

# Algorithmen 2

## Kapitel 11: Stringology Teil 1 – Strings Sortieren, Mustersuche und Suffix-Bäume

Florian Kurpicz

The slides are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License © ⓘ ⓘ: [www.creativecommons.org/licenses/by-sa/4.0](http://www.creativecommons.org/licenses/by-sa/4.0) | commit e9153df compiled at 2021-11-17-10:38

# Stringology

## Algorithmen und Datenstrukturen für Strings

- (String-)Sortierung
- Mustersuche
- Text-Kompression
- Dokumenten-Retrieval
- ...

## Texte sind Überall

- Wikipedia , Bücher , Nachrichten 
- strukturierte Informationen (z.B. XML) 
- Code   
- DNS , Proteine 

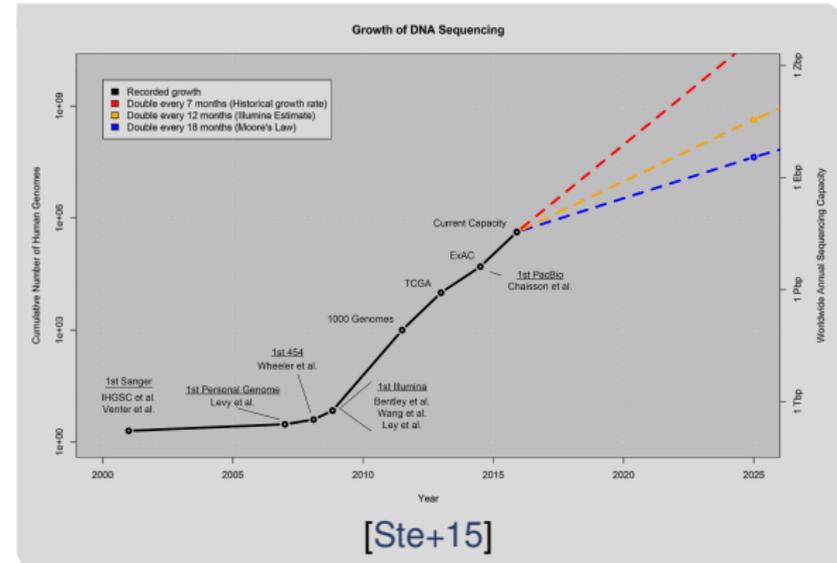
# Stringology

## Algorithmen und Datenstrukturen für Strings

- (String-)Sortierung
- Mustersuche
- Text-Kompression
- Dokumenten-Retrieval
- ...

## Texte sind Überall

- Wikipedia , Bücher , Nachrichten 
- strukturierte Informationen (z.B. XML) 
- Code 
- DNS , Proteine 



# Stringology in Algorithmen 2

## Über die Veranstaltung

- die nächsten 3 Dienstagsvorlesungen
- bei Interesse Vorlesung **Text-Indexierung**
  - ❗ Wahlbereich/Vertiefung

## Über Mich

- Mail: [kurpicz@kit.edu](mailto:kurpicz@kit.edu)
- Sprechstunde: Montags 13:45–14:45 Uhr
- Betreuung Abschlussarbeit: Gerne Melden!

# Stringology in Algorithmen 2

## Über die Veranstaltung

- die nächsten 3 Dienstagsvorlesungen
- bei Interesse Vorlesung **Text-Indexierung**
  - ❗ Wahlbereich/Vertiefung

## Über Mich

- Mail: [kurpicz@kit.edu](mailto:kurpicz@kit.edu)
  - Sprechstunde: Montags 13:45–14:45 Uhr
  - Betreuung Abschlussarbeit: Gerne Melden!
- 
- 🚫 bedeutet nicht Prüfungsrelevant

# PINGO – Interaktiv an der Vorlesung teilnehmen



<https://pingo.scc.kit.edu/410273>

# Stringology-Notationen

## Definition: Text

- $\Sigma$  bezeichnet das **Alphabet**
- $T \in \Sigma^*$  ist ein Text
- $|T| = n$  ist die Länge des Textes
- $T = T[1]T[2] \dots T[n]$

# Stringology-Notationen

## Definition: Text

- $\Sigma$  bezeichnet das **Alphabet**
- $T \in \Sigma^*$  ist ein Text
- $|T| = n$  ist die Länge des Textes
- $T = T[1]T[2] \dots T[n]$

## Definition: Muster

- für einen Text  $T \in \Sigma^*$
- ist  $P \in \Sigma^*$  ein Muster
- $|P| = m$  ist die Länge des Musters
- $P = P[1]P[2] \dots P[m]$

# Stringology-Notationen

## Definition: Text

- $\Sigma$  bezeichnet das **Alphabet**
- $T \in \Sigma^*$  ist ein Text
- $|T| = n$  ist die Länge des Textes
- $T = T[1]T[2] \dots T[n]$

## Definition: Muster

- für einen Text  $T \in \Sigma^*$
- ist  $P \in \Sigma^*$  ein Muster
- $|P| = m$  ist die Länge des Musters
- $P = P[1]P[2] \dots P[m]$

## Definition: Alphabet-Arten

- **Alphabet konstanter Größe:** Endliche Menge deren Größe nicht von  $n$  abhängig ist
- **Ganzzahliges Alphabet:** Menge  $\{1, \dots, \sigma\}$  wobei  $\sigma$  in eine konstante Anzahl Wörter passt
- **Geordnetes Alphabet:** Zeichen können nur verglichen werden

# Stringology-Notationen

## Definition: Text

- $\Sigma$  bezeichnet das **Alphabet**
- $T \in \Sigma^*$  ist ein Text
- $|T| = n$  ist die Länge des Textes
- $T = T[1]T[2] \dots T[n]$

## Definition: Muster

- für einen Text  $T \in \Sigma^*$
- ist  $P \in \Sigma^*$  ein Muster
- $|P| = m$  ist die Länge des Musters
- $P = P[1]P[2] \dots P[m]$

## Definition: Alphabet-Arten

- **Alphabet konstanter Größe:** Endliche Menge deren Größe nicht von  $n$  abhängig ist
- **Ganzzahliges Alphabet:** Menge  $\{1, \dots, \sigma\}$  wobei  $\sigma$  in eine konstante Anzahl Wörter passt
- **Geordnetes Alphabet:** Zeichen können nur verglichen werden

## Definition: Menge an Strings

Sei  $S = \{s_1, \dots, s_k\}$  eine Menge an Strings, wobei

- $s_i \in \Sigma^*$  für alle  $k \in [1..k]$  mit
- Gesamtlänge  $N = \sum_{i=1}^k |s_i|$

# Strings Sortieren

a	b	e	y	a	n	c	e	\$				
a	b	a	n	d	o	n	\$					
a	b	d	u	c	t	i	o	n	\$			
e	n	f	o	r	c	e	r	\$				
e	n	e	r	v	a	t	e	\$				
a	b	h	o	r	r	e	n	t	\$			
e	n	f	e	e	b	l	e	m	e	n	t	\$
a	b	a	t	t	o	i	r	\$				
e	n	d	u	r	a	n	c	e	\$			
a	b	e	r	r	a	n	\$					
e	n	e	r	g	i	z	e	r	\$			

# Strings Sortieren

a b e y a n c e \$  
 a b a n d o n \$  
 a b d u c t i o n \$  
 e n f o r c e r \$  
 e n e r v a t e \$  
 a b h o r r e n t \$  
 e n f e e b l e m e n t \$  
 a b a t t o i r \$  
 e n d u r a n c e \$  
 a b e r r a n \$  
 e n e r g i z e r \$

a b a n d o n \$  
 a b a t t o i r \$  
 a b d u c t i o n \$  
 a b e r r a n \$  
 a b e y a n c e \$  
 a b h o r r e n t \$  
 e n d u r a n c e \$  
 e n e r g i z e r \$  
 e n e r v a t e \$  
 e n f e e b l e m e n t \$  
 e n f o r c e r \$

# Eindeutiges und Längstes Gemeinsames Präfix

⊥	a	b	a	n	d	o	n	\$					
3	a	b	a	t	t	o	i	r	\$				
2	a	b	d	u	c	t	i	o	n	\$			
2	a	b	e	r	r	a	n	\$					
3	a	b	e	y	a	n	c	e	\$				
2	a	b	h	o	r	r	e	n	t	\$			
0	e	n	d	u	r	a	n	c	e	\$			
2	e	n	e	r	g	i	z	e	r	\$			
4	e	n	e	r	v	a	t	e	\$				
2	e	n	f	e	e	b	l	e	m	e	n	t	\$
3	e	n	f	o	r	c	e	r	\$				

## Längstes Gemeinsames Präfix

- Längstes gemeinsames Präfix von
- zwei lex. aufeinander folgenden Strings

## Eindeutiges Präfix

- Anzahl  $d$  an Zeichen, die
- mindestens verglichen werden muss, um
- alle Strings zu sortieren

# Strings Sortieren: Most Significant Digit Radix Sort

a	b	e	y	a	n	c	e	\$				
a	b	a	n	d	o	n	\$					
a	b	d	u	c	t	i	o	n	\$			
e	n	f	o	r	c	e	r	\$				
e	n	e	r	v	a	t	e	\$				
a	b	h	o	r	r	e	n	t	\$			
e	n	f	e	e	b	l	e	m	e	n	t	\$
a	b	a	t	t	o	i	r	\$				
e	n	d	u	r	a	n	c	e	\$			
a	b	e	r	r	a	n	\$					
e	n	e	r	g	i	z	e	r	\$			

- zähle Zeichen auf aktueller Tiefe,
- teile Strings nach Zeichen auf,
- erhöhe Tiefe um eins und
- wiederhole für jedes Teil, bis Zeichen eindeutig

# Strings Sortieren: Most Significant Digit Radix Sort

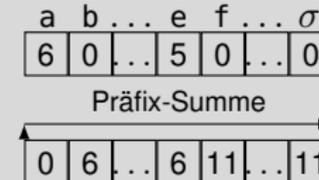
a	b	e	y	a	n	c	e	\$				
a	b	a	n	d	o	n	\$					
a	b	d	u	c	t	i	o	n	\$			
e	n	f	o	r	c	e	r	\$				
e	n	e	r	v	a	t	e	\$				
a	b	h	o	r	r	e	n	t	\$			
e	n	f	e	e	b	l	e	m	e	n	t	\$
a	b	a	t	t	o	i	r	\$				
e	n	d	u	r	a	n	c	e	\$			
a	b	e	r	r	a	n	\$					
e	n	e	r	g	i	z	e	r	\$			

