Karlsruher Institut für Technologie

Algorithmen 2

Kapitel 11: Stringology Teil 2 – Suffix-Arrays und LCP-Arrays

Florian Kurpicz

Wiederholung: 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\$

Wiederholung: 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

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

	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$

Wiederholung: 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

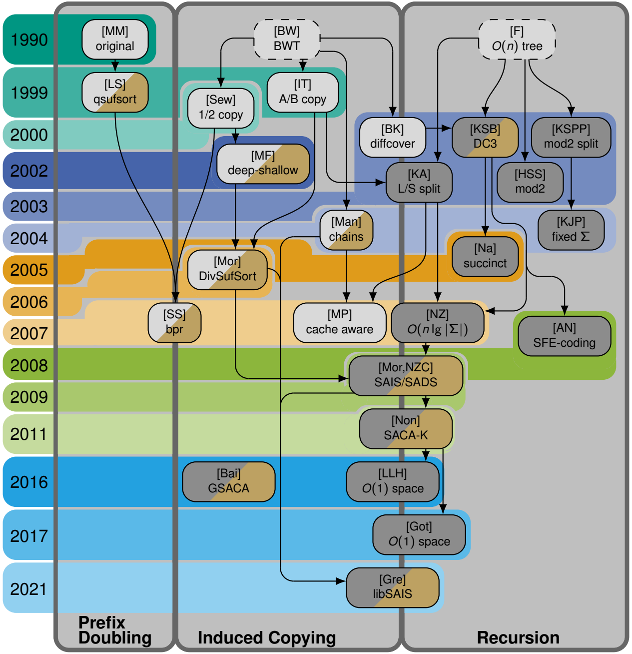
Für einen Text T der Länge n und sein Suffix-Array SA ist das **LCP-array** definiert als

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

	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\$

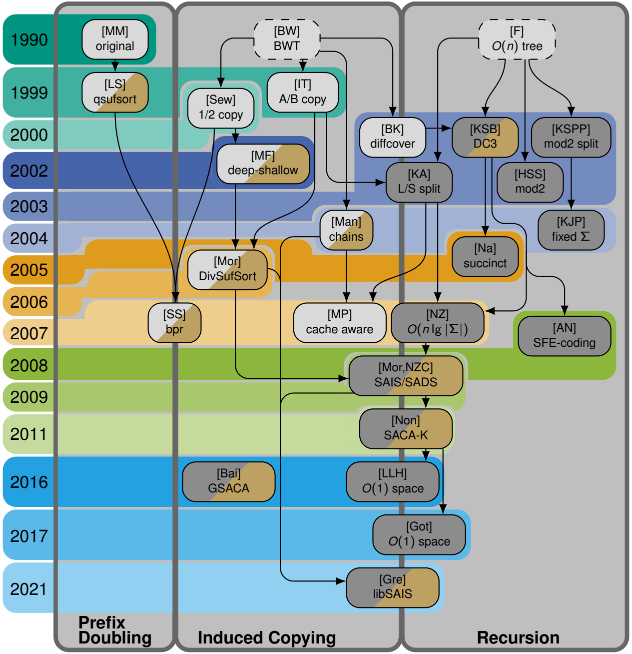
PINGO

https://pingo.scc.kit.edu/390528



Entwicklung Sequentielle Suffix-Array-Konstruktionsalgorithmen

- based on [Bah+19; Bin18; Kur20; PST07]
- **grau**: lineare Laufzeit
- **braun**: Implementierung verfügbar

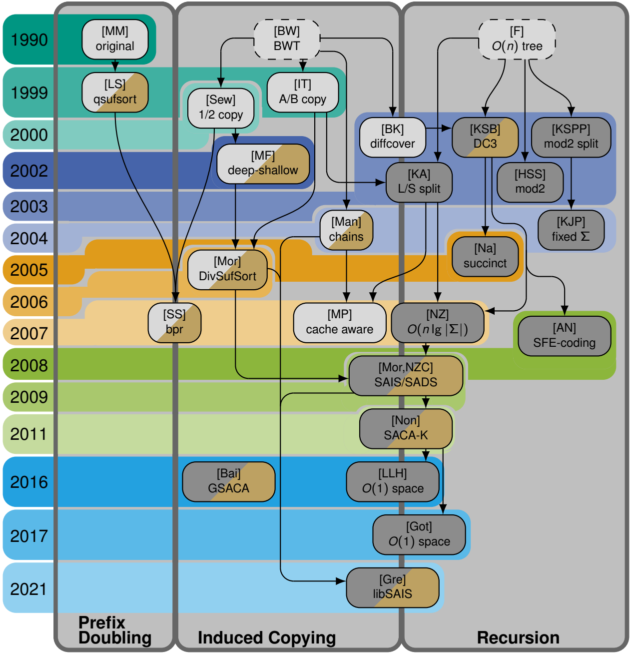


Entwicklung Sequentielle Suffix-Array-Konstruktionsalgorithmen

- based on [Bah+19; Bin18; Kur20; PST07]
- **grau**: lineare Laufzeit
- **braun**: Implementierung verfügbar

Meilensteine

- DC3 erster $O(n)$-Algorithmus
- $O(n)$ Laufzeit und $O(1)$ Platz für ganzzahlige Alphabete möglich

