technische universität
dortmund

Algorithmen auf Sequenzen

Volltext-Indizes - Teil 2

Dominik Kopczynski

Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik
TU Dortmund

Überblick

- Mit Hilfe eines Suffix-Trees kann ein Suffix-Array erstellt werden, indem in lexographischer Reihenfolge die Einträge in den Blättern aufgelistet werden.
- Bei großen Texten kann es jedoch sein, dass der Speicherverbrauch für einen Tree zu hoch.

Überblick

- Mit Hilfe eines Suffix-Trees kann ein Suffix-Array erstellt werden, indem in lexographischer Reihenfolge die Einträge in den Blättern aufgelistet werden.
- Bei großen Texten kann es jedoch sein, dass der Speicherverbrauch für einen Tree zu hoch.
- Auf einem Suffix-Array lässt sich mit binärer Suche ein Muster finden.
- Mit einem weiteren Hilfsarray lassen sich auch die anderen bekannten Anfragen (längster wiederholter Teilstring, etc.) auf einem Suffix-Array effizient abfragen.

Suffix-Array Konstruktion (naiv)

```
def build_suffixarray_naive(T):
    n = len(T)
    return sorted(range(n), key = lambda i: T[i:])
```

Ist es wirklich so einfach, wo ist der Haken?

Suffix-Array Konstruktion (naiv)

```python
def build_suffixarray_naive(T):
    n = len(T)
    return sorted(range(n), key = lambda i: T[i:])
```

Ist es wirklich so einfach, wo ist der Haken?

Dieser Ansatz hat eine $\mathcal{O}(n^2 \log n)$ Laufzeit!

Zur Erinnerung, mit dem Ukkonen-Algorithmus konnte man einen Suffix-Tree und das Suffix-Array in $\mathcal{O}(n)$ erstellen.

Suffix-Array Konstruktion (naiv)

```
def build_suffixarray_naive(T):
    n = len(T)
    return sorted(range(n), key = lambda i: T[i:])
```

Ist es wirklich so einfach, wo ist der Haken?

Dieser Ansatz hat eine $\mathcal{O}(n^2 \log n)$ Laufzeit!

Zur Erinnerung, mit dem Ukkonen-Algorithmus konnte man einen Suffix-Tree und das Suffix-Array in $\mathcal{O}(n)$ erstellen.

Mit dem Suffix-Array Induced Sorting (SAIS) Algorithmus kann das Suffix-Array in $\mathcal{O}(n)$ erstellt werden.

Beispiel eines Suffix-Array

$i =$	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	$
$pos =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3

Beispiel eines Suffix-Array

$$\begin{array}{rcccccccccccccc} i = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ T = & \mathrm{m} & \mathrm{i} & \mathrm{i} & \mathrm{s} & \mathrm{s} & \mathrm{i} & \mathrm{s} & \mathrm{s} & \mathrm{i} & \mathrm{p} & \mathrm{p} & \mathrm{i} & \mathrm{i} & \$ \end{array}$$

$$pos = \underbrace{13}_{\$}\ \underbrace{12\quad 11\quad 1\quad 8\quad 5\quad 2}_{\mathrm{i}}\ \underbrace{0}_{\mathrm{m}}\ \underbrace{10\quad 9}_{\mathrm{p}}\ \underbrace{7\quad 4\quad 6\quad 3}_{\mathrm{s}}$$

Aufteilung der Startbuchstaben der Suffixe in Buckets.

Eigenschaften eines Suffix-Arrays

Bei Teilstrings der Form $c_i \geq c_{i+1} \geq \cdots \geq c_j$ (z.B.: ...dccbba...) kommen die Indizes in umgekehter Reihenfolge im Suffix-Array vor, also z.B.: $\ldots j \ldots i+2 \ldots i+1 \ldots i \ldots$

Eigenschaften eines Suffix-Arrays

Bei Teilstrings der Form $c_i \geq c_{i+1} \geq \cdots \geq c_j$ (z.B.: ...dccbba...) kommen die Indizes in umgekehter Reihenfolge im Suffix-Array vor, also z.B.: $\ldots j \ldots i+2 \ldots i+1 \ldots i \ldots$

Bei Teilstrings der Form $c_i \leq c_{i+1} \leq \cdots \leq c_j$ (z.B.: ...abbccd...) kommen die Indizes in der selben Reihenfolge im Suffix-Array vor, also z.B.: $\ldots i \ldots i+1 \ldots i+2 \ldots j \ldots$

Eigenschaften eines Suffix-Arrays

Lemma

Für alle $\sigma_i \in T$ *gilt: wenn sich* σ_i *in einer aufsteigenden Reihenfolge befindet, kommt Index i im* σ*-Bucket nach allen* σ_j *vor, die sich in einer absteigenden Reihenfolge befinden.*

Eigenschaften eines Suffix-Arrays

Lemma

Für alle $\sigma_i \in T$ gilt: wenn sich σ_i in einer aufsteigenden Reihenfolge befindet, kommt Index i im σ-Bucket nach allen σ_j vor, die sich in einer absteigenden Reihenfolge befinden.

Beispiel für den Buchstaben i:

$$\begin{array}{rcccccccccccccc} i = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ T = & \mathrm{m} & \mathrm{i}_* & \mathrm{i}_* & \mathrm{s} & \mathrm{s} & \mathrm{i}_* & \mathrm{s} & \mathrm{s} & \mathrm{i}_* & \mathrm{p} & \mathrm{p} & \mathrm{i} & \mathrm{i} & \$ \end{array}$$

$$pos = \underbrace{13}_{\$}\;\;\underbrace{12\;\;11\;\;\underset{*}{1}\;\;\underset{*}{8}\;\;\underset{*}{5}\;\;\underset{*}{2}}_{\mathrm{i}}\;\;\underbrace{0}_{\mathrm{m}}\;\;\underbrace{10\;\;9}_{\mathrm{p}}\;\;\underbrace{7\;\;4\;\;6\;\;3}_{\mathrm{s}}$$

L-Type und S-Type

Definition (L-Type Zeichen)

Ein Zeichen $T[i]$ sei *L-Type* (larger), wenn sein Nachfolger $T[i+1]$ lexikographisch kleiner ist oder bei Gleichheit auch L-Type ist.

L-Type und S-Type

Definition (L-Type Zeichen)

Ein Zeichen $T[i]$ sei *L-Type* (larger), wenn sein Nachfolger $T[i+1]$ lexikographisch kleiner ist oder bei Gleichheit auch L-Type ist.

Definition (S-Type Zeichen)

Ein Zeichen $T[i]$ sei *S-Type* (smaller), wenn sein Nachfolger $T[i+1]$ lexikographisch größer ist oder bei Gleichheit auch S-Type ist.

Der Sentinal \$ ist per Definition S-Type.

LMS-Type

Definition (LMS-Type Zeichen)

Ein Zeichen $T[i]$ sei *LMS-Type* (left-most-smaller), wenn $T[i]$ S-Type ist und sein Vorgänger $T[i-1]$ L-Type ist.

LMS-Type

Definition (LMS-Type Zeichen)

Ein Zeichen $T[i]$ sei *LMS-Type* (left-most-smaller), wenn $T[i]$ S-Type ist und sein Vorgänger $T[i-1]$ L-Type ist.

Beispiel:

```
   T =    miississippii$
Type :    LSSLLSLLSLLLLS
 LMS :     *   *  *    *
```

LMS-Type

Definition (LMS-Type Zeichen)

Ein Zeichen $T[i]$ sei *LMS-Type* (left-most-smaller), wenn $T[i]$ S-Type ist und sein Vorgänger $T[i-1]$ L-Type ist.

