Linear Time Runs over General Ordered Alphabets

Jonas Ellert, Johannes Fischer

technische universität
dortmund

department of
computer science

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$
- it is maximal: $S[2] \neq S[2+5]$ and $S[16] \neq S[16-5]$

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$
- it is maximal: $S[2] \neq S[2+5]$ and $S[16] \neq S[16-5]$

[Bannai et al. 2017]: at most n runs in length-n string

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$
- it is maximal: $S[2] \neq S[2+5]$ and $S[16] \neq S[16-5]$

[Bannai et al. 2017]: at most n runs in length-n string

[Kolpakov and Kucherov 1999]: $\mathcal{O}(n)$ time for linearly-sortable alphabets

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$
- it is maximal: $S[2] \neq S[2+5]$ and $S[16] \neq S[16-5]$

[Bannai et al. 2017]: at most n runs in length-n string

[Kolpakov and Kucherov 1999]: $\mathcal{O}(n)$ time for linearly-sortable alphabets

[Crochemore et al. 2016]: $\mathcal{O}(n\alpha(n))$ time for general ordered alphabets

Maximal Periodic Substrings

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
S	$=$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

The substring $S[3, 15]$ is a run because

- it is a repetition $\mu^e = (\texttt{bcaba})^{2.6}$ with exponent $e \geq 2$
- it is maximal: $S[2] \neq S[2+5]$ and $S[16] \neq S[16-5]$

[Bannai et al. 2017]: at most n runs in length-n string

[Kolpakov and Kucherov 1999]: $\mathcal{O}(n)$ time for linearly-sortable alphabets

[Crochemore et al. 2016]: $\mathcal{O}(n\alpha(n))$ time for general ordered alphabets

[this work]: $\mathcal{O}(n)$ time for general ordered alphabets

A Note on Alphabet Types

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time

$\mathcal{O}(n \lg \sigma)$

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

$\mathcal{O}(n \lg \sigma)$

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

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv			
Suffix Sorting			

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$		
Suffix Sorting	$\Theta(n)$		

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$	$\Theta(n \lg \sigma)$	
Suffix Sorting	$\Theta(n)$	$\Theta(n \lg \sigma)$	

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
  $\mathcal{O}(n \lg \sigma)$
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time
  $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Compute All Runs	$\boldsymbol{\Theta(n)}$		
	Kolpakov & Kucherov '99		

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Compute All Runs	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\Theta(n)}$	
	Kolpakov & Kucherov '99	**this work '21**	

A Note on Alphabet Types

- **linearly-sortable alphabet (LSA)**:
  sort n symbols in $\mathcal{O}(n)$ time
  e.g. $\{1, \ldots, n^{\mathcal{O}(1)}\}$
- **general ordered alphabet (GOA)**:
  test $c_1 < c_2$ in constant time
- **general unordered alphabet (GUA)**:
  test $c_1 = c_2$ in constant time

GOA → LSA: $\mathcal{O}(n \lg \sigma)$

GUA → LSA: $\mathcal{O}(n\sigma)$

	LSA	GOA	GUA
Lempel-Ziv	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Suffix Sorting	$\Theta(n)$	$\Theta(n \lg \sigma)$	$\Theta(n\sigma)$
Compute All Runs	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\Theta(n)}$	$\boldsymbol{\mathcal{O}(n \lg n)}$
	Kolpakov & Kucherov '99	**this work '21**	**e.g. Main & Lorentz '94**

- A Note on Alphabet Types
- **Reduction of Runs to Next Smaller Suffixes and LCEs**
- Linear Time Next Smaller Suffixes
- Linear Time LCEs
- Practical Aspects & Conclusion

Maximal Periodic Substrings

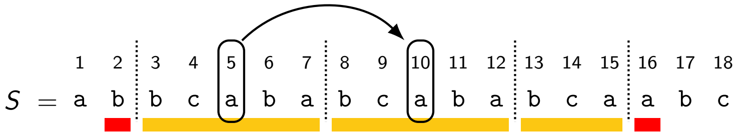
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

Maximal Periodic Substrings

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

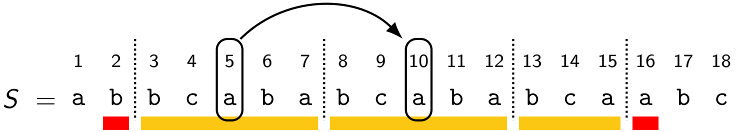
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



$$S_5 = \texttt{a b a b c a b} \ldots$$

[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings

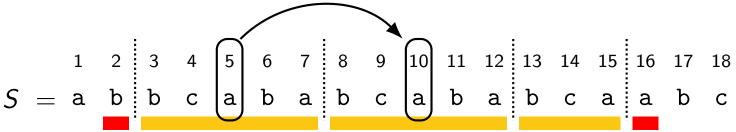
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$S =$	a	b	b	c	a	b	a	b	c	a	b	a	b	c	a	a	b	c

S_5 = a b a b c a b . . .

