Suffix Arrays

02-714
Slides by Carl Kingsford
Suffix Arrays

- Even though Suffix Trees are $O(n)$ space, the constant hidden by the big-Oh notation is somewhat “big”: $\approx 20$ bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. “Linear” but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.
Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

$s = \text{attcatg}$

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of $s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>attcatg$</td>
</tr>
<tr>
<td>2</td>
<td>ttcatg$</td>
</tr>
<tr>
<td>3</td>
<td>tcatg$</td>
</tr>
<tr>
<td>4</td>
<td>catg$</td>
</tr>
<tr>
<td>5</td>
<td>atg$</td>
</tr>
<tr>
<td>6</td>
<td>tg$</td>
</tr>
<tr>
<td>7</td>
<td>g$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

Sort the suffixes alphabetically.

The indices just “come along for the ride”.

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of $s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>atg$</td>
</tr>
<tr>
<td>1</td>
<td>attcatg$</td>
</tr>
<tr>
<td>4</td>
<td>catg$</td>
</tr>
<tr>
<td>7</td>
<td>g$</td>
</tr>
<tr>
<td>3</td>
<td>tcatg$</td>
</tr>
<tr>
<td>6</td>
<td>tg$</td>
</tr>
<tr>
<td>2</td>
<td>ttcatg$</td>
</tr>
</tbody>
</table>
Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>attcatg$</td>
</tr>
<tr>
<td>2</td>
<td>ttcatg$</td>
</tr>
<tr>
<td>3</td>
<td>tcatg$</td>
</tr>
<tr>
<td>4</td>
<td>catg$</td>
</tr>
<tr>
<td>5</td>
<td>atg$</td>
</tr>
<tr>
<td>6</td>
<td>tg$</td>
</tr>
<tr>
<td>7</td>
<td>g$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

sort the suffixes alphabetically

the indices just “come along for the ride”

8
5
1
4
7
3
6
2
Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>2</td>
<td>attcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$</td>
</tr>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
</tbody>
</table>

sort the suffixes alphabetically
the indices just "come along for the ride"
Another Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

$s = \text{cattcat}$

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>2</td>
<td>attcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

Sort the suffixes alphabetically:

- the indices just “come along for the ride”

Indices:

- 8: cattcat$
- 6: attcat$
- 2: ttcat$
- 5: tcat$
- 1: cat$
- 7: at$
- 4: t$
- 3: $
Search via Suffix Arrays

\[ s = \text{cattcat}\$ \]

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>6</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>at$</td>
<td>attcat$</td>
<td>cat$</td>
<td>cattcat$</td>
<td>t$</td>
<td>tcat$</td>
<td>ttcat$</td>
</tr>
</tbody>
</table>

- Does string “at” occur in \( s \)?
- Binary search to find “at”.
- What about “tt”?
Counting via Suffix Arrays

$s = \texttt{cattcat}$

- How many times does “at” occur in the string?
- All the suffixes that start with “at” will be next to each other in the array.
- Find one suffix that starts with “at” (using binary search).
- Then count the neighboring sequences that start with at.
**K-mer counting**

**Problem:** Given a string $s$, an integer $k$, output all pairs $(b, i)$ such that $b$ is a length-$k$ substring of $s$ that occurs exactly $i$ times.

$k = 2$

<table>
<thead>
<tr>
<th></th>
<th>CurrentCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$1$</td>
</tr>
<tr>
<td>6</td>
<td>$1$</td>
</tr>
<tr>
<td>2</td>
<td>$2$</td>
</tr>
<tr>
<td>5</td>
<td>$1$ (at,2)</td>
</tr>
<tr>
<td>1</td>
<td>$2$</td>
</tr>
<tr>
<td>7</td>
<td>$1$ (ca,2)</td>
</tr>
<tr>
<td>4</td>
<td>$1$ (t$,1)$</td>
</tr>
<tr>
<td>3</td>
<td>$1$ (tc,1)</td>
</tr>
</tbody>
</table>

1. Build a suffix array.

2. Walk down the suffix array, keeping a `CurrentCount` count.
   - If the current suffix has length $< k$, skip it.
   - If the current suffix starts with the same length-$k$ string as the previous suffix: increment `CurrentCount`.
   - Else:
     - output `CurrentCount` and previous length-$k$ suffix
     - `CurrentCount := 1`

Output `CurrentCount` & length-$k$ suffix.
Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:
  sort the n suffixes, which takes $O(n \log n)$ comparisons, where each comparison takes $O(n)$.

