Karlsruher Institut für Technologie

Algorithmen 2

Kapitel 14: Stringology Teil 1 – Strings Sortieren, Mustersuche und Suffix-Bäume

Florian Kurpicz

Stringology

Algorithmen und Datenstrukturen für Strings

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

Texte sind Überall

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

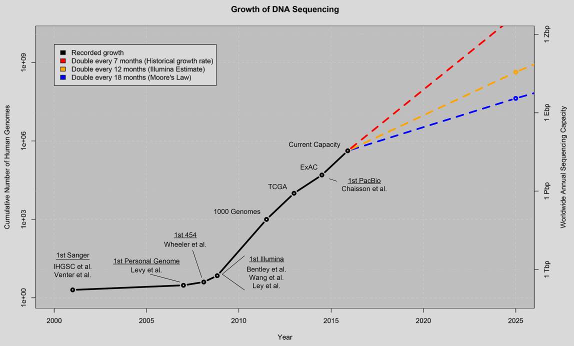
Stringology

Algorithmen und Datenstrukturen für Strings

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

Texte sind Überall

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



[Ste+15]

Stringology in Algorithmen 2

Über die Veranstaltung

- diese und nächste Woche
- bei Interesse Vorlesung Text-Indexierung
  Wahlbereich/Vertiefung

Über Mich

- Mail: kurpicz@kit.edu
- Sprechstunde: Montags 15:30–16:00 Uhr
- Betreuung Abschlussarbeit: Gerne Melden!

Stringology in Algorithmen 2

Über die Veranstaltung

- diese und nächste Woche
- bei Interesse Vorlesung Text-Indexierung
  Wahlbereich/Vertiefung

Über Mich

- Mail: kurpicz@kit.edu
- Sprechstunde: Montags 15:30–16:00 Uhr
- Betreuung Abschlussarbeit: Gerne Melden!

- 🌶 bedeutet nicht Prüfungsrelevant

PINGO – Interaktiv an der Vorlesung teilnehmen

https://pingo.scc.kit.edu/792798

Stringology-Notationen

Karlsruher Institut für Technologie

Definition: Text

- Σ 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

- Σ 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

- Σ 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 σ in eine konstante Anzahl Wörter passt
- **Geordnetes Alphabet**: Zeichen können nur verglichen werden

Stringology-Notationen

Karlsruher Institut für Technologie

Definition: Text

- Σ 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 σ 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

abeyance$
abandon$
abduction$
enforcer$
enervate$
abhorrent$
enfeeblement$
abattoir$
endurance$
aberran$
energizer$

Strings Sortieren

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

Eindeutiges und Längstes Gemeinsames Präfix

$\perp$	**aban**don$
3	**abat**toir$
2	**abd**uction$
2	**aber**ran$
3	**abey**ance$
2	**abh**orrent$
0	**end**urance$
2	**energ**izer$
4	**enerv**ate$
2	**enfe**eblement$
3	**enfo**rcer$

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

Eindeutiges und Längstes Gemeinsames Präfix

$\perp$	**aban**don$
3	**abat**toir$
2	**abd**uction$
2	**aber**ran$
3	**abey**ance$
2	**abh**orrent$
0	**end**urance$
2	**energ**izer$
4	**enerv**ate$
2	**enfe**eblement$
3	**enfo**rcer$

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

PINGO Was ist größer: Die Summe aller eindeutigen Präfixe oder die Summe der eindeutigen Präfixe?

