Karlsruher Institut für Technologie

Algorithmen 2

Kapitel 11: Stringology Teil 3 – Text-Kompression und Wavelet-Trees

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$
abacabba$
abba$
abcabba$
abcabcabba$
ba$
bacabcabba$
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$

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

PINGO

https://pingo.scc.kit.edu/096973

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

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

- $\{0, 1\}$ 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

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

- $\{0, 1\}$ 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

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

- $\{0, 1\}$ 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

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

- $C = B_0 \cdot B_1$
  - $\{0, 1\}$ 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

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

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

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

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

2. Sortiere Gesampelte Suffixe

- für $k = 0, 1$ sei

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

- $R = R_0 \cdot R_1$
- 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

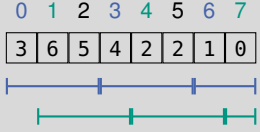
Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1

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

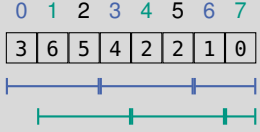
Suffix-Array-Konstruktion mit DC3 (3/6)

Rekursion: Schritt 1



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

Rekursion: Schritt 1



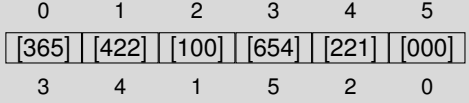
- $C = \{0, 3, 6, 1, 4, 7\}$

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

Rekursion: Schritt 1

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

Rekursion: Schritt 2



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)

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	⊥	4	2	⊥	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	⊥	7	2	⊥	6	1	⊥	5	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	⊥	7	2	⊥	6	1	⊥	5	0	⊥

