Tomasz Kociumaka

Randomized Algorithms and Probabilistic Analysis of Algorithms


February 8, 2023
Definition

The *Longest Common Extension* $\text{LCE}(i, j)$ of positions $i, j$ of a string $T$ (of length $n$) is the length of the longest common prefix of the suffixes $T[i \ldots n]$ and $T[j \ldots n]$ of $T$. 

$LCE(16, 7) = 4$
The Longest Common Extension $LCE(i, j)$ of positions $i, j$ of a string $T$ (of length $n$) is the length of the longest common prefix of the suffixes $T[i..n]$ and $T[j..n]$ of $T$. 

$$LCE(16, 7) = 4$$

Used as a subroutine in many algorithms and data structures such as: approximate pattern matching (the kangaroo method), discovery of repetitions in strings, construction of text indexing data structures.
The **Longest Common Extension** $\text{LCE}(i, j)$ of positions $i, j$ of a string $T$ (of length $n$) is the length of the longest common prefix of the suffixes $T[i..n]$ and $T[j..n]$ of $T$. 

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

$\text{LCE}(16, 7) = 4$
Definition

The *Longest Common Extension* $\text{LCE}(i, j)$ of positions $i, j$ of a string $T$ (of length $n$) is the length of the longest common prefix of the suffixes $T[i..n]$ and $T[j..n]$ of $T$.

\[
\text{LCE}(16, 7) = 4
\]
Definition

The *Longest Common Extension* \( \text{LCE}(i, j) \) of positions \( i, j \) of a string \( T \) (of length \( n \)) is the length of the longest common prefix of the suffixes \( T[i..n] \) and \( T[j..n] \) of \( T \).

![Diagram showing LCE(16, 7) = 4](image.png)
The Longest Common Extension \( \text{LCE}(i, j) \) of positions \( i, j \) of a string \( T \) (of length \( n \)) is the length of the longest common prefix of the suffixes \( T[i..n] \) and \( T[j..n] \) of \( T \).

\[
\begin{aligned}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{aligned}
\]

\[
\text{LCE}(16, 7) = 4
\]

Used as a subroutine in many algorithms and data structures such as:

- approximate pattern matching (the kangaroo method),
- discovery of repetitions in strings,
- construction of text indexing data structures.
Components:

```
1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0
```
Components:

1. Lexicographic rank of each suffix,
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
Classic Data Structure for LCE Queries

Components:

1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

<table>
<thead>
<tr>
<th>LCP[i]</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>0010100</td>
<td>0</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>2</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>3</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>4</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>5</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>6</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>7</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>8</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>9</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>10</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>11</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>12</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>13</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>14</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>15</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>16</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>17</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>18</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>19</td>
</tr>
<tr>
<td>00101010100101000</td>
<td>20</td>
</tr>
</tbody>
</table>

Tomasz Kociumaka
Longest Common Extension Queries via String Synchronizing Sets
Classic Data Structure for LCE Queries

Components:

1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

<table>
<thead>
<tr>
<th>i</th>
<th>LCP[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>0010100</td>
</tr>
<tr>
<td>3</td>
<td>0010101</td>
</tr>
<tr>
<td>4</td>
<td>0010101010010100</td>
</tr>
<tr>
<td>5</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>6</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>7</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>8</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>9</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>10</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>11</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>12</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>13</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>14</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>15</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>16</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>17</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>18</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>19</td>
<td>001010101010010100</td>
</tr>
<tr>
<td>20</td>
<td>11010010101001010010010100</td>
</tr>
</tbody>
</table>

Tomasz Kociumaka  Longest Common Extension Queries via String Synchronizing Sets  3/10
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: $\text{LCE}(i, j) = \min \text{LCP}(\text{rank}[i] \ldots \text{rank}[j])$
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: \( \text{LCE}(i, j) = \min \text{LCP}(\text{rank}[i]..\text{rank}[j]) \)

