

# **Algorithmen II**

**Peter Sanders, Timo Bingmann**

**Übungen:**

**Sebastian Lamm, Demian Hesse**

Institut für Theoretische Informatik

Web:

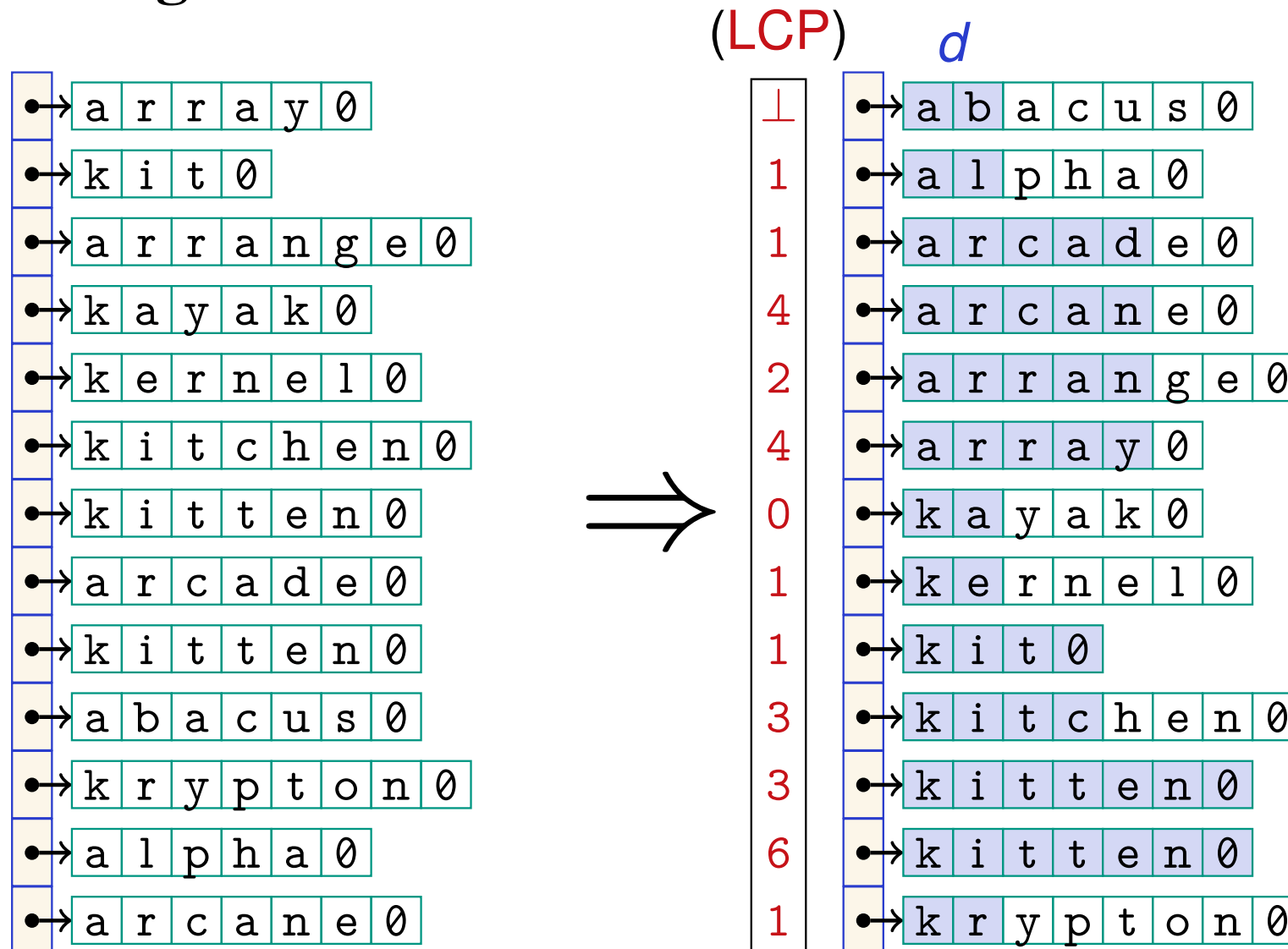
[http://algo2.iti.kit.edu/AlgorithmenII\\_WS18.php](http://algo2.iti.kit.edu/AlgorithmenII_WS18.php)

# 11 Stringology

## (Zeichenkettenalgorithmen)

- Strings sortieren
- Patterns suchen
  - Pattern vorverarbeiten
  - Text vorverarbeiten
    - \* Invertierte Indizes
    - \* Suffix Trees / Suffix Arrays
- Datenkompression
- Pattern suchen in komprimierten Indizes

# Strings Sortieren



Eingabe:  $n$  Strings mit  $N$  Zeichen insgesamt.

# Sortieren: Most Significant Digit Radix Sort

a r r a y

k i t

a r r a n g e

k a y a k

k e r n e l

k i t c h e n

k i t t e n

a r c a d e

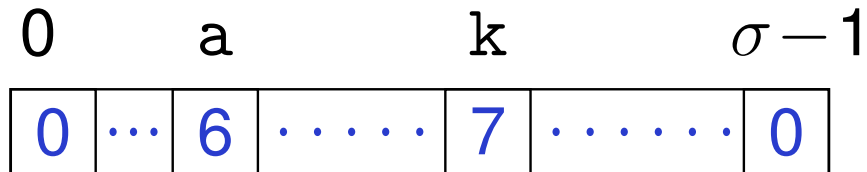
k i t e

a b a c u s

k r y p t o n

a l p h a

a r c a i c



# Sortieren: Most Significant Digit Radix Sort

a r r a y

k i t

a r r a n g e

k a y a k

k e r n e l

k i t c h e n

k i t t e n

a r c a d e

k i t e

a b a c u s

k r y p t o n

a l p h a

a r c a i c

0            a            k             $\sigma - 1$

0	...	6	.....	7	.....	0
---	-----	---	-------	---	-------	---

0	...	0	6	...	6	13	.....	13	13
---	-----	---	---	-----	---	----	-------	----	----

# Sortieren: Most Significant Digit Radix Sort

a r r a y

k i t

a r r a n g e

k a y a k

k e r n e l

k i t c h e n

k i t t e n

a r c a d e

k i t e

a b a c u s

k r y p t o n

a l p h a

a r c a i c

0            a            k             $\sigma - 1$

0	...	6	.....	7	.....	0
---	-----	---	-------	---	-------	---

0	...	0	6	...	6	13	.....	13	13
---	-----	---	---	-----	---	----	-------	----	----

# Sortieren: Most Significant Digit Radix Sort

a r r a y

a r r a n g e

a r c a d e

a b a c u s

a l p h a

a r c a i c

k i t

k a y a k

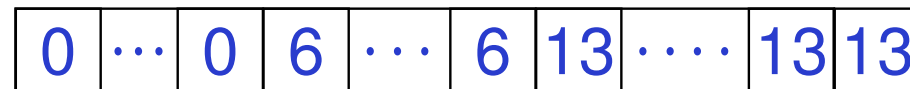
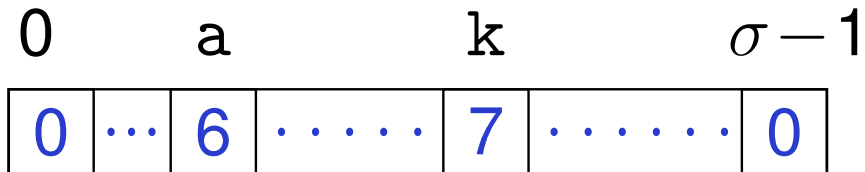
k e r n e l

k i t c h e n

k i t t e n

k i t e

k r y p t o n



- Laufzeit:  $\mathcal{O}(d + r\sigma + n \log \sigma)$
- Varianten: out-of-place und in-place!

# Sortieren: Most Significant Digit Radix Sort

a	r	r	a	y		
k	i	t				
a	r	r	a	n	g	e
k	a	y	a	k		
k	e	r	n	e	l	
k	i	t	c	h	e	n
k	i	t	t	e	n	
a	r	c	a	d	e	
k	i	t	e			
a	b	a	c	u	s	
k	r	y	p	t	o	n
a	l	p	h	a		
a	r	c	a	i	c	

0            a                    k                             $\sigma - 1$

0	...	6	.....	7	.....	0
---	-----	---	-------	---	-------	---

0	...	0	6	...	6	13	.....	13	13
---	-----	---	---	-----	---	----	-------	----	----

$p_1$	0	...	2	.....	3	.....	0
$p_2$	0	...	2	.....	3	.....	0
$p_3$	0	...	2	.....	1	.....	0



# Sortieren: Most Significant Digit Radix Sort

a	r	r	a	y		
k	i	t				
a	r	r	a	n	g	e
k	a	y	a	k		
k	e	r	n	e	l	
k	i	t	c	h	e	n
k	i	t	t	e	n	
a	r	c	a	d	e	
k	i	t	e			
a	b	a	c	u	s	
k	r	y	p	t	o	n
a	l	p	h	a		
a	r	c	a	i	c	

0	a	k	$\sigma - 1$			
0	...	6	.....	7	.....	0

0	...	0	6	...	6	13	.....	13	13
---	-----	---	---	-----	---	----	-------	----	----

$p_1$	0	...	2	.....	3	.....	0
$p_2$	0	...	2	.....	3	.....	0
$p_3$	0	...	2	.....	1	.....	0

$p_1$	0	...	0	6	...	6	13	.....	13	13
$p_2$	0	...	2	6	...	9	13	.....	13	
$p_3$	0	...	4	6	...	12	13	.....	13	

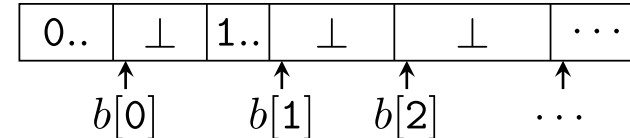
# Sortieren: Most Significant Digit Radix Sort

**Algorithm :** Sequential Radix Sort “CE0”, adapted from [KR08; Ran07]

```

1 Function RadixSortCE0( $\mathcal{S}, h$ )
   Input :  $\mathcal{S} = [s_0, \dots, s_{n-1}]$  an array of  $n$  strings with common prefix  $h$ .
2    $c := [0, \dots, 0]$  // Allocate  $|\Sigma|$  integer counters initialized with zero,
3   for  $i = 0, \dots, n - 1$  do  $c[s_i[h]]++$  // and count character occurrences.
4    $b := [0, \perp, \dots, \perp]$  // Calculate exclusive prefix sum of counters
5   for  $i = 1, \dots, n - 1$  do  $b[i] := b[i - 1] + c[i - 1]$  // as bucket pointers.
6    $\mathcal{T} := \text{allocate}(n, \text{string pointer})$  // Allocate temporary array for sorted output.
7   for  $i = 0, \dots, n - 1$  do // Reorder
8      $\mathcal{T}[b[s_i[h]]] := \text{move}(s_i)$  // into
9      $b[s_i[h]]++$  // buckets
10   $\text{copy}(\mathcal{T} \rightarrow \mathcal{S}), \text{deallocate}(\mathcal{T}, b)$ .
11   $x := c[0]$  // Track beginning of bucket as  $x$ ,
12  for  $i = 1, \dots, |\Sigma| - 1$  do // recurse into every unfinished bucket,
13     $\text{StringSort}(\mathcal{S}[x .. x + c[i]], h + 1)$  // except for the first (zero-termination),
14     $x := x + c[i]$  // which contains all fully sorted strings.
   Output : The array  $\mathcal{S}$  is fully sorted lexicographically.