- Rest zur eigenständigen 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]):
1   ℓ = 0, LCP[1] = 0
2   for i = 1, ..., n do
3       if ISA[i] ≠ 1 then
4           j = SA[ISA[i] − 1]
5           while T[i + ℓ] = T[j + ℓ] do
6               ℓ = ℓ + 1
7           LCP[ISA[i] = ℓ]
8           ℓ = max{0, ℓ − 1}
9   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, $SA[1..n]$)**:**

```
1  ℓ = 0, LCP[1] = 0
2  for i = 1, ..., n do
3      if ISA[i] ≠ 1 then
4          j = SA[ISA[i] − 1]
5          while T[i + ℓ] = T[j + ℓ] do
6              ℓ = ℓ + 1
7          LCP[ISA[i] = ℓ]
8          ℓ = max{0, ℓ − 1}
9  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

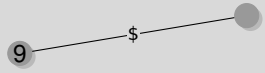
	1	2	3	4	5	6	7	8	9
T	a	b	b	a	a	b	b	a	$
SA	9	8	4	5	1	7	3	6	2
LCP	0	0	1	1	4	0	2	1	3

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

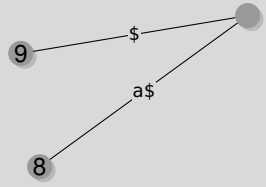


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

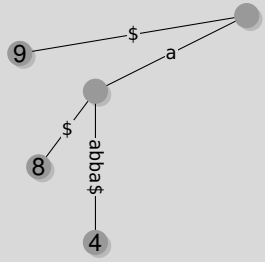


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

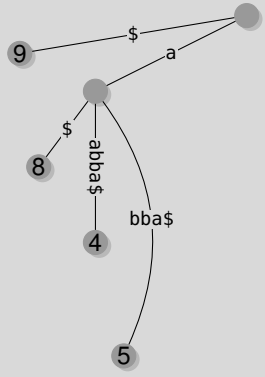


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

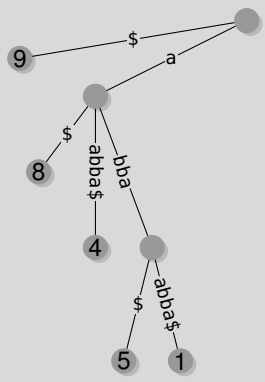


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

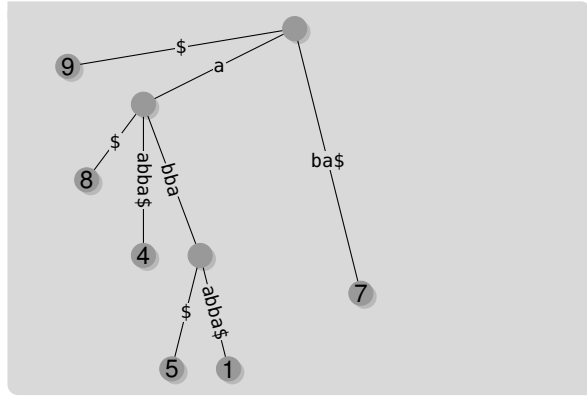


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

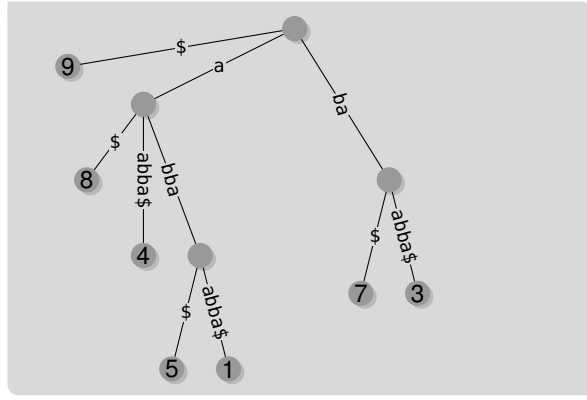


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

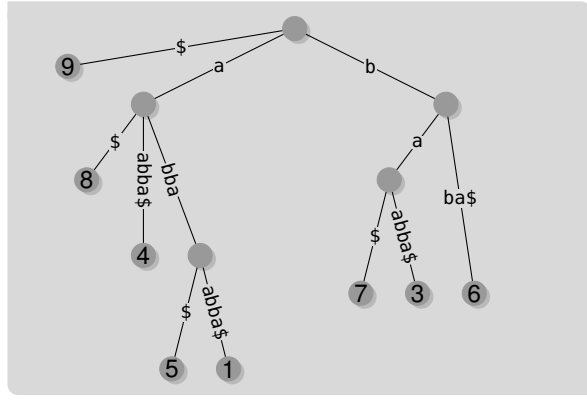


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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

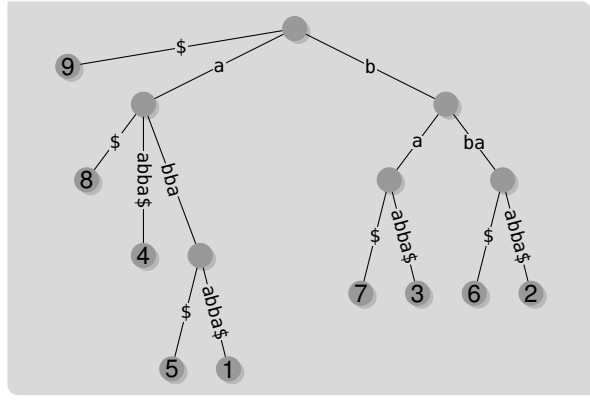


Suffix-Baum Konstruktion mit Suffix-Array und LCP-Array

- 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



Text-Kompression

- verlustbehaftete Kompression 👎
  - Bilder, Musik, . . .
- verlustfreie Kompression 👍
  - Texte, generell Archivierung, . . .

- betrachte Text zeichenweise
- betrachte größeren Kontext
- betrachte größte mögliche Überdeckungen

Huffman-Codes

- jedem Zeichen wird ein Code zugeordnet
- häufige Zeichen erhalten kurze Codes

Lempel-Ziv-Kompression

- Verweise für erneute Vorkommen
- Vorkommen können überlappen

Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $\mathit{Hist}[\alpha]$
- erstelle neuen Knoten für die zwei Knoten ohne Eltern mit dem geringsten Gewicht
- neuer Knoten hat als Gewicht das Gesamtgewicht der Kinder
- wiederhole so lange, bis nur ein Knoten ohne Eltern übrig bleibt

- Kantenbeschriftung:
  - linke Kante: 0
  - rechte Kante: 1
- Pfad zu Kind entspricht Code für Zeichen

$T =$ cbcacaa

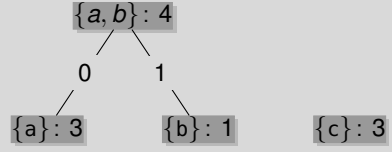
{a}: 3 {b}: 1 {c}: 3

Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $\mathit{Hist}[\alpha]$
- erstelle neuen Knoten für die zwei Knoten ohne Eltern mit dem geringsten Gewicht
- neuer Knoten hat als Gewicht das Gesamtgewicht der Kinder
- wiederhole so lange, bis nur ein Knoten ohne Eltern übrig bleibt

- Kantenbeschriftung:
  - linke Kante: 0
  - rechte Kante: 1
- Pfad zu Kind entspricht Code für Zeichen

$T =$ cbcacaa

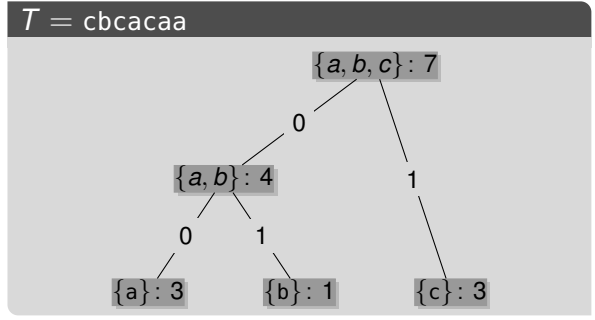


Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $Hist[\alpha]$
- erstelle neuen Knoten für die zwei Knoten ohne Eltern mit dem geringsten Gewicht
- neuer Knoten hat als Gewicht das Gesamtgewicht der Kinder
- wiederhole so lange, bis nur ein Knoten ohne Eltern übrig bleibt

- Kantenbeschriftung:
  - linke Kante: 0
  - rechte Kante: 1
- Pfad zu Kind entspricht Code für Zeichen

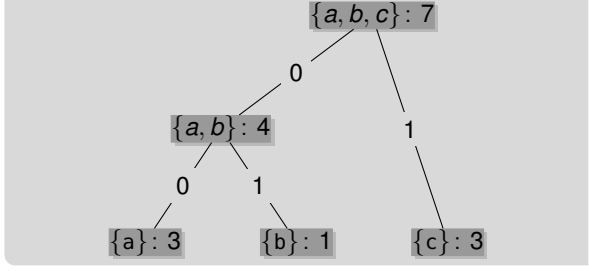
$T =$ cbcacaa



Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $Hist[\alpha]$
