Suffix Arrays

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2024

Overview

Previous Lecture

- Suffix trees
  - Applications (pattern search, longest repeated substring, shortest unique substring)
  - Linear time construction

Overview

Previous Lecture

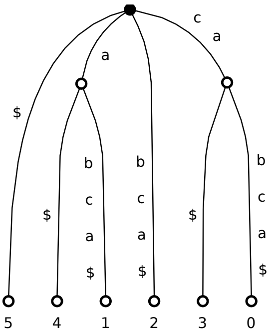
- Suffix trees
  - Applications (pattern search, longest repeated substring, shortest unique substring)
  - Linear time construction

Today

- Suffix arrays
- Applications
  - pattern search
  - longest repeated substring
  - shortest unique substring,
  - longest common substring
  - maximal unique matches (MUMs)
- LCP arrays and linear-time computation
- **Next lecture:** Linear-time construction of suffix arrays

Suffix trees and suffix arrays

T = cabca\$



Definition

The **suffix array** of a string s\$ with $|s\$| = n$ is the permutation pos of $\{0, \ldots, n-1\}$ that represents the lexicographic ordering of all suffixes of s\$:
$\text{pos} = [5, 4, 1, 2, 3, 0]$.

Motivation for Suffix Arrays

Why switch from tree to array?

- High memory requirements for suffix tree ($O(n) \approx 20n$ bytes)
- With alphabetically sorted outgoing edges:
  Sequence of leaf numbers
  $=$ starting positions of lexicographically sorted suffixes
- Array: $4n$ bytes (for 32-bit integers, $n < 2^{32}$)

Motivation for Suffix Arrays

Why switch from tree to array?

- High memory requirements for suffix tree ($O(n) \approx 20n$ bytes)
- With alphabetically sorted outgoing edges:
  Sequence of leaf numbers
  $=$ starting positions of lexicographically sorted suffixes
- Array: $4n$ bytes (for 32-bit integers, $n < 2^{32}$)
- Represents only the leaf level of the suffix tree
- Representation of tree structure with additional arrays
- Some questions can be solved directly with cache-efficient algorithms

Example of a Suffix Array

Notation: p for text positions, r for lexicographic ranks.
In a auffix array, $\mathrm{pos}[r]$ is the text position where the r-th smallest suffix starts.

$p =$	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	\$

$$\begin{array}{c|cccccccccccccc} r = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \mathrm{pos} = & \underbrace{13}_{\$} & \multicolumn{6}{c}{\underbrace{12 \quad 11 \quad 1 \quad 8 \quad 5 \quad 2}_{\mathrm{i}}} & \underbrace{0}_{\mathrm{m}} & \multicolumn{2}{c}{\underbrace{10 \quad 9}_{\mathrm{p}}} & \multicolumn{4}{c}{\underbrace{7 \quad 4 \quad 6 \quad 3}_{\mathrm{s}}} \end{array}$$

We may partition the suffixes into “buckets” according to their first letter.

Construction of Suffix Arrays

Three possibilities

1. from the suffix tree by scanning the leaves, in $O(n)$ time
   Disadvantage: high memory consumption for intermediate tree

Construction of Suffix Arrays

Three possibilities

1. from the suffix tree by scanning the leaves, in $O(n)$ time
   Disadvantage: high memory consumption for intermediate tree
2. directly by some standard sorting algorithm:
   ```
   def build_suffixarray_naive(T):
       suffixes = lambda p: T[p:]
       return sorted(range(len(T)), key=suffixes)
   ```
   Disadvantage: Running time

Construction of Suffix Arrays

Three possibilities

1. from the suffix tree by scanning the leaves, in $O(n)$ time
   Disadvantage: high memory consumption for intermediate tree
2. directly by some standard sorting algorithm:

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

   Disadvantage: Running time $O(n^2 \log n)$ and intermediate memory

Construction of Suffix Arrays

Three possibilities

1. from the suffix tree by scanning the leaves, in $O(n)$ time
   Disadvantage: high memory consumption for intermediate tree
2. directly by some standard sorting algorithm:
   ```
   def build_suffixarray_naive(T):
       suffixes = lambda p: T[p:]
       return sorted(range(len(T)), key=suffixes)
   ```
   Disadvantage: Running time $O(n^2 \log n)$ and intermediate memory
3. directly by an efficient linear-time algorithm (later)
   Disadvantage: complicated algorithm

Search with suffix arrays

Definitions

- Pattern $P \in \Sigma^m$ and text $T \in \Sigma^n$
- Define

$$L := \min\left[\{r | P \leq T[\mathrm{pos}[r]\ldots]\} \cup \{n\}\right],$$
$$R := \max\left[\{r | P \geq T[\mathrm{pos}[r]\ldots\mathrm{pos}[r] + |P|]\} \cup \{-1\}\right].$$

- All suffixes in the interval $[L, R]$ start with P.
- P occurs in T if (and only if) $R \geq L$.
- Searching in suffix array $\Longleftrightarrow$ determining $[L, R]$
- Use two **binary searches** to determine $[L, R]$.

Example: Binary search in Suffix Arrays

Search for "$\texttt{is}$", then for "$\texttt{sp}$".

$p =$	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
$\texttt{pos} =$	13	12	11	1	8	5	2	0	10	9	7	4	6	3

Running Time for Searching

1. **Decision problem**:
   As we have seen, the running time is $O(m \log n)$.
2. **How often does P occur in T?**
   Same as above, because the number of occurrences is $z = R - L + 1$.
3. **Where does P occur in T?**
   Once the interval $[L, R]$ is known, the start positions can be found by scanning through the interval in additional $O(z)$ time.

Running Time for Searching

1. **Decision problem**:
   As we have seen, the running time is $O(m \log n)$.
2. **How often does P occur in T?**
   Same as above, because the number of occurrences is $z = R - L + 1$.
3. **Where does P occur in T?**
   Once the interval $[L, R]$ is known, the start positions can be found by scanning through the interval in additional $O(z)$ time.

Note: With a different approach (Backward search; later), the factor $\log n$ can be saved.

Motivation: Enhanced Suffix Arrays

Can we use suffix arrays just like suffix trees?

Motivation: Enhanced Suffix Arrays

Can we use suffix arrays just like suffix trees?
Not like defined so far. . . . We need more structure!

Motivation: Enhanced Suffix Arrays

Can we use suffix arrays just like suffix trees?
Not like defined so far. . . . We need more structure!

- Enhancing suffix arrays with **Longest Common Prefix (LCP)** arrays to represent the tree structure above the leaf level
- **Applications** of enhanced suffix arrays
  - Longest repeated substring
  - Shortest unique substring
  - Longest common substring
  - Maximal unique matches (MUMs)

Longest Common Prefix (LCP) arrays

LCP Array by Example

$p =$	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	
`lcp` =	-1	0						0	0		0				-1
	$	i						m	p		s				

`lcp` represents longest common prefixes
of lexicographically adjacent suffixes (looking left).

LCP Array

Definition: longest common prefix array

Let $T \in \Sigma^n$ be a text and let pos be the corresponding suffix array.
We define `lcp` to be an array of length $(n+1)$ such that

$$\mathtt{lcp}[r] = \begin{cases} -1 & \text{if } r = 0 \text{ or } r = n, \\ lcp(T[\mathrm{pos}[r-1]\ldots], T[\mathrm{pos}[r]\ldots]) & \text{otherwise,} \end{cases}$$

where

$$lcp(s,t) := \max\left\{ i \in \mathbb{N}_0 \mid s[:i] - t[:i] \right\}.$$

LCP Array

Definition: longest common prefix array

Let $T \in \Sigma^n$ be a text and let $\mathtt{pos}$ be the corresponding suffix array.
We define $\mathtt{lcp}$ to be an array of length $(n+1)$ such that

$$\mathtt{lcp}[r] = \begin{cases} -1 & \text{if } r = 0 \text{ or } r = n, \\ lcp\big(T[\mathtt{pos}[r-1]\ldots], T[\mathtt{pos}[r]\ldots]\big) & \text{otherwise,} \end{cases}$$

where

$$lcp(s,t) := \max\left\{ i \in \mathbb{N}_0 \mid s[:i] - t[:i] \right\}.$$

Terminology

A suffix array plus auxiliary arrays like $\mathtt{lcp}$ is called **enhanced suffix array**.