```



# Strings Sortieren: Multikey Quicksort

Auch: MKQS / ternary quicksort

**Function** mkqSort( $S$  : Sequence of String,  $\ell$  :  $\mathbb{N}$ ) : Sequence of String

**assert**  $\forall e, e' \in S : e[1..\ell - 1] = e'[1..\ell - 1]$

**if**  $|S| \leq 1$  **then return**  $S$  // base case

pick  $p \in S$  uniformly at random // pivot string

**return** concatenation of  
 $\text{mkqSort}(\langle e \in S : e[\ell] < p[\ell] \rangle, \ell)$ ,  
 $\text{mkqSort}(\langle e \in S : e[\ell] = p[\ell] \rangle, \ell + 1)$ , and  
 $\text{mkqSort}(\langle e \in S : e[\ell] > p[\ell] \rangle, \ell)$

- Laufzeit:  $O(|S| \log |S| + \sum_{t \in S} |t|)$
- genauer:  $O(|S| \log |S| + d)$  ( $d$ : Summe der **eindeutigen Präfixe**)
- Übung: **in-place!**

# Strings Sortieren: Multikey Quicksort

S A A L  
B I E N E  
E H R E  
H A U S  
A R M  
M I E T E  
T A S S E  
M O R D  
H A N D  
S E E  
H U N D  
H A L L E  
N A C H T

# Strings Sortieren: Multikey Quicksort

S A A L  
B I E N E  
E H R E  
H A U S  
A R M  
M I E T E  
T A S S E  
M O R D  
**H** A N D  
S E E  
H U N D  
H A L L E  
N A C H T

*p*

# Strings Sortieren: Multikey Quicksort

B I E N E

E H R E

A R M

H A U S

H A N D

H U N D

H A L L E

S A A L

T A S S E

M I E T E

M O R D

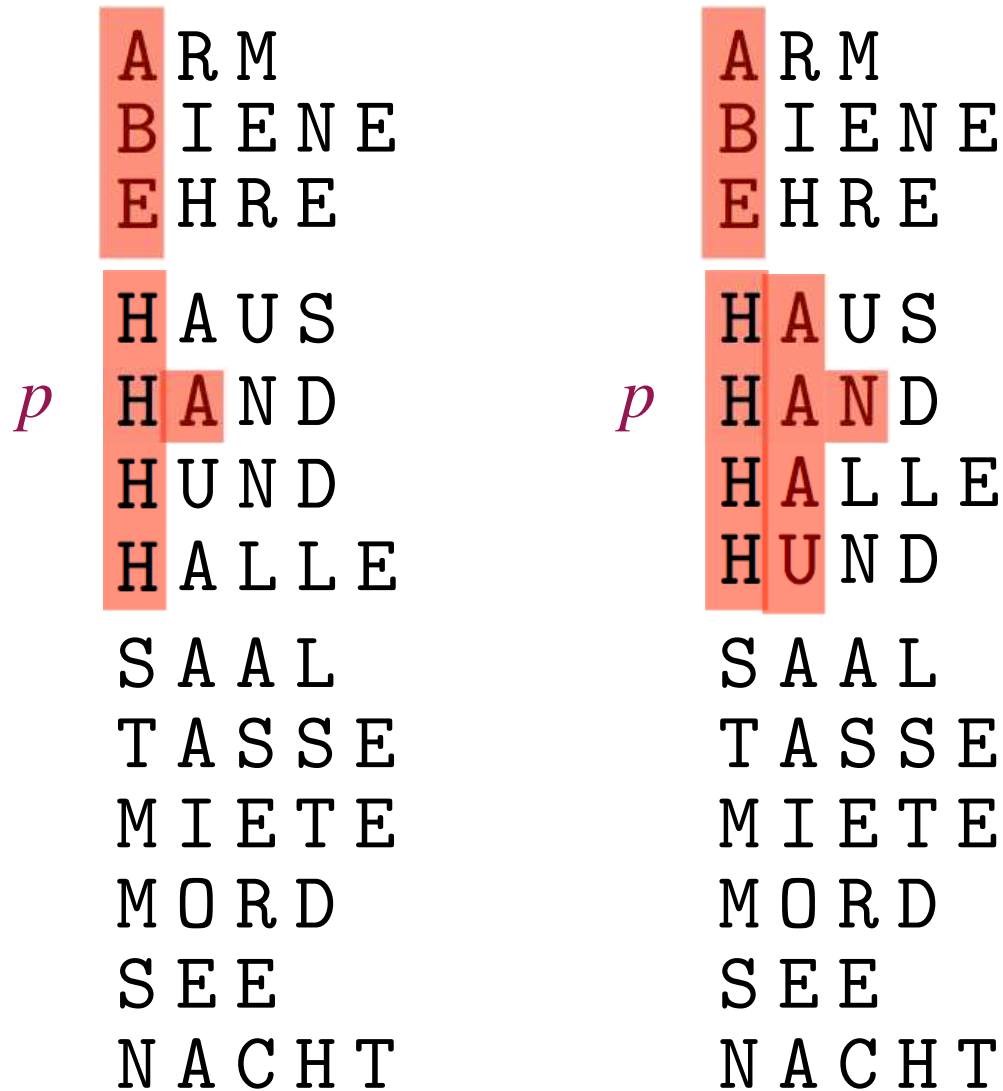
S E E

N A C H T

# Strings Sortieren: Multikey Quicksort

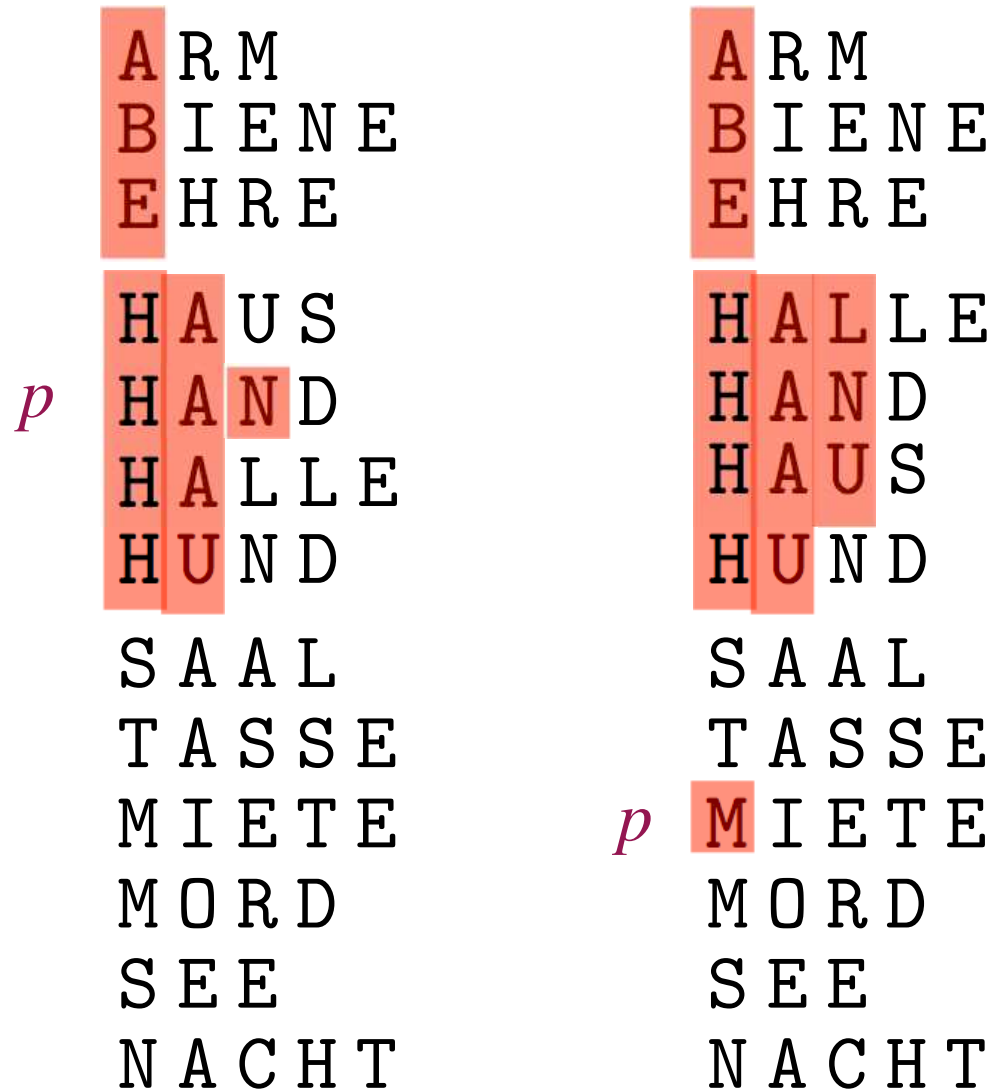
	B I E N E		A R M
<i>p</i>	E H R E		B I E N E
	A R M		E H R E
	H A U S		H A U S
	H A N D	<i>p</i>	H A N D
	H U N D		H U N D
	H A L L E		H A L L E
	S A A L		S A A L
	T A S S E		T A S S E
	M I E T E		M I E T E
	M O R D		M O R D
	S E E		S E E
	N A C H T		N A C H T

# Strings Sortieren: Multikey Quicksort

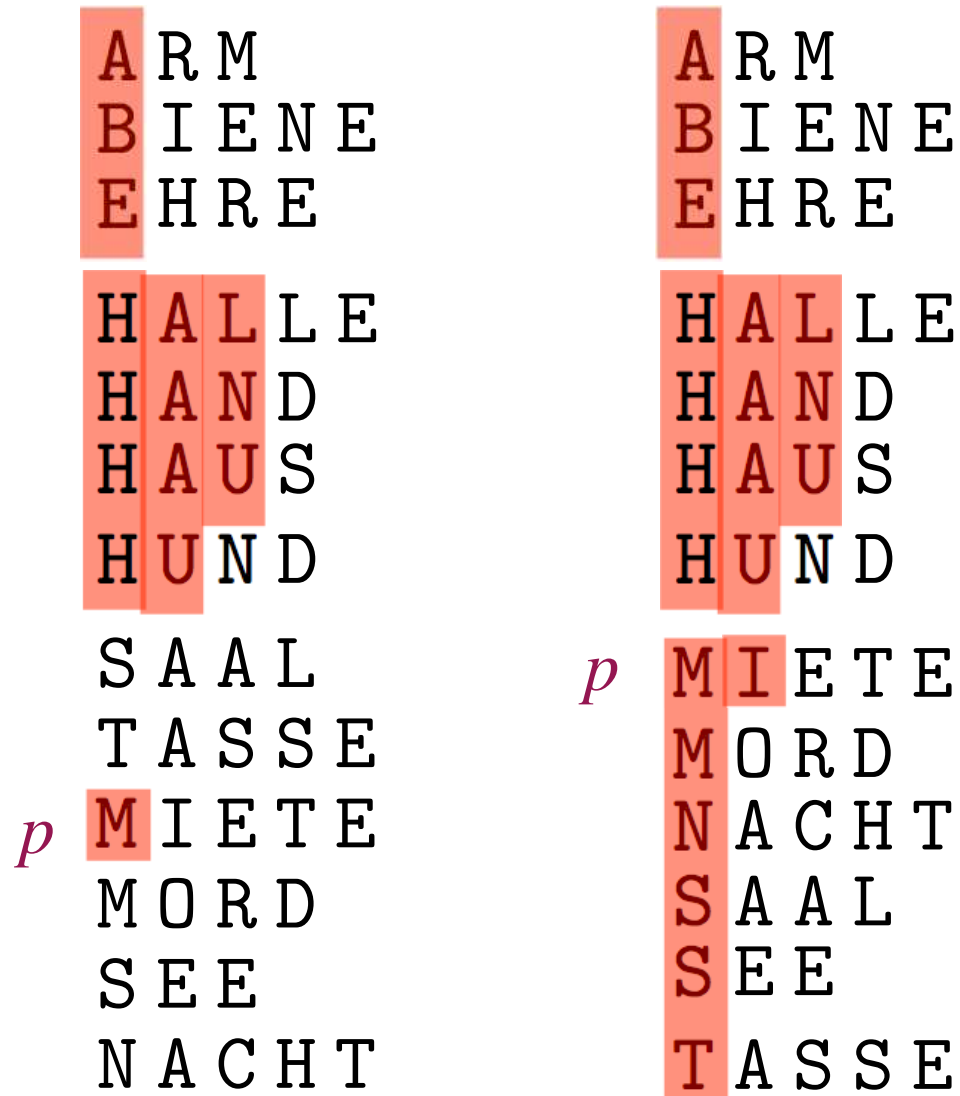




# Strings Sortieren: Multikey Quicksort



# Strings Sortieren: Multikey Quicksort



# Strings Sortieren: Multikey Quicksort

A R M  
B I E N E  
E H R E

H A L L E  
H A N D  
H A U S  
H U N D

*p* M I E T E  
M O R D  
N A C H T  
S A A L  
S E E  
T A S S E

A R M  
B I E N E  
E H R E

H A L L E  
H A N D  
H A U S  
H U N D

*p* M I E T E  
M O R D  
N A C H T  
S A A L  
S E E  
T A S S E

## Strings Sortieren (ohne Endzeichen)

**Function** mkqSort( $S$  : Sequence **of** String,  $\ell$  :  $\mathbb{N}$ ) : Sequence **of** String

**if**  $|S| \leq 1$  **then return**  $S$

$S_{\perp} \leftarrow \langle e \in S : |e| = \ell \rangle$ ;  $S \leftarrow S \setminus S_{\perp}$

select pivot  $p \in S$

$S_{<} \leftarrow \langle e \in S : e[\ell] < p[\ell] \rangle$

$S_{=} \leftarrow \langle e \in S : e[\ell] = p[\ell] \rangle$

$S_{>} \leftarrow \langle e \in S : e[\ell] > p[\ell] \rangle$

**return** concatenation of  $S_{\perp}$ ,

mkqSort( $S_{<}$ ,  $\ell$ ),

mkqSort( $S_{=}$ ,  $\ell + 1$ ), and

mkqSort( $S_{>}$ ,  $\ell$ )