- erstelle neuen Knoten für die zwei Knoten ohne Eltern mit dem geringsten Gewicht
- neuer Knoten hat als Gewicht das Gesamtgewicht der Kinder
- wiederhole so lange, bis nur ein Knoten ohne Eltern übrig bleibt

$T =$ cbcacaa



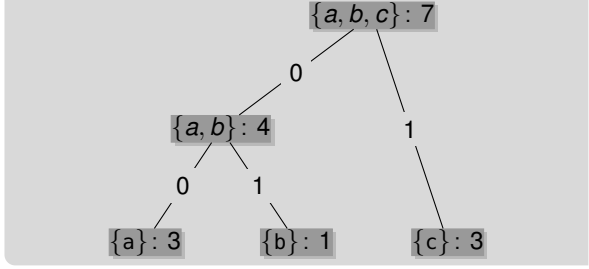
- Kantenbeschriftung:
  - linke Kante: 0
  - rechte Kante: 1
- Pfad zu Kind entspricht Code für Zeichen

- Codes haben variable Länge und sind präfix-frei
- Baum/Wörterbuch für Dekodierung benötigt

Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $Hist[\alpha]$
- erstelle neuen Knoten für die zwei Knoten ohne Eltern mit dem geringsten Gewicht
- neuer Knoten hat als Gewicht das Gesamtgewicht der Kinder
- wiederhole so lange, bis nur ein Knoten ohne Eltern übrig bleibt

$T =$ cbcacaa



- Kantenbeschriftung:
  - linke Kante: 0
  - rechte Kante: 1
- Pfad zu Kind entspricht Code für Zeichen

- Codes haben variable Länge und sind präfix-frei
- Baum/Wörterbuch für Dekodierung benötigt
- **PINGO** Sind Texte mit Wiederholungen gut Huffman-Kodierbar?

Probleme der Huffman-Kodierung

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

- funktioniert nicht gut für Wiederholungen
- besser beschreiben als 605 $\times$ a

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ `abababbbbaba$`

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba$

- $f_1 =$ a

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba\$

- $f_1 =$ a
- $f_2 =$ b

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbaba\$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ aba

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{abab}$
- $f_4 = \texttt{bbb}$
- $f_5 = \texttt{aba}$
- $f_6 = \texttt{\$}$

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T = \texttt{ababababbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{abab}$
- $f_4 = \texttt{bbb}$
- $f_5 = \texttt{aba}$
- $f_6 = \texttt{\$}$

$T = \underbrace{\texttt{aaa}\ldots\texttt{aa}}_{n-1 \text{ mal}}\texttt{\$}$

- $f_1 = \texttt{a}$
- $f_2 = \underbrace{\texttt{aaa}\ldots\texttt{aa}}_{n-2 \text{ mal}}$
- $f_3 = \texttt{\$}$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T =$ abababbbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ aba
- $f_6 =$ $

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a} = (0, \texttt{a})$
- $f_2 = \texttt{b} = (0, \texttt{b})$
- $f_3 = \texttt{abab} = (4, 1)$
- $f_4 = \texttt{bbb} = (3, 6)$
- $f_5 = \texttt{aba} = (3, 1) = (3, 3)$
- $f_6 = \texttt{\$} = (0, \texttt{\$})$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a} = (0, \texttt{a})$
- $f_2 = \texttt{b} = (0, \texttt{b})$
- $f_3 = \texttt{abab} = (4, 1)$
- $f_4 = \texttt{bbb} = (3, 6)$
- $f_5 = \texttt{aba} = (3, 1) = (3, 3)$
- $f_6 = \texttt{\$} = (0, \texttt{\$})$

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

	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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

Mit Bezug auf das *SA*

- in der Nähe im *SA*
- lexikographisch ähnlich
- längstes möglichstes Präfix
- vor dem Suffix in Textreihenfolge

	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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

Mit Bezug auf das *SA*

- in der Nähe im *SA*
- lexikographisch ähnlich
- längstes möglichstes Präfix
- vor dem Suffix in Textreihenfolge

	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
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

- **PINGO** Welche Laufzeit hat die NSV/PSV-Vorberechnung?

Previous- und Next-Smaller-Values (2/2)

- beide Arrays können in Linearzeit konstruiert werden
- betrachte PSV-Array
  - NSV funktioniert analog
- füge $-\infty$ als Index 0 an

Function ComputePSV(*SA mit* $-\infty$)**:**