S_6 = b . . .

[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



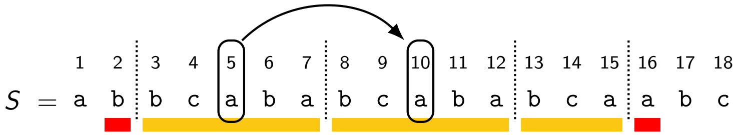
S_5 = a b a b c a b ...

S_6 = **b** ...

S_7 = a b **c** ...

[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

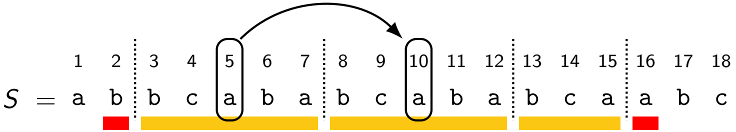
Maximal Periodic Substrings



$$S_5 = \texttt{a b a b c a b} \ldots$$
$$S_6 = \color{red}{\texttt{b}} \ldots$$
$$S_7 = \texttt{a b } \color{red}{\texttt{c}} \ldots$$
$$S_8 = \color{red}{\texttt{b}} \ldots$$

[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

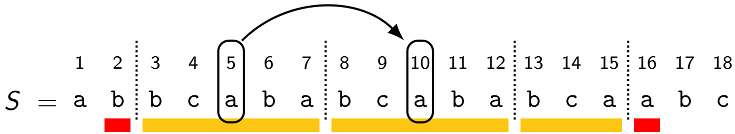
Maximal Periodic Substrings



$$S_5 = \texttt{a b a b c a b} \ldots$$
$$S_6 = \texttt{b} \ldots$$
$$S_7 = \texttt{a b c} \ldots$$
$$S_8 = \texttt{b} \ldots$$
$$S_9 = \texttt{c} \ldots$$

[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

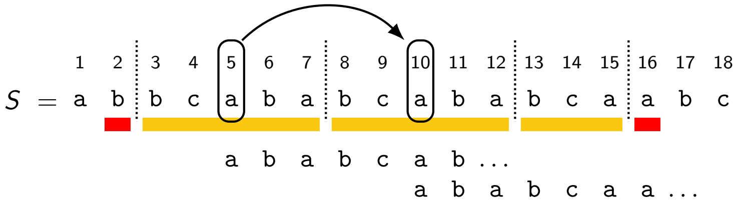
Maximal Periodic Substrings



$$S_5 = \mathtt{a\ b\ a\ b\ c\ a\ b} \ldots$$
$$S_6 = \mathtt{b} \ldots$$
$$S_7 = \mathtt{a\ b\ c} \ldots$$
$$S_8 = \mathtt{b} \ldots$$
$$S_9 = \mathtt{c} \ldots$$
$$S_{10} = \mathtt{a\ b\ a\ b\ c\ a\ a} \ldots$$

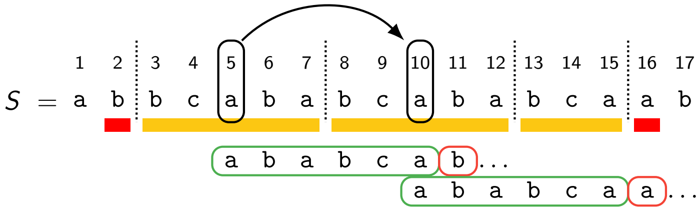
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



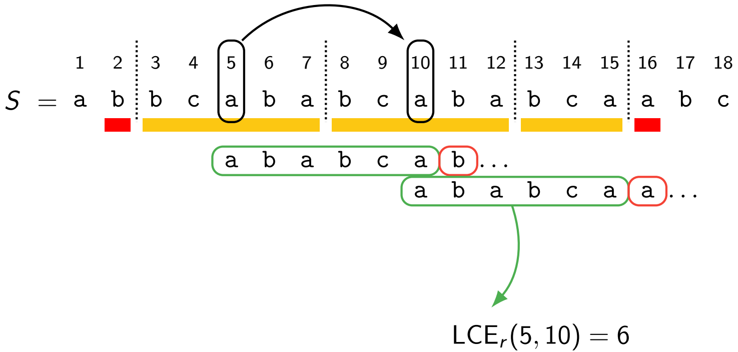
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length *p***.