- zähle Zeichen auf aktueller Tiefe,
- teile Strings nach Zeichen auf,
- erhöhe Tiefe um eins und
- wiederhole für jedes Teil, bis Zeichen eindeutig

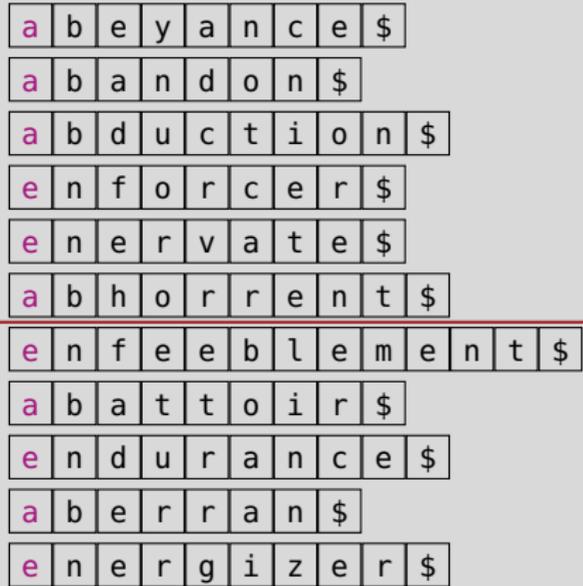
# Strings Sortieren: Most Significant Digit Radix Sort

a	b	e	y	a	n	c	e	\$				
a	b	a	n	d	o	n	\$					
a	b	d	u	c	t	i	o	n	\$			
e	n	f	o	r	c	e	r	\$				
e	n	e	r	v	a	t	e	\$				
a	b	h	o	r	r	e	n	t	\$			
e	n	f	e	e	b	l	e	m	e	n	t	\$
a	b	a	t	t	o	i	r	\$				
e	n	d	u	r	a	n	c	e	\$			
a	b	e	r	r	a	n	\$					
e	n	e	r	g	i	z	e	r	\$			

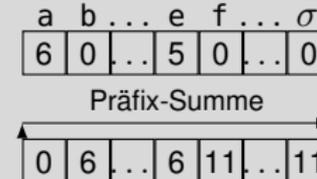
- zähle Zeichen auf aktueller Tiefe,
- teile Strings nach Zeichen auf,
- erhöhe Tiefe um eins und
- wiederhole für jedes Teil, bis Zeichen eindeutig



# Strings Sortieren: Most Significant Digit Radix Sort



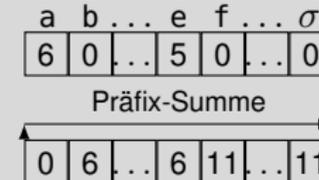
- zähle Zeichen auf aktueller Tiefe,
- teile Strings nach Zeichen auf,
- erhöhe Tiefe um eins und
- wiederhole für jedes Teil, bis Zeichen eindeutig



# Strings Sortieren: Most Significant Digit Radix Sort

a	b	e	y	a	n	c	e	\$				
a	b	a	n	d	o	n	\$					
a	b	d	u	c	t	i	o	n	\$			
e	n	f	o	r	c	e	r	\$				
e	n	e	r	v	a	t	e	\$				
a	b	h	o	r	r	e	n	t	\$			
<hr/>												
e	n	f	e	e	b	l	e	m	e	n	t	\$
a	b	a	t	t	o	i	r	\$				
e	n	d	u	r	a	n	c	e	\$			
a	b	e	r	r	a	n	\$					
e	n	e	r	g	i	z	e	r	\$			

- zähle Zeichen auf aktueller Tiefe,
- teile Strings nach Zeichen auf,
- erhöhe Tiefe um eins und
- wiederhole für jedes Teil, bis Zeichen eindeutig



- Laufzeit:  $O(d + \#Runden \cdot \sigma + k \lg \sigma)$
- Varianten: out-of-place und **in-place**
- einfach zu parallelisieren

# Strings Sortiere: Multikey Quicksort (1/2)

a l g o \$

c o d i n g \$

p l a n a r \$

**g** r a p h \$

a l g o r i t h m \$

g r e a t e r \$

m e m o r y \$

p a r a l l e l \$

h a m i l t o n \$

s i m p l e \$

a d d i t i o n \$

# Strings Sortiere: Multikey Quicksort (1/2)

a	l	g	o	\$					
c	o	d	i	n	g	\$			
p	l	a	n	a	r	\$			
g	r	a	p	h	\$				
a	l	g	o	r	i	t	h	m	\$
g	r	e	a	t	e	r	\$		
m	e	m	o	r	y	\$			
p	a	r	a	l	l	e	l	\$	
h	a	m	i	l	t	o	n	\$	
s	i	m	p	l	e	\$			
a	d	d	i	t	i	o	n	\$	

a	l	g	o	\$					
c	o	d	i	n	g	\$			
a	l	g	o	r	i	t	h	m	\$
a	d	d	i	t	i	o	n	\$	
<hr/>									
g	r	a	p	h	\$				
g	r	e	a	t	e	r	\$		
<hr/>									
p	l	a	n	a	r	\$			
m	e	m	o	r	y	\$			
p	a	r	a	l	l	e	l	\$	
h	a	m	i	l	t	o	n	\$	
s	i	m	p	l	e	\$			

# Strings Sortiere: Multikey Quicksort (1/2)

a l g o \$  
 c o d i n g \$  
 p l a n a r \$  
**g** r a p h \$  
 a l g o r i t h m \$  
 g r e a t e r \$  
 m e m o r y \$  
 p a r a l l e l \$  
 h a m i l t o n \$  
 s i m p l e \$  
 a d d i t i o n \$

**a** l g o \$  
 c o d i n g \$  
 a l g o r i t h m \$  
 a d d i t i o n \$  
 -----  
 g r a p h \$  
**g** r e a t e r \$  
 -----  
 p l a n a r \$  
 m e m o r y \$  
**p** a r a l l e l \$  
 h a m i l t o n \$  
 s i m p l e \$

a l g o \$  
 a l g o r i t h m \$  
**a** d d i t i o n \$  
 -----  
 c o d i n g \$  
 -----  
 g r **a** p h \$  
 g r e a t e r \$  
 -----  
 h a m i l t o n \$  
**m** e m o r y \$  
 -----  
 p **l** a n a r \$  
 p a r a l l e l \$  
 -----  
 s i m p l e \$

## Strings Sortieren: Multikey Quicksort (2/2)

### Function

*MulikeyQS*( $S = \{s_1, \dots, s_k\}, \ell \in \mathbb{N}$ ):

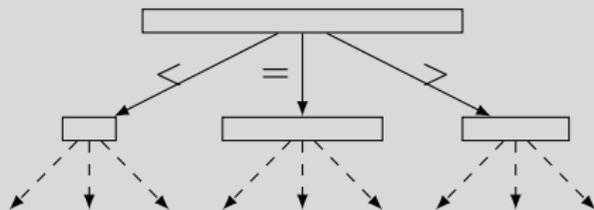
```
1 | if  $k \leq 1$  then  
2 | | return  $S$   
3 | Wähle  $p \in S$  mit Gleichverteilung  
4 | return Konkatination von  
   |  $\text{MulikeyQS}(\{s \in S: s[\ell] < p[\ell]\}, \ell) \cdot$   
   |  $\text{MulikeyQS}(\{s \in S: s[\ell] = p[\ell]\}, \ell + 1) \cdot$   
   |  $\text{MulikeyQS}(\{s \in S: s[\ell] > p[\ell]\}, \ell)$ 
```

# Strings Sortieren: Multikey Quicksort (2/2)

## Function

$MultikeyQS(S = \{s_1, \dots, s_k\}, \ell \in \mathbb{N}):$

- 1 **if**  $k \leq 1$  **then**
- 2     | **return**  $S$
- 3     Wähle  $p \in S$  mit Gleichverteilung
- 4 **return** Konkatination von  
      $MultikeyQS(\{s \in S: s[\ell] < p[\ell]\}, \ell) \cdot$   
      $MultikeyQS(\{s \in S: s[\ell] = p[\ell]\}, \ell + 1) \cdot$   
      $MultikeyQS(\{s \in S: s[\ell] > p[\ell]\}, \ell)$



# Strings Sortieren: Multikey Quicksort (2/2)

## Function

$MultikeyQS(S = \{s_1, \dots, s_k\}, \ell \in \mathbb{N}):$

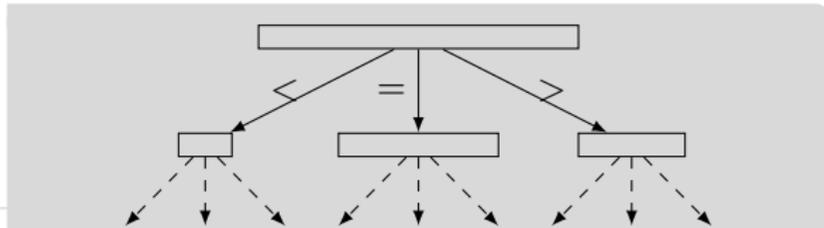
```

1  | if  $k \leq 1$  then
2  |   | return  $S$ 
3  |   | Wähle  $p \in S$  mit Gleichverteilung
4  |   | return Konkatination von
   |   |  $MultikeyQS(\{s \in S : s[\ell] < p[\ell]\}, \ell) \cdot$ 
   |   |  $MultikeyQS(\{s \in S : s[\ell] = p[\ell]\}, \ell + 1) \cdot$ 
   |   |  $MultikeyQS(\{s \in S : s[\ell] > p[\ell]\}, \ell)$ 