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

Karlsruher Institut für Technologie

```
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

Karlsruher Institut für Technologie

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

a	b	...	e	f	...	σ
6	0	...	5	0	...	0

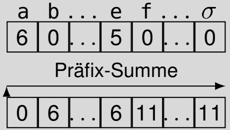
Präfix-Summe

0	6	...	6	11	...	11

Strings Sortieren: Most Significant Digit Radix Sort

abeyance$
abandon$
abduction$
enforcer$
enervate$
abhorrent$
enfeeblement$
abattoir$
endurance$
aberran$
energizer$

- 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

a	b	...	e	f	...	σ
6	0	...	5	0	...	0

Präfix-Summe

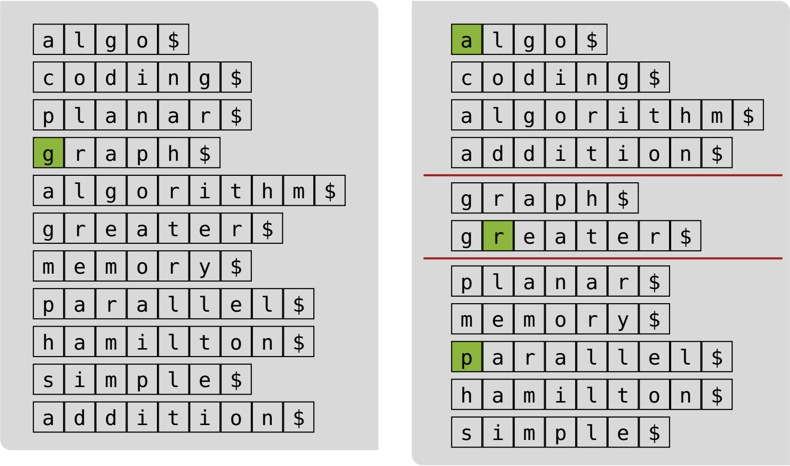
0	6	...	6	11	...	11

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

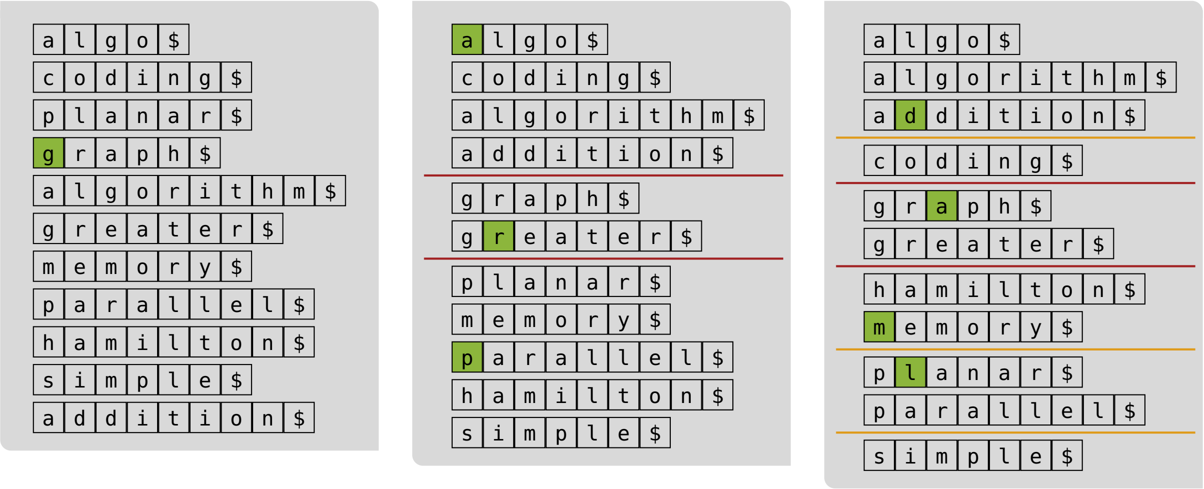
Strings Sortiere: Multikey Quicksort (1/2)

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)



Strings Sortiere: Multikey Quicksort (1/2)



Strings Sortieren: Multikey Quicksort (2/2)

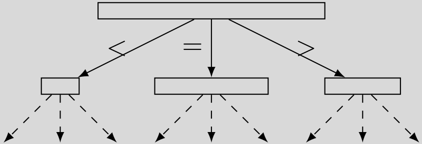
Function
$\mathtt{MulikeyQS}(S = \{s_1, \ldots, s_k\}, \ell \in \mathbb{N})$**:**

> **if** $k \leq 1$ **then**
>
> > **return** S
>
> Wähle $p \in S$ mit Gleichverteilung
>
> **return** Konkatenation von
> $\mathtt{MulikeyQS}(\{s \in S \colon s[\ell] < p[\ell]\}, \ell) \cdot$
> $\mathtt{MulikeyQS}(\{s \in S \colon s[\ell] = p[\ell]\}, \ell + 1) \cdot$
> $\mathtt{MulikeyQS}(\{s \in S \colon s[\ell] > p[\ell]\}, \ell)$

Strings Sortieren: Multikey Quicksort (2/2)

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

if $k \leq 1$ **then**
  return S
Wähle $p \in S$ mit Gleichverteilung
return Konkatenation von
 $\texttt{MulikeyQS}(\{s \in S\colon s[\ell] < p[\ell]\}, \ell) \cdot$
 $\texttt{MulikeyQS}(\{s \in S\colon s[\ell] = p[\ell]\}, \ell + 1) \cdot$
 $\texttt{MulikeyQS}(\{s \in S\colon s[\ell] > p[\ell]\}, \ell)$



Strings Sortieren: Multikey Quicksort (2/2)

Function

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

if $k \leq 1$ **then**

  return S

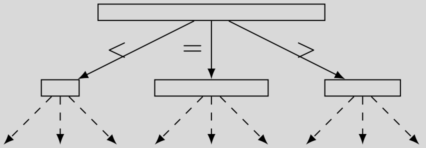
Wähle $p \in S$ mit Gleichverteilung

return Konkatenation von

$\texttt{MulikeyQS}(\{s \in S\colon s[\ell] < p[\ell]\}, \ell) \cdot$

$\texttt{MulikeyQS}(\{s \in S\colon s[\ell] = p[\ell]\}, \ell + 1) \cdot$

$\texttt{MulikeyQS}(\{s \in S\colon s[\ell] > p[\ell]\}, \ell)$

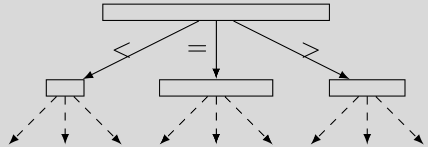


- **PINGO** Welche Laufzeit hat MKQS?

Strings Sortieren: Multikey Quicksort (2/2)

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

- **if** $k \leq 1$ **then**
  - **return** S
- Wähle $p \in S$ mit Gleichverteilung
- **return** Konkatenation von
  - $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] < p[\ell]\}, \ell) \cdot$
  - $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] = p[\ell]\}, \ell + 1) \cdot$
  - $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] > p[\ell]\}, \ell)$

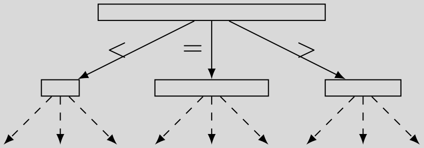


- **PINGO** Welche Laufzeit hat MKQS?
- Laufzeit: $O(k \lg k + N)$
- genauer: $O(k \lg k + d)$
- Übung: in-place Variante

Strings Sortieren: Multikey Quicksort (2/2)

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

> **if** $k \leq 1$ **then**
>
> > **return** S
>
> Wähle $p \in S$ mit Gleichverteilung
>
> **return** Konkatenation von
>
> $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] < p[\ell]\}, \ell) \cdot$
>
> $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] = p[\ell]\}, \ell + 1) \cdot$
>
> $\texttt{MulikeyQS}(\{s \in S \colon s[\ell] > p[\ell]\}, \ell)$



- **PINGO** Welche Laufzeit hat MKQS?
- Laufzeit: $O(k \lg k + N)$
- genauer: $O(k \lg k + d)$
- Übung: in-place Variante

Laufzeitanalyse (Skizze)

- $s[\ell] = p[\ell]$
  - $s[\ell]$ wird nicht mehr betrachtet, da
  - ℓ 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

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

- $T[i..j] = T[i]\ldots T[j]$ 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 Σ.

- 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 Σ.

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

Weitere Stringology-Notationen

Karlsruher Institut für Technologie

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 Σ.

- 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

- $T[1..n] = \texttt{abbaabba}$ und $T[5..n] = \texttt{abba}$

Weitere Stringology-Notationen

Karlsruher Institut für Technologie

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 Σ.

- 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

- $T[1..n] = \mathtt{abbaabba}$ und $T[5..n] = \mathtt{abba}$

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 $\texttt{NaiveMS}(T[1..n], P[1..m])$**:**