LCE(16, 7) =

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>00</th>
<th>00101010010100100100</th>
<th>00101010010100100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>10100100</td>
<td>10100100</td>
<td></td>
</tr>
</tbody>
</table>
Components:

1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: \( \text{LCE}(i, j) = \min \text{LCP}([\text{rank}[i], \ldots, \text{rank}[j]]) \)

\[
\text{LCE}(16, 7) =
\]
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: \( \text{LCE}(i, j) = \min \text{LCP}(\text{rank}[i] \ldots \text{rank}[j]) \)

\( \text{LCE}(16, 7) = \)

\[ \begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
20 & 17 & 7 & 14 & 4 & 11 & 19 & 10 & 18 & 9 & 16 & 6 & 13 & 3 & 8 & 15 & 5 & 12 & 2 & 1 \\
\end{array} \]
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: \( \text{LCE}(i, j) = \min \text{LCP}\left(\text{rank}[i] .. \text{rank}[j]\right) \)

\( \text{LCE}(16, 7) = 4 \)

Efficiency:
- Query time: \( O(1) \);
- Data structure size: \( O(n) \);
- Construction time: \( O(n) \).
Classic Data Structure for LCE Queries

Components:
1. Lexicographic rank of each suffix,
2. LCP lengths for subsequent suffixes.

Queries: \( \text{LCE}(i, j) = \min \text{LCP}(\text{rank}[i], \ldots, \text{rank}[j]) \)

\( \text{LCE}(16, 7) = 4 \)

Efficiency:
- query time: \( O(1) \);
- data structure size: \( O(n) \);
- construction time: \( O(n) \).
Space-Efficient Data Structures for LCE Queries

Is $\mathcal{O}(n)$ space optimal?

The string $T$ can be encoded in $\mathcal{O}(n \log \sigma)$ bits, where $\sigma$ is the alphabet size. This is $\mathcal{O}(n / \log \sigma)$ words in the word RAM model. For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.

The arrays rank and LCP take approximately $n \log n$ bits each.

Questions:

1. What query time can be achieved in $\mathcal{O}(n \log \sigma)$ bits of space?

   Still $\mathcal{O}(1)$ query time, with $\mathcal{O}(n / \log \sigma)$ construction time.

2. What about $s$ extra space on top of $T$ for $1 \leq s \leq n / \log \sigma$?

   $\mathcal{O}(n / (s \log \sigma n))$ query time!
Is $\mathcal{O}(n)$ space optimal?

- The string $T$ can be encoded in $\mathcal{O}(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $\mathcal{O}(n/\log_\sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
Is $O(n)$ space optimal?

- The string $T$ can be encoded in $O(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $O(n/\log \sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
- The arrays rank and LCP take approximately $n \log n$ bits each.
Space-Efficient Data Structures for LCE Queries

Is $O(n)$ space optimal?
- The string $T$ can be encoded in $O(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $O(n/ \log \sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
- The arrays rank and LCP take approximately $n \log n$ bits each.

Questions:
1. What query time can be achieved in $O(n \log \sigma)$ bits of space?
Space-Efficient Data Structures for LCE Queries

Is $\mathcal{O}(n)$ space optimal?

- The string $T$ can be encoded in $\mathcal{O}(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $\mathcal{O}(n/\log_\sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
- The arrays rank and LCP take approximately $n \log n$ bits each.

Questions:

1. What query time can be achieved in $\mathcal{O}(n \log \sigma)$ bits of space?
   - Still $\mathcal{O}(1)$ query time, with $\mathcal{O}(n/\log_\sigma n)$ construction time.
Is $O(n)$ space optimal?

- The string $T$ can be encoded in $O(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $O(n/\log \sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
- The arrays rank and LCP take approximately $n \log n$ bits each.