```

## Laufzeitanalyse (Skizze)

- $s[\ell] = p[\ell]$ 
  - $e[\ell]$  wird nicht mehr betrachtet, da
  - $\ell$  um eins erhöht wird
  - nur eindeutiges Präfixe werden verglichen
- $s[\ell] \neq p[\ell]$ 
  - $s$  ist in einer der Mengen  $S_{<}$  oder  $S_{>}$ , die die Größe  $k/2$  habe **i** mit optimaler Pivot-Wahl
  - nach  $O(\lg k)$  Schritten ist  $s$  korrekt sortiert



# Strings Sortieren: Multikey Quicksort (2/2)

## Function

$MultikeyQS(S = \{s_1, \dots, s_k\}, \ell \in \mathbb{N}):$

```

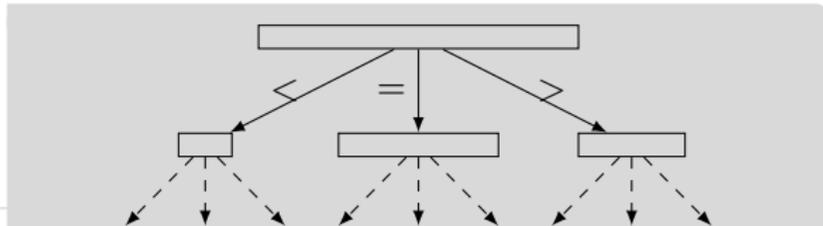
1  | if  $k \leq 1$  then
2  |   | return  $S$ 
3  |   | Wähle  $p \in S$  mit Gleichverteilung
4  |   | return Konkatination von
   |   |  $MultikeyQS(\{s \in S : s[\ell] < p[\ell]\}, \ell) \cdot$ 
   |   |  $MultikeyQS(\{s \in S : s[\ell] = p[\ell]\}, \ell + 1) \cdot$ 
   |   |  $MultikeyQS(\{s \in S : s[\ell] > p[\ell]\}, \ell)$ 

```

## Laufzeitanalyse (Skizze)

- $s[\ell] = p[\ell]$ 
  - $e[\ell]$  wird nicht mehr betrachtet, da
  - $\ell$  um eins erhöht wird
  - nur eindeutiges Präfixe werden verglichen
- $s[\ell] \neq p[\ell]$ 
  - $s$  ist in einer der Mengen  $S_{<}$  oder  $S_{>}$ , die die Größe  $k/2$  habe **i** mit optimaler Pivot-Wahl
  - nach  $O(\lg k)$  Schritten ist  $s$  korrekt sortiert

- Laufzeit:  $O(k \lg k + N)$
- genauer:  $O(k \lg k + d)$
- Übung: **in-place** Variante



# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[i..n]$  ist ein **Suffix** von  $T$

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[i..n]$  ist ein **Suffix** von  $T$

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

## Vereinfachung durch Sentinel

Sei  $T$  ein Text der Länge  $n$  über dem Alphabet  $\Sigma$ .

- wir nehmen an, dass  $T[n] = \$$  mit
- $\$ \notin \Sigma$  und  $\$ < \alpha$  für alle  $\alpha \in \Sigma$ , damit

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[i..n]$  ist ein **Suffix** von  $T$

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

## Vereinfachung durch Sentinel

Sei  $T$  ein Text der Länge  $n$  über dem Alphabet  $\Sigma$ .

- wir nehmen an, dass  $T[n] = \$$  mit
- $\$ \notin \Sigma$  und  $\$ < \alpha$  für alle  $\alpha \in \Sigma$ , damit

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[i..n]$  ist ein **Suffix** von  $T$

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

## Vereinfachung durch Sentinel

Sei  $T$  ein Text der Länge  $n$  über dem Alphabet  $\Sigma$ .

- wir nehmen an, dass  $T[n] = \$$  mit
- $\$ \notin \Sigma$  und  $\$ < \alpha$  für alle  $\alpha \in \Sigma$ , damit
- kein Suffix Präfix eines anderen Suffixes ist

1	2	3	4	5	6	7	8
a	b	b	a	a	b	b	a

- $T[1..n] = abbaabba$  und  $T[5..n] = abba$

# Weitere Stringology-Notationen

## Definition: Teilstring, Präfix und Suffix

Sei  $T = T[1]T[2] \dots T[n]$  ein Text der Länge  $n$ :

- $T[i..j] = T[i] \dots T[j]$  ist ein **Teilstring**,

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[1..i]$  ist ein **Präfix** und

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

- $T[i..n]$  ist ein **Suffix** von  $T$

a	b	b	a	a	b	b	a	\$
---	---	---	---	---	---	---	---	----

## Vereinfachung durch Sentinel

Sei  $T$  ein Text der Länge  $n$  über dem Alphabet  $\Sigma$ .

- wir nehmen an, dass  $T[n] = \$$  mit
- $\$ \notin \Sigma$  und  $\$ < \alpha$  für alle  $\alpha \in \Sigma$ , damit
- kein Suffix Präfix eines anderen Suffixes ist

1	2	3	4	5	6	7	8
a	b	b	a	a	b	b	a

- $T[1..n] = abbaabba$  und  $T[5..n] = abba$

## Definition: Präfix-Frei

Ein Text **Präfix-frei**, wenn kein Suffix Präfix eines anderen Suffixes ist

# Mustersuche

## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

# Mustersuche

## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n], P[1..m]$ ):

```
1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 | |   while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 | | |    $j = j + 1$ 
5 | | |   if  $j > m$  then
6 | | | |    $P$  kommt in  $T$  an Position  $i$  vor
7 | | |    $i = i + 1$ 
8 | |    $j = 1$ 
```

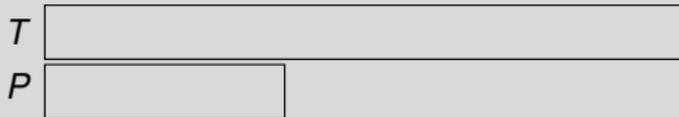
# Mustersuche

## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n], P[1..m]$ ):

```
1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 | |   while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 | | |    $j = j + 1$ 
5 | | |   if  $j > m$  then
6 | | | |    $P$  kommt in  $T$  an Position  $i$  vor
7 | | |    $i = i + 1$ 
8 | |    $j = 1$ 
```



# Mustersuche

## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function**  $NaiveMS(T[1..n], P[1..m]):$

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + 1$ 
8  |       |  $j = 1$ 

```



# Mustersuche

## Jetzt

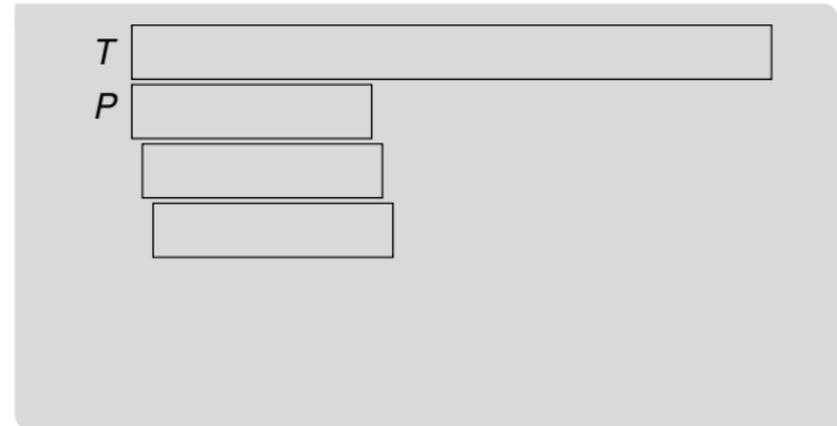
- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n], P[1..m]$ ):

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + 1$ 
8  |       |  $j = 1$ 

```



# Mustersuche

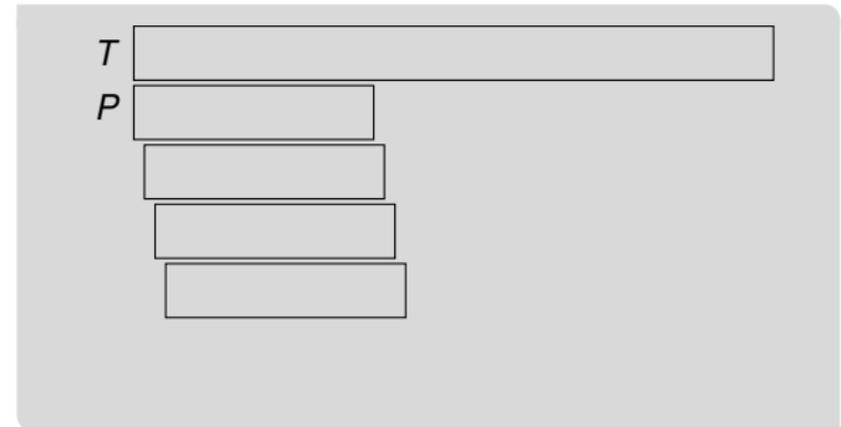
## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function**  $NaiveMS(T[1..n], P[1..m]):$

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + 1$ 
8  |       |  $j = 1$ 
  
```



# Mustersuche

## Jetzt

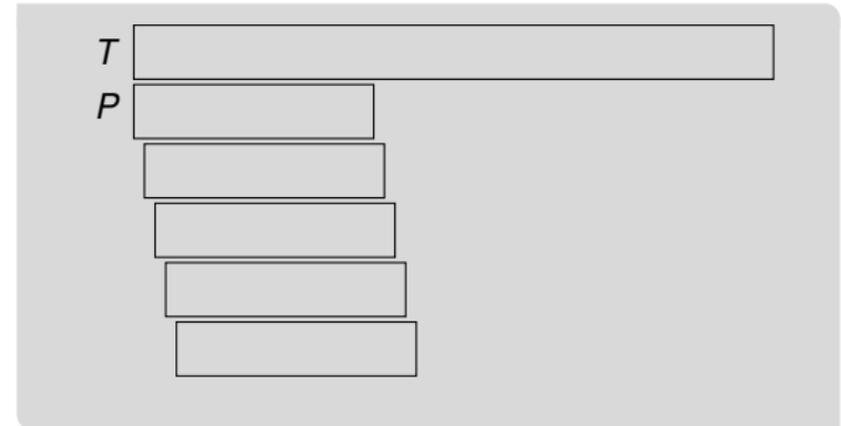
- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n]$ ,  $P[1..m]$ ):

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + 1$ 
8  |       |  $j = 1$ 