# Strings Sortieren – Laufzeitanalyse

Hauptarbeit in den Buchstabenvergleichen. Zwei Fälle:

- $e[\ell] = p[\ell]$ : Ordne den Vergleich dem Zeichen  $e[\ell]$  zu.
  - $e[\ell]$  wird danach nicht mehr betrachtet (Rekursion mit  $\ell + 1$ )
  - Maximale Vergleichszahl pro String  $e$ ? Maximale Länge des längsten gemeinsamen Präfix von  $e$  mit  $e' \in S$ .
  
- $e[\ell] \neq p[\ell]$ : Ordne den Vergleich dem String  $e$  zu.
  - $e$  wird zu  $S_{<}$  oder  $S_{>}$  zugeordnet. Mit optimaler Pivotwahl sind beide Mengen höchstens  $|S|/2$ .
  - Nach höchstens  $\log |S|$  Schritten ist  $e$  richtig sortiert.

# Strings Sortieren: Algorithmen-Übersicht

## Sequentielle Basis-Algorithmen

- Radix Sort  $O(d + n \log \sigma)$  [McIlroy et al. '95]
- Multikey Quicksort  $O(d + n \log n)$  exp. [Bentley, Sedgewick '97]
- Burstsor  $O(d + n \log \sigma)$  exp. [Sinha, Zobel '04]
- Binary LCP-Mergesort  $O(d + n \log n)$  [Ng, Kakehi '08]

## Theoretische Parallele Algorithmen

- “Optimal Parallel String Algorithms: ...” [Hagerup '94]  
 $O(\log N / \log \log N)$  time and  $O(N \log \log N)$  work on CRCW PRAM

## Praktische Parallele und neue Basis-Algorithmen

- Parallel Super Scalar String Sample Sort (pS<sup>5</sup>) [B, Sanders, ESA'13]
- Parallel  $K$ -way LCP-aware Merge(sort) [B, et al. Algorithmica'17]

# Vergleich Sequentielle Algorithmen

**Experiment [B'18]:** Vergleich von 39 sequentiellen String Sorting Algorithmen und getunten Varianten auf **sieben Eingaben** und **sechs Maschinen**.

Unten: Repräsentative Auswahl der Basis-Algorithmen.

Rang	Algorithmus	GeoM	Rang	Algorithmus	GeoM
1	KR.radixsort-CE6	1.27	10	B.Seq-S <sup>5</sup> -UI	1.67
2	KRB.radixsort-CE3s	1.28	14	R.burstersort-vec	1.72
3	KR.radixsort-CE7	1.28	16	SZ.burstersortA	1.78
4	KRB.radixsort-CI3s	1.33	23	BS.mkqs	2.36
5	R.mkqs-cache8	1.34	28	NK.LCP-Mergesort	2.96
6	KR.radixsort-CE2	1.49	34	MBM.radixsort	4.82
7	KR.radixsort-DB	1.55	35	AN.ForwardRadix16	4.88

## Ergebnisse:

- **Hardware**-spezifische **Beschleunigungen** sind sehr wichtig.
- **Cachen** von Zeichen reduziert Random-Zugriffe aber kostet Speicher.

# Naives Pattern Matching

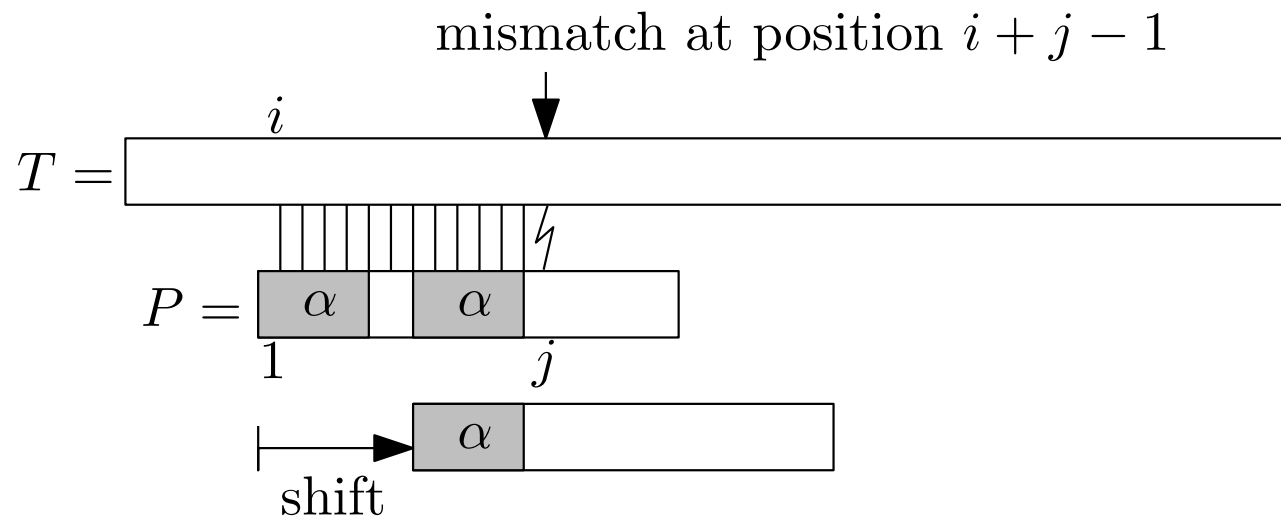
- Aufgabe: Finde alle Vorkommen von  $P$  in  $T$ 
  - $n$ : Länge von  $T$
  - $m$ : Länge von  $P$
- naiv in  $O(nm)$  Zeit

```
 $i, j := 1$  // indexes in  $T$  and  $P$   
while  $i \leq n - m + 1$   
    while  $j \leq m$  and  $t_{i+j-1} = p_j$  do  $j++$  // compare characters  
    if  $j > m$  then print " $P$  occurs at position  $i$  in  $T$ "  
     $i++$  // advance in  $T$   
     $j := 1$  // restart
```



# Knuth-Morris-Pratt (1977)

- besserer Algorithmus in  $O(n + m)$  Zeit
- Idee: beachte bereits gematchten Teil



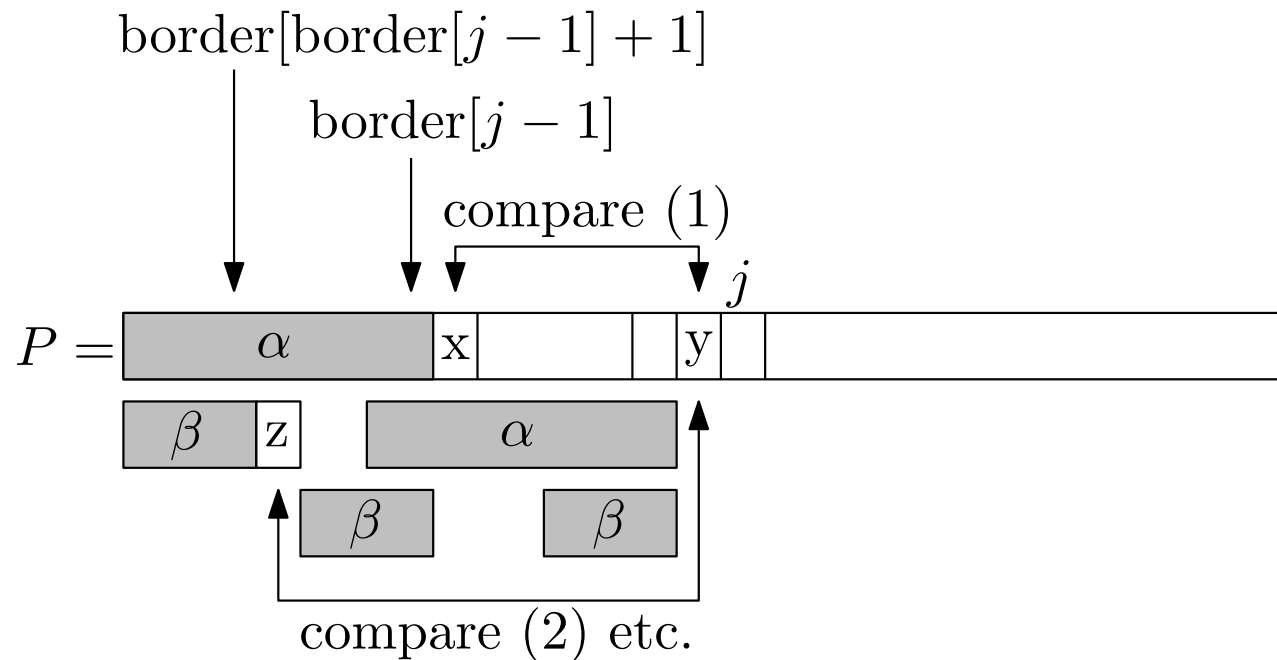
- $\text{border}[j] =$  längstes echtes Präfix von  $P_{1\dots j-1}$ , das auch (echtes) Suffix von  $P_{1\dots j-1}$  ist.  $\text{border}[1] := -1$ ,  $\text{border}[2] = 0$ .

## Knuth-Morris-Pratt (1977)

```
i := 1 // index in T
j := 1 // index in P
while  $i \leq n - m + 1$ 
    while  $j \leq m$  and  $t_{i+j-1} = p_j$  do  $j++$  // compare characters
    if  $j > m$  then
        print "P occurs at position i in T"
     $i := i + j - \text{border}[j] - 1$  // advance in T
     $j := \max\{1, \text{border}[j] + 1\}$  // skip first  $\text{border}[j]$  characters of P
```

# Berechnung des Border-Arrays

- seien die Werte bis zur Position  $j - 1$  bereits berechnet



## Berechnung des Border-Arrays

□ in  $O(m)$  Zeit:

$\text{border}[1] := -1$

$i := \text{border}[1]$

// position in  $P$

**for**  $j = 2, \dots, m + 1$

**while**  $i \geq 0$  and  $p_{i+1} \neq p_{j-1}$  **do**  $i = \text{border}[i + 1]$

$i++$

$\text{border}[j] := i$

# Volltextsuche von Langsam bis Superschnell

**Gegeben:** Text  $S$  ( $n := |S|$ ), Muster (Pattern)  $P$  ( $m := |P|$ ),  $n \gg m$

**Gesucht:** Alle/erstes/nächstes Vorkommen von  $P$  in  $S$

naiv:  $O(nm)$

$P$  vorverarbeiten:  $O(n + m)$

Mit Fehlern: ???