- There are several direct $O(n)$ algorithms for constructing suffix arrays that use very little space.

- The Skew Algorithm is one that is based on divide-and-conquer.

- An simple $O(n)$ algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)
Relationship Between Suffix Trees & Suffix Arrays

\[ \Sigma = \{\$, a, c, t\} \]

\[ s = \text{cattcat} \$

12345678

Red #s = starting position of the suffix ending at that leaf

Leaf labels left to right: 86251743

Edges leaving each node are sorted by label (left-to-right).
Relationship Between Suffix Trees & Suffix Arrays

\[ \Sigma = \{\$, a, c, t\} \]
\[ s = \text{cattcat}\$

12345678

Red #s = starting position of the suffix ending at that leaf

Edges leaving each node are sorted by label (left-to-right).

Leaf labels left to right: 86251743

\[ s = \text{cattcat}\$

\begin{array}{|c|}
\hline
8 & \$
\hline
6 & \text{at}$
\hline
2 & \text{attcat}$
\hline
5 & \text{cat}$
\hline
1 & \text{cattcat}$
\hline
7 & t$
\hline
4 & \text{tcat}$
\hline
3 & \text{ttcat}$
\hline
\end{array}
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prefix-Doubling</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM [Manber and Myers 1993]</td>
<td>$O(n \log n)$</td>
<td>30</td>
<td>$8n$</td>
</tr>
<tr>
<td>LS [Larsson and Sadakane 1999]</td>
<td>$O(n \log n)$</td>
<td>3</td>
<td>$8n$</td>
</tr>
<tr>
<td><strong>Recursive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KA [Ko and Aluru 2003]</td>
<td>$O(n)$</td>
<td>2.5</td>
<td>7–10$n$</td>
</tr>
<tr>
<td>KS [Kärkkäinen and Sanders 2003]</td>
<td>$O(n)$</td>
<td>4.7</td>
<td>10–13$n$</td>
</tr>
<tr>
<td>KSPP [Kim et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>HSS [Hon et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>KJP [Kim et al. 2004]</td>
<td>$O(n \log \log n)$</td>
<td>3.5</td>
<td>13–16$n$</td>
</tr>
<tr>
<td>N [Na 2005]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Induced Copying</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT [Itoh and Tanaka 1999]</td>
<td>$O(n^2 \log n)$</td>
<td>6.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>S [Seward 2000]</td>
<td>$O(n^2 \log n)$</td>
<td>3.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK [Burkhardt and Kärkkäinen 2003]</td>
<td>$O(n \log n)$</td>
<td>3.5</td>
<td>5–6$n$</td>
</tr>
<tr>
<td>MF [Manzini and Ferragina 2004]</td>
<td>$O(n^2 \log n)$</td>
<td>1.7</td>
<td>$5n$</td>
</tr>
<tr>
<td>SS [Schürmann and Stoye 2005]</td>
<td>$O(n^2)$</td>
<td>1.8</td>
<td>9–10$n$</td>
</tr>
<tr>
<td>BB [Baron and Bresler 2005]</td>
<td>$O(n \sqrt{\log n})$</td>
<td>2.1</td>
<td>18$n$</td>
</tr>
<tr>
<td>M [Maniscalco and Puglisi 2007]</td>
<td>$O(n^2 \log n)$</td>
<td>1.3</td>
<td>5–6$n$</td>
</tr>
<tr>
<td>MP [Maniscalco and Puglisi 2006]</td>
<td>$O(n^2 \log n)$</td>
<td>1</td>
<td>5–6$n$</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT+KA</td>
<td>$O(n^2 \log n)$</td>
<td>4.8</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK+IT+KA</td>
<td>$O(n \log n)$</td>
<td>2.3</td>
<td>5–6$n$</td>
</tr>
<tr>
<td>BK+S</td>
<td>$O(n \log n)$</td>
<td>2.8</td>
<td>5–6$n$</td>
</tr>
<tr>
<td><strong>Suffix Tree</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K [Kurtz 1999]</td>
<td>$O(n \log \sigma)$</td>
<td>6.3</td>
<td>13–15$n$</td>
</tr>
</tbody>
</table>

