

# Suffix Arrays

Eine Datenstruktur für Stringalgorithmen

10. Vorlesung

08/10.Mai 2007



Universität Dortmund, FB Informatik, LS11

## Überblick

- Suffix Array – Anwendungsfelder, Struktur
- Aufbau:  
Algorithmus von Kärkkäinen, Sanders, Burkhardt 2004
- Suchen in Suffix Arrays  
Naiv, Beschleunigung
- Anwendungen

## Problemstellung

- Suche von Zeichenketten in Zeichenketten
- In Bioinformatik:
  - Proteinidentifikation
  - Genom Alignment
  - Repeatsuche
- Kompression: Lempel-Ziv (GIF, PDF,...), Burrows-Wheeler (BZIP2)
- Circular String Linearization in der Chemie
- Web Suche, Wissensnetze

TAVIKKLLKQINEEQREGLMDDL RGVNLSKF  
INEEQREG ?

## Problemstellung

- Suche von Zeichenketten in Zeichenketten
- In Bioinformatik:
  - Proteinidentifikation
  - Genom Alignment
  - Repeatsuche
- Kompression: Lempel-Ziv (GIF, PDF,...), Burrows-Wheeler (BZIP2)
- Circular String Linearization in der Chemie
- Web Suche, Wissensnetze

TAVIKKLLKQ**INEEQ**REGLMDDL RGVNLSKF  
**INEEQREG ?**

## Problemstellung

- Hürde: Riesige Datenmengen (menschl. Genom ca. 3 Mrd. Basenpaare)
- Quadratisch viele Substrings – nicht explizit speicherbar
- Klassische Algorithmen überfordert (Suchzeit linear)

## „Lösung“

- Suffix Array:
- Platzsparende Datenstruktur
  - Schneller Aufbau (amortisiert über Suchen)
  - Schnelle Suche
  - Gut für External Memory Nutzung
  - Flexibel auch als Ersatz z.B. für Suffix Tree und viele spezielle Fragestellungen

## Engineering Aspekte

- In Praxis: Komplexität kritisch
- Zwar: Speicherplatzbedarf geringer als bisherige Lösungen
- Aber: Schon Konstanten kritisch → Rallye nach impl. Verbesserungen für
  - Speicherplatz (Aufbau)
  - Laufzeit (Aufbau)
  - Laufzeit (Suche)
- Prakt. Effizienz hängt ab von Textlänge, Musterlänge, Alphabet, Anzahl Suche/Repeats

## Lösungsansatz

- Jeder Teilstring ist Präfix eines Suffix

INEEQREG

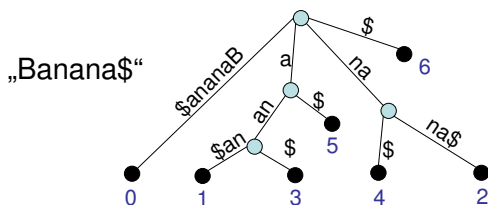
TAVIKKLLKQINEEQREGLMDDLGRGVNLSKF\$

Es gibt nur n Suffixe!

- ⇒ Nur Suffixe anschauen, Präfix testen
- Eindeutigkeit?
  - ⇒ Kein Suffix als Präfix von Suffix: Sentinel \$

## Exkurs: Suffix Tree (McCreight/Weiner 73)

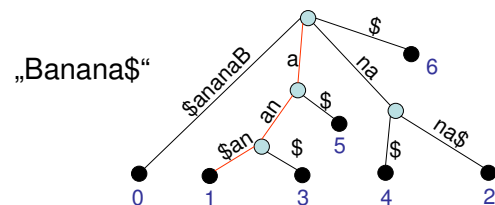
Idee: Darstellung aller Suffixe als Baum