$S$  vorverarbeiten: Textindizes. Erstes Vorkommen:

Invertierter Index: gute heuristik

Suffix Array:  $O(m \log n) \dots O(m)$

# Invertierter Index

- 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

term $t$	$f_t$	Invertierte Liste für $t$
and	1	(6,2)
big	2	(2,2), (3,1)
dark	1	(6,1)
did	1	(4,1)
gown	1	(2,1)
had	1	(3,1)
house	2	(2,1), (3,1)
in	5	(1,1), (2,2), (3,1), (5,1), (6,2)
keep	3	(1,1), (3,1), (5,1)
...	...	...

# Etwas “Stringology”-Notation

Alphabet  $\Sigma$ : Menge  $\{a, b, c, \dots\}$ .

String  $S$ : Array  $S[0..n) := S[0..n - 1] := [S[0], \dots, S[n - 1]]$   
von Buchstaben aus  $\Sigma$ .

Suffix:  $S_i := S[i..n)$

Endmarkierungen:  $S[n] := S[n + 1] := \dots := 0$   
 $0$  ist kleiner als alle anderen Zeichen

$S = \text{banana} :$

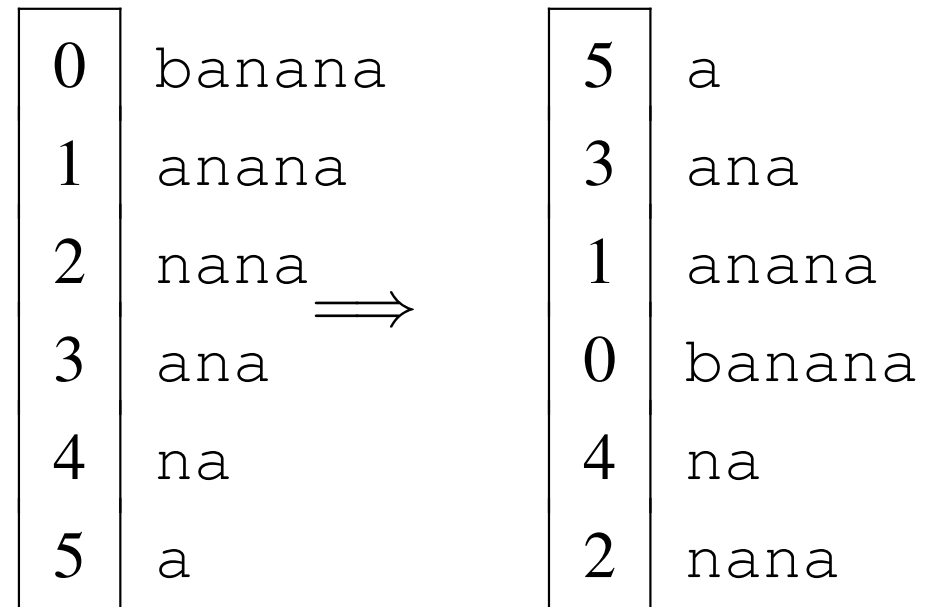
0	banana
1	anana
2	nana
3	ana
4	na
5	a

# Suffixe Sortieren

Sortiere die Menge  $\{S_0, S_1, \dots, S_{n-1}\}$   
 von Suffixen des Strings  $S$  der Länge  $n$   
 (Alphabet  $[1, n] = \{1, \dots, n\}$ )  
 in lexikographische Reihenfolge.

□ suffix  $S_i = S[i, n]$  für  $i \in [0..n - 1]$

$S = \text{banana}$ :





# Anwendungen

- Volltextsuche
- Burrows-Wheeler Transformation (bzip2 Kompressor)
- Ersatz für kompliziertere Suffixbäume
- Bioinformatik: Wiederholungen suchen (in  $O(n)$  möglich?) ,...

# Volltextsuche

Suche **Muster (pattern)**  $P[0..m)$  im Text  $S[0..n)$   
mittels Suffix-Tabelle  $SA$  of  $S$ .

Binäre Suche:  $O(m \log n)$  gut für kurze Muster

Binäre Suche mit lcp:  $O(m + \log n)$  falls wir die  
**längsten gemeinsamen (common) Präfixe**  
zwischen verglichenen Zeichenketten vorberechnen

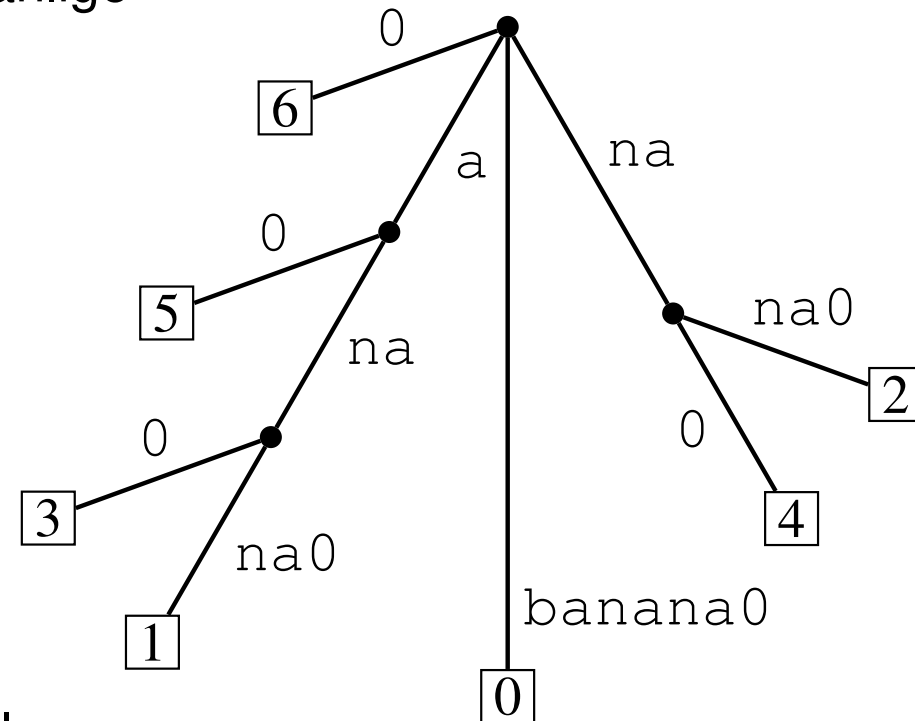
Suffix-Baum:  $O(n)$  kann aus **SA berechnet werden**

# Suffix-Baum

[Weiner '73][McCreight '76][Ukkonen '95]

- kompaktierter Trie der Suffixe
- + Zeit  $O(n)$  [Farach 97] für ganzzahlige Alphabete
- + Mächtigstes Werkzeug der Stringology?
- Hoher Platzverbrauch
- Effiziente direkte Konstruktion ist kompliziert
- kann aus SA in Zeit  $O(n)$  abgelesen werden

$S = \text{banana0}$



## Suche in Suffix-Bäumen

- Suche (alle/ein) Vorkommen von  $P_{1..m}$  in  $T$ :
- Wenn ausgehende Kanten Arrays der Größe  $|\Sigma|$ :
  - $O(m)$  Suchzeit
  - $O(n|\Sigma|)$  Gesamtplatz
- Wenn ausgehende Kanten Arrays der Größe prop. zur #Kinder:
  - $O(m \log |\Sigma|)$  Suchzeit
  - $O(n)$  Platz

# Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur **verglichen** werden

Konstante Alphabetgröße: endliche Menge  
deren Größe nicht von  $n$  abhängt.

**Ganzzahliges Alphabet:** Alphabet ist  $\{1, \dots, \sigma\}$   
für eine ganze Zahl  $\sigma \geq 2$

# Geordnetes $\rightarrow$ ganzzahliges Alphabet

Sortiere die Zeichen von  $S$

Ersetze  $S[i]$  durch seinen Rang

012345      135024

banana  $\rightarrow$  aaabnn

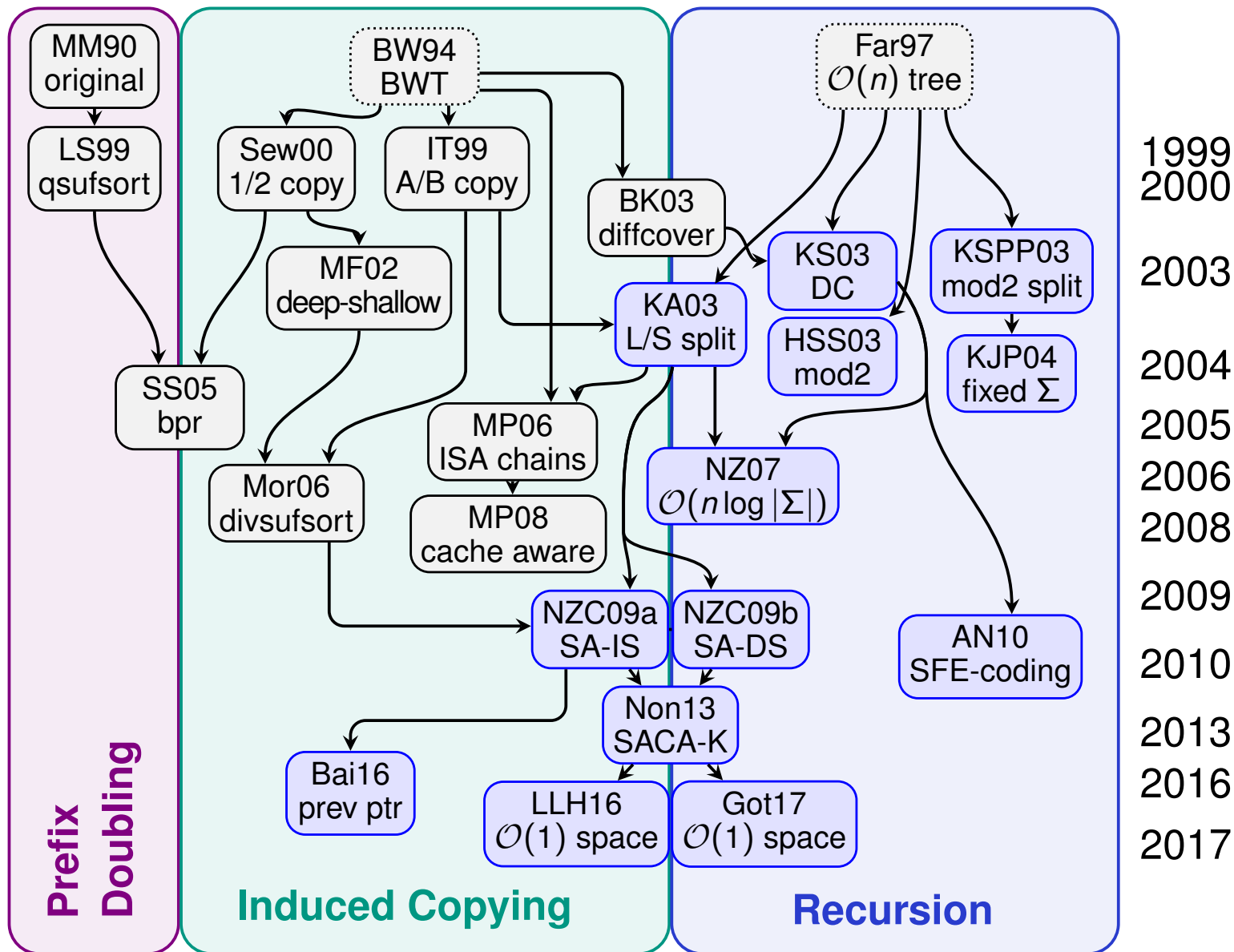
213131  $\leftarrow$  111233

## **Verallgemeinerung: Lexikographische Namen**

Sortiere die  $k$ -Tupel  $S[i..i+k)$  für  $i \in 1..n$

Ersetze  $S[i]$  durch den Rang von  $S[i..i+k)$  unter den Tupeln

# Suffix Array Konstruktionsalgorithmen





# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>0</sub> [i]	T <sub>i</sub>	ISA <sub>0</sub> [i]	L
0	0	t o b e o r n o t t o b e \$	0	-
1	1	o b e o r n o t t o b e \$	1	-
2	2	b e o r n o t t o b e \$	2	-
3	3	e o r n o t t o b e \$	3	-
4	4	o r n o t t o b e \$	4	-
5	5	r n o t t o b e \$	5	-
6	6	n o t t o b e \$	6	-
7	7	o t t o b e \$	7	-
8	8	t t o b e \$	8	-
9	9	t o b e \$	9	-
10	10	o b e \$	10	-
11	11	b e \$	11	-
12	12	e \$	12	-
13	13	\$	13	-

