

# Suffix Arrays

Eine Datenstruktur für  
Stringalgorithmen



# Anwendungen

- Kompression: Lempel-Ziv (GIF, PDF,...), Burrows-Wheeler (BZIP2)...
- Bioinformatik: Genom Alignment, Repeatsuche
- Information Retrieval, Datenbanken
- Circular String Linearization in der Chemie

<b>CGCTTTA</b>	<b>-CGCTTTA</b>
<b>ACGTT</b>	<b>ACG-TT--</b>

# Engineering Aspekte

- Praktische Anforderungen: Komplexität kritisch
- Speicherplatzbedarf: Geringer als bisherige Lösungen
- Aber: Konstanten kritisch → Rallye nach impl. Verbesserungen für
  - Speicherplatz
  - Laufzeit Erstellung
  - Laufzeit Suche

# Problemstellung

- Suche von Zeichenketten in Zeichenketten
- Hürde: Riesige Datenmengen (Wissensnetze, Bioinformatik)
- Quadratisch viele Substrings – nicht explizit speicherbar (menschl. Genom ca. 3 Mrd. Basenpaare)

klsadjfsdfscsdcdcefwefefefeefe

# Lösungsansatz

- j-ter Suffix  $S_j$  von  $T[0..n]$  ist  $T[j..n]$

Banana

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

# Lösungsansatz

- Lösung für Suche:  
Jeder Teilstring ist Präfix eines Suffix

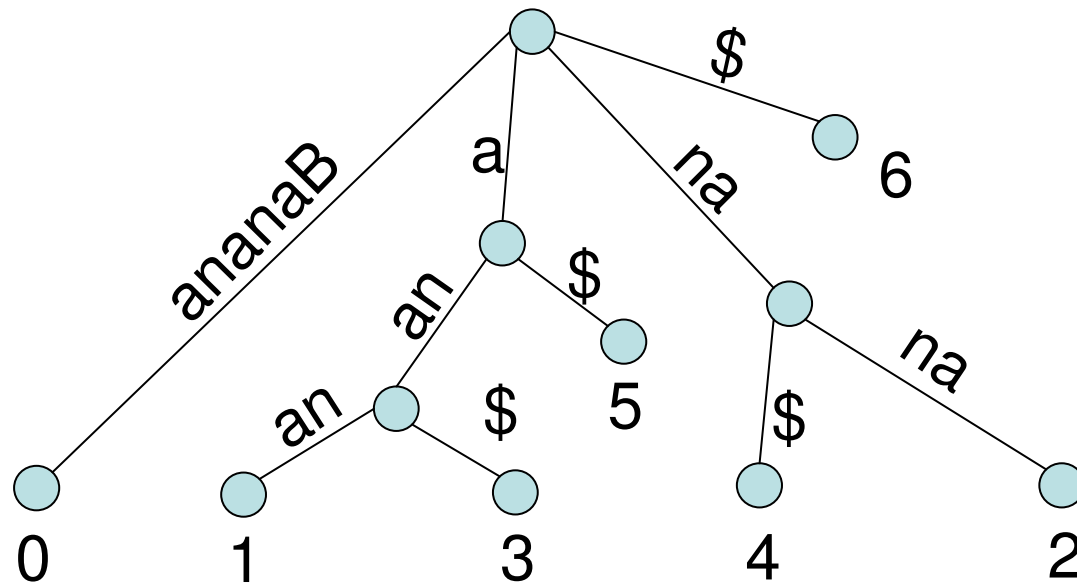
klsadjfsdfscsdcdcefwefefefefefe\$

- Es gibt nur n Suffixe
- Eindeutigkeit: Kein Suffix als Präfix  
von Suffix: Sentinel \$

# Suffix Tree (McCreight/Weiner 73)

- Idee: Darstellung aller Suffixe als Baum

Banana



# Suffix Tree

- Problem:
  - Bau/Suche linear, Abhängig von Alphabetgröße  $\alpha$
  - Platzbedarf, ca. 15 Byte pro Zeichen
  - Implementierung aufwendig

# Suffix Array (Manber/Myers 91)

- Entwicklungsziel: Platzsparend für extrem grosse Datenmengen
- Aufbau damals:  $O(n \log n)$  worst case
- Heute: Aufbau linear, kleiner Faktor
- Besser als Suffix Tree