Naive Construction of LCP Array

```python
def lcp_naive(pos,T):
    lcp = [-1]  # first -1 (at index 0)
    for r in range(1, len(T)):
        # compare suffix starting at pos[r-1]
        # to suffix starting at pos[r]
        L = 0
        while T[pos[r-1] + L] == T[pos[r] + L]:
            L += 1  # cannot run off the string (sentinel!)
        lcp.append(L)
    lcp.append(-1) # trailing -1 (at index n)
    return lcp
```

Naive Construction of LCP Array

```
def lcp_naive(pos,T):
    lcp = [-1]  # first -1 (at index 0)
    for r in range(1, len(T)):
        # compare suffix starting at pos[r-1]
        # to suffix starting at pos[r]
        L = 0
        while T[pos[r-1] + L] == T[pos[r] + L]:
            L += 1  # cannot run off the string (sentinel!)
        lcp.append(L)
    lcp.append(-1) # trailing -1 (at index n)
    return lcp
```

Running time: worst case $O(n^2)$, repetitive texts are bad.
Improved by Kasai's algorithm (soon).

Applications of Enhanced Suffix Arrays

Longest Repeated Substring (by Enhanced Suffix Array)

Example

The longest repeated substring in cabca is ca.

Question

How do we find the **longest repeated substring** using suffix and LCP arrays?

Longest Repeated Substring (by Enhanced Suffix Array)

Example

The longest repeated substring in cabca is ca.

Question

How do we find the **longest repeated substring** using suffix and LCP arrays?

Answer

- Just look for maximum value in LCP array
- Suffix array at that rank r tells where the substring starts

Longest Repeated Substring (by Enhanced Suffix Array)

Example

The longest repeated substring in cabca is ca.

Question

How do we find the **longest repeated substring** using suffix and LCP arrays?

Answer

- Just look for maximum value in LCP array
- Suffix array at that rank r tells where the substring starts
- Running time $O(n)$
- Note that this algorithm is simpler than using the suffix tree.

Example: Longest Repeated Substring via ESA

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

Shortest Unique Substring (Enhanced Suffix Array)

Idea

- For every suffix of $T = s\$$, determine the shortest prefix that is unique; i.e. for each i, determine the smallest j such that $T[i \dots j]$ is unique in T.
- This is easy using the LCP array:
  The length of the string must be $\ell := \max\{\mathtt{lcp}[r], \mathtt{lcp}[r+1]\} + 1$, so

  $$j = i + \max\{\mathtt{lcp}[r], \mathtt{lcp}[r+1]\},$$

  where $i = \mathtt{pos}[r] = i$.
- However, we must exclude cases where $j = n - 1$, meaning that $T[i \dots j]$ is only unique due to the sentinel $T[n-1] = \$$.

Code: Shortest Unique Substring

```
def shortest_unique_substring(pos, lcp):
    n = len(pos)
    # full text (without sentinel) is always unique
    best_i = 0
    best_j = n-1
    for r in range(len(pos)):
        i = pos[r]
        j = i + max(lcp[r], lcp[r+1]) + 1
        if j == n: continue
        if (j-i) < (best_j-best_i):
            best_i, best_j = i, j
    return best_i, best_j
```

Running time: $O(n)$

Longest Common Substrings (using Suffix Arrays)

Problem

Given two strings s, t, find their **longest common substring**.