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>1</sub> [i]	T <sub>i</sub>	ISA <sub>1</sub> [i]	L
0	13	0 \$	11	-1
1	2	1 b e o r n o t t o b e \$	6	2
2	11	1 b e \$	1	-
3	3	3 e o r n o t t o b e \$	3	2
4	12	3 e \$	6	-
5	6	5 n o t t o b e \$	10	-1
6	1	6 o b e o r n o t t o b e \$	5	4
7	4	6 o r n o t t o b e \$	6	-
8	7	o t t o b e \$	11	-
9	10	o b e \$	11	-
10	5	10 r n o t t o b e \$	6	-1
11	0	11 t o b e o r n o t t o b e \$	1	3
12	8	t t o b e \$	3	-
13	9	t o b e \$	0	-

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>1</sub> [i]	T <sub>i</sub>	ISA <sub>1</sub> [i]	L
0	13	0 \$	11	-1
1	2	1 b e 3	6	2
2	11	1 b e 3	1	-
3	3	3 e o 6	3	2
4	12	3 e \$ 0	6	-
5	6	5 n o t t o b e \$	10	-1
6	1	6 o b 1	5	4
7	4	6 o r 10	6	-
8	7	6 o t 11	11	-
9	10	6 o b 1	11	-
10	5	10 r n o t t o b e \$	6	-1
11	0	11 t o 6	1	3
12	8	11 t t 11	3	-
13	9	11 t o 6	0	-

ISA[SA[i] + 1]

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>2</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 1]	ISA <sub>1</sub> [i]	L
0	13	0	\$	11	-1
1	2	1	b e 3	6	2
2	11	1	b e 3	1	-
3	12	3	e \$ 0	3	-3
4	3	3	e o 6	6	-
5	6	5	n o t t o b e \$	10	-
6	1	6	o b 1 e o r n o t t o b e \$	5	2
7	10	6	o b 1 e \$	6	-
8	4	10	o r 10 o t t o b e \$	11	-3
9	7	10	o t 11 o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	0	11	t o 6 o e o r n o t t o b e \$	1	2
12	9	11	t o 6 o e \$	3	-
13	8	11	t t 11 o b e \$	0	-1

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>2</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 1]	ISA <sub>2</sub> [i]	L
0	13	0	\$	11	-1
1	2	1	b e 3	6	2
2	11	1	b e 3	1	-
3	12	3	e \$ 0	4	-3
4	3	4	e o 6	8	-
5	6	5	n o t t o b e \$	10	-
6	1	6	o b 1	5	2
7	10	6	o b 1	9	-
8	4	8	o r 10	13	-3
9	7	9	o t 11	11	-
10	5	10	r n o t t o b e \$	6	-
11	0	11	t o 6	1	2
12	9	11	t o 6	3	-
13	8	13	t t 11	0	-1

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>2</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 2]	ISA <sub>2</sub> [i]	L
0	13	0	\$	11	-1
1	2	1	b e o r n o t t o b e \$	6	2
2	11	1	b e \$	1	-
3	12	3	e \$	4	-3
4	3	4	e o r n o t t o b e \$	8	-
5	6	5	n o t t o b e \$	10	-
6	1	6	o b e o r n o t t o b e \$	5	2
7	10	6	o b e \$	9	-
8	4	8	o r n o t t o b e \$	13	-3
9	7	9	o t t o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	0	11	t o b e o r n o t t o b e \$	1	2
12	9	11	t o b e \$	3	-
13	8	13	t t o b e \$	0	-1

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>4</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 2]	ISA <sub>2</sub> [i]	L
0	13	0	\$	11	-11
1	11	1	b e \$0	6	-
2	2	1	b e o8r n o t t o b e \$	1	-
3	12	3	e \$	4	-
4	3	4	e o r n o t t o b e \$	8	-
5	6	5	n o t t o b e \$	10	-
6	10	6	o b e3\$	5	-
7	1	6	o b e4o r n o t t o b e \$	9	-
8	4	8	o r n o t t o b e \$	13	-
9	7	9	o t t o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	0	11	t o b1e o r n o t t o b e \$	1	2
12	9	11	t o b1e \$	3	-
13	8	13	t t o b e \$	0	-1

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>4</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 2]	ISA <sub>4</sub> [i]	L
0	13	0	\$	11	-11
1	11	1	b e \$0	7	-
2	2	2	b e o8r n o t t o b e \$	2	-
3	12	3	e \$	4	-
4	3	4	e o r n o t t o b e \$	8	-
5	6	5	n o t t o b e \$	10	-
6	10	6	o b e3\$	5	-
7	1	7	o b e4o r n o t t o b e \$	9	-
8	4	8	o r n o t t o b e \$	13	-
9	7	9	o t t o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	0	11	t o b1e o r n o t t o b e \$	1	2
12	9	11	t o b1e \$	3	-
13	8	13	t t o b e \$	0	-1



# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>4</sub> [i]	T <sub>i</sub>	ISA <sub>4</sub> [i]	L
0	13	0	11	-11
1	11	1	7	-
2	2	2	2	-
3	12	3	4	-
4	3	4	8	-
5	6	5	10	-
6	10	6	5	-
7	1	7	9	-
8	4	8	13	-
9	7	9	11	-
10	5	10	6	-
11	0	11	1	2
12	9		3	-
13	8	13	0	-1

T <sub>i</sub>	ISA[SA[i] + 4]
0	\$
1	b e \$
2	b e o r n o t t o b e \$
3	e \$
4	e o r n o t t o b e \$
5	n o t t o b e \$
6	o b e \$
7	o b e o r n o t t o b e \$
8	o r n o t t o b e \$
9	o t t o b e \$
10	r n o t t o b e \$
11	t o b e o r n o t t o b e \$
12	t o b e \$
13	t t o b e \$

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>8</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 4]	ISA <sub>4</sub> [i]	L
0	13	0	\$	11	-14
1	11	1	b e \$	7	-
2	2	2	b e o r n o t t o b e \$	2	-
3	12	3	e \$	4	-
4	3	4	e o r n o t t o b e \$	8	-
5	6	5	n o t t o b e \$	10	-
6	10	6	o b e \$	5	-
7	1	7	o b e o r n o t t o b e \$	9	-
8	4	8	o r n o t t o b e \$	13	-
9	7	9	o t t o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	9	11	t o b e \$ 0	3	-
12	0	12	t o b e o r 8 n o t t o b e \$	1	-
13	8	13	t t o b e \$	0	-

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

i	SA <sub>8</sub> [i]	T <sub>i</sub>	ISA[SA[i] + 4]	ISA <sub>8</sub> [i]	L
0	13	0	\$	12	-14
1	11	1	b e \$	7	-
2	2	2	b e o r n o t t o b e \$	2	-
3	12	3	e \$	4	-
4	3	4	e o r n o t t o b e \$	8	-
5	6	5	n o t t o b e \$	10	-
6	10	6	o b e \$	5	-
7	1	7	o b e o r n o t t o b e \$	9	-
8	4	8	o r n o t t o b e \$	13	-
9	7	9	o t t o b e \$	11	-
10	5	10	r n o t t o b e \$	6	-
11	9	11	t o b e \$ 0	3	-
12	0	12	t o b e o r 8 n o t t o b e \$	1	-
13	8	13	t t o b e \$	0	-

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

**Idee (aus [MM93]):** Sortiere Suffixe bis zu  $2h$  Zeichen tief mittels Informationen über die Ordnung der Suffixe bis zur Tiefe  $h$ .

**Def.:** Die  $h$ -Ordnung ist die lexikographische Ordnung bis zur Tiefe  $h$ .

**Original [MM93]:** Sortiere alle Suffix  $s_i$  mit dem Rang von  $s_i$  in einer  $h$ -Ordnung als ersten Sortier-Schlüssel und dem Rang von  $s_i + h$  als zweiten Sortier-Schlüssel. Dies ergibt eine  $2h$ -Ordnung.

**Definitionen [LS99]:** Wenn  $SA_h$  in einer  $h$ -Ordnung sind, dann

1. nennt man eine maximale Sequenz aufeinanderfolgender Suffixe in  $SA_h$  mit den gleichen  $h$  initialen Zeichen eine  $h$ -Gruppe.
2. Eine  $h$ -Gruppe der Länge eins nennt man sortiert oder Singleton, anderenfalls heißt eine Gruppe unsortiert. Eine maximale Sequenz von sortierten Gruppen heißt eine konsolidierte sortierte Gruppe.

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

Algorithmus [LS'99]: Präfix Verdopplung

Gegeben ist ein Text  $T$  der Länge  $n := |T|$ .

1. Enumeriere alle Suffixe in einem Array  $SA_0$  durch  $SA_0[i] = i$ .
2. Sortiere  $SA_0$  nach  $T[i]$  für Index  $i$  und erhalte  $SA_1$ . Setze  $h = 1$ .
3. Für  $i = 0, \dots, n$  setze  $ISA_1[i]$  auf den aktuellen Rang der 1-Gruppe von Suffix  $i$ . Dies ist der kleinste Index in  $SA_1[i]$  der ein Suffix mit dem gleichen ersten Buchstaben wie das Suffix  $i$ .
4. Für jede unsortierte Gruppe in  $SA_1[a..b]$  setze  $L[a]$  auf dessen Länge. Für jede konsolidierte sortierte Gruppe in  $SA_1[a..b]$ , setze  $L[a]$  auf seine negierte Länge.

## SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

- Sortiere jede unsortierte Gruppe in  $SA_h$  (mit multikey quicksort), mittels  $ISA_h[SA_h[i] + h]$  als den Sortier-Schlüssel für Index  $i$ . Verwende  $L$  um die unsortierten Gruppen zu finden. Erhalte  $SA_{2h}$ .
- Betrachte Folgen gleicher Schlüssel in den eben sortierten Gruppen: markiere Singletons mit  $-1$  in  $L$  und unsortierte  $2h$ -Gruppen mit positiven Längen. Konsolidiere die negativen Sprungwerte aufeinanderfolgende sortierte Gruppen.
- Aktualisiere  $ISA_{2h}$  für alle bearbeiteten Gruppen, setze  $h := 2h$ .
- Wenn  $SA_h$  noch unsortierte Gruppen enthält ( $L[0] \neq -n$ ), goto 5.

Laufzeit:  $\mathcal{O}(n \log n)$ . Platz:  $8n (+ 1n)$ . Worst-case Eingabe:  $[a^n]$ .

# Suffixtabellen

aus

# Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

## Ein erster Teile-und-Herrsche-Ansatz

1.  $SA^1 = \text{sort} \{S_i : i \text{ ist ungerade}\}$  (Rekursion)
2.  $SA^0 = \text{sort} \{S_i : i \text{ ist gerade}\}$  (einfach mittels  $SA^1$ )
3. Mische  $SA^0$  und  $SA^1$  (schwierig)

Problem: wie vergleicht man gerade und ungerade Suffixe?

[Farach 97] hat einen Linearzeitalgorithmus für

Suffix-**Baum**-Konstruktion entwickelt, der auf dieser Idee beruht.

Sehr **kompliziert**.

Das war auch der einzige bekannte Algorithmus für Suffix-**Tabellen**

(läßt sich leicht aus S-Baum ablesen.)