```
1  for i = 1, ..., n do
2      j = i − 1
3      while j ≥ 1 und SA[i] < SA[j] do
4          j = PSV[j]
5      PSV[i] = j
6  return PSV
```

Previous- und Next-Smaller-Values (2/2)

- beide Arrays können in Linearzeit konstruiert werden
- betrachte PSV-Array
  ℹ NSV funktioniert analog
- füge $-\infty$ als Index 0 an

Function ComputePSV(*SA mit* $-\infty$)**:**

```
1  for i = 1, ..., n do
2      j = i − 1
3      while j ≥ 1 und SA[i] < SA[j] do
4          j = PSV[j]
5      PSV[i] = j
6  return PSV
```

- folge schon berechneten Werten
- dazwischen kann nicht *PSV* sein
- vergleiche jedes Element maximal zweimal
- berechne *PSV* und *NSV* in $O(n)$ Zeit
- Beispiel an der Tafel

RMQs und das LCP-Array

Definition: Range-Minimum-Query

Sei $A[1..m)$ ein Array. Eine **Range-Minimum-Query** für den Bereich $\ell \leq r \in [1, n)$ ergibt

$$RMQ_A(\ell, r) = \arg\min\{A[k] \colon k \in [\ell, r]\}$$

- $lcp(i, j) = \max\{k \colon T[i..i + k) = T[j..j + k)\} = LCP[RMQ_{LCP}(i + 1, j)]$
- RMQs können in $O(1)$ Zeit beantwortet werden und
- benötigen $O(n)$ Wörter Platz
- werden in Advanced Data Structures behandelt

	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$
abaabcabcabba$
abbba$
abcabba$
abcabcabba$
ba$
babcabcabba$
bba$
bcabba$
bcabcabba$
cabba$
cabcabba$

LZ77-Faktorisierung mit *SA*, *ISA*, *LCP*, *NSV*, *PSV* und *RMQ*s

Function $LZ77(T, SA, ISA, LCP, RMQ, PSV, NSV)$**:**

1 $pos = 1$
2 **while** $pos \leq n$ **do**
3 $psv = SA[PSV[ISA[pos]]]$
4 $nsv = SA[NSV[ISA[pos]]]$
5 **if** $lcp(pos, psv + 1) > lcp(pos + 1, nsv)$ **then**
6 $\ell = lcp(pos, psv + 1)$ und $p = psv$
7 **else**
8 $\ell = lcp(pos + 1, nsv)$ und $p = nsv$
9 **if** $\ell = 0$ **then** $p = T[pos]$
10 **neuer Faktor** (ℓ, p)
11 $pos = pos + \max\{\ell, 1\}$