```
i, j = 1
while i ≤ n − m + 1 do
    while j ≤ m and T[i + j − 1] = P[j] do
        j = j + 1
    if j > m then
        P kommt in T an Position i vor
    i = i + 1
    j = 1
```

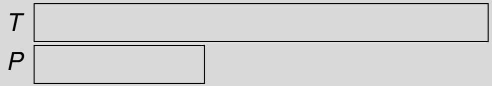
Mustersuche

Jetzt

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

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

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + 1$
  $j = 1$



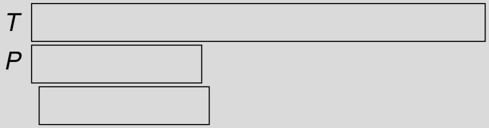
Mustersuche

Jetzt

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

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

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + 1$
  $j = 1$



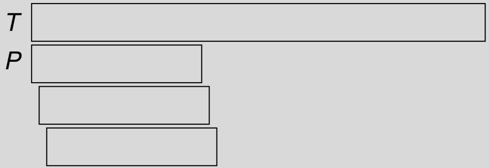
Mustersuche

Jetzt

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

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

```
i, j = 1
while i ≤ n − m + 1 do
    while j ≤ m and T[i + j − 1] = P[j] do
        j = j + 1
    if j > m then
        P kommt in T an Position i vor
    i = i + 1
    j = 1
```



Mustersuche

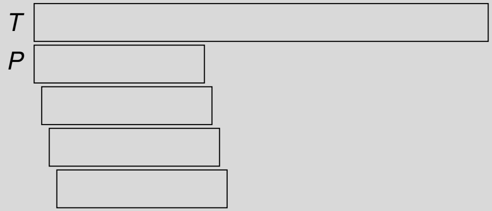
Karlsruher Institut für Technologie

Jetzt

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

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

```
i, j = 1
while i ≤ n − m + 1 do
    while j ≤ m and T[i + j − 1] = P[j] do
        j = j + 1
    if j > m then
        P kommt in T an Position i vor
    i = i + 1
    j = 1
```



Mustersuche

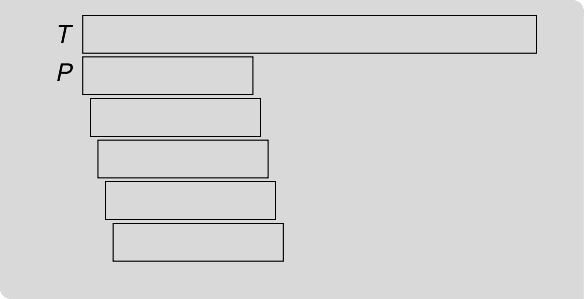
Karlsruher Institut für Technologie

Jetzt

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

Function $\texttt{NaiveMS}(T[1..n], P[1..m])$:

```
i, j = 1
while i ≤ n − m + 1 do
    while j ≤ m and T[i + j − 1] = P[j] do
        j = j + 1
    if j > m then
        P kommt in T an Position i vor
    i = i + 1
    j = 1
```



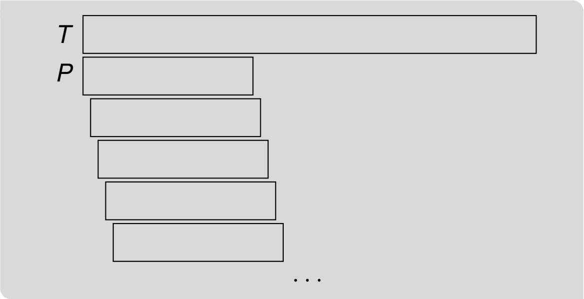
Mustersuche

Jetzt

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

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

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + 1$
  $j = 1$



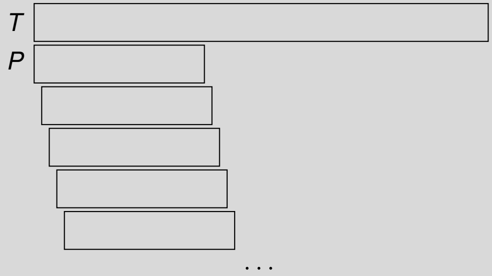
Mustersuche

Jetzt

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

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

$i, j = 1$
while $i \leq n - m + 1$ **do**
    while $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
        $j = j + 1$
    if $j > m$ **then**
        P kommt in T an Position i vor
    $i = i + 1$
    $j = 1$



- **PINGO** Welche Laufzeit hat die naive Mustersuche?

Mustersuche

Karlsruher Institut für Technologie

Jetzt

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

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

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + 1$
  $j = 1$

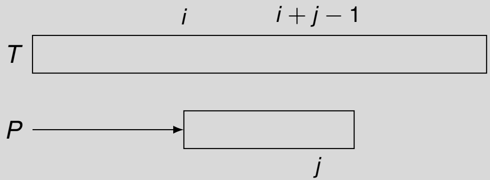
- **PINGO** Welche Laufzeit hat die naive Mustersuche?
- (`NaiveMS`) hat eine Laufzeit von $O(n \cdot m)$

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

- Teile im Muster können sich wiederholen

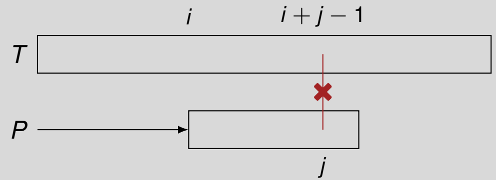
Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



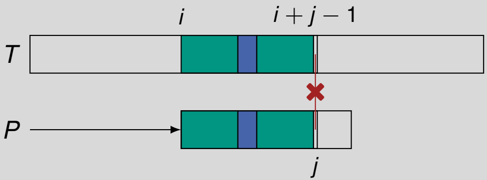
Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



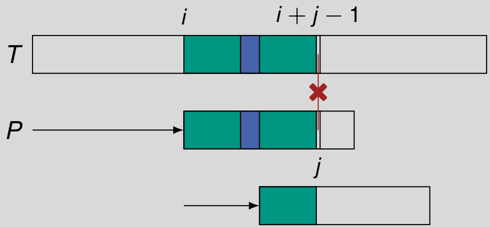
Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



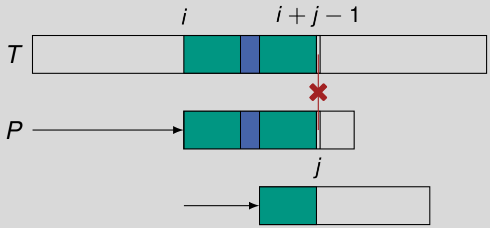
Mustersuche mit Knuth-Morris-Pratt (1/3) [KMP77]

- Teile im Muster können sich wiederholen



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

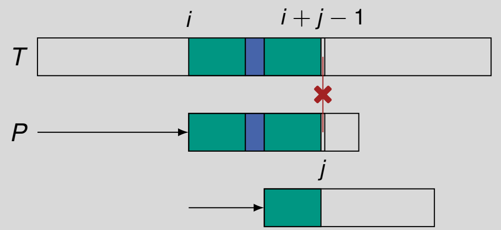
- Teile im Muster können sich wiederholen



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

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

- Teile im Muster können sich wiederholen

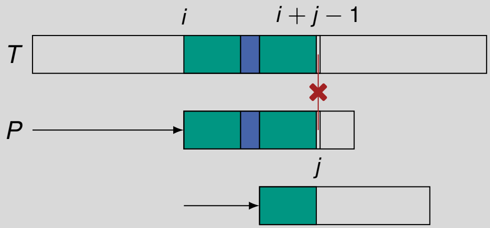


- für Mismatch an Position $j \leq m$ finde
- $\max\{k \in [1..j) \colon P[1..k) = P[j-k-1..j)\}$
- Muster an Position $i+j-k-1$ verschieben

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

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

- Teile im Muster können sich wiederholen



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

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

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

- $i, j = 1$
- **while** $i \leq n - m + 1$ **do**
  - **while** $j \leq m$ **and** $T[i+j-1] = P[j]$ **do**
    - $j = j + 1$
  - **if** $j > m$ **then**
    - P kommt in T an Position i vor
  - $i = i + j - border[j] - 1$
  - $j = \max\{1, border[j] + 1\}$

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

Function $\textit{KMP}(T[1..n], P[1..m], \textit{border}[1..m+1])$**:**

- $i, j = 1$
- **while** $i \leq n - m + 1$ **do**
    - **while** $j \leq m$ **and** $T[i + j - 1] = P[j]$ **do**
        - $j = j + 1$
    - **if** $j > m$ **then**
        - P kommt in T an Position i vor
    - $i = i + j - \textit{border}[j] - 1$
    - $j = \max\{1, \textit{border}[j] + 1\}$

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

Function $\mathit{KMP}(T[1..n], P[1..m], \mathit{border}[1..m+1])$**:**

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i+j-1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + j - \mathit{border}[j] - 1$
  $j = \max\{1, \mathit{border}[j] + 1\}$

Laufzeit

- **PINGO** Welche Laufzeit hat `KMP`?

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

Karlsruher Institut für Technologie

Function $\mathit{KMP}(T[1..n], P[1..m], \mathit{border}[1..m+1])$**:**

$i, j = 1$
while $i \leq n - m + 1$ **do**
  while $j \leq m$ **and** $T[i+j-1] = P[j]$ **do**
    $j = j + 1$
  if $j > m$ **then**
    P kommt in T an Position i vor
  $i = i + j - \mathit{border}[j] - 1$
  $j = \max\{1, \mathit{border}[j] + 1\}$

Laufzeit

- **PINGO** Welche Laufzeit hat KMP?
- Mustersuche mit KMP benötigt $O(n + m)$ Zeit

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

Function $\mathit{KMP}(T[1..n], P[1..m], \mathit{border}[1..m+1])$**:**

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 $i = i + j - \mathit{border}[j] - 1$
8 $j = \max\{1, \mathit{border}[j] + 1\}$

Laufzeit

- **PINGO** Welche Laufzeit hat KMP?
- Mustersuche mit KMP benötigt $O(n + m)$ Zeit

Proof (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 $\textit{KMP}(T[1..n], P[1..m], \textit{border}[1..m+1])$**:**