Beispiel:

$T =$	m	i	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 :		$\star$				$\star$			$\star$					$\star$

Die LMS-Type Zeichen stellen jeweils das Ende einer absteigenden Reihenfolge dar.

LMS-Suffix und LMS-Substring

Definition (LMS-Suffix)

Ein Suffix $T[i:]$ sei ein *LMS-Suffix*, wenn sein Anfangszeichen $T[i]$ LMS-Type ist.

LMS-Suffix und LMS-Substring

Definition (LMS-Suffix)

Ein Suffix $T[i:]$ sei ein *LMS-Suffix*, wenn sein Anfangszeichen $T[i]$ LMS-Type ist.

Definition (LMS-Substring)

Ein Substring $T[i:j+1]$ sei ein *LMS-Substring*, wenn seine Anfangs- und Endzeichen $T[i]$, $T[j]$ LMS-Type sind und alle Zeichen $T[k]$ mit $i < k < j$ nicht LMS-Type sind. Der Sentinal gilt als einbuchstabiger LMS-Substring.

Grundannahmen des SAIS-Algorithmus

Lemma

Wenn alle LMS-Suffixe in sortierter Reihenfolge an den Enden ihrer jeweiligen Buckets stehen, dann werden alle L-Type Zeichen durch einen Links-Induktions-Scan an ihre korrekte Position im Suffix-Array sortiert.

Grundannahmen des SAIS-Algorithmus

Lemma

Wenn alle LMS-Suffixe in sortierter Reihenfolge an den Enden ihrer jeweiligen Buckets stehen, dann werden alle L-Type Zeichen durch einen Links-Induktions-Scan an ihre korrekte Position im Suffix-Array sortiert.

Links-Induktions-Scan:

1. Initiiere ein Suffix-Array der Länge n mit jeweils -1.
2. Füge ans Ende aller jeweiligen Buckets die Indizes der sortierten LMS-Suffixe hinzu.

Grundannahmen des SAIS-Algorithmus

Lemma

Wenn alle LMS-Suffixe in sortierter Reihenfolge an den Enden ihrer jeweiligen Buckets stehen, dann werden alle L-Type Zeichen durch einen Links-Induktions-Scan an ihre korrekte Position im Suffix-Array sortiert.

Links-Induktions-Scan:

1. Initiiere ein Suffix-Array der Länge n mit jeweils -1.
2. Füge ans Ende aller jeweiligen Buckets die Indizes der sortierten LMS-Suffixe hinzu.
3. Scane das Suffix-Array von links nach rechts.
4. Für $SA[i] \neq -1$: Wenn $s = SA[i] - 1$ ist L-Type, füge s an die vorderste freie Stelle seines Buckets hinzu.

Links-Induktions-Scan

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

$$\begin{array}{rcccccccccccccc} i = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array}$$

$$pos = \underbrace{\underset{\uparrow}{\underline{13}}}_{\$}\;\underbrace{\underset{\uparrow}{-1}\;\;-1\;\;-1\;\;1\;\;8\;\;5}_{i}\;\underbrace{\underset{\uparrow}{-1}}_{m}\;\underbrace{\underset{\uparrow}{-1}\;\;-1}_{p}\;\underbrace{\underset{\uparrow}{-1}\;\;-1\;\;-1\;\;-1}_{s}$$

Links-Induktions-Scan

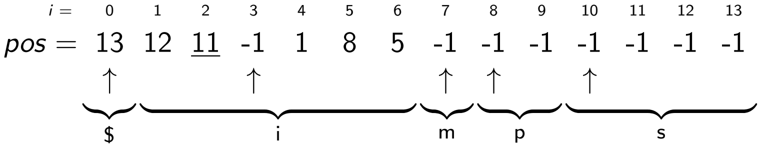
```
             0             1
             01234567890123
     T =     miississippii$
  Type :     LSSLLSLLSLLLLS
   LMS :      *   *  *    *
   seq =      [13, 1, 8, 5]
```

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

technische universität
dortmund

Links-Induktions-Scan

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



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

Links-Induktions-Scan

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

$$pos = \underbrace{\overset{0}{\underset{\uparrow}{13}}}_{\$}\;\underbrace{\overset{1}{12}\;\overset{2}{11}\;\overset{3}{\underset{\uparrow}{\underline{-1}}}\;\overset{4}{1}\;\overset{5}{8}\;\overset{6}{5}}_{i}\;\underbrace{\overset{7}{\underset{\uparrow}{-1}}}_{m}\;\underbrace{\overset{8}{10}\;\overset{9}{\underset{\uparrow}{-1}}}_{p}\;\underbrace{\overset{10}{\underset{\uparrow}{-1}}\;\overset{11}{-1}\;\overset{12}{-1}\;\overset{13}{-1}}_{s}$$

Links-Induktions-Scan

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

i =	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

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

$$\overset{i\,=}{pos} = \underbrace{\underset{\uparrow}{\overset{0}{13}}}_{\$}\ \underbrace{\overset{1}{12}\ \overset{2}{11}\ \underset{\uparrow}{\overset{3}{-1}}\ \overset{4}{1}\ \overset{5}{\underline{8}}\ \overset{6}{5}}_{i}\ \underbrace{\overset{7}{0}}_{m}\ \underbrace{\overset{8}{10}\ \underset{\uparrow}{\overset{9}{-1}}}_{p}\ \underbrace{\underset{\uparrow}{\overset{10}{-1}}\ \overset{11}{-1}\ \overset{12}{-1}\ \overset{13}{-1}}_{s}$$

Links-Induktions-Scan

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

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

Links-Induktions-Scan

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

$i =$ 0 1 2 3 4 5 6 7 8 9 10 11 12 13

$$pos = \underbrace{\underset{\uparrow}{13}}_{\$}\ \underbrace{12\ \ 11\ \ \underset{\uparrow}{-1}\ \ 1\ \ 8\ \ 5}_{\text{i}}\ \underbrace{\underline{0}}_{\text{m}}\ \underbrace{10\ \ \underset{\uparrow}{-1}}_{\text{p}}\ \underbrace{7\ \ 4\ \ \underset{\uparrow}{-1}\ \ -1}_{\text{s}}$$

Links-Induktions-Scan

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

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

Links-Induktions-Scan

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

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

Links-Induktions-Scan

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

$i =$	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	$\underline{7}$	4	-1	-1
	↑			↑									↑	
	$	i						m	p		s			

Links-Induktions-Scan

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

$i =$	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	$\underline{4}$	6	-1
	↑			↑										↑
	$	i						m	p		s			

Links-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		*				*			*					*

seq = [13, 1, 8, 5]

$i =$	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	$\underline{6}$	3
	↑			↑										
	$	i						m	p		s			

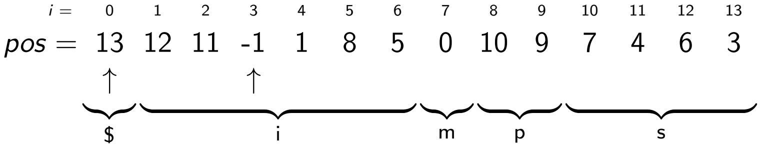
Links-Induktions-Scan

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

$i =$	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	$\underline{3}$
	↑			↑										
	$	i						m	p		s			

