

# Algorithmen auf Sequenzen

## **Volltext-Indexdatenstrukturen: Suffixarrays**

Sven Rahmann

Genominformatik  
Universitätsklinikum Essen  
Universität Duisburg-Essen  
Universitätsallianz Ruhr

# Motivation für Suffixarrays

- Speicherverbrauch für Suffixbaum hoch ( $\mathcal{O}(n) \approx 20n$  Bytes)

# Motivation für Suffixarrays

- Speicherverbrauch für Suffixbaum hoch ( $\mathcal{O}(n) \approx 20n$  Bytes)
- Bei alphabetisch sortierten ausgehenden Kanten:  
Sequenz der Blattnummern  
= Startpositionen der lexikographisch sortierten Suffixe
- Array:  $\mathcal{O}(4n)$  Bytes (32-bit-Zahlen)
- erlaubt binäre Suche nach Mustern in  $\mathcal{O}(m \log n)$  Zeit

# Motivation für Suffixarrays

- Speicherverbrauch für Suffixbaum hoch ( $\mathcal{O}(n) \approx 20n$  Bytes)
- Bei alphabetisch sortierten ausgehenden Kanten:  
Sequenz der Blattnummern  
= Startpositionen der lexikographisch sortierten Suffixe
- Array:  $\mathcal{O}(4n)$  Bytes (32-bit-Zahlen)
- erlaubt binäre Suche nach Mustern in  $\mathcal{O}(m \log n)$  Zeit
- Abbilden der Baumstruktur mit zusätzlichen Arrays
- Manche Anwendungen bekommen direkt cache-effiziente Algorithmen

# Beispiel eines Suffixarrays

Notation:  $p$  indiziert Textpositionen,  $r$  Ränge.

Im Suffixarray ist  $pos[r]$  die Startposition des lexikographisch  $r$ -t kleinsten Suffix.

$p =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$T =$	m	i	i	s	s	i	s	s	i	p	p	i	i	\$
$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3

# Beispiel eines Suffixarrays

Notation:  $p$  indiziert Textpositionen,  $r$  Ränge.

Im Suffixarray ist  $pos[r]$  die Startposition des lexikographisch  $r$ -t kleinsten Suffix.

$p =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$T =$	m	i	i	s	s	i	s	s	i	p	p	i	i	\$
$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3
	\$	i						m		p		s		

Aufteilung der Suffixe in "Buckets" gemäß Startbuchstaben.

# Konstruktion eines Suffixarrays

- 1 aus dem Suffixbaum durch Ablesen in  $\mathcal{O}(n)$  Zeit  
Nachteil: hoher Speicherverbrauch für Baumkonstruktion

# Konstruktion eines Suffixarrays

- 1 aus dem Suffixbaum durch Ablesen in  $\mathcal{O}(n)$  Zeit  
Nachteil: hoher Speicherverbrauch für Baumkonstruktion
- 2 direkt durch naives Sortieren

```
1 def build_suffixarray_naive(T):  
2     n = len(T)  
3     suffixes = lambda p: T[p:]  
4     return sorted(range(n), key=suffixes)
```

Laufzeit:



# Konstruktion eines Suffixarrays

- 1 aus dem Suffixbaum durch Ablesen in  $\mathcal{O}(n)$  Zeit  
Nachteil: hoher Speicherverbrauch für Baumkonstruktion
- 2 direkt durch naives Sortieren

```
1 def build_suffixarray_naive(T):  
2     n = len(T)  
3     suffixes = lambda p: T[p:]  
4     return sorted(range(n), key=suffixes)
```

Laufzeit:  $\mathcal{O}(n^2 \log n)$

# Konstruktion eines Suffixarrays

- 1 aus dem Suffixbaum durch Ablesen in  $\mathcal{O}(n)$  Zeit  
Nachteil: hoher Speicherverbrauch für Baumkonstruktion
- 2 direkt durch naives Sortieren

```
1 def build_suffixarray_naive(T):  
2     n = len(T)  
3     suffixes = lambda p: T[p:]  
4     return sorted(range(n), key=suffixes)
```

Laufzeit:  $\mathcal{O}(n^2 \log n)$

- 3 direkt durch Linearzeitalgorithmus in  $\mathcal{O}(n)$  Zeit  
hier: Suffix Array Induced Sorting (SAIS) Algorithmus

# Typen von Positionen

Bei Teilstrings der Form  $T_p \geq T_{p+1} \geq \dots \geq T_q$  (z.B.: ...dccbba...) kommen die Positionen in umgekehrter Reihenfolge im Suffixarray als Teilsequenz vor, also  $q \dots (p + 2) \dots (p + 1) \dots p \dots$

# Typen von Positionen

Bei Teilstrings der Form  $T_p \geq T_{p+1} \geq \dots \geq T_q$  (z.B.: ...dccbba...) kommen die Positionen in umgekehrter Reihenfolge im Suffixarray als Teilsequenz vor, also  $q \dots (p+2) \dots (p+1) \dots p \dots$

Bei Teilstrings der Form  $T_p \leq T_{p+1} \leq \dots \leq T_q$  (z.B.: ...abbccd...) kommen die Positionen in der selben Reihenfolge im Suffixarray als Teilsequenz vor, also z.B.:  $p \dots (p+1) \dots (p+2) \dots q \dots$

# Typen von Positionen

Bei Teilstrings der Form  $T_p \geq T_{p+1} \geq \dots \geq T_q$  (z.B.: ...dccbba...) kommen die Positionen in umgekehrter Reihenfolge im Suffixarray als Teilsequenz vor, also  $q \dots (p+2) \dots (p+1) \dots p \dots$

Bei Teilstrings der Form  $T_p \leq T_{p+1} \leq \dots \leq T_q$  (z.B.: ...abbccd...) kommen die Positionen in der selben Reihenfolge im Suffixarray als Teilsequenz vor, also z.B.:  $p \dots (p+1) \dots (p+2) \dots q \dots$

## Definition (L-type-Position, S-type-Position)

Position  $p$  heisst vom Typ L (*L-type*; larger), wenn  $T[p:] > T[(p+1):]$ .  
(d.h.  $T[p] > T[p+1]$ , oder bei Gleichheit war schon Position  $p+1$  L-type)

# Typen von Positionen

Bei Teilstrings der Form  $T_p \geq T_{p+1} \geq \dots \geq T_q$  (z.B.: ...dccbba...) kommen die Positionen in umgekehrter Reihenfolge im Suffixarray als Teilsequenz vor, also  $q \dots (p + 2) \dots (p + 1) \dots p \dots$

Bei Teilstrings der Form  $T_p \leq T_{p+1} \leq \dots \leq T_q$  (z.B.: ...abbccd...) kommen die Positionen in der selben Reihenfolge im Suffixarray als Teilsequenz vor, also z.B.:  $p \dots (p + 1) \dots (p + 2) \dots q \dots$

## Definition (L-type-Position, S-type-Position)

Position  $p$  heisst vom Typ L (*L-type*; larger), wenn  $T[p:] > T[(p + 1):]$ .  
(d.h.  $T[p] > T[p + 1]$ , oder bei Gleichheit war schon Position  $p + 1$  L-type)  
Position  $p$  heisst vom Typ S (*S-type*; smaller), wenn  $T[p:] < T[(p + 1):]$ .  
Der Wächter \$ wird als S-type definiert.

