

Suffix Arrays

Eine Datenstruktur für
Stringalgorithmen

11. Vorlesung

08/10.Mai 2007



Universität Dortmund, FB Informatik, LS11

Übersicht

- Kurze Wiederholung...
- Suche: Beschleunigung 2. Teil
- Anwendung / Anpassung
 - Burrows-Wheeler Transformation
 - Enhanced Suffix Arrays

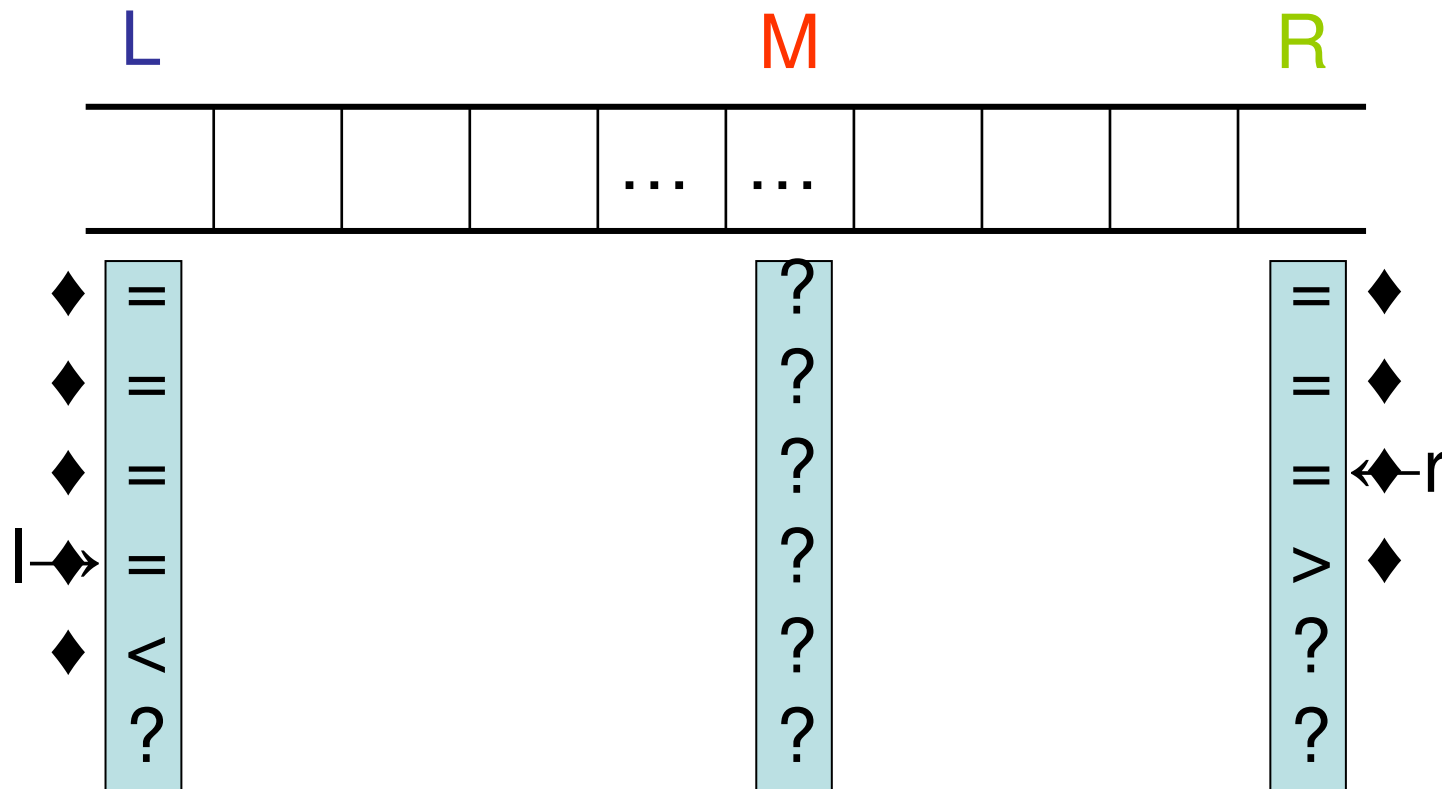
Suffix Arrays

- Datenstruktur für Stringalgorithmen (Textsuche)
 - Textindex durch Suffixsortierung
-
- Aufbauzeit $O(n)$ bei Stringlänge n
 - Platzbedarf $O(n)$, $4n$ in Praxis
 - Unabhängig von Textalphabet
 - Suchzeit naiv $O(m(\log n + k))$, Trick zur Beschleunigung durch Auslassen gemeinsamer Präfixe (*lcp*)

Aufbau

- Aufbaualgorithmus: DC3
- Dreiteilung der Suffixe durch modulo Operation
- Zwei Sortiermengen S_C (Sample) und S_0
- „Geschicktes“ Sortieren mit Rekursion und Ausnutzung der Suffixstruktur
- Einfaches Mergen (auch Suffixstruktur)
- Spezialfall von DC

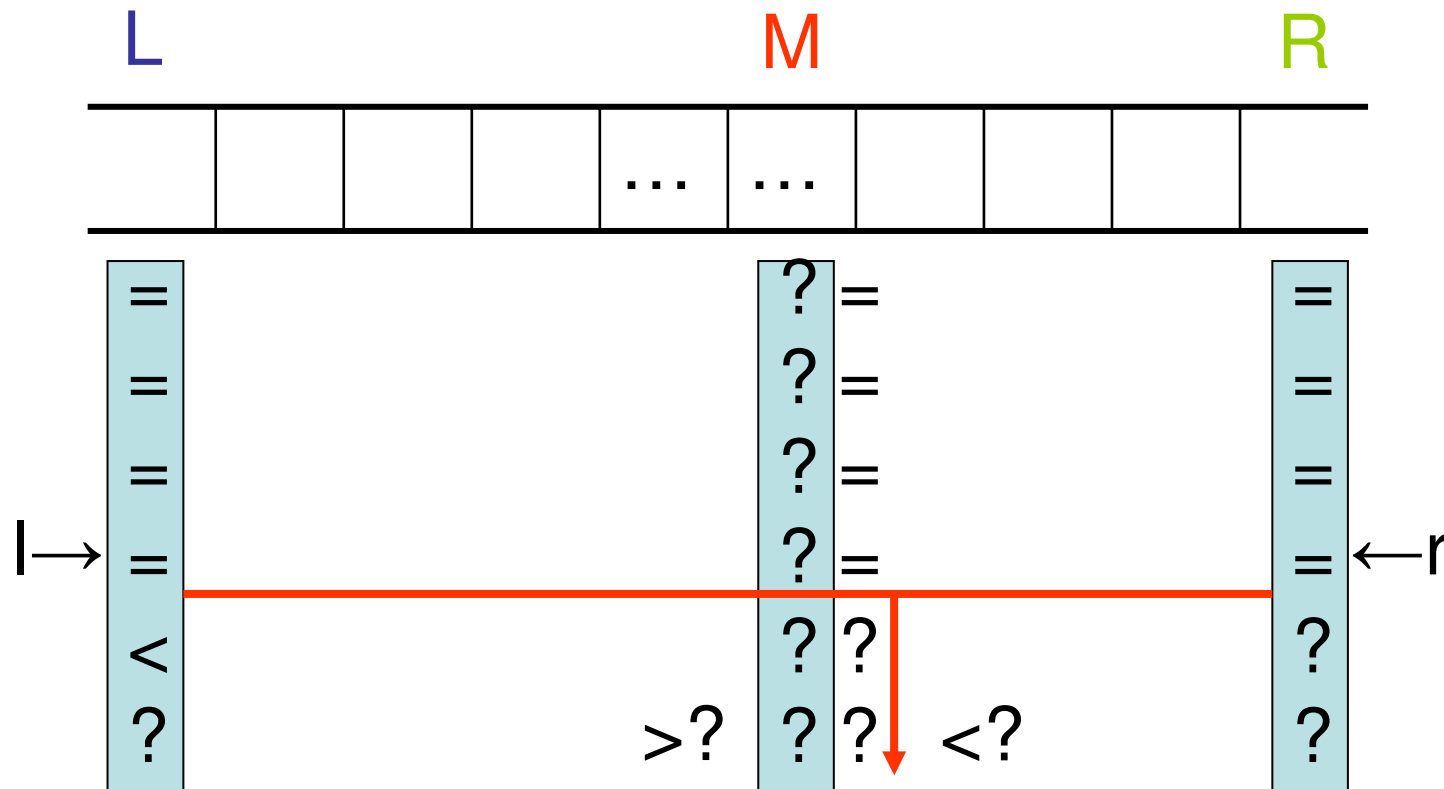
Beschleunigung II



Ab welcher Position vergleichen? $\text{lcp}(p, S_{\text{SA}[M]})$?