## $SA^1$ berechnen

- Erstes Zeichen weglassen.

banana  $\rightarrow$  anana

- Ersetze Buchstabenpaare durch Ihre **lexikographischen Namen**

an	an	a0
----	----	----

 $\rightarrow$  221

- Rekursion

$\langle 1, 21, 221 \rangle$

- Rückübersetzen

$\langle a, ana, anana \rangle$

# Berechne $SA^0$ aus $SA^1$

1	anana	⇒	5	a
3	ana		3	ana
5	a		1	anana

Ersetze  $S_i, i \bmod 2 = 0$  durch  $(S[i], r(S_{i+1}))$

mit  $r(S_{i+1}) :=$  Rang von  $S_{i+1}$  in  $SA^1$

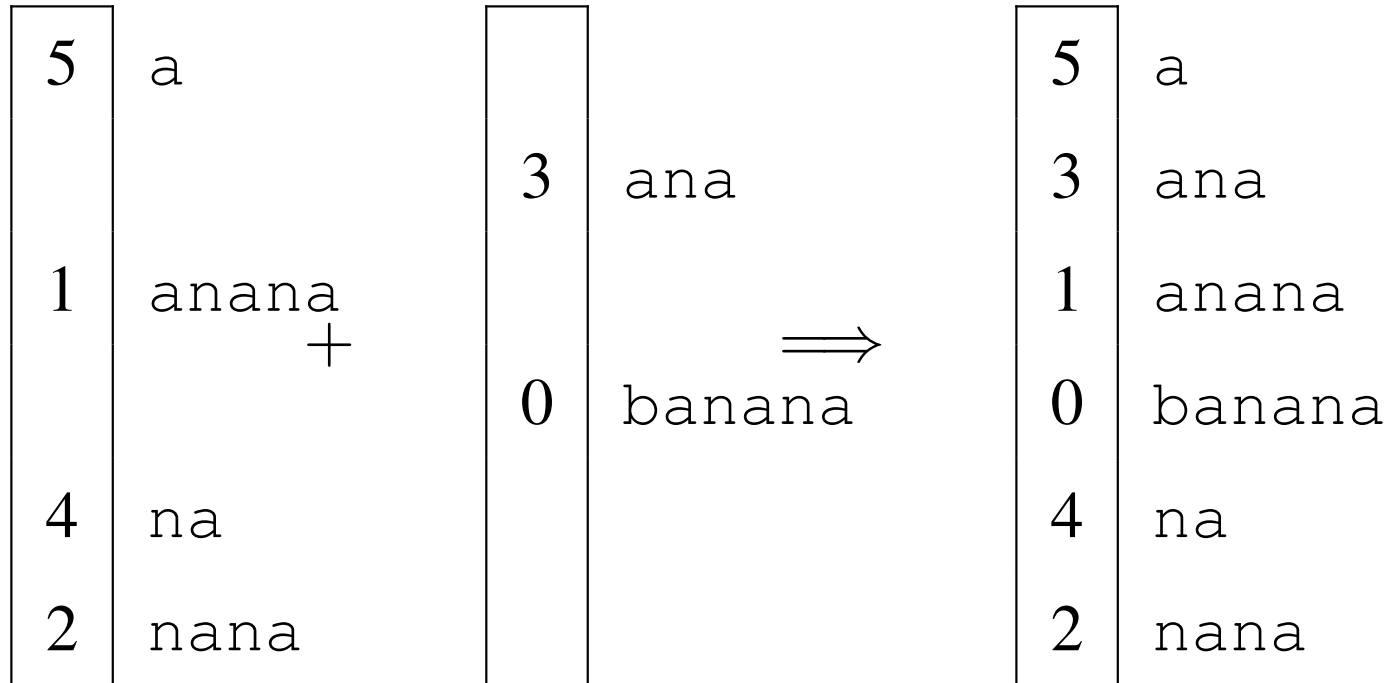
0	b 3 (anana)	⇒	0	banana
2	n 2 (ana)		4	na
4	n 1 (a)		2	nana

Radix-Sort

# Asymmetrisches Divide-and-Conquer

1.  $SA^{12}$  = sort  $\{S_i : i \bmod 3 \neq 0\}$  (Rekursion)
2.  $SA^0$  = sort  $\{S_i : i \bmod 3 = 0\}$  (einfach mittels  $SA^{12}$ )
3. Mische  $SA^{12}$  und  $SA^0$  (einfach!)

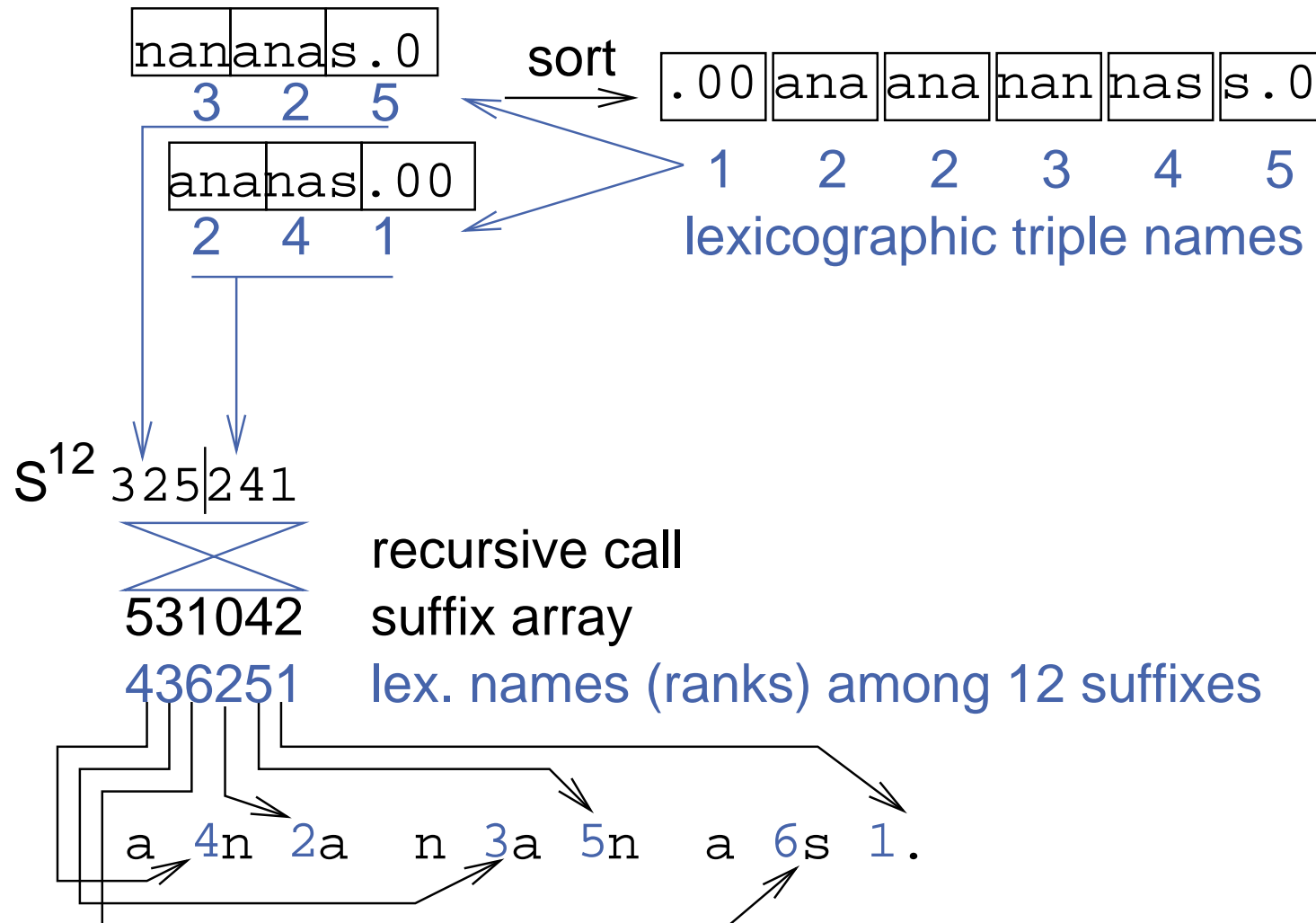
$S = \text{banana}$



# Rekursion, Beispiel

012345678

S anananas.



# Rekursion

- **sortiere Tripel**  $S[i..i+2]$  für  $i \bmod 3 \neq 0$   
(LSD Radix-Sortieren)
- Finde **lexikographische Namen**  $S'[1..2n/3]$  der Tripel,  
(d.h.,  $S'[i] < S'[j]$  gdw  $S[i..i+2] < S[j..j+2]$ )
- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$ ,  
Suffix  $S_i^{12}$  von  $S^{12}$  repräsentiert  $S_{3i+1}$   
Suffix  $S_{n/3+i}^{12}$  von  $S^{12}$  repräsentiert  $S_{3i+2}$
- **Rekursion** auf  $(S^{12})$  (Alphabetgröße  $\leq 2n/3$ )
- Annotiere die 12-Suffixe mit ihrer Position in rek. Lösung

# Least Significant Digit First Radix Sort

Hier: Sortiere  $n$  3-Tupel von ganzen Zahlen  $\in [0..n]$  in  
**lexikographische** Reihenfolge

Sortiere nach 3. Position

Elemente sind nach Pos. 3 sortiert

Sortiere **stabil** nach 2. Position

Elemente sind nach Pos. 2,3 sortiert

Sortiere **stabil** nach 1. Position

Elemente sind nach Pos. 1,2,3 sortiert

# Stabiles Ganzzahliges Sortieren

Sortiere  $a[0..n)$  nach  $b[0..n)$  mit  $\text{key}(a[i]) \in [0..n]$

$c[0..n] := [0, \dots, 0]$

for  $i \in [0..n)$  do  $c[a[i]]++$

$s := 0$

for  $i \in [0..n)$  do  $(s, c[i]) := (s + c[i], s)$

for  $i \in [0..n)$  do  $b[c[a[i]]++] := a[i]$

Zähler

zähle

Präfixsummen

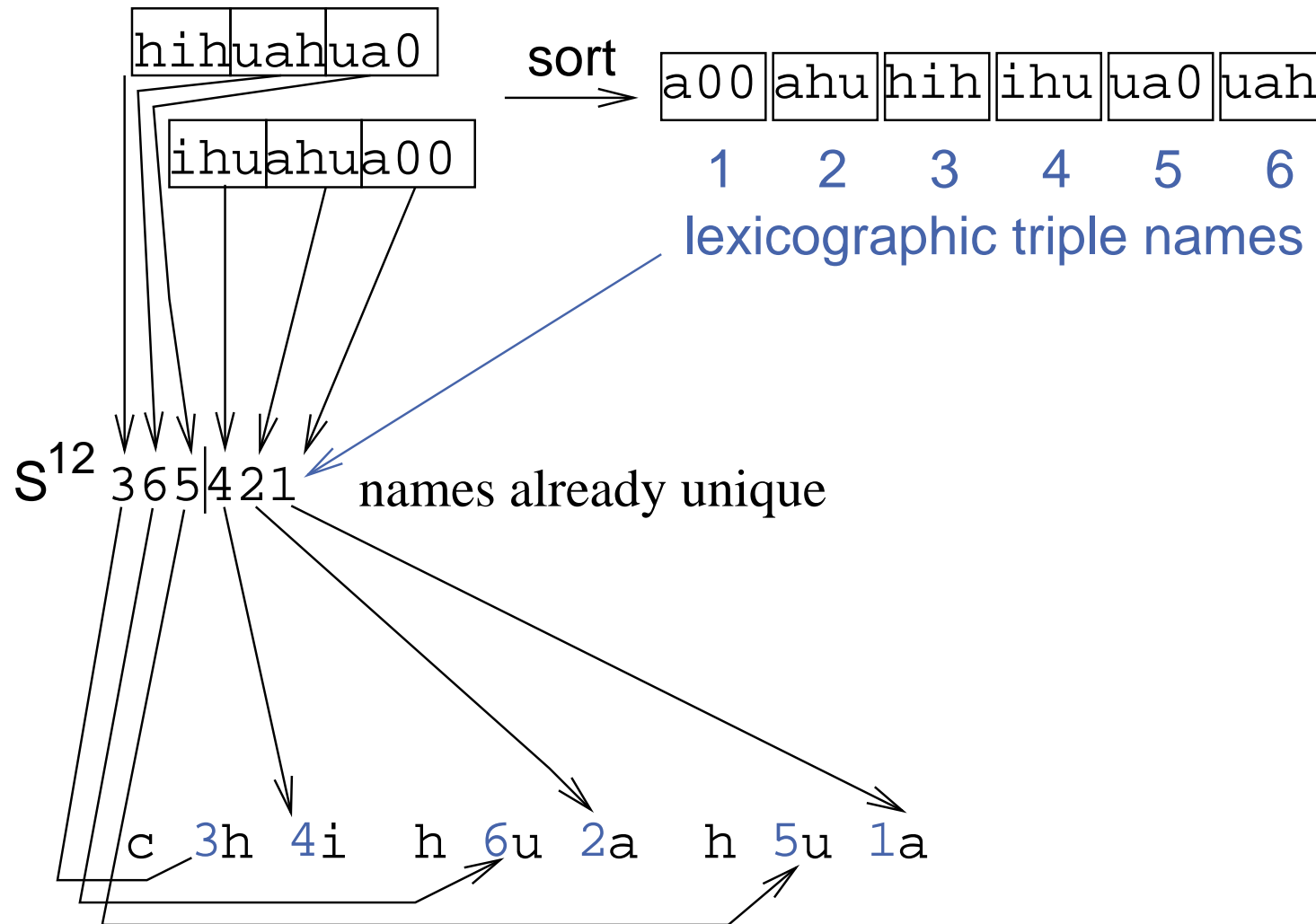
bucket sort

Zeit  $O(n)$  !

