Suffix Arrays – Eine Datenstruktur für Stringalgorithmen

Karsten Klein

Vorlesung

Algorithmen und Datenstrukturen

WS 08/09 – 13.November 2008

TU Dortmund, Fakultät Informatik, LS11 Algorithm Engineering

Übersicht

- Kurze Wiederholung…
- Suche in Suffix Arrays
- Anwendung / Anpassung
  - Burrows-Wheeler Transformation
  - Enhanced Suffix Arrays

Suffix Arrays

- Datenstruktur für Stringalgorithmen (Textsuche)
- Textindex durch Suffixsortierung
- Integerarray von Suffixstartpositionen

- Aufbauzeit *O(n)* bei Stringlänge *n*
- Platzbedarf *O(n), 4n* in Praxis
- Unabhängig von Textalphabet

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: Tradeoff zwischen Zeit und Platz möglich

Stringsuche in Suffix Arrays

Suche

Matching: Vorkommen von Muster p in String s

Brute Force: O((n-m+1)m)

Knuth-Morris-Pratt: θ(n+m)

scsdcdc

klsadjfsdfscsdcdcefwefefefeefefe

Wir wissen:

- Vorkommen ist Präfix eines Suffix
- Sortierte Suffixe in Suffix Array vorhanden

Suche in Suffix Arrays

Suffixsortierung → Binäre Suche

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

Beispiel: Suche „sip“ in „mississippi“

SA	L	7	4	1	0	M	8	6	3	5	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
sip < ssi				p	i				i		p
sip > i				p	p						i
sip > pi				i	p						
					i						

Suche: sip

SA	10	7	4	1	0	L	8	6	M	5	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	10	7	4	1	0	L	8	M	R	5	2

	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						
					p						
					i						

1 Vorkommen

Mehrere Vorkommen?

Vorkommen benachbart!

sip = sip

SA	10	7	4	1	0	9	8	6	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

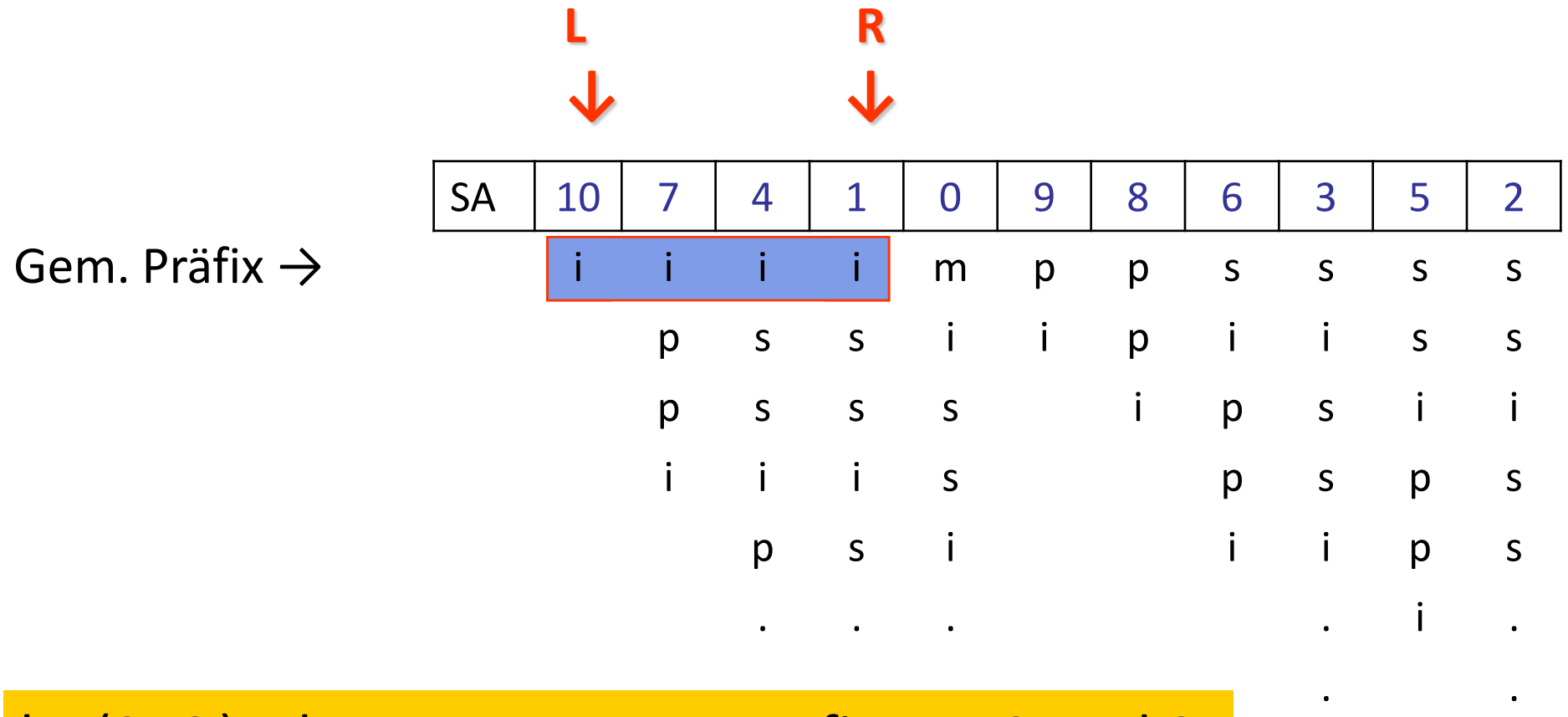
2 Vorkommen

Suche

- Laufzeit naiv:
  logn Strings, Musterlänge *m*, linear *k* Nachbarn suchen
  ➔ *O(m(logn+k))*
- Lange gleiche Präfixe – hohe Laufzeit
- Verbesserung praktisch / theoretisch ?

Beschleunigung

Auslassen bekannter Präfixe



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

Beschleunigung

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

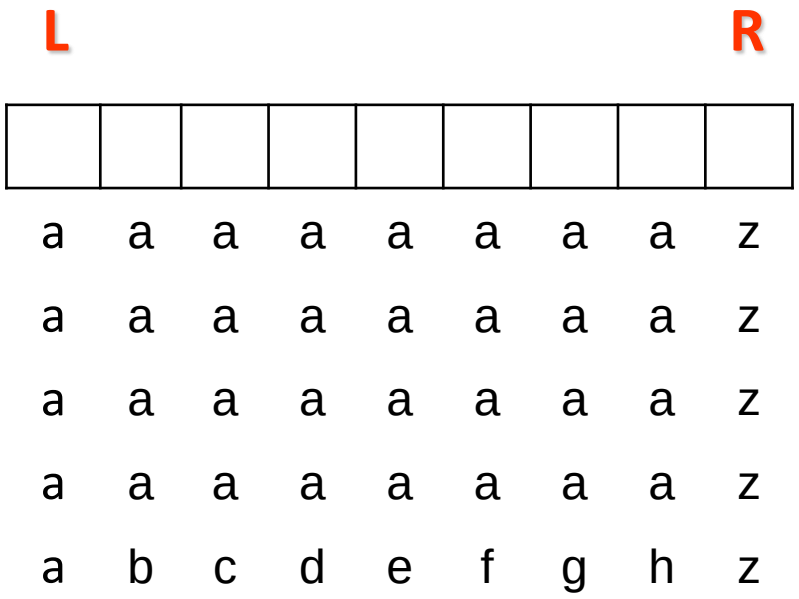
Laufzeitverbesserung:

- Praktisch: Klar
- Theoretisch: Worst-Case?

$min = 0$

➔ keine Verbesserung der asympt. Worst-Case Laufzeit

Beispiel



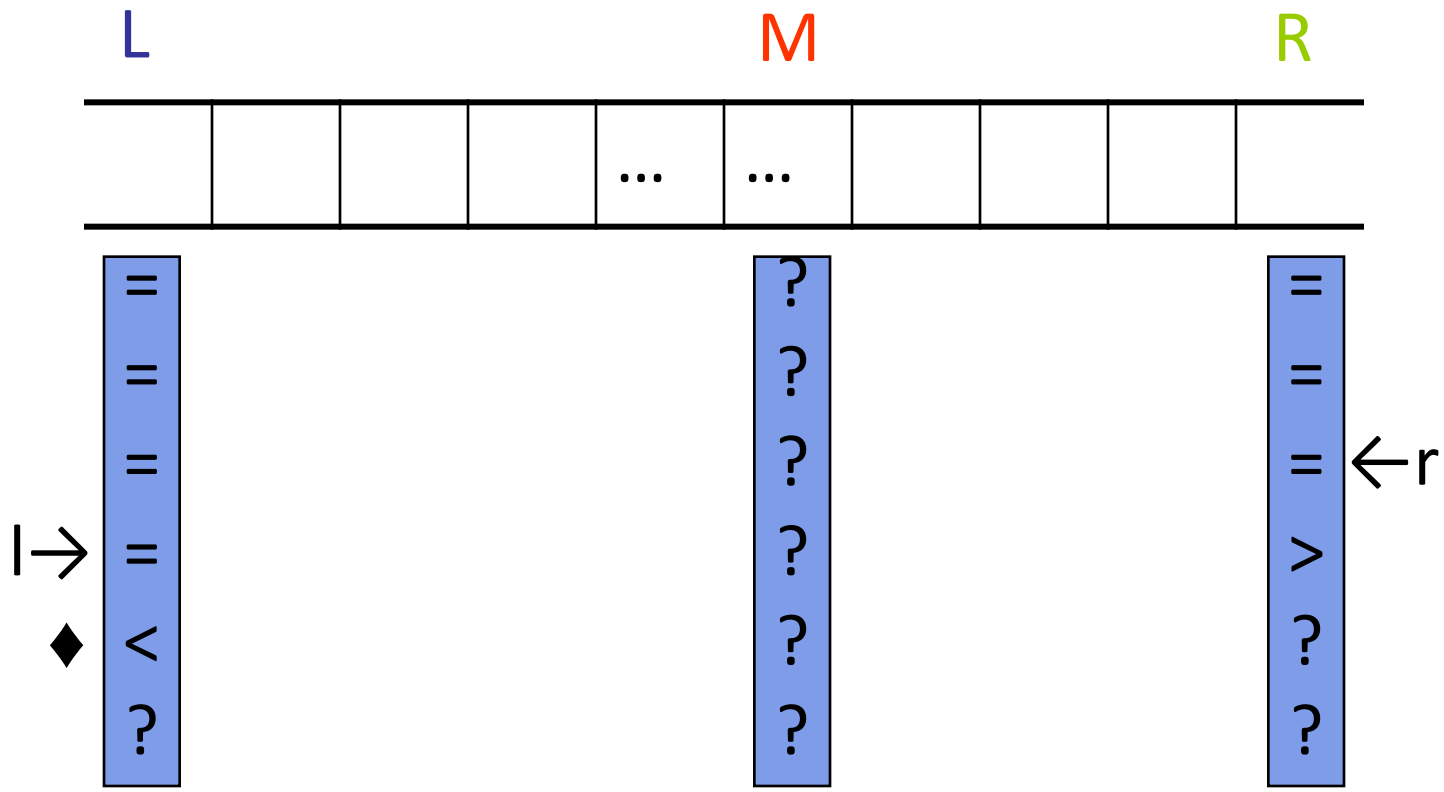
Suche nach Muster aaaah:

Rechter Rand bleibt immer gleich, $l_{match} = min(4,0) = 0$

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

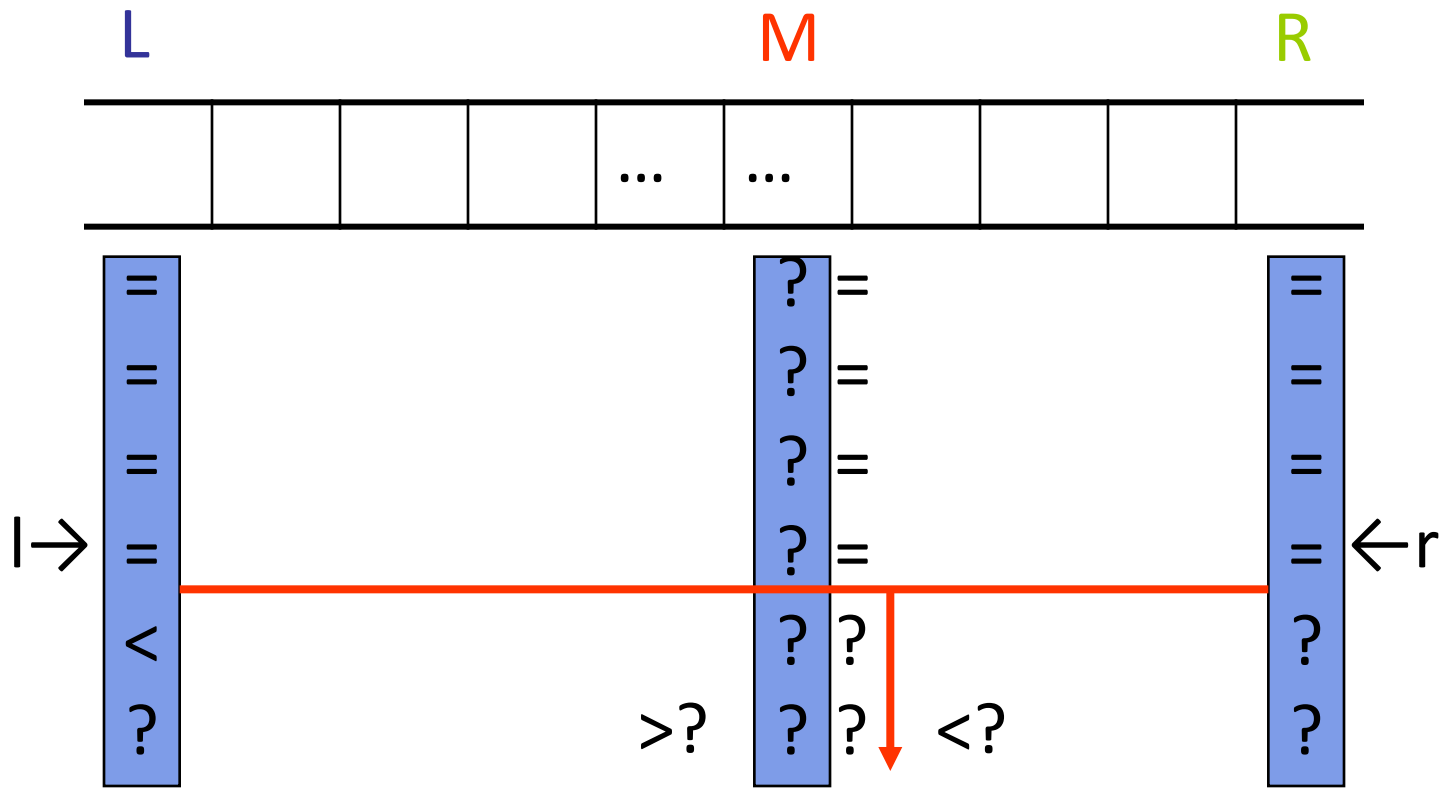
Beschleunigung II



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

Suche links oder rechts von M fortsetzen?