- Beispiel für Kompression und Dekompression an der Tafel

LZ77: Laufzeit

Lemma: LZ77 Laufzeit

Die LZ77-Faktorisierung von einem Text der Länge n kann in $O(n)$ Zeit berechnet werden

Proof (Sketch)

- SA, LCP, PSV, NSV, RMQ_{LCP} können in $O(n)$ Zeit berechnet werden
- für jedes Zeichen maximal $O(1)$ Zeit

Rank- und Select-Anfragen

- zum Abschluss noch einen Index
- Anwendung als Textindex und
- für Bereichsanfragen

Definition:

Sei T ein Text der Länge n über einem Alphabet $\Sigma = [1, \sigma]$ und $\alpha \in \Sigma$, dann ist

- $rank_\alpha(i) = |\{j \in [1, i) \colon T[j] = \alpha\}|$
- $select_\alpha(i) = \min\{j \in [2, n+1] \colon rank_\alpha(j) = i\} - 1$

- rank: Wie oft kommt ein Zeichen vor mir vor?
- select: Position des j-ten Vorkommens.

Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank-Anfragen in Bit-Vektoren (1/2)

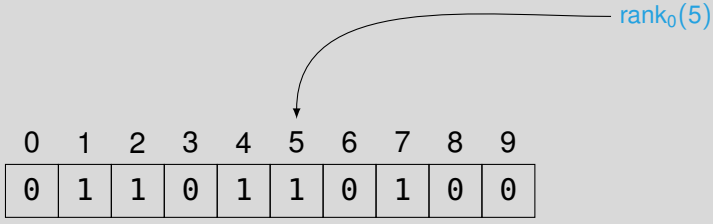
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α

$\text{rank}_0(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

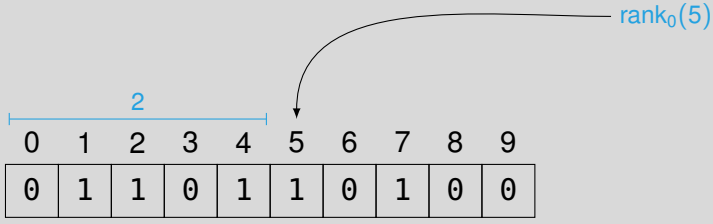
Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



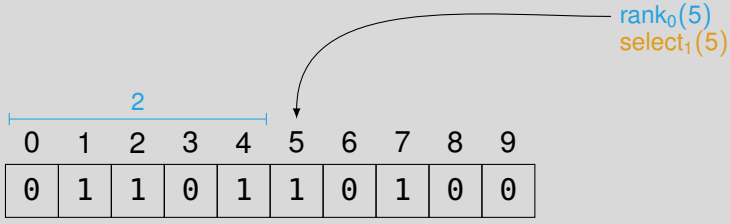
Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

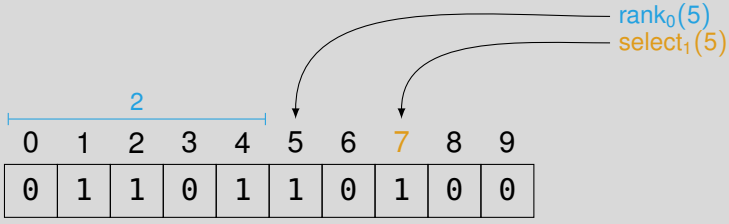
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

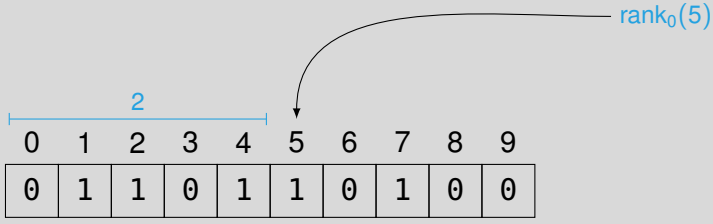
$\text{rank}_\alpha(i)$ # an αs vor Position i

$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

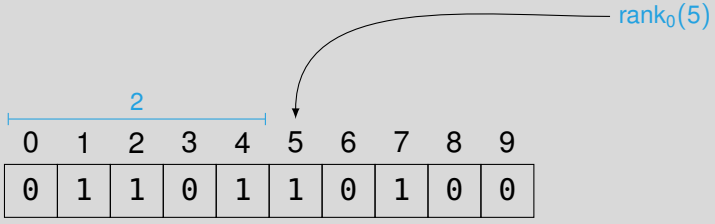
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

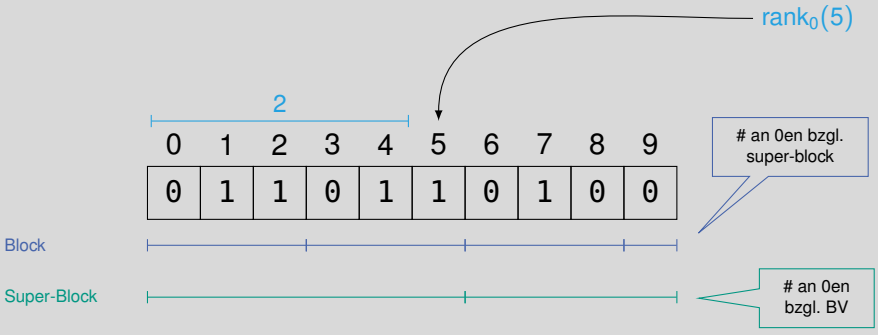
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

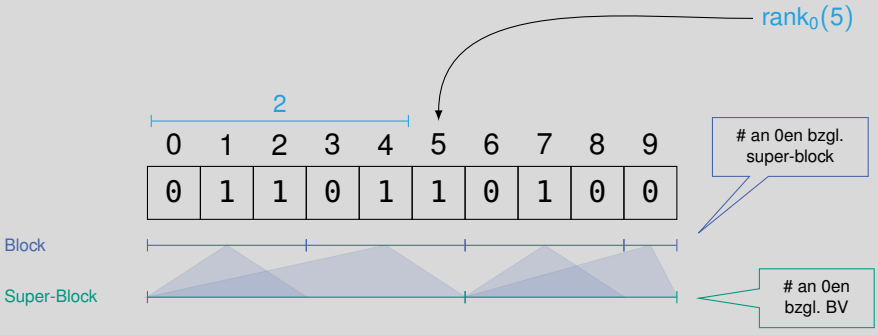
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