Questions:

1. What query time can be achieved in $O(n \log \sigma)$ bits of space?
   - Still $O(1)$ query time, with $O(n/\log \sigma n)$ construction time.
2. What about $s$ extra space on top of $T$ for $1 \leq s \leq n/\log \sigma n$?
Space-Efficient Data Structures for LCE Queries

Is $O(n)$ space optimal?

- The string $T$ can be encoded in $O(n \log \sigma)$ bits, where $\sigma$ is the alphabet size.
  - This is $O(n/\log \sigma n)$ words in the word RAM model.
  - For example, if $\sigma = 2$, then 64 characters (bits) can be stored in an integer variable.
- The arrays rank and LCP take approximately $n \log n$ bits each.

Questions:

1. What query time can be achieved in $O(n \log \sigma)$ bits of space?
   - Still $O(1)$ query time, with $O(n/\log \sigma n)$ construction time.
2. What about $s$ extra space on top of $T$ for $1 \leq s \leq n/\log \sigma n$?
   - $O(n/(s \log \sigma n))$ query time!
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).
A naive algorithm computes \( \ell = \text{LCE}(i, j) \) in \( \mathcal{O}(1 + \ell) \) time (no data structure needed).

If the machine word fits \( \alpha \) characters, this can be improved to \( \mathcal{O}(1 + \ell/\alpha) \) time.

This is always \( \mathcal{O}(1 + \ell / \log_\sigma n) \) time because \( \alpha = \Omega(\log_\sigma n) \).
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).

If the machine word fits $\alpha$ characters, this can be improved to $O(1 + \ell/\alpha)$ time.

- This is always $O(1 + \ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

\[
\text{LCE}(2, 9) =
\]

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
\]
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).

If the machine word fits $\alpha$ characters, this can be improved to $O(1 + \ell/\alpha)$ time.

This is always $O(1 + \ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}
\]

$\text{LCE}(2, 9) =$

Extract $T[i \ldots i + \alpha]$: $1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1$

Extract $T[j \ldots j + \alpha]$: $1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1$
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).

- If the machine word fits $\alpha$ characters, this can be improved to $O(1 + \ell/\alpha)$ time.
  - This is always $O(1 + \ell/\log\sigma n)$ time because $\alpha = \Omega(\log\sigma n)$.

$$
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

$$
\text{LCE}(2, 9) = 
$$

Extract $T[i \ldots i + \alpha]$: 1 0 1 0 0 1 0 1
Extract $T[j \ldots j + \alpha]$: 1 0 1 0 1 0 0 1
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).

If the machine word fits $\alpha$ characters, this can be improved to $O(1 + \ell/\alpha)$ time.

This is always $O(1 + \ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.
A naive algorithm computes $\ell = \text{LCE}(i, j)$ in $O(1 + \ell)$ time (no data structure needed).

If the machine word fits $\alpha$ characters, this can be improved to $O(1 + \ell/\alpha)$ time.

This is always $O(1 + \ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

$L\text{CE}(2, 9) = 4$

Extract $T[i \ldots i + \alpha]$: 1 0 1 0 0 1 0 1 0 1 0 1 0 0

Extract $T[j \ldots j + \alpha]$: 1 0 1 0 1 0 0 1

$\text{xor}$: 0 0 0 0 1 1 0 0

$\text{clz}$: 4
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at synchronizing positions (set S).

```
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0
```
Build the rank and LCP tables only for suffixes starting at *synchronizing positions* (set $S$).

This would guarantee runtime $O(1 + \frac{n}{|S| \log \sigma n})$. 

Desired properties of a $\tau$-synchronizing set $S$:
- **Consistency**: $T[i..i+2\tau]$ determines if $i \in S$.
- **Density**: $S \cap [i..i+\tau) = \emptyset$ for every $i$.
- **Small size**: $|S| = O(\frac{n}{\tau})$.
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at *synchronizing positions* (set $S$).