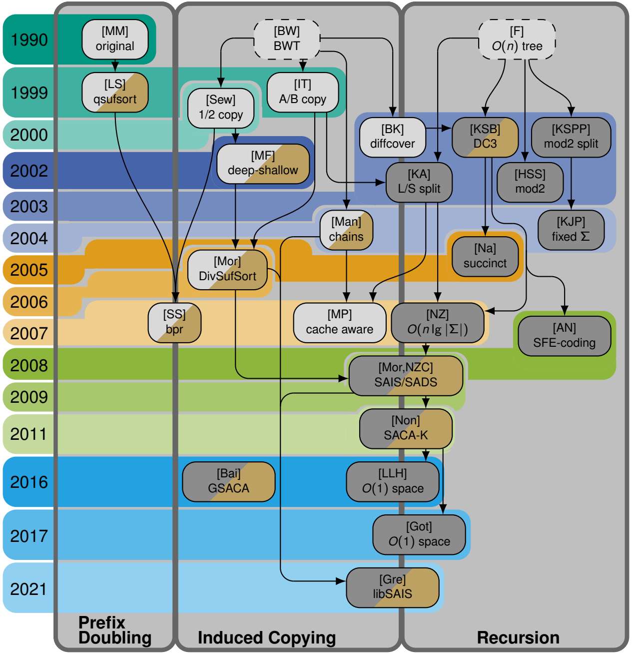


Entwicklung Sequentielle Suffix-Array-Konstruktionsalgorithmen

- based on [Bah+19; Bin18; Kur20; PST07]
- **grau**: lineare Laufzeit
- **braun**: Implementierung verfügbar

Meilensteine

- DC3 erster $O(n)$-Algorithmus
- $O(n)$ Laufzeit und $O(1)$ Platz für ganzzahlige Alphabete möglich
- bis 2021: DivSufSort schnellster SACA in Praxis mit $O(n \lg n)$ Laufzeit



Entwicklung Sequentielle Suffix-Array-Konstruktionsalgorithmen

- based on [Bah+19; Bin18; Kur20; PST07]
- grau: lineare Laufzeit
- braun: Implementierung verfügbar

Meilensteine

- DC3 erster $O(n)$-Algorithmus
- $O(n)$ Laufzeit und $O(1)$ Platz für ganzzahlige Alphabete möglich
- bis 2021: DivSufSort schnellster SACA in Praxis mit $O(n \lg n)$ Laufzeit
- seit 2021: libSAIS schnellster SACA in Praxis mit $O(n)$ Laufzeit

Inverses Suffix-Array und Ränge

Definition: Inverses Suffix-Array

- Das **inverse** Suffix-Array (ISA) ist die inverse Permutation von SA
- Für einen Text T der Länge n und sein SA ist

$$ISA[SA[i]] = i$$

	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
ISA	3	8	6	11	13	5	10	12	4	9	7	2	1

Inverses Suffix-Array und Ränge

Definition: Inverses Suffix-Array

- Das **inverse** Suffix-Array (ISA) ist die inverse Permutation von SA
- Für einen Text T der Länge n und sein SA ist

$$ISA[SA[i]] = i$$

Definition: Rang

- Der **Rang** eines Suffix ist die Anzahl der lexikographisch kleineren Suffixe plus eins
- Der i-te ISA-Eintrag entspricht dem Rang von $T[i..n]$

	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
ISA	3	8	6	11	13	5	10	12	4	9	7	2	1

h-Ordnung, *h*-Gruppen und *h*-Ränge

Karlsruher Institut für Technologie

Definition: *h*-Ordnung

- ***h*-Ordnung**:
  $T[i..n] \leq_h T[j..n] \iff T[i..i+h) \leq T[j..j+h)$
- SA_h ist das Suffix-Array der nach der *h*-Ordnung sortierten Suffixe ℹ nicht eindeutig

h-Ordnung, h-Gruppen und h-Ränge

Definition: h-Ordnung

- ***h*-Ordnung**:
  $T[i..n] \leq_h T[j..n] \iff T[i..i+h) \leq T[j..j+h)$
- SA_h ist das Suffix-Array der nach der h-Ordnung sortierten Suffixe ℹ nicht eindeutig

Definition: h-Ränge und h-Gruppe

- Alle Suffixe, die nach h-Ordnung gleich sind, sind in einer ***h*-Gruppe**
- Der ***h*-Rang** eines Suffixes ist die Anzahl der lexikographisch kleineren h-Gruppen plus eins

h-Ordnung, h-Gruppen und h-Ränge

Definition: h-Ordnung

- ***h*-Ordnung**:
  $T[i..n] \leq_h T[j..n] \iff T[i..i+h) \leq T[j..j+h)$
- SA_h ist das Suffix-Array der nach der h-Ordnung sortierten Suffixe nicht eindeutig