# Sortierung von String-Suffixen

- Ordnung auf Alphabet

Banana

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

Suffix Array SA von  $s$ : Integerarray mit

$SA[i] = j$  : Suffix  $S_j$  hat Rang  $i$  in lexikogr. Ordnung

# Suffix Array

Ind	0	1	2	3	4	5	6	7	8	9	10
T	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

- Integerarray
- $|int|n$  bytes, Praxis:  $\sim 4n$  Byte
- Größe unabhängig von Alphabetgröße  $\alpha$

Algorithmen:

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

# Aufbau

- Sortierung in quadratischer Zeit
- DFS in Suffix Tree,  $n \cdot \alpha$ , Baum nötig

Bessere Idee: Divide & Conquer, rekursiv

# Schema

- Teile Suffixe in Mengen auf
- Erzeuge Suffixarray für jede Menge
- Merge die erzeugten Arrays

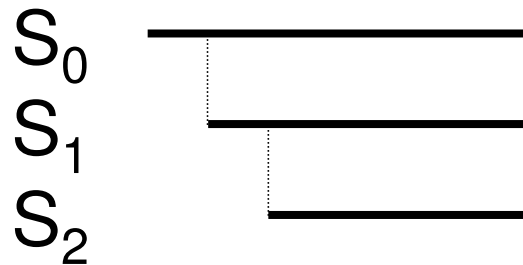
Welche Mengen?

Wie kann man Merge durchführen?

# Ideen

- Auswahl und Mergen:
  1. Auswahlmenge ist leicht zu sortieren
  2. Auswahlmenge nützlich für Restsortierung:  
Suffixstruktur + bekannte Teilsortierung

$T = t_0 t_1 t_2 t_3 \dots$



# Linearzeitalgorithmus

- Algorithmus von Kärkkäinen, Sanders, Burkhardt 2004

- Auswahlmengen: Dreiteilung

$$B_k = \{i \in [0, \dots, n] \mid i \bmod 3 = k\}$$

für  $k = 0, 1, 2$

- Samplepositionen  $C = B_1 \cup B_2$

Samplesuffixe  $S_C$

# Beispiel

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

mississippi

1: ississippi

2: ssissippi

4: issippi

5: ssippi

7: ippi

8: ppi

10: i

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

# Linearzeitalgorithmus

- Erzeuge Suffixarray für  $C$  ( $= 2/3n$ )
- Sortierung des Restes ( $i \bmod 3 = 0$ ):  
Für Suffix  $S_i$  gilt  $S_i = t_i S_{i+1}$
- Wir kennen aber bereits Sortierungsrang für alle solche Suffixe  $S_{i+1}$
- Also für  $i, j \in B_0$  Vergleich  
$$S_i \leq S_j \Leftrightarrow (t_i, \text{Rang}(S_{i+1})) \leq (t_j, \text{Rang}(S_{j+1}))$$

# Beispiel

**S<sub>c</sub>**

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

**S<sub>0</sub>**

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

# Berechnung für Menge C

C: Positionen  $i \bmod 3 = 1, 2$

TRICK!

Rechnung modulo 3  $\rightarrow$  Zeichentripel  $[t_{i,i+1,i+2}]$

$$R_k = [t_{k,k+1,k+2}] \cdots [t_{\max B_k, \max B_{k+1}, \max B_{k+2}}], \quad k = 1, 2$$

$$R = R_1 \oplus R_2$$

NOCH MEHR TRICK!

iss iss ipp i\$\$ ssi ssi ppi

Suffixe in  $S_C$  und  $R$  entsprechen sich!

$S_i$  entspricht  $[t_{i,i+1,i+2}][t_{i+3,i+4,i+5}] \cdots$

## 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  $S_C$

Laufzeit?  $O(n)$

# Linearzeitalgorithmus

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

## Konkret: Schritt 3

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

# Beispiel Merging

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

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

mississippi

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

# Aufbau

- Aufbau in  $O(n)$  Zeit und Platz
- Mehrere Algorithmen mit untersch. Eigenschaften bzgl. Platz/Zeit je nach Eingabe
- External Memory Algorithmen verfügbar
- Komprimierte Suffix Arrays
- Aufbauzeit kann über mehrere Suchen amortisiert werden
- Suche?