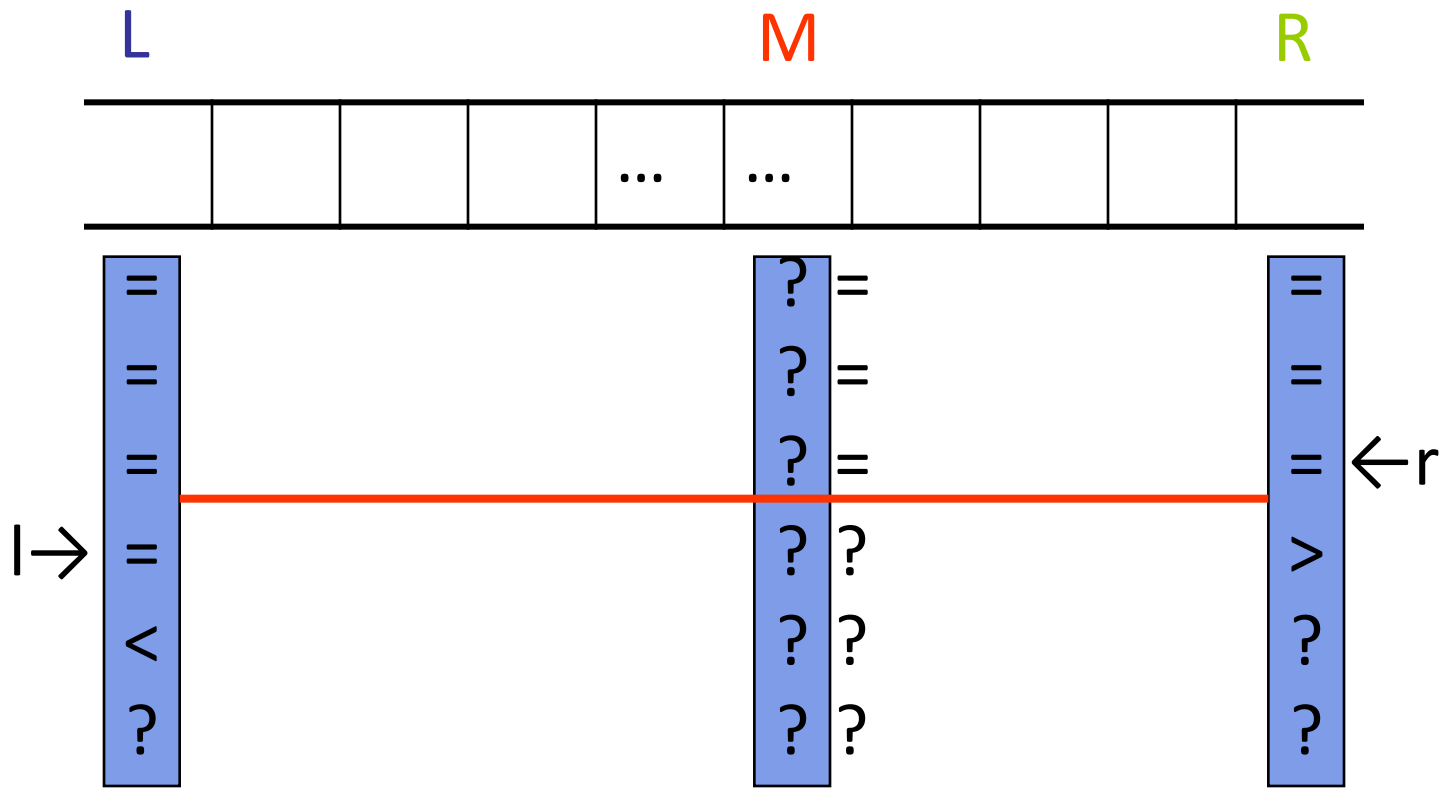
Beschleunigung II



Fall 1: l = r $\Rightarrow$ Vergleiche ab Position l+1 bis Unterschied

Links oder rechts von M fortsetzen, l/r anpassen

Beschleunigung II



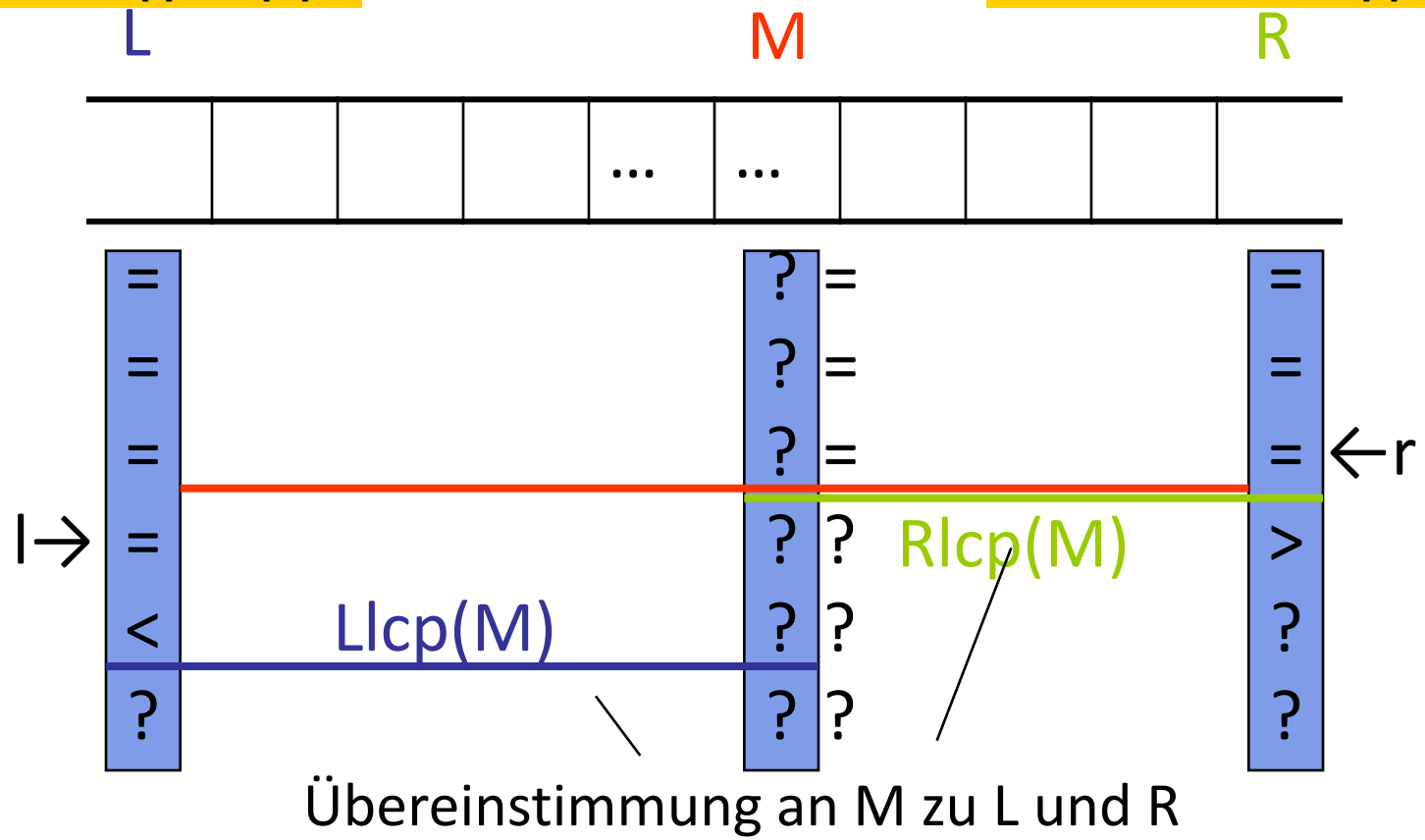
Fall 2: l > r

Wir nutzen jetzt nicht nur Übereinstimmung von p mit L, R, sondern auch von M zu L und R