- für alle Bitvektoren der Länge s, speichere für jede Position i die Anzahl der 0en bis zu der Position
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

- für alle Bitvektoren der Länge s, speichere für jede Position i die Anzahl der 0en bis zu der Position
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ Bits Platz

- $rank_0(i) = i - rank_1(i)$

Select-Anfragen in Bitvektoren

- Select-Anfragen auch in $O(1)$ Zeit und
- $o(n)$ Bits Platz

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

MSB	0	0	1	1	0	1	1	0	1	0
	0	0	1	1	0	0	0	1	1	1
LSB	0	1	0	1	1	1	0	0	0	1

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

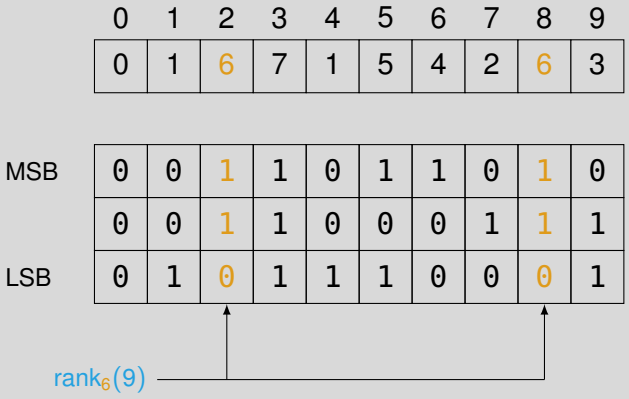
0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

MSB	0	0	1	1	0	1	1	0	1	0
	0	0	1	1	0	0	0	1	1	1
LSB	0	1	0	1	1	1	0	0	0	1

$\text{rank}_6(9)$

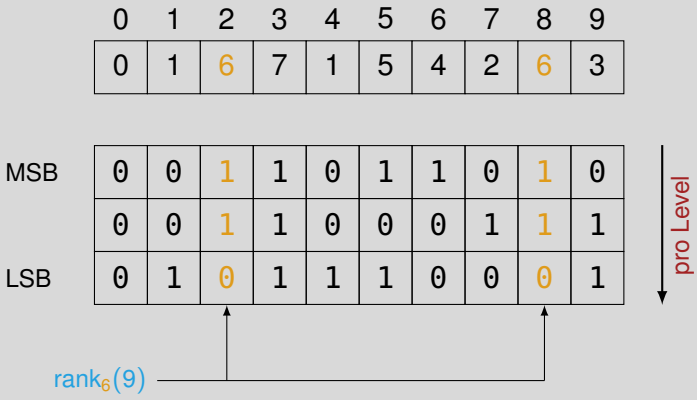
Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Wavelet-Trees [GGV03] (1/2)

Definition: Wavelet-Tree

Für einen Text T der Länge n über dem Alphabet $\Sigma = [1, \sigma]$ ist ein **Wavelet-Tree** ein Binärbaum, bei dem

- jeder Knoten Zeichen in $[\ell, r] \subseteq [1, \sigma]$ repräsentiert
- das linke und rechte Kind eines Knotens der Zeichen in $[\ell, r]$ repräsentiert
- repräsentieren Zeichen in $[\ell, (\ell + r)/2)$ und $[(\ell + r)/2, r]$
- ein Knoten ist ein Blatt, wenn $\ell + 2 \geq r$
- Zeichen werden durch ein Bit in einem Bitvektor repräsentieren
- Eintrag ist 1, wenn Zeichen im rechten Kind repräsentiert wird, sonst 0

Wavelet-Trees (2/2)

$[0, 7]$

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

→

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

Wavelet-Trees (2/2)

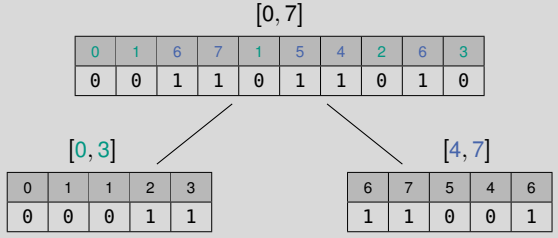
[0, 7]

