Longest Common Extension Queries via String Synchronizing Sets

Tomasz Kociumaka

Randomized Algorithms and Probabilistic Analysis of Algorithms

[1] Tomasz Kociumaka. Efficient Data Structures for Internal Queries in Texts. PhD thesis, University of Warsaw, 2018.

[2] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. STOC 2019.

[3] Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical Performance of Space Efficient Data Structures for Longest Common Extensions. ESA 2020.

February 8, 2023

Longest Common Extension Queries

Definition

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

Longest Common Extension Queries

Definition

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

Longest Common Extension Queries

Definition

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 $T[i\mathinner{.\,.} n]$ and $T[j\mathinner{.\,.} n]$ of T.

$$\begin{array}{cccccccccccccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 & \scriptstyle 12 & \scriptstyle 13 & \scriptstyle 14 & \scriptstyle 15 & \scriptstyle 16 & \scriptstyle 17 & \scriptstyle 18 & \scriptstyle 19 & \scriptstyle 20 \\ 1 & 1 & 0 & 1 & 0 & 0| & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0| & 1 & 0 & 1 & 0 & 0 \end{array}$$

$$\mathsf{LCE}(16,7) =$$

Longest Common Extension Queries

Definition

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

$$\mathsf{LCE}(16,7) =$$

Definition

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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

Longest Common Extension Queries

Definition

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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

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.

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

Components:

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

1. Lexicographic rank of each suffix,

i	
1	0
2	00
3	0010100
4	0010101010010100
5	0100
6	010010100
7	01001010101010010100
8	010100
9	01010010100
10	0101010010100
11	010101010010100
12	100
13	10010100
14	10010101010010100
15	10100
16	1010010100
17	101001010101010010100
18	101010010100
19	10101010010100
20	1101001010101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

1. Lexicographic rank of each suffix,

i	
1	0
2	00
3	0010100
4	0010101010010100
5	0100
6	010010100
7	0100101010100101 00
8	010100
9	01010010100
10	0101010010100
11	010101010010100
12	100
13	10010100
14	100101010100101 00
15	10100
16	1010010100
17	10100101010100101 00
18	101010010100
19	10101010010100
20	1101001010101001010 0

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

LCP$[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	010010101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	15	10100
5	16	1010010100
9	17	1010010101010010100
4	18	101010010100
6	19	10101010010100
1	20	11010010101010010100

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

LCP$[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	0100101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	01010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	15	10100
5	16	1010010100
9	17	10100101010010100
4	18	101010010100
6	19	1010101010010100
1	20	110100101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

$\mathsf{LCP}[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	010010101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	15	10100
5	16	1010010100
9	17	1010010101010010100
4	18	101010010100
6	19	10101010010100
1	20	11010010101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

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

LCP$[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	010010101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	15	10100
5	16	1010010100
9	17	1010010101010010100
4	18	101010010100
6	19	10101010010100
1	20	110100101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

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

LCP$[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	01001010101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	1001010101010010100
2	**15**	10100
5	16	1010010100
9	17	101001010101010010100
4	18	101010010100
6	**19**	10101010010100
1	20	110100101010101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

$$\mathsf{LCE}(16,7) =$$

LCP[i]	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	0100101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	**15**	10100
5	16	1010010100
9	17	1010010101010010100
4	18	101010010100
6	**19**	10101010010100
1	20	1101001010101010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

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

LCP$[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	010010101010010100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	**15**	10100
$\boxed{5}$	16	1010010100
$\boxed{9}$	17	1010010101010010100
$\boxed{\mathbf{4}}$	18	101010010100
$\boxed{6}$	**19**	10101010010100
1	20	11010010101010010100

Classic Data Structure for LCE Queries

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	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

Components:

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

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

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

Efficiency:

- query time: $\mathcal{O}(1)$;
- data structure size: $\mathcal{O}(n)$;
- construction time: $\mathcal{O}(n)$.

$\mathsf{LCP}[i]$	i	
-	1	0
1	2	00
2	3	0010100
6	4	0010101010010100
1	5	0100
4	6	010010100
8	7	0100101010100 10100
3	8	010100
6	9	01010010100
5	10	0101010010100
7	11	010101010010100
0	12	100
3	13	10010100
7	14	10010101010010100
2	**15**	10100
5	16	1010010100
9	17	1010010101010010100
4	18	101010010100
6	**19**	10101010010100
1	20	11010010101010010100

Space-Efficient Data Structures for LCE Queries

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

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

- The string T can be encoded in $\mathcal{O}(n \log \sigma)$ bits, where σ 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 $\mathcal{O}(n)$ space optimal?