2. Reduce arbitrary $\text{LCE}(i, j)$ to $\text{LCE}(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in S$.

$LCE(16, 7) =$

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
10 7 4 9 3 8 2 6 1 5

-  1 010100
  2 010100101000
  3 010100101000
  4 0101010100010100
  5 100
  6 10010100
  7 100101010100010100
  8 101010010100
  9 10101010010100
 10 11010010101010010100
```
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at *synchronizing positions* (set $S$).
2. Reduce arbitrary $\text{LCE}(i, j)$ to $\text{LCE}(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in S$.

$LCE(16, 7) = LCE(18, 9) + 2 =$
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at synchronizing positions (set S).
2. Reduce arbitrary LCE$(i, j)$ to LCE$(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in S$.

$LCE(16, 7) = LCE(18, 9) + 2 = 4$
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at synchronizing positions (set S).
2. Reduce arbitrary LCE\((i, j)\) to LCE\((i + \delta, j + \delta)\) for \(i + \delta, j + \delta \in S\).

LCE(16, 7) = LCE(18, 9) + 2 = 2 + 2 = 4
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at \textit{synchronizing positions} (set $S$).
2. Reduce arbitrary LCE$(i, j)$ to LCE$(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in S$.
3. Make sure that LCE$(i, j) = \mathcal{O}(n/|S|)$ or there is $\delta = \mathcal{O}(n/|S|)$ with $i + \delta, j + \delta \in S$.
   - This would guarantee runtime $\mathcal{O}(1 + n/(|S| \log_\sigma n))$.

Let $S = \{1, 4, 5, 10, 16, 17, 18, 19, 20\}$.

\[
\begin{array}{cccccccccccccccc}
10 & 7 & 4 & 9 & 3 & 8 & 2 & 6 & 1 & \color{red}5 & & & & & & & & & & \\
\end{array}
\]

LCE(16, 7) = LCE(18, 9) + 2 = 2 + 2 = 4
LCE Queries with a String Synchronizing Set

1. Build the rank and LCP tables only for suffixes starting at *synchronizing positions* (set $S$).
2. Reduce arbitrary $\text{LCE}(i, j)$ to $\text{LCE}(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in S$.
3. Make sure that $\text{LCE}(i, j) = \mathcal{O}(n/|S|)$ or there is $\delta = \mathcal{O}(n/|S|)$ with $i + \delta, j + \delta \in S$.
   - This would guarantee runtime $\mathcal{O}(1 + n/(|S| \log_{\sigma} n))$.

$LCE(16, 7) = LCE(18, 9) + 2 = 2 + 2 = 4$

**Desired properties of a $\tau$-synchronizing set $S$:**

- **Consistency** $T[i \ldots i + 2\tau)$ determines if $i \in S$.
- **Density** $S \cap [i \ldots i + \tau) = \emptyset$ for every $i$.
- **Small size** $|S| = \mathcal{O}(n/\tau)$.
Issues with Periodic Regions

**Desired properties of a $\tau$-synchronizing set $S$:**

- **consistency** $T[i \ldots i + 2\tau)$ determines if $i \in S$.
- **density** $S \cap [i \ldots i + \tau) = \emptyset$ for every $i$.
- **small size** $|S| = \mathcal{O}(n/\tau)$.

Issue: Such a set $S$ cannot always exist.

The problematic case is when $T$ contains length-$\tau$ substrings with period $o(\tau)$.

Workarounds:

1. Relax the density condition for periodic regions, slightly adapt the query algorithm. The actual solution with worst-case guarantees.
2. Give up on the small size if $T$ has many periodic regions. Reasonable for most real-life strings. Simple to explain and analyze.
Issues with Periodic Regions

**Desired properties of a $\tau$-synchronizing set $S$:**

- **consistency** $T[i..i + 2\tau)$ determines if $i \in S$.
- **density** $S \cap [i..i + \tau) = \emptyset$ for every $i$.
- **small size** $|S| = \mathcal{O}(n/\tau)$.

**Issue:** Such a set $S$ cannot always exist.

---