```
1  i, j = 1
2  while i ≤ n − m + 1 do
3      while j ≤ 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

- **PINGO** Welche Laufzeit hat KMP?
- Mustersuche mit KMP benötigt $O(n+m)$ Zeit

Proof (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)

Karlsruher Institut für Technologie

Function $\mathit{KMP}(T[1..n], P[1..m], \mathit{border}[1..m+1])$**:**

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   $i = i + j - \mathit{border}[j] - 1$
8   $j = \max\{1, \mathit{border}[j] + 1\}$

Laufzeit

- **PINGO** Welche Laufzeit hat KMP?
- Mustersuche mit KMP benötigt $O(n + m)$ Zeit

Proof (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

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

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

```
1  i, j = 1
2  while i ≤ n − m + 1 do
3      while j ≤ 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}
```

Proof (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

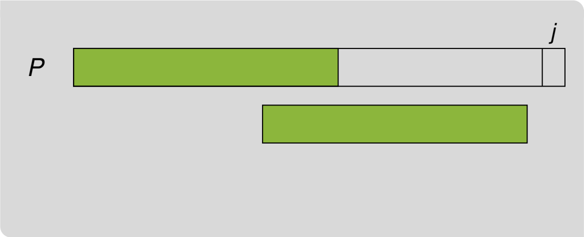
Laufzeit

- **PINGO** Welche Laufzeit hat KMP?
- Mustersuche mit KMP benötigt $O(n+m)$ Zeit

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

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

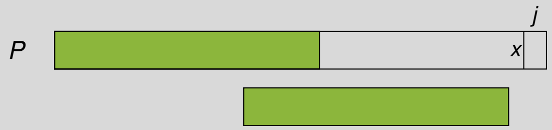
- $border[1] = -1$
- $i = border[1] + 1$
- **for** $j = 2, \ldots, m+1$ **do**
  - **while** $i \geq 0$ **and** $P[i+1] \neq P[j-1]$ **do**
    - $i = border[i+1]$
  - $i = i + 1$
  - $border[j] = i$
- **return** *border*