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

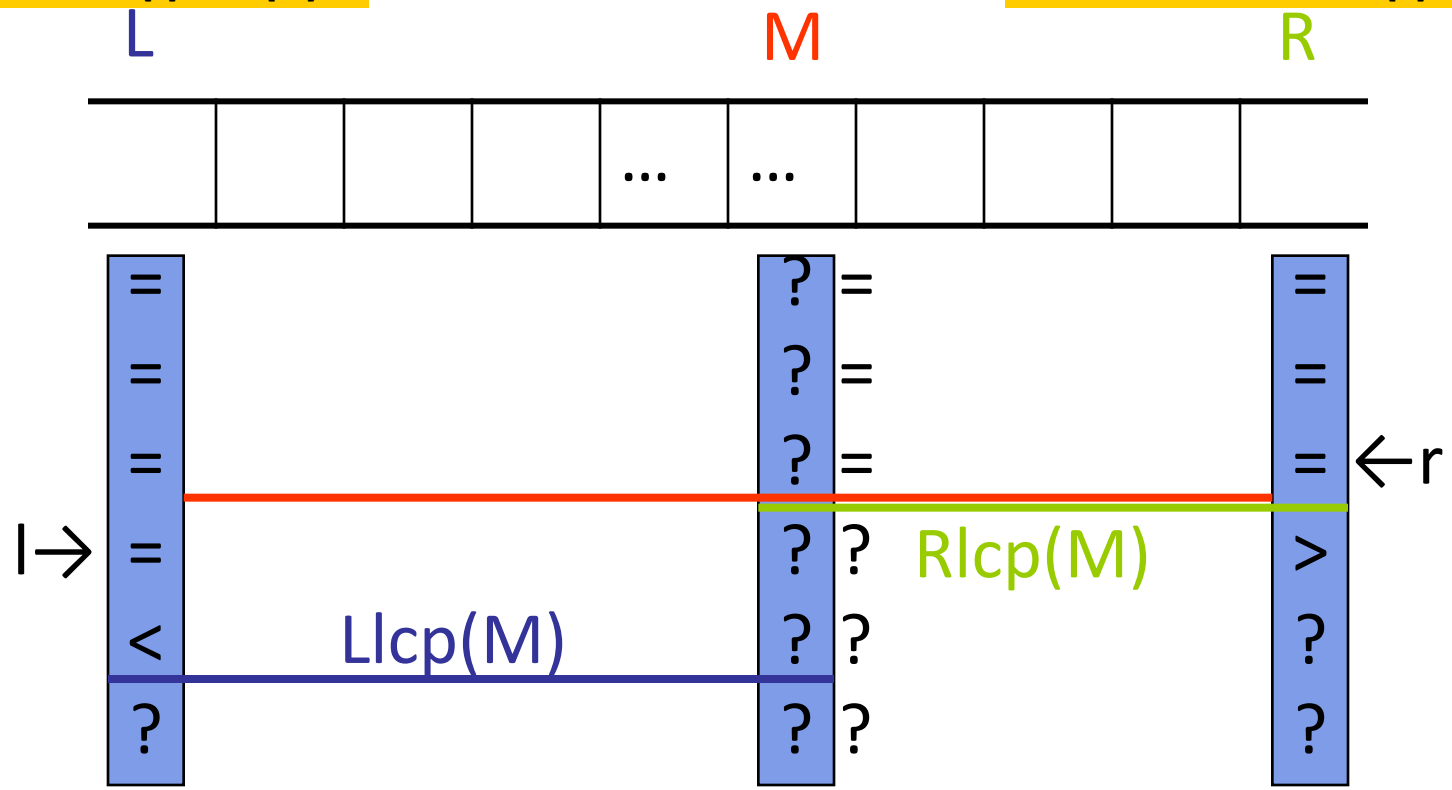
Fall 2: l > r

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

Beschleunigung II

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

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



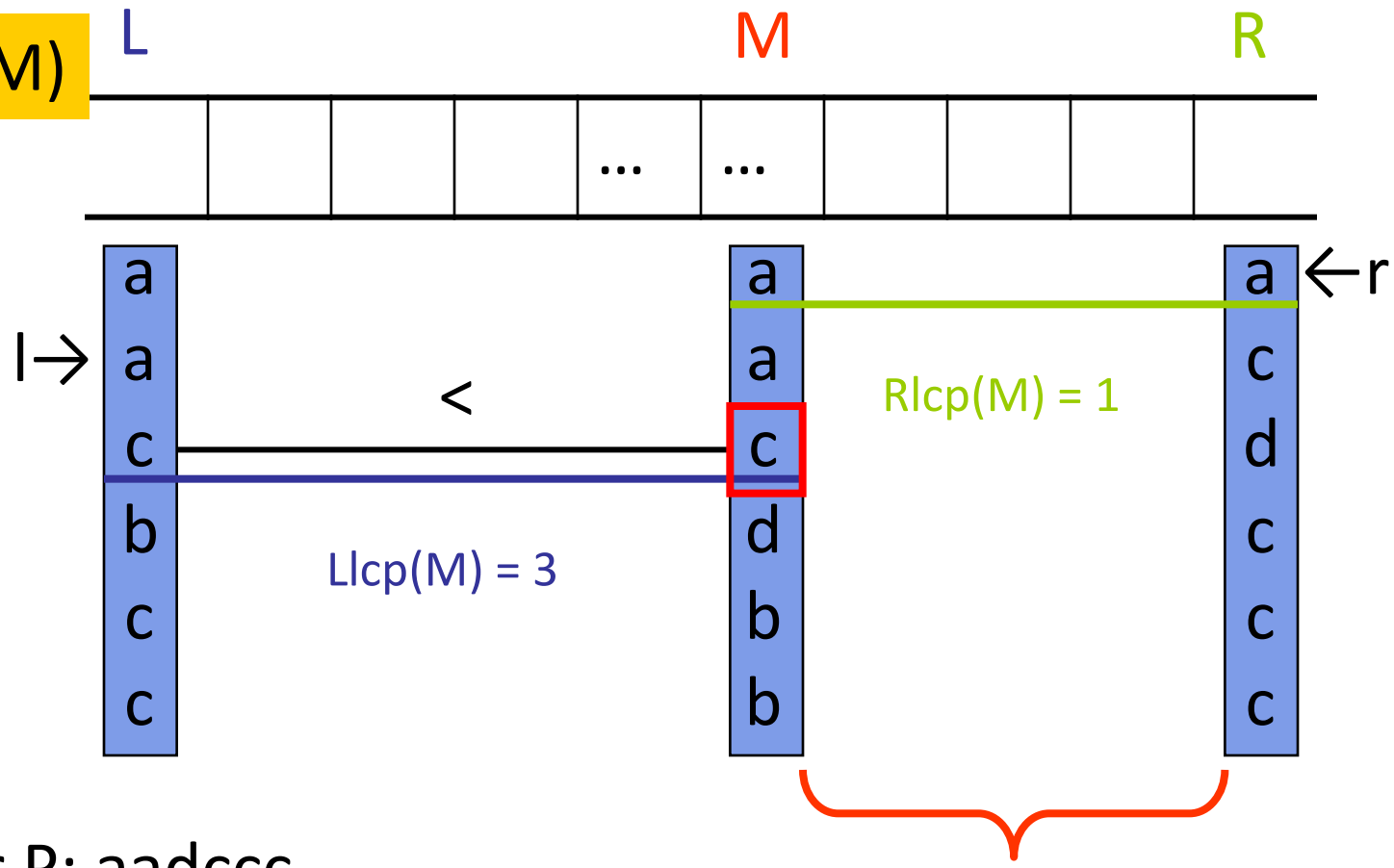
Fall 2: l > r

Llcp und Rlcp sind statisch berechenbar (Texteigenschaft) und L/Rlcp(M) ≥ min(l,r)

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

Beschleunigung II

Fall 2a)
l<Llcp(M)



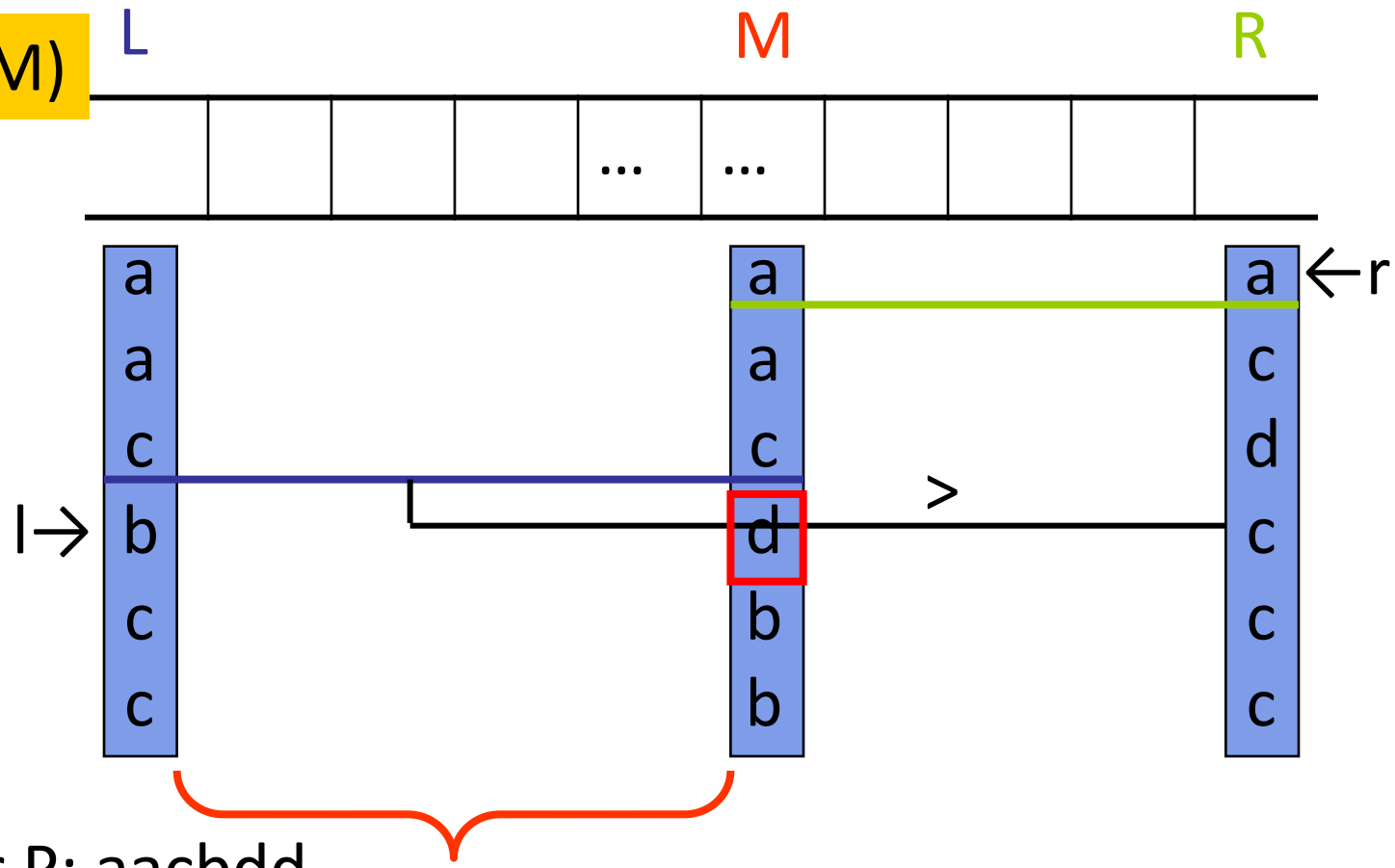
Muster P: aadccc

Nächstes Zeichen an M kleiner

Lösung rechts, l bleibt

Beschleunigung II

Fall 2b)
l>Llcp(M)



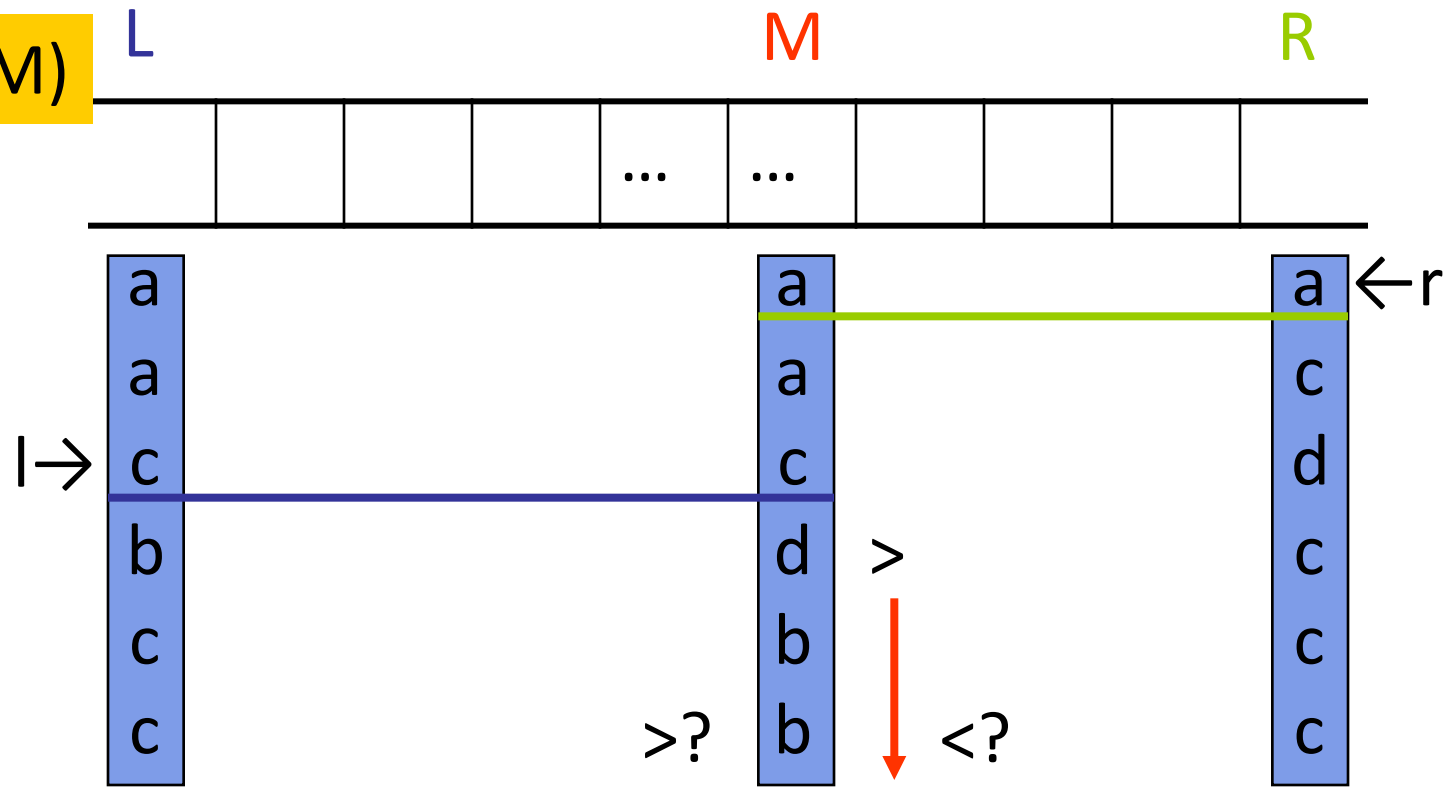
Muster P: aacbdd

Zeichen an M größer

Lösung links, r = Llcp(M)

Beschleunigung II

Fall 2c)
l=Llcp(M)



Muster P: aaccdd

Überspringe l Zeichen, vergleiche ab l+1

Beschleunigung II

Fall 3 (r < l) symmetrisch
Test für max(Llcp, Rlcp) ➔ maximale Schrittweite

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

Gesamtlaufzeit der Suche?