- The string T can be encoded in $\mathcal{O}(n \log \sigma)$ bits, where σ 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.

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 σ 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?

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 σ 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.

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 σ 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.
2. What about s *extra* space *on top of* T for $1 \le s \le n/\log_\sigma n$?

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 σ 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.
2. What about s *extra* space *on top of* T for $1 \le s \le n/\log_\sigma n$?
   - $\mathcal{O}(n/(s \log_\sigma n))$ query time!

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i, j)$ in $\mathcal{O}(1 + \ell)$ time (no data structure needed).

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

$$\begin{array}{cccccccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ 1 & \boxed{1\ 0\ 1\ 0\ 0\ 1\ 0\ 1} & & & & & & & & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

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

Extract $T[i \mathinner{.\,.} i+\alpha)$: 1 0 1 0 0 1 0 1

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

$$\begin{array}{cccccccccccccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 & \scriptstyle 12 & \scriptstyle 13 & \scriptstyle 14 & \scriptstyle 15 & \scriptstyle 16 & \scriptstyle 17 & \scriptstyle 18 & \scriptstyle 19 & \scriptstyle 20 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

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

Extract $T[i\mathinner{.\,.} i+\alpha)$: 1 0 1 0 0 1 0 1
Extract $T[j\mathinner{.\,.} j+\alpha)$: 1 0 1 0 1 0 0 1

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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

Extract $T[i \mathinner{.\,.} i+\alpha)$:	1	0	1	0	0	1	0	1
Extract $T[j \mathinner{.\,.} j+\alpha)$:	1	0	1	0	1	0	0	1
xor:	0	0	0	0	1	1	0	0

Exploiting Bit-Parallelism

- A naive algorithm computes $\ell = \mathsf{LCE}(i,j)$ in $\mathcal{O}(1+\ell)$ time (no data structure needed).
- If the machine word fits α characters, this can be improved to $\mathcal{O}(1+\ell/\alpha)$ time.
  - This is always $\mathcal{O}(1+\ell/\log_\sigma n)$ time because $\alpha = \Omega(\log_\sigma n)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

$$\mathsf{LCE}(2,9) = 4$$

Extract $T[i\mathinner{.\,.} i+\alpha)$:	1	0	1	0	0	1	0	1
Extract $T[j\mathinner{.\,.} j+\alpha)$:	1	0	1	0	1	0	0	1
`xor`:	0	0	0	0	1	1	0	0
`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	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\underline{1}$	1	0	$\underline{1}$	0	$\underline{0}$	$\underline{1}$	$\underline{0}$	$\underline{1}$	$\underline{0}$	1	0	$\underline{1}$	0	$\underline{0}$	1	0	$\underline{1}$	0	0

LCE Queries with a String Synchronizing Set

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	8	2			6		1			5		

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	5	100
3	6	10010100
7	7	10010101010010100
2	8	101010010100
6	9	10101010010100
1	10	1101001010101010010100

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.

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	8	2			6		1			5		

LCE$(16, 7) =$

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	5	100
3	6	10010100
7	7	10010101010010100
2	8	101010010100
6	9	10101010010100
1	10	1101001010101010010100

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.

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	**8**	2			6		1			**5**		

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

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	**5**	100
3	6	10010100
7	7	1001010101010010100
2	**8**	101010010100
6	9	10101010010100
1	10	110100101010101010010100

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.

position	1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
text	1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
rank	10			7		4	9	3	**8**	2			6		1			**5**		

$$\text{LCE}(16, 7) = \text{LCE}(18, 9) + 2 =$$

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	**5**	100
$\boxed{3}$	6	10010100
$\boxed{7}$	7	10010101010010100
$\boxed{2}$	**8**	101010010100
6	9	10101010010100
1	10	110100101010100 10100

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.

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	**8**	2			6		1			**5**		

$$\mathsf{LCE}(16,7) = \mathsf{LCE}(18,9) + 2 = 2 + 2 = 4$$

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	**5**	100
3	6	10010100
7	7	10010101010010100
2	**8**	101010010100
6	9	10101010010100
1	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 $\mathsf{LCE}(i, j)$ to $\mathsf{LCE}(i + \delta, j + \delta)$ for $i + \delta, j + \delta \in \mathsf{S}$.
3. Make sure that $\mathsf{LCE}(i, j) = \mathcal{O}(n/|\mathsf{S}|)$ or there is $\delta = \mathcal{O}(n/|\mathsf{S}|)$ with $i + \delta, j + \delta \in \mathsf{S}$.
   - This would guarantee runtime $\mathcal{O}(1 + n/(|\mathsf{S}| \log_\sigma n))$.