## Definition (LMS-type Position)

Position  $p$  heißt *LMS-Type* (leftmost-smaller),  
wenn  $p$  eine S-type Position ist und  $(p - 1)$  eine L-type Position ist.

# LMS-type

## Definition (LMS-type Position)

Position  $p$  heißt *LMS-Type* (leftmost-smaller),  
wenn  $p$  eine S-type Position ist und  $(p - 1)$  eine L-type Position ist.

$T =$	miissippii\$
Type :	LSSLLSLLSLLLLS
LMS :	* * * *

Die LMS-type Positionen stellen jeweils das Ende einer absteigenden Reihenfolge dar.



# LMS-type

## Definition (LMS-type Position)

Position  $p$  heißt *LMS-Type* (leftmost-smaller), wenn  $p$  eine S-type Position ist und  $(p - 1)$  eine L-type Position ist.

$T =$	miissippii\$
Type :	LSSLLSLLSLLLLS
LMS :	* * * *

Die LMS-type Positionen stellen jeweils das Ende einer absteigenden Reihenfolge dar.

## Definition (LMS-Suffix, LMS-Substring)

Ein Suffix  $T[p : ]$  heißt *LMS-Suffix*, wenn  $p$  vom LMS-Typ ist.

Ein Substring  $T[p..q]$  heißt *LMS-Substring*, wenn  $p$  und  $q$  LMS-Positionen sind, aber keine Position dazwischen. Der Wächter gilt als einbuchstabiger LMS-Substring.

Links-Induktions-Scan:

- 1 Initialisiere ein Suffixarray der Länge  $n$  mit jeweils  $-1$  (undefiniert).
- 2 Sortiere alle LMS-Suffixe (rekursiv, später)
- 3 Stelle die sortierten LMS-Suffixe jeweils ans Ende ihrer Buckets.

Links-Induktions-Scan:

- 1 Initialisiere ein Suffixarray der Länge  $n$  mit jeweils  $-1$  (undefiniert).
- 2 Sortiere alle LMS-Suffixe (rekursiv, später)
- 3 Stelle die sortierten LMS-Suffixe jeweils ans Ende ihrer Buckets.
- 4 Scane das Suffixarray  $pos$  von links nach rechts mit Index  $r$
- 5 Für  $pos[r] \neq -1$ :  
Ist  $p := pos[r] - 1$  L-type, füge  $p$  an vorderste freie Stelle seines Buckets ein.

Links-Induktions-Scan:

- 1 Initialisiere ein Suffixarray der Länge  $n$  mit jeweils  $-1$  (undefiniert).
- 2 Sortiere alle LMS-Suffixe (rekursiv, später)
- 3 Stelle die sortierten LMS-Suffixe jeweils ans Ende ihrer Buckets.
- 4 Scane das Suffixarray  $pos$  von links nach rechts mit Index  $r$
- 5 Für  $pos[r] \neq -1$ :  
Ist  $p := pos[r] - 1$  L-type, füge  $p$  an vorderste freie Stelle seines Buckets ein.

## Lemma

Alle L-type Positionen werden durch einen Links-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.





# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miississip <i>p</i> ii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	* * * *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	<u>11</u>	-1	1	8	5	-1	-1	-1	-1	-1	-1	-1
	↑		↑					↑	↑		↑			
	⏟		⏟				⏟		⏟		⏟			
	\$				i			m	p					s

# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miissippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	*	* * *
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	<u>-1</u>	1	8	5	-1	10	-1	-1	-1	-1	-1
	↑		↑					↑	↑	↑				
	⏟		⏟				⏟		⏟		⏟			
	\$	i			m		p	s						



# Links-Induktions-Scan

$T =$	0            1
	01234567890123
	miississippii\$
Type :	LSSLLSLLSLLLLS
LMS :	*       *       *       *
seq =	[13, 1, 8, 5]

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	<u>1</u>	8	5	-1	10	-1	-1	-1	-1	-1
	↑			↑				↑		↑	↑			
	⏟		⏟			⏟		⏟	⏟		⏟			
	\$	i			m		p	s						

# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miississippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	*	* * *
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	<u>8</u>	5	0	10	-1	-1	-1	-1	-1
	↑		↑							↑	↑			
	⏟		⏟				⏟		⏟		⏟			
	\$	i				m		p		s				

# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miississippi\$	
Type :	LSSLLSLLSLLLLS	
LMS :	* * * *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	<u>5</u>	0	10	-1	7	-1	-1	-1
	↑		↑							↑		↑		
	└──────────┘		└──────────────────────────┘				└──┘		└──┘		└──────────────────────────┘			
	\$	i				m		p		s				

# Links-Induktions-Scan

$T =$	0	1
Type :	01234567890123	
LMS :	miissippii\$	
seq =	LSSLLSLLSLLLLS	
	* * * *	
	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	<u>0</u>	10	-1	7	4	-1	-1
	↑			↑						↑			↑	
	⏟	⏟			⏟			⏟	⏟	⏟				
	\$	i			m			p	s					

# Links-Induktions-Scan

	0	1
	01234567890123	
$T =$	mi	ssissippii\$
Type :	L	SLLSLLSLLLLS
LMS :	*	* * * *
seq =	[13,	1, 8, 5]

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	<u>10</u>	-1	7	4	-1	-1
	↑			↑						↑			↑	
	}	}				}			}		}			
	\$	i				m			p	s				

# Links-Induktions-Scan

$T =$	0	1
Type :	01234567890123	
LMS :	miississippi\$	
seq =	LSSLLSLLSLLLLS	
	* * * *	
	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	<u>9</u>	7	4	-1	-1
	↑		↑										↑	
	}	}			}			}	}	}				
	\$	i			m		p	s						

# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miissi <span style="color: red;">s</span> sippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	*   *   *   *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	9	<u>7</u>	4	-1	-1
	↑		↑										↑	
	⏟	⏟			⏟			⏟	⏟	⏟				
	\$	i			m			p	s					

# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	mi <i>ss</i> issippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	* * * *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	9	7	<u>4</u>	6	-1
	↑		↑											↑
	⏟	⏟			⏟			⏟	⏟		⏟			
	\$	i			m			p	s					



# Links-Induktions-Scan

$T =$	0	1
	01234567890123	
	miissiissippi\$	
Type :	LSSLLSLLSLLLLS	
LMS :	* * * *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	9	7	4	<u>6</u>	3
	↑		↑											
	⏟		⏟			⏟		⏟	⏟		⏟			
	\$	i			m		p	s						

# Links-Induktions-Scan

$T =$	0	1
Type :	01234567890123	
LMS :	mi <i>i</i> ssissippii\$	
seq =	LSSLLSLLSLLLLS	
	* * * *	
	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	9	7	4	6	<u>3</u>
	↑		↑											
	}	}			}			}	}		}			
	\$	i			m			p	s					

# Links-Induktions-Scan

	0	1
	01234567890123	
$T =$	miissippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	* * * *	
seq =	[13, 1, 8, 5]	

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	5	0	10	9	7	4	6	3
	↑		↑											
	}	}			}			}	}		}			
	\$	i			m			p	s					

## Lemma

Alle L-type Positionen werden durch einen Links-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.