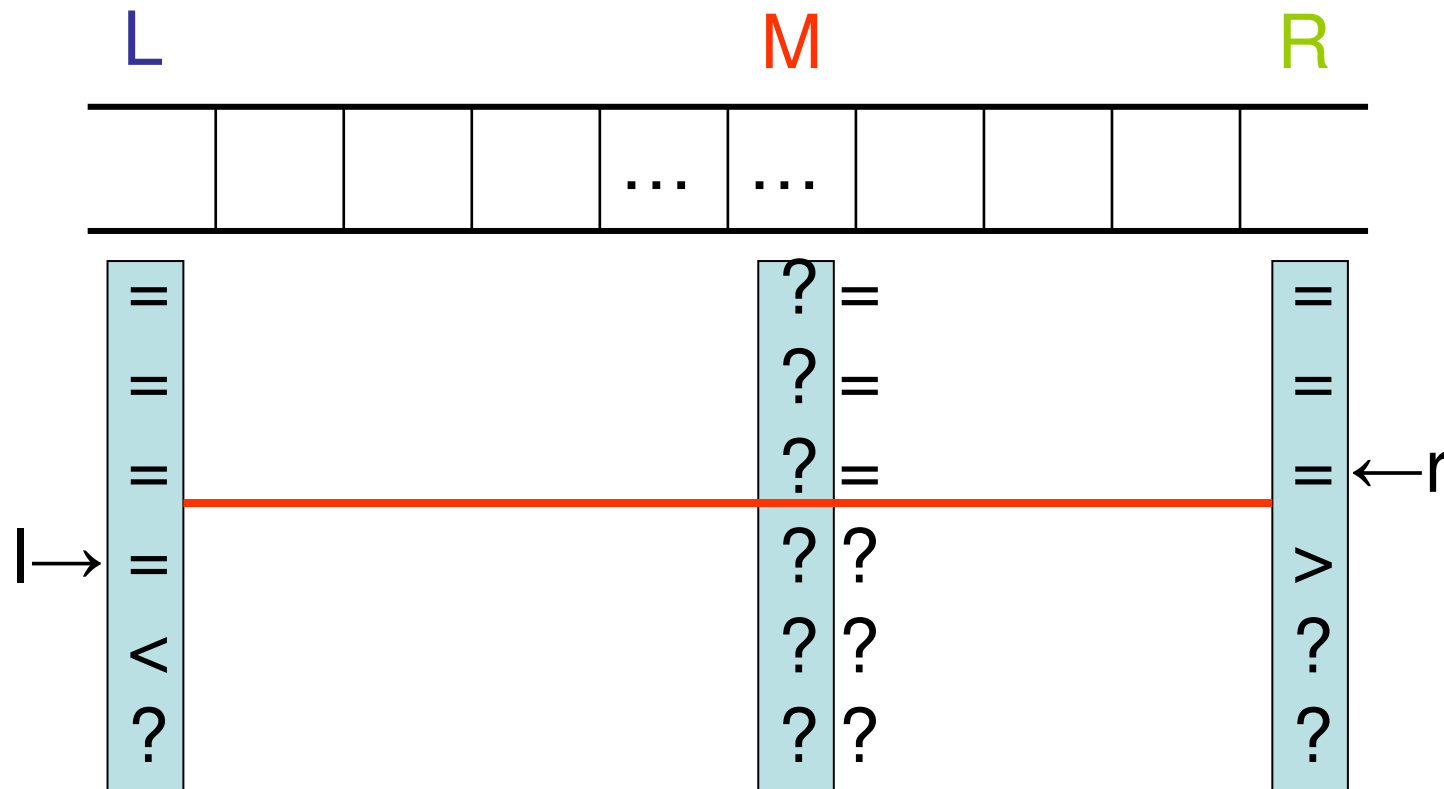
Suche links oder rechts von M fortsetzen?

Beschleunigung II



Fall 1: $l = r \Rightarrow$ Vergleiche ab Position $l+1$ bis Unterschied Links oder rechts von M fortsetzen, l/r anpassen

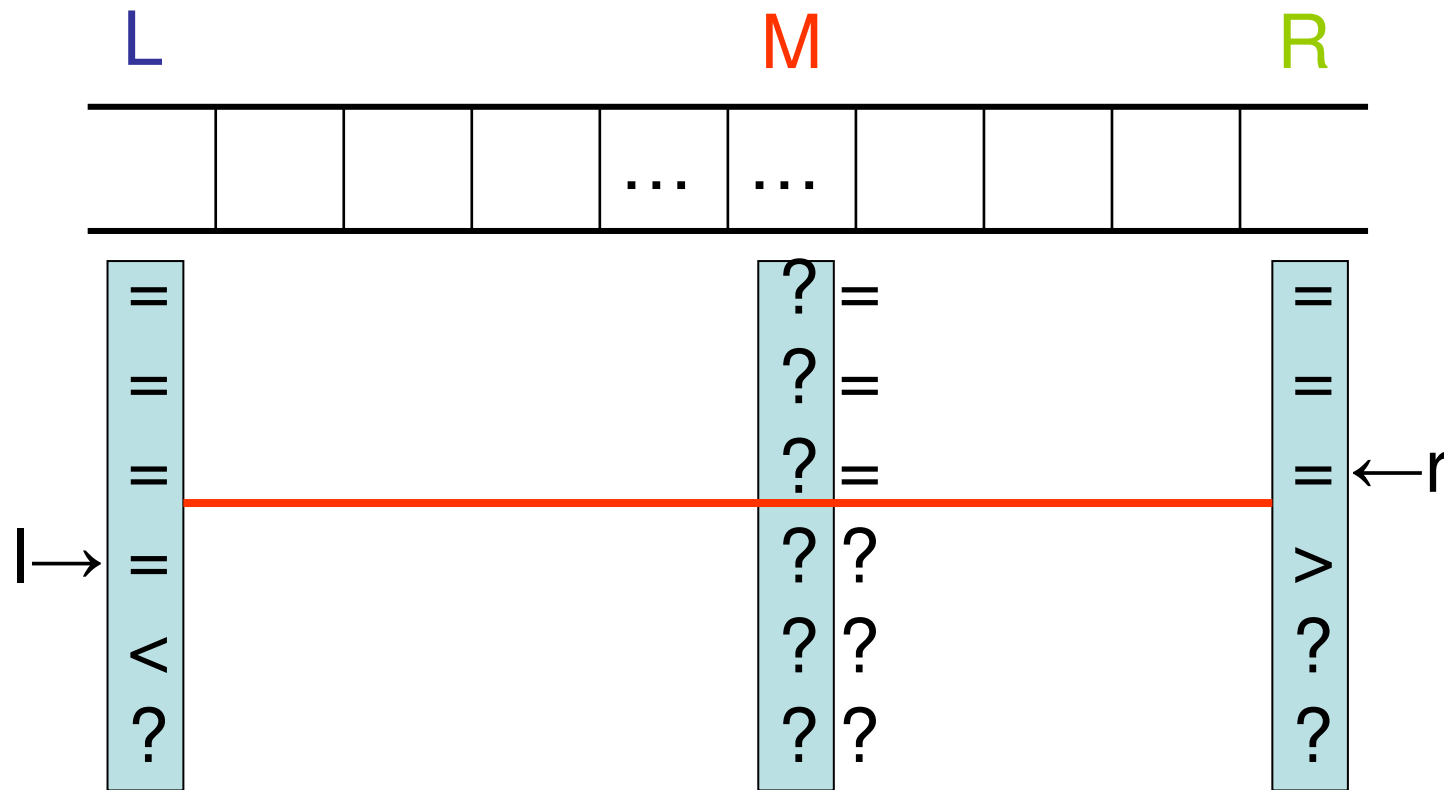
Beschleunigung II



Fall 2: $l > r$

Asymptotisch noch keine Verbesserung
Wie kriegen wir das hin?

Beschleunigung II



Fall 2: $l > r$

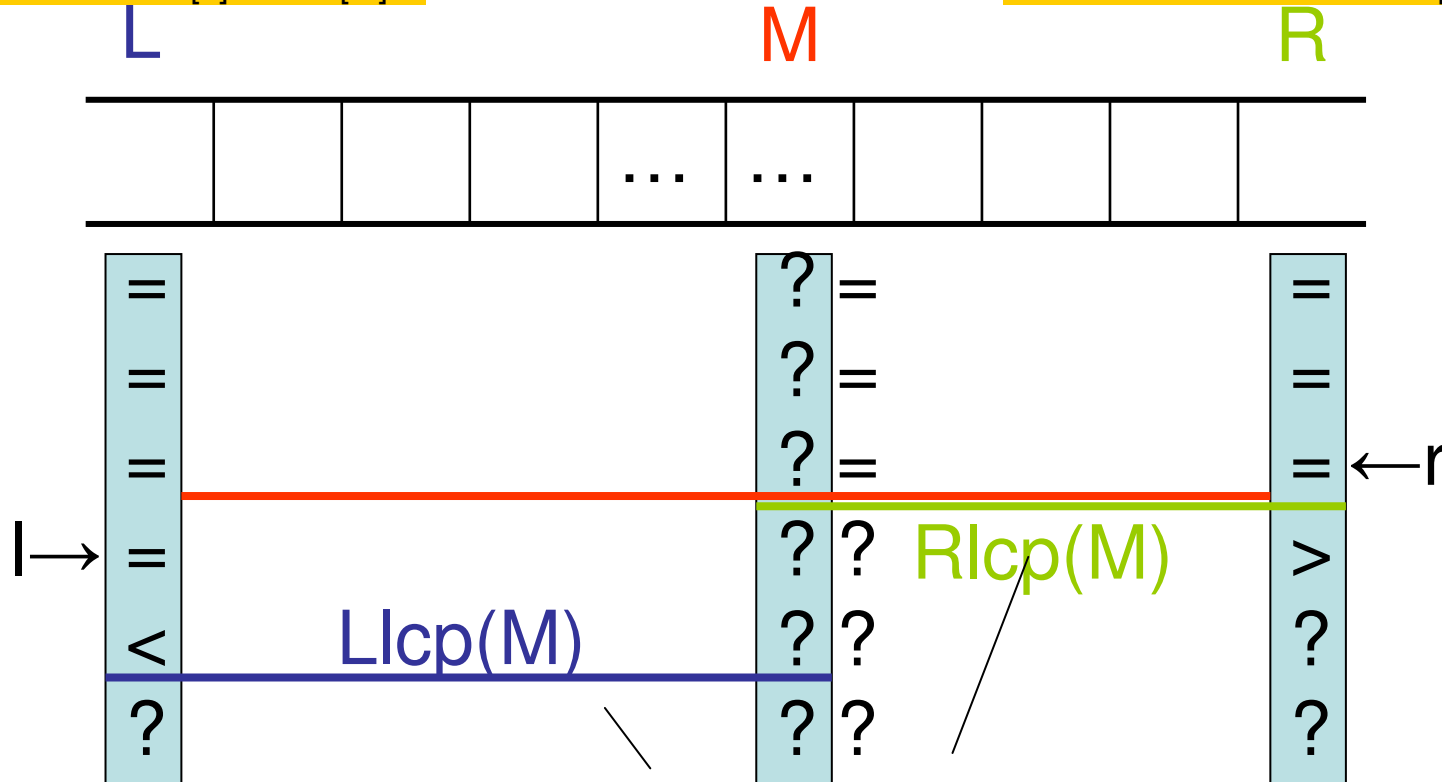
Wir nutzen Eigenschaft des Textes:
Übereinstimmung von M zu L,R