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

→

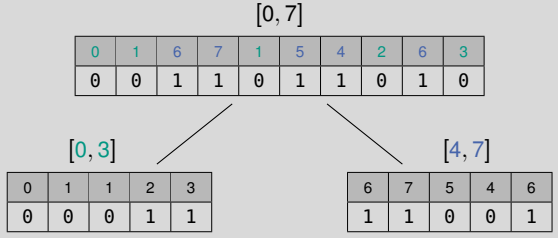
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)



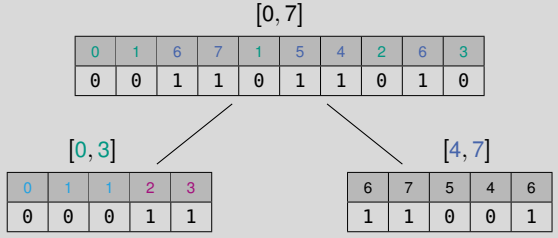
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)



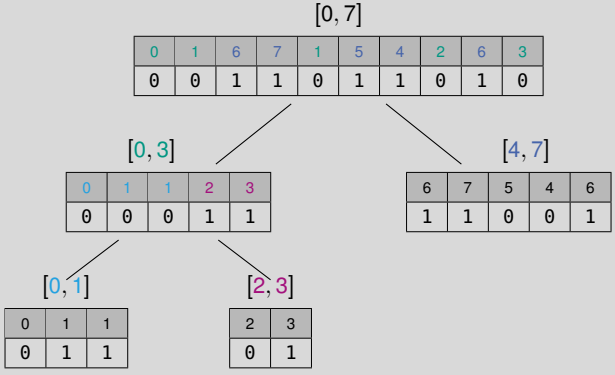
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)



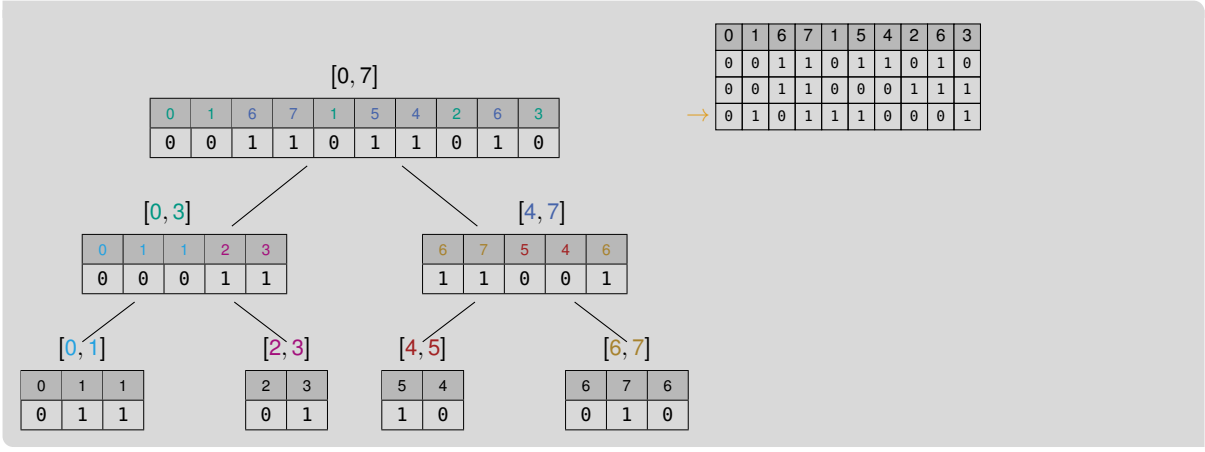
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)

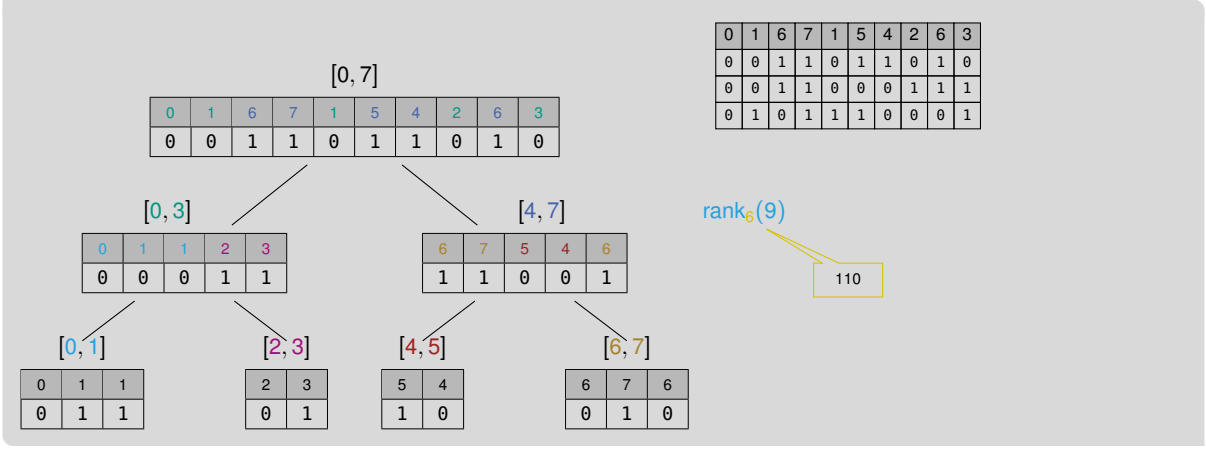


0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

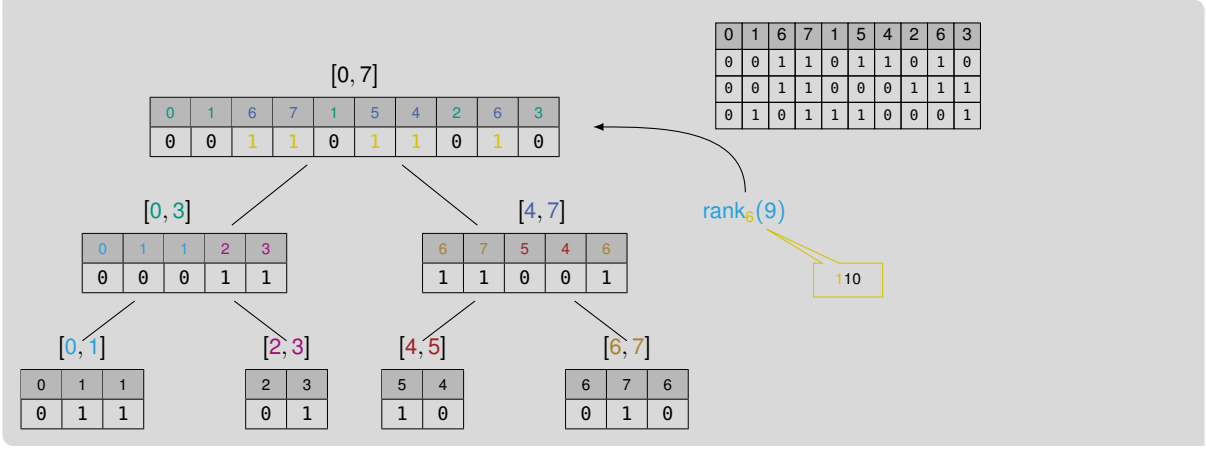
Wavelet-Trees (2/2)



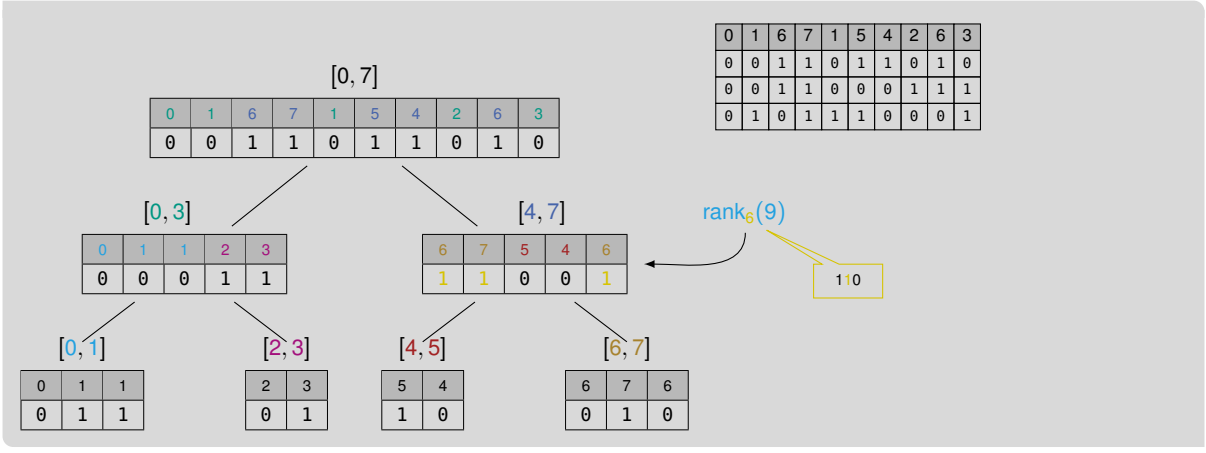
Wavelet-Trees (2/2)



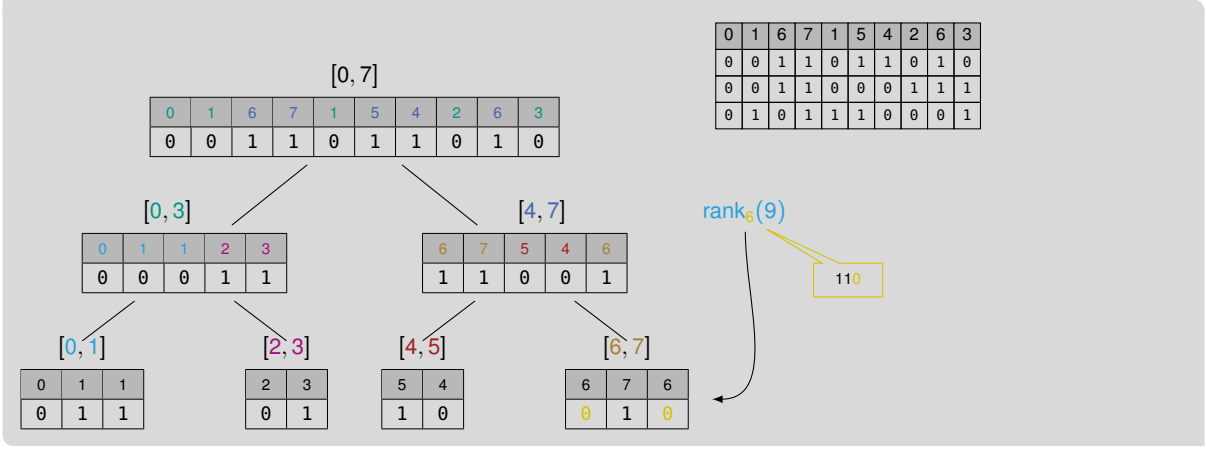
Wavelet-Trees (2/2)



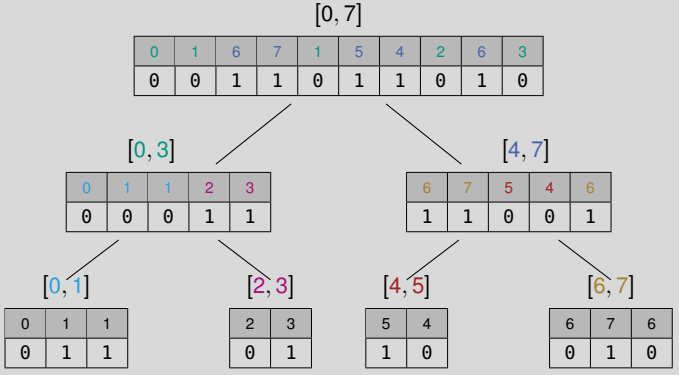
Wavelet-Trees (2/2)



Wavelet-Trees (2/2)

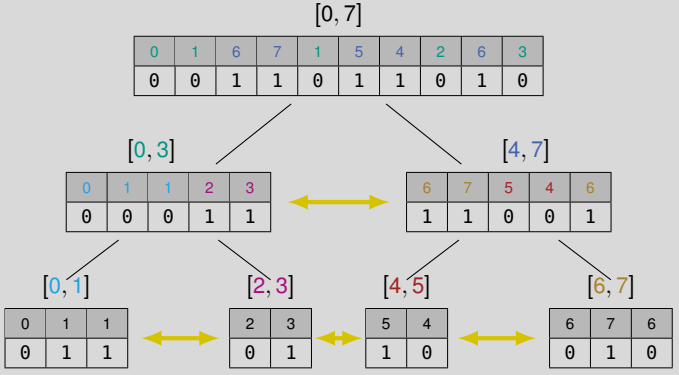


Wavelet-Trees (2/2)



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

Wavelet-Trees (2/2)



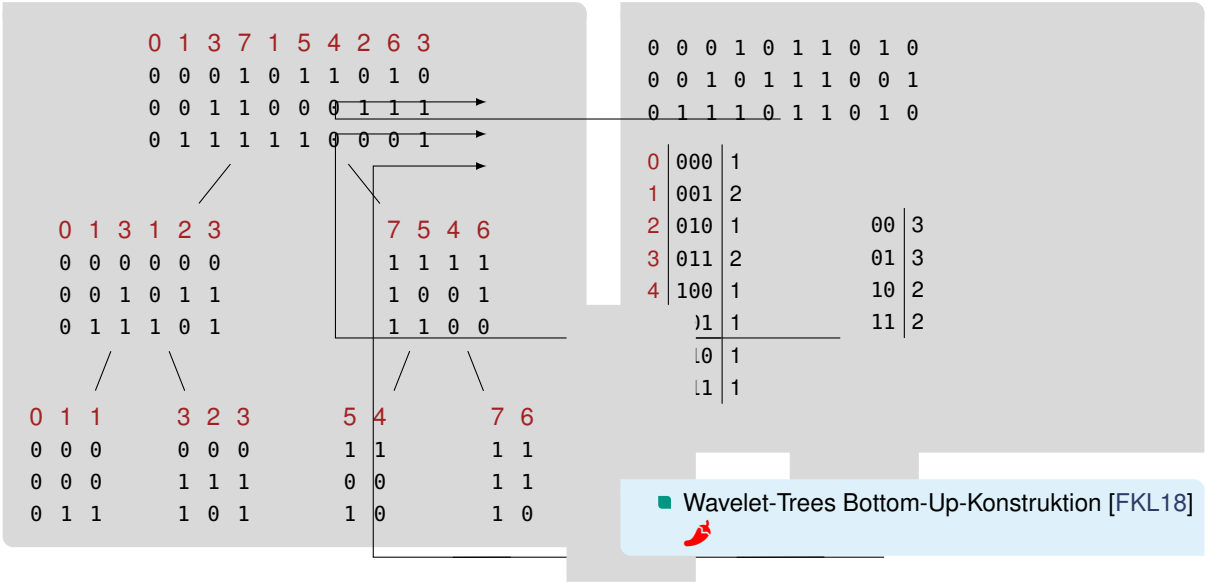
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees: Konstruktion und Anfragen

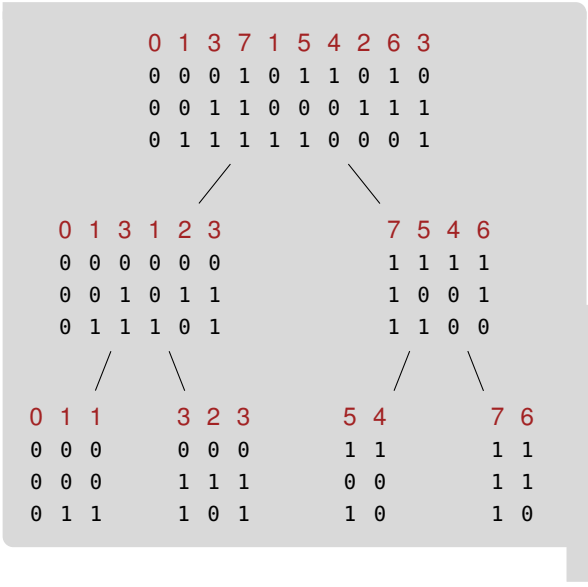
Der Wavelet-Tree

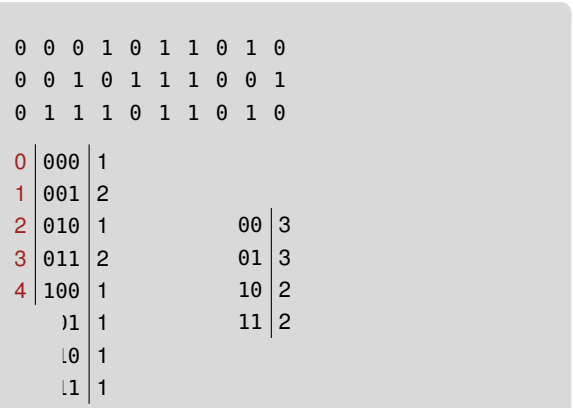
- kann naiv in $O(n \lg \sigma)$ Zeit konstruiert werden
- kann in $O(n \lg \sigma / \sqrt{\lg n})$ Zeit konstruiert werden [Bab+15; MNV16] 
- benötigt $n\lceil \lg \sigma \rceil (1 + o(1))$ Bits Platz

- Rank- und Select-Anfragen können in $O(\lg \sigma)$ Zeit beantwortet werden
- Zugriff auf Zeichen in $O(\lg \sigma)$ Zeit möglich

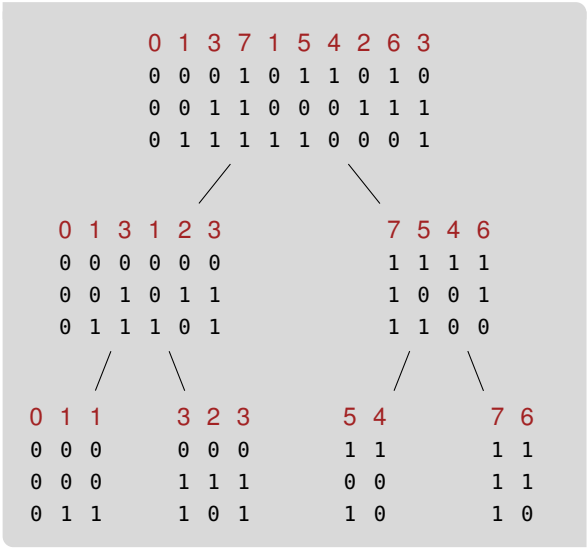


- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]





- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]

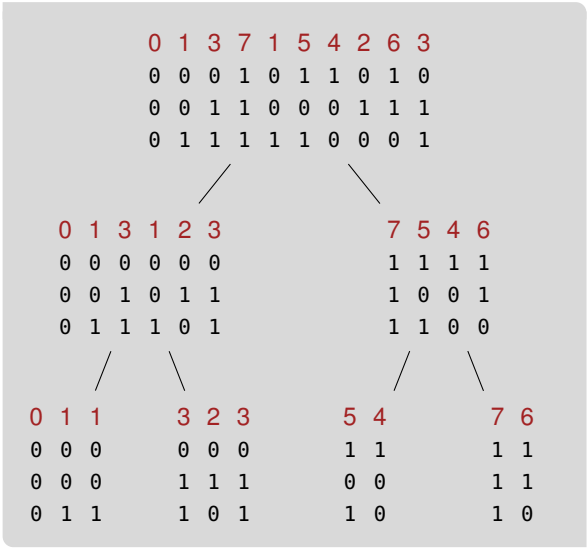


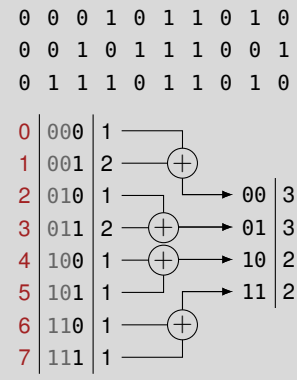