## Lemma

Alle L-type Positionen werden durch einen Links-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.

- In jedem Bucket stehen L-type Positionen **vor** S-type Positionen.
- L-type-Positionen können nur von LMS-type- oder anderen L-type-Positionen eingetragen werden.
- LMS-Positionen (auch Wächterposition) ist korrekt eingetragen.

## Lemma

Alle L-type Positionen werden durch einen Links-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.

- In jedem Bucket stehen L-type Positionen **vor** S-type Positionen.
- L-type-Positionen können nur von LMS-type- oder anderen L-type-Positionen eingetragen werden.
- LMS-Positionen (auch Wächterposition) ist korrekt eingetragen.
- Steht Textposition  $p$  an Index  $r$  von  $pos$  und ist  $p - 1$  eine Typ-L-Position, dann steht  $p - 1$  an einem Index  $r' > r$ .

## Lemma

Alle L-type Positionen werden durch einen Links-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.

- In jedem Bucket stehen L-type Positionen **vor** S-type Positionen.
- L-type-Positionen können nur von LMS-type- oder anderen L-type-Positionen eingetragen werden.
- LMS-Positionen (auch Wächterposition) ist korrekt eingetragen.
- Steht Textposition  $p$  an Index  $r$  von  $pos$  und ist  $p - 1$  eine Typ-L-Position, dann steht  $p - 1$  an einem Index  $r' > r$ .
- Angenommen, Position  $q$  wird hinter einer Position  $p$  in dasselbe Bucket eingetragen, aber  $T[p:] > T[q:]$ . Dann ist aber  $T[p] = T[q]$ ; die Fehlsortierung hätte also schon für (bereits eingetragene)  $p + 1$  und  $q + 1$  eintreten müssen.  $\neq$

Rechts-Induktions-Scan:

- 1 Annahme: Alle L-type Positionen sind im Suffixarray korrekt eingetragen.
- 2 Scane das Suffixarray von rechts nach links mit Laufindex  $r$
- 3 Ist  $p := pos[r] - 1$  S-type, schreibe  $p$  an die hinterste freie Stelle seines Buckets.



Rechts-Induktions-Scan:

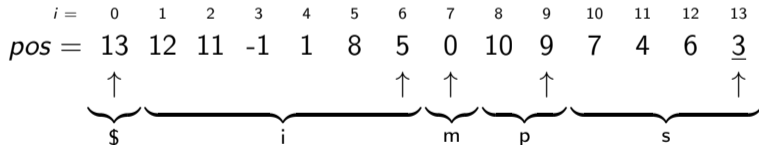
- 1 Annahme: Alle L-type Positionen sind im Suffixarray korrekt eingetragen.
- 2 Scane das Suffixarray von rechts nach links mit Laufindex  $r$
- 3 Ist  $p := pos[r] - 1$  S-type, schreibe  $p$  an die hinterste freie Stelle seines Buckets.

## Lemma

Wenn alle L-Type Zeichen an korrekter Position im Suffixarray stehen, dann werden alle S-Type Zeichen durch einen Rechts-Induktions-Scan an ihre korrekte Position im Suffixarray sortiert.

# Rechts-Induktions-Scan

$T =$	0	1
	01234567890123	
	mi <i>iss</i> issippii\$	
Type :	LSSLLSLLSLLLLS	
LMS :	*        *        *        *	



# Rechts-Induktions-Scan

$T =$

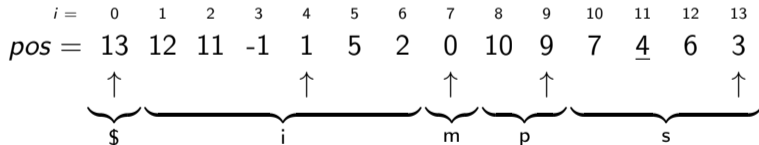
	0												1	
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
	m	i	s	s	i	s	s	i	p	p	i	i	\$	
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L
LMS :		*			*			*			*		*	

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	8	2	0	10	9	7	4	<u>6</u>	3
	↑					↑		↑		↑				↑
	}	}				}			}	}	}			
	\$	i				m			p	s				

# Rechts-Induktions-Scan

$T =$

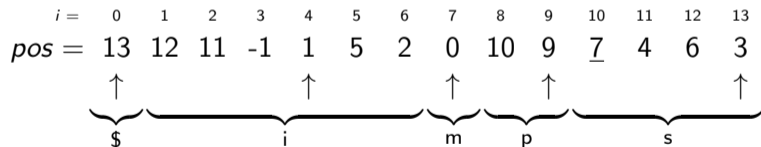
	0												1	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	m	i	s	s	i	s	s	i	p	p	i	i	\$	
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L
LMS :		*			*			*			*			*



# Rechts-Induktions-Scan

$T =$

	0												1		
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	
	m	i	i	s	s	i	p	p	i	i	\$				
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L	S
LMS :		*		*		*		*		*		*		*	



# Rechts-Induktions-Scan

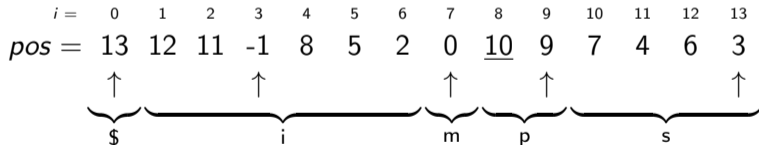
	0	1
	01234567890123	
$T =$	mi	ssissippi\$
Type :	L	SLLSLLSLLLLS
LMS :	*	* * *

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	1	5	2	0	10	<u>9</u>	7	4	6	3
	↑				↑			↑		↑				↑
	}	}				}			}	}				
	\$	i				m			p	s				

# Rechts-Induktions-Scan

$T =$

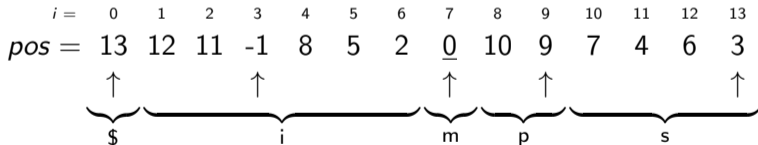
	0		1												
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	
	m	i	s	s	i	s	s	i	p	p	i	i	\$		
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L	S
LMS :		*			*			*			*			*	



# Rechts-Induktions-Scan

$T =$

	0		1											
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
	m	i	s	s	i	s	s	i	p	p	i	i	\$	
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	S
LMS :		*		*		*		*		*		*		*





# Rechts-Induktions-Scan

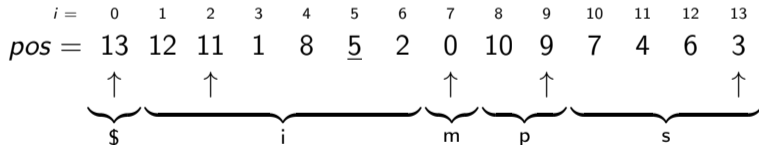
$T =$	0	1													
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	
	m	i	s	s	i	s	s	i	p	p	i	i	\$		
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L	S
LMS :		*					*		*		*		*		*