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	**8**	2			6		1			**5**		

$$\mathsf{LCE}(16, 7) = \mathsf{LCE}(18, 9) + 2 = 2 + 2 = 4$$

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	**5**	100
3	6	10010100
7	7	100101010010100
2	**8**	101010010100
6	9	10101010010100
1	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 $\mathsf{LCE}(i, j)$ to $\mathsf{LCE}(i+\delta, j+\delta)$ for $i+\delta, j+\delta \in \mathsf{S}$.
3. Make sure that $\mathsf{LCE}(i, j) = \mathcal{O}(n/|\mathsf{S}|)$ or there is $\delta = \mathcal{O}(n/|\mathsf{S}|)$ with $i+\delta, j+\delta \in \mathsf{S}$.
   - This would guarantee runtime $\mathcal{O}(1 + n/(|\mathsf{S}| \log_\sigma n))$.

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	**16**	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0
10			7		4	9	3	**8**	2			6		1			**5**		

$\mathsf{LCE}(16, 7) = \mathsf{LCE}(18, 9) + 2 = 2 + 2 = 4$

-	1	010100
6	2	01010010100
5	3	0101010010100
7	4	010101010010100
0	**5**	100
$\boxed{3}$	6	10010100
$\boxed{7}$	7	10010101010010100
$\boxed{\mathbf{2}}$	**8**	101010010100
6	9	10101010010100
1	10	11010010101010010100

Desired properties of a τ-synchronizing set S:

consistency $T[i \mathinner{.\,.} i+2\tau)$ determines if $i \in \mathsf{S}$.

density $\mathsf{S} \cap [i \mathinner{.\,.} i+\tau) = \emptyset$ for every i.

small size $|\mathsf{S}| = \mathcal{O}(n/\tau)$.

Desired properties of a τ-synchronizing set S:

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

Issues with Periodic Regions

Desired properties of a τ-synchronizing set S:

consistency	$T[i\mathinner{.\,.} i+2\tau)$ determines if $i \in \mathsf{S}$.
density	$\mathsf{S} \cap [i\mathinner{.\,.} i+\tau) = \emptyset$ for every i.
small size	$\lvert\mathsf{S}\rvert = \mathcal{O}(n/\tau)$.

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

Desired properties of a τ-synchronizing set S:

consistency $T[i\,..\,i+2\tau)$ determines if $i \in \mathsf{S}$.

density $\mathsf{S} \cap [i\,..\,i+\tau) = \emptyset$ for every i.

small size $|\mathsf{S}| = \mathcal{O}(n/\tau)$.

Issue: Such a set S cannot always exist.

$$\underline{0}\ 0\ 0\ 0\ 0$$

Issues with Periodic Regions

Desired properties of a τ-synchronizing set S:

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

Issue: Such a set S cannot always exist.

$$\underline{0}\ 0\ 0\ 0\ 0$$

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

Issues with Periodic Regions

Desired properties of a τ-synchronizing set S:

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

Issue: Such a set S cannot always exist.

$$\underline{0}\ 0\ 0\ 0\ 0$$

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

Workarounds:

Issues with Periodic Regions

Desired properties of a τ-synchronizing set S:

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

Issue: Such a set S cannot always exist.

$$\underline{0}\ 0\ 0\ 0\ 0$$

The problematic case is when T contains length-τ 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 τ-synchronizing set S:

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

Issue: Such a set S cannot always exist.

$$\underline{0}\ 0\ 0\ 0\ 0$$

The problematic case is when T contains length-τ 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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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

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

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

$$\begin{array}{cccccccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ \boxed{1 & 1 & 0 & 1 & 0 & 0} & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

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 $T[i \,..\, i+2\tau)$ is
   - the prefix $T[i \,..\, i+\tau)$, or
   - the suffix $T[i+\tau \,..\, i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i\mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i\mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau\mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i\mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i\mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau\mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau \mathinner{.\,.} i+2\tau)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0

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 $T[i \mathinner{.\,.} i + 2\tau)$ is
   - the prefix $T[i \mathinner{.\,.} i + \tau)$, or
   - the suffix $T[i + \tau \mathinner{.\,.} i + 2\tau)$.

String Synchronizing Sets: Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\underline{1}$	1	0	$\underline{1}$	0	$\underline{0}$	$\underline{1}$	$\underline{0}$	$\underline{1}$	$\underline{0}$	1	0	$\underline{1}$	0	$\underline{0}$	1	0	$\underline{1}$	0	0

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 $T[i\,..\,i+2\tau)$ is
   - the prefix $T[i\,..\,i+\tau)$, or
   - the suffix $T[i+\tau\,..\,i+2\tau)$.
3. Consistency and density are easy to argue.