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].
The Skew Algorithm
Kärkkäinen & Sanders, 2003

- **Main idea: Divide suffixes into 3 groups:**
  - Those starting at positions $i=0,3,6,9,\ldots$ ($i \mod 3 = 0$)
  - Those starting at positions $1,4,7,10,\ldots$ ($i \mod 3 = 1$)
  - Those starting at positions $2,5,8,11,\ldots$ ($i \mod 3 = 2$)

- For simplicity, assume text length is a multiple of 3 after padding with a special character.

```
mississippi$$
```

Basic Outline:
- Recursively handle suffixes from the $i \mod 3 = 1$ and $i \mod 3 = 2$ groups.
- Merge the $i \mod 3 = 0$ group at the end.
Handing the 1 and 2 groups

\[ s = \text{mississippi} \]

triples for groups 1 and 2 groups

assign each triple a token in lexicographical order

recursively compute the suffix array for tokenized string

4321765

Every suffix of t corresponds to a suffix of s (maybe with some cruft at the end of it).
Relationship Between $t$ and $s$

\[ s = \text{mississippi} $$ \]

\[ t = \text{CCBAEED} \]

Key Point #1: The lexicographical order of the suffixes of $t$ is the same as the order of the group 1 & 2 suffixes of $s$.

Why?

Every suffix of $t$ corresponds to some suffix of $s$ (perhaps with some extra letters at the end of it --- in this case EED)

Because the tokens are sorted in the same order as the triples, the sort order of the suffix of $t$ matches that of $s$.

So: The recursive computational of the suffix array for $t$ gives you the ordering of the group 1 and group 2 suffixes.
Radix Sort

- O(n)-time sort for n items when items can be divided into constant # of digits.

- Put into buckets based on least-significant digit, flatten, repeat with next-most significant digit, etc.

- Example items: 100 123 042 333 777 892 236

- # of passes = # of digits
- Each pass goes through the numbers once.
Handling 0 Suffixes

- First: sort the group 0 suffixes, using the representation \((s[i], S_{i+1})\)

- Since the \(S_{i+1}\) suffixes are already in the array sorted, we can just stably sort them with respect to \(s[i]\), using radix sort.

\[
\begin{array}{l}
\text{1,2-array: } \text{ippississi}$pptississi$ \\
\text{0-array: } \text{mispi}$sip$ sis
\end{array}
\]

- We have to merge the group 0 suffixes into the suffix array for group 1 and 2.
- Given suffix \(S_i\) and \(S_j\), need to decide which should come first.
  - If \(S_i\) and \(S_j\) are both either group 1 or group 2, then the recursively computed suffix array gives the order.
  - If one of \(i\) or \(j\) is 0 (mod 3), see next slide.
Comparing 0 suffix $S_j$ with 1 or 2 suffix $S_i$

Represent $S_i$ and $S_j$ using subsequent suffixes:

\[
\begin{align*}
&i \pmod{3} = 1: \\
&(s[i], S_{i+1}) < (s[j], S_{j+1}) \\
&\equiv 2 \pmod{3} \quad \equiv 1 \pmod{3}
\end{align*}
\]

\[
\begin{align*}
&i \pmod{3} = 2: \\
&(s[i], s[i+1], S_{i+2}) < (s[j], s[j+1], S_{j+2}) \\
&\equiv 1 \pmod{3} \quad \equiv 2 \pmod{3}
\end{align*}
\]