Suche

Beschleunigung II

$$Llcp(M) = lcp(S_{SA[L]}, S_{SA[M]})$$

$$Rlcp(M) = lcp(S_{SA[R]}, S_{SA[M]})$$



Übereinstimmung an M zu L und R

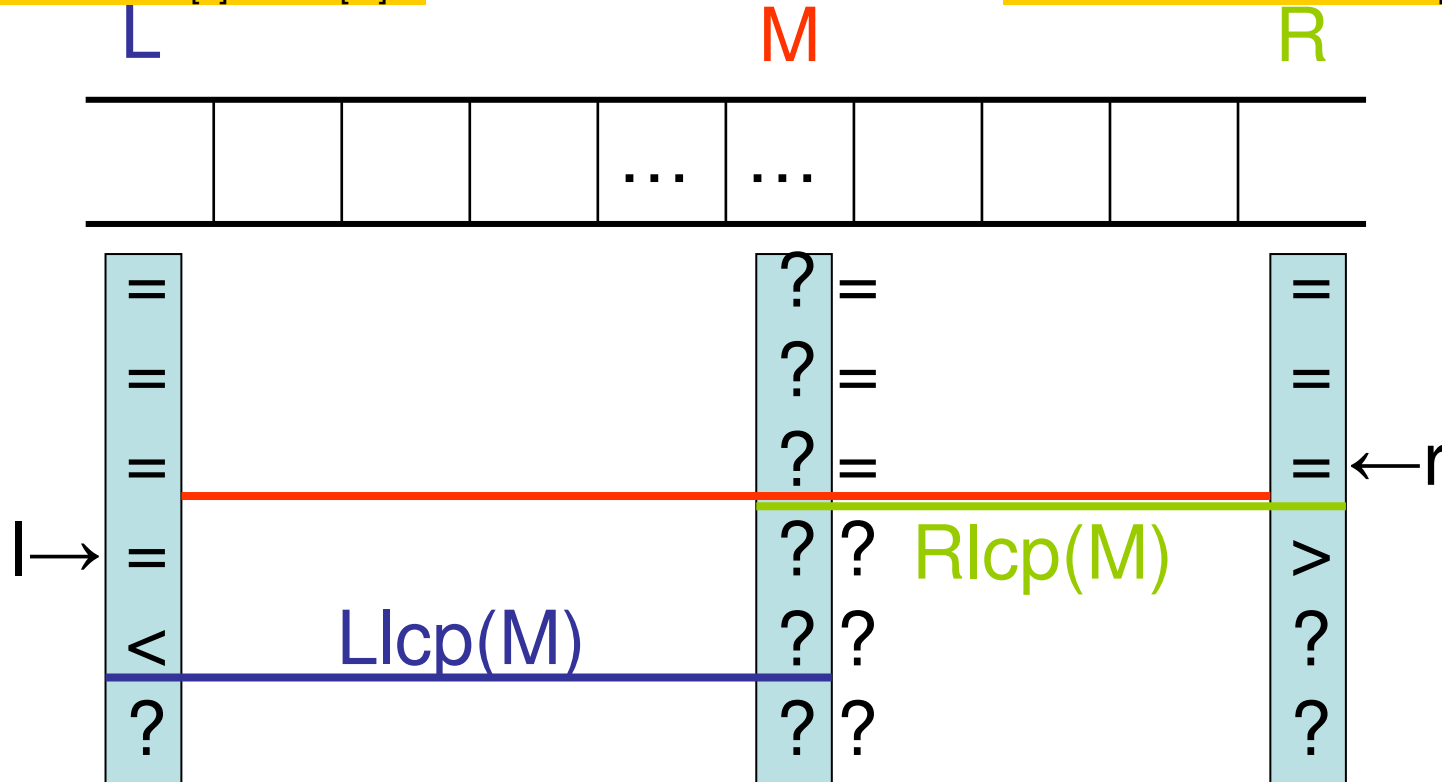
Fall 2: $l > r$

Suche

Beschleunigung II

$$Llcp(M) = lcp(S_{SA[L]}, S_{SA[M]})$$

$$Rlcp(M) = lcp(S_{SA[R]}, S_{SA[M]})$$



Fall 2: $l > r$ $Llcp$ und $Rlcp$ sind statisch berechenbar
und $L/Rlcp(M) \geq \min(l, r)$

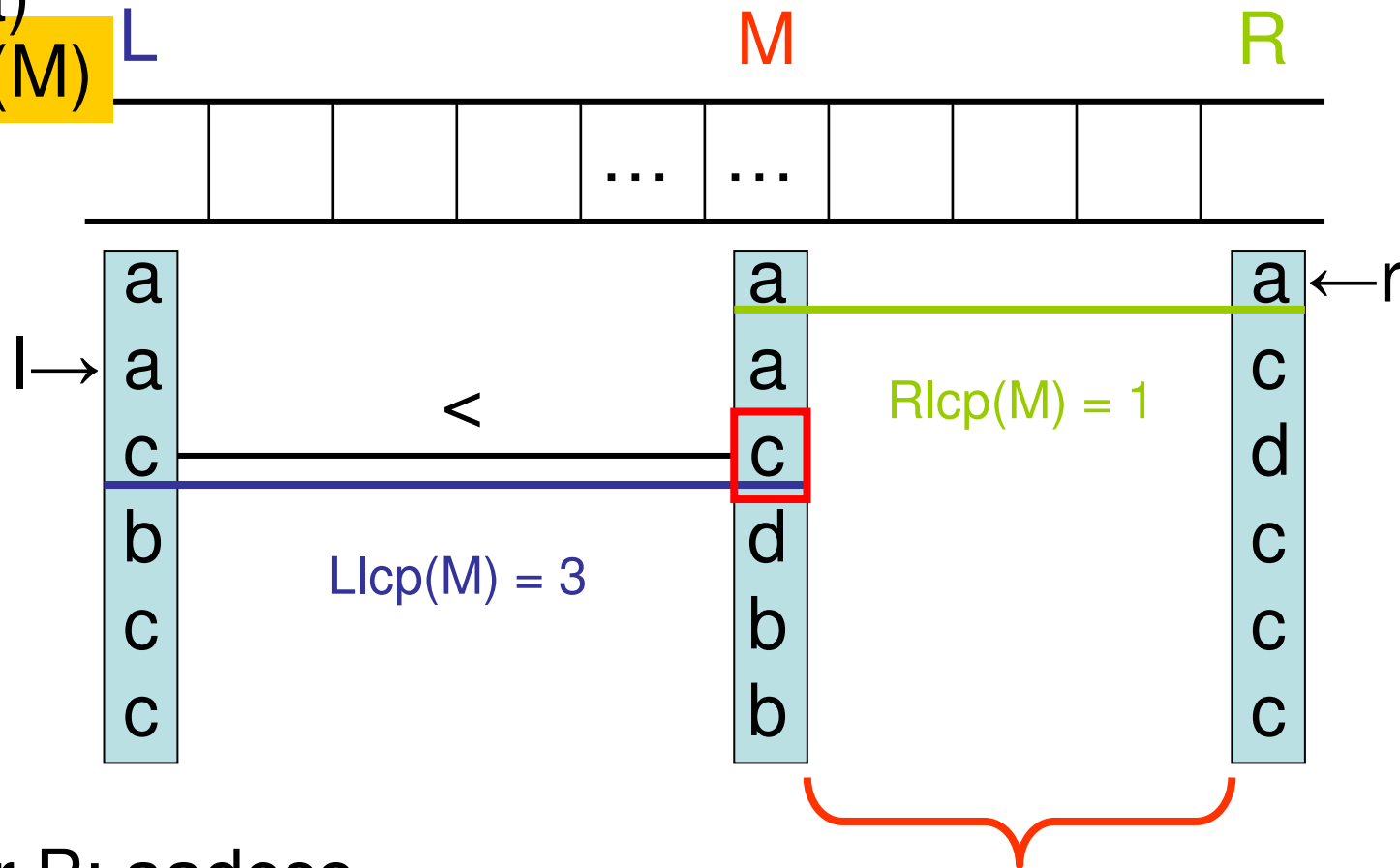
Wir unterscheiden drei Fälle: $l \{<=>\} Llcp(M)$

Suche

Beschleunigung II

Fall 2a)