The problematic case is when $T$ contains length-$\tau$ substrings with period $o(\tau)$.

**Workarounds:**

1. Relax the density condition for periodic regions, slightly adapt the query algorithm.
2. Give up on the small size if $T$ has many periodic regions.
   - Reasonable for most real-life strings.
   - Simple to explain and analyze.
Issues with Periodic Regions

Desired properties of a $\tau$-synchronizing set $S$:

- **consistency** $T[i..i+2\tau)$ determines if $i \in S$.
- **density** $S \cap [i..i+\tau) = \emptyset$ for every $i$.
- **small size** $|S| = \mathcal{O}(n/\tau)$.

**Issue:** Such a set $S$ cannot always exist.

![Example string pattern](image-url)
Issues with Periodic Regions

Desired properties of a $\tau$-synchronizing set $S$:

- **consistency**: $T[i \ldots i + 2\tau)$ determines if $i \in S$.
- **density**: $S \cap [i \ldots i + \tau) = \emptyset$ for every $i$.
- **small size**: $|S| = \mathcal{O}(n/\tau)$.

**Issue**: Such a set $S$ cannot always exist.

The problematic case is when $T$ contains length-$\tau$ substrings with period $o(\tau)$.

![Example sequence](example_sequence.png)
Issues with Periodic Regions

**Desired properties of a \( \tau \)-synchronizing set \( S \):**

- **consistency** \( T[i..i+2\tau) \) determines if \( i \in S \).
- **density** \( S \cap [i..i+\tau) = \emptyset \) for every \( i \).
- **small size** \( |S| = \mathcal{O}(n/\tau) \).

**Issue:** Such a set \( S \) cannot always exist.

![Periodic String Example](image)

The problematic case is when \( T \) contains length-\( \tau \) substrings with period \( o(\tau) \).

**Workarounds:**

1. Relax the density condition for periodic regions, slightly adapt the query algorithm.
2. Give up on the small size if \( T \) has many periodic regions.
   Reasonable for most real-life strings.
   Simple to explain and analyze.
Issues with Periodic Regions

Desired properties of a $\tau$-synchronizing set $S$:

- **consistency** $T[i..i+2\tau)$ determines if $i \in S$.
- **density** $S \cap [i..i+\tau) = \emptyset$ for every $i$.
- **small size** $|S| = \mathcal{O}(n/\tau)$.

**Issue:** Such a set $S$ cannot always exist.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The problematic case is when $T$ contains length-$\tau$ substrings with period $o(\tau)$.

**Workarounds:**

1. Relax the density condition for periodic regions, slightly adapt the query algorithm.
   - The actual solution with worst-case guarantees.
Issues with Periodic Regions

**Desired properties of a $\tau$-synchronizing set $S$:**

- **consistency** $T[i..i + 2\tau)$ determines if $i \in S$.
- **density** $S \cap [i..i + \tau) = \emptyset$ for every $i$.
- **small size** $|S| = \mathcal{O}(n/\tau)$.

**Issue:** Such a set $S$ cannot always exist.

The problematic case is when $T$ contains length-$\tau$ substrings with period $o(\tau)$.

**Workarounds:**

1. Relax the density condition for periodic regions, slightly adapt the query algorithm.
   - The actual solution with worst-case guarantees.
2. Give up on the small size if $T$ has many periodic regions.
   - Reasonable for most real-life strings.
   - Simple to explain and analyze.
String Synchronizing Sets: Construction

Draw a random linear order on the set of length-\(\tau\) substrings of \(T\):

\[100 \prec 110 \prec 101 \prec 001 \prec 010\]

Add \(i\) to \(S\) if the earliest length-\(\tau\) substring of \(T[i..i+2\tau]\) is the prefix \(T[i..i+\tau]\), or the suffix \(T[i+\tau..i+2\tau]\).