Example

Let $s = \texttt{ANANAS}$ and $t = \texttt{BANANA}$, then $lcs(s, t) = \texttt{ANANA}$.

Longest Common Substrings (using Suffix Arrays)

Problem

Given two strings s, t, find their **longest common substring**.

Example

Let $s = \texttt{ANANAS}$ and $t = \texttt{BANANA}$, then $lcs(s, t) = \texttt{ANANA}$.

Idea

- Build **generalized** enhanced suffix array of s and t, i.e. build the enhanced suffix array $T = s\#t\$$.
- Common substring $\rightarrow$ **consecutive positions** in suffix array
- Length given by LCP value
- Distinguish: repeat in one string vs. common substring

Code: Longest Common Substring

```
def longest_common_substring(s,t):
    T = s + '#' + t + '$'
    pos, lcp = sa_and_lcp(T)
    lcs = ''
    for r in range(1, len(pos)):
        # do both suffixes start in the same string => skip r
        if (pos[r] <= len(s) and pos[r-1] <= len(s)) \
        or (pos[r] > len(s) and pos[r-1] > len(s)):
            continue
        if lcp[r] > len(lcs):
            lcs = T[pos[r]:pos[r]+lcp[r]]  # line 11
    return lcs
```

Code: Longest Common Substring

```
def longest_common_substring(s,t):
    T = s + '#' + t + '$'
    pos, lcp = sa_and_lcp(T)
    lcs = ''
    for r in range(1, len(pos)):
        # do both suffixes start in the same string => skip r
        if (pos[r] <= len(s) and pos[r-1] <= len(s)) \
        or (pos[r] > len(s) and pos[r-1] > len(s)):
            continue
        if lcp[r] > len(lcs):
            lcs = T[pos[r]:pos[r]+lcp[r]]  # line 11
    return lcs
```

Running time: $O(n)$, assuming setting `lcs` in line 11 is $O(1)$

Maximal Unique Matches (MUMs)

Definitions

- Let two strings $s, t \in \Sigma^*$ be given.
- A string u is a **unique match** if it occurs **exactly** once in s and t, respectively.
- A unique match u is **maximal** if there is no $a \in \Sigma$, such that au or ua is a unique match.

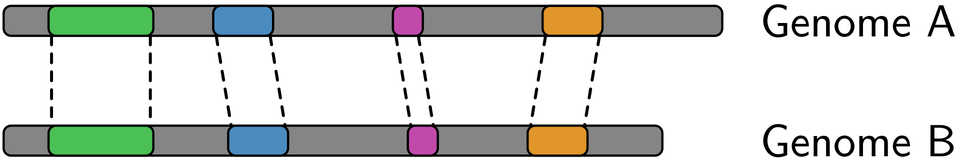
Maximal Unique Matches (MUMs)

Definitions

- Let two strings $s, t \in \Sigma^*$ be given.
- A string u is a **unique match** if it occurs **exactly** once in s and t, respectively.
- A unique match u is **maximal** if there is no $a \in \Sigma$, such that au or ua is a unique match.

Significance of MUMs

MUMs can be used as anchor points for aligning long sequences.



Idea: Computing MUMs using Enhanced Suffix Arrays

Reuse from longest common substrings:

- Build **generalized** enhanced suffix array of s and t, i.e. build the enhanced suffix array $T = s\#t\$$.
- Common substring $\rightarrow$ **consecutive positions** in suffix array
- Length given by LCP value
- Distinguish: repeat in one string vs. common substring

Idea: Computing MUMs using Enhanced Suffix Arrays

Reuse from longest common substrings:

- Build **generalized** enhanced suffix array of s and t, i.e. build the enhanced suffix array $T = s\#t\$$.
- Common substring $\rightarrow$ **consecutive positions** in suffix array
- Length given by LCP value
- Distinguish: repeat in one string vs. common substring

Additional considerations for MUMs

- Ensure hits are unique: **isolated** local maxima in LCP table
- Check that we cannot extend to the left