Blätter entsprechen Suffix

## Exkurs: Suffix Tree (McCreight/Weiner 73)

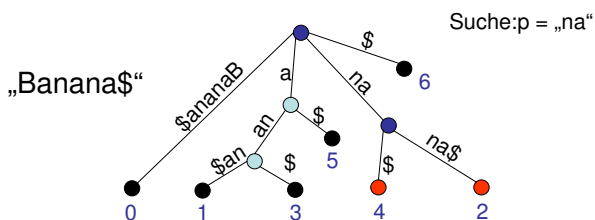
Idee: Darstellung aller Suffixe als Baum



Pfad von Wurzel zu Blatt mit Suffixtext annotiert

## Exkurs: Suffix Tree (McCreight/Weiner 73)

Idee: Darstellung aller Suffixe als Baum



Vorkommen in Unterbaum  
Bei endl. Alphabet Suche in  $O(|p| + \#Treffer)$

## Suffix Tree – Vor-/Nachteile

- Sehr gut untersucht, sehr viele Verbesserungen
  - Zeit für Bau/Suche linear, abhängig von Alphabetgröße  $O(n \log |\Sigma|)$  bzw.  $O(|p| \log |\Sigma|)$
  - Platzbedarf abhängig von Alphabet, 10-20 Bytes pro Zeichen
  - Schlecht für Externspeicher anpassbar (Lokalität!)
  - Implementierung (relativ) „aufwendig“
- ⇒ Suffix Array

## Suffix Array (Manber/Myers 91)

- Repräsentiert sortierte Liste der Suffixe in einem Integer Array
- Entwicklungsziel: Platzsparend für extrem grosse Datenmengen
- Aufbauzeit damals:  $O(n \log n)$  (direkt)
- Heute: Zeit linear, kleiner Faktor (große Menge verschiedener Algorithmen in kurzer Zeitspanne)
- Für Praxis besser als Suffix Tree, unabhängig von Alphabet, deutlich kleiner
- Suche sublinear
- Auch: PAT-Array (Gonnet et al.)

## Suffix-Array

### Sortierung von String-Suffixen

Ordnung auf Alphabet  $\Rightarrow$  lexikographisch Sortieren

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

SA

SA[i]: An welcher Position beginnt i-ter Suffix in Sortierung?

## Suffix-Array

### Sortierung von String-Suffixen

Ordnung auf Alphabet  $\Rightarrow$  lexikographisch Sortieren

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

SA

Suffix Array SA von s: Integerarray mit  
 SA[i] = j : Suffix  $S_j$  hat Rang i in lexikogr. Ordnung

### Suffix Array - Beispiel

Ind	0	1	2	3	4	5	6	7	8	9	10
s	M	I	S	S	I	S	S	I	P	P	I

SA	10	7	4	1	0	9	8	6	3	5	2
----	----	---	---	---	---	---	---	---	---	---	---

Reverse Array R: SA[R[i]] = i

R	4	3	10	8	2	9	7	1	6	5	0
---	---	---	----	---	---	---	---	---	---	---	---

## Suffix Arrays

- Textindex durch Suffixsortierung
- Speicherung als Integerarray
- $|int|n$  bytes, Praxis:  $\sim 4n$  bytes +  $n$  bytes Text
- Größe unabhängig von Alphabetgröße

Algorithmen:

- Aufbau
- Suche
- Anwendungen: Repeat Suche, Kompression,...

### Aufbau von Suffix Arrays

## Aufbauvarianten

- Traversal des Suffix Tree: Aufwendig, Platzbedarf  $\geq 15n$ , Alphabetabhängig, Zeit  $O(n)$
- Einfacher Aufbau durch Sortierung  
**Achtung: Sortierung von Strings!**  
 $\Rightarrow$  Ternary (Multikey) Quicksort oder Bucketsort mit Tricks, Zeit  $O(n \log n)$
- „Geschicktere“ Sortierung in Linearzeit?  
Viele Variationen unter Ausnutzung der Suffixeigenschaft (Larsson-Sadakane, Itoh-Tanaka, Seward, Burkhardt-Kärkkäinen, ....)