Consistency and density are easy to argue.

If no length-\(\tau\) substrings of \(T[i..i+2\tau]\) has period \(o(\tau)\), then \(\Pr[i \in S] = O(1/\tau)\).

\(T[i..i+2\tau]\) contains \(\Omega(\tau)\) distinct length-\(\tau\) substrings.
Draw a random linear order on the set of length-$\tau$ substrings of $T$:

$$100 \prec 110 \prec 101 \prec 001 \prec 010$$
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   $100 \prec 110 \prec 101 \prec 001 \prec 010$

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i+2\tau)$ is
   - the prefix $T[i..i+\tau)$, or
   - the suffix $T[i+\tau..i+2\tau)$.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

$$100 \prec 110 \prec 101 \prec 001 \prec 010$$

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i+2\tau)$ is
   - the prefix $T[i..i+\tau)$, or
   - the suffix $T[i+\tau..i+2\tau)$.

$T[i..i+2\tau)$ contains $\Omega(\tau)$ distinct length-$\tau$ substrings.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-\(\tau\) substrings of \(T\):

\[
100 \prec 110 \prec 101 \prec 001 \prec 010
\]

2. Add \(i\) to \(S\) if the earliest length-\(\tau\) substring of \(T[i \ldots i+2\tau]\) is
   - the prefix \(T[i \ldots i+\tau]\), or
   - the suffix \(T[i+\tau \ldots i+2\tau]\).
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   $100 \prec 110 \prec 101 \prec 001 \prec 010$

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i \ldots i+2\tau)$ is
   - the prefix $T[i \ldots i+\tau)$, or
   - the suffix $T[i+\tau \ldots i+2\tau)$.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   100 ≺ 110 ≺ 101 ≺ 001 ≺ 010

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i+2\tau)$ is
   - the prefix $T[i..i+\tau)$, or
   - the suffix $T[i+\tau..i+2\tau)$. 

Consistency and density are easy to argue.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

\[
100 \prec 110 \prec 101 \prec 001 \prec 010
\]

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i+2\tau)$ is
   - the prefix $T[i..i+\tau)$, or
   - the suffix $T[i+\tau..i+2\tau)$.

Consistency and density are easy to argue.
Draw a random linear order on the set of length-$\tau$ substrings of $T$: 

$100 \prec 110 \prec 101 \prec 001 \prec 010$

Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i \ldots i + 2\tau)$ is

- the prefix $T[i \ldots i + \tau)$, or
- the suffix $T[i + \tau \ldots i + 2\tau)$.
1. Draw a random linear order on the set of length-τ substrings of $T$:

$$100 \prec 110 \prec 101 \prec 001 \prec 010$$

2. Add $i$ to $S$ if the earliest length-τ substring of $T[i..i+2τ)$ is

- the prefix $T[i..i+τ)$, or
- the suffix $T[i+τ..i+2τ)$.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   \[100 \prec 110 \prec 101 \prec 001 \prec 010\]

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i+2\tau)$ is
   - the prefix $T[i..i+\tau)$, or
   - the suffix $T[i+\tau..i+2\tau)$.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   $100 \prec 110 \prec 101 \prec 001 \prec 010$

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i..i + 2\tau)$ is
   - the prefix $T[i..i + \tau)$, or
   - the suffix $T[i + \tau..i + 2\tau)$.

Consistency and density are easy to argue.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-τ substrings of $T$:

$$100 \prec 110 \prec 101 \prec 001 \prec 010$$

2. Add $i$ to $S$ if the earliest length-τ substring of $T[i..i+2τ)$ is
   - the prefix $T[i..i+τ)$, or
   - the suffix $T[i+τ..i+2τ)$.

3. Consistency and density are easy to argue.