Definition: h-Ränge und h-Gruppe

- Alle Suffixe, die nach h-Ordnung gleich sind, sind in einer ***h*-Gruppe**
- Der ***h*-Rang** eines Suffixes ist die Anzahl der lexikographisch kleineren h-Gruppen plus eins

```
m i s s i s s i p p i $
i s s i s s i p p i $
s s i s s i p p i $
s i s s i p p i $
i s s i p p i $
s s i p p i $
s i p p i $
i p p i $
p p i $
p i $
i $
$
```

h-Ordnung, h-Gruppen und h-Ränge

Karlsruher Institut für Technologie

Definition: h-Ordnung

- ***h*-Ordnung**:
  $T[i..n] \leq_h T[j..n] \iff T[i..i+h) \leq T[j..j+h)$
- SA_h ist das Suffix-Array der nach der h-Ordnung sortierten Suffixe ℹ nicht eindeutig

Definition: h-Ränge und h-Gruppe

- Alle Suffixe, die nach h-Ordnung gleich sind, sind in einer ***h*-Gruppe**
- Der ***h*-Rang** eines Suffixes ist die Anzahl der lexikographisch kleineren h-Gruppen plus eins

```
m i s s i s s i p p i $
i s s i s s i p p i $
s s i s s i p p i $
s i s s i p p i $
i s s i p p i $
s s i p p i $
s i p p i $
i p p i $
p p i $
p i $
i $
$
```

h-Ordnung, h-Gruppen und h-Ränge

Karlsruher Institut für Technologie

Definition: h-Ordnung

- ***h*-Ordnung**:
  $T[i..n] \leq_h T[j..n] \iff T[i..i+h) \leq T[j..j+h)$
- SA_h ist das Suffix-Array der nach der h-Ordnung sortierten Suffixe ℹ nicht eindeutig

Definition: h-Ränge und h-Gruppe

- Alle Suffixe, die nach h-Ordnung gleich sind, sind in einer ***h*-Gruppe**
- Der ***h*-Rang** eines Suffixes ist die Anzahl der lexikographisch kleineren h-Gruppen plus eins

```
m i s s i s s i p p i $
i s s i s s i p p i $
s s i s s i p p i $
s i s s i p p i $
i s s i p p i $
s s i p p i $
s i p p i $
i p p i $
p p i $
p i $
i $
$
```

Präfix-Verdoppelung (Prefix-Doubling): Idee

- 1-Rang entspricht erstem Zeichen

Präfix-Verdoppelung (Prefix-Doubling): Idee

- 1-Rang entspricht erstem Zeichen
- 2-Rang kann aus Rängen der ersten zwei Zeichen berechnet werden

Präfix-Verdoppelung (Prefix-Doubling): Idee

- 1-Rang entspricht erstem Zeichen
- 2-Rang kann aus Rängen der ersten zwei Zeichen berechnet werden
- 3-Rang kann aus Rängen der ersten drei Zeichen berechnet werden

Präfix-Verdoppelung (Prefix-Doubling): Idee

Karlsruher Institut für Technologie

- 1-Rang entspricht erstem Zeichen
- 2-Rang kann aus Rängen der ersten zwei Zeichen berechnet werden
- 3-Rang kann aus Rängen der ersten drei Zeichen berechnet werden
- 4-Rang kann aus Rängen der ersten vier Zeichen berechnet werden

Präfix-Verdoppelung (Prefix-Doubling): Idee

- 1-Rang entspricht erstem Zeichen
- 2-Rang kann aus Rängen der ersten zwei Zeichen berechnet werden
- 3-Rang kann aus Rängen der ersten drei Zeichen berechnet werden
- 4-Rang kann aus Rängen der ersten vier Zeichen berechnet werden
- 4-Rang kann aus zwei 2-Rängen berechnet werden

Präfix-Verdoppelung (Prefix-Doubling): Idee

Karlsruher Institut für Technologie

- 1-Rang entspricht erstem Zeichen
- 2-Rang kann aus Rängen der ersten zwei Zeichen berechnet werden
- 3-Rang kann aus Rängen der ersten drei Zeichen berechnet werden
- 4-Rang kann aus Rängen der ersten vier Zeichen berechnet werden
- 4-Rang kann aus zwei 2-Rängen berechnet werden

- berechne 2^{k+1}-Ränge aus zwei 2^k-Rängen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \ \& \ ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

m	i	s	s	i	s	s	i	p	p	i	$
i	s	s	i	s	s	i	p	p	i	$	
s	s	i	s	s	i	p	p	i	$		
s	i	s	s	i	p	p	i	$			
i	s	s	i	p	p	i	$				
s	s	i	p	p	i	$					
s	i	p	p	i	$						
i	p	p	i	$							
p	p	i	$								
p	i	$									
i	$										
$											

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i+2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

m	i	s	s	i	s	s	i	p	p	i	$
i	s	s	i	s	s	i	p	p	i	$	
s	s	i	s	s	i	p	p	i	$		
s	i	s	s	i	p	p	i	$			
i	s	s	i	p	p	i	$				
s	s	i	p	p	i	$					
s	i	p	p	i	$						
i	p	p	i	$							
p	p	i	$								
p	i	$									
i	$										
$											

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

2	`m i s s i s s i p p i $`
1	`i s s i s s i p p i $`
4	`s s i s s i p p i $`
4	`s i s s i p p i $`
1	`i s s i p p i $`
4	`s s i p p i $`
4	`s i p p i $`
1	`i p p i $`
3	`p p i $`
3	`p i $`
1	`i $`
0	`$`