```



# Mustersuche

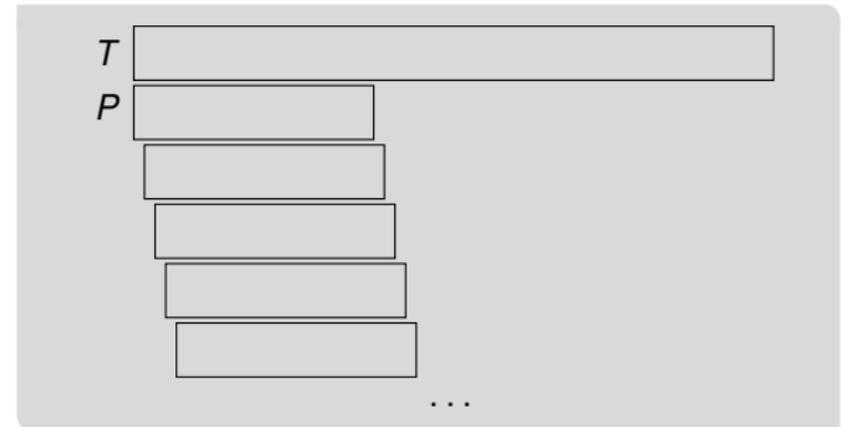
## Jetzt

- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n], P[1..m]$ ):

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |   |   |  $j = j + 1$ 
5  |   | if  $j > m$  then
6  |   |   |  $P$  kommt in  $T$  an Position  $i$  vor
7  |   |   |  $i = i + 1$ 
8  |   |   |  $j = 1$ 
  
```



# Mustersuche

## Jetzt

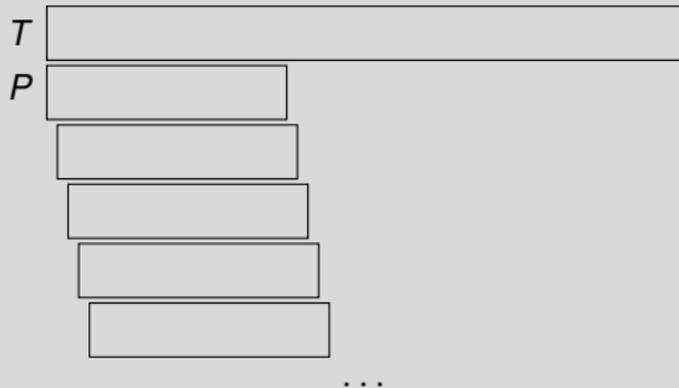
- finde alle Vorkommen von Muster  $P[1..m]$  in
- Text  $T[1..n]$

**Function** *NaiveMS*( $T[1..n], P[1..m]$ ):

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + 1$ 
8  |       |  $j = 1$ 

```



## Laufzeit

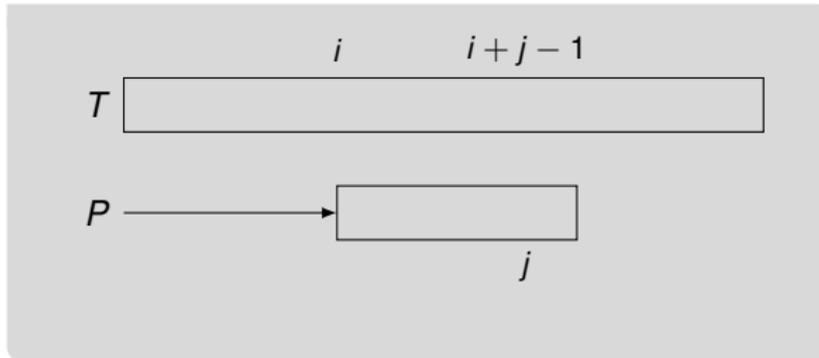
- Die Naive Mustersuche (*NaiveMS*) hat eine Laufzeit von  $O(n \cdot m)$

# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen

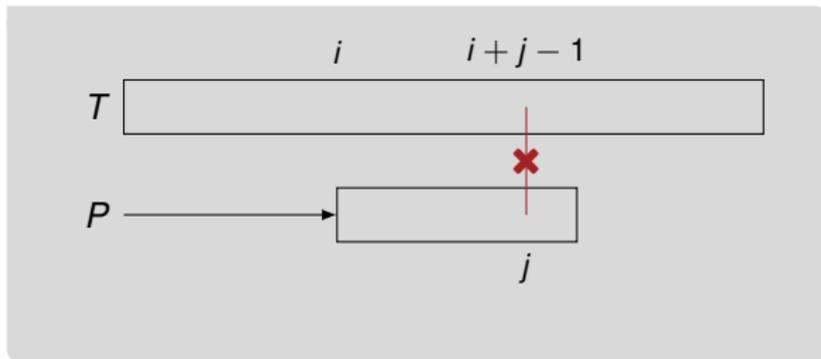
# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



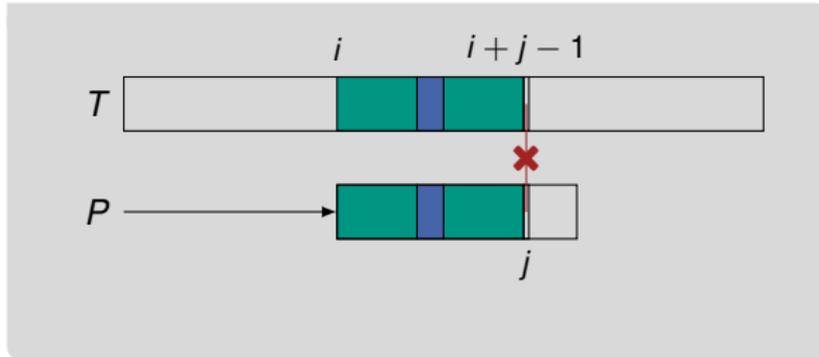
# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



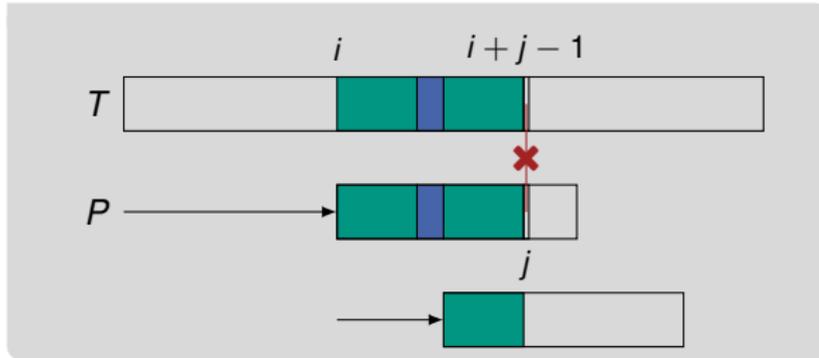
# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



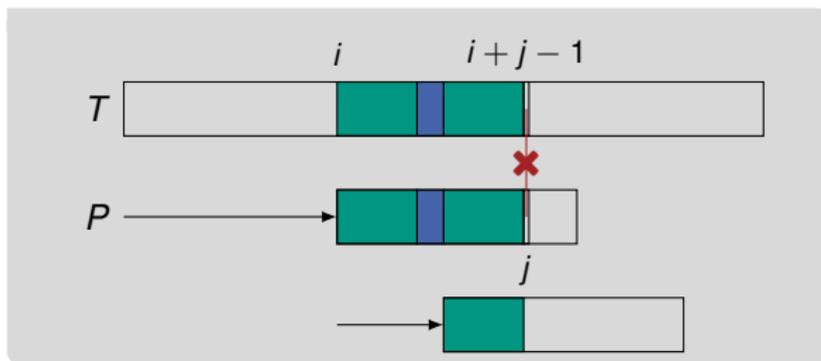
# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

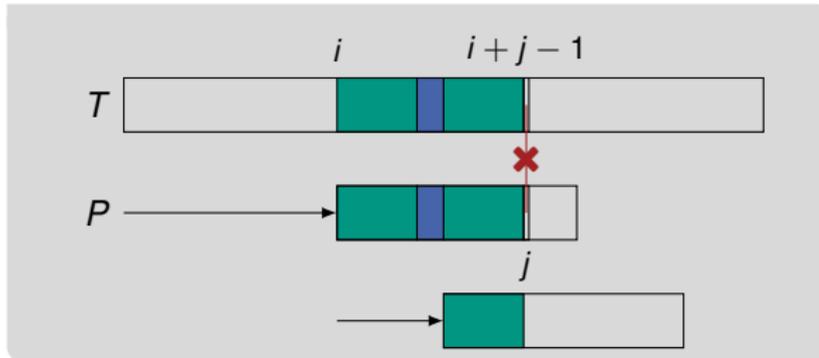
- Teile im Muster können sich wiederholen



- für Mismatch an Position  $j \leq m$  finde
- $\max\{k \in [1..j]: P[1..k] = P[j-k-1..j]\}$
- Muster an Position  $i+j-k-1$  verschieben

# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen

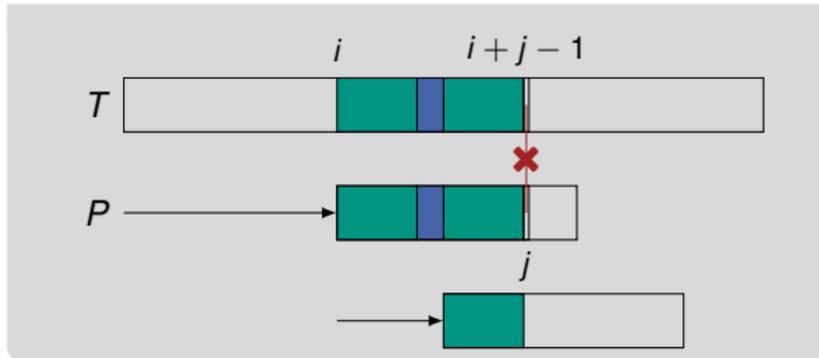


- $k$  für jedes  $j \leq m$  in  $border[1..m]$  vorberechnen
- $border[1] = -1$  und  $border[2] = 0$