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

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

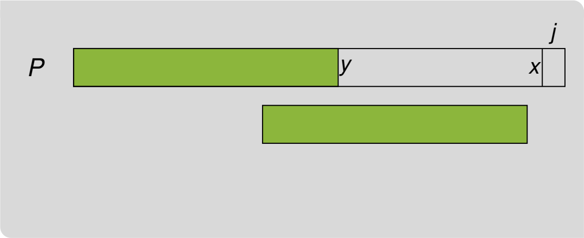
```
border[1] = −1
i = border[1] + 1
for j = 2, ..., m + 1 do
    while i ≥ 0 and P[i + 1] ≠ P[j − 1] do
        i = border[i + 1]
    i = i + 1
    border[j] = i
return border
```



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

Function $\texttt{KMP-Border}(P[1..m+1])$**:**

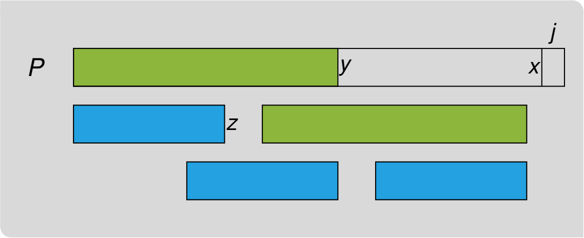
```
border[1] = -1
i = border[1] + 1
for j = 2, ..., m + 1 do
    while i ≥ 0 and P[i + 1] ≠ P[j - 1] do
        i = border[i + 1]
    i = i + 1
    border[j] = i
return border
```



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

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

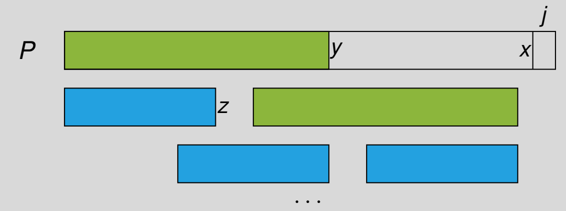
- *border*$[1] = -1$
- $i = \mathit{border}[1] + 1$
- **for** $j = 2, \ldots, m+1$ **do**
  - **while** $i \geq 0$ **and** $P[i+1] \neq P[j-1]$ **do**
    - $i = \mathit{border}[i+1]$
  - $i = i + 1$
  - $\mathit{border}[j] = i$
- **return** *border*



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

Function $\texttt{KMP-Border}(P[1..m+1])$:

```
1  border[1] = −1
2  i = border[1] + 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
```



Proof (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

Text-Indizes

- 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 ℹ 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

$$(\texttt{Dokument Id}, \texttt{Anzahl in Dokument})$$

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

$$(\texttt{Dokument Id}, \texttt{Anzahl in Dokument})$$

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

Karlsruher Institut für Technologie

Für eine Liste von Dokumenten und jedes Wort w ist

- f_w Anzahl Dokumente, die w enthalten
- L_w eine Liste mit Tupeln

$$(\texttt{Dokument Id}, \texttt{Anzahl in Dokument})$$

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

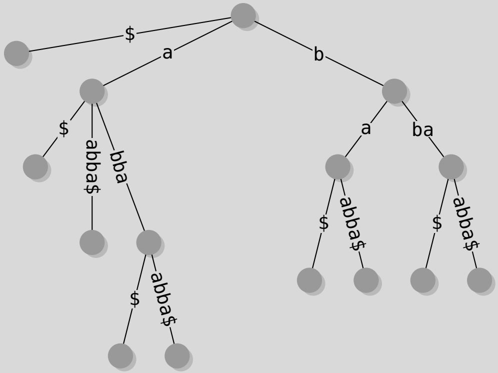
Wort w	f_w	Invertierte Liste L_w für w
and	1	$(6, 2)$
big	2	$(2, 2), (3, 1)$
dark	1	$(6, 1)$
...	...	...
had	1	$(3, 1)$
house	2	$(2, 1), (3, 1)$
in	5	$(1, 1), (2, 2), (3, 1), (5, 1), (6, 2)$
...	...	...

Suffix-Baum (1/4)

Definition: Suffix-Baum [Wei73]

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

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



1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

Suffix-Baum (1/4)

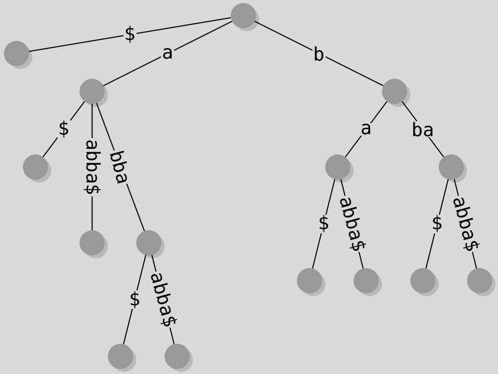
Definition: Suffix-Baum [Wei73]