# Suche in Suffix Arrays

- **Matching:** Vorkommen von Muster p in String T
- Vorkommen ist Präfix eines Suffix
- Suffixsortierung → Binäre Suche

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

mississippi

SA	L	8	5	2	1	M	9	7	4	6	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					

sip < ssi

sip > i

sip > pi

Suche: sip

SA	11	8	5	2	1	L	9	7	M	6	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						

sip < sis

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				p	i	i
		i	i	s				p		p
			p	i				i		p
			p	p						i
			i	p						
				i						



1 Vorkommen

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

# Beschleunigung

- Auslassen bekannter Präfixe

		<b>L</b>		<b>R</b>							
		↓		↓							
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	.
			.	.	.				.	.	.

$\text{lcp}(S_i, S_j)$  – longest common prefix

# Beschleunigung

- $l_{\text{match}} = \min(\text{lcp}(p, S_{\text{SA}[L]}), \text{lcp}(p, S_{\text{SA}[R]}))$

- $\forall K = L, \dots, R:$

$$p[1..l_{\text{match}}] = T[\text{SA}[K] .. \text{SA}[K] + l_{\text{match}} - 1]$$

→ Nicht vergleichen

Laufzeitverbesserung:

- Praktisch: Klar
- Theoretisch: Worst-Case?           $\min = 0$

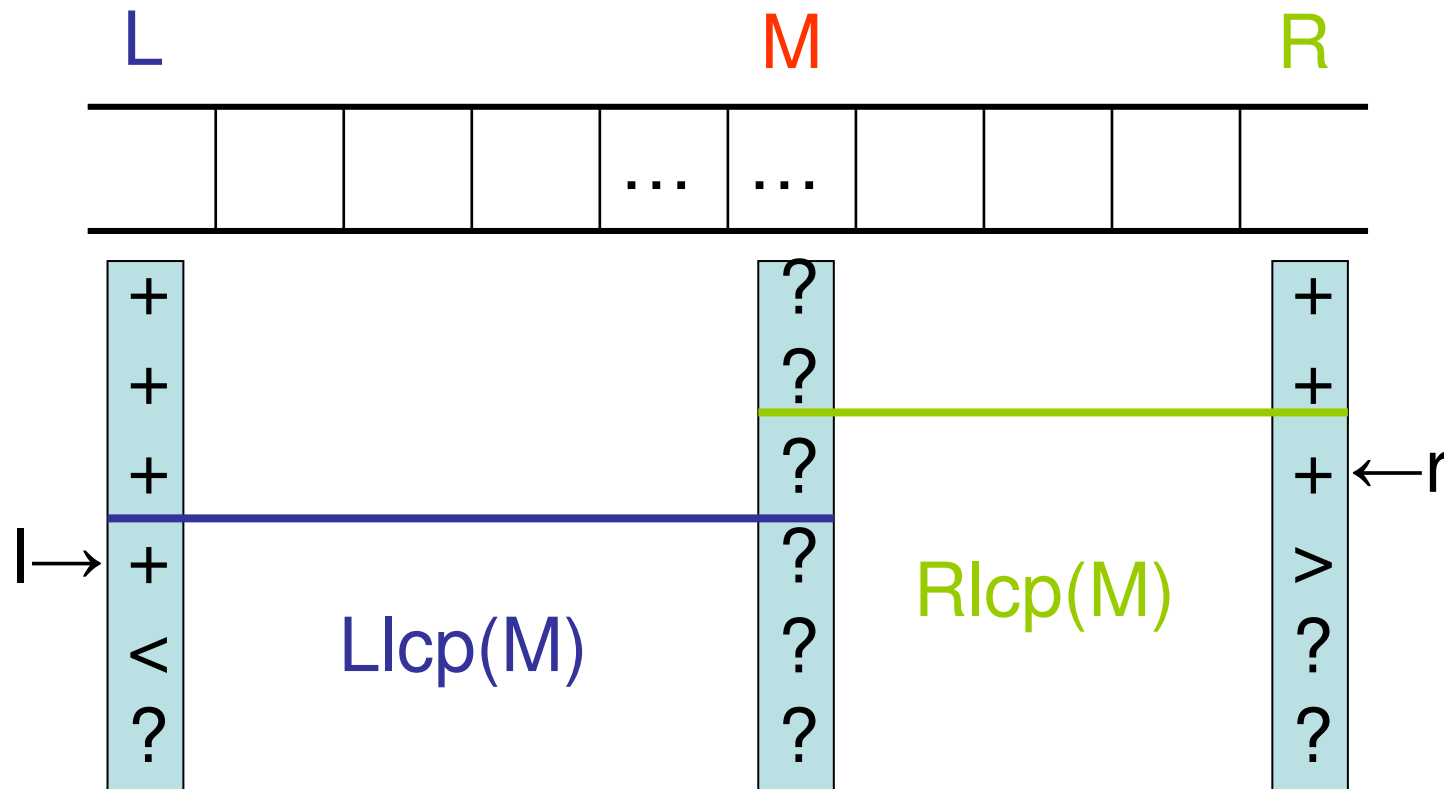
→ keine Verbesserung der asympt. Worst-Case Laufzeit