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	-1	8	5	<u>2</u>	0	10	9	7	4	6	3
	↑			↑				↑		↑				↑
	}	}						}		}	}			
	\$	i						m	p	s				

# Rechts-Induktions-Scan

$T =$   
 Type :  
 LMS :

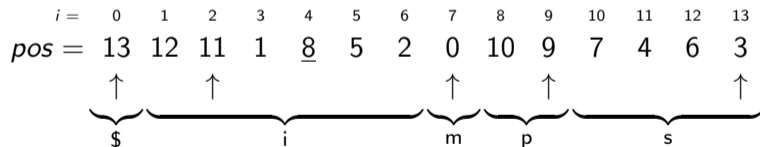
0	1
01234567890123	
miississippi\$	
LSSLLSLLSLLLLS	
*	* * *



# Rechts-Induktions-Scan

$T =$

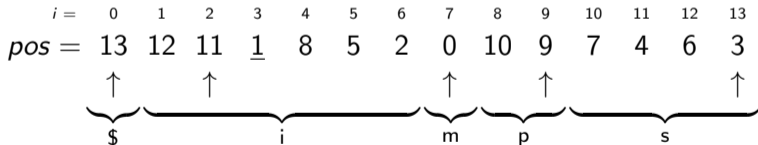
	0												1	
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
	m	i	s	s	i	s	s	i	p	p	i	i	\$	
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L
LMS :		*					*		*		*		*	



# Rechts-Induktions-Scan

$T =$   
 Type :  
 LMS :

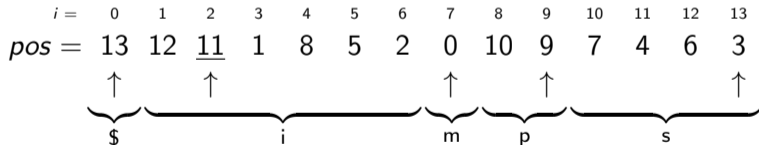
0	1												
0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	i	i	s	s	s	i	p	p	i	i	\$		
L	S	S	L	L	S	L	L	S	L	L	L	L	L
*	*	*	*	*	*	*	*	*	*	*	*	*	*



# Rechts-Induktions-Scan

$T =$

	0		1												
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	
	m	i	s	s	i	s	s	i	p	p	i	i	\$		
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L	S
LMS :			*				*		*		*			*	



# Rechts-Induktions-Scan

$T =$

	0												1	
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
	m	i	s	s	i	s	s	i	p	p	i	i	\$	
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L
LMS :		*					*		*		*		*	

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$pos =$	13	<u>12</u>	11	1	8	5	2	0	10	9	7	4	6	3	
	↑		↑					↑		↑				↑	
	}		}					}		}	}				
	\$	i					m	p	s						

# Rechts-Induktions-Scan

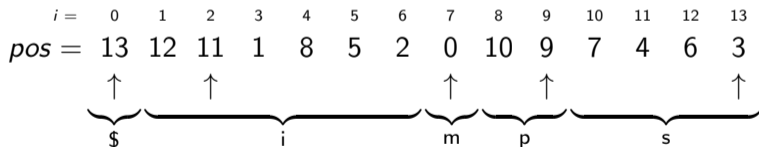
$T =$

	0												1		
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	
	m	i	s	s	i	s	s	i	p	p	i	i	\$		
Type :	L	S	S	L	L	S	L	L	S	L	L	L	L	L	S
LMS :		*					*		*		*		*		*

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	<u>13</u>	12	11	1	8	5	2	0	10	9	7	4	6	3
	↑		↑					↑		↑				↑
	⏟	⏟					⏟	⏟	⏟	⏟				
	\$	i					m	p	s					

# Rechts-Induktions-Scan

$T =$	0	1												
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	miissippii\$													
Type :	LSSLLSLLSLLLLS													
LMS :							*		*		*		*	



Beweis für Korrektheit des Rechts-Induktions-Scans ist analog zum Vorherigen.



- Sofern die sortierte Reihenfolge der LMS-Suffixe bekannt ist, kann man das vollständige Suffixarray korrekt induzieren.
- Die Laufzeit beträgt hierbei  $\mathcal{O}(n)$ .

- Sofern die sortierte Reihenfolge der LMS-Suffixe bekannt ist, kann man das vollständige Suffixarray korrekt induzieren.
- Die Laufzeit beträgt hierbei  $\mathcal{O}(n)$ .
  - Das einmalige Zählen der Buchstaben dauert  $\mathcal{O}(n)$ .
  - Das erstellen der Bucketanfangs- bzw. Endpositionen dauert  $\mathcal{O}(|\Sigma|)$ .
  - Der Links-Induktions-Scan dauert  $\mathcal{O}(n)$ .
  - Der Rechts-Induktions-Scan dauert  $\mathcal{O}(n)$ .

- Sofern die sortierte Reihenfolge der LMS-Suffixe bekannt ist, kann man das vollständige Suffixarray korrekt induzieren.
- Die Laufzeit beträgt hierbei  $\mathcal{O}(n)$ .
  - Das einmalige Zählen der Buchstaben dauert  $\mathcal{O}(n)$ .
  - Das erstellen der Bucketanfangs- bzw. Endpositionen dauert  $\mathcal{O}(|\Sigma|)$ .
  - Der Links-Induktions-Scan dauert  $\mathcal{O}(n)$ .
  - Der Rechts-Induktions-Scan dauert  $\mathcal{O}(n)$ .
- Wie kann man aber effizient die LMS-Suffixe sortieren?

# Reduktion des Problems

- LMS-Suffixe bestehen aus LMS-Substrings.
- Sortiere LMS-Substrings und stelle die durch ihren Rang in der sortierten Reihenfolge dar.

$$\begin{array}{rcl}
 T = & & \text{miississippi\$} \\
 LMS : & & * \quad * \quad * \quad * \\
 T_R = & & [1, 3, 2, 0]
 \end{array}$$

- $T_R$  repräsentiert LMS-Suffixe von  $T$  in derselben Reihenfolge.
- Wenn alle LMS-Substrings verschieden sind, sind wir “fertig”.
- Wenn es wiederholte Ränge gibt (gleiche LMS-Substrings), müssen wir wieder ein Suffixsortier-Problem lösen (Suffixarray berechnen), aber der String ist kürzer ... und zwar höchstens halb so lang wie vorher!