Nicht erfolgreicher Vergleich ➔ Intervallhalbierung

Erfolgreicher Vergleich ➔ Suchindex + 1

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

Beschleunigung II

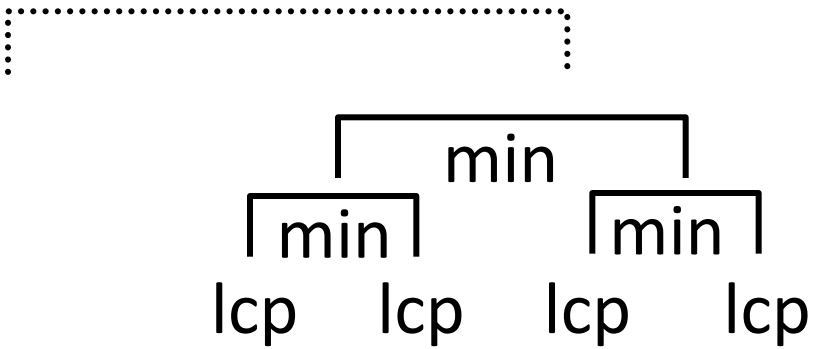
Berechnung von Llcp, Rlcp: Wieviele (L,M,R)?

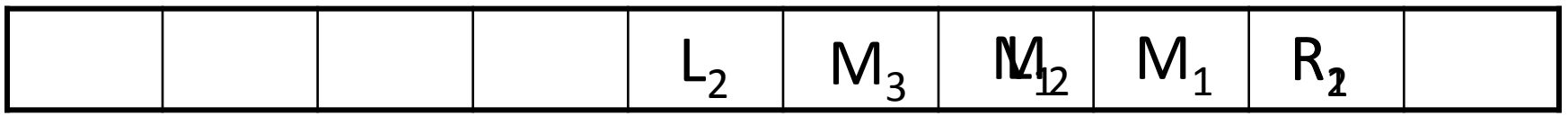
n-2 verschiedene M, dabei L,R fix

Wir beginnen mit lcp benachbarter Suffixe

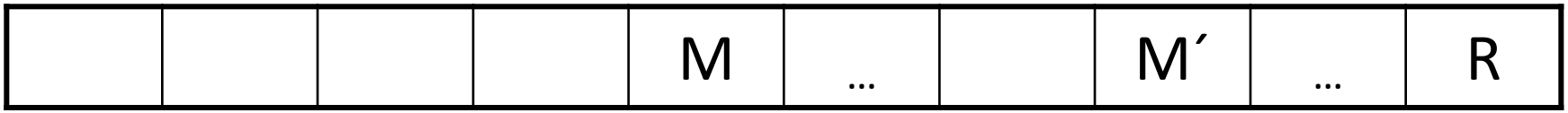
lcp hier $lcp(S_{SA[i]}, S_{SA[i+1]})$

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





Beschleunigung II

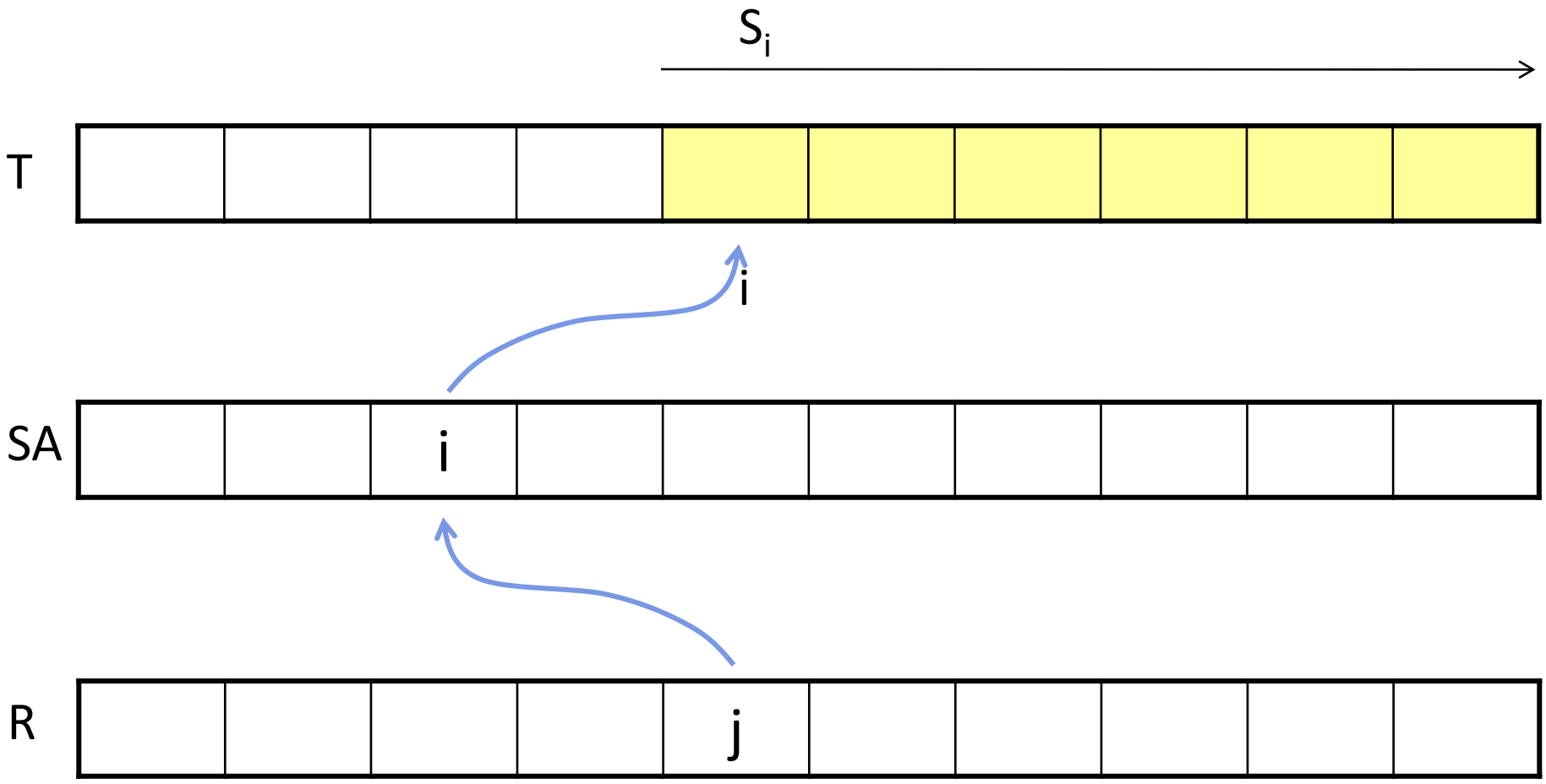


Bottom-up traversal der lcp-Werte für Nachbarn

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