$l < Llcp(M)$



Muster P: aadccc

Nächstes Zeichen an M kleiner

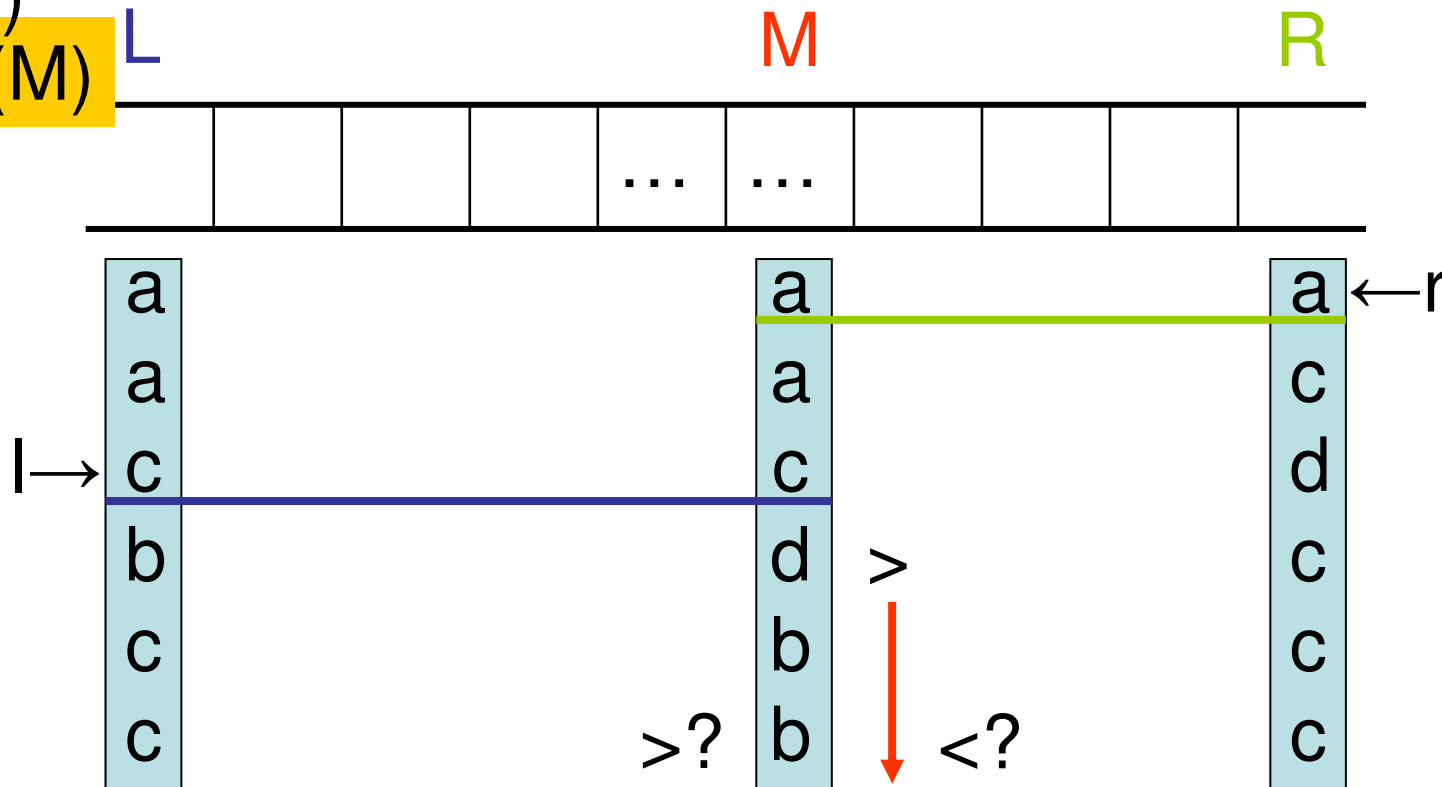
Lösung rechts, l bleibt

Suche

Beschleunigung II

Fall 2c)

$l = \text{Llcp}(M)$



Muster P: aaccdd

Überspringe l Zeichen, vergleiche ab $l+1$

Beschleunigung II

Fall 3 ($r < l$) symmetrisch

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

Statische Vorberechnung der Llcp/Rlcp-Werte
möglich (nächste Folie)

Gesamtlaufzeit der Suche?

Nicht erfolgreicher Vergleich \rightarrow Intervallhalbierung

Erfolgreicher Vergleich \rightarrow Suchindex + 1

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

Beschleunigung II

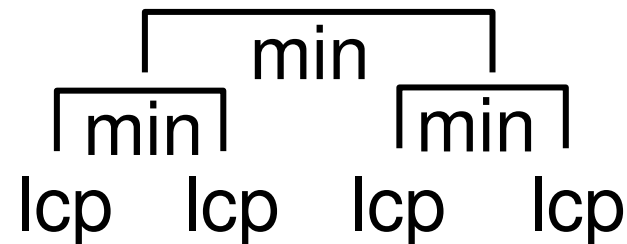
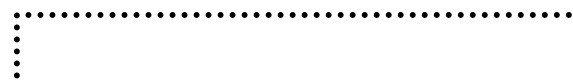
Berechnung von Lcp, Rlcp: Wieviele (L,M,R)?

n-2 verschiedene M, dabei L,R fix

Wir beginnen mit lcp benachbarter Suffixe

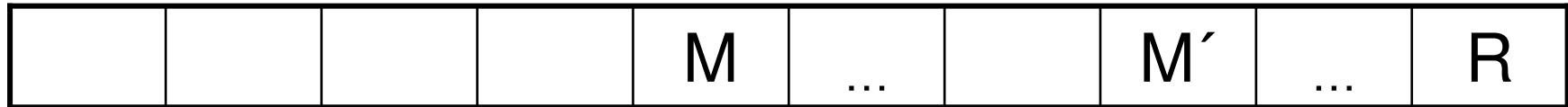
lcp hier $\text{lcp}(S_{SA[i]}, S_{SA[i+1]})$

$$\text{lcp}(S_{SA[i]}, S_{SA[j]}) = \min\{\text{lcp}(S_{SA[k-1]}, S_{SA[k]}) : k \in [i+1, j]\}$$



				L_2	M_3	M_2	M_1	R_2	
--	--	--	--	-------	-------	-------	-------	-------	--

Beschleunigung II



Bottom-up traversal der lcp-Werte für Nachbarn

- $Rlcp(M) = \min\{Llcp(M'), Rlcp(M')\}$ wobei M' schon eine Ebene tiefer berechnet
- Llcp analog

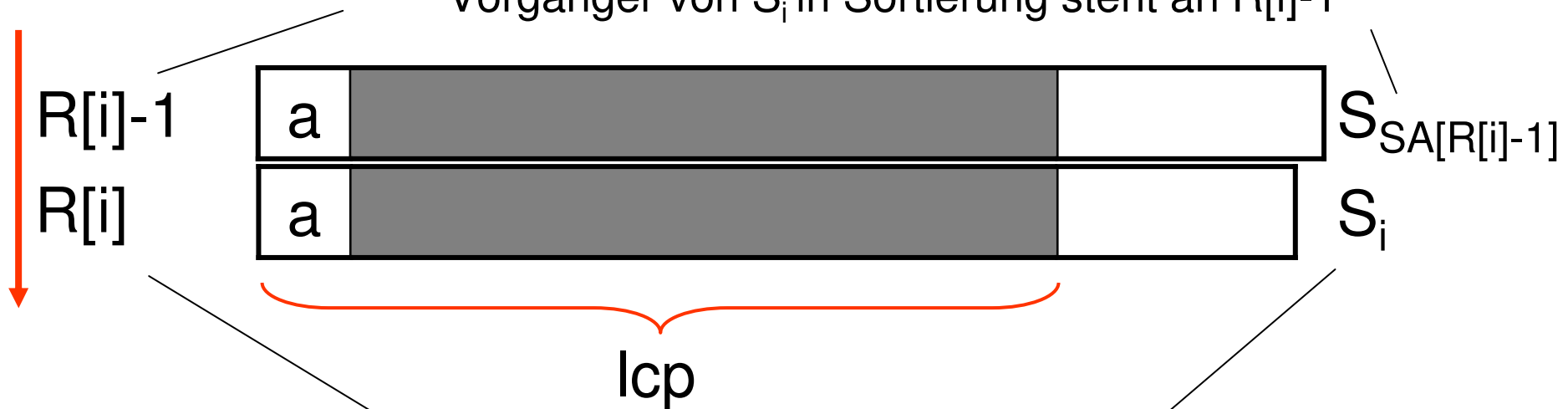
→ Vorbereitung in Linearzeit wenn $lcp(S_{SA[i-1]}, S_{SA[i]})$ bekannt