- für Mismatch an Position  $j \leq m$  finde
- $\max\{k \in [1..j]: P[1..k] = P[j-k-1..j]\}$
- Muster an Position  $i+j-k-1$  verschieben

# Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



- für Mismatch an Position  $j \leq m$  finde
- $\max\{k \in [1..j) : P[1..k] = P[j-k-1..j]\}$
- Muster an Position  $i+j-k-1$  verschieben

- $k$  für jedes  $j \leq m$  in  $border[1..m]$  vorberechnen
- $border[1] = -1$  und  $border[2] = 0$

**Function**  $KMP(T[1..n], P[1..m], border[1..m])$ :

```

1   $i, j = 1$ 
2  while  $i \leq n - m + 1$  do
3    while  $j \leq m$  and  $T[i+j-1] = P[j]$  do
4       $j = j + 1$ 
5    if  $j > m$  then
6       $P$  kommt in  $T$  an Position  $i$  vor
7       $i = i + j - border[j] - 1$ 
8       $j = \max\{1, border[j] + 1\}$ 
  
```

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```
1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 | | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 | | |  $j = j + 1$ 
5 | | if  $j > m$  then
6 | | |  $P$  kommt in  $T$  an Position  $i$  vor
7 | | |  $i = i + j - border[j] - 1$ 
8 | | |  $j = \max\{1, border[j] + 1\}$ 
```

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```
1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 | | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 | | |  $j = j + 1$ 
5 | | if  $j > m$  then
6 | | |  $P$  kommt in  $T$  an Position  $i$  vor
7 | | |  $i = i + j - border[j] - 1$ 
8 | | |  $j = \max\{1, border[j] + 1\}$ 
```

### Laufzeit

Mustersuche mit KMP benötigt  $O(n + m)$  Zeit

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```

1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 |     |  $j = j + 1$ 
5 |     | if  $j > m$  then
6 |       |  $P$  kommt in  $T$  an Position  $i$  vor
7 |       |  $i = i + j - border[j] - 1$ 
8 |       |  $j = \max\{1, border[j] + 1\}$ 
  
```

### Laufzeit

Mustersuche mit KMP benötigt  $O(n + m)$  Zeit

### Beweis (Skizze)

- Schleife in Zeile 2 wird  $O(n)$  mal durchlaufen
- Schleife in Zeile 3 wird  $O(n)$  mal durchlaufen
  - kein Zeichen wird unnötig verglichen
  - verschiebe Muster immer so weit wie möglich

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```
1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 | | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 | | |  $j = j + 1$ 
5 | | if  $j > m$  then
6 | | |  $P$  kommt in  $T$  an Position  $i$  vor
7 | | |  $i = i + j - border[j] - 1$ 
8 | | |  $j = \max\{1, border[j] + 1\}$ 
```

### Laufzeit

Mustersuche mit KMP benötigt  $O(n + m)$  Zeit

### Beweis (Skizze)

- Schleife in Zeile 2 wird  $O(n)$  mal durchlaufen
- Schleife in Zeile 3 wird  $O(n)$  mal durchlaufen
  - kein Zeichen wird unnötig verglichen
  - verschiebe Muster immer so weit wie möglich
- Mustersuche dauert  $O(n)$  Zeit

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```

1 |  $i, j = 1$ 
2 | while  $i \leq n - m + 1$  do
3 |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4 |     |  $j = j + 1$ 
5 |   | if  $j > m$  then
6 |     |  $P$  kommt in  $T$  an Position  $i$  vor
7 |     |  $i = i + j - border[j] - 1$ 
8 |     |  $j = \max\{1, border[j] + 1\}$ 
  
```

### Laufzeit

Mustersuche mit KMP benötigt  $O(n + m)$  Zeit

### Beweis (Skizze)

- Schleife in Zeile 2 wird  $O(n)$  mal durchlaufen
- Schleife in Zeile 3 wird  $O(n)$  mal durchlaufen
  - kein Zeichen wird unnötig verglichen
  - verschiebe Muster immer so weit wie möglich
- Mustersuche dauert  $O(n)$  Zeit
- Berechnung von  $border$  benötigt  $O(m)$  Zeit
  - ① bleibt zu zeigen

## Mustersuche mit Knuth-Morris-Pratt (2/3)

**Function**  $KMP(T[1..n], P[1..m], border[1..m]):$

```

1  |  $i, j = 1$ 
2  | while  $i \leq n - m + 1$  do
3  |   | while  $j \leq m$  and  $T[i + j - 1] = P[j]$  do
4  |     |  $j = j + 1$ 
5  |     | if  $j > m$  then
6  |       |  $P$  kommt in  $T$  an Position  $i$  vor
7  |       |  $i = i + j - border[j] - 1$ 
8  |       |  $j = \max\{1, border[j] + 1\}$ 
  
```

### Laufzeit

Mustersuche mit KMP benötigt  $O(n + m)$  Zeit

### Beweis (Skizze)

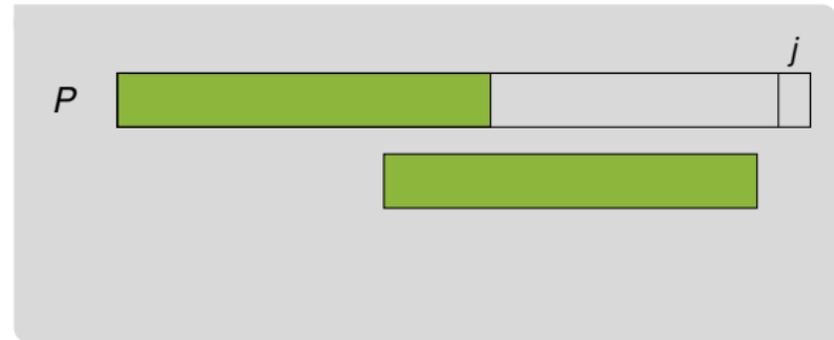
- Schleife in Zeile 2 wird  $O(n)$  mal durchlaufen
- Schleife in Zeile 3 wird  $O(n)$  mal durchlaufen
  - kein Zeichen wird unnötig verglichen
  - verschiebe Muster immer so weit wie möglich
- Mustersuche dauert  $O(n)$  Zeit
- Berechnung von  $border$  benötigt  $O(m)$  Zeit
  - ⓘ bleibt zu zeigen
- insgesamt  $O(n + m)$  Laufzeit

## Mustersuche mit Knuth-Morris-Pratt (3/3)

**Function** *KMP-Border*( $P[1..m]$ ):

```

1  | border[1] = -1
2  | i = border[1]
3  | for  $j = 2, \dots, m + 1$  do
4  |   | while  $i \geq 0$  and  $P[i + 1] \neq P[j - 1]$  do
5  |     | i = border[i + 1]
6  |     | i = i + 1
7  |     | border[j] = i
8  | return border
  
```

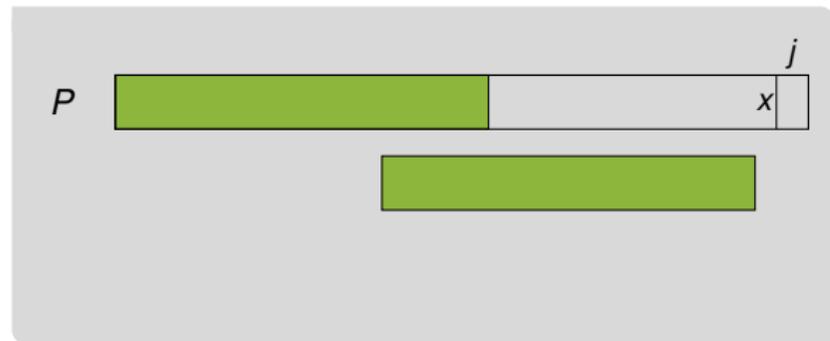


## Mustersuche mit Knuth-Morris-Pratt (3/3)

**Function** *KMP-Border*( $P[1..m]$ ):

```

1  | border[1] = -1
2  | i = border[1]
3  | for  $j = 2, \dots, m + 1$  do
4  |   | while  $i \geq 0$  and  $P[i + 1] \neq P[j - 1]$  do
5  |     | i = border[i + 1]
6  |     | i = i + 1
7  |     | border[j] = i
8  | return border
  
```



## Mustersuche mit Knuth-Morris-Pratt (3/3)

**Function** *KMP-Border*( $P[1..m]$ ):

```

1  | border[1] = -1
2  | i = border[1]
3  | for  $j = 2, \dots, m + 1$  do
4  |   | while  $i \geq 0$  and  $P[i + 1] \neq P[j - 1]$  do
5  |     | i = border[i + 1]
6  |     | i = i + 1
7  |     | border[j] = i
8  | return border
  
```

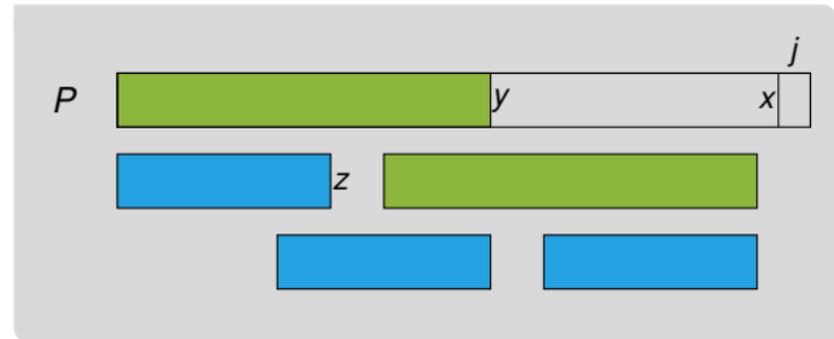


# Mustersuche mit Knuth-Morris-Pratt (3/3)

**Function** *KMP-Border*( $P[1..m]$ ):