String Synchronizing Sets: Construction

$$\begin{array}{cccccccccccccccccccc} 1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20\\ \underline{1}&1&0&\underline{1}&0&\underline{0}&\underline{1}&\underline{0}&\underline{1}&\underline{0}&1&0&\underline{1}&0&\underline{0}&1&0&\underline{1}&0&0 \end{array}$$

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 $T[i\mathinner{.\,.} i+2\tau)$ is
   - the prefix $T[i\mathinner{.\,.} i+\tau)$, or
   - the suffix $T[i+\tau\mathinner{.\,.} i+2\tau)$.
3. Consistency and density are easy to argue.
4. If no length-τ substrings of $T[i\mathinner{.\,.} i+2\tau)$ has period $o(\tau)$, then $\Pr[i \in \mathsf{S}] = \mathcal{O}(1/\tau)$.
   - $T[i\mathinner{.\,.} i+2\tau)$ contains $\Omega(\tau)$ distinct length-τ substrings.

Summary

LCE queries in small space:

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

Summary

LCE queries in small space:

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

Summary

LCE queries in small space:

- A τ-synchronizing S set with $\tau = \Theta(n/s)$.
- Expected size $\mathrm{Exp}[|S|] = \mathcal{O}(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-τ 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.

LCE queries in small space:

- A τ-synchronizing S set with $\tau = \Theta(n/s)$.
- Expected size $\mathrm{Exp}[|S|] = \mathcal{O}(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-τ 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:

		for $\tau = \Theta(\log_\sigma n)$
size	$\mathcal{O}(|\mathsf{S}|) = \mathcal{O}(n/\tau)$	$\mathcal{O}(n/\log_\sigma n)$
query time	$\mathcal{O}(1 + \tau/\log_\sigma n)$	$\mathcal{O}(1)$

Summary

LCE queries in small space:

- A τ-synchronizing S set with $\tau = \Theta(n/s)$.
- Expected size $\mathrm{Exp}[|S|] = \mathcal{O}(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-τ 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:

		for $\tau = \Theta(\log_\sigma n)$
size	$\mathcal{O}(\lvert\mathsf{S}\rvert) = \mathcal{O}(n/\tau)$	$\mathcal{O}(n/\log_\sigma n)$
query time	$\mathcal{O}(1 + \tau/\log_\sigma n)$	$\mathcal{O}(1)$
construction time	$\mathcal{O}(n)$	$\mathcal{O}(n/\log_\sigma n)$

Summary

LCE queries in small space:

- A τ-synchronizing S set with $\tau = \Theta(n/s)$.
- Expected size $\mathrm{Exp}[|S|] = \mathcal{O}(n/\tau)$.
  - Requires adaptations for periodic regions or assuming no length-τ 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:		for $\tau = \Theta(\log_\sigma n)$
size	$\mathcal{O}(\lvert\mathsf{S}\rvert) = \mathcal{O}(n/\tau)$	$\mathcal{O}(n/\log_\sigma n)$
query time	$\mathcal{O}(1 + \tau/\log_\sigma n)$	$\mathcal{O}(1)$
construction time	$\mathcal{O}(n)$	$\mathcal{O}(n/\log_\sigma n)$

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
K**K**, STOC'19	Burrows–Wheeler Transform construction (`bzip2`, etc.)
C**K**PR, ESA'21	Longest Common Substring problem

Bigger Picture

Further applications of synchronizing sets:

KRRW, SODA'15	Internal Pattern Matching queries, Period queries
K**K**, STOC'19	Burrows–Wheeler Transform construction (`bzip2`, etc.)
C**K**PR, ESA'21	Longest Common Substring problem
K**K**, STOC'22	dynamic suffix array

Bigger Picture

Further applications of synchronizing sets:

KRRW, SODA'15	Internal Pattern Matching queries, Period queries
K**K**, STOC'19	Burrows–Wheeler Transform construction (`bzip2`, etc.)
C**K**PR, ESA'21	Longest Common Substring problem
K**K**, STOC'22	dynamic suffix array
AJ, SODA'22	quantum algorithm for Longest Common Substring
JN, SODA'23	quantum data structure for LCE queries

Bigger Picture

Further applications of synchronizing sets:

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

Bigger Picture

Further applications of synchronizing sets:

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

Implementations:

DFH**K**K, ESA'20	LCE queries, ignoring the periodic case
in progress	LCE queries, with worst-case guarantees

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
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)