

Algorithmen 2

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

Florian Kurpicz

The slides are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License © ⓘ ⓘ: www.creativecommons.org/licenses/by-sa/4.0 | commit e9153df compiled at 2021-11-17-10:07

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

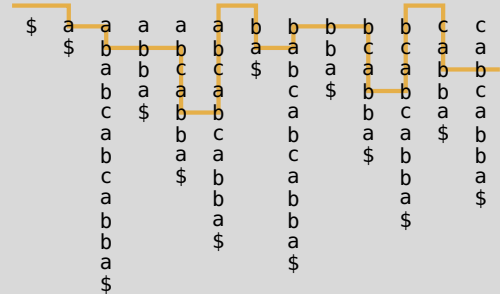
$$T[SA[i]..n] \leq T[SA[j]..n]$$

Definition: Longest-Common-Prefix-Array

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

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

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





<https://pingo.scc.kit.edu/221136>

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

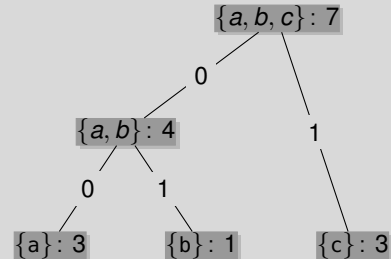
- Verweise für erneute Vorkommen
- Vorkommen können überlappen

Huffman-Kodierung [Huf52]

- erstelle Binärbaum
- jedes Zeichen α ist ein Blatt mit Gewicht $Hist[\alpha]$
- 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

$T = cbcacaa$



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

Probleme der Huffman-Kodierung

aa
aa
aa
aa
aa
aa
aa
aa
aa
aa
aa

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

$T = \text{abababbbbaba\$}$

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

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

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

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
 - $f_i = T[p_i..p_i + \ell_i)$

$T = \text{abababbbbaba\$}$

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

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

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

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

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


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 (2/2)

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

- 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 

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

```

1  |   for  $i = 1, \dots, n$  do
2  |      $j = i - 1$ 
3  |     while  $j \geq 1$  und  $SA[i] < SA[j]$  do
4  |       |  $j = PSV[j]$ 
5  |       |  $PSV[i] = j$ 
6  |   return PSV
  
```

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

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

- $lcp(i, j) = \max\{k : T[i..i+k] = T[j..j+k]\} = LCP[RMQ_{LCP}(i+1, j)]$
- 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	a	a	a	a	b	b	b	b	b	c	c	
\$	b	b	b	b	b	a	a	b	b	c	a	a	
	a	b	a	a	a	\$	b	c	a	a	b	b	
	b	a	\$	b	b		a	b	b	b	a	a	
	c			b	c		c	a	a	a	\$	b	
	a			a	a		b	c	\$	b		b	
	b			a	b		a	a		a		a	
	c			\$	b		b	b		b		a	
	a				a		a	b		a		\$	
	b				\$		b	a					
	a						\$						
	\$												