Links-Induktions-Scan

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



$i =$	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

Grundannahmen des SAIS-Algorithmus

Beweis

- *L-Type Zeichen können nur von LMS-Type oder anderen L-Type Zeichen eingetragen werden.*
- *Am Anfang ist nur Sentinal korrekt sortiert.*
- *Das Zeichen vor dem Sentinal wird auf jeden Fall korrekt eingetragen.*

Grundannahmen des SAIS-Algorithmus

Beweis

- *L-Type Zeichen können nur von LMS-Type oder anderen L-Type Zeichen eingetragen werden.*
- *Am Anfang ist nur Sentinal korrekt sortiert.*
- *Das Zeichen vor dem Sentinal wird auf jeden Fall korrekt eingetragen.*
- *Angenommen Suffix* $T[j:]$ *wird nach einem Index* i *mit* $i < j$ *in ein Bucket eingetragen, wobei aber* $T[i:] > T[j:]$ *gilt.*
- *In dem Fall muss* $T[i] = T[j]$ *gelten.*
- *Folglich hätte eine Fehlsortierung von* $T[j+1:]$ *und* $T[i+1:]$ *vorher eintreten müssen.* ↯ □

Grundannahmen des SAIS-Algorithmus

Lemma

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

Grundannahmen des SAIS-Algorithmus

Lemma

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

Rechts-Induktions-Scan:

1. Scane das vorhandene Suffix-Array von rechts nach links.
2. Wenn $s = SA[i] - 1$ ist S-Type, füge s an die hinterste freie Stelle seines Buckets hinzu.

Rechts-Induktions-Scan

```
          0         1
          01234567890123
T =       miississippii$
Type :    LSSLLSLLSLLLLS
LMS :      *   *  *    *
```

$i =$	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	$\underline{3}$
	↑						↑	↑		↑				↑
	$	i						m	p		s			

Rechts-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		*				*			*					*

$i =$	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	$\underline{6}$	3
	↑					↑		↑		↑				↑
	$	i						m	p		s			

Rechts-Induktions-Scan

	0 1
	01234567890123
$T =$	miissisissippii$
Type :	LSSLLSLLSLLLLS
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	9	7	$\underline{4}$	6	3
	↑				↑			↑		↑				↑
	$	i						m	p		s			

Rechts-Induktions-Scan

```
          0         1
          01234567890123
T =       miississippii$
Type :    LSSLLSLLSLLLLS
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	9	$\underline{7}$	4	6	3
	↑				↑			↑		↑				↑
	$	i						m	p		s			

Rechts-Induktions-Scan

```
                0             1
                01234567890123
      T =       miississippii$
  Type :        LSSLLSLLSLLLLS
   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	$\underline{9}$	7	4	6	3
	↑				↑			↑		↑				↑
	$	i						m	p		s			

Rechts-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		⋆				⋆			⋆					⋆

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

Rechts-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		*				*			*					*

$$
\begin{array}{r}
i = \\
pos =
\end{array}
\underbrace{\overset{0}{\underset{\uparrow}{13}}}_{\$}\;
\underbrace{\overset{1}{12}\;\overset{2}{11}\;\overset{3}{\underset{\uparrow}{\text{-1}}}\;\overset{4}{8}\;\overset{5}{5}\;\overset{6}{2}}_{i}\;
\underbrace{\overset{7}{\underset{\uparrow}{\underline{0}}}}_{m}\;
\underbrace{\overset{8}{10}\;\overset{9}{\underset{\uparrow}{9}}}_{p}\;
\underbrace{\overset{10}{7}\;\overset{11}{4}\;\overset{12}{6}\;\overset{13}{\underset{\uparrow}{3}}}_{s}
$$

Rechts-Induktions-Scan

```
              0         1
              01234567890123
       T =    miississippii$
    Type :    LSSLLSLLSLLLLS
     LMS :     *   *  *    *
```

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

Rechts-Induktions-Scan

```
          0         1
          01234567890123
  T =     miissississippii$
Type :    LSSLLSLLSLLLLS
 LMS :     *   *  *    *
```

$$pos = \underbrace{\overset{0}{\underset{\uparrow}{13}}}_{\$}\ \underbrace{\overset{1}{12}\ \overset{2}{\underset{\uparrow}{11}}\ \overset{3}{1}\ \overset{4}{8}\ \overset{5}{\underline{5}}\ \overset{6}{2}}_{i}\ \underbrace{\overset{7}{\underset{\uparrow}{0}}}_{m}\ \underbrace{\overset{8}{10}\ \overset{9}{\underset{\uparrow}{9}}}_{p}\ \underbrace{\overset{10}{7}\ \overset{11}{4}\ \overset{12}{6}\ \overset{13}{\underset{\uparrow}{3}}}_{s}$$

Rechts-Induktions-Scan

```
        0         1
        01234567890123
T =     miississippii$
Type :  LSSLLSLLSLLLLS
LMS :    *   *  *    *
```

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

Rechts-Induktions-Scan

```
        0         1
        01234567890123
T =     miississippii$
Type :  LSSLLSLLSLLLLS
LMS :    *   *  *    *
```

$$\overset{i\,=}{pos} = \underbrace{\underset{\uparrow}{\overset{0}{13}}}_{\$}\ \underbrace{\overset{1}{12}\ \underset{\uparrow}{\overset{2}{11}}\ \overset{3}{\underline{1}}\ \overset{4}{8}\ \overset{5}{5}\ \overset{6}{2}}_{i}\ \underbrace{\underset{\uparrow}{\overset{7}{0}}}_{m}\ \underbrace{\overset{8}{10}\ \underset{\uparrow}{\overset{9}{9}}}_{p}\ \underbrace{\overset{10}{7}\ \overset{11}{4}\ \overset{12}{6}\ \underset{\uparrow}{\overset{13}{3}}}_{s}$$

Rechts-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		$\star$				$\star$			$\star$					$\star$

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

Rechts-Induktions-Scan

	0										1			
	0	1	2	3	4	5	6	7	8	9	0	1	2	3
$T =$	m	i	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 :		*				*			*					*

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

Rechts-Induktions-Scan

```
            0         1
            01234567890123
$T =$       miississippii$
Type :      LSSLLSLLSLLLLS
LMS :        *   *  *    *
```

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

Rechts-Induktions-Scan

```
                0         1
                01234567890123
       T =      miississippii$
    Type :      LSSLLSLLSLLLLS
     LMS :       *   *  *    *
```

$$pos = \underbrace{\underset{\uparrow}{\overset{0}{13}}}_{\$}\ \underbrace{\overset{1}{12}\ \underset{\uparrow}{\overset{2}{11}}\ \overset{3}{1}\ \overset{4}{8}\ \overset{5}{5}\ \overset{6}{2}}_{i}\ \underbrace{\underset{\uparrow}{\overset{7}{0}}}_{m}\ \underbrace{\overset{8}{10}\ \underset{\uparrow}{\overset{9}{9}}}_{p}\ \underbrace{\overset{10}{7}\ \overset{11}{4}\ \overset{12}{6}\ \underset{\uparrow}{\overset{13}{3}}}_{s}$$

($i =$ 0 … 13 über den Einträgen)

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

Beobachtungen

- Sofern die sortierte Reihenfolge der LMS-Suffixe *seq* bekannt ist, kann man das korrekte Suffix-Array induzieren.
- Die Laufzeit beträgt hierbei $\mathcal{O}(n)$:

Beobachtungen

- Sofern die sortierte Reihenfolge der LMS-Suffixe *seq* bekannt ist, kann man das korrekte Suffix-Array 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 $\mathcal{O}(n)$.