$\Rightarrow$ the suffixes can be compared quickly because the relative order of $S_{i+1}$, $S_{j+1}$ or $S_{i+2}$, $S_{j+2}$ is known from the 1,2-array we already computed.
Running Time

\[ T(n) = O(n) + T(2n/3) \]

- Expand big-O notation: \( T(n) \leq cn + T(2n/3) \) for some \( c \).
- Guess: \( T(n) \leq 3cn \)
- Induction step: assume that is true for all \( i < n \).
- \( T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn \) \( \square \)
Faster Suffix Array Search

• The basic binary search takes $O(|P| \log |T|)$:
  • it takes $O(\log |T|)$ iterations
  • each comparison taking $O(|P|)$ time

• We can do it faster by avoiding the $O(|P|)$ time for comparison if we are willing to keep some extra values associated with the array.

• Follows Gusfield, section 7.14
Speedup #1

\[ u := \text{length of longest prefix of } U \text{ that matches a prefix of } P \]

\[ d := \text{length of longest prefix of } D \text{ that matches a prefix of } P \]

**Speedup:** maintain \( u,d \) throughout algorithm. Begin comparison of \( P \) with \( M \) at position \( \min(u,d) \)
**Speedup #2: Lcp(i,j) array**

**Def.** \( lcp(X,Y) \) is the length of the longest common prefix of strings \( X \) and \( Y \).

**Case #1:** \( lcp(U,M) > u = lcp(U,P) \)

\[ u \quad \downarrow \quad lcp(U,M) \]
\[ \quad \downarrow \quad \]
\[ \underline{x} \quad \cdots \quad U \]

\[ x < z \text{ because } U < P \text{ (since } P \in [U,D]\] \[ \implies M < P \text{ and therefore the new range should be } [M,D] \]

\[ \underline{x} \quad \cdots \quad M \]

\[ \underline{z} \quad \cdots \quad P \]
**Speedup #2: Case #2 & #3**

**Case #2:** \( \text{lcp}(U,M) < u = \text{lcp}(U,P) \)

\[ \begin{align*}
\text{lcp}(U,M) & \quad u \\
\hline
x & \cdots \quad U \\
\hline
\text{lcp}(U,M) & \quad u \\
\hline
x & \cdots \quad P \\
\hline
\text{lcp}(U,M) & \quad u \\
\hline
z & \cdots \quad M
\end{align*} \]

- \( x < z \) because \( U < M \) (by definition)
- \( \implies P < M \) and the new range should be \([U,M]\)

**Case #3:** \( \text{lcp}(U,M) = u = \text{lcp}(U,P) \)

\[ \begin{align*}
\text{lcp}(U,M) & \quad u \\
\hline
a & \cdots \quad U \\
\hline
b & \cdots \quad P \\
\hline
c & \cdots \quad M
\end{align*} \]

We have no information about \( a, b, c \) but we can start comparing at position \( u \).
**Algorithm**

**Case #0:** $u = d$: start comparing from position $u+1 = d+1$.

**Algorithm:** If $u = d$, apply case 0.

If $u > d$, apply case 1, 2 or 3 as appropriate.

If $u < d$, apply cases $1'$, $2'$, or $3'$ that are the symmetric version of cases 1,2,3 swapping D for U.
Proof. Only cases 0 and 3 (and 3’) actually compare any characters.

They always start comparing at max\((u, d)\) [for case 0 this is trivial, for case 3 this is true b/c we assume \(u > d\)].

If they match \(k\) characters of \(P\), then one of \(u\) or \(d\) will be incremented by \(k\), and those characters will never be compared again, so there are at most \(O(|P|)\) such comparisons.

The mismatch character may be compared more than once.

But there can be only 1 mismatch / iteration. There are \(O(\log m)\) iterations, so there are at most \(O(\log m)\) mismatches.

\[\therefore \text{Total # of comparisons} = O(|P| + \log m). \blacksquare\]
Pre-computing the $\text{Lcp}(i,j)$ values

**Notation.** $\text{Lcp}(i,j) :=$ longest common prefix between $A(i)$ and $A(j)$, where $A(i)$ is the suffix in position $i$ of the suffix array $A$.