LZ77-Faktorisierung mit SA , ISA , LCP , NSV , PSV und $RMQs$

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

```

1  |   pos = 1
2  |   while pos ≤ 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  |       |       |   ℓ = lcp(pos, psv + 1) und p = psv
7  |       |       |   else
8  |       |       |       |   ℓ = lcp(pos + 1, nsv) und p = nsv
9  |       |       |       |   if ℓ = 0 then p = T[pos]
10 |       |       |       |   neuer Faktor (ℓ, p)
11 |       |       |       |   pos = pos + max{ℓ, 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

Beweis (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

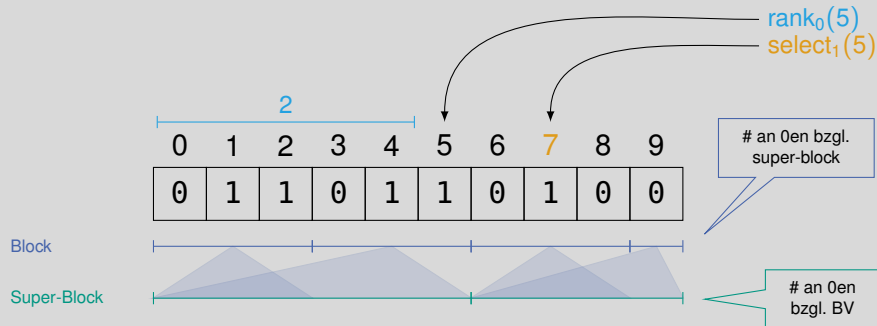
- $rank_{\alpha}(i) = |\{j \in [1, i) : T[j] = \alpha\}|$
- $select_{\alpha}(i) = \min\{j \in [2, n + 1] : rank_{\alpha}(j) = i\} - 1$

- 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 α s vor Position i

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



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

- 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

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

pro Level

$\text{rank}_6(9)$



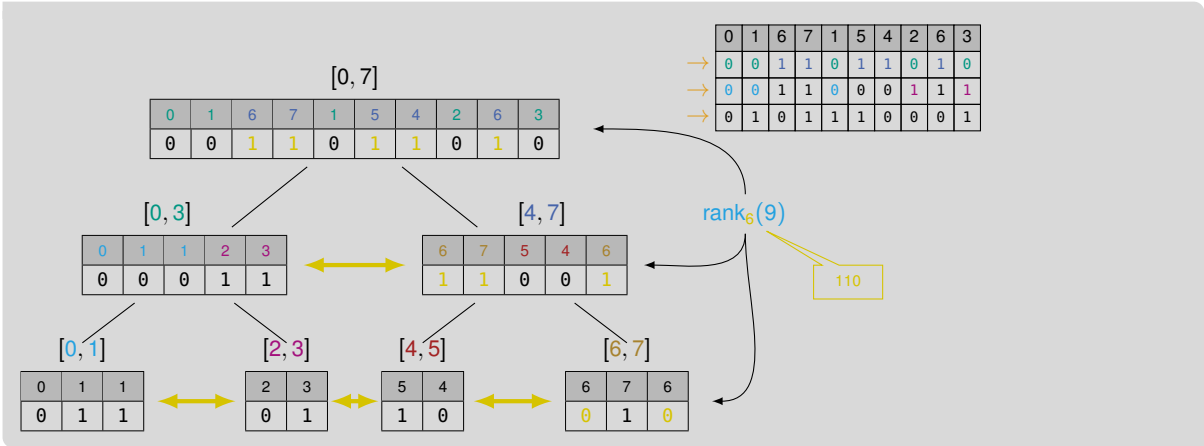
Wavelet-Trees [GGV03] (1/2)

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 $[\ell, r]$ repräsentieren
- 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

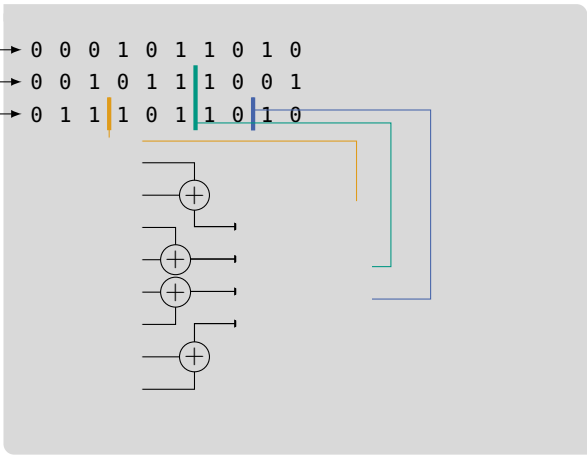
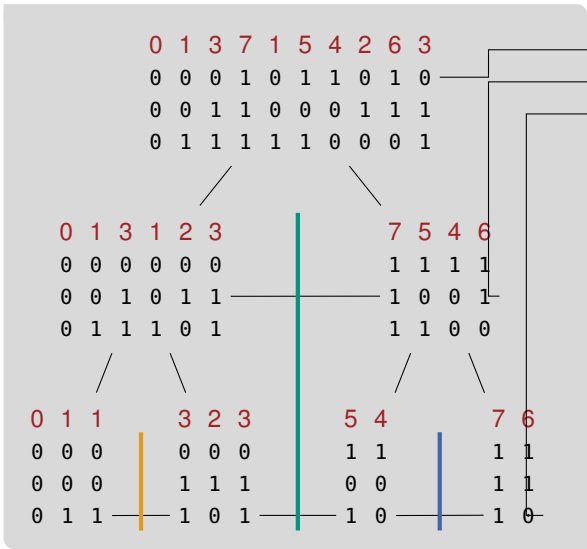
Wavelet-Trees (2/2)




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]



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](https://doi.org/10.1137/1.9781611973730.39).
- [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](https://doi.org/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.
- [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](https://doi.org/10.1109/JRPROC.1952.273898).

Literatur II

- [MM93] Udi Manber und Eugene W. Myers. „Suffix Arrays: A New Method for On-Line String Searches“. In: *SIAM J. Comput.* 22.5 (1993), Seiten 935–948. DOI: [10.1137/0222058](https://doi.org/10.1137/0222058).
- [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](https://doi.org/10.1016/j.tcs.2015.11.011).
- [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](https://doi.org/10.1109/TIT.1977.1055714).