Beobachtungen

- Sofern die sortierte Reihenfolge der LMS-Suffixe *seq* bekannt ist, kann man das korrekte Suffix-Array 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

- Die LMS-Substrings bilden Basis-Blöcke der LMS-Suffixe.
- Wenn es möglich ist die Blöcke durch eine kürzere Repräsentation darzustellen, z.B. ihren Rang, dann könnte das Problem mittels einem Divide-And-Conquer-Verfahren gelöst werden.
- Der Rang eines LMS-Substrings gibt seine lexikographische Reihenfolge unter allen LMS-Substrings im Text an.

Reduktion des Problems

- Die LMS-Substrings bilden Basis-Blöcke der LMS-Suffixe.
- Wenn es möglich ist die Blöcke durch eine kürzere Repräsentation darzustellen, z.B. ihren Rang, dann könnte das Problem mittels einem Divide-And-Conquer-Verfahren gelöst werden.
- Der Rang eines LMS-Substrings gibt seine lexikographische Reihenfolge unter allen LMS-Substrings im Text an.

Beispiel:

```
T   = miississippii$
LMS :  *   *  *    *
T_R =   [1, 3, 2, 0]
```

Bestimmung des Rangs

Definition (Bestimmung des Rangs)

Um die Reihenfolge zweier beliebiger LMS-Substrings zu bestimmen, werden beide Substrings von vorne nach hinten gelesen.

Bestimmung des Rangs

Definition (Bestimmung des Rangs)

Um die Reihenfolge zweier beliebiger LMS-Substrings zu bestimmen, werden beide Substrings von vorne nach hinten gelesen. Die Reihenfolge wird durch die lexikographische Reihenfolge des ersten nicht-matchenden Zeichens bestimmt.

Bestimmung des Rangs

Definition (Bestimmung des Rangs)

Um die Reihenfolge zweier beliebiger LMS-Substrings zu bestimmen, werden beide Substrings von vorne nach hinten gelesen. Die Reihenfolge wird durch die lexikographische Reihenfolge des ersten nicht-matchenden Zeichens bestimmt. Sollten alle Zeichen stimmen, gibt der Typ der Zeichen Aufschluss über die Reihenfolge, wobei S-Type Zeichen die höhere Priorität haben.

Bestimmung des Rangs

- Aus der Definition folgt, dass zwei LMS-Substring genau dann den selben Rang haben, wenn sie die selbe Länge, die selbe Anordnung der Zeichen und Anordnung der Typen haben.
- S-Type Zeichen bekommen eine höhere Priorität (S-Type > L-Type); zur Erinnerung: L-Type Zeichen kommen im Bucket immer vor S-Type Zeichen.
- $T[i:] > T[j:]$, wenn 1) $T[i] > T[j]$ oder 2) $T[i] = T[j]$ und $T[i], T[j]$ S-Type und L-Type sind.

Reduktion des Problems

- Angenommen die LMS-Substrings wären bereits sortiert.
- In dem Fall könnte der aktuelle Substring mit seinem Vorgänger verglichen und sein Rang bestimmt werden.

Reduktion des Problems

- Angenommen die LMS-Substrings wären bereits sortiert.
- In dem Fall könnte der aktuelle Substring mit seinem Vorgänger verglichen und sein Rang bestimmt werden.
- Da die Anzahl der LMS-Substrings auf $|T|/2$ begrenzt ist, ist das Rang-Array T_{R} maximal nur halb so groß wie T.

Beweis

Da für ein LMS-Type Zeichen links vom S-Type ein L-Type sein muss, muss ein LMS-Substring mindestens länge 3 haben.

Reduktion des Problems

- Angenommen die LMS-Substrings wären bereits sortiert.
- In dem Fall könnte der aktuelle Substring mit seinem Vorgänger verglichen und sein Rang bestimmt werden.
- Da die Anzahl der LMS-Substrings auf $|T|/2$ begrenzt ist, ist das Rang-Array T_R maximal nur halb so groß wie T.

Beweis

Da für ein LMS-Type Zeichen links vom S-Type ein L-Type sein muss, muss ein LMS-Substring mindestens länge 3 haben. Da die Substrings sich um ein Zeichen überschneiden, werden auf jeden Fall mindestens 2 Zeichen zu einem Rang reduziert. ☐

Reduktion des Problems

Lemma

Der Sentinal bleibt im reduzierten Rang-Array in seinen Eigenschaften erhalten.

Reduktion des Problems

Lemma

Der Sentinal bleibt im reduzierten Rang-Array in seinen Eigenschaften erhalten.

Beweis

Der Sentinal ist per Definition ein einelementiger LMS-Substring. Folglich bekommt dieser den Rang 0. Da der nächste LMS-Substring mit einem lexikographisch größerem Zeichen beginnt, bekommt dieser den Rang 1.

Reduktion des Problems

Lemma

Der Sentinal bleibt im reduzierten Rang-Array in seinen Eigenschaften erhalten.

Beweis

Der Sentinal ist per Definition ein einelementiger LMS-Substring. Folglich bekommt dieser den Rang 0. Da der nächste LMS-Substring mit einem lexikographisch größerem Zeichen beginnt, bekommt dieser den Rang 1. Da der Sentinal am Ende des Textes steht, steht im Rang-Array sein Rang (kleinstes einmaliges Zeichen) folglich auch am Ende. □

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

$$\begin{array}{r cccccccccccccc}
 & & \downarrow & & & & & & & & & & & & \downarrow^{*} \\
T = & \mathtt{m} & \mathtt{i} & \mathtt{i} & \mathtt{s} & \mathtt{s} & \mathtt{i} & \mathtt{s} & \mathtt{s} & \mathtt{i} & \mathtt{p} & \mathtt{p} & \mathtt{i} & \mathtt{i} & \mathtt{\$} \\
LMS: & & \star & & & & \star & & & \star & & & & & \star \\
T_{\mathrm{R}} = & \multicolumn{14}{r}{\mathtt{[\;\;,\quad\;,\quad\;,\;0]}}
\end{array}$$

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

$$\begin{array}{rl}
 & \downarrow^{*} \qquad\quad \downarrow \\
T = & \texttt{miississippii\$} \\
LMS: & \star \quad\ \ \star \quad \star \quad\ \ \star \\
T_{\mathrm{R}} = & \texttt{[1, \ \ , \ \ , 0]}
\end{array}$$

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

$$\begin{array}{rl} & \qquad\quad \downarrow^{*} \qquad\quad\; \downarrow \\ T = & \texttt{miississippii\$} \\ LMS: & \;\;\star \quad\;\; \star \quad \star \qquad \star \\ T_{\mathrm{R}} = & \qquad \texttt{[1,\ \ ,\ \ ,\ 0]} \end{array}$$

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array *P* gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der *j*-te Substring ist, kommt der Wert an die *j*-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

```
           ↓  ↓*
T =   miississippii$
LMS :  *   *  *    *
T_R =   [1,  , 2, 0]
```

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

$$\begin{array}{rl} & \qquad\qquad\ \ \downarrow\quad\downarrow^{*} \\ T = & \texttt{miississippii\$} \\ LMS: & \ \star\quad\ \ \star\quad\star\qquad\ \star \\ T_R = & \quad\texttt{[1,\ \ ,\ 2,\ 0]} \end{array}$$