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \ \& \ ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

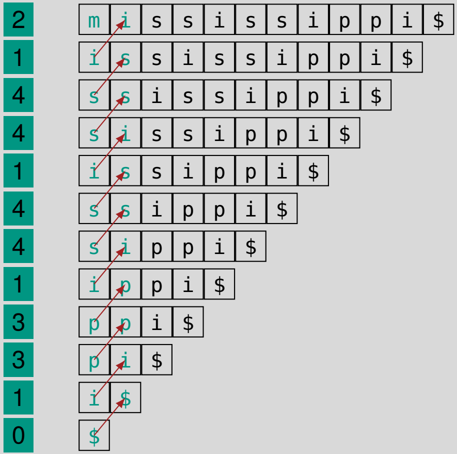
2	`m i s s i s s i p p i $`
1	`i s s i s s i p p i $`
4	`s s i s s i p p i $`
4	`s i s s i p p i $`
1	`i s s i p p i $`
4	`s s i p p i $`
4	`s i p p i $`
1	`i p p i $`
3	`p p i $`
3	`p i $`
1	`i $`
0	`$`

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \ \& \ ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen



Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \ \& \ ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. SA aus ISA berechnen

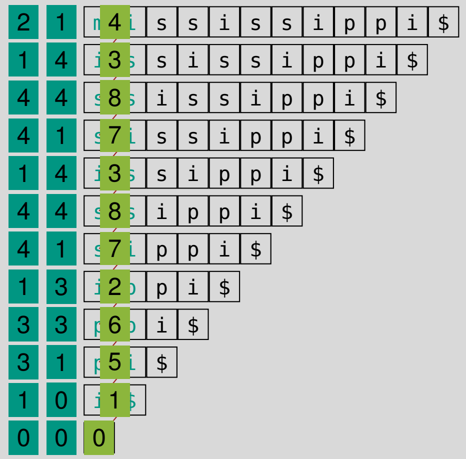
2	1	m i s s i s s i p p i $
1	4	i s s i s s i p p i $
4	4	s s i s s i p p i $
4	1	s i s s i p p i $
1	4	i s s i p p i $
4	4	s s i p p i $
4	1	s i p p i $
1	3	i p p i $
3	3	p p i $
3	1	p i $
1	0	i $
0	0	$

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. SA aus ISA berechnen



Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \; \& \; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \; \& \; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \ \& \ ISA_{2^k}[i + 2^k]$$

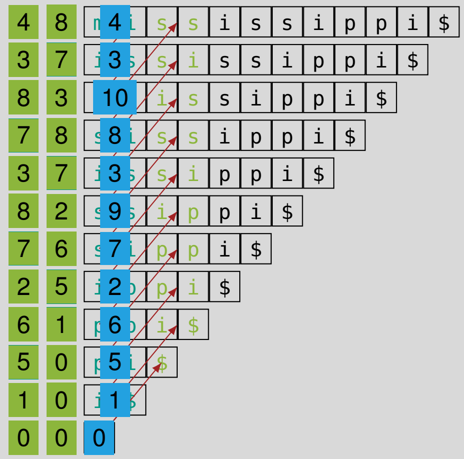
4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. SA aus ISA berechnen



Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. SA aus ISA berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. SA aus ISA berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \;\&\; ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Präfix-Verdoppelung (Prefix-Doubling): Beispiel

1. initialer Rang ist $T[i]$
2. für $k = 0$ bis $\lceil \lg n \rceil$
3. neue Ränge basierend auf

$$ISA_{2^k}[i] \,\&\, ISA_{2^k}[i + 2^k]$$

4. wenn alle Ränge einzigartig, fertig
5. *SA* aus *ISA* berechnen

Einfacher Algorithmus

- N. Jesper Larsson und Kunihiko Sadakane. „Faster Suffix Sorting“. In: *Theor. Comput. Sci.* 387.3 (2007), Seiten 258–272. DOI: `10.1016/j.tcs.2007.07.017`

Präfix-Verdoppelung (Prefix-Doubling): In der Praxis

ISA_h Benutzen

- ISA_{2^k} benutzen, um Rang-Tupel zu konstruieren
- an Position i den Rang $ISA_{2^k}[i + 2^k]$ benutzen
- wenn $i + 2^k > n$, dann ist zweiter Rang 0

Präfix-Verdoppelung (Prefix-Doubling): In der Praxis

ISA_h Benutzen

- ISA_{2^k} benutzen, um Rang-Tupel zu konstruieren
- an Position i den Rang $ISA_{2^k}[i + 2^k]$ benutzen
- wenn $i + 2^k > n$, dann ist zweiter Rang 0

Nach Text-Position Sortieren

- im externen und verteilten Speicher *ISA*-Zugriff schwierig
- sortiere Tupel (Textposition i, Rang r)
- mit Ordnung $(i, r) \leq (j, r')$ genau dann, wenn

$$(i \bmod 2^k, \lfloor i/2^k \rfloor) < (j \bmod 2^k, \lfloor j/2^k \rfloor)$$

Präfix-Verdoppelung (Prefix-Doubling): Laufzeit

Karlsruher Institut für Technologie

- Laufzeit: $O(n \lg n)$
- Platzbedarf: $8n(+n)$ Wörter ℹ für Texte $\leq 4\,\text{GiB}$
- Worst-Case Eingabe: $T = a^{n-1}\$$