$Lcp(i,j)$ values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?
Pre-computing the $Lcp(i,j)$ values

**Notation.** $Lcp(i,j) :=$ longest common prefix between $A(i)$ and $A(j)$, where $A(i)$ is the suffix in position $i$ of the suffix array $A$.

$Lcp(i,j)$ values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?
Pre-computing the Lcp(i,j) values

**Notation.** $\text{Lcp}(i,j) :=$ longest common prefix between $A(i)$ and $A(j)$, where $A(i)$ is the suffix in position $i$ of the suffix array $A$.

$Lcp(i,j)$ values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?
**Pre-computing the Lcp(i,j) values**

**Notation.** Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?
**Pre-computing the Lcp(i,j) values**

**Notation.** Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?
Pre-computing the $Lcp(i,j)$ values

**Notation.** $Lcp(i,j) :=$ longest common prefix between $A(i)$ and $A(j)$, where $A(i)$ is the suffix in position $i$ of the suffix array $A$.

$Lcp(i,j)$ values depend only on the suffix array.

While there are $O(m^2)$ possible values, only $O(m)$ of them will ever be accessed in any suffix array search. Why?

= complete binary tree with $m$ leaves.
Has $O(m)$ nodes entirely, so at most $O(m)$ ranges are considered.
Computing $\text{Lcp}(i, j)$

**Thm.** $\text{Lcp}(i,j) = \min \text{Lcp}(k, k+1)$, where $k = i,...,j-1$.

$L\text{cp}(k, k+1) \geq L\text{cp}(i, j)$ for all $k$ because everything in this range shares the same $L\text{cp}(i, j)$ prefix at least.

**Other consecutive pairs can have larger $L\text{cp}$, but not smaller than $k^*$ by the minimality of $L\text{cp}(k^*, k^*+1)$.**

$\Rightarrow$ By transitivity, $L\text{cp}(i, j) \geq L\text{cp}(k^*, k^*+1)$.

$L\text{cp}(k, k+1)$ can be computed in $O(m)$ time by traversing a suffix tree to find the depth of the lca of $k$ and $k+1$. 
Computing $Lcp(i, j)$

**Thm.** $Lcp(i,j) = \min Lcp(k, k+1)$, where $k = i, ..., j-1$.

$Lcp(k, k+1) \geq Lcp(i, j)$ for all $k$ because everything in this range shares the same $Lcp(i, j)$ prefix at least.

Other consecutive pairs can have larger $Lcp$, but not smaller than $k^*$ by the minimality of $Lcp(k^*, k^* + 1)$.

$\Rightarrow$ By transitivity, $Lcp(i, j) \geq Lcp(k^*, k^* + 1)$.

$Lcp(k, k+1)$ can be computed in $O(m)$ time by traversing a suffix tree to find the depth of the lca of $k$ and $k+1$. 
Computing $\text{Lcp}(i, j)$

**Thm.** $\text{Lcp}(i,j) = \min \text{Lcp}(k, k+1)$, where $k = i, \ldots, j-1$.

$Lcp(k,k+1) \geq Lcp(i,j)$ for all $k$ because everything in this range shares the same $Lcp(i,j)$ prefix at least.

Other consecutive pairs can have larger $Lcp$, but not smaller than $k^*$ by the minimality of $Lcp(k^*, k^* + 1)$.

$\Rightarrow$ By transitivity, $Lcp(i,j) \geq Lcp(k^*, k^* + 1)$.

$Lcp(k,k+1)$ can be computed in $O(m)$ time by traversing a suffix tree to find the depth of the lca of $k$ and $k+1$. 
• Suffix arrays can be used to search and count substrings.

• Construction:
  • Easily constructed in $O(n^2 \log n)$
  • Simple algorithms to construct them in $O(n)$ time.
  • More complicated algorithms to construct them in $O(n)$ time using even less space.

• More space efficient than suffix trees: just storing the original string + a list of integers.