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

Einschub: Inverses Suffix Array R

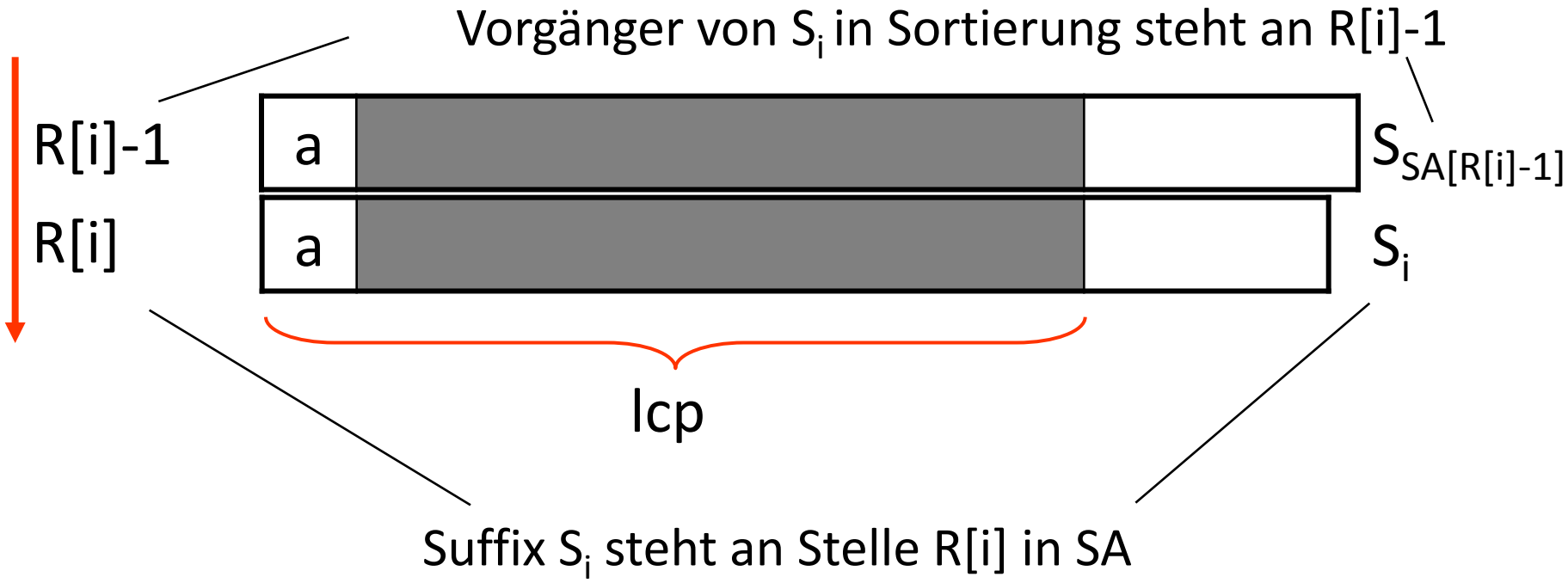


Reverse Array R: SA[R[i]] = i

LCP-Tabelle für Nachbarn

Berechnung der speziellen lcp-Tabelle L

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

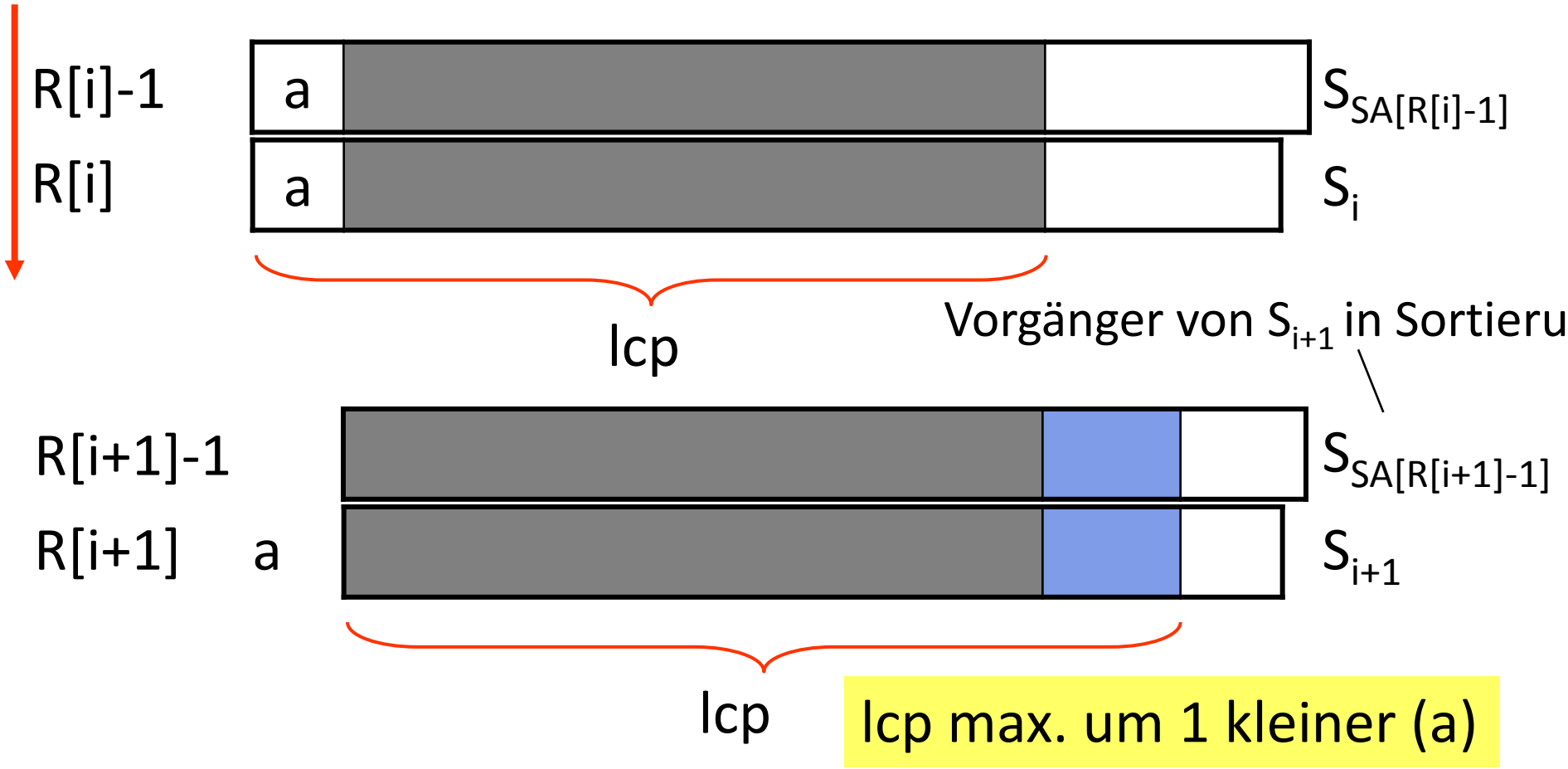


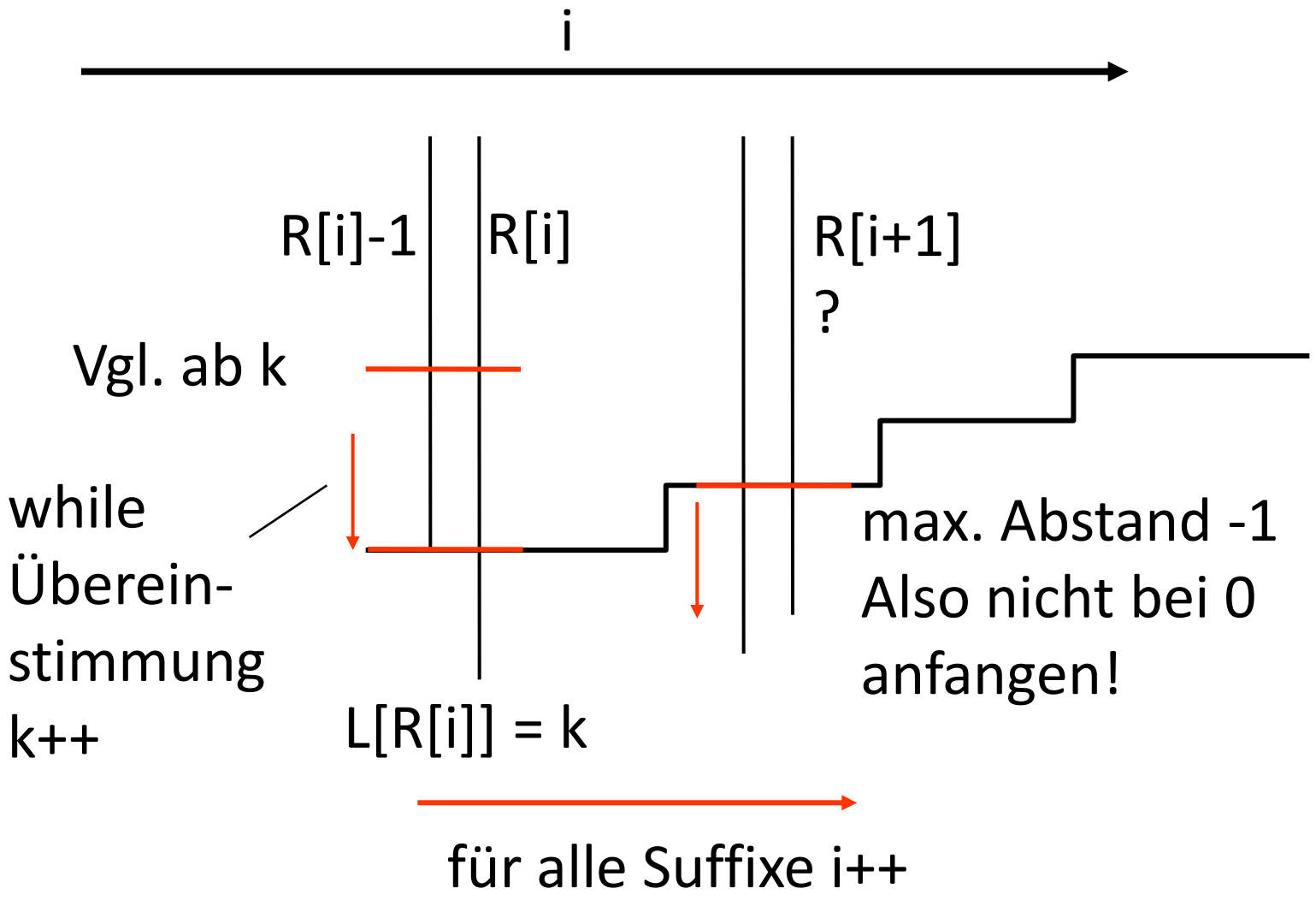
Wir laufen alle Suffixe der Reihe nach ab, nächstes Suffix ist ein Zeichen kürzer

LCP-Tabelle für Nachbarn

Berechnung der speziellen lcp-Tabelle L

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





Max. 2n Schritte

LCP-Tabelle

Vorgängerbeziehung:

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

```
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 + logn)* mit linearem Aufbau

Hauptpunkte

- Suffix Array ist lexikographisch geordneter Array der Suffixe eines Strings
- Aufbau in Linearzeit/-platz
- Platzeffiziente Datenstruktur (ca. 4n bytes)
- Anwendung: Stringvergleiche, Suche
- Suche in Zeit O(m + logn)) mittels Berechnung von lcp-Tabellen
- 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: „Vorkommenswahrscheinlichkeit von Zeichen“, Effizient (Kompressionsrate), aber eher langsam
    Huffman (Kodierer /Datenmodell),
    arithmetische Kodierung, PPM („qu“)
- Blockorientierte Verfahren
  - Kontext-orientiert: Burrows-Wheeler

Statisch: Wörterbuch/Codetabelle benötigt
Adaptiv: Während (De)Kompression aufgebaut

Kompression Burrows-Wheeler

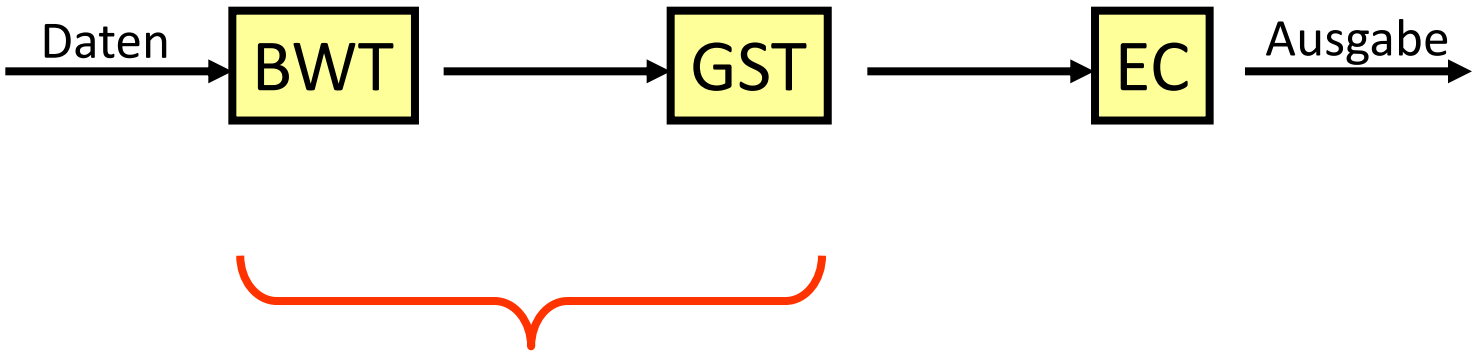
- 1994 veröffentlicht („A block-sorting lossless data compression algorithm“), seither Vielzahl von Verbesserungen
- Hohe Kompressionsraten bei hoher Geschwindigkeit möglich (blockabh.), verwendet z.B. in bzip2
- „Relativ“ geringer Platzverbrauch
- Blockorientiert, je größer desto besser
  $\Rightarrow$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



Keine Komprimierung! Clevere Kombination von Verfahren, um Komprimierung zu verbessern.

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

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

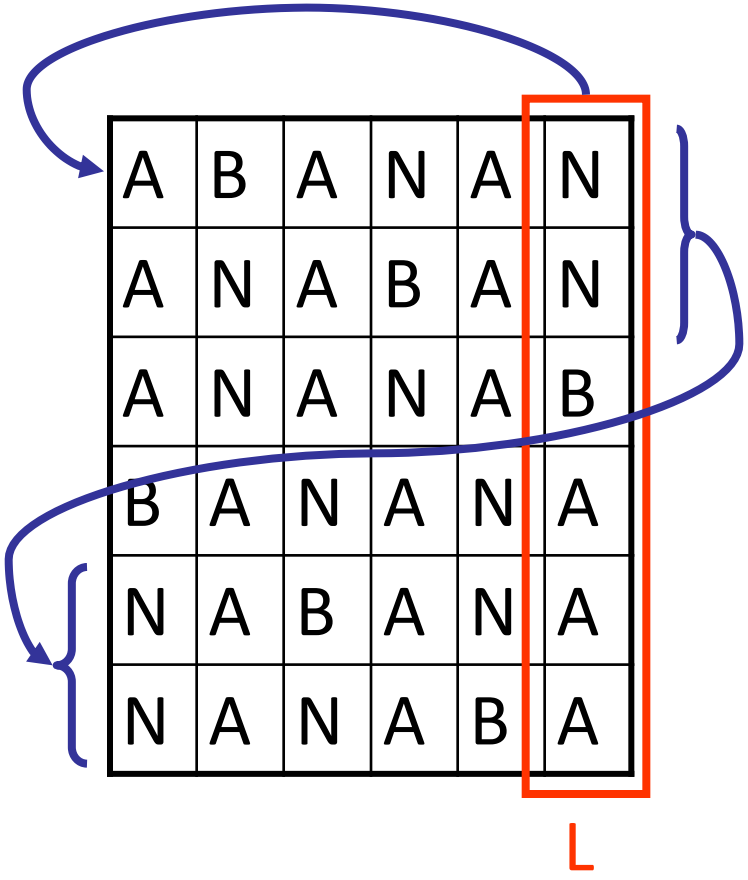
L

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

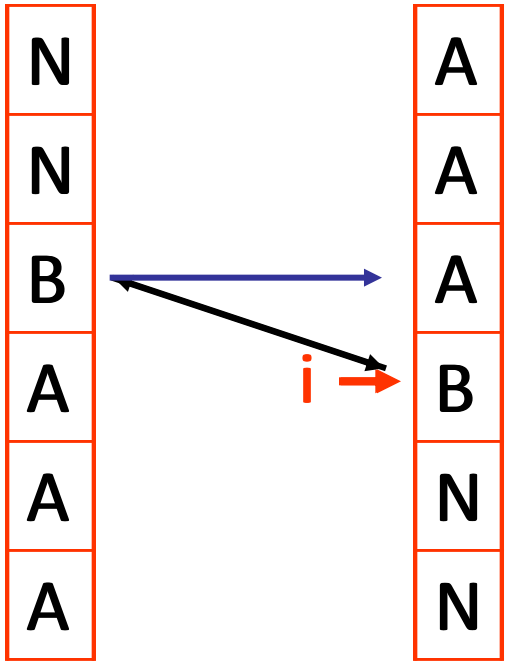


Rücktransformation

- Jede Spalte erhält alle Zeichen, also auch L

$\Rightarrow$ Erste Spalte durch Sortierung der Zeichen

$\Rightarrow$ Erstes Zeichen des Blocks steht an i



B A

I bekannt, also erstes Zeichen B

Wegen Rotation steht nächstes Zeichen in 1. Spalte derselben Zeile, A

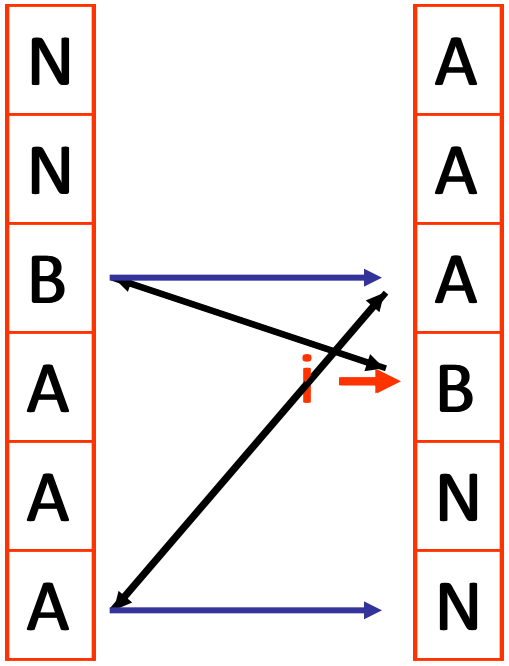
1. Spalte?

Rücktransformation