```
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0
```

0	000	1
1	001	2
2	010	1
3	011	2
4	100	1
5	101	1
6	110	1
7	111	1

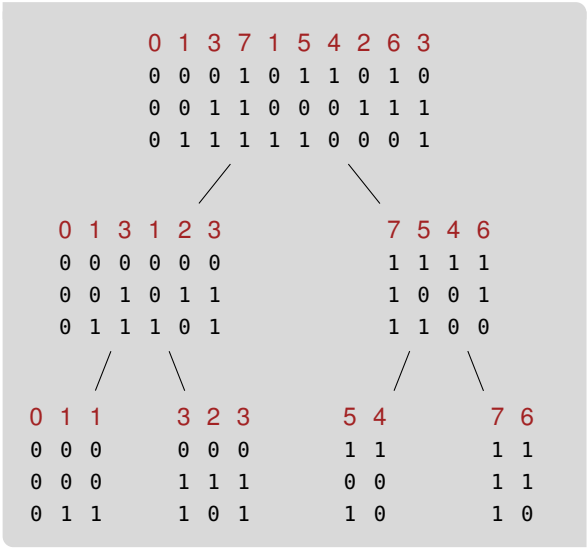
00	3
01	3
10	2
11	2

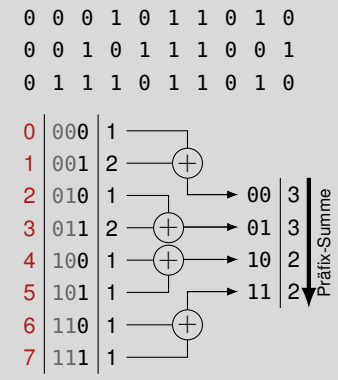
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]



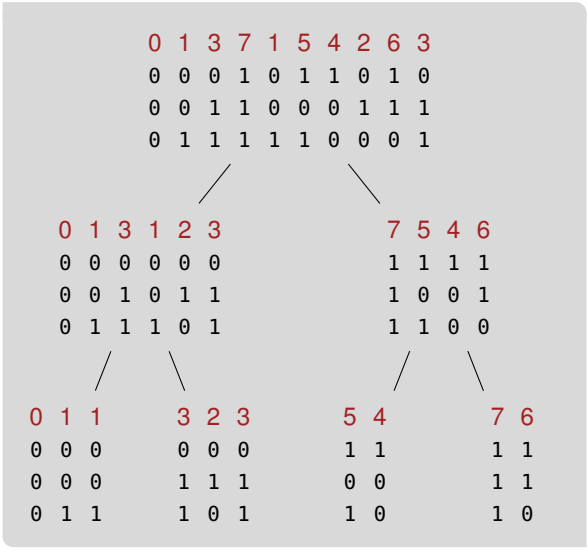


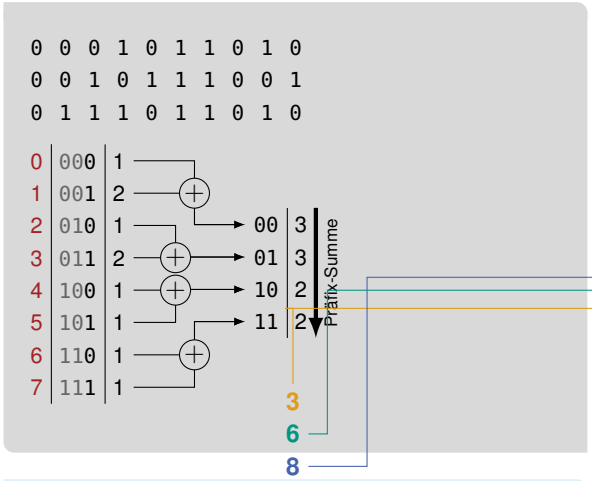
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]





- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]





- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]

Literatur I

[Bab+15] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka und Tatiana Starikovskaya. „Wavelet Trees Meet Suffix Trees“. In: *SODA*. SIAM, 2015, Seiten 572–591. DOI: 10.1137/1.9781611973730.39.

[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.

[FKL18] Johannes Fischer, Florian Kurpicz und Marvin Löbel. „Simple, Fast and Lightweight Parallel Wavelet Tree Construction“. In: *ALENEX*. SIAM, 2018, Seiten 9–20. DOI: 10.1137/1.9781611975055.2.

[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.

[GGV03] Roberto Grossi, Ankur Gupta und Jeffrey Scott Vitter. „High-Order Entropy-Compressed Text Indexes“. In: *SODA*. ACM/SIAM, 2003, Seiten 841–850.

Literatur II

[Huf52] David A. Huffman. „A Method for the Construction of Minimum-Redundancy Codes“. In: *Proceedings of the IRE* 40.9 (1952), Seiten 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[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.

[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.

[MNV16] J. Ian Munro, Yakov Nekrich und Jeffrey Scott Vitter. „Fast construction of wavelet trees“. In: *Theor. Comput. Sci.* 638 (2016), Seiten 91–97. DOI: 10.1016/j.tcs.2015.11.011.

Literatur III

[ZL77] Jacob Ziv und Abraham Lempel. „A Universal Algorithm for Sequential Data Compression“. In: *IEEE Trans. Inf. Theory* 23.3 (1977), Seiten 337–343. DOI: 10.1109/TIT.1977.1055714.