Suche

## Praxis: Icp-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

# Beschleunigung II

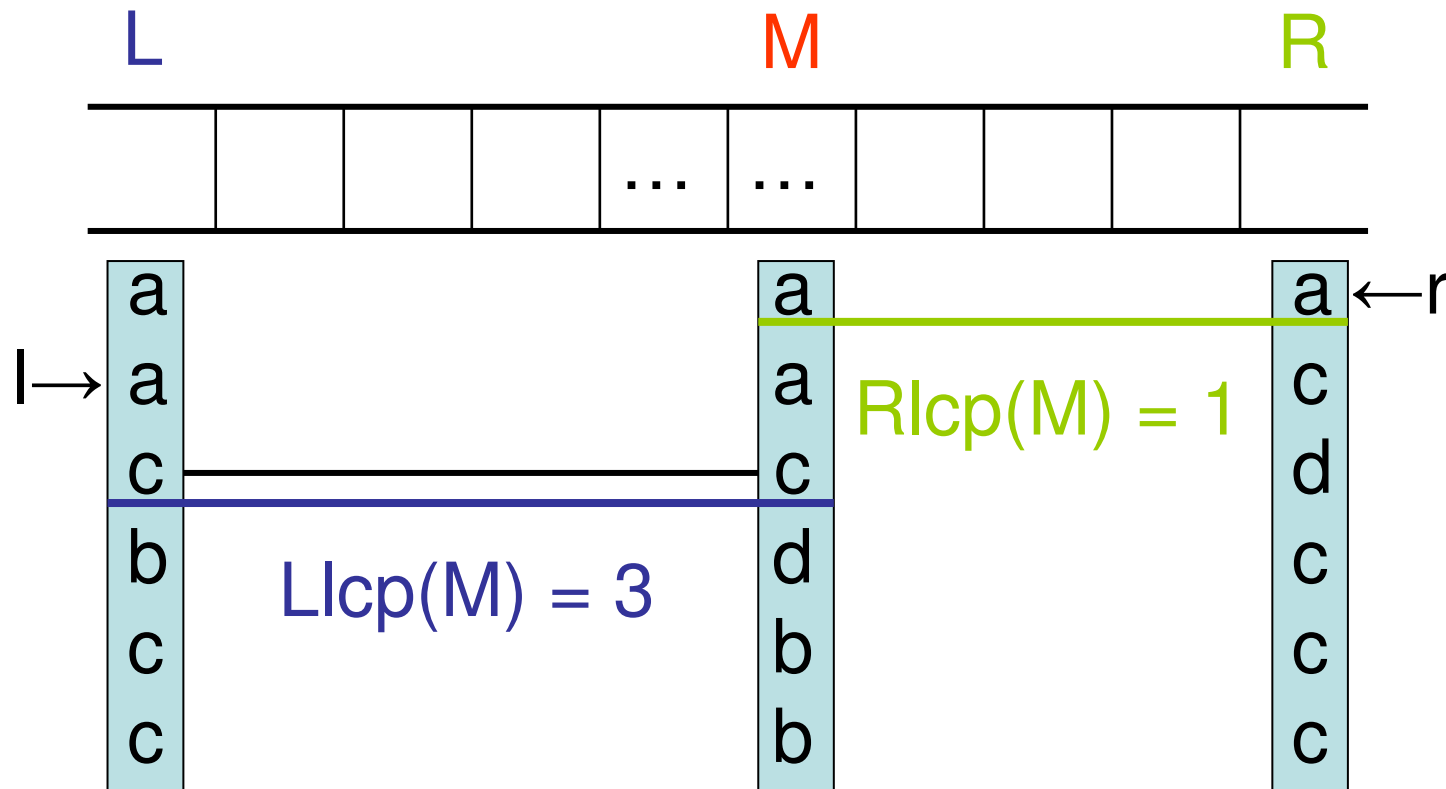


$lcp(p, S_{SA[M]})?$

Suche links oder rechts von  $M$ ?