# Reduktion des Problems

- LMS-Suffixe bestehen aus LMS-Substrings.
- Sortiere LMS-Substrings und stelle die durch ihren Rang in der sortierten Reihenfolge dar.

$$\begin{array}{rcl}
 T = & & \text{miissippii\$} \\
 LMS : & & * \quad * \quad * \quad * \\
 T_R = & & [1, 3, 2, 0]
 \end{array}$$

- $T_R$  repräsentiert LMS-Suffixe von  $T$  in derselben Reihenfolge.
- Wenn alle LMS-Substrings verschieden sind, sind wir “fertig”.
- Wenn es wiederholte Ränge gibt (gleiche LMS-Substrings), müssen wir wieder ein Suffixsortier-Problem lösen (Suffixarray berechnen), aber der String ist kürzer

# Reduktion des Problems

- LMS-Suffixe bestehen aus LMS-Substrings.
- Sortiere LMS-Substrings und stelle die durch ihren Rang in der sortierten Reihenfolge dar.

$$\begin{array}{rcl}
 T = & & \text{miissippii\$} \\
 LMS : & & * \quad * \quad * \quad * \\
 T_R = & & [1, 3, 2, 0]
 \end{array}$$

- $T_R$  repräsentiert LMS-Suffixe von  $T$  in derselben Reihenfolge.
- Wenn alle LMS-Substrings verschieden sind, sind wir “fertig”.
- Wenn es wiederholte Ränge gibt (gleiche LMS-Substrings), müssen wir wieder ein Suffixsortier-Problem lösen (Suffixarray berechnen), aber der String ist kürzer ... und zwar höchstens halb so lang wie vorher!

- 1 Sortiere LMS-Substrings (siehe später)
- 2 Beginne mit Rang 0 (Wächter ist stets LMS-Substring)
- 3 Vergleiche nächsten LMS-Substring in sortierter Reihenfolge mit vorigem:  
Zwei LMS-Substrings haben genau dann den gleichen Rang, wenn sie dieselbe Länge, dieselbe Zeichensequenz und Positionstypensequenz haben.

# Details der Reduktion

- 1 Sortiere LMS-Substrings (siehe später)
- 2 Beginne mit Rang 0 (Wächter ist stets LMS-Substring)
- 3 Vergleiche nächsten LMS-Substring in sortierter Reihenfolge mit vorigem:  
Zwei LMS-Substrings haben genau dann den gleichen Rang, wenn sie dieselbe Länge, dieselbe Zeichensequenz und Positionstypensequenz haben.

Die Rangsequenz ist höchstens halb so lang wie der Ausgangstext.



# Details der Reduktion

- 1 Sortiere LMS-Substrings (siehe später)
- 2 Beginne mit Rang 0 (Wächter ist stets LMS-Substring)
- 3 Vergleiche nächsten LMS-Substring in sortierter Reihenfolge mit vorigem:  
Zwei LMS-Substrings haben genau dann den gleichen Rang, wenn sie dieselbe Länge, dieselbe Zeichensequenz und Positionstypensequenz haben.

Die Rangsequenz ist höchstens halb so lang wie der Ausgangstext.

- LMS-Teilstring hat die Typform  $S^+L^+S$ , wobei das letzte  $S$  gleichzeitig Beginn des nächsten LMS-Teilstrings ist.
- Es werden immer mindestens zwei Zeichen (SL) zu einem Rang reduziert.

# Details der Reduktion

- 1 Sortiere LMS-Substrings (siehe später)
- 2 Beginne mit Rang 0 (Wächter ist stets LMS-Substring)
- 3 Vergleiche nächsten LMS-Substring in sortierter Reihenfolge mit vorigem:  
Zwei LMS-Substrings haben genau dann den gleichen Rang, wenn sie dieselbe Länge, dieselbe Zeichensequenz und Positionstypensequenz haben.

Die Rangsequenz ist höchstens halb so lang wie der Ausgangstext.

- LMS-Teilstring hat die Typform  $S^+L^+S$ , wobei das letzte  $S$  gleichzeitig Beginn des nächsten LMS-Teilstrings ist.
- Es werden immer mindestens zwei Zeichen (SL) zu einem Rang reduziert.

Der Wächter bleibt in seinen Eigenschaften (lexikographisch kleinster LMS-Teilstring) bei der Reduktion erhalten.

# Details der Reduktion: Reduziertes Suffixarray

Gibt es gleiche Ränge, erfolgt ein rekursiver Aufruf.

Sind alle Ränge verschieden, kann durch einmaliges Durchlaufen von  $T_R$  das Suffixarray der LMS-Suffixe  $pos_R$  direkt bestimmt werden (inverse Permutation):  
 $pos_R[T_R[i]] := i$  für  $0 \leq i < |T_R|$ .



# Details der Reduktion: Reduziertes Suffixarray

Gibt es gleiche Ränge, erfolgt ein rekursiver Aufruf.

Sind alle Ränge verschieden, kann durch einmaliges Durchlaufen von  $T_R$  das Suffixarray der LMS-Suffixe  $pos_R$  direkt bestimmt werden (inverse Permutation):  
 $pos_R[T_R[i]] := i$  für  $0 \leq i < |T_R|$ .

	0	1		
	0	1	2	3
$T =$	m	i	s	s
$LMS :$	*	*	*	*
$T_R =$	[1,	3,	2,	0]
$pos_R =$	[ ,	0,	,	]

# Details der Reduktion: Reduziertes Suffixarray

Gibt es gleiche Ränge, erfolgt ein rekursiver Aufruf.

Sind alle Ränge verschieden, kann durch einmaliges Durchlaufen von  $T_R$  das Suffixarray der LMS-Suffixe  $pos_R$  direkt bestimmt werden (inverse Permutation):  
 $pos_R[T_R[i]] := i$  für  $0 \leq i < |T_R|$ .

	0	1		
	0	1	2	3
$T =$	m	i	s	s
$LMS :$	*	*	*	*
$T_R =$	[1,	3,	2,	0]
$pos_R =$	[ ,	0,	,	1]

# Details der Reduktion: Reduziertes Suffixarray

Gibt es gleiche Ränge, erfolgt ein rekursiver Aufruf.

Sind alle Ränge verschieden, kann durch einmaliges Durchlaufen von  $T_R$  das Suffixarray der LMS-Suffixe  $pos_R$  direkt bestimmt werden (inverse Permutation):  
 $pos_R[T_R[i]] := i$  für  $0 \leq i < |T_R|$ .

	0	1		
	0	1	2	3
$T =$	m	i	s	s
$LMS :$	*	*	*	*
$T_R =$	[1,	3,	2,	0]
$pos_R =$	[ ,	0,	2,	1]

# Details der Reduktion: Reduziertes Suffixarray

Gibt es gleiche Ränge, erfolgt ein rekursiver Aufruf.

Sind alle Ränge verschieden, kann durch einmaliges Durchlaufen von  $T_R$  das Suffixarray der LMS-Suffixe  $pos_R$  direkt bestimmt werden (inverse Permutation):  
 $pos_R[T_R[i]] := i$  für  $0 \leq i < |T_R|$ .