Präfix-Verdoppelung (Prefix-Doubling): Laufzeit

- Laufzeit: $O(n \lg n)$
- Platzbedarf: $8n(+n)$ Wörter ℹ für Texte $\leq 4\,\text{GiB}$
- Worst-Case Eingabe: $T = a^{n-1}\$$

Generalisierung: Präfix-Verαung

- nicht nur Verdopplung möglich
- berechne α^{k+1}-Ränge aus α vielen α^k-Rängen
- kann I/Os im externen Speicher sparen 🌶

  ℹ $\alpha = 4$ hat 30 % weniger I/Os als $\alpha = 2$ [Dem+08]

Suffix-Array-Konstruktion in Linearzeit

- erster direkter Linearzeitalgorithmus: DC3
- Suffix-Baum in Linearzeit (mit ähnlicher Idee) [Far97]
- Juha Kärkkäinen, Peter Sanders und Stefan Burkhardt. „Linear work suffix array construction". In: *J. ACM* 53.6 (2006), Seiten 918–936. DOI: `10.1145/1217856.1217858`
- basiert auf Difference Cover
  ℹ Differenzüberdeckung

Differenzüberdeckung (Difference Cover)

Definition: Differenzüberdeckung

Die Menge $D \subseteq [0, \nu)$ ist eine **Differenzüberdeckung** modulo ν, wenn

$$\{(i - j) \mod \nu \colon i, j \in D\} = [0, \nu)$$

- $\{0, 1\}$ ist Differenzüberdeckung modulo 3
- $\{0, 1, 3\}$ ist Differenzüberdeckung modulo 7
- $\{0, 1, 3, 9\}$ ist Differenzüberdeckung modulo 13

Differenzüberdeckung (Difference Cover)

Definition: Differenzüberdeckung

Die Menge $D \subseteq [0, \nu)$ ist eine **Differenzüberdeckung** modulo ν, wenn

$$\{(i - j) \mod \nu \colon i, j \in D\} = [0, \nu)$$

- $\{0, 1\}$ ist Differenzüberdeckung modulo 3
- $\{0, 1, 3\}$ ist Differenzüberdeckung modulo 7
- $\{0, 1, 3, 9\}$ ist Differenzüberdeckung modulo 13

- $0 \equiv 0 - 0 \pmod 3$
- $1 \equiv 1 - 0 \pmod 3$
- $2 \equiv 0 - 1 \pmod 3$

Differenzüberdeckung (Difference Cover)

Definition: Differenzüberdeckung

Die Menge $D \subseteq [0, \nu)$ ist eine **Differenzüberdeckung** modulo ν, wenn

$$\{(i - j) \mod \nu \colon i, j \in D\} = [0, \nu)$$

- $\{0, 1\}$ ist Differenzüberdeckung modulo 3
- $\{0, 1, 3\}$ ist Differenzüberdeckung modulo 7
- $\{0, 1, 3, 9\}$ ist Differenzüberdeckung modulo 13

- $0 \equiv 0 - 0 \pmod 3$
- $1 \equiv 1 - 0 \pmod 3$
- $2 \equiv 0 - 1 \pmod 3$

- $0 \equiv 0 - 0 \pmod 7$
- $1 \equiv 1 - 0 \pmod 7$
- $2 \equiv 3 - 1 \pmod 7$
- $3 \equiv 3 - 0 \pmod 7$
- $4 \equiv 0 - 3 \pmod 7$
- $5 \equiv 1 - 3 \pmod 7$
- $6 \equiv 0 - 1 \pmod 7$

Suffix-Array-Konstruktion mit DC3 (1/6)

1. Sample Suffixe

- für $i \in \{0, 1, 2\}$ sei

$$B_i = \{i \in [0, n) \colon i \mod 3 = k\}$$

- $C = B_0 \cdot B_1$
  - $\{0, 1\}$ ist Differenzüberdeckung mod. 3

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

Suffix-Array-Konstruktion mit DC3 (1/6)

1. Sample Suffixe

- für $i \in \{0, 1, 2\}$ sei

$$B_i = \{i \in [0, n) \colon i \mod 3 = k\}$$

- $C = B_0 \cdot B_1$

  ℹ $\{0, 1\}$ ist Differenzüberdeckung mod. 3

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

Suffix-Array-Konstruktion mit DC3 (1/6)

1. Sample Suffixe

- für $i \in \{0,1,2\}$ sei

$$B_i = \{i \in [0,n) : i \mod 3 = k\}$$

- $C = B_0 \cdot B_1$
  - $\{0,1\}$ ist Differenzüberdeckung mod. 3

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

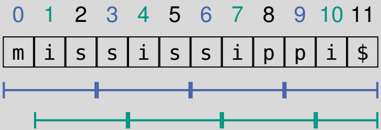
Suffix-Array-Konstruktion mit DC3 (1/6)

1. Sample Suffixe

- für $i \in \{0, 1, 2\}$ sei

$$B_i = \{i \in [0, n) : i \mod 3 = k\}$$

- $C = B_0 \cdot B_1$
  - $\{0, 1\}$ ist Differenzüberdeckung mod. 3



- $C = \{0, 3, 6, 9, 1, 4, 7, 10\}$

Suffix-Array-Konstruktion mit DC3 (2/6)