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]	r	pos[r]	lcp[r]
0	11	-1	12	5	2
1	21	0	13	2	1
2	10	0	14	14	4
3	20	1	15	17	1
4	4	1	16	9	0
5	13	2	17	19	2
6	16	2	18	1	1
7	0	1	19	8	1
8	6	2	20	18	3
9	3	0	21	7	2
10	12	3	22	-	-1
11	15	3			

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]	r	pos[r]	lcp[r]
0	11	-1	12	5	2
1	21	0	13	2	1
2	10	0	14	14	4
3	20	1	15	17	1
4	4	1	16	9	0
5	13	2	17	19	2
6	16	2	18	1	1
7	0	1	19	8	1
8	6	2	20	18	3
9	3	0	21	7	2
10	12	3	22	-	-1
11	15	3			

Local maxima

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]	r	pos[r]	lcp[r]
0	11	-1	12	5	2
1	21	0	13	2	1
2	10	0	14	14	4
3	20	1	15	17	1
4	4	1	16	9	0
5	13	2	17	19	2
6	16	2	18	1	1
7	0	1	19	8	1
8	6	2	20	18	3
9	3	0	21	7	2
10	12	3	22	-	-1
11	15	3			

Local maxima

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]
0	11	-1
1	21	0
2	10	0
3	20	1
4	4	1
5	13	2
6	16	2
7	0	1
8	6	2
9	3	0
10	12	3
11	15	3

r	pos[r]	lcp[r]
12	5	2
13	2	1
		4
		1
16	9	0
17	19	2
18	1	1
	8	1
	3	3
21	7	2
22	-	-1

Not maximal!

Not unique!

Local maxima

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]	r	pos[r]	lcp[r]
0	11	-1	12	5	2
1	21	0	13	2	1
2	10	0	14	14	4
3	20	1	15	17	1
4	4	1	16	9	0
5	13	2	17	19	2
6	16	2	18	1	1
7	0	1	19	8	1
8	6	2	20	[illegible]	3
9	3	0	21	[illegible]	2
10	12	3	22	–	-1
11	15	3			

Not maximal!

Same string!

Local maxima

Example: Computing MUMs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	C	B	B	A	B	A	C	C	C	A	$\$_1$	B	A	B	B	A	B	C	C	A	$\$_2$

r	pos[r]	lcp[r]
0	11	-1
1	21	0
2	10	0
3	20	1
4	4	1
5	13	2
6	16	2
7	0	1
8	6	2
9	3	0
10	12	3
11	15	3

r	pos[r]	lcp[r]
12	5	2
13	2	1
14	14	4
15	17	1
16	9	0
17	19	2
18	1	1
19	8	1
20	18	3
21	7	2
22	-	-1

Local maxima

Valid MUMs:

CCA BBAB

Code: Computing MUMs

```
def compute_mums(s,t):
    T = s + '#' + t + '$'
    pos, lcp = sa_and_lcp(T)
    for r in range(1, len(pos)):
        p1, p2 = pos[r-1], pos[r]
        if (p1 <= len(s)) and (p2 <= len(s)):
            continue
        if (p1 > len(s)) and (p2 > len(s)):
            continue
        if (lcp[r-1] >= lcp[r]) or
           (lcp[r+1] >= lcp[r]):
            continue
        if (p1 == 0) or (p2 == 0) or
           (T[p1-1] != T[p2-1]):
            yield T[p1:p1+lcp[r]]
```

Constructing LCP Arrays in Linear Time

Inverting the Suffix Array

Observation

- Any suffix array is a **permutation** of numbers from 0 to $n - 1$.
- A suffix array can thus be **inverted** (in linear time).

Inverting the Suffix Array

Observation

- Any suffix array is a **permutation** of numbers from 0 to $n-1$.
- A suffix array can thus be **inverted** (in linear time).

Terminology

- **Suffix array:** `pos[r]` is the start **pos**ition of the suffix with lexicographical rank `r`.
- **Inverted suffix array:** `rank[p]` is the lexicographical **rank** of the suffix that starts at position `p`.