LCP-Tabelle für Nachbarn

Berechnung der speziellen lcp-Tabelle L

$$L[k] = \text{lcp}(S_{SA[k-1]}, S_{SA[k]}) \text{ für } k \in [1..n]$$

Vorgänger von S_i in Sortierung steht an $R[i]-1$



Suffix S_i steht an Stelle $R[i]$ in SA

Wir laufen alle Suffixe der Reihe nach ab

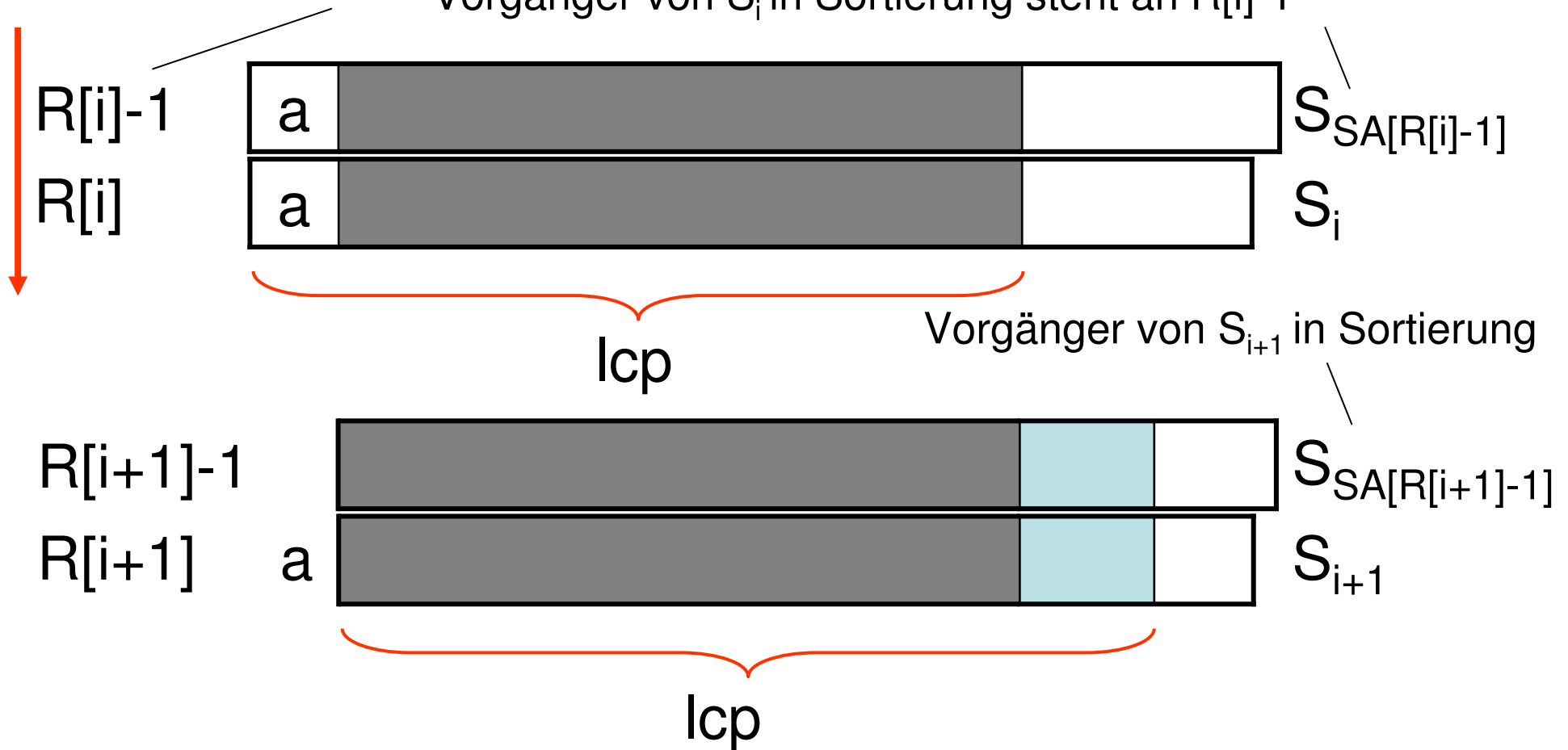
Nächstes Suffix ist ein Zeichen kürzer

LCP-Tabelle für Nachbarn

Berechnung der speziellen lcp-Tabelle L

$$L[k] = \text{lcp}(S_{SA[k-1]}, S_{SA[k]}) \text{ für } k \in [1..n]$$

Vorgänger von S_i in Sortierung steht an $R[i]-1$

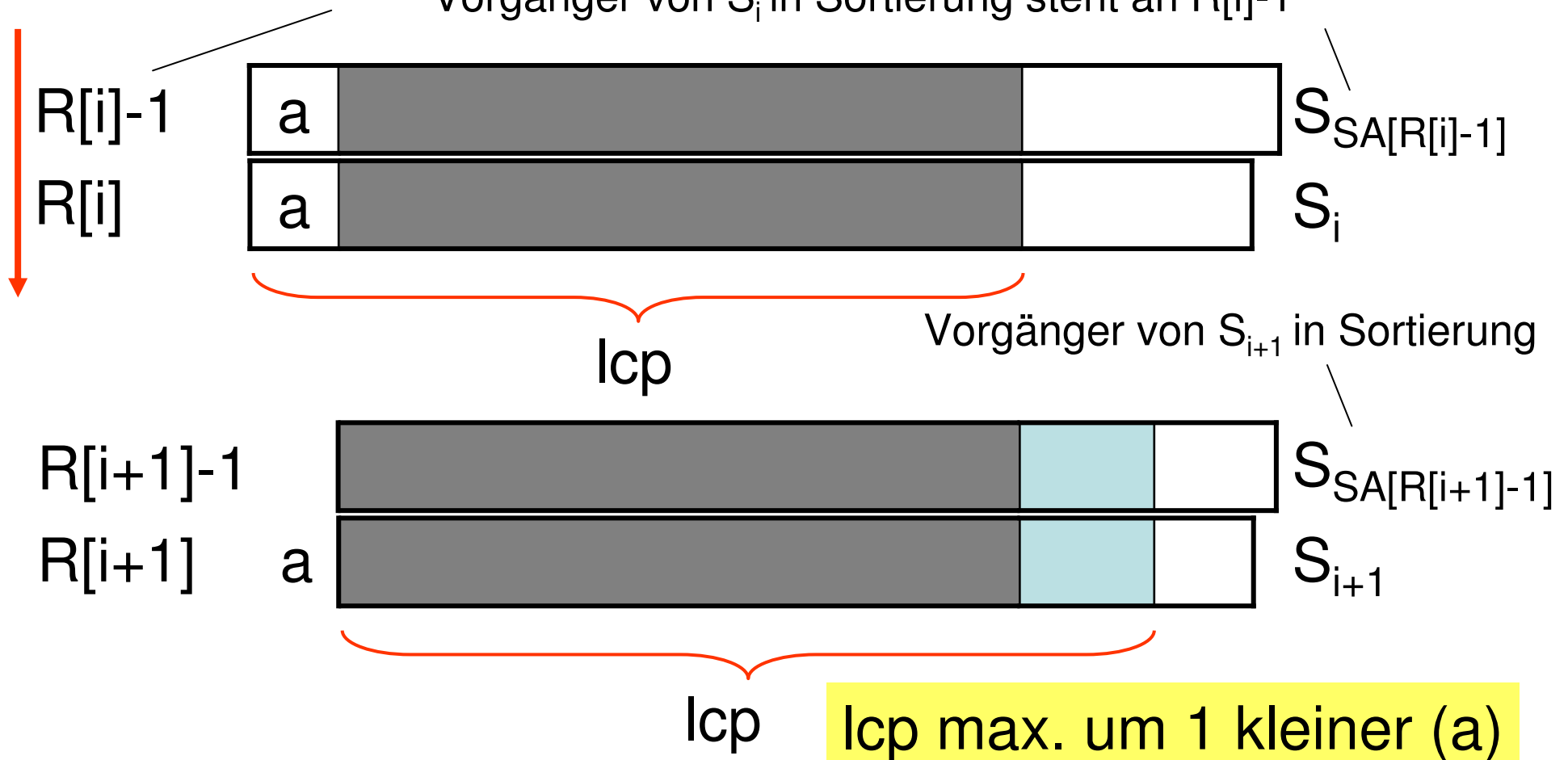


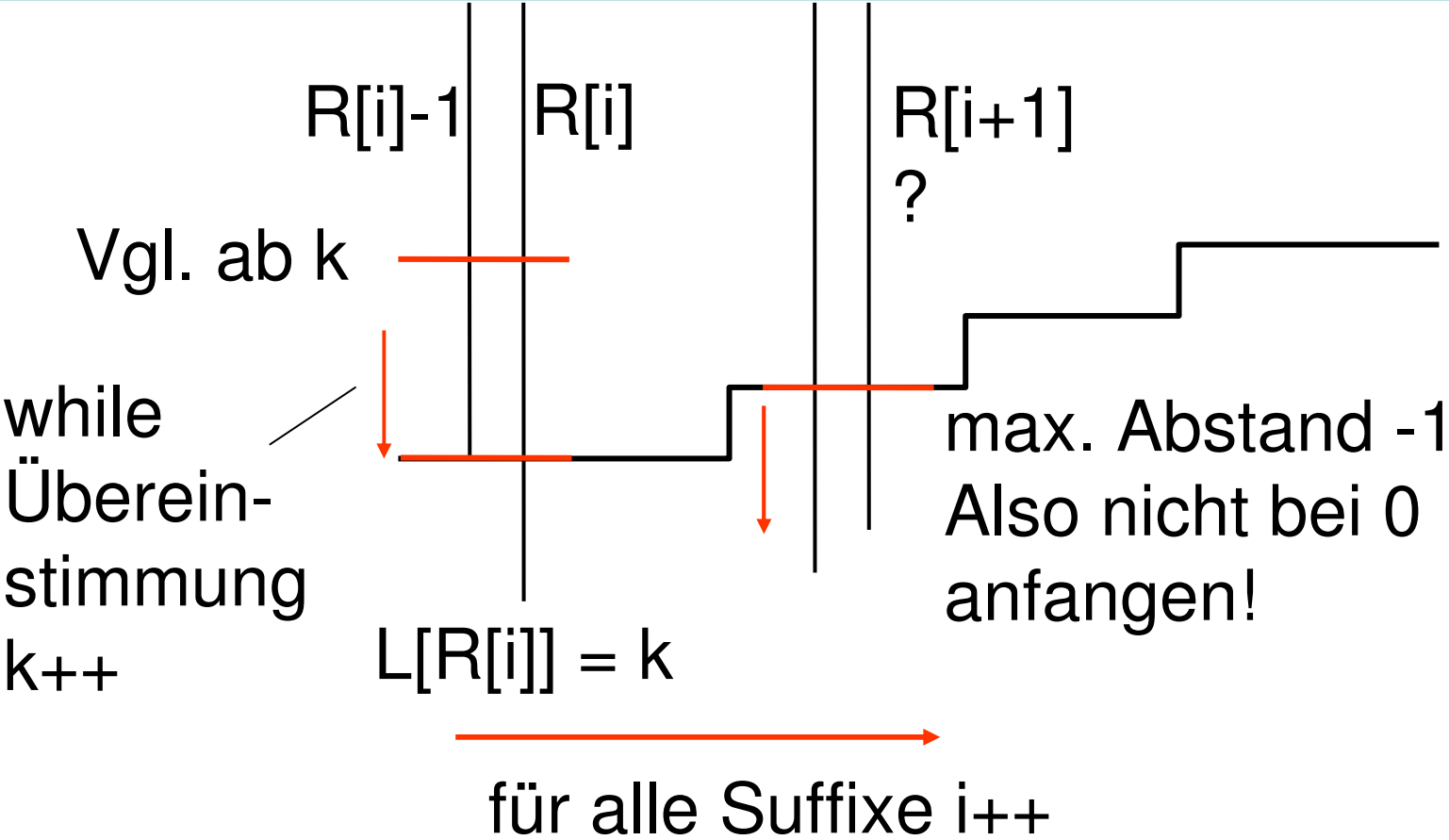
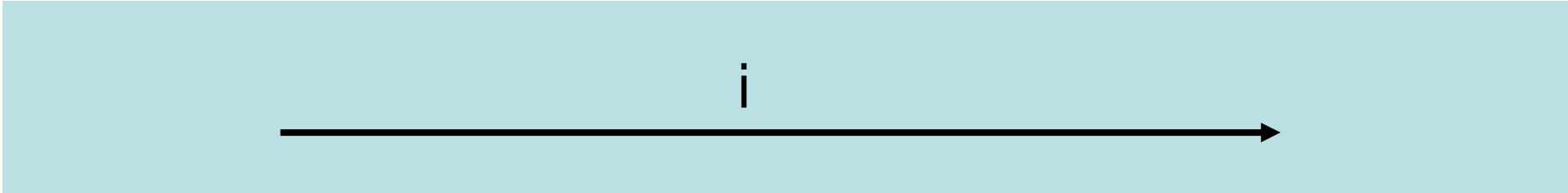
LCP-Tabelle für Nachbarn

Berechnung der speziellen lcp-Tabelle L

$$L[k] = \text{lcp}(S_{SA[k-1]}, S_{SA[k]}) \text{ für } k \in [1..n]$$

Vorgänger von S_i in Sortierung steht an $R[i]-1$





Max. $2n$ Schritte

LCP-Tabelle

Vorgängerbeziehung:

$$\text{lcp}(SA_{R[i+1]}, SA_{R[i+1] - 1}) \geq \text{lcp}(SA_{R[i]}, SA_{R[i] - 1}) - 1$$

k = 0;

for i = 1 .. n

 if R[i] > 0

 j = SA[R[i] - 1]; //S_j lex. direkt vor S_i

 while (s_{i+k} = s_{j+k}) k++; //Übereinstimmung

 L[R[i]] = k;

 k = max(0, k-1); //wg. max. Abstand 1

Beschleunigung II

Fazit: Preprocessing mit Berechnung von

- lcp-Wert Array L
- Llcp- und Rlcp-Werten

ist in Linearzeit möglich!!