```

1  |  $border[1] = -1$ 
2  |  $i = border[1]$ 
3  | for  $j = 2, \dots, m + 1$  do
4  |   | while  $i \geq 0$  and  $P[i + 1] \neq P[j - 1]$  do
5  |     |  $i = border[i + 1]$ 
6  |     |  $i = i + 1$ 
7  |     |  $border[j] = i$ 
8  | return  $border$ 
  
```

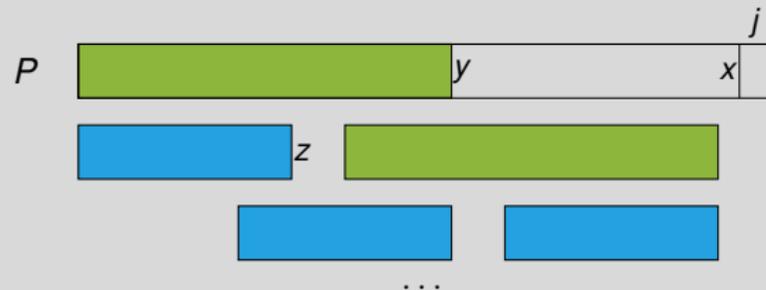


## Mustersuche mit Knuth-Morris-Pratt (3/3)

**Function** *KMP-Border*( $P[1..m]$ ):

```

1  border[1] = -1
2  i = border[1]
3  for j = 2, ..., m + 1 do
4    while i ≥ 0 and P[i + 1] ≠ P[j - 1] do
5      | i = border[i + 1]
6      i = i + 1
7      border[j] = i
8  return border
  
```



### Beweis (Skizze)

- Schleife in Zeile 3 für jedes Zeichen einmal
- Schleife in Zeile 4 wird
  - maximal  $m$ -mal reduziert
  - nur einmal pro Zeichen erhöht
- insgesamt  $O(m)$  Laufzeit

- schneller nach Muster suchen
- häufig nach unterschiedlichen Mustern Suchen
- $O(m + n)$  Zeit ist zu langsam

# Text-Indizes

- schneller nach Muster suchen
- häufig nach unterschiedlichen Mustern Suchen
- $O(m + n)$  Zeit ist zu langsam

## Text Index

- Datenstruktur zur Beschleunigung
- $O(m)$  Zeit für Mustersuche möglich
- $O(n)$  Konstruktionszeit

# Text-Indizes

- schneller nach Muster suchen
- häufig nach unterschiedlichen Mustern Suchen
- $O(m + n)$  Zeit ist zu langsam

## Text Index

- Datenstruktur zur Beschleunigung
  - $O(m)$  Zeit für Mustersuche möglich
  - $O(n)$  Konstruktionszeit
- 
- Invertierter Index ⓘ gute Approximation
  - Suffix-Baum
  - Suffix-Array

# Invertierter Index

Für eine Liste von Dokumenten und jedes Wort  $w$  ist

- $f_w$  Anzahl Dokumente, die  $w$  enthalten
- $L_w$  eine Liste mit Tupeln  
(Dokument Id, Anzahl in Dokument)