Erstellen des Rang-Arrays

1. Zuerst werden alle Startpositionen der LMS-Substrings in einem Pointer-Array P gespeichert.
2. Der Sentinal bekommt den Rang 0 und da dieser der j-te Substring ist, kommt der Wert an die j-te Stelle im Rang-Array.
3. Nun wird der nächste Substring mit seinem Vorgänger verglichen. Wenn die Substrings sich unterscheiden, bekommt der nächste einen um 1 inkrementieren Rang, sonst den selben.

```
T   = miississippii$
LMS :  *   *  *    *
T_R =   [1, 3, 2, 0]
```

Reduktion des Problems

Beispiel für das Pointerarray:

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

Reduktion des Problems

Beispiel für das Pointerarray:

$$\begin{array}{rl}
 & \texttt{0\ \ \ \ \ \ \ \ \ 1} \\
 & \texttt{01234567890123} \\
T = & \texttt{miississippii\$} \\
LMS: & \texttt{\ }\star\texttt{\ \ \ }\star\texttt{\ \ }\star\texttt{\ \ \ \ }\star \\
P = & \texttt{[1, 5, 8, 13]} \\
T_{\mathrm{R}} = & \texttt{[1, 3, 2, 0]}
\end{array}$$

Lemma

Die relative Reihenfolge zweier Suffixe $T_R[i:]$, $T_R[j:]$ *in* T_R *ist die selbe wie die von* $T[P[i]:]$, $T[P[j]:]$ *in* T.

Reduktion des Problems

Beweis

- *Fall 1, $T_R[i] \neq T_R[j]$: Die Korrektheit erfolgt aus der Berechnung der Ränge, da die LMS-Substrings lexikographisch aufsteigend paarweise verglichen werden und bei Ungleichheit der aktuelle Substring einen höheren Rang als sein Vorgänger bekommt.*

Reduktion des Problems

Beweis

- *Fall 2,* $T_R[i] = T_R[j]$*: In dem Fall hängt die Reihenfolge vom ersten nachfolg. Mismatch ab, also* $T_R[i+k] \neq T_R[j+k]$*. Es gilt:* $T_R[i : i+k] = T_R[j : j+k]$*, daraus folgt:* $P[i+k] - P[i] = P[j+k] - P[j]$.

Reduktion des Problems

Beweis

- *Fall 2,* $T_R[i] = T_R[j]$*: In dem Fall hängt die Reihenfolge vom ersten nachfolg. Mismatch ab, also* $T_R[i+k] \neq T_R[j+k]$*. Es gilt:* $T_R[i:i+k] = T_R[j:j+k]$*, daraus folgt:* $P[i+k] - P[i] = P[j+k] - P[j]$*. Folglich haben* $T[P[i]:P[i+k]+1]$ *und* $T[P[j]:P[j+k]+1]$ *die selbe Länge.*

Reduktion des Problems

Beweis

- *Fall 2, $T_R[i] = T_R[j]$: In dem Fall hängt die Reihenfolge vom ersten nachfolg. Mismatch ab, also $T_R[i+k] \neq T_R[j+k]$. Es gilt: $T_R[i:i+k] = T_R[j:j+k]$, daraus folgt: $P[i+k] - P[i] = P[j+k] - P[j]$. Folglich haben $T[P[i]:P[i+k]+1]$ und $T[P[j]:P[j+k]+1]$ die selbe Länge. Dies zeigt, dass das Sortieren von $T_R[i:i+k+1]$ und $T_R[j:j+k+1]$ äquivalent zum Sortieren von $T[P[i]:P[i+k]+1]$ und $T[P[j]:P[j+k]+1]$ ist.* ☐

Reduktion des Problems

- Es wurde also gezeigt, dass das Sortieren der Suffixe in T_{R} äquivalent zum Sortieren der Suffixe in T ist.
- Das Problem reduziert sich auf höchstens die halbe Länge.

Reduktion des Problems

- Es wurde also gezeigt, dass das Sortieren der Suffixe in T_R äquivalent zum Sortieren der Suffixe in T ist.
- Das Problem reduziert sich auf höchstens die halbe Länge.
- Das Sortieren von T_R kann rekursiv mit $\max(T_R) + 1$ Buchstaben aufgerufen werden, wenn die Ränge nicht eindeutig sind.
- Andernfalls kann das Suffix-Array pos_R direkt aus T_R induziert werden.

Induktion des $\mathbf{pos}_\mathrm{R}$-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_\mathrm{R}[T_\mathrm{R}[i]] := i$ für alle $0 \leq i < |T_\mathrm{R}|$.

Induktion des pos_R-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_\mathrm{R}[T_\mathrm{R}[i]] := i$ für alle $0 \leq i < |T_\mathrm{R}|$.

	`0         1`
	`01234567890123`
$T =$	`miississippii$`
$LMS:$	` *   *  *    *`
$T_\mathrm{R} =$	`[1, 3, 2, 0]`
$pos_\mathrm{R} =$	`[ ,  ,  ,  ]`

Induktion des pos_R-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_\mathrm{R}[T_\mathrm{R}[i]] := i$ für alle $0 \leq i < |T_\mathrm{R}|$.

```
           0         1
           01234567890123
    T =    miississippii$
  LMS :     *   *  *    *
  T_R =      [1, 3, 2, 0]
pos_R =      [ , 0,  ,  ]
```

Induktion des pos_R-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_\text{R}[T_\text{R}[i]] := i$ für alle $0 \leq i < |T_\text{R}|$.

```
          0         1
          01234567890123
T     =   miississippii$
LMS   :    *   *  *    *
T_R   =     [1, 3, 2, 0]
pos_R =     [ , 0,  , 1]
```

Induktion des pos_R-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_R[T_R[i]] := i$ für alle $0 \leq i < |T_R|$.

	`0         1`
	`01234567890123`
$T =$	`miississippii$`
$LMS :$	` *   *  *    *`
$T_R =$	`  [1, 3, 2, 0]`
$pos_R =$	`  [ , 0, 2, 1]`

Induktion des pos_R-Arrays

Durch einmaliges Durchlaufen von T_R kann pos_R direkt induziert werden, sei $pos_\text{R}[T_\text{R}[i]] := i$ für alle $0 \leq i < |T_\text{R}|$.

	`0         1`
	`01234567890123`
$T =$	`miississippii$`
$LMS :$	` *   *  *    *`
$T_\text{R} =$	`  [1, 3, 2, 0]`
$pos_\text{R} =$	`  [3, 0, 2, 1]`

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_{R}-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe *seq* berechnet werden mit $seq[i] := P[pos_{\mathrm{R}}[i]]$.

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_R-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe *seq* berechnet werden mit $seq[i] := P[pos_R[i]]$.

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

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_R-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe *seq* berechnet werden mit $seq[i] := P[pos_R[i]]$.

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

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_R-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe *seq* berechnet werden mit $seq[i] := P[pos_\mathrm{R}[i]]$.