Suche

# Beschleunigung II

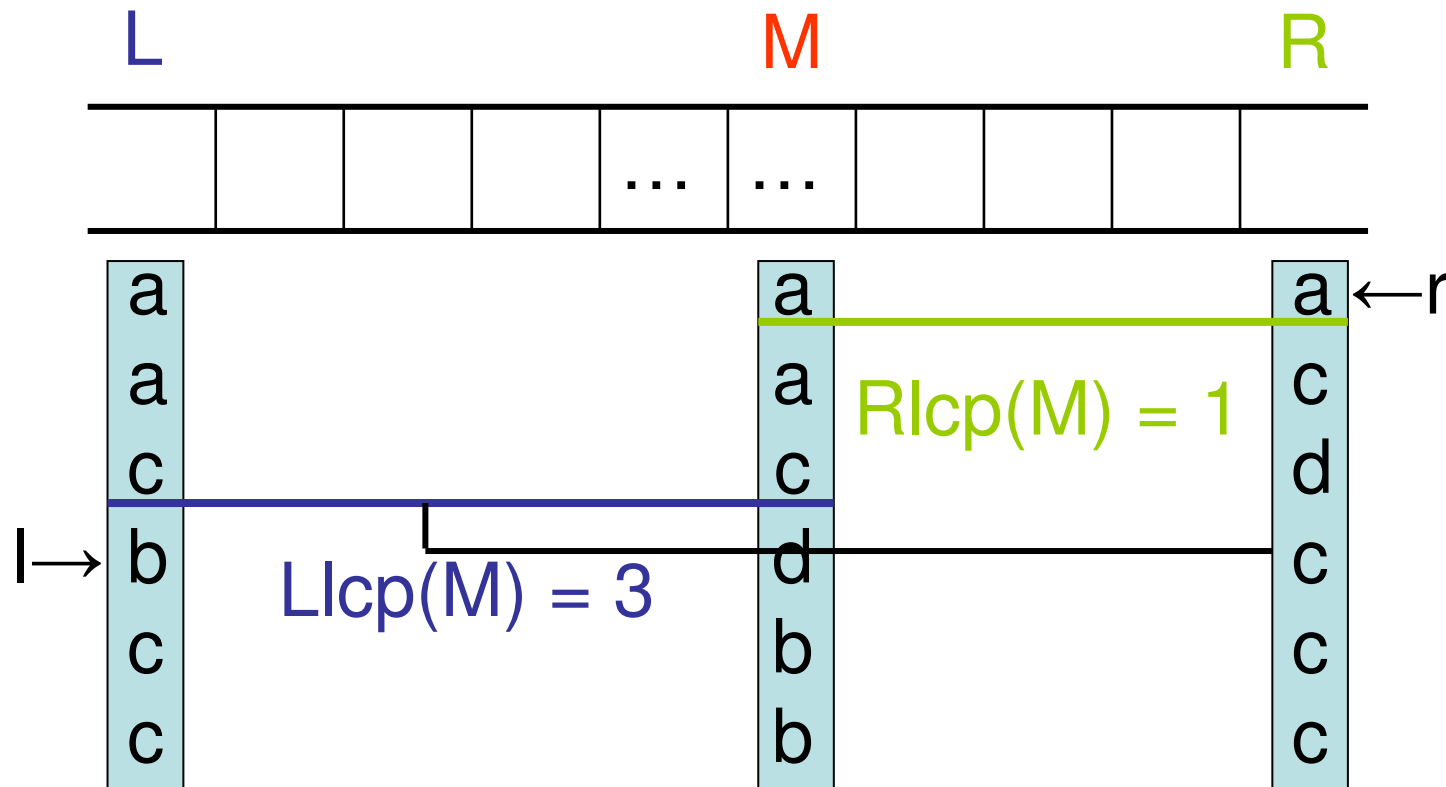


Muster P: aadccc

$Llcp > l$ : Lösung rechts, l bleibt

Suche

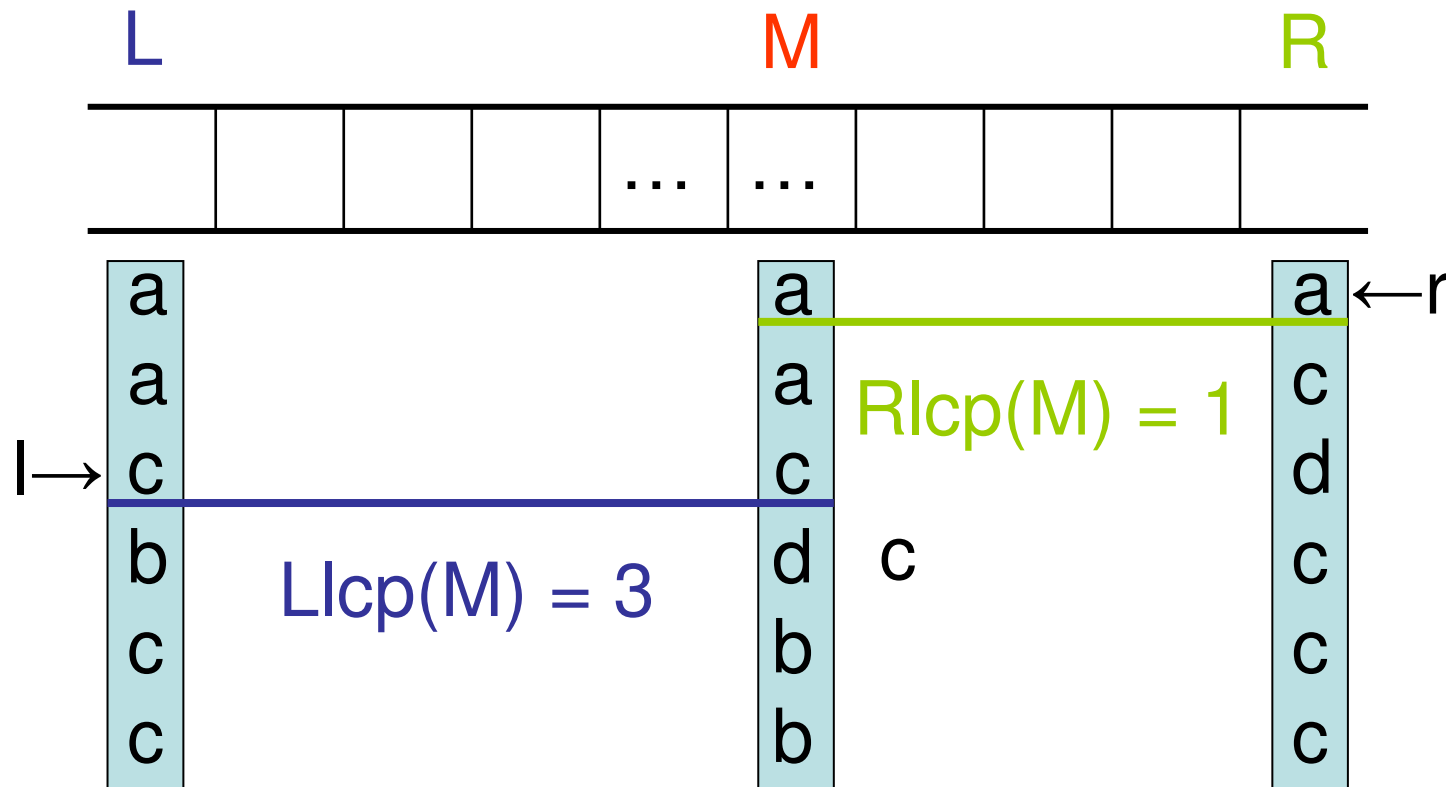
# Beschleunigung II



Muster P: aacbdd

$Llcp < l$ : Lösung links,  $r = Llcp(M)$

# Beschleunigung II



Muster P: aaccdd

$Llcp = l$ : Überspringe  $l$  Zeichen an M, teste weiter

# Beschleunigung II

Test für  $\max(Llcp, Rlcp) \rightarrow$  maximale Schrittweite

Statische Vorberechnung der Llcp/Rlcp-Werte

Laufzeit?

Nicht erfolgreicher Vergleich  $\rightarrow$  Intervallhalbierung

Erfolgreicher Vergleich  $\rightarrow$  Suchindex + 1

Laufzeit:  $O(m + \log n)$