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

- ist ein **kompakter Trie**
- über $S = \{T[1..n], T[2..n], \ldots, T[n..n]\}$
  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 Konkatenation der Label auf dem Pfad von r nach v und
- $d(v) = |\lambda(v)|$ die **String-Tiefe** von v
  String-Tiefe $\neq$ Tiefe



1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

Suffix-Baum (1/4)

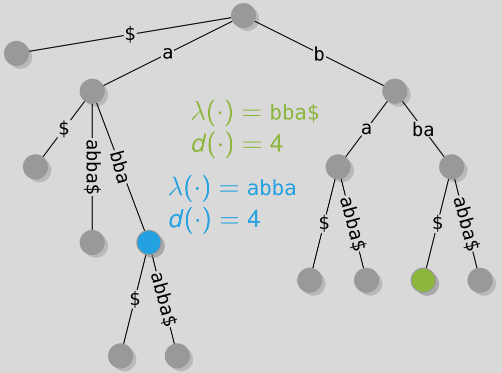
Definition: Suffix-Baum [Wei73]

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



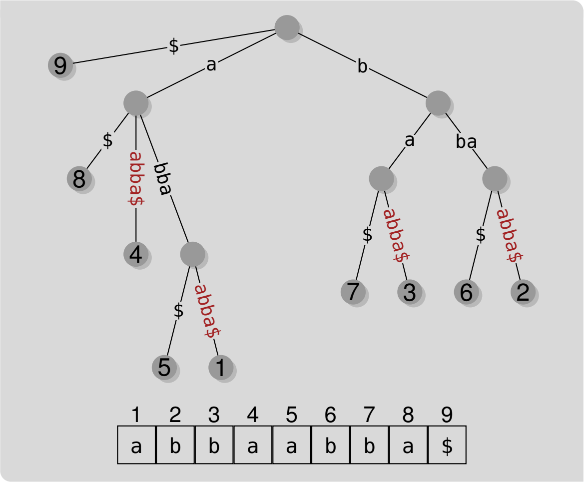
1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

Suffix-Baum (2/4)

Repräsentation der Label

- **PINGO** Wie viel Platz benötigt ein Suffix-Baum mit expliziten Kantenbeschriftungen?

- zur Visualisierung(!) zeigen wir explizite Labels

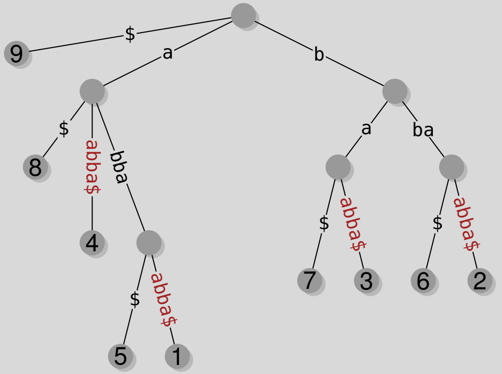


Suffix-Baum (2/4)

Repräsentation der Label

- **PINGO** Wie viel Platz benötigt ein Suffix-Baum mit expliziten Kantenbeschriftungen?
- explizites Abspeichern benötigt $O(n^2)$ Wörter
- Referenzen benötigen nur $O(n)$ Wörter

- zur Visualisierung(!) zeigen wir explizite Labels



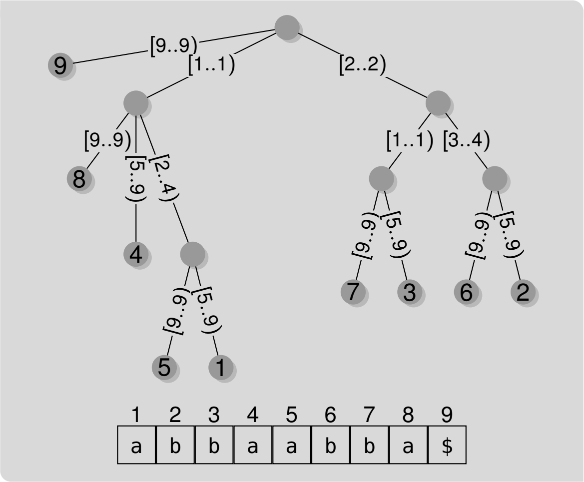
1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

Suffix-Baum (2/4)

Repräsentation der Label

- **PINGO** Wie viel Platz benötigt ein Suffix-Baum mit expliziten Kantenbeschriftungen?
- explizites Abspeichern benötigt $O(n^2)$ Wörter
- Referenzen benötigen nur $O(n)$ Wörter

- zur Visualisierung(!) zeigen wir explizite Labels



Suffix-Baum (2/4)

