Muster
- Aber: Großer Speicherbedarf!



# 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 69 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.

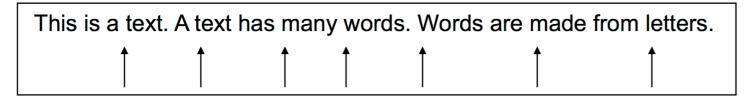
Terme	Vorkommen
Letters	4
Made	4
Many	2
Text	1, 2
words	3

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



#### Suffix-Bäume

- Indexpunkte k\u00f6nnen Wortanf\u00e4nge oder alle Zeichenkettenpositionen sein.
- Text ab Position: Suffix.



#### Suffizes:

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.

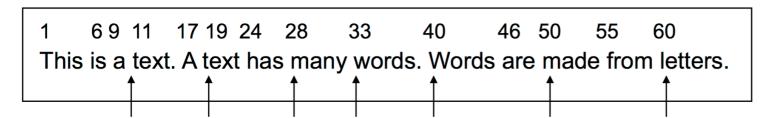
made from letters.

letters.

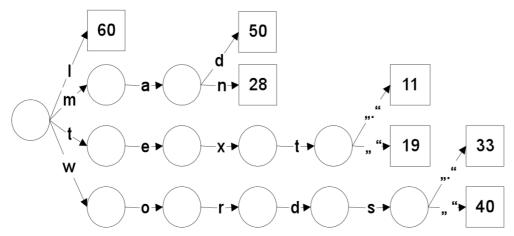


#### **Suffix-Tries**

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



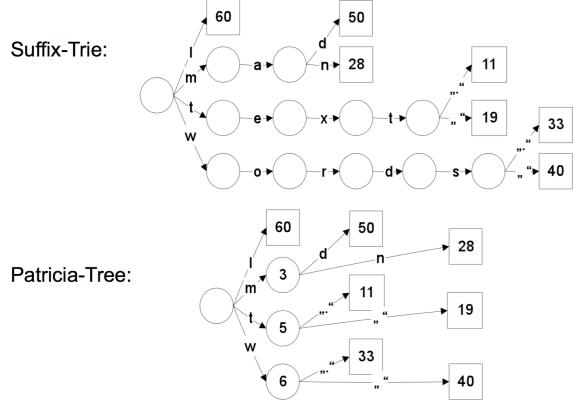
Suffix-Trie:





#### Patricia-Tries

• Ersetze interne Knoten mit nur einer ausgehenden Kante durch "Überleseknoten", beschrifte sie mit der nächsten zu beachtenden Textposition.



#### 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.



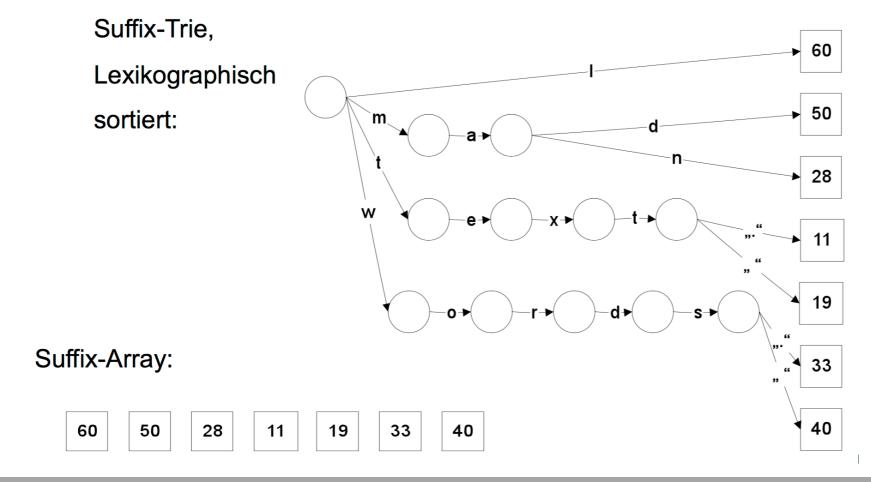
#### Suffix-Bäume

- Konstruktion: O(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.

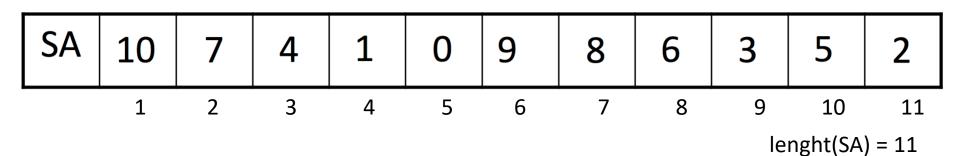


#### Suche in Suffix-Feldern: Binärsuche

```
procedure find(p, A): Integer
  // Suchzeichenkette p, Suffix-Feld A
  // liefert Position von p in A oder -1, falls p nicht in A
  s := 1
  e := length(A)
   while s < e do
     m := [(e + s) / 2] // nach oben runden
     if p = Text(A[m]) then
       return m
     if p < Text(A[m])
      then e := m
      else s := m
   return -1
```