→ Suche in $O(m + \log n)$ mit linearem Aufbau

Hauptpunkte

- Suffix Array ist lexikographisch geordneter Array der Suffixe eines Strings
- Aufbau in Linearzeit
- Platzeffiziente Datenstruktur ($\sim 4n\text{Byte}$)
- Anwendung: Stringvergleiche, Suche
- Suche in Zeit $O(m + \log n)$
- Amortisierung des Aufbaus über viele Suchen (Indexierung)

Anwendung / Anpassung

- Kompression von Daten
 - Burrows-Wheeler
 - (Lempel-Ziv(-Welsh))
- Enhanced Suffix Arrays
 - Baumtraversierung auf Suffix Array abbilden

Verlustfreie Datenkompression

- Sequentielle Verfahren
 - Referenzierend: Lempel-Ziv, *Substituierende* Kompression: Ersetzt Repeats durch Referenz auf voriges Vorkommen („Wörterbuch“)
 - Statistisch: Huffman (Kodierer /Datenmodell), arithmetische Kodierung, PPM
- Blockorientierte Verfahren
 - Kontext-orientiert: Burrows-Wheeler

Statisch: Wörterbuch/Codetabelle benötigt

Adaptiv: Während (De)Kompression aufgebaut

Kompression Burrows-Wheeler

- 1994 veröffentlicht („A block-sorting lossless data compression algorithm“), seither Vielzahl von Verbesserungen
- Hohe Kompressionsraten bei hoher Geschwindigkeit möglich (blockabh.), verwendet z.B. in bzip2
- „Relativ“ geringer Platzverbrauch
- Blockorientiert, je größer desto besser
⇒ nicht für Audio-/Videostreams
- Geschwindigkeit ähnlich wörterbuchbasierte, Kompressionsrate ähnlich statistische Verfahren

Kompression Burrows-Wheeler (Schema)

- Mehrere Stufen hintereinandergeschaltet
- Daten werden blockweise verarbeitet und weitergereicht (Größe im Mb Bereich)



Kompression Burrows-Wheeler (Schema)

- Mehrere Stufen hintereinandergeschaltet
- Daten werden blockweise verarbeitet und weitergereicht (Größe im Mb Bereich)



Burrows-Wheeler-Transformation:

- Permutation als Preprocessing
- Ziel ist Sortierung nach nachfolgendem „Kontext“
- Führt zu langen Folgen gleicher Zeichen

Kompression Burrows-Wheeler (Schema)

- Mehrere Stufen hintereinandergeschaltet
- Daten werden blockweise verarbeitet und weitergereicht (Größe im Mb Bereich)



Global-Structure-Transformation:

- Sehr viele verschiedene Variationen
- Original (Einfach): *Move-To-Front coding* erzeugt Folge von kleinen Ints (Indizes in Liste des Alphabets)
- Auch *Run-length Encoding*: Folgen gleicher Zeichen zusammenfassen

Kompression Burrows-Wheeler (Schema)

- Mehrere Stufen hintereinandergeschaltet
- Daten werden blockweise verarbeitet und weitergereicht (Größe im Mb Bereich)



Entropiekodierung:

- Daten sind noch nicht komprimiert (ausser RLE)
- Überführe Indexfolgen in möglichst „kurze“ Bitfolgen
- Benutze dazu z.B. Huffmancode oder arithmetische Kodierung

Kompression Burrows-Wheeler (Schema)

- Mehrere Stufen hintereinandergeschaltet
- Daten werden blockweise verarbeitet und weitergereicht (Größe im Mb Bereich)



Herzstück ist BWT

Bottleneck ist BWT!

Burrows-Wheeler Transformation

- Reine Permutation der Eingabezeichen
- Daten mit ähnlichem „Kontext“ rücken zusammen
- Ergebnis ist idR besser komprimierbar
- Reversibel

Vorwärtstransformation

Beispielblock „BANANA“

1. Bilde alle möglichen Rotationen der Blockdaten

B	A	N	A	N	A
A	N	A	N	A	B
N	A	N	A	B	A
A	N	A	B	A	N
N	A	B	A	N	A
A	B	A	N	A	N

Vorwärtstransformation

2. Sortiere die Zeilen

B	A	N	A	N	A
A	N	A	N	A	B
N	A	N	A	B	A
A	N	A	B	A	N
N	A	B	A	N	A
A	B	A	N	A	N

i →

A	B	A	N	A	N
A	N	A	B	A	N
A	N	A	N	A	B
B	A	N	A	N	A
N	A	B	A	N	A
N	A	N	A	B	A

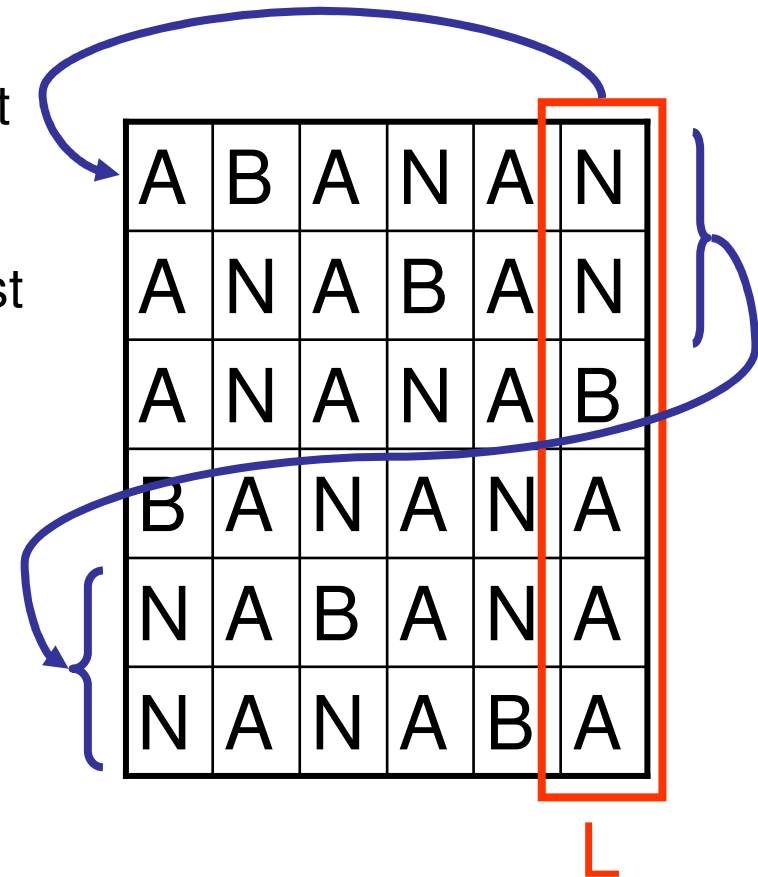
L

Ausgabe: Letzte Spalte L und Index i des Originals

Vorwärtstransformation

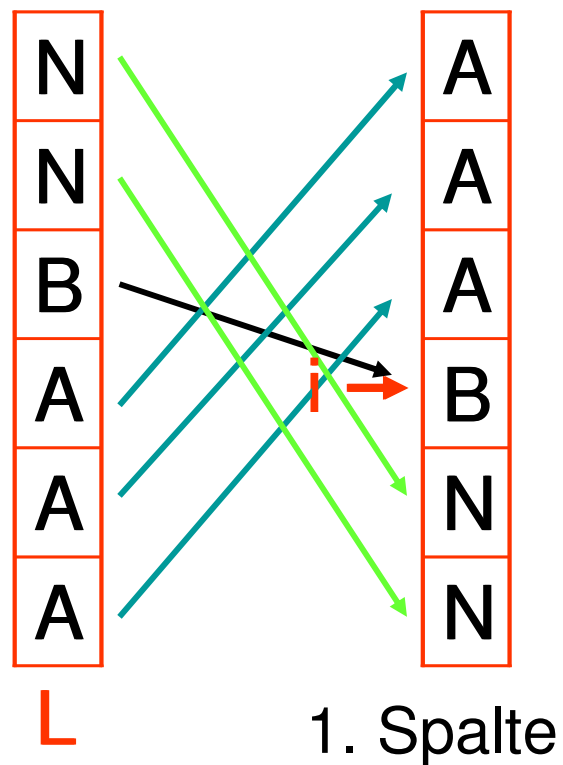
Eigenschaften:

- L nach nachfolgendem „Kontext“ sortiert
- Erste Spalte sortiert
- Bei gleichem Zeichen in letzter Spalte ist die Reihenfolge der Zeilen dieselbe wie die der Zeilen, in denen genau dieses Zeichen an erster Stelle kommt
- Gruppierung in letzter Spalte: Sei z.B. „Teil“ häufiger Teilstring → Rotationen mit „eil“ am Anfang werden zusammen sortiert → „T“ lokal gehäuft
- Lokale Häufung führt zu guter Komprimierbarkeit



Rücktransformation

- Jede Spalte erhält alle Zeichen, also auch L
- ⇒ Erste Spalte durch Sortierung der Zeichen
- ⇒ Erstes Zeichen des Blocks steht an i

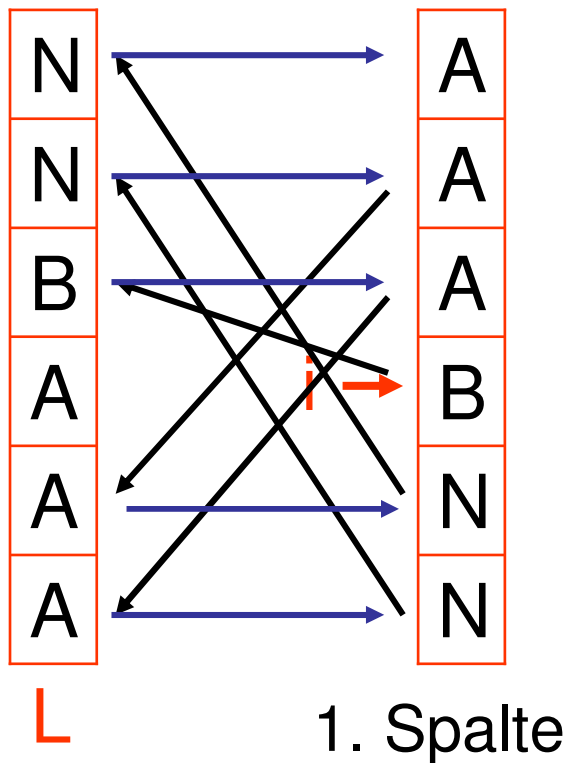


Text: B

Zeilen mit gleichem Zeichen entsprechen sich

Rücktransformation

- Jede Spalte erhält alle Zeichen, also auch L
- ⇒ Erste Spalte durch Sortierung der Zeichen
- ⇒ Erstes Zeichen des Blocks steht an i



Text: BANANA

Starte an i und laufe Entsprechungen ab,
nächstes Zeichen steht in 1. Spalte

Zeilenreihenfolge heißt auch
„Transformationsvektor“

Burrows-Wheeler-Transformation

Was hat das denn mit Suffix-Arrays zu tun?

B	A	N	A	N	A
A	N	A	N	A	
N	A	N	A		
A	N	A			
N	A				
A					

A					
A	N	A			
A	N	A	N	A	
B	A	N	A	N	A
N	A				
N	A	N	A		

?

(Achtung: Im Allgemeinen Sentinel \$ anhängen)

Burrows-Wheeler-Transformation

BWT kann mit Suffix Array berechnet werden!

$$\text{BWT}[i] = \begin{cases} s[\text{SA}[i]-1] \text{ falls } \text{SA}[i] \neq 0 \\ s[n-1] \text{ bzw. } \$ \text{ falls } \text{SA}[i] = 0 \end{cases}$$

BANANA
0 1 2 3 4 5

N	N	B	A	A	A
5	3	1	0	4	2

Anwendungen der BWT

- Kompression
- Repeatsuche
- Aufbau kompakter Suffix Arrays (!)

Suffix Arrays

Wir haben:

- Aufbau in Linearzeit
- Suche in Linearzeit
- Einfache Anwendung

Jetzt:

- Erweiterung für praktische Anwendung
- Viele gute Algorithmen (z.B. Repeatsuche) baumbasiert, z.B. mit bottom-up, top-down traversal

Enhanced Suffix Arrays

- Idee: Baumverfahren wie bottom-up-traversal in Suffix Arrays abbilden
- Speichern lcp-Tabelle L zum SA
- Immer noch Platzvorteil!

Konzept des lcp-Intervalls:

Maximales Intervall im SA, in dem alle lcp-Werte zwischen Nachbarn $\geq m$ sind „m-Intervall“

lcp-Intervall

Formal:

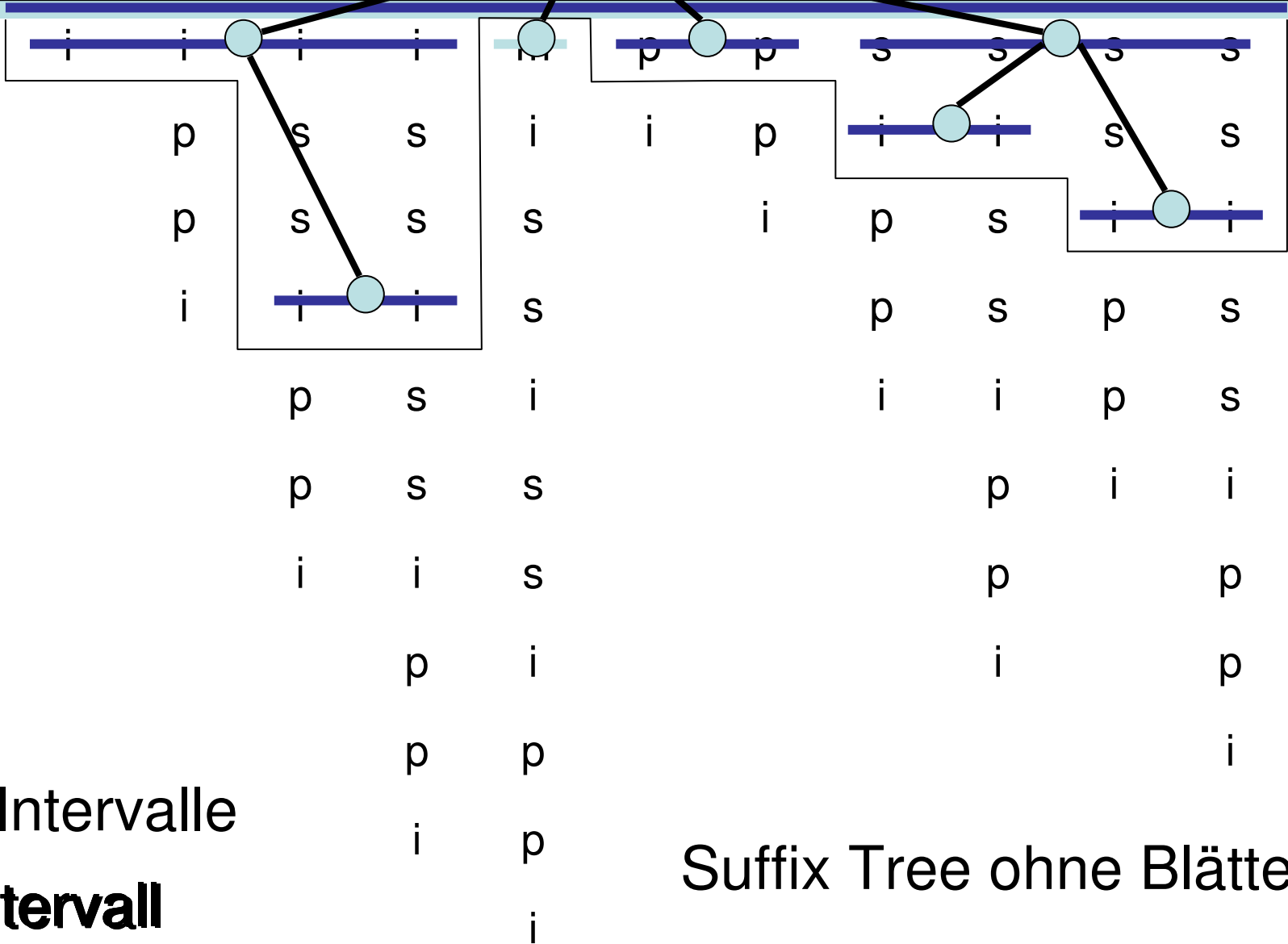
Intervall $[i..j]$ mit $0 \leq i < j \leq n$ ist ein

lcp-Intervall mit lcp-Wert m

wenn

1. $L[i] < m$
2. $L[k] \geq m$ für alle $i+1 \leq k \leq j$
3. $L[k] = m$ für mind. ein solches k
4. $L[j+1] < m$

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



lcp-Intervalle

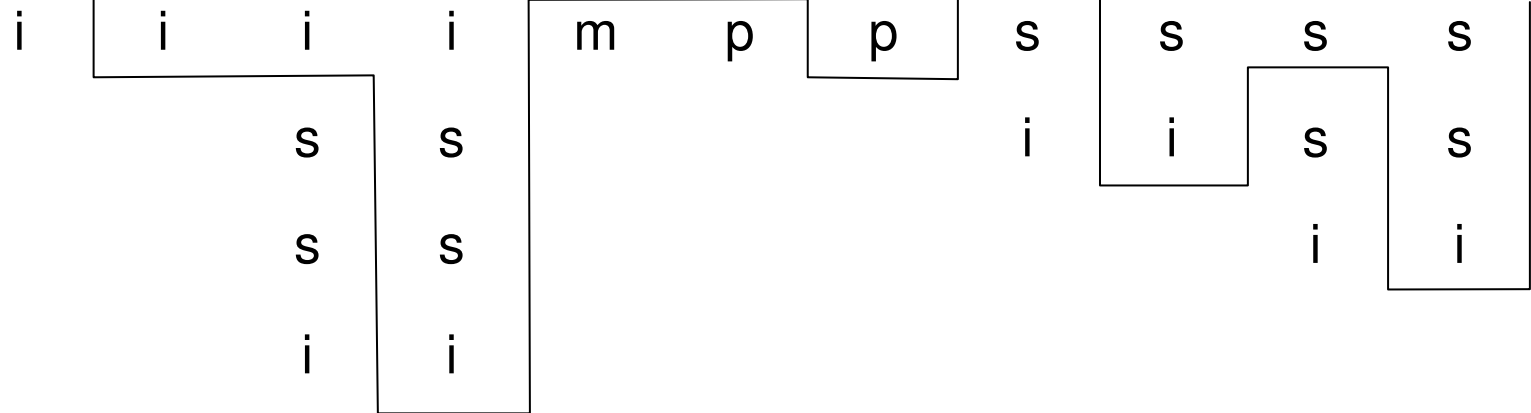
0-Intervall

Suffix Tree ohne Blätter!