2. Sortiere Gesampelte Suffixe

- für $k = 0, 1$ sei

$$R_k = [T[k]T[k+1]T[k+2]][T[k+3]T[k+4]T[k+5]] \ldots [T[\max B_k]T[\max B_k + 1]T[\max B_k + 2]]$$

- $R = R_0 \cdot R_1$
- sortiere R mithilfe von Radix Sort in $O(n)$ Zeit
- alle Zeichen unterschiedlich: Ränge gesampelter Suffixe bekannt
- sonst: Algorithmus rekursiv auf R aufrufen

Suffix-Array-Konstruktion mit DC3 (2/6)

2. Sortiere Gesampelte Suffixe

- für $k = 0, 1$ sei

$$R_k = [T[k]T[k+1]T[k+2]][T[k+3]T[k+4]T[k+5]]\ldots[T[\max B_k]T[\max B_k+1]T[\max B_k+2]]$$

- $R = R_0 \cdot R_1$
- sortiere R mithilfe von Radix Sort in $O(n)$ Zeit
- alle Zeichen unterschiedlich: Ränge gesampelter Suffixe bekannt
- sonst: Algorithmus rekursiv auf R aufrufen

0	1	2	3	4	5	6	7
$[mis]$	$[sis]$	$[sip]$	$[pi\$]$	$[iss]$	$[iss]$	$[ipp]$	$[i\$\$]$
3	6	5	4	2	2	1	0

Suffix-Array-Konstruktion mit DC3 (2/6)

2. Sortiere Gesampelte Suffixe

- für $k = 0, 1$ sei

$$R_k = [T[k]T[k+1]T[k+2]][T[k+3]T[k+4]T[k+5]] \ldots [T[\max B_k]T[\max B_k + 1]T[\max B_k + 2]]$$

- $R = R_0 \cdot R_1$
- sortiere R mithilfe von Radix Sort in $O(n)$ Zeit
- alle Zeichen unterschiedlich: Ränge gesampelter Suffixe bekannt
- sonst: Algorithmus rekursiv auf R aufrufen

0	1	2	3	4	5	6	7
$[mis]$	$[sis]$	$[sip]$	$[pi\$]$	$[iss]$	$[iss]$	$[ipp]$	$[i\$\$]$
3	6	5	4	2	2	1	0

Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

0	1	2	3	4	5	6	7
3	6	5	4	2	2	1	0

Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

0	1	2	3	4	5	6	7
3	6	5	4	2	2	1	0

Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

0	1	2	3	4	5	6	7
3	6	5	4	2	2	1	0

Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

0	1	2	3	4	5	6	7
3	6	5	4	2	2	1	0

- $C = \{0, 3, 6, 1, 4, 7\}$

Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

0	1	2	3	4	5	6	7
3	6	5	4	2	2	1	0

- $C = \{0, 3, 6, 1, 4, 7\}$

Rekursion: Schritt 2

0	1	2	3	4	5
[365]	[422]	[100]	[654]	[221]	[000]
3	4	1	5	2	0

Suffix-Array-Konstruktion mit DC3 (4/6)

3. Sortiere nicht-Gesamplte Suffixe

- seien $i, j \in B_2$, dann gilt

$$S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$$

- Ränge der folgenden Suffixe bekannt
- Sortiere Tupel (in B_2) mit Radix Sort
- $O(n)$ Zeit

Suffix-Array-Konstruktion mit DC3 (4/6)

3. Sortiere nicht-Gesamplte Suffixe

- seien $i, j \in B_2$, dann gilt

$$S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$$

- Ränge der folgenden Suffixe bekannt
- Sortiere Tupel (in B_2) mit Radix Sort
- $O(n)$ Zeit

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

Suffix-Array-Konstruktion mit DC3 (4/6)

3. Sortiere nicht-Gesamplte Suffixe

- seien $i, j \in B_2$, dann gilt

$$S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$$

- Ränge der folgenden Suffixe bekannt
- Sortiere Tupel (in B_2) mit Radix Sort
- $O(n)$ Zeit

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

Suffix-Array-Konstruktion mit DC3 (4/6)

3. Sortiere nicht-Gesamplte Suffixe

- seien $i, j \in B_2$, dann gilt

$$S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$$

- Ränge der folgenden Suffixe bekannt
- Sortiere Tupel (in B_2) mit Radix Sort
- $O(n)$ Zeit

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

- $(0,0,0) \leq (2,0,0)$

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

- $(0,0,0) \leq (2,0,0)$
- $(1,0) \leq (2,1)$

Suffix-Array-Konstruktion mit DC3 (5/6)

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

- $(0,0,0) \leq (2,0,0)$
- $(1,0) \leq (2,1)$
- $(2,1,0) \leq (2,2,1)$
- ...