	`0         1`
	`01234567890123`
$T =$	`miississippii$`
$LMS:$	` *   *  *    *`
$P =$	`[1, 5, 8, 13]`
$T_\mathrm{R} =$	`[1, 3, 2, 0]`
$pos_\mathrm{R} =$	`[3, 0, 2, 1]`
$seq =$	`[13, 1,  ,  ]`

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_R-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe seq berechnet werden mit $seq[i] := P[pos_\mathrm{R}[i]]$.

```
         0         1
         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,  ]
```

Reihenfolge der LMS-Suffixe bestimmen

Ist das pos_R-Array durch den rekursiven Aufruf oder durch die Induktion erzeugt worden, kann die Reihenfolge der LMS-Suffixe *seq* berechnet werden mit $seq[i] := P[pos_\mathrm{R}[i]]$.

```
                 0         1
                 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

- Die größte Herausforderung, das Sortieren der LMS-Substrings, kann sehr elegant gelöst werden.
- Es reicht die LMS-Positionen in beliebiger Reihenfolge (aufsteigend nach Vorkommen im Text) an das Ende ihrer Buckets in ein leeres Suffix-Array zu schreiben.

Sortieren der LMS-Substrings

- Die größte Herausforderung, das Sortieren der LMS-Substrings, kann sehr elegant gelöst werden.
- Es reicht die LMS-Positionen in beliebiger Reihenfolge (aufsteigend nach Vorkommen im Text) an das Ende ihrer Buckets in ein leeres Suffix-Array zu schreiben.
- Erstaunlicherweise kann man wieder den Links-Induktions-Scan und Rechts-Induktions-Scan anwenden, um die Substrings zu sortieren.

Sortieren der LMS-Substrings

- LMS-Substrings haben die Form S^+L^+S.
- Nach der Initialisierung sind zumindest alle Suffixe der Länge 1 der Substrings korrekt sortiert.

Sortieren der LMS-Substrings

- LMS-Substrings haben die Form $\mathsf{S}^+\mathsf{L}^+\mathsf{S}$.
- Nach der Initialisierung sind zumindest alle Suffixe der Länge 1 der Substrings korrekt sortiert.
- Gemäß dem Beweis für den Links-Induktions-Scan sind nach dessen Ausführung alle Substrings der Form $\mathsf{L}^+\mathsf{S}$ korrekt sortiert.

Sortieren der LMS-Substrings

- LMS-Substrings haben die Form S^+L^+S.
- Nach der Initialisierung sind zumindest alle Suffixe der Länge 1 der Substrings korrekt sortiert.
- Gemäß dem Beweis für den Links-Induktions-Scan sind nach dessen Ausführung alle Substrings der Form L^+S korrekt sortiert.
- Analog sind nach dem Rechts-Induktions-Scan alle Substrings der Form S^+L^+S sortiert.
- Klar: Gleiche Substrings können in falscher Reihenfolge stehen.

Implementierung

```
def SAIS(T, K):
    if isinstance(T, str): T = bytes(T, 'utf-8')
    P, counts, types, LMS = init_aux_arrays(T, K)

    pos = init_suffixarray(T, P, counts)
    left_induction_scan(types, pos, T, counts)
    right_induction_scan(types, pos, T, counts)

    TR, max_rank = get_rankarray(T, pos, types, LMS)
    if max_rank < len(TR) - 1: posR = SAIS(TR, max_rank + 1)
    else: posR = induce_pos(TR)

    seq = [P[c] for c in posR]
    pos = init_suffixarray(T, seq, counts)
    left_induction_scan(types, pos, T, counts)
    right_induction_scan(types, pos, T, counts)

    return pos
```

Implementierung

```
def init_aux_arrays(T, K):
    counts, n = [0] * K, len(T)
    types, LMS = [False] * n, [False] * n
    # True = S-Type
    counts[T[-1]], types[-1] = 1, True

    for i in range(n - 2, -1, -1):
        counts[T[i]] += 1
        types[i] = T[i] < T[i + 1] \
            or (T[i] == T[i + 1] and types[i + 1])
        LMS[i + 1] = types[i + 1] and not types[i]

    P = [i for i in range(n) if LMS[i]]
    return P, counts, types, LMS
```

Implementierung

```python
def getBuckets(counts, start):
    bkt, cum_sum = list(counts), 0
    for i, v in enumerate(bkt):
        cum_sum += v
        bkt[i] = cum_sum - v * start
    return bkt

def init_suffixarray(T, seq, counts):
    pos = [-1] * len(T)
    # find ends of buckets, False => end positions
    bkt = getBuckets(counts, False)
    for c in seq[::-1]:
        bkt[T[c]] -= 1
        pos[bkt[T[c]]] = c
    return pos
```

Implementierung

```
def left_induction_scan(types, pos, T, counts):
    bkt = getBuckets(counts, True)
    for i in range(len(pos)):
        s = pos[i] - 1
        if s >= 0 and not types[s]:
            pos[bkt[T[s]]] = s
            bkt[T[s]] += 1

def right_induction_scan(types, pos, T, counts):
    bkt = getBuckets(counts, False)
    for i in range(len(pos) - 1, -1, -1):
        s = pos[i] - 1
        if s >= 0 and types[s]:
            bkt[T[s]] -= 1
            pos[bkt[T[s]]] = s
```

Implementierung

```python
def get_rankarray(T, pos, types, LMS):
    n, max_rank, prev = len(T), 0, pos[0]
    TR, nR = [-1] * n, 1
    TR[prev] = 0

    for p in pos[1:]:
        if not LMS[p]: continue
        curr, diff, nR = p, False, nR + 1
        for d in range(n):
            if T[curr + d] != T[prev + d] \
                or types[curr + d] != types[prev + d]:
                diff = True
                break
            elif d > 0 and (LMS[curr + d] or LMS[prev + d]):
                break
        max_rank += diff
        TR[curr], prev = max_rank, curr

    return [tr for tr in TR if tr >= 0], max_rank
```

Implementierung

```
def induce_pos(TR):
    posR = [0] * len(TR)
    for i, tr in enumerate(TR):
        posR[tr] = i
    return posR
```

Laufzeitanalyse

- Im Ungünstigsten Fall: $K = n, |LMS| = n/2$.
- Erstellen der Hilfsarrays: $2 \cdot n$
- 2x Initiieren des Suffixarrays: $2(n + n/2) = 3 \cdot n$

Laufzeitanalyse

- Im Ungünstigsten Fall: $K = n, |LMS| = n/2$.
- Erstellen der Hilfsarrays: $2 \cdot n$
- 2x Initiieren des Suffixarrays: $2(n + n/2) = 3 \cdot n$
- 2x Links-Induktions-Scan: $2(n + n) = 4 \cdot n$
- 2x Rechts-Induktions-Scan: $2(n + n) = 4 \cdot n$

Laufzeitanalyse

- Im Ungünstigsten Fall: $K = n, |LMS| = n/2$.
- Erstellen der Hilfsarrays: $2 \cdot n$
- 2x Initiieren des Suffixarrays: $2(n + n/2) = 3 \cdot n$
- 2x Links-Induktions-Scan: $2(n + n) = 4 \cdot n$
- 2x Rechts-Induktions-Scan: $2(n + n) = 4 \cdot n$
- Rank-Array erstellen: $n + n/2 = 3/2 \cdot n$
- pos_R-Array induzieren: $n/2$

Laufzeitanalyse

- Ohne Rekursion beträgt die Laufzeit $15 \cdot n = \mathcal{O}(n)$
- Mit maxmialer Rekursionstiefe: $30 \cdot n = \mathcal{O}(n)$

Laufzeitanalyse

- Ohne Rekursion beträgt die Laufzeit $15 \cdot n = \mathcal{O}(n)$
- Mit maxmialer Rekursionstiefe: $30 \cdot n = \mathcal{O}(n)$
- $\sum_{i=0}^{\infty} n/2^i = 2 \cdot n$.
- Somit ist der SAIS ein Linearzeitalgorithmus.

Mustersuche mit dem Suffix-Array

- Binäre Suche möglich.
- Pro Binärschritt aber bis zu $\mathcal{O}(m)$ Vergleiche nötig.
- 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 Array gefunden werden.

$i =$	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	$
$pos =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3

Array-Abschnitt suchen