	0	1		
	0	1	2	3
$T =$	m	i	s	s
$LMS :$	*	*	*	*
$T_R =$	[1,	3,	2,	0]
$pos_R =$	[3,	0,	2,	1]



## Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

# Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

	01234567890123
$T =$	miississippii\$
$LMS :$	* * * *
$P =$	[1, 5, 8, 13]
$T_R =$	[1, 3, 2, 0]
$pos_R =$	[3, 0, 2, 1]
$seq =$	[ , , , ]

# Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

	01234567890123
$T =$	miississippii\$
$LMS :$	* * * *
$P =$	[1, 5, 8, 13]
$T_R =$	[1, 3, 2, 0]
$pos_R =$	[3, 0, 2, 1]
$seq =$	[13, , , ]

# Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

	01234567890123
$T =$	miississippii\$
$LMS :$	* * * *
$P =$	[1, 5, 8, 13]
$T_R =$	[1, 3, 2, 0]
$pos_R =$	[3, 0, 2, 1]
$seq =$	[13, 1, , ]

# Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

	01234567890123
$T =$	miississippii\$
$LMS :$	* * * *
$P =$	[1, 5, 8, 13]
$T_R =$	[1, 3, 2, 0]
$pos_R =$	[3, 0, 2, 1]
$seq =$	[13, 1, 8, ]

# Details: Reihenfolge der LMS-Suffixe bestimmen

Ist  $pos_R$  rekursiv oder direkt berechnet, wird die Folge  $seq$  der lexikographisch sortierten Startpositionen der LMS-Suffixe berechnet:

$$seq[i] = P[pos_R[i]],$$

wobei  $P$  die aufsteigenden Startpositionen der LMS-Suffixe im Text enthält.

	01234567890123
$T =$	miississippii\$
$LMS :$	* * * *
$P =$	[1, 5, 8, 13]
$T_R =$	[1, 3, 2, 0]
$pos_R =$	[3, 0, 2, 1]
$seq =$	[13, 1, 8, 5]

# Sortieren der LMS-Substrings

- Es bleibt die Frage offen, wie die LMS-Substrings sortiert werden.

# Sortieren der LMS-Substrings

- Es bleibt die Frage offen, wie die LMS-Substrings sortiert werden.
- LMS-Substrings haben die Form  $S^+L^+S$ .
- Es genügt, die LMS-Startpositionen in beliebiger Reihenfolge (aufsteigend) an das Ende ihrer Buckets in ein leeres Suffixarray zu schreiben.
- Ein Links-Induktions-Scan und Rechts-Induktions-Scan sortiert die LMS-Substrings.  
(Gleiche Substrings können in beliebiger Reihenfolge stehen.)



- Es bleibt die Frage offen, wie die LMS-Substrings sortiert werden.
- LMS-Substrings haben die Form  $S^+L^+S$ .
- Es genügt, die LMS-Startpositionen in beliebiger Reihenfolge (aufsteigend) an das Ende ihrer Buckets in ein leeres Suffixarray zu schreiben.
- Ein Links-Induktions-Scan und Rechts-Induktions-Scan sortiert die LMS-Substrings.  
(Gleiche Substrings können in beliebiger Reihenfolge stehen.)
- Zu Beginn alle Suffixe der Länge 1 ( $S$ ) der Substrings korrekt sortiert.
- Gemäß dem Beweis für den Links-Induktions-Scan sind nach dessen Ausführung alle Substrings vom Typ  $L^+S$  korrekt sortiert.
- Nach dem Rechts-Induktions-Scan sind Substrings vom Typ  $S^+L^+S$  sortiert.

Sei  $T(n)$  die Zeit, um ein Suffixarray eines Textes der Länge  $n$  mit SAIS zu erstellen.

$$\begin{aligned}T(1) &= \mathcal{O}(1); \\T(n) &\leq c_1 n + T(n/2) + c_2 n \\&= Cn + T(n/2) \\&\leq Cn + Cn/2 + T(n/4) \\&\leq Cn(1 + 1/2 + 1/4 + \dots) + T(1) \\&= 2Cn + \mathcal{O}(1) = \mathcal{O}(n).\end{aligned}$$

Also hat SAIS die Laufzeit  $\mathcal{O}(n)$ .

# Anwendung: Mustersuche mit Suffixarrays

- Binäre Suche
- Pro Binärschritt  $\mathcal{O}(m)$  Vergleiche
- Die Gesamtlaufzeit beträgt also  $\mathcal{O}(m \log(n))$ .
- Um alle Vorkommen zu finden, muss mit zwei binären Suchen zuerst die linke Grenze und danach die rechte Grenze im Suffixarray gefunden werden.

$p =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$T =$	m	i	i	s	s	i	s	s	i	p	p	i	i	\$

$r =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pos =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3
	⏟	⏟			⏟			⏟	⏟		⏟			
	\$	i			m			p	s					

Suffixarray bildet Blattebene des Suffixbaums ab, nicht aber Baumstruktur!

- Suffixarray und Text bestimmen Baumstruktur, stellen sie aber nicht explizit dar.
- Stringtiefe innerer Knoten “zwischen” zwei Blättern?

- Suffixarray und Text bestimmen Baumstruktur, stellen sie aber nicht explizit dar.
- Stringtiefe innerer Knoten “zwischen” zwei Blättern?
- Gemeinsame Präfixe!
- $lcp[r] := \max\{|s| \mid s \text{ Präfix von } T[pos[r] :] \text{ und } T[pos[r-1] :]\}$   
Länge des längsten gemeinsamen Präfixes des lexikographisch  $r$ -t kleinsten Suffixes und seines lexikographischen Vorgängers

$r$	$pos[r]$	$lcp[r]$	$T[pos[r] :]$
0	13	-	\$
1	12	0	i\$
2	11	1	ii\$
3	1	2	iississippii\$
4	8	1	ippii\$
5	5	1	issippii\$
6	2	4	issippii\$
7	0	0	mississippii\$
8	10	0	pii\$
9	9	1	ppii\$
10	7	0	sippii\$
11	4	2	sissippii\$
12	6	1	ssippii\$
13	3	3	ssissippii\$

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen?



# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen? Gesucht: Rang  $r$ , so dass  $pos[r] = 0$

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen? Gesucht: Rang  $r$ , so dass  $pos[r] = 0$
- Wo beginnt das mit  $T[p : ]$  zu vergleichende Suffix?

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen? Gesucht: Rang  $r$ , so dass  $pos[r] = 0$
- Wo beginnt das mit  $T[p : ]$  zu vergleichende Suffix? Rang  $r$  mit  $pos[r] = p - 1$

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen? Gesucht: Rang  $r$ , so dass  $pos[r] = 0$
- Wo beginnt das mit  $T[p : ]$  zu vergleichende Suffix? Rang  $r$  mit  $pos[r] = p - 1$
- Das nächste Suffix im Text  $T[1 : ]$  muss mindestens einen lcp-Wert von  $\ell - 1$  haben (entspricht Folgen eines Suffixlinks im Baum)
- Diese  $\ell - 1$  Positionen müssen nicht verglichen werden.
- Man kann direkt ab der  $\ell$ -ten Position vergleichen.

# Konstruktion des lcp-Arrays

- Naiv: sortierte Suffixe in lexikographischer Reihenfolge mit ihren Vorgängern vergleichen. Laufzeit:  $\mathcal{O}(n^2)$ .
- Angenommen man beginnt mit dem längsten Suffix  $T = T[0 : ]$ , vergleicht dieses mit seinem lexikographischen Vorgänger und erhält den lcp-Wert  $\ell$ .
- Wo soll man ihn eintragen? Gesucht: Rang  $r$ , so dass  $pos[r] = 0$
- Wo beginnt das mit  $T[p : ]$  zu vergleichende Suffix? Rang  $r$  mit  $pos[r] = p - 1$
- Das nächste Suffix im Text  $T[1 : ]$  muss mindestens einen lcp-Wert von  $\ell - 1$  haben (entspricht Folgen eines Suffixlinks im Baum)
- Diese  $\ell - 1$  Positionen müssen nicht verglichen werden.
- Man kann direkt ab der  $\ell$ -ten Position vergleichen.
- Beantwortung der Ranganfragen durch inverse Permutation  $rank$  von  $pos$ :  
Setze  $rank[pos[p]] := p$  für alle  $p = 0, \dots, n - 1$ .