Suffix-Array-Konstruktion mit DC3 (5/6)

Karlsruher Institut für Technologie

4. Merge Suffixe

- sei $i \in C$ und $j \in B_2$, dann gilt
  - wenn $i \in B_0$, dann
    $S_i \leq S_j \iff$
    $(T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$
  - wenn $i \in B_1$, dann
    $S_i \leq S_j \iff$
    $(T[i], T[i+1], Rang(S_{i+2})) \leq$
    $(T[j], T[j+1], Rang(S_{j+2}))$

	0	1	2	3	4	5	6	7
	3	6	5	4	2	2	1	0
Ränge	3	5	$\perp$	4	2	$\perp$	1	0

- $\underbrace{(2,1)}_{S_2} \leq \underbrace{(5,4)}_{S_5}$

- $(0,0,0) \leq (2,0,0)$
- $(1,0) \leq (2,1)$
- $(2,1,0) \leq (2,2,1)$
- ...
- Ränge: 4 7 6 5 3 2 1 0

Suffix-Array-Konstruktion mit DC3 (6/6)

Rekursion Auflösen

0	1	2	3	4	5	6	7
[*mis*]	[*sis*]	[*sip*]	[*pi*$]	[*iss*]	[*iss*]	[*ipp*]	[*i*$$]
4	7	6	5	3	2	1	0

Suffix-Array-Konstruktion mit DC3 (6/6)

Rekursion Auflösen

0	1	2	3	4	5	6	7
[*mis*]	[*sis*]	[*sip*]	[*pi*$]	[*iss*]	[*iss*]	[*ipp*]	[*i*$$]
4	7	6	5	3	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11
	m	i	s	s	i	s	s	i	p	p	i	$
Ränge	4	3	$\perp$	7	2	$\perp$	6	1	$\perp$	5	0	$\perp$

Suffix-Array-Konstruktion mit DC3 (6/6)

Rekursion Auflösen

0	1	2	3	4	5	6	7
[*mis*]	[*sis*]	[*sip*]	[*pi*$]	[*iss*]	[*iss*]	[*ipp*]	[*i*$$]
4	7	6	5	3	2	1	0

	0	1	2	3	4	5	6	7	8	9	10	11
	m	i	s	s	i	s	s	i	p	p	i	$
Ränge	4	3	$\perp$	7	2	$\perp$	6	1	$\perp$	5	0	$\perp$

- Rest bleibt also Übung ℹ Lösung: 11 10 7 4 1 0 9 8 6 3 5 2

DC3: Laufzeiten

- alles außer der Rekursion in $O(n)$ Zeit
- wir sortieren nur Tupel der Größe ≤ 3
- Radix Sort in $O(n)$ Zeit

DC3: Laufzeiten

- alles außer der Rekursion in $O(n)$ Zeit
- wir sortieren nur Tupel der Größe ≤ 3
- Radix Sort in $O(n)$ Zeit

- Rekursion auf Text der Größe $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$

DC3: Laufzeiten

- alles außer der Rekursion in $O(n)$ Zeit
- wir sortieren nur Tupel der Größe ≤ 3
- Radix Sort in $O(n)$ Zeit

- Rekursion auf Text der Größe $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$

Verallgemeinerung DCX

- funktioniert für alle Differenzüberdeckung
- Sortieren etwas komplizierter
- Laufzeit: $O(\nu n)$

LCP-Array-Konstruktion in Linearzeit [Kas+01]

Function LinearTimeLCP(T, $SA[1..n]$):

$\ell = 0, LCP[1] = 0$
for $i = 1, \ldots, n$ **do**
    if $ISA[i] \neq 1$ **then**
        $j = SA[ISA[i] - 1]$
        while $T[i + \ell] = T[j + \ell]$ **do**
            $\ell = \ell + 1$
        $LCP[ISA[i]] = \ell$
        $\ell = \max\{0, \ell - 1\}$
return LCP

- naive in $O(n^2)$ Zeit
- berechne LCP-Einträge in Textreihenfolge
- Nutze *ISA* um passendes lex. kleineres Suffix zu finden

LCP-Array-Konstruktion in Linearzeit [Kas+01]

Function LinearTimeLCP(T, $SA[1..n]$):

$\ell = 0, LCP[1] = 0$
for $i = 1, \ldots, n$ **do**
  if $ISA[i] \neq 1$ **then**
    $j = SA[ISA[i] - 1]$
    while $T[i + \ell] = T[j + \ell]$ **do**
      $\ell = \ell + 1$
    $LCP[ISA[i]] = \ell$
    $\ell = \max\{0, \ell - 1\}$
return LCP

- naive in $O(n^2)$ Zeit
- berechne LCP-Einträge in Textreihenfolge
- Nutze ISA um passendes lex. kleineres Suffix zu finden

	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
ISA	3	8	6	11	13	5	10	12	4	9	7	2	1
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

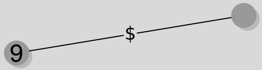
	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

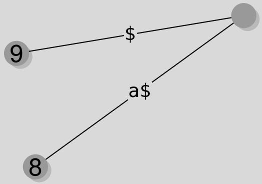


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	\$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

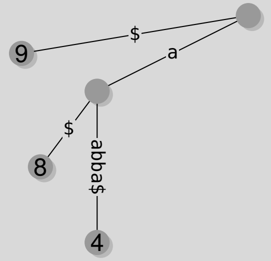


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

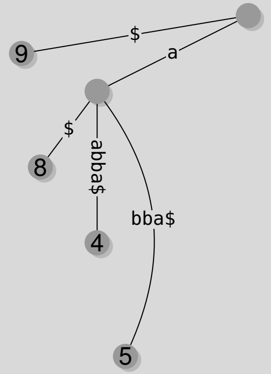


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	\$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