```
1 The old night keeper keeps the keep in the town
2 In the big old house in the big old gown
3 The house in the town had the big old keep
4 Where the old night keeper never did sleep
5 The night keeper keeps the keep in the night
6 And keeps in the dark and sleeps in the light
```

# Invertierter Index

Für eine Liste von Dokumenten und jedes Wort  $w$  ist

- $f_w$  Anzahl Dokumente, die  $w$  enthalten
- $L_w$  eine Liste mit Tupeln  
(Dokument Id, Anzahl in Dokument)

- 1 The old night keeper keeps the keep in the town
- 2 In the big old house in the big old gown
- 3 The house in the town had the big old keep
- 4 Where the old night keeper never did sleep
- 5 The night keeper keeps the keep in the night
- 6 And keeps in the dark and sleeps in the light

Wort $w$	$f_w$	Invertierte Liste $L_w$ für $w$
and	1	(6, 2)
big	2	(2, 2), (3, 1)
dark	1	(6, 1)
...	...	...
had	1	(3, 1)
house	2	(2, 1), (3, 1)
in	5	(1, 1), (2, 2), (3, 1), (5, 1), (6, 2)
...	...	...

# Invertierter Index

Für eine Liste von Dokumenten und jedes Wort  $w$  ist

- $f_w$  Anzahl Dokumente, die  $w$  enthalten
- $L_w$  eine Liste mit Tupeln  
(Dokument Id, Anzahl in Dokument)

- 1 The old night keeper keeps the keep **in** the town
- 2 **In** the **big** old **house** **in** the **big** old gown
- 3 The **house** **in** the town **had** the **big** old keep
- 4 Where the old night keeper never did sleep
- 5 The night keeper keeps the keep **in** the night
- 6 **And** keeps **in** the **dark** **and** sleeps **in** the light

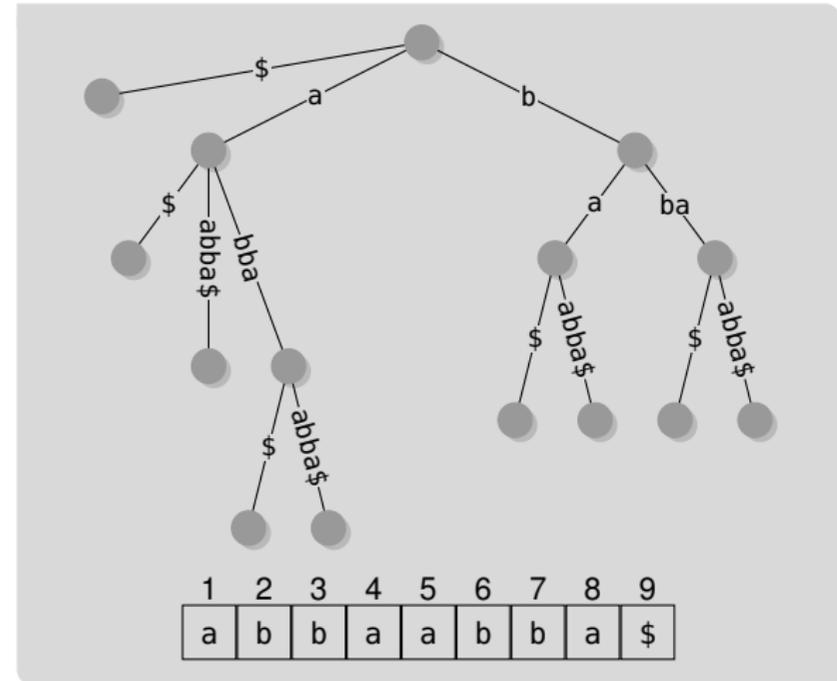
Wort $w$	$f_w$	Invertierte Liste $L_w$ für $w$
<b>and</b>	1	(6, 2)
<b>big</b>	2	(2, 2), (3, 1)
<b>dark</b>	1	(6, 1)
...	...	...
<b>had</b>	1	(3, 1)
<b>house</b>	2	(2, 1), (3, 1)
<b>in</b>	5	(1, 1), (2, 2), (3, 1), (5, 1), (6, 2)
...	...	...

# Suffix-Baum (1/4)

## Definition: Suffix-Baum [Wei73]

Ein Suffix-Baum ( $ST$ ) für einen Text  $T$  der Länge  $n$

- ist ein **kompakter Trie**
- über  $S = \{T[1..n], T[2..n], \dots, T[n..n]\}$ 
  - ⓘ Suffixe sind Präfix-frei



# Suffix-Baum (1/4)

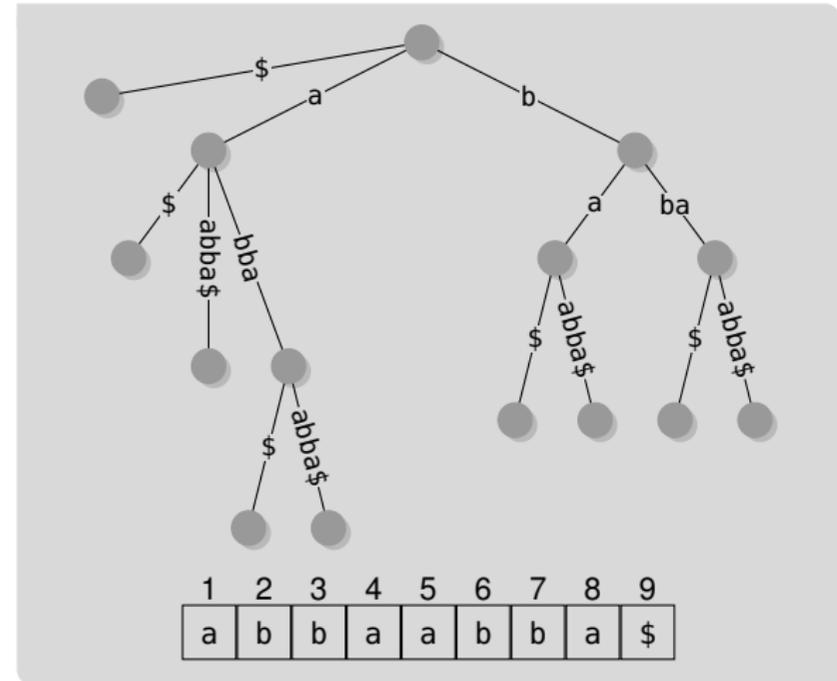
## Definition: Suffix-Baum [Wei73]

Ein Suffix-Baum ( $ST$ ) für einen Text  $T$  der Länge  $n$

- ist ein **kompakter Trie**
- über  $S = \{T[1..n], T[2..n], \dots, T[n..n]\}$ 
  - ⓘ Suffixe sind Präfix-frei

Sei  $G = (V, E)$  ein kompakter Trie mit Wurzel  $r$  und  $v \in V$  ein Knoten, dann

- ist  $\lambda(v)$  die Konkatination der Label auf dem Pfad von  $r$  nach  $v$  und
- $d(v) = |\lambda(v)|$  die **String-Tiefe** von  $v$ 
  - ⓘ String-Tiefe  $\neq$  Tiefe



# Suffix-Baum (1/4)

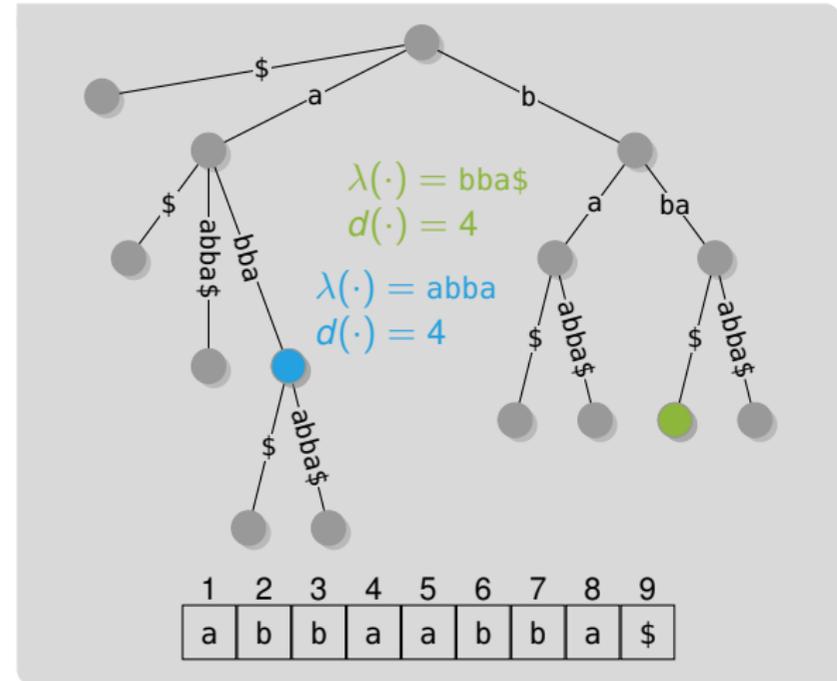
## Definition: Suffix-Baum [Wei73]

Ein Suffix-Baum ( $ST$ ) für einen Text  $T$  der Länge  $n$

- ist ein **kompakter Trie**
- über  $S = \{T[1..n], T[2..n], \dots, T[n..n]\}$ 
  - ⓘ Suffixe sind Präfix-frei

Sei  $G = (V, E)$  ein kompakter Trie mit Wurzel  $r$  und  $v \in V$  ein Knoten, dann

- ist  $\lambda(v)$  die Konkatination der Label auf dem Pfad von  $r$  nach  $v$  und
- $d(v) = |\lambda(v)|$  die **String-Tiefe** von  $v$ 
  - ⓘ String-Tiefe  $\neq$  Tiefe

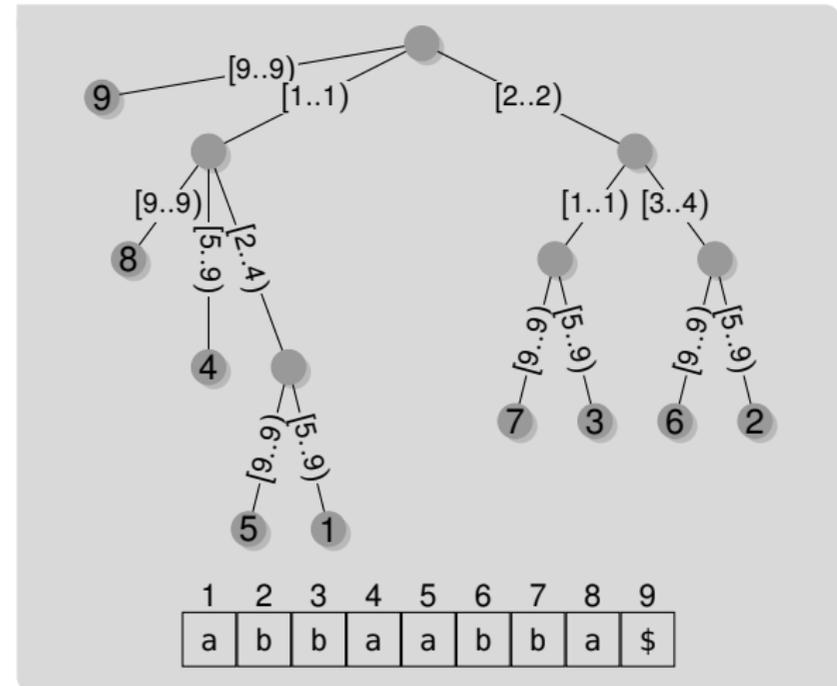




# Suffix-Baum (2/4)

## Repräsentation der Label

- explizites Abspeichern benötigt  $O(n^2)$  Wörter
  - Referenzen benötigen nur  $O(n)$  Wörter
- 
- zur Visualisierung(!) zeigen wir explizite Labels



# Suffix-Baum (2/4)

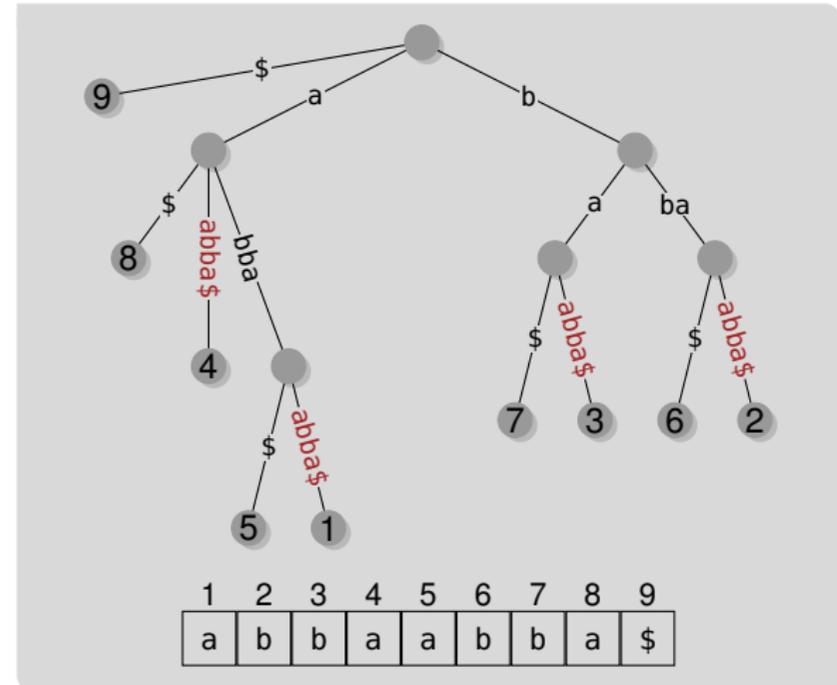
## Repräsentation der Label

- explizites Abspeichern benötigt  $O(n^2)$  Wörter
- Referenzen benötigen nur  $O(n)$  Wörter

- zur Visualisierung(!) zeigen wir explizite Labels

## Suffix Information

- Blätter entsprechen jeweils einem Suffix
- Reihenfolge der Blätter in Tiefensuche entspricht **Suffix-Array**

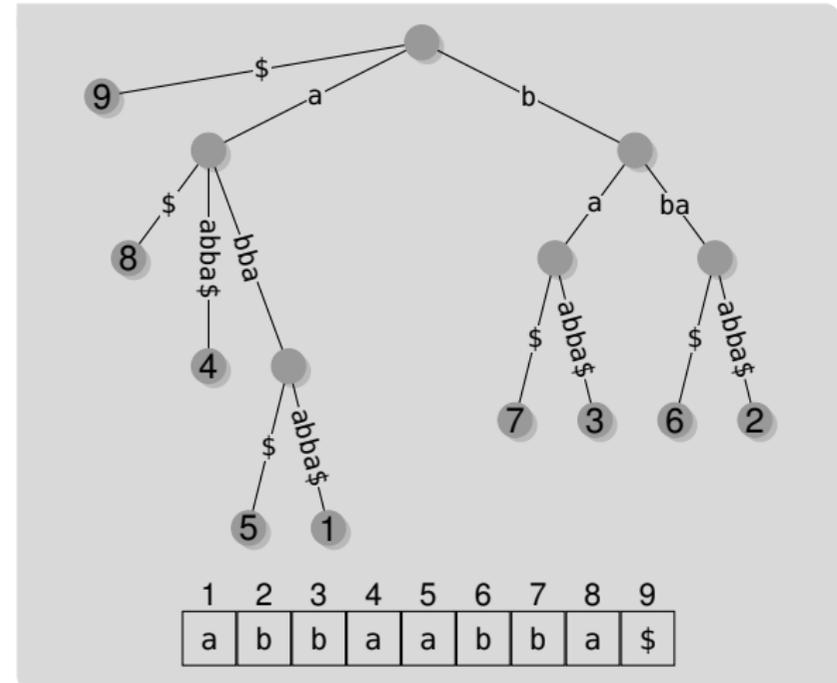


# Suffix-Baum (3/4)

## Mustersuche im Suffix-Baum

- suche Muster  $P[1..m]$
- folge Kanten so lange, wie möglich
- Suchzeit abhängig von Suchzeit der Kinder

- $O(m)$  Suchzeit mit  $O(n\sigma)$  Wörter Platz

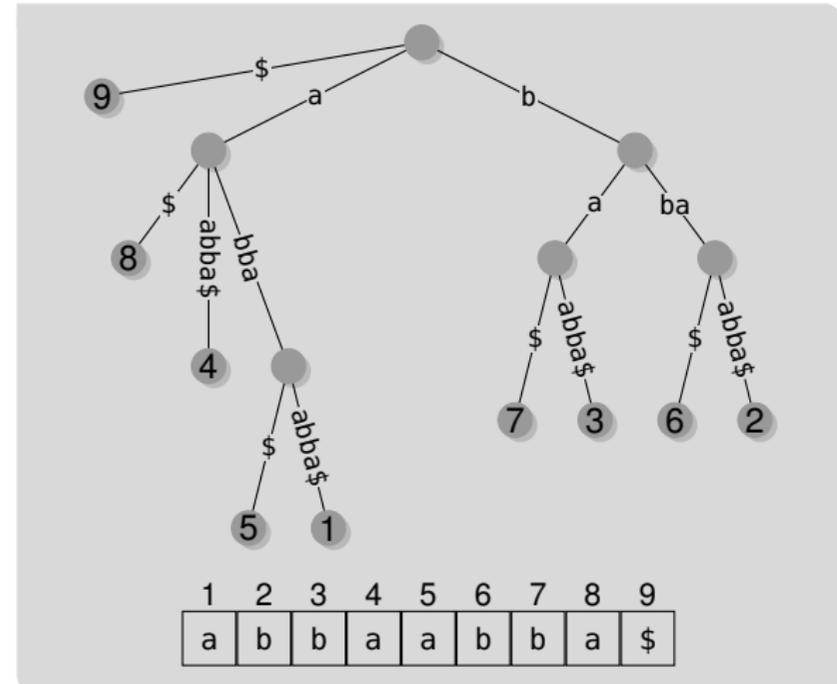


# Suffix-Baum (3/4)

## Mustersuche im Suffix-Baum

- suche Muster  $P[1..m]$