Bausteine

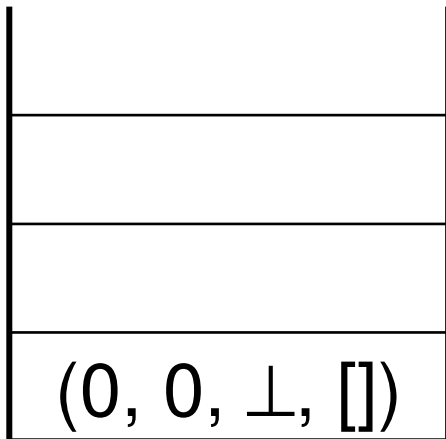
- Suffix Array SA
- Lcp Tabelle L
- Lcp Intervall Baum IB (konzeptuell)
- Traversal des IB ohne Aufbau

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



L

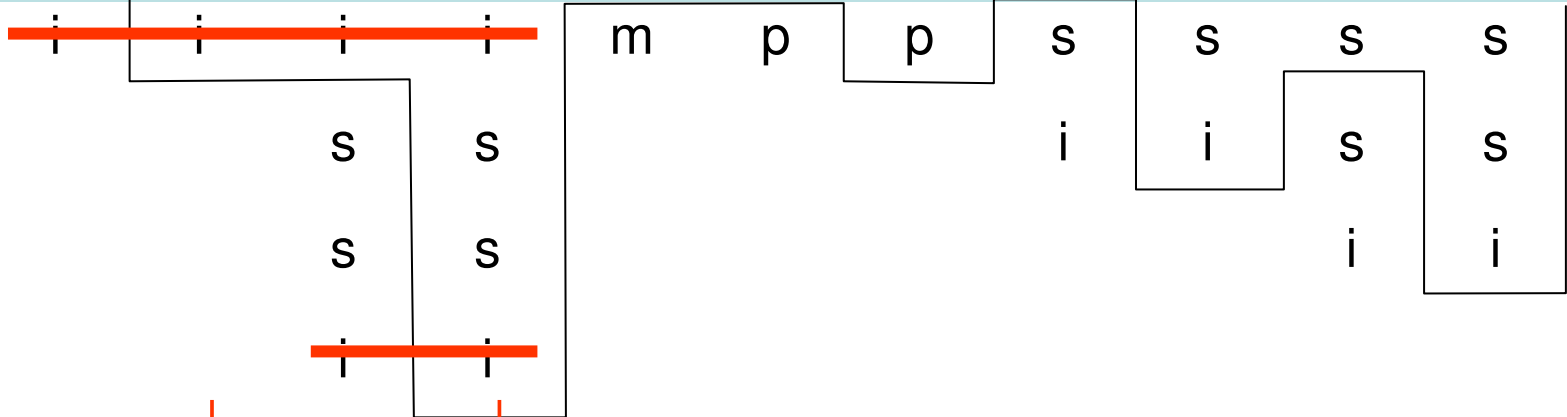
0	1	1	4	0	0	1	0	2	1	3
---	---	---	---	---	---	---	---	---	---	---



Stack statt Baum für lcp-Intervalle

Tupel (lcp, lb, rb, children)

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



L

0	1	1	4	0	0	1	0	2	1	3
---	---	---	---	---	---	---	---	---	---	---

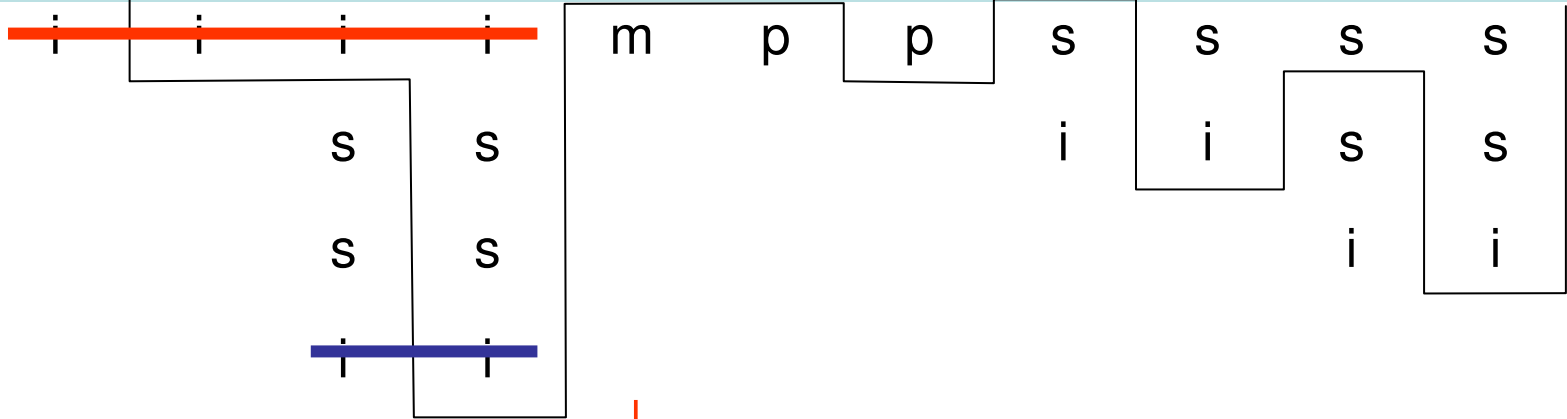
Wenn lcp steigt, Tupel auf Stack legen

(4, 2, ⊥, [])
(1, 0, ⊥, [])
(0, 0, ⊥, [])

push(L[i], i-1, ⊥, [])

Tupel (lcp, lb, rb, children)

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



L

0	1	1	4	0	0	1	0	2	1	3
---	---	---	---	---	---	---	---	---	---	---

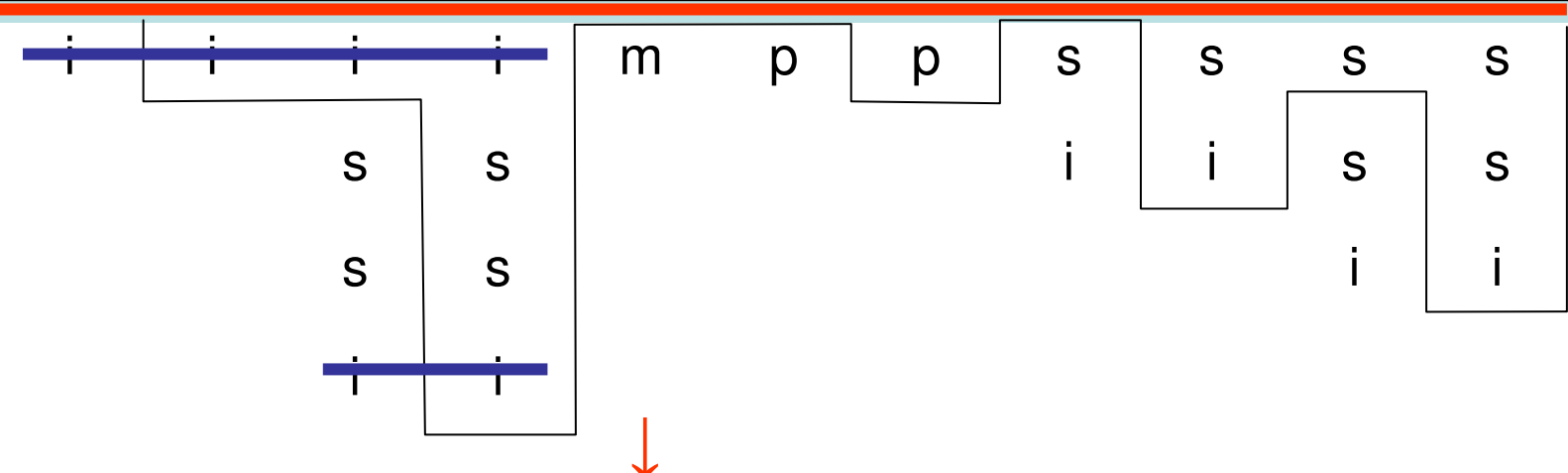
Wenn lcp fällt: Intervallende, pop

(1, 0, ⊥, li1)
(0, 0, ⊥, [])

li1 = (4, 2, 3, [])

Tupel (lcp, lb, rb, children)

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

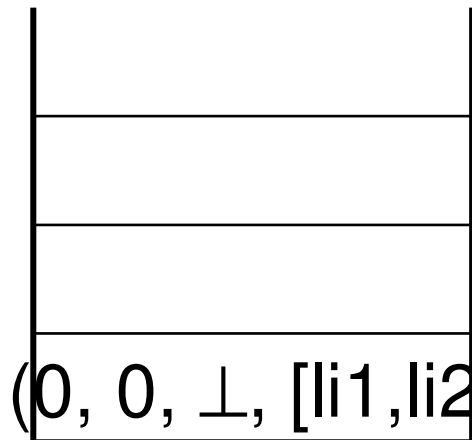


L

0	1	1	4	0	0	1	0	2	1	3
---	---	---	---	---	---	---	---	---	---	---

Immer noch $L[i] < \text{top.lcp}$!
Also noch ein pop

$$Li2 = (1, 0, 3, li1)$$



$(0, 0, \perp, [li1, li2])$ Tupel (lcp, lb, rb, children)

LCP Tabelle

- Problem: Verdopplung des Speicherplatzbedarfs
- Idee: In Praxis $lcp \ll \maxInt$

L

17
94
255
23
255

L+

(2, 328)
(4, 455)

$n + k$ bytes

Zugriff in $\log(|L+|)$
für random access,
sonst konstant



Ende!