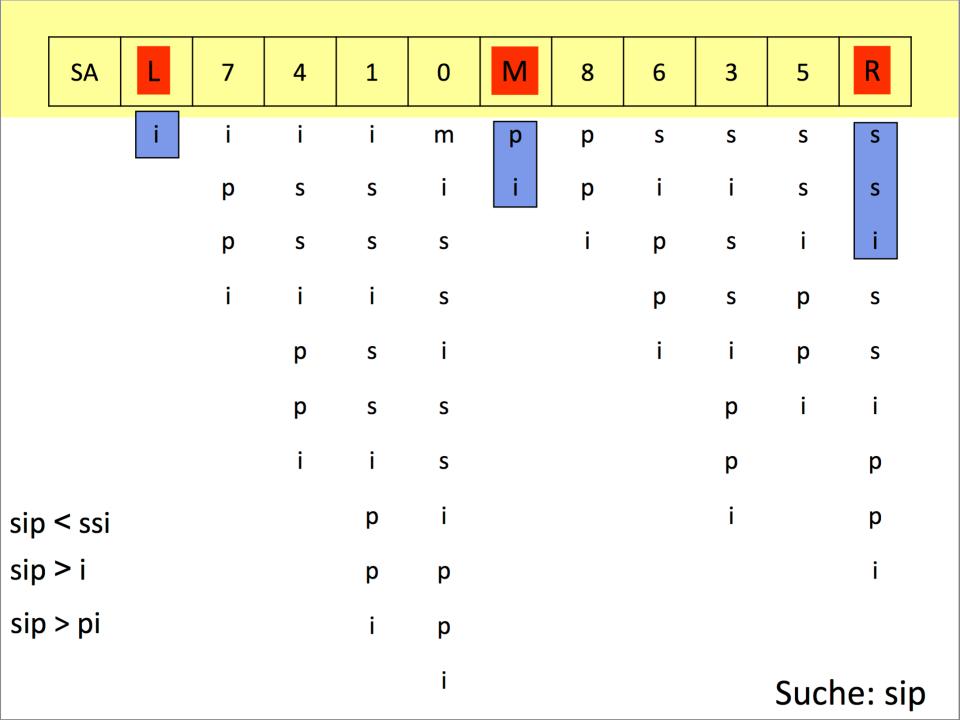


## Suffixsortierung → Binäre Suche



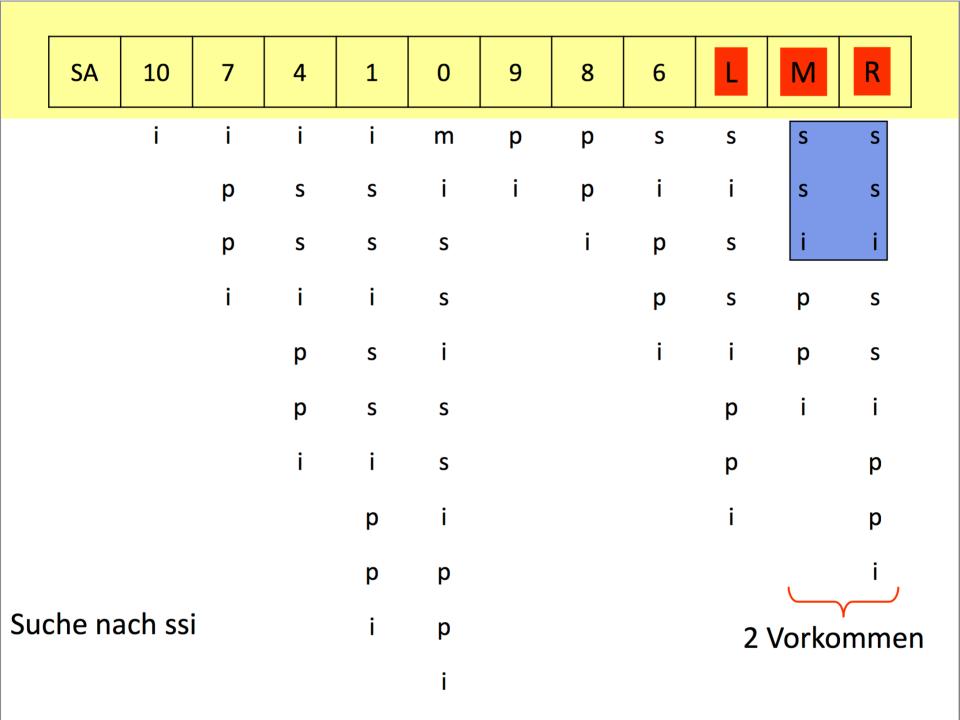
Beispiel: Suche "sip" in "mississippi"

51



	SA	10	7	4	1	0	L	8	6	M	5	R
		i	i	i	i	m	р	р	S	S	S	S
			р	S	S	i	i	р	i	i	S	S
			р	S	S	S		i	p	S	i	i
			i	i	i	S			p	S	р	S
				p	S	i			i	i	р	S
				p	S	S				p	i	i
				i	i	S				p		р
					р	i				i		р
					р	р						i
sip	< sis				i	p						
						i						