- folge Kanten so lange, wie möglich
- Suchzeit abhängig von Suchzeit der Kinder

- $O(m)$  Suchzeit mit  $O(n\sigma)$  Wörter Platz
- $O(m \cdot \lg \sigma)$  Suchzeit mit  $O(n)$  Wörter Platz

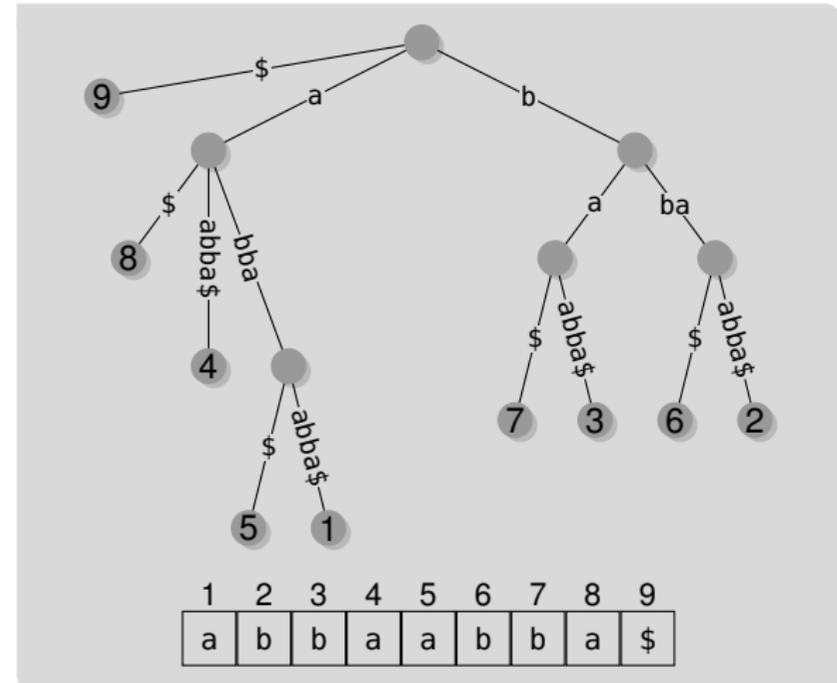


# Suffix-Baum (3/4)

## Mustersuche im Suffix-Baum

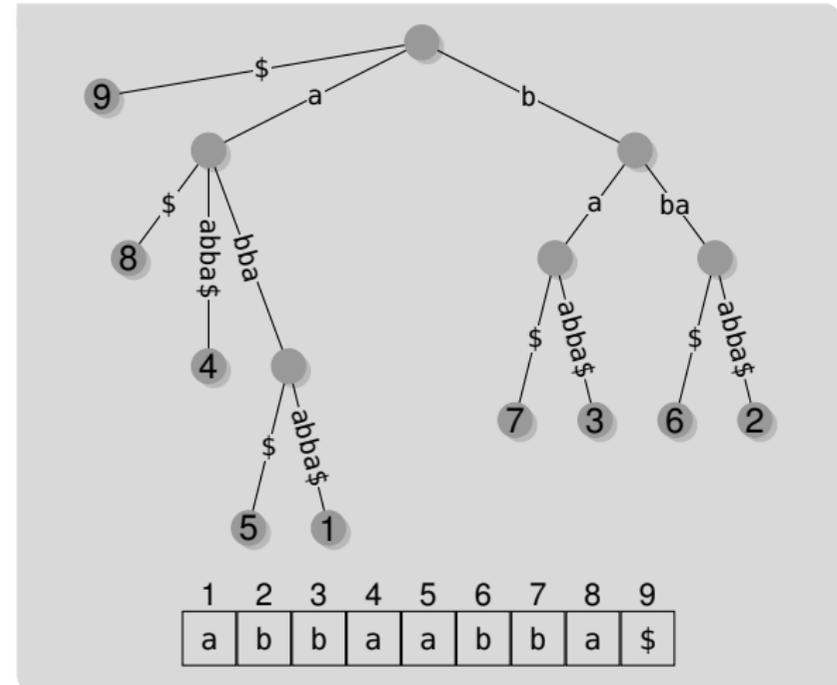
- suche Muster  $P[1..m]$
- folge Kanten so lange, wie möglich
- Suchzeit abhängig von Suchzeit der Kinder

- $O(m)$  Suchzeit mit  $O(n\sigma)$  Wörter Platz
- $O(m \cdot \lg \sigma)$  Suchzeit mit  $O(n)$  Wörter Platz
- $O(m + \lg \sigma)$  Suchzeit mit  $O(n)$  Wörter Platz 🍷  
 ⓘ wird in Text-Indexierung behandelt



# Suffix-Baum (4/4)

- mächtigstes Werkzeug der Stringology(?)
- aber hoher Platzverbrauch
- effiziente und direkte Konstruktion für konstante Alphabete  $O(n)$  Zeit [Ukk95]
- Konstruktion in  $O(n)$  Zeit für ganzzahlige Alphabete [Far97]
- kann einfach in  $O(n)$  Zeit aus dem Suffix-Array konstruiert werden
- **nächste Vorlesung:** Suffix-Array-Konstruktion in  $O(n)$  Zeit



# Suffix-Array und LCP-Array

## Definition: Suffix Array [GBS92; MM93]

Das **Suffix-Array** (SA) für einen Text  $T$  der Länge  $n$  ist die Permutation von  $[1, n]$ , so dass für  $i \leq j \in [1, n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T$	a	b	a	b	c	a	b	c	a	b	b	a	\$
$SA$	13	12	1	9	6	3	11	2	10	7	4	8	5
$LCP$	0	0	1	2	2	5	0	2	1	1	4	0	3

\$	a	a	a	a	a	b	b	b	b	b	c	c
	\$	b	b	b	b	a	a	b	b	c	a	a
		a	b	c	c	\$	b	a	a	a	b	b
		b	a	a	a		c	c	b	b	b	c
		c	\$	b	b		a	a	b	a	\$	a
		a		a	c		b	b	a			b
		b		b	a		c	c	b			b
		c		\$	b		a	a	b			a
		a			a		b	b	a			b
		b			\$		a	a	\$			a
		a										\$

# Suffix-Array und LCP-Array

## Definition: Suffix Array [GBS92; MM93]

Das **Suffix-Array** (SA) für einen Text  $T$  der Länge  $n$  ist die Permutation von  $[1, n]$ , so dass für  $i \leq j \in [1, n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

## Definition: Longest-Common-Prefix-Array

Für einen Text  $T$  der Länge  $n$  und sein Suffix-Array  $SA$  ist das **LCP-array** definiert als

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell: T[SA[i]..SA[i] + \ell] = \\ T[SA[i-1]..SA[i-1] + \ell]\} & i \neq 1 \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T$	a	b	a	b	c	a	b	c	a	b	b	a	\$
$SA$	13	12	1	9	6	3	11	2	10	7	4	8	5
$LCP$	0	0	1	2	2	5	0	2	1	1	4	0	3

\$	a	a	a	a	a	b	b	b	b	b	c	c
\$	\$	b	b	b	b	a	a	b	b	c	a	a
		a	b	c	c	\$	b	c	a	a	b	b
		b	a	a	a		c	a	\$	b	b	c
		c	\$	b	b		a	b	b	a	a	a
		a		b	c		b	c	a	a	\$	b
		b		a	a		c	a	\$	b	b	b
		c		\$	b		a	b		b	a	a
		a			b		b	a		a		\$
		b			a		a	b		b		b
		b			\$		\$	a		\$		a
		a						\$				\$
		\$										



# Literatur I

- [Far97] Martin Farach. „Optimal Suffix Tree Construction with Large Alphabets“. In: *FOCS*. IEEE Computer Society, 1997, Seiten 137–143. DOI: [10.1109/SFCS.1997.646102](https://doi.org/10.1109/SFCS.1997.646102).
- [GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates und Tim Snider. „New Indices for Text: Pat Trees and Pat Arrays“. In: *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992, Seiten 66–82.
- [KMP77] Donald E. Knuth, James H. Morris Jr. und Vaughan R. Pratt. „Fast Pattern Matching in Strings“. In: *SIAM J. Comput.* 6.2 (1977), Seiten 323–350. DOI: [10.1137/0206024](https://doi.org/10.1137/0206024).
- [MM93] Udi Manber und Eugene W. Myers. „Suffix Arrays: A New Method for On-Line String Searches“. In: *SIAM J. Comput.* 22.5 (1993), Seiten 935–948. DOI: [10.1137/0222058](https://doi.org/10.1137/0222058).
- [Ste+15] Zachary D Stephens., Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha und Gene E. Robinson. „Big Data: Astronomical or Genomical?“ In: *PLOS Biology* 13.7 (Juli 2015), Seiten 1–11. DOI: [10.1371/journal.pbio.1002195](https://doi.org/10.1371/journal.pbio.1002195).

## Literatur II

- [Ukk95] Esko Ukkonen. „On-Line Construction of Suffix Trees“. In: *Algorithmica* 14.3 (1995), Seiten 249–260. DOI: [10.1007/BF01206331](https://doi.org/10.1007/BF01206331).
- [Wei73] Peter Weiner. „Linear Pattern Matching Algorithms“. In: *SWAT (FOCS)*. IEEE Computer Society, 1973, Seiten 1–11. DOI: [10.1109/SWAT.1973.13](https://doi.org/10.1109/SWAT.1973.13).