# Linearzeit-Konstruktion des lcp-Arrays

```
1 def lcp_linear(pos, text):
2     n = len(pos)
3     lcp = [-1] * n
4     rank = [0] * n
5     for r in range(n):
6         rank[pos[r]] = r
7     lp = 0 # aktuelle Praefixlaenge
8     for p in range(n-1):
9         r = rank[p]
10        pp = pos[r-1]
11        while text[p+lp]==text[pp+lp]:
12            lp += 1
13        lcp[r] = lp
14        lp = max(lp-1, 0)
15    return lcp
```

- Konstruktion von *rank* dauert  $\mathcal{O}(n)$  Zeit
- Pro  $p$ -Schleifendurchlauf konstante Zeit bis auf while-Schleife
- Pro  $p$ -Schleifendurchlauf fällt  $lp$  um 1, aber nicht unter 0.
- while-Schleife erhöht  $lp$  um 1
- $lp$  kann nicht größer als  $n$  sein.
- Gesamtzahl der while-Durchläufer daher durch  $\mathcal{O}(n)$  beschränkt.

Gesamtlaufzeit ist  $\mathcal{O}(n)$ , also linear in Textlänge.

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13		iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		i <del>ss</del> iss...
	7	0	10		m <del>ii</del> ssi...
$pp = 2$	8	10	4		pii\$
$p = 0$	9	9	9		ppii\$
	10	7	8		sippii...
$lp = 0$	11	4	2		sissip...
	12	6	1		ssippii...
	13	3	0		ssissi...



# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13		iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 11$	8	10	4		p ii\$
$p = 1$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 0$	11	4	2		sissip...
	12	6	1		ssippii...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13		iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 11$	8	10	4		p ii\$
$p = 1$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 1$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13		iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 11$	8	10	4		p ii\$
$p = 1$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 2$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		i <span style="color:red">s</span> sipp...
	6	2	12		i <span style="color:red">s</span> sisss...
	7	0	10	0	miissi...
$pp = 5$	8	10	4		p <i>i</i> i\$
$p = 2$	9	9	9		pp <i>i</i> i\$
	10	7	8		sippii...
$lp = 1$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 5$	8	10	4		p ii\$
$p = 2$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 2$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 5$	8	10	4		p ii\$
$p = 2$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 3$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12		ississ...
	7	0	10	0	miissi...
$pp = 5$	8	10	4		p ii\$
$p = 2$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 4$	11	4	2		sissip...
	12	6	1		ssippi...
	13	3	0		ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 6$	8	10	4		p ii\$
$p = 3$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 3$	11	4	2		sissip...
	12	6	1		ssi <del>p</del> pi...
	13	3	0		ssi <del>s</del> si...



# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 7$	8	10	4		p <i>i</i> i\$
$p = 4$	9	9	9		pp <i>i</i> \$
	10	7	8		si <i>p</i> pi...
$lp = 2$	11	4	2		si <i>s</i> ip...
	12	6	1		ssi <i>p</i> pi...
	13	3	0	3	ssi <i>ssi</i> ...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5		issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 8$	8	10	4		p ii\$
$p = 5$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 1$	11	4	2	2	sissip...
	12	6	1		ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 4$	8	10	4		p ii\$
$p = 6$	9	9	9		pp ii\$
	10	7	8		sippii...
$lp = 0$	11	4	2	2	<b>s</b> iSSIP...
	12	6	1		<b>s</b> SiPPI...
	13	3	0	3	SSiSSI...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 4$	8	10	4		p <i>i</i> i\$
$p = 6$	9	9	9		pp <i>i</i> i\$
	10	7	8		sippii...
$lp = 1$	11	4	2	2	s <i>i</i> ssip...
	12	6	1		ss <i>i</i> ppi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 9$	8	10	4		p <i>i</i> i\$
$p = 7$	9	9	9		<b>p</b> pii\$
	10	7	8		<b>s</b> ippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11		ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 1$	8	10	4		p ii\$
$p = 8$	9	9	9		pp ii\$
	10	7	8	0	sippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	i <span style="color:red">i</span> ssis...
	4	8	11		i <span style="color:red">p</span> pii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 1$	8	10	4		pii\$
$p = 8$	9	9	9		ppii\$
	10	7	8	0	sippii...
$lp = 1$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 10$	8	10	4		pii\$
$p = 9$	9	9	9		ppii\$
	10	7	8	0	sippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...



# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 10$	8	10	4		pii\$
$p = 9$	9	9	9		ppii\$
	10	7	8	0	sippii...
$lp = 1$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 0$	8	10	4		pii\$
$p = 10$	9	9	9	1	ppii\$
	10	7	8	0	sippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] : ]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 12$	8	10	4	0	p ii\$
$p = 11$	9	9	9	1	pp ii\$
	10	7	8	0	sippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6		ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 12$	8	10	4	0	p ii\$
$p = 11$	9	9	9	1	pp ii\$
	10	7	8	0	sippii...
$lp = 1$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3		i\$
	2	11	6	1	ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp = 13$	8	10	4	0	p ii\$
$p = 12$	9	9	9	1	pp ii\$
	10	7	8	0	sippii...
$lp = 0$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Beispiel: Linearzeit-Konstruktion des lcp-Arrays

	$p r$	$pos[r]$	$rank[p]$	$lcp[r]$	$T[pos[r] :]$
	0	13	7	-	\$
	1	12	3	0	i\$
	2	11	6	1	ii\$
	3	1	13	2	iissis...
	4	8	11	1	ippii\$
	5	5	5	1	issipp...
	6	2	12	4	ississ...
	7	0	10	0	miissi...
$pp =$	8	10	4	0	p ii\$
$p =$	9	9	9	1	pp ii\$
	10	7	8	0	sippii...
$lp =$	11	4	2	2	sissip...
	12	6	1	1	ssippi...
	13	3	0	3	ssissi...