# Rekursions-Beispiel: Einfacher Fall

012345678

S chihuahua





## Sortieren der mod 0 Suffixe

0	c	3	(h	4	i	h	6	u	2	a	h	5	u	1	a)
1															
2															
3	h	6	(u	2	a	h	5	u	1	a)					
4															
5															
6	h	5	(u	1	a)										
7															
8															

Benutze Radix-Sort (LSD-Reihenfolge bereits bekannt)

# Mische $SA^{12}$ und $SA^0$

$0 < 1 \Leftrightarrow c_n < c_n$	4:	$(6) u \ 2$ (ahua)
$0 < 2 \Leftrightarrow cc_n < cc_n$	7:	$(5) u \ 1$ (a)
3: h $6$ u $2$ (ahua)	2:	$(4) i \ h \ 6$ (uahua)
6: h $5$ u $1$ (a)	1:	$(3) h \ 4$ (ihuahua)
0: c $3$ h $4$ (ihuahua)	5:	$(2) a \ h \ 5$ (ua)
	8:	$(1) a \ 0 \ 0 \ 0$ (0)

⇓

- 8: a
- 5: ahua
- 0: chihuahua
- 1: hihuahua
- 6: hua
- 3: huahua
- 2: ihuahua
- 7: ua
- 4: uahua

# Analyse

1. Rekursion:  $T(2n/3)$  plus

Tripel extrahieren:  $O(n)$  (forall  $i, i \bmod 3 \neq 0$  do ...)

Tripel sortieren:  $O(n)$

(e.g., LSD-first radix sort — 3 Durchgänge)

Lexikographisches benennen:  $O(n)$  (scan)

Rekursive Instanz konstruieren:  $O(n)$  (forall names do ...)

2.  $SA^0$  =sortiere  $\{S_i : i \bmod 3 = 0\}$ :  $O(n)$

(1 Radix-Sort Durchgang)

3. mische  $SA^{12}$  and  $SA^0$ :  $O(n)$

(gewöhnliches Mischen mit merkwürdiger Vergleichsfunktion)

Insgesamt:  $T(n) \leq cn + T(2n/3)$

$\Rightarrow T(n) \leq 3cn = O(n)$

# Implementierung: Vergleichs-Operatoren

```
inline bool leq(int a1, int a2,    int b1, int b2) {  
    return(a1 < b1 || a1 == b1 && a2 <= b2);  
}  
inline bool leq(int a1, int a2, int a3,    int b1, int b2, int b3) {  
    return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));  
}
```

# Implementierung: Radix-Sortieren

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];           // counter array
  for (int i = 0; i <= K; i++) c[i] = 0; // reset counters
  for (int i = 0; i < n; i++) c[r[a[i]]]++; // count occurrences
  for (int i = 0, sum = 0; i <= K; i++) { // exclusive prefix sums
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i]; // sort
  delete [] c;
}
```

# Implementierung: Tripel Sortieren

```
void suffixArray(int* s, int* SA, int n, int K) {
    int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
    int* s12 = new int[n02 + 3]; s12[n02]=s12[n02+1]=s12[n02+2]=0;
    int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
    int* s0 = new int[n0];
    int* SA0 = new int[n0];

    // generate positions of mod 1 and mod 2 suffixes
    // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
    for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) s12[j++] = i;

    // lsb radix sort the mod 1 and mod 2 triples
    radixPass(s12 , SA12, s+2, n02, K);
    radixPass(SA12, s12 , s+1, n02, K);
    radixPass(s12 , SA12, s , n02, K);
}
```

# Implementierung: Lexikographisches Benennen

```
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0; i < n02; i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
        name++; c0 = s[SA12[i]]; c1 = s[SA12[i]+1]; c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3] = name; } // left half
    else { s12[SA12[i]/3 + n0] = name; } // right half
}
```

# Implementierung: Rekursion

```
// recurse if names are not yet unique
if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
    for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;
```



# Implementierung: Sortieren der mod 0 Suffixe

```
for (int i=0, j=0; i < n02; i++)  
    if (SA12[i] < n0) s0[j++] = 3*SA12[i];  
radixPass(s0, SA0, s, n0, K);
```

# Implementierung: Mischen