	SA	10	7	4	1	0	L	8	M	R	5	2		
		i	i	i	i	m	р	р	S	S	S	S		
			p	S	S	i	i	p	i	i	S	S		
			p	S	S	S		i	р	S	i	i		
			i	i	i	S			p	S	p	S		
				р	S	i			. i	i	р	S		
				р	S	S	4.1.		<b>С</b>	р	i	i		
				i	i	S	1 V	orkom	nmen	p		р		
					р	i				i		p		
					p	p	Me	hrere	Vorko	mme	n?	i		
sip	= sip				i	p i	Vorkommen benachbart!							



### Suche: Analyse der Aufwände

- 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 I durch u

 $\rightarrow$  Augo ersetze g durch D

→ AuDo lösche o

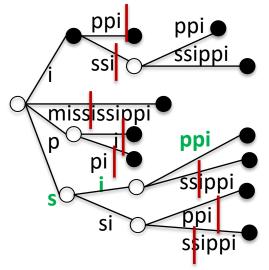
 $\rightarrow$  AuD



### Approximativer Zeichenkettenabgleich

- Suche in y Unterzeichenketten, die Editierabstand von höchstens k von x haben
  - Effizient in 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!



### 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] = \{C[i-1, j-1], wenn x[i] = y[j] \text{ Buchstabe stimmt überein! } 1+min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, sonst Buchstabe: hinzufügen löschen ersetzen
<math>O(nm)
```



```
    C[0, j] = j
    C[i, 0] = i
    C[i-1, j-1], wenn x[i] = y[j] Buchstabe stimmt überein!
    C[i, j] = C[i-1, j-1], C[i-1, j-1], C[i-1, j-1], sonst Buchstabe: hinzufügen löschen ersetzen
```

		а	b	С	а	b	b	b	а	а
	0	1	2	3	4	5	6	7	8	9
С	1	1								
b	2									
а	3									
b	4									
а	5									
С	6									

```
• C[0, j] = j

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

Buchstabe: hinzufügen löschen ersetzen

		а	b	С	а	b	b	b	а	а
	0	1	2	3	4	5	6	7	8	9
С	1	1	2							
b	2									
а	3									
b	4									
а	5									
С	6									

```
• C[0, j] = j
```

Buchstabe: hinzufügen löschen ersetzen

		а	b	С	а	b	b	b	а	а
	0	1	2	3	4	5	6	7	8	9
С	1	1	2	2						
b	2									
а	3									
b	4									
а	5									
С	6									

• 
$$C[0, j] = j$$

Buchstabe: hinzufügen löschen ersetzen

		а	b	С	а	b	b	b	а	а
	0	1	2	3	4	5	6	7	8	9
С	1	1	2	2	3	4	5	6	7	8
b	2	2	1	2	3	3	4	5	6	7
а	3	2	2	2	2	3	4	5	5	6
b	4	3	2	3	3	2	3	4	5	6
а	5	4	3	3	3	3	3	4	4	5
С	6	5	4	3	4	4	4	4	5	5

											1	
		а	b	С	а	b	b	b	а	а		
	0	1	2	3	4	5	6	7	8	9		Löschen
С	1	1	2	2	3	4	5	6	7	8	V	
b	2	2	1=	2	3	3	4	5	6	7	$\longrightarrow$	Einfügen
а	3	2	2	2	2=	3	4	5	5	6		
b	4	3	2	3	3	2	3 =	4	5	6	A	Substitution
а	5	4	3	3	3	3	3	4	4 =	5	\ <u>-</u>	Keine
С	6	5	4	3	4	4	4	4	5	5	7	Änderung

cbabac -> ababac -> abcabbac -> abcabbac -> abcabbbac ->



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

											•	
		а	b	С	а	b	b	b	а	а		
	0	1	2	3	4	5	6	7	8	9		Löschen
С	1	1	2	2	3	4	5	6	7	8	V	
b	2	2	1	2	3	3	4	5	6	7	$\longrightarrow$	Einfügen
а	3	2	2	2	2=	3	4	5	5	6		
b	4	3	2	3	3	2	3=	4	5	6	A	Substitution
а	5	4	3	3	3	3	3	4	4 =	5		Keine
С	6	5	4	3	4	4	4	4	5	5	7	Änderung

Ganze Spalte >2 => Abbruch falls k≤2 gefordert!

(Vgl. Approx.-suche in Trie)

Weiterhin: **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** 

#### Suffix-Bäume und Suffix-Felder

- Zeichenkettensuche mit Suffix-Bäumen und –Feldern kein Problem, aber jede Zeichenkettenposition muss indexiert werden. Speicherbedarf empirisch 120-240% der Textsammlung.
- Aktuelle Themen in der Forschung: Suche in komprimiertem Text, Suche mit komprimierten Indexdateien