String Synchronizing Sets: Construction

1. Draw a random linear order on the set of length-$\tau$ substrings of $T$:

   \[100 \prec 110 \prec 101 \prec 001 \prec 010\]

2. Add $i$ to $S$ if the earliest length-$\tau$ substring of $T[i \ldots i + 2\tau]$ is
   - the prefix $T[i \ldots i + \tau]$, or
   - the suffix $T[i + \tau \ldots i + 2\tau]$.

3. Consistency and density are easy to argue.

4. If no length-$\tau$ substrings of $T[i \ldots i + 2\tau]$ has period $o(\tau)$, then $\Pr[i \in S] = \Theta(1/\tau)$.
   - $T[i \ldots i + 2\tau]$ contains $\Omega(\tau)$ distinct length-$\tau$ substrings.
LCE queries in small space:

- A \( \tau \)-synchronizing \( S \) set with \( \tau = \Theta(n/s) \).
- Expected size \( \text{Exp}[|S|] = \mathcal{O}(n/\tau) \).
  - Requires adaptations for periodic regions or assuming no length-\( \tau \) substring has period \( o(\tau) \).
Summary

LCE queries in small space:

- A $\tau$-synchronizing $S$ set with $\tau = \Theta(n/s)$.
- Expected size $\text{Exp}[|S|] = O(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-$\tau$ substring has period $o(\tau)$.
- Worst-case size $O(n/\tau)$ after $O(1)$ attempts (in expectation).
Summary

**LCE queries in small space:**

- A $\tau$-synchronizing $S$ set with $\tau = \Theta(n/s)$.
- Expected size $\text{Exp}[|S|] = \mathcal{O}(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-$\tau$ substring has period $o(\tau)$.
- Worst-case size $\mathcal{O}(n/\tau)$ after $\mathcal{O}(1)$ attempts (in expectation).
- The construction can be derandomized using the method of pessimistic estimators.
Summary

**LCE queries in small space:**

- A $\tau$-synchronizing $S$ set with $\tau = \Theta(n/s)$.
- Expected size $\text{Exp}[|S|] = O(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-$\tau$ substring has period $o(\tau)$.
- Worst-case size $O(n/\tau)$ after $O(1)$ attempts (in expectation).
- The construction can be derandomized using the method of pessimistic estimators.

**Final data structure:**

$$
\begin{align*}
\text{size} & \quad O(|S|) = O(n/\tau) \\
\text{query time} & \quad O(1 + \tau/\log_\sigma n)
\end{align*}
$$

for $\tau = \Theta(\log_\sigma n)$

$$
\begin{align*}
O(n/\log_\sigma n) & \quad O(1)
\end{align*}
$$

Open question:

$O(n/\log_\sigma n)$ construction time for $\tau = \Omega(\log_\sigma n)$. 
Summary

LCE queries in small space:

- A $\tau$-synchronizing $S$ set with $\tau = \Theta(n/s)$.
- Expected size $\text{Exp}[|S|] = O(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-$\tau$ substring has period $o(\tau)$.
- Worst-case size $O(n/\tau)$ after $O(1)$ attempts (in expectation).
- The construction can be derandomized using the method of pessimistic estimators.

Final data structure:

<table>
<thead>
<tr>
<th></th>
<th>for $\tau = \Theta(\log_\sigma n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>size $O(</td>
<td>S</td>
</tr>
<tr>
<td>query time $O(1 + \tau/\log_\sigma n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>construction time $O(n)$</td>
<td>$O(n/\log_\sigma n)$</td>
</tr>
</tbody>
</table>
Summary

LCE queries in small space:
- A $\tau$-synchronizing $S$ set with $\tau = \Theta(n/s)$.
- Expected size $\text{Exp}[|S|] = \mathcal{O}(n/\tau)$.
- Requires adaptations for periodic regions or assuming no length-$\tau$ substring has period $o(\tau)$.
- Worst-case size $\mathcal{O}(n/\tau)$ after $\mathcal{O}(1)$ attempts (in expectation).