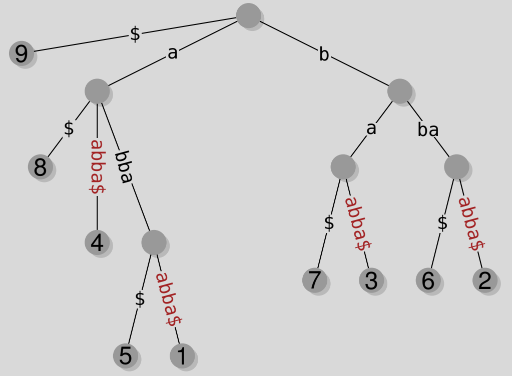
Repräsentation der Label

- **PINGO** Wie viel Platz benötigt ein Suffix-Baum mit expliziten Kantenbeschriftungen?
- 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



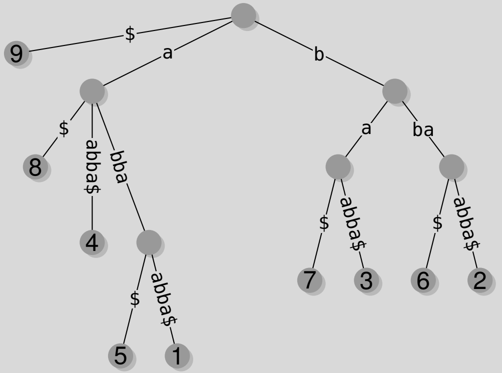
1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

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



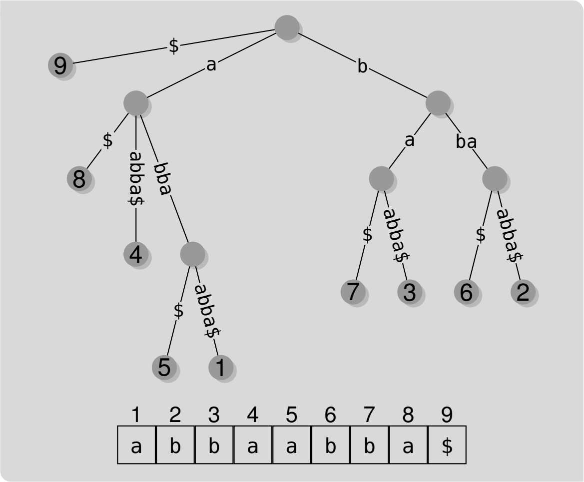
1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

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



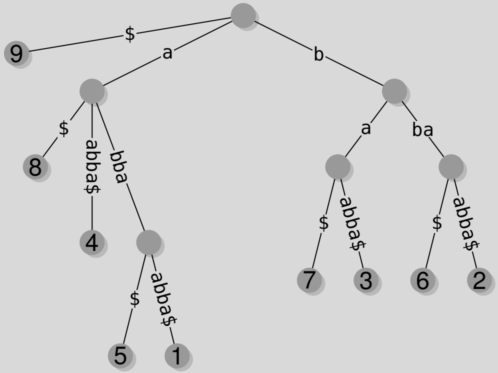
Suffix-Baum (3/4)

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 🌶

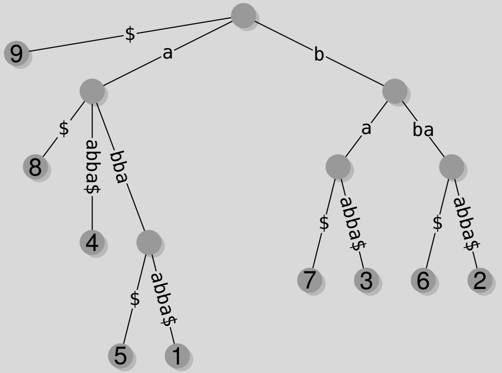
  wird in Text-Indexierung behandelt



1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	a	$

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



1	2	3	4	5	6	7	8	9
a	b	b	a	a	b	b	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 $i \leq j \in [1, n]$

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

	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$	ababcabcabba$	abba$	abcabba$	abcabcabba$	ba$	babcabcabba$	bba$	bcabba$	bcabcabba$	cabba$	cabcabba$

Suffix-Array und LCP-Array

Karlsruher Institut für Technologie

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

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

	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\$	ababcabcabba\$	abba\$	abcabba\$	abcabcabba\$	ba\$	babcabcabba\$	bba\$	bcabba\$	bcabcabba\$	cabba\$	cabcabba\$

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 $i \leq j \in [1, n]$

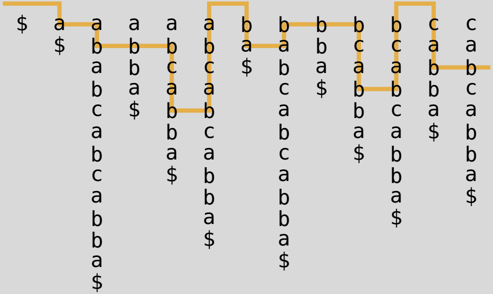
$$T[SA[i]..n] \leq T[SA[j]..n]$$

Definition: Longest-Common-Prefix-Array

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

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

	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



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.

[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.

[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.

[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.

Literatur II

[Ukk95] Esko Ukkonen. „On-Line Construction of Suffix Trees“. In: *Algorithmica* 14.3 (1995), Seiten 249–260. DOI: 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.