# Längster wiederholter Teilstring (LRS)

Um den LRS in Text  $T$  zu finden, muss der Index  $r^*$  des größten  $lcp$ -Werts gefunden werden.

$$r^* = \arg \max_{r < |T|} \{lcp[r]\}$$

$$LRS(T) = T[pos[r^*] : pos[r^*] + lcp[r^*]]$$

# Längster wiederholter Teilstring (LRS)

Um den LRS in Text  $T$  zu finden, muss der Index  $r^*$  des größten  $lcp$ -Werts gefunden werden.

$$r^* = \arg \max_{r < |T|} \{lcp[r]\}$$

$$LRS(T) = T[pos[r^*] : pos[r^*] + lcp[r^*]]$$

Beispiel: `miississippii$`

$$r^* = 6$$

$$pos[r^*] = 2$$

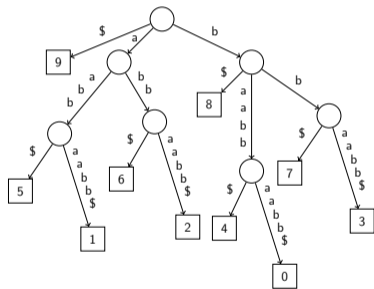
$$lcp[r^*] = 4$$

$$LRS(T) = T[2 : 2 + 4] = \text{issi}$$



# Kürzester eindeutiger Teilstring (SUS)

Beispiel:  $T = \text{baabbaabb}\$$



$r$	$pos[r]$	$LCP[r]$	$T[pos[r] :]$
0	9	-	$\$$
1	5	0	$\text{aabb}\$$
2	1	4	$\text{aabbaabb}\$$
3	6	1	$\text{abb}\$$
4	2	3	$\text{abbaabb}\$$
5	8	0	$\text{b}\$$
6	4	1	$\text{baabb}\$$
7	0	5	$\text{baabbaabb}\$$
8	7	1	$\text{bb}\$$
9	3	2	$\text{bbaabb}\$$

# Kürzester eindeutiger Teilstring (SUS)

- Stringtiefe des inneren Knoten, von dem Blatt  $p$  ausgeht, lässt sich ermitteln, indem das Maximum der lcp-Werte genommen wird, die Position  $p$  betrachten. Sei also  $r := \text{rank}[p]$ .
- Eindeutig für Suffix  $p$  ist also Stringlänge  $ulen(r) := 1 + \max(\text{lcp}[r], \text{lcp}[r + 1])$ .

# Kürzester eindeutiger Teilstring (SUS)

- Stringtiefe des inneren Knoten, von dem Blatt  $p$  ausgeht, lässt sich ermitteln, indem das Maximum der lcp-Werte genommen wird, die Position  $p$  betrachten. Sei also  $r := \text{rank}[p]$ .
- Eindeutig für Suffix  $p$  ist also Stringlänge  $ulen(r) := 1 + \max(\text{lcp}[r], \text{lcp}[r + 1])$ .
- Gesucht ist ein Teilstring minimaler Stringtiefe, der nicht mit dem Wächter endet.
- Sei  $r^* := \operatorname{argmin}_{r < |T|} \{ulen(r) \mid \text{pos}[r] + ulen(r) < |T|\}$ .

# Kürzester eindeutiger Teilstring (SUS)

- Stringtiefe des inneren Knoten, von dem Blatt  $p$  ausgeht, lässt sich ermitteln, indem das Maximum der lcp-Werte genommen wird, die Position  $p$  betrachten. Sei also  $r := \text{rank}[p]$ .
- Eindeutig für Suffix  $p$  ist also Stringlänge  $\text{ulen}(r) := 1 + \max(\text{lcp}[r], \text{lcp}[r + 1])$ .
- Gesucht ist ein Teilstring minimaler Stringtiefe, der nicht mit dem Wächter endet.
- Sei  $r^* := \operatorname{argmin}_{r < |T|} \{ \text{ulen}(r) \mid \text{pos}[r] + \text{ulen}(r) < |T| \}$ .
- Dann ist  $\text{SUS}(T) = T[\text{pos}[r^*] : \text{pos}[r^*] + \text{ulen}(r^*)]$ .

$T = \text{baabbaabb}\$$ :

$r^* = 9 \rightarrow \text{pos}[9] = 3, \text{ulen}(9) = 3 \rightarrow \text{sus}(T) = T[3 : 3 + 3] = \text{bba}$

# Längster gemeinsamer Teilstring (LCF)

- Gegeben seien die Texte  $T_1$  und  $T_2$ .
- Gesucht ist der längste gemeinsame Teilstring von  $T_1$  und  $T_2$ .

# Längster gemeinsamer Teilstring (LCF)

- Gegeben seien die Texte  $T_1$  und  $T_2$ .
- Gesucht ist der längste gemeinsame Teilstring von  $T_1$  und  $T_2$ .
- Sei der verallgemeinerte Text  $T := T_1\$_1T_2\$_2$  mit  $\$_1 < \$_2$ .
- Alle Suffixe, die vor  $\$_1$  anfangen, werden mit 1 beschriftet, die anderen mit 2.
- Sei  $label[r]$  die Beschriftung von  $pos[r]$ .

# Längster gemeinsamer Teilstring (LCF)

- Gegeben seien die Texte  $T_1$  und  $T_2$ .
- Gesucht ist der längste gemeinsame Teilstring von  $T_1$  und  $T_2$ .
- Sei der verallgemeinerte Text  $T := T_1\$_1T_2\$_2$  mit  $\$_1 < \$_2$ .
- Alle Suffixe, die vor  $\$_1$  anfangen, werden mit 1 beschriftet, die anderen mit 2.
- Sei  $label[r]$  die Beschriftung von  $pos[r]$ .
- Gesucht ist  $r^* = \arg \max_{r < |T|} \{lcp[r] \mid label[r] \neq label[r - 1]\}$ .

# Längster gemeinsamer Teilstring (LCF)

Beispiel:  $T = \text{baabb\#aaba\$}$

$r$	$pos[r]$	$lcp[r]$	$label[r]$	$T[pos[r] :]$
0	5	-	1	\#aaba\\$
1	10	0	2	\\$
2	9	0	2	a\\$
3	6	1	2	aaba\\$
4	1	3	1	aabb\#aaba\\$
5	7	1	2	aba\\$
6	2	2	1	abb\#aaba\\$
7	4	0	1	b\#aaba\\$
8	8	1	2	ba\\$
9	0	2	1	baabb\#aaba\\$
10	3	1	1	bb\#aaba\\$



# Längster gemeinsamer Teilstring (LCF)

Beispiel:  $T = \text{baabb\#aaba\$}$

$r$	$pos[r]$	$lcp[r]$	$label[r]$	$T[pos[r] : ]$
0	5	-	1	\#aaba\\$
1	10	0	2	\\$
2	9	0	2	a\\$
3	6	1	2	aaba\\$
4	1	3	1	aabb\#aaba\\$
5	7	1	2	aba\\$
6	2	2	1	abb\#aaba\\$
7	4	0	1	b\#aaba\\$
8	8	1	2	ba\\$
9	0	2	1	baabb\#aaba\\$
10	3	1	1	bb\#aaba\\$

$r^* = 4 \rightarrow pos[4] = 1, lcp[4] = 3 \rightarrow T[1 : 1 + 3] = \text{aab}$

- Der SAIS-Algorithmus erstellt ein Suffixarray in  $\mathcal{O}(n)$  Zeit.
- Verschiedene Fragestellungen können mit dem Suffixarray und lcp-Array effizient beantwortet werden:
  - Kommt  $P$  in  $T$  vor? Wenn ja, wo?  $\mathcal{O}(m \log n)$  Zeit.
  - Welcher ist der längste wiederholte Teilstring im Text  $T$ ?
  - Welcher ist der kürzeste eindeutige Teilstring im Text  $T$ ?
  - Welcher ist der längste gemeinsame Teilstring der beiden Texte  $T_1$  und  $T_2$ ?
- Der Speicherverbrauch liegt bei  $2 \cdot (n \lceil \log(n) \rceil)$  Bits.