## Aufbau

Divide & Conquer / Rekursion

Linearzeitalgorithmus **DC3** von Kärkkäinen, Sanders, Burkhardt 2004

Aufbau

## Schema DC3

1. Partitioniere Suffixe in Auswahlmenge („**Sample**“) und Rest
2. Erzeuge Sortierung für Sample
3. Sortiere Rest
4. Merge die erzeugten Arrays

Welche Partitionen?  
Wie kann man Merge durchführen?

Aufbau

## Ideen

- Anforderung an Sample und Mergen:
  1. Sample ist leicht zu sortieren ( $\Rightarrow$  nicht zufällig)
  2. Sample nützlich für Restsortierung:  
Suffixstruktur + bekannte Teilsortierung ausnutzen

$$\begin{aligned} S_0 &= t_0 t_1 t_2 t_3 \dots \\ S_1 &= t_1 t_2 t_3 \dots \\ S_2 &= t_2 t_3 \dots \\ S_3 &= t_3 \dots \end{aligned}$$

Aufbau

## Linearzeitalgorithmus DC3

1. Auswahlmengen : Dreiteilung der Indizes  
 $B_k = \{i \in [0, \dots, n] \mid i \bmod 3 = k\}$  für  $k = 0, 1, 2$   
Samplepositionen  $C = B_1 \cup B_2$   
Samplesuffixe  $S_C$

Aufbau

## Beispiel

- Samplepositionen  $i \bmod 3 \neq 0$ :

mississippi

1: ississippi  
2: ssissippi  
4: issippi  
5: ssippi  
7: ippi  
8: ppi  
10: i

Samplepositionen  $C = \{1, 2, 4, 5, 7, 8, 10\}$

Aufbau

## Linearzeitalgorithmus

- Erzeuge Sortierung für  $C$  ( $= 2/3n$ )
  - Sortierung des Restes  $B_0$  (mit  $i \bmod 3 = 0$ ):  
Für Suffix  $S_i$  gilt  $S_i = t_i S_{i+1}$   $i+1 \bmod 3 = 1$
- ⇒ Statt Stringvergleich nutze bekannten Sortierungsrang für Suffixe  $S_{i+1}$
- Also für  $i, j \in B_0$  Vergleich
- $$S_i \leq S_j \Leftrightarrow (t_i, \text{Rang}(S_{i+1})) \leq (t_j, \text{Rang}(S_{j+1}))$$

Aufbau

## Beispiel

Text  $s = \text{mississippi}$

$S_C$

1: ississippi 4  
 2: ssissippi  
 4: issippi 3  
 5: ssippi  
 7: ippi 2  
 8: ppi  
 10: i 1

$S_{B_0}$

0: m-ississippi (m,4)  
 3: s-issippi (s, 3)  
 6: s-ippi (s, 2)  
 9: p-i (p, 1)

Ergebnis 0,9,6,3

Aufbau

## Berechnung für Menge C

- $C$ : Positionen  $i \bmod 3 = 1,2$  **TRICK!**
- Rechnung modulo 3 → Zeichentripel  $[t_{i+1}, i+2]$
- Tripel beginnen mit Zeichen mit gleichem mod-Rest
- $$R_k = [t_{k, k+1, k+2}] \dots [t_{\max B_k, \max B_{k+1}, \max B_{k+2}}], \quad k=1,2$$
- $R_1 = \text{iss iss ipp i\$\$}$  **NOCH MEHR TRICKS!**
- $R = R_1 \oplus R_2 = \text{iss iss ipp i\$\$ ssi ssi ppi}$
- Suffixe in  $S_C$  und  $R$  entsprechen sich!
- $S_i$  entspricht  $[t_{i+1}, i+2][t_{i+3}, i+4, i+5] \dots$