- Jede Spalte erhält alle Zeichen, also auch L

$\Rightarrow$ Erste Spalte durch Sortierung der Zeichen

$\Rightarrow$ Erstes Zeichen des Blocks steht an i



B A N

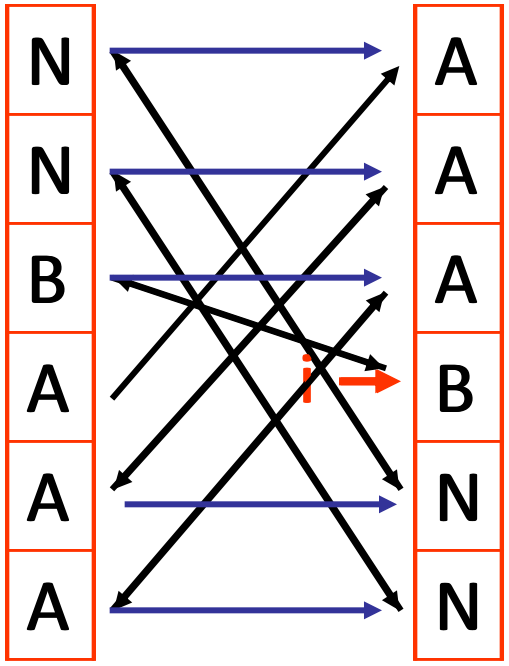
Zeilen mit gleichen Zeichen haben dieselbe Reihenfolge in erster und letzter Spalte, also nächstes Zeichen N

Rücktransformation

- Jede Spalte erhält alle Zeichen, also auch L

$\Rightarrow$ Erste Spalte durch Sortierung der Zeichen

$\Rightarrow$ Erstes Zeichen des Blocks steht an i

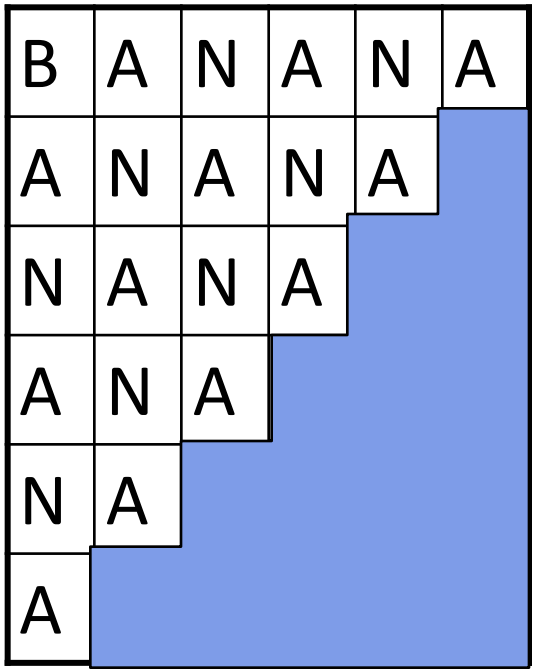


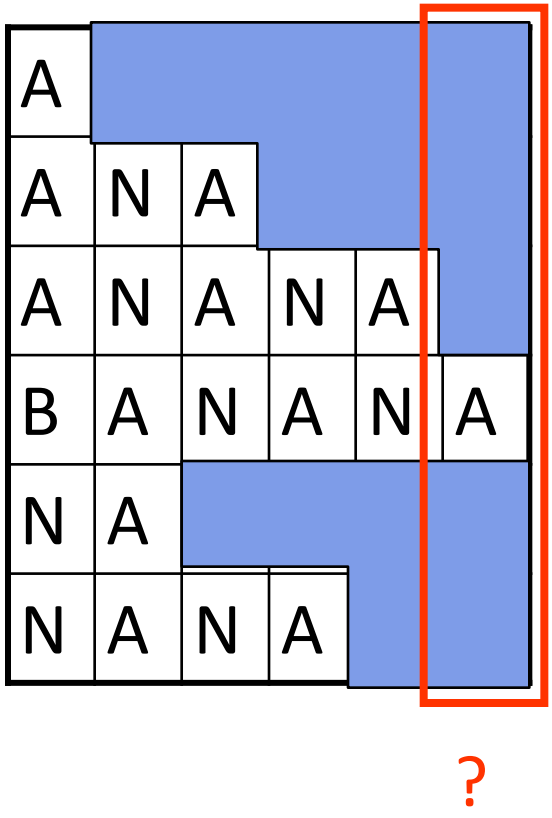
B A N A N A

Zeilenreihenfolge heißt auch „Transformationsvektor“

Burrows-Wheeler-Transformation

Was hat das denn mit Suffix-Arrays zu tun?





Burrows-Wheeler-Transformation

BWT kann mit Suffix Array berechnet werden!

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

BANANA

0 1 2 3 4 5

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

Burrows-Wheeler-Transformation

BWT kann mit Suffix Array berechnet werden!

- Speicherung des Suffix Array kann schon zu viel Platz brauchen (!!)

$\Rightarrow$ Nur Teile des SA speichern, die gerade benötigt werden [Kärkkäinen: *Fast BWT in small space by blockwise suffix sorting*, 2007]

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 ≥ m sind „m-Intervall“

lcp-Intervall

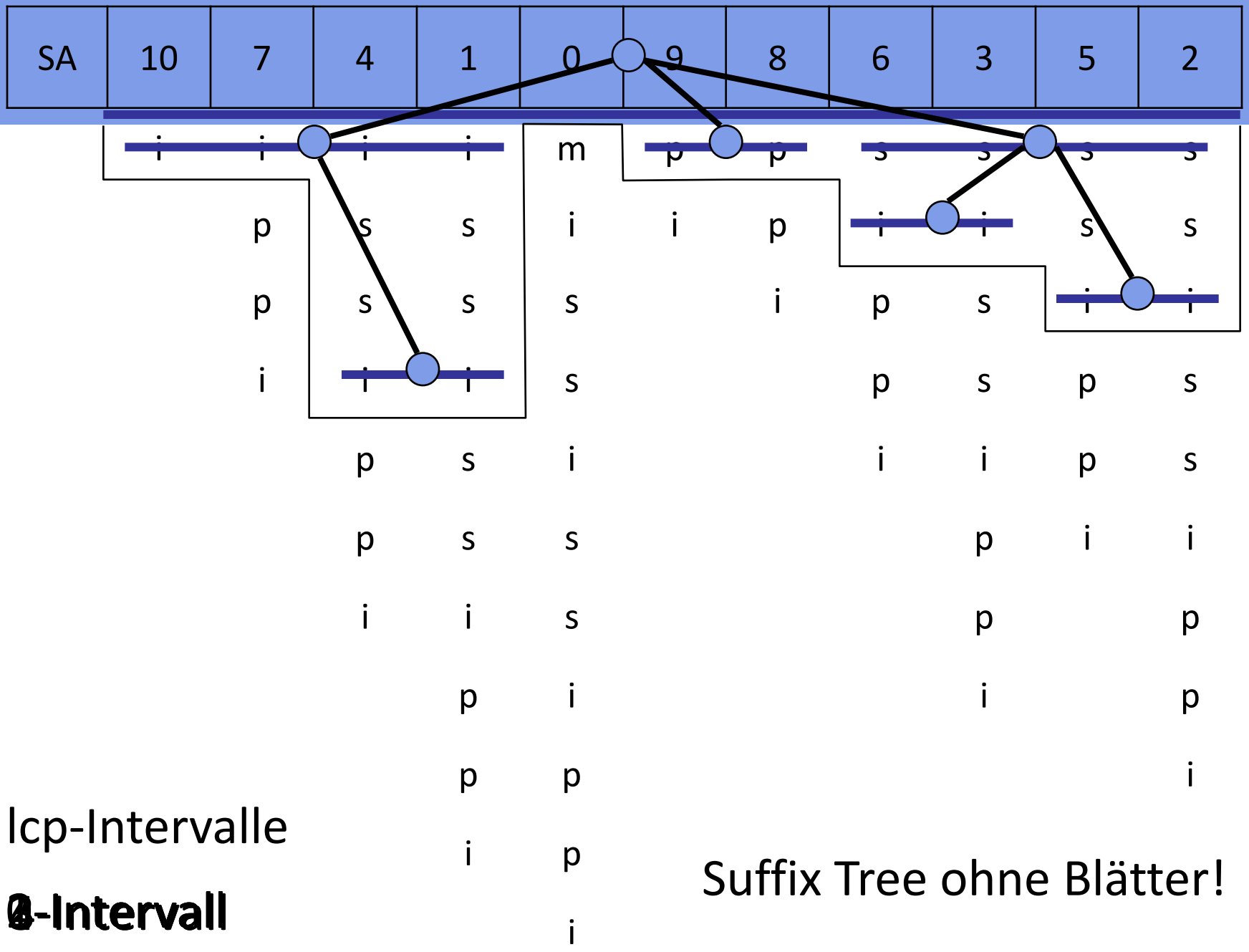
Formal:

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

lcp-Intervall mit lcp-Wert m

wenn

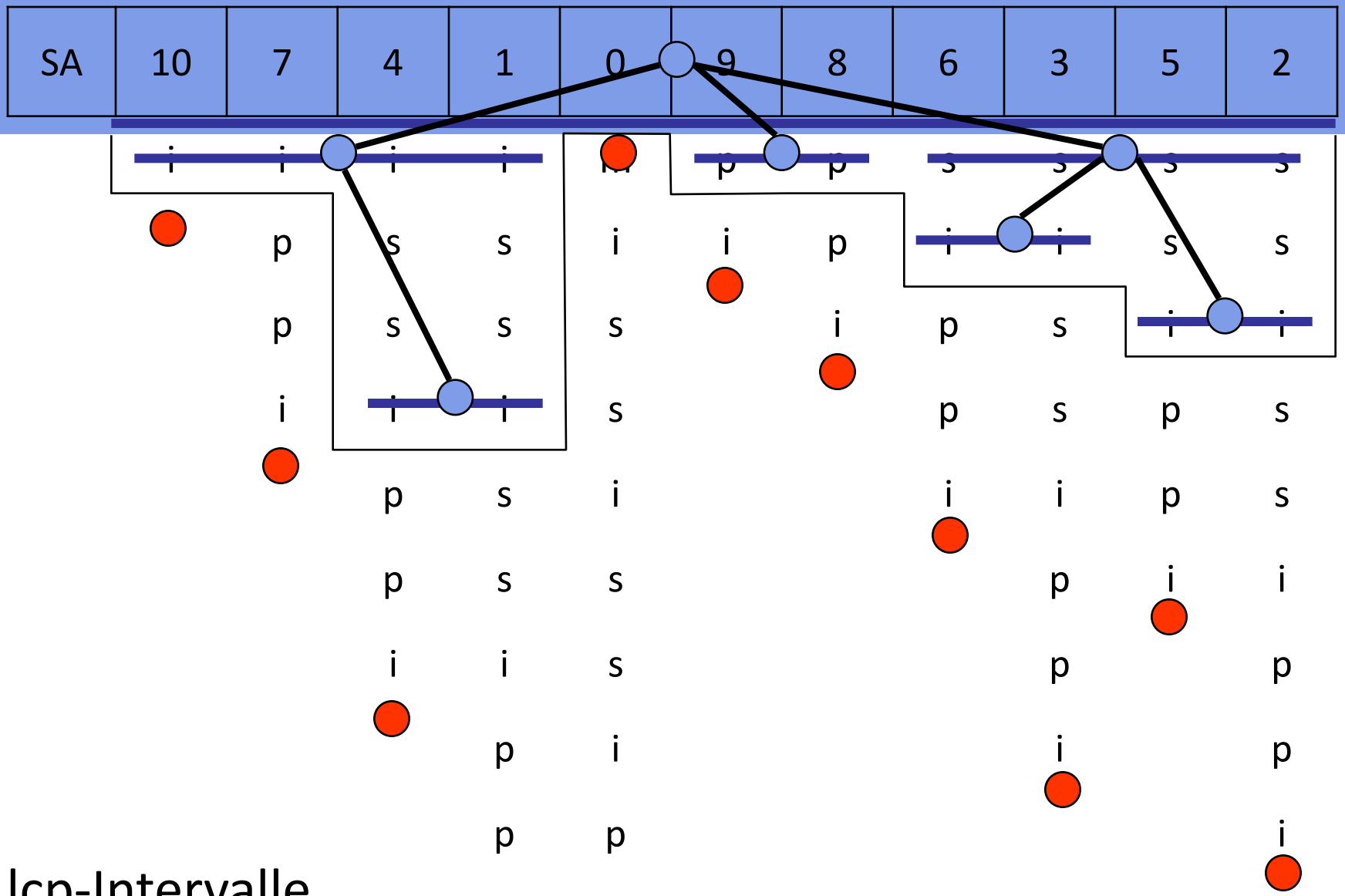
1. $L[i] < m$
2. $L[k] \geq m$ für alle $i+1 \leq k \leq j$
3. $L[k] = m$ für mind. ein solches k
4. $L[j+1] < m$



lcp-Intervall

4-Intervall

Suffix Tree ohne Blätter!



lcp-Intervalle

4-Intervall

Suffix Tree ohne Blätter!

Bausteine

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

LCP Tabelle

- Problem: Verdopplung des Speicherplatzbedarfs
- Idee: In Praxis lcp << 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!