- The construction can be derandomized using the method of pessimistic estimators.

Final data structure:
- size $\mathcal{O}(|S|) = \mathcal{O}(n/\tau)$
- query time $\mathcal{O}(1 + \tau / \log_\sigma n)$
- construction time $\mathcal{O}(n)$

for $\tau = \Theta(\log_\sigma n)$

$\mathcal{O}(n/\log_\sigma n)$

$\mathcal{O}(1)$

$\mathcal{O}(n/\log_\sigma n)$

Open question:
- $\mathcal{O}(n/\log_\sigma n)$ construction time for $\tau = \Omega(\log_\sigma n)$. 
Bigger Picture

Further applications of synchronizing sets:

- **KRRW, SODA’15** Internal Pattern Matching queries, Period queries
- **KK, STOC’19** Burrows–Wheeler Transform construction (bzip2, etc.)
- **CKPR, ESA’21** Longest Common Substring problem
Further applications of synchronizing sets:

KRRW, SODA’15 Internal Pattern Matching queries, Period queries

KK, STOC’19 Burrows–Wheeler Transform construction (bzip2, etc.)

CKPR, ESA’21 Longest Common Substring problem

KK, STOC’22 dynamic suffix array

Implementations:

DFH, ESA’20 LCE queries, ignoring the periodic case

in progress LCE queries, with worst-case guarantees

Related techniques in applications (w/o theoretical guarantees):

Bioinformatics (minimizers)
Plagiarism detection (winnowing)
Storage systems (content-based chunking)
Further applications of synchronizing sets:

- **KRRW, SODA’15** Internal Pattern Matching queries, Period queries
- **KK, STOC’19** Burrows–Wheeler Transform construction (bzip2, etc.)
- **CKPR, ESA’21** Longest Common Substring problem
- **KK, STOC’22** dynamic suffix array
- **AJ, SODA’22** quantum algorithm for Longest Common Substring
- **JN, SODA’23** quantum data structure for LCE queries
Further applications of synchronizing sets:

**KRRW, SODA’15** Internal Pattern Matching queries, Period queries

**KK, STOC’19** Burrows–Wheeler Transform construction (bzip2, etc.)

**CKPR, ESA’21** Longest Common Substring problem

**KK, STOC’22** dynamic suffix array

**AJ, SODA’22** quantum algorithm for Longest Common Substring

**JN, SODA’23** quantum data structure for LCE queries

**KK, SODA’23** compressed suffix arrays and text indexes
Further applications of synchronizing sets:

- **KRRW, SODA’15** Internal Pattern Matching queries, Period queries
- **KK, STOC’19** Burrows–Wheeler Transform construction (bzip2, etc.)
- **CKPR, ESA’21** Longest Common Substring problem
- **KK, STOC’22** dynamic suffix array
- **AJ, SODA’22** quantum algorithm for Longest Common Substring
- **JN, SODA’23** quantum data structure for LCE queries
- **KK, SODA’23** compressed suffix arrays and text indexes

Implementations:

- **DFHKK, ESA’20** LCE queries, ignoring the periodic case
- **in progress** LCE queries, with worst-case guarantees
Further applications of synchronizing sets:

**KRRW, SODA’15** Internal Pattern Matching queries, Period queries

**KK, STOC’19** Burrows–Wheeler Transform construction (bzip2, etc.)

**CKPR, ESA’21** Longest Common Substring problem

**KK, STOC’22** dynamic suffix array

**AJ, SODA’22** quantum algorithm for Longest Common Substring

**JN, SODA’23** quantum data structure for LCE queries

**KK, SODA’23** compressed suffix arrays and text indexes

Implementations:

**DFHKK, ESA’20** LCE queries, ignoring the periodic case

in progress  LCE queries, with worst-case guarantees

Related techniques in applications (w/o theoretical guarantees):

- Bioinformatics (minimizers)
- Plagiarism detection (winnowing)
- Storage systems (content-based chunking)