Aufbau

## Berechnung für Menge C

- Sortiere Tripel und benenne sie mit Rang
- Eindeutig? → Suffixsortierung nach erstem Tripel
- Nicht eindeutig? → Rekursiver Aufruf mit Größe  $2/3$

3 3 2 1 5 5 4  
 iss iss ipp i\\$\\$ ssi ssi ppi

- Rang für Suffixe  $\text{rang}(S_i)$  für  $R$  bzw.  $S_C$
- Laufzeit?  $T(n) = T(2/3n) + O(n) \Rightarrow O(n)$

Aufbau

## Linearzeitalgorithmus

- Erzeuge Suffixarray für Positionen  $i \bmod 3 \neq 0$   
Rekursiv mit Aufteilung in Problem der Größe  $2/3$
- Erzeuge Suffixarray für  $i \bmod 3 = 0$  unter Benutzung des Arrays aus 1.
- Merge die zwei Mengen

Wie Merge ausführen?

Aufbau

## Merge

- Standardmerge mit Durchlauf der Arrays
- Vergleiche  $S_i \in S_C$  mit  $S_j \in S_{B_0}$ 
  - $i \in B_1: (t_i, \text{rang}(S_{i+1})) \leq (t_j, \text{rang}(S_{j+1}))$
  - $i \in B_2: (t_i, t_{i+1}, \text{rang}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rang}(S_{j+2}))$