Inverting the Suffix Array

Observation

- Any suffix array is a **permutation** of numbers from 0 to $n - 1$.
- A suffix array can thus be **inverted** (in linear time).

Terminology

- **Suffix array:** `pos[r]` is the start **pos**ition of the suffix with lexicographical rank `r`.
- **Inverted suffix array:** `rank[p]` is the lexicographical **rank** of the suffix that starts at position `p`.

Linear-time inversion

Note: `rank` is filled in random-access order.

```
rank = [-1] * n
for r in range(n): rank[pos[r]] = r
```

Linear Time LCP Construction: Kasai's Algorithm

Input

Text T, suffix array pos, its inverse rank.

Idea

- Compare each suffix, starting at text position $p = 0, 1, \ldots, n-1$, to its respective predecessor (lexicographically next smaller suffix)
- Get predecessor by using suffix array (pos) and its inverse (rank): For the suffix starting at p, find text position $\texttt{pos}[\texttt{rank}[p] - 1]$.
- Fill in LCP table in $\texttt{rank}[p]$-order (not from left to right or r-order!)

Linear Time LCP Construction: Kasai's Algorithm

Input

Text `T`, suffix array `pos`, its inverse `rank`.

Idea

- Compare each suffix, starting at text position $p = 0, 1, \ldots, n-1$, to its respective predecessor (lexicographically next smaller suffix)
- Get predecessor by using suffix array (`pos`) and its inverse (`rank`): For the suffix starting at p, find text position $\mathtt{pos}[\mathtt{rank}[p] - 1]$.
- Fill in LCP table in $\mathtt{rank}[p]$-order (not from left to right or r-order!)
- Moving from p to $p+1$, we keep the computed common prefix, without the first character, similarly to following a suffix link. This is what saves us time.

Example: Kasai's Algorithm

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

Example: Kasai's Algorithm

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

Example: Kasai's Algorithm

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

Code: Kasai's Algorithm

```
def compute_lcp(T, pos, rank):
    n = len(pos)
    lcp = [-1] * (n+1)
    l = 0  # current common prefix length
    for p in range(n-1):
        r = rank[p]
        pleft = pos[r-1]
        while T[p+l] == T[pleft + l]:
            l += 1
        lcp[r] = l
        l = max(l-1, 0)  # next suffix: lose first character
    return lcp
```

Why Does Kasai's Algorithm Run in Linear Time?

```
for p in range(n-1):  # line 1
    r = rank[p]
    pleft = pos[r-1]
    while T[p+l] == T[pleft + l]:  # line 4
        l += 1  # line 5
    lcp[r] = l
    l = max(l-1, 0)  # line 7
```

Test in line 5 can be performed at most $2n$ times:

- Mismatch: `while` loop terminated: at most $n-1$ times.
- Match: `l` is incremented in line 5 and can decrease by at most 1 in line 7.
- p increased in line 1;
  $\rightarrow$ `p+l` is larger when next reaching Line 4;
  $\rightarrow$ can happen at most n times.

Summary

- Suffix arrays
- LCP array
- Enhanced suffix array can often replace suffix tree
- Applications
  - Longest repeated substring
  - Shortest unique substring
  - Longest common substring
  - Maximal unique matches (MUMs)
- Kasai's algorithm: linear time **LCP array** construction

Possible Exam Questions

- Define: What is a suffix array of a string?
- Construct a suffix array for an example string.
- Explain pattern search with suffix arrays.
- Give the definition of the LCP array and explain it.
- Construct the LCP array for a given string.
- What is the advantage of an enhanced suffix array over a suffix tree?
- Define one of the following problems, and explain how it can be solved using an enhanced suffix array: longest repeated substring, shortest unique substring, longest common substring, maximal unique matches.
- Why and how can a suffix array be inverted?
- Explain Kasai's algorithm. What is its running time?
- Apply Kasai's algorithm to a given example.