```
for (int p=0, t=n0-n1, k=0; k < n; k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0 suffix
    if (SA12[t] < n0 ?
        leq(s[i], s12[SA12[t] + n0], s[j], s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    { // suffix from SA12 is smaller
        SA[k] = i; t++;
        if (t == n02) { // done --- only SA0 suffixes left
            for (k++; p < n0; p++, k++) SA[k] = SA0[p];
        }
    } else {
        SA[k] = j; p++;
        if (p == n0) { // done --- only SA12 suffixes left
            for (k++; t < n02; t++, k++) SA[k] = GetI();
        }
    }
}
delete [] s12; delete [] SA12; delete [] SA0; delete [] s0; }
```

# Verallgemeinerung: Differenzenüberdeckungen

Ein **Differenzenüberdeckung**  $D$  modulo  $v$  ist eine Teilmenge von  $[0, v)$ ,  
so dass  $\forall i \in [0, v) : \exists j, k \in D : i \equiv k - j \pmod{v}$ .

Beispiel:

$\{1, 2\}$  ist eine Differenzenüberdeckung modulo 3.

$\{1, 2, 4\}$  ist eine Differenzenüberdeckung modulo 7.

- Führt zu platzeffizienterer Variante
- Schneller für kleine Alphabete

## Verbesserungen / Verallgemeinerungen

- tuning
- größere **Differenzenüberdeckungen**
- Kombiniere mit den besten Alg. für einfache Eingaben  
[Manzini Ferragina 02, Schürmann Stoye 05, Yuta Mori 08]

# Suffixtabellenkonstruktion: Zusammenfassung

- einfache, direkte, Linearzeit für Suffixtabellenkonstruktion
- einfach anpassbar auf fortgeschrittene Berechnungsmodelle
- Verallgemeinerung auf Diff-Überdeckungen ergibt platzeffiziente Implementierung

## Suche in Suffix Arrays

Given:  $T$ ,  $SA$ ,  $P$ .

$l := 1; r := n + 1$

**while**  $l < r$  **do** //search left index

$q := \lfloor \frac{l+r}{2} \rfloor$

**if**  $P >_{\text{lex}} T_{SA[q] \dots \min\{SA[q]+m-1, n\}}$

**then**  $l := q + 1$  **else**  $r := q$

$s := l; l--; r := n$

**while**  $l < r$  **do** //search right index

$q := \lceil \frac{l+r}{2} \rceil$

**if**  $P =_{\text{lex}} T_{SA[q] \dots \min\{SA[q]+m-1, n\}}$

**then**  $l := q$  **else**  $r := q - 1$

**return**  $[s, l]$

□ Zeit  $O(m \log n)$  (geht auch:  $O(m + \log n)$ )

# LCP-Array

speichert Längen der **längsten gemeinsamen Präfixe** lexikographisch benachbarter Suffixe!

$S = \text{banana}$ :

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

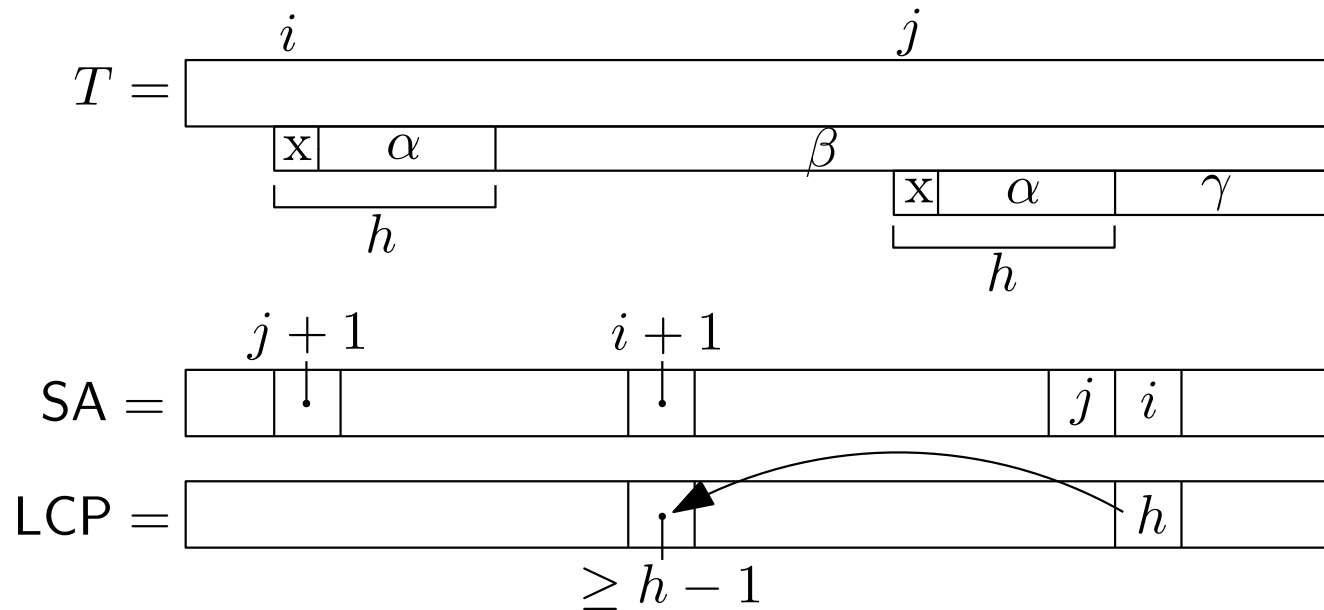
5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

⊥	a
1	<b>a</b> na
3	<b>ana</b> na
0	banana
0	na
2	<b>na</b> na

# LCP-Array: Berechnung

- **naiv**  $O(n^2)$
- inverses Suffix-Array:  $SA^{-1}[SA[i]] = i$  (wo steht  $i$  in SA?)
- For all  $1 \leq i < n$ :  $LCP[SA^{-1}[i+1]] \geq LCP[SA^{-1}[i]] - 1$ .





## LCP-Array: Berechnung

□ For all  $1 \leq i < n$ :  $\text{LCP}[\text{SA}^{-1}[i+1]] \geq \text{LCP}[\text{SA}^{-1}[i]] - 1$ .

$h := 0, \text{LCP}[1] := 0$

**for**  $i = 1, \dots, n$  **do**

**if**  $\text{SA}^{-1}[i] \neq 1$  **then**

**while**  $t_{i+h} = t_{\text{SA}[\text{SA}^{-1}[i]-1]+h}$  **do**  $h++$

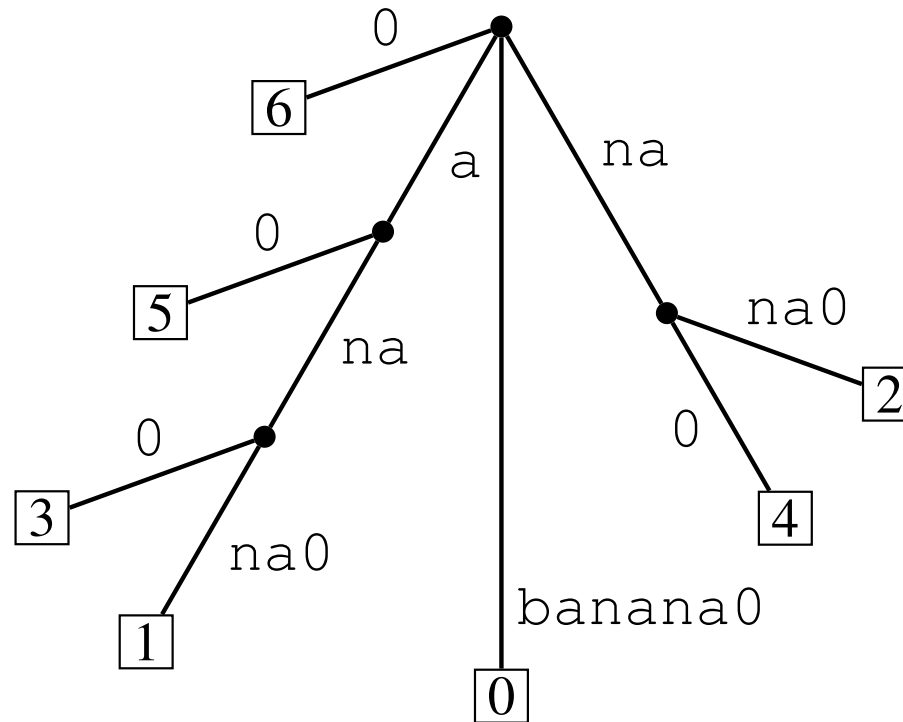
$\text{LCP}[\text{SA}^{-1}[i]] := h$

$h := \max(0, h - 1)$

□ Zeit:  $O(n)$

# Suffix-Baum aus SA und LCP

$S = \text{banana0}$



□ naiv:  $O(n^2)$

□ mit Suffix-Array: in **lexikographischer Reihenfolge**

# Suffix-Baum aus SA und LCP

- LCP-Werte helfen!
- Betrachte nur **rechtesten Pfad!**
- Finde tiefsten Knoten mit String-Tiefe  $\leq \text{LCP}[i] \rightsquigarrow$  **Einfügepunkt!**

	0	1	2	3	4	5	6
SA =	6	5	3	1	0	4	2
LCP =	0	0	1	3	0	0	2

- Zeit  $O(n)$

# Datenkompression

Problem: bei naiver Speicherung verbrauchen Daten sehr viel **Speicherplatz / Kommunikationsbandbreite**. Das läßt sich oft reduzieren.

## Varianten:

- Verlustbehaftet (mp3, jpg, ...) / **Verlustfrei** (Text, Dateien, Suchmaschinen, Datenbanken, medizin. Bildverarbeitung, Profifotografie, ...)
- 1D** (text, Zahlenfolgen, ...) / 2D (Bilder) / 3D (Filme)
- nur Speicherung** / mit Operationen ( $\rightsquigarrow$  succinct data structures)

# Verlustfreie Textkompression

**Gegeben:** Alphabet  $\Sigma$

String  $S = \langle s_1, \dots, s_n \rangle \in \Sigma^*$

Textkompressionsalgorithmus  $f : S \rightarrow f(S)$  mit  $|f(S)|$  (z.B. gemessen in bits) möglichst klein.

# Theorie Verlustfreier Textkompression

Informationstheorie. Zum Beispiel

**Entropie:**  $H(S) = \sum_{c \in \Sigma} p(c) \log(1/p(c))$  wobei

$p(c) = |\{s_i : s_i = c\}|/n$  die relative Häufigkeit von  $c$  ist.

untere Schranke für # bits pro Zeichen falls Text einer Zufallsquelle entspränge.

↪ Huffman-Coding ist annähernd optimal! (Entropiecodierung) ????

Schon eher:

**Entropie höherer Ordnung** betrachte Teilstrings fester Länge

“Ultimativ”: Kolmogorov Komplexität. Leider nicht berechenbar.

# Theorie Verlustfreier Textkompression

**Entropie höherer Ordnung:** Gegeben ein Text  $S$  der Länge  $n$  über dem Alphabet  $\Sigma$ . Wir definieren  $N(\omega, S)$  als Konkatenation aller Zeichen, die in  $S$  auf Vorkommen von  $\omega \in \Sigma^k$  folgen. Wir definieren die **empirische Entropie der Ordnung  $k$**  wie folgt:

$$H_k(S) = \sum_{\omega \in \Sigma^k} \frac{|N(\omega, S)|}{n} H(N(\omega, S))$$

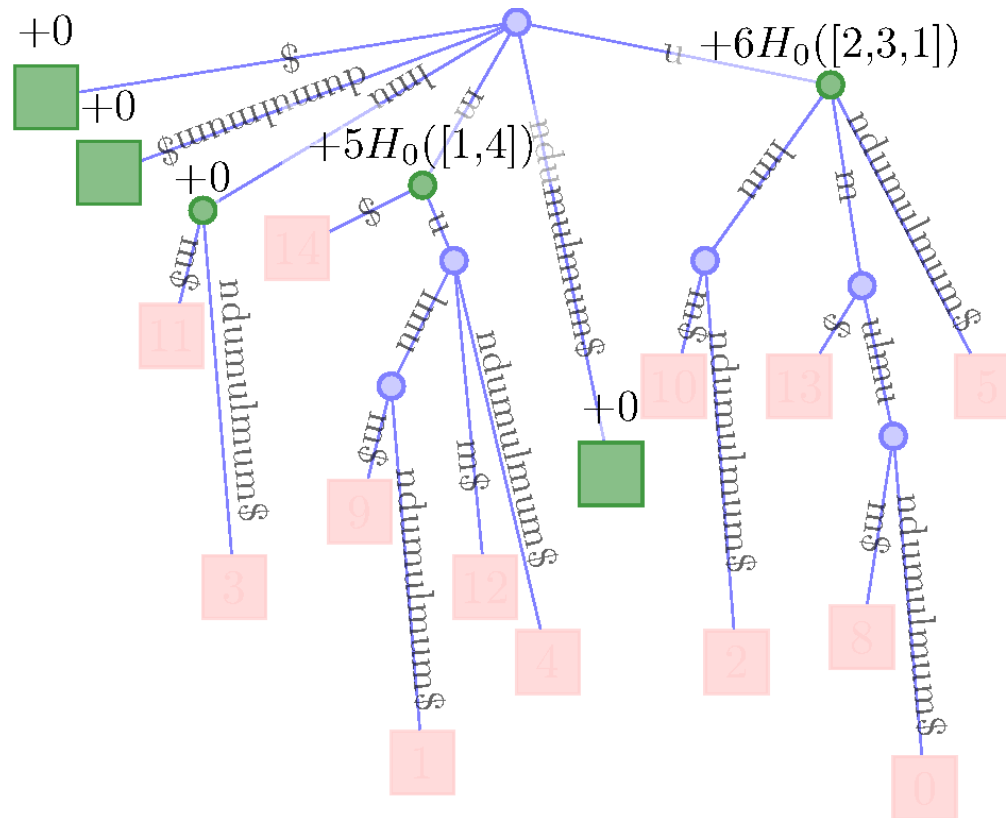
Beispiel:  $S = \text{ananas}$ ,  $k = 2$

$N(\text{an}, S) = \text{aa}$ ,  $N(\text{na}, S) = \text{ns}$ ,  $N(\text{as}, S) = \varepsilon$ .

$H_2(\text{ananas}) = \frac{2}{6} H(\text{ns}) = \frac{1}{3}$  bits

# Theorie Verlustfreier Textkompression

$H_k$ : Berechnung mittels Suffixbaumes (Beispiel:  $H_1$ )





# Theorie Verlustfreier Textkompression

Werte der empirischen Entropie in der Praxis

$H_k(S)$  in bits und (# eindeutiger Kontexte/ $|S|$  in Prozent)

$k$	dblp.xml		DNA		english		proteins	
	bits	prozent	bits	prozent	bits	prozent	bits	prozent
0	5.26	0.0	1.97	0.0	4.53	0.0	4.20	0.0
1	3.48	0.0	1.93	0.0	3.62	0.0	4.18	0.0
2	2.17	0.0	1.92	0.0	2.95	0.0	4.16	0.0
3	1.43	0.1	1.92	0.0	2.42	0.0	4.07	0.0
4	1.05	0.4	1.91	0.0	2.06	0.3	3.83	0.1
5	0.82	1.3	1.90	0.0	1.84	1.0	3.16	1.7
6	0.70	2.7	1.88	0.0	1.67	2.7	1.50	17.4

# Wörterbuchbasierte Textkompression

Grundidee: wähle  $\Sigma' \subseteq \Sigma^*$  und ersetze  $S \in \Sigma^*$  durch  $S' = \langle s'_1, \dots, s'_k \rangle \in \Sigma'^*$ , so dass  $S = s'_1 \cdot s'_2 \cdots s'_k$ . (mit ‘ $\cdot$ ’ = Zeichenkettenkonkatenation).

Platz  $n \lceil \log \Sigma \rceil \rightarrow k \lceil \log \Sigma' \rceil$  mit Entropiecodierung der Zeichen aus  $\Sigma'$  sogar  $k \text{Entropie}(S')$

**Problem:** zusätzlicher Platz für Wörterbuch.

OK für sehr große Datenbestände.

# Wörterbuchbasierte Textkompression – Beispiel

Volltextsuchmaschinen verwenden oft  $\Sigma' :=$  durch Leerzeichen (etc.)  
separierte Wörter der zugrundeliegenden natürlichen Sprache.

Spezialbehandlung von Trennzeichen etc.

Gallia est omnis divisa in partes tres, ...

→ 

gallia	est	omnis	divisa	in	partes	tres	...
--------	-----	-------	--------	----	--------	------	-----

# Lempel-Ziv Kompression (LZ)

Idee: baue Wörterbuch “on the fly” bei Codierung und Decodierung.

Ohne explizite Speicherung!

## Naive Lempel-Ziv Kompression (LZ)

**Procedure** naiveLZCompress( $\langle s_1, \dots, s_n \rangle, \Sigma$ )

$D := \Sigma$

// Init Dictionary

$p := s_1$

// current string

**for**  $i := 2$  **to**  $n$  **do**

**if**  $p \cdot s_i \in D$  **then**  $p := p \cdot s_i$

**else**

output code for  $p$

$D := D \cup p \cdot s_i$

$p := s_i$

output code for  $p$

## Naive LZ Dekompression

**Procedure** naiveLZDecode( $\langle c_1, \dots, c_k \rangle$ )

$D := \Sigma$

output decode( $c_1$ )

**for**  $i := 2$  **to**  $k$  **do**

**if**  $c_i \in D$  **then**

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_i)[1]$

**else** // where  $s[1]$  is the first letter.

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_{i-1})[1]$

  output decode( $c_i$ )

# LZ Beispiel: abracadabra

#	$p$	output	input	$DU =$
1	$\perp$	-	a	a,b,c,d,r
2	a	a	b	ab
3	b	b	r	br
4	r	r	a	ra
5	a	a	c	ac
6	c	c	a	ca
7	a	a	d	ad
8	d	d	a	da
9	a	-	b	-
10	ab	ab	r	abr
11	r	-	a	-
-	ra	ra	-	-

## LZ-Verfeinerungen

- Wörterbuchgröße begrenzen, z.B.  $|D| \leq 4096 \rightsquigarrow$  12bit codes.
- Von vorn wenn Wörterbuch voll  $\rightsquigarrow$  Blockweise arbeiten
- Kodierung mit **variabler Zahl Bits** (z.B. Huffman, arithmetic coding)
- Selten benutzte Wörterbucheinträge löschen ???
- Wörterbuch effizient implementieren:  
(universelles) hashing
- verwendet in zip/gzip mit Huffman Kodierung