```python
def search_l(pos, T, P):
    lft, rht, m = 0, len(T) - 1, len(P)
    while (lft <= rht):
        mid = (lft + rht) >> 1
        if P <= T[pos[mid]:pos[mid] + m]: rht = mid - 1
        else: lft = mid + 1
    if mid < len(T) - 1 and T[pos[mid]:pos[mid] + m] != P \
        and T[pos[mid + 1]:pos[mid + 1] + m] == P: mid += 1
    return mid

def search_r(pos, T, P):
    lft, rht, n, m = 0, len(T) - 1, len(T), len(P)
    while (lft <= rht):
        mid = (lft + rht) >> 1
        if P < T[pos[mid]:pos[mid] + m]: rht = mid - 1
        else: lft = mid + 1
    if T[pos[mid]:pos[mid] + m] != P: mid -= 1
    return mid
```

Array-Abschnitt suchen

Achtung: rechte Grenze einschließlich!

```
def search_l(pos, T, P):
    lft, rht, m = 0, len(T) - 1, len(P)
    while (lft <= rht):
        mid = (lft + rht) >> 1
        if P <= T[pos[mid]:pos[mid] + m]: rht = mid - 1
        else: lft = mid + 1
    if mid < len(T) - 1 and T[pos[mid]:pos[mid] + m] != P \
        and T[pos[mid + 1]:pos[mid + 1] + m] == P: mid += 1
    return mid

def search_r(pos, T, P):
    lft, rht, n, m = 0, len(T) - 1, len(T), len(P)
    while (lft <= rht):
        mid = (lft + rht) >> 1
        if P < T[pos[mid]:pos[mid] + m]: rht = mid - 1
        else: lft = mid + 1
    if T[pos[mid]:pos[mid] + m] != P: mid -= 1
    return mid
```

Weitere Anfragen

- Allein mit dem Suffix-Array ist es sehr kompliziert den längster wiederholter Teilstring oder den kürzesten eindeutigen Teilstring abzufragen.
- Mit einem Hilfsarray lässt sich die Baumstruktur des Suffix-Trees rekonstruieren somit lassen sich diese Anfragen effizient bearbeiten.

Weitere Anfragen

- Allein mit dem Suffix-Array ist es sehr kompliziert den längster wiederholter Teilstring oder den kürzesten eindeutigen Teilstring abzufragen.
- Mit einem Hilfsarray lässt sich die Baumstruktur des Suffix-Trees rekonstruieren somit lassen sich diese Anfragen effizient bearbeiten.
- Deshalb wird das *longest common prefix* (LCP)-Array eingeführt.

LCP-Array

i	$pos[i]$	$LCP[i]$	$T[pos[i]:]$
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	`ississippii$`
7	0	0	`miississippii$`
8	10	0	`pii$`
9	9	1	`ppii$`
10	7	0	`sippii$`
11	4	2	`sissippii$`
12	6	1	`ssippii$`
13	3	3	`ssissippii$`

LCP-Array erstellen

Naive Methode: alle sortierten Suffixe nacheinander mit ihren Vorgängern überprüfen.

LCP-Array erstellen

Naive Methode: alle sortierten Suffixe nacheinander mit ihren Vorgängern überprüfen. Laufzeit: $\mathcal{O}(n^2)$.

```
def lcp_naive(T, pos):
    n = len(pos)
    lcp, prev = [-1] * n, pos[0]
    for i, curr in enumerate(pos[1:]):
        for d in range(n):
            if T[prev + d] != T[curr + d]: break
        lcp[i + 1], prev = d, curr
    return lcp
```

LCP-Array erstellen

- Angenommen man beginnt mit dem längsten Suffix T, vergleicht diesen mit seinem lexikographischen Vorgänger und erhält einen LCP-Wert der Länge l.
- Folglich muss das nächst-kleinere Suffix $T[1:]$ mindestens einen LCP-Wert von $l-1$ haben.

LCP-Array erstellen

- Angenommen man beginnt mit dem längsten Suffix T, vergleicht diesen mit seinem lexikographischen Vorgänger und erhält einen LCP-Wert der Länge l.
- Folglich muss das nächst-kleinere Suffix $T[1:]$ mindestens einen LCP-Wert von $l - 1$ haben.
- Diese $l - 1$ Positionen müssen nicht mehr miteinander verglichen werden.
- Man kann direkt von der $1 + l$-ten Position den Vergleich starten.

LCP-Array erstellen

- Angenommen man beginnt mit dem längsten Suffix T, vergleicht diesen mit seinem lexikographischen Vorgänger und erhält einen LCP-Wert der Länge l.
- Folglich muss das nächst-kleinere Suffix $T[1:]$ mindestens einen LCP-Wert von $l-1$ haben.
- Diese $l-1$ Positionen müssen nicht mehr miteinander verglichen werden.
- Man kann direkt von der $1+l$-ten Position den Vergleich starten.
- Wo beginnt das lexikographisch kleinere Suffix vom Suffix $T[i:]$?

Rank-Array

- Sei das Rank-Array *rank* definiert als die Gegenfunktion zum Suffix-Array *pos*, mit $rank[pos[i]] = i = pos[rank[i]]$.
- Das Suffix T beginnt also im Suffix-Array an der Position $rank[0]$.

Rank-Array

- Sei das Rank-Array $rank$ definiert als die Gegenfunktion zum Suffix-Array pos, mit $rank[pos[i]] = i = pos[rank[i]]$.
- Das Suffix T beginnt also im Suffix-Array an der Position $rank[0]$.
- Das lexikographisch kleinere Suffix beginnt folglich im Text an Position $pos[rank[0] - 1]$.

LCP-Array konstruieren

```
def build_lcp(T, pos):
    n, l = len(T), 0
    rank, lcp = [0] * n, [-1] * n
    for i, c in enumerate(pos): rank[c] = i