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



i

0	9	6	3
---	---	---	---



mississippi

mississippi

SA	10													
----	----	--	--	--	--	--	--	--	--	--	--	--	--	--

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



ippi

0	9	6	3
---	---	---	---



mississippi

mississippi

SA	10	7												
----	----	---	--	--	--	--	--	--	--	--	--	--	--	--

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



issippi

0	9	6	3
---	---	---	---



mississippi

mississippi

SA	10	7	4											
----	----	---	---	--	--	--	--	--	--	--	--	--	--	--

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



ississippi

0	9	6	3
---	---	---	---



mississippi

mississippi

SA	10	7	4	1										
----	----	---	---	---	--	--	--	--	--	--	--	--	--	--

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



ppi

0	9	6	3
---	---	---	---



mississippi

mississippi

SA	10	7	4	1	0									
----	----	---	---	---	---	--	--	--	--	--	--	--	--	--

### Beispiel Merging

10	7	4	1	8	5	2
----	---	---	---	---	---	---



ppi

0	9	6	3
---	---	---	---

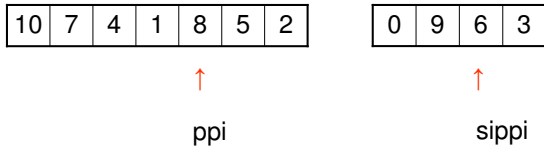


pi

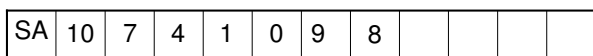
mississippi

SA	10	7	4	1	0	9								
----	----	---	---	---	---	---	--	--	--	--	--	--	--	--

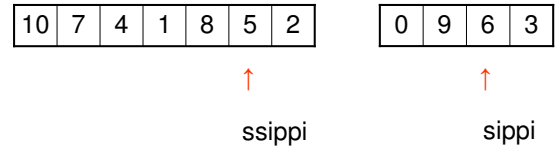
### Beispiel Merging



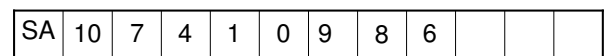
mississippi



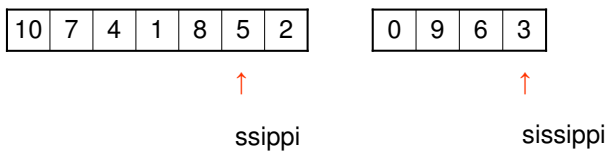
### Beispiel Merging



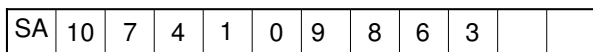
mississippi



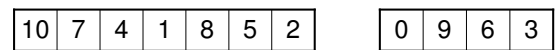
### Beispiel Merging



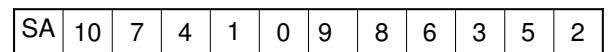
mississippi



### Beispiel Merging



mississippi



### Allgemeiner Ansatz

- DC3 ist ein Spezialfall:
  - Auswahl von  $C$  beruht auf sog. „Difference Cover“ modulo  $v=3$
  - Beliebige  $v$  aus  $[1, \sqrt{(n)}]$  möglich
- DC erlaubt Zeit-Platz Tradeoff für Wahl von  $v$ :
  - Zeit  $O(vn)$
  - Platz  $O(n / \sqrt{(v)})$  ohne Eingabe / SA
- Samplegröße  $O(n / \sqrt{(v)})$
- Für  $S_i, S_j: \exists$  innerhalb Abstand  $k \leq v$  Paar  $S_{i+k}, S_{j+k}$  dessen Vergleich bekannt ist  $\Rightarrow$  Nur  $\leq k$  Zeichen vergleichen

### Aufbau

- Aufbau in  $O(n)$  Zeit und Platz
- Mehrere Algorithmen mit untersch. Eigenschaften bzgl. Platz/Zeit je nach Eingabe
- Aufbauzeit kann über mehrere Suchen amortisiert werden

- External Memory Algorithmen verfügbar: Crauser/Ferragina, Dementiev et al., ...
- Komprimierte/Kompaktierte Suffix Arrays

Suche?



SA	11	8	5	2	1	L	9	M	R	6	3
----	----	---	---	---	---	---	---	---	---	---	---

i i i i m p p s s s s  
 p s s i i p i i s s  
 p s s s i p s i i  
 i i i s p s p s  
 p s i i i p s  
 p s s i i p i  
 i i s 1 Vorkommen p p  
 p i i p  
 p p Mehrere Vorkommen? i  
 i p Vorkommen benachbart!  
 i

sip = sip

SA	11	8	5	2	1	10	9	7	L	M	R
----	----	---	---	---	---	----	---	---	---	---	---

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

Suche nach ssi

2 Vorkommen

## Suche

- Laufzeit naiv:  
logn Strings, Musterlänge  $m$ , linear  $k$  Nachbarn suchen  
→  $O(m(\log n + k))$
- Lange gleiche Präfixe – hohe Laufzeit
- Verbesserung praktisch / theoretisch ?

## Suche

### Beschleunigung

Auslassen bekannter Präfixe

L                      R  
 ↓                                      ↓

SA	11	8	5	2	1	10	9	7	4	6	3
----	----	---	---	---	---	----	---	---	---	---	---

Gem. Präfix →

i i i i m p p s s s s  
 p s s i i p i i s s  
 p s s s i p s i i  
 i i i s p s p s  
 p s i i i p s  
 . . . . i .

$lcp(S_i, S_j)$  – longest common prefix von  $S_i$  und  $S_j$

## Suche

### Beschleunigung

- $l_{match} = \min(lcp(p, S_{SA[L]}), lcp(p, S_{SA[R]}))$
- $\forall k = L, \dots, R:$   
 $p[1..l_{match}] = s[SA[k] .. SA[k] + l_{match} - 1]$   
 → Nicht vergleichen

- Laufzeitverbesserung:
- Praktisch: Klar
  - Theoretisch: Worst-Case?  $min = 0$
- keine Verbesserung der asympt. Worst-Case Laufzeit

## Suche

### Praxis: lcp( $S_i, S_{i+1}$ )-Werte

Datei	Average	Maximum	Größe
Swissprot	89	7.373	110Mb
Chrom. 22	1.980	200.000	34 Mb
gcc3 src	8.600	856.970	87 Mb