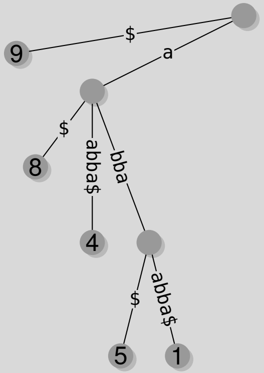


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3



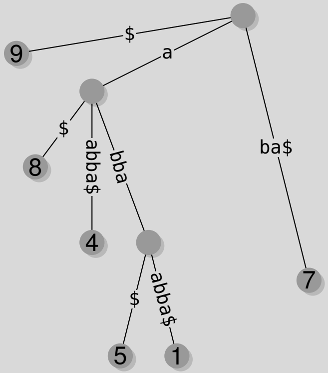
Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

Karlsruher Institut für Technologie

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	\$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

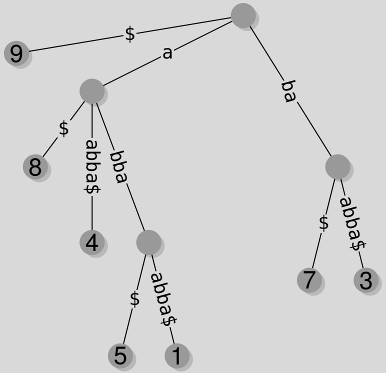


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	\$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

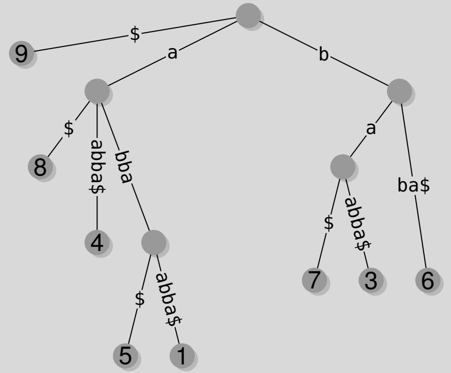


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq$ $LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	\$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

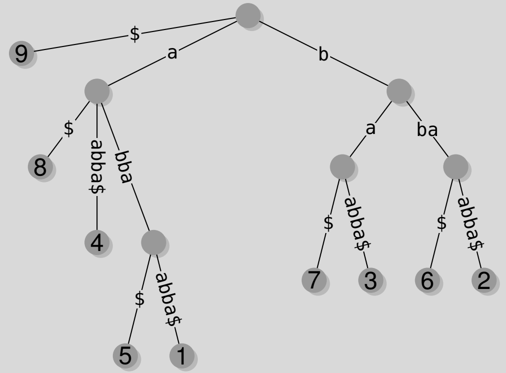


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- naiv in $O(n^2)$ Zeit

- nutze *SA* und *LCP*
- betrachte nur rechtesten Pfad im Baum
- finde tiefsten Knoten mit String-Tiefe $\leq LCP[i]$
- insgesamt $O(n)$ Zeit

	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3



Literatur I

[Bah+19] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper und Christopher Poeplau. „SACABench: Benchmarking Suffix Array Construction“. In: *SPIRE*. Band 11811. Lecture Notes in Computer Science. Springer, 2019, Seiten 407–416. DOI: 10.1007/978-3-030-32686-9_29.

[Bin18] Timo Bingmann. „Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools“. Dissertation. Karlsruhe Institute of Technology, Germany, 2018. DOI: 10.5445/IR/1000085031.

[Dem+08] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert und Peter Sanders. „Better External Memory Suffix Array Construction“. In: *ACM J. Exp. Algorithmics* 12 (2008), 3.4:1–3.4:24. DOI: 10.1145/1227161.1402296.

[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.

Literatur II

[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.

[Kas+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa und Kunsoo Park. „Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications“. In: *CPM*. Band 2089. Lecture Notes in Computer Science. Springer, 2001, Seiten 181–192. DOI: 10.1007/3-540-48194-X_17.

[KSB06] Juha Kärkkäinen, Peter Sanders und Stefan Burkhardt. „Linear work suffix array construction“. In: *J. ACM* 53.6 (2006), Seiten 918–936. DOI: 10.1145/1217856.1217858.

[Kur20] Florian Kurpicz. „Parallel Text Index Construction“. Dissertation. Technical University of Dortmund, Germany, 2020. DOI: 10.17877/DE290R-21114.

[LS07] N. Jesper Larsson und Kunihiko Sadakane. „Faster Suffix Sorting“. In: *Theor. Comput. Sci.* 387.3 (2007), Seiten 258–272. DOI: 10.1016/j.tcs.2007.07.017.

Literatur III

[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.

[PST07] Simon J. Puglisi, William F. Smyth und Andrew Turpin. „A Taxonomy of Suffix Array Construction Algorithms“. In: *ACM Comput. Surv.* 39.2 (2007), Seite 4. DOI: 10.1145/1242471.1242472.