    for curr in range(n - 1):
        l = max(0, l - 1)
        prev = pos[rank[curr] - 1]
        while T[curr + l] == T[prev + l]: l += 1
        lcp[rank[curr]] = l
    return lcp
```

LCP-Array

$prev = 2$

$curr = 0$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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		`miissi...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 11$

$curr = 1$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 11$

$curr = 1$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 11$

$curr = 1$

$l = 2$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 5$

$curr = 2$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 5$

$curr = 2$

$l = 2$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 5$

$curr = 2$

$l = 3$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 5$

$curr = 2$

$l = 4$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 6$

$curr = 3$

$l = 3$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0		`ssissi...`

LCP-Array

$prev = 7$

$curr = 4$

$l = 2$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2		`sissip...`
12	6	1		`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 8$

$curr = 5$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2	2	`sissip...`
12	6	1		`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 4$

$curr = 6$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...
8	10	4		pii$
9	9	9		ppii$
10	7	8		sippii...
11	4	2	2	sissip...
12	6	1		ssippi...
13	3	0	3	ssissi...

LCP-Array

$prev = 4$

$curr = 6$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2	2	`sissip...`
12	6	1		`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 9$

$curr = 7$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i] :]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8		`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 1$

$curr = 8$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 1$

$curr = 8$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 10$

$curr = 9$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 10$

$curr = 9$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9		`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 0$

$curr = 10$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4		`pii$`
9	9	9	1	`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 12$

$curr = 11$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...
8	10	4	0	pii$
9	9	9	1	ppii$
10	7	8	0	sippii...
11	4	2	2	sissip...
12	6	1	1	ssippi...
13	3	0	3	ssissi...

LCP-Array

$prev = 12$

$curr = 11$

$l = 1$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4	0	`pii$`
9	9	9	1	`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev = 13$

$curr = 12$

$l = 0$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4	0	`pii$`
9	9	9	1	`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

LCP-Array

$prev =$

$curr =$

$l =$

i	$pos[i]$	$rank[i]$	$LCP[i]$	$T[pos[i]:]$
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...`
8	10	4	0	`pii$`
9	9	9	1	`ppii$`
10	7	8	0	`sippii...`
11	4	2	2	`sissip...`
12	6	1	1	`ssippi...`
13	3	0	3	`ssissi...`

Laufzeitanalyse der LCP-Array-Konstruktion

- Pro for-Schleifen-Durchlauf wird *curr* um 1 erhöht.
- Gleichzeitig wird l um 1 verringert, kann aber nicht unter 0 fallen.

Laufzeitanalyse der LCP-Array-Konstruktion

- Pro for-Schleifen-Durchlauf wird $curr$ um 1 erhöht.
- Gleichzeitig wird l um 1 verringert, kann aber nicht unter 0 fallen.
- Im ganzen Verlauf wird der Wert $curr + l$ immer höher.
- $curr + l$ kann auch nicht größer werden als $n - 2$, da es sonst ein Mismatch mit dem Sentinal gibt.

Laufzeitanalyse der LCP-Array-Konstruktion

- Pro for-Schleifen-Durchlauf wird *curr* um 1 erhöht.
- Gleichzeitig wird l um 1 verringert, kann aber nicht unter 0 fallen.
- Im ganzen Verlauf wird der Wert $curr + l$ immer höher.
- $curr + l$ kann auch nicht größer werden als $n - 2$, da es sonst ein Mismatch mit dem Sentinal gibt.
- Folglich wird die while-Schleife im gesamten Durchlauf amortisiert $\mathcal{O}(n)$ mal betreten.
- Die Gesamtlaufzeit der LCP-Array-Konstruktion beträgt also amortisiert $\mathcal{O}(n)$.

Längster wiederholter Teilstring

Um den LRS im Text T zu finden, muss lediglich der größte Wert im LCP-Array gefunden werden.

$$i^* = \underset{i<|T|}{\arg\max}\{lcp_T[i]\}$$

$$LRS(T) = T[pos_T[i^*] : pos_T[i^*] + lcp_T[i^*]]$$

Längster wiederholter Teilstring

Um den LRS im Text T zu finden, muss lediglich der größte Wert im LCP-Array gefunden werden.

$$i^* = \underset{i<|T|}{\arg\max}\{lcp_T[i]\}$$

$$LRS(T) = T[pos_T[i^*] : pos_T[i^*] + lcp_T[i^*]]$$

Beispiel: `miississippii$`

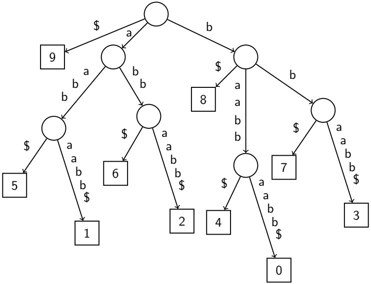
$$\begin{aligned} i^* &= 6 \\ pos_T[i^*] &= 2 \\ lcp_T[i^*] &= 4 \\ LRS(T) &= T[2 : 2 + 4] = \texttt{issi} \end{aligned}$$

Kürzester eindeutiger Teilstring

Der kürzeste eindeutige Teilstring (SUS) lässt sich sehr einfach mit dem LCP-Array finden.

Kürzester eindeutiger Teilstring

Der kürzeste eindeutige Teilstring (SUS) lässt sich sehr einfach mit dem LCP-Array finden. Beispiel: $T =$ `baabbaabb$`



i	$pos[i]$	$LCP[i]$	$T[pos[i]:]$
0	9	-	`$`
1	5	0	`aabb$`
2	1	4	`aabbaabb$`
3	6	1	`abb$`
4	2	3	`abbaabb$`
5	8	0	`b$`
6	4	1	`baabb$`
7	0	5	`baabbaabb$`
8	7	1	`bb$`
9	3	2	`bbaabb$`

Kürzester eindeutiger Teilstring

- Die Stringtiefe eines inneren Knotens von dem Blatt i ausgeht, lässt sich ermitteln, indem das Maximum des aktuellen und nachfolgenden LCP-Wertes genommen wird.
- Eindeutig sei also $ulen(i) = 1 + \max(lcp[i], lcp[i+1])$.

Kürzester eindeutiger Teilstring

- Die Stringtiefe eines inneren Knotens von dem Blatt i ausgeht, lässt sich ermitteln, indem das Maximum des aktuellen und nachfolgenden LCP-Wertes genommen wird.
- Eindeutig sei also $ulen(i) = 1 + \max(lcp[i], lcp[i+1])$.
- Gesucht ist also der Teilstring mit minimaler Stringtiefe, der nicht mit dem Wächter endet.
- Sei $i^* = \operatorname{argmin}_{i<|T|}\{d = ulen(i) \mid pos[i] + d < |T|\}$.

Kürzester eindeutiger Teilstring

- Die Stringtiefe eines inneren Knotens von dem Blatt i ausgeht, lässt sich ermitteln, indem das Maximum des aktuellen und nachfolgenden LCP-Wertes genommen wird.
- Eindeutig sei also $ulen(i) = 1 + \max(lcp[i], lcp[i+1])$.
- Gesucht ist also der Teilstring mit minimaler Stringtiefe, der nicht mit dem Wächter endet.
- Sei $i^* = \operatorname{argmin}_{i<|T|}\{d = ulen(i) \mid pos[i] + d < |T|\}$.
- Es gilt $sus(T) = T[pos[i^*] : pos[i^*] + ulen(i^*)]$.

Im Beispiel: $T = \texttt{baabbaabb\$}$
$i^* = 9 \rightarrow pos[9] = 3, ulen(i^*) = 3 \rightarrow sus(T) = T[3 : 3 + 3] = \texttt{bba}$

Längster gemeinsamer Teilstring

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

Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1T_2\$_2$ mit $\$_1 < \$_2$.
- Alle Suffixe, die vor $\$_1$ anfangen, werden mit 1 gelabelt, alle Anderen mit 2.

Längster gemeinsamer Teilstring

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

Längster gemeinsamer Teilstring

Beispiel: $T =$ `baabb#aaba$`

i	$pos[i]$	$LCP[i]$	$label[i]$	$T[pos[i]:]$
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

Beispiel: T = `baabb#aaba$`

i	$pos[i]$	$LCP[i]$	$label[i]$	$T[pos[i]:]$
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

Beispiel: $T =$ `baabb#aaba$`

i	$pos[i]$	$LCP[i]$	$label[i]$	$T[pos[i]:]$
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$`

$i^* = 4 \rightarrow pos[i] = 1, LCP[i] = 3 \rightarrow T[1 : 1+3] = \texttt{aab}$

Zusammenfassung

- Der SAIS-Algorithmus erstellt ein Suffix-Array in $\mathcal{O}(n)$.
- Verschiedene Fragestellungen können mit dem Suffix-Array beantwortet werden:
    - Kommt P in T vor? Wenn ja, wo?
    - 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.