Maximal Periodic Substrings



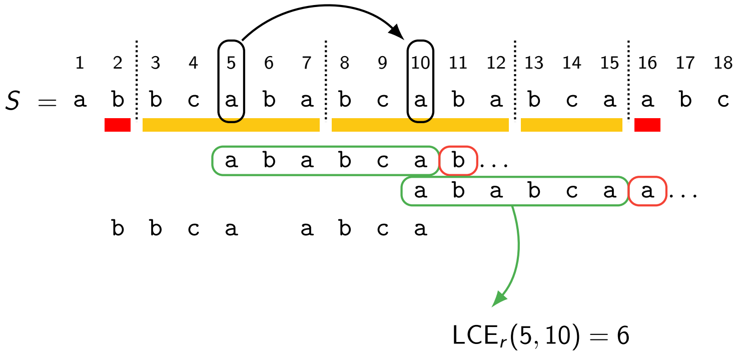
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length** $\boldsymbol{p}$.

Maximal Periodic Substrings



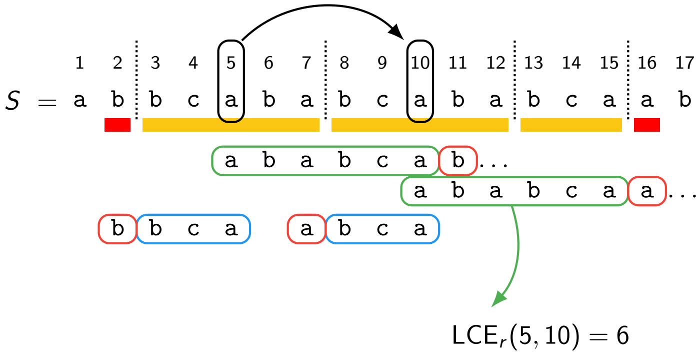
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



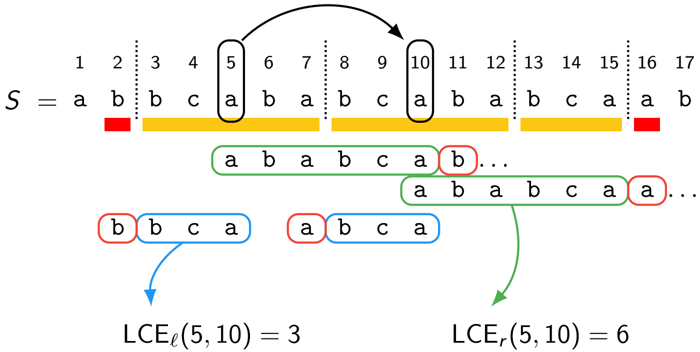
[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

Maximal Periodic Substrings



[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length** $\boldsymbol{p}$.

Maximal Periodic Substrings



[Bannai et al. 2017]: Every run with period p contains either a **next-smaller-suffix edge** or a **next-larger-suffix edge of length p**.

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

1: Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].

for $i \in \{1, \ldots, n\}$ **do**

    $j \leftarrow \text{nss}[i]$

    if $j < n + 1$ **then**

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

```
Precompute all next-smaller-suffix edges
in O(n) time [Bille et al. 2020].
for i ∈ {1, ..., n} do
    j ← nss[i]
    if j < n + 1 then
        p ← j − i
```

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

```
Precompute all next-smaller-suffix edges
in O(n) time [Bille et al. 2020].
for i ∈ {1, . . . , n} do
    j ← nss[i]
    if j < n + 1 then
        p ← j − i
        s ← i − LCE_ℓ(i, j) + 1
        e ← j + LCE_r(i, j) − 1
```

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].

for $i \in \{1, \ldots, n\}$ **do**
  $j \leftarrow \text{nss}[i]$
  if $j < n + 1$ **then**
    $p \leftarrow j - i$
    $s \leftarrow i - \text{LCE}_\ell(i, j) + 1$
    $e \leftarrow j + \text{LCE}_r(i, j) - 1$
    if $e - s + 1 \geq 2p$ **then**
      $S[s..e]$ is a run with period p.

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

1: Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].
2: **for** $i \in \{1, \dots, n\}$ **do**
3: $\quad j \leftarrow \text{nss}[i]$
4: $\quad$ **if** $j < n + 1$ **then**
5: $\quad\quad p \leftarrow j - i$
6: $\quad\quad s \leftarrow i - \text{LCE}_\ell(i, j) + 1$
7: $\quad\quad e \leftarrow j + \text{LCE}_r(i, j) - 1$
8: $\quad\quad$ **if** $e - s + 1 \geq 2p$ **then**
9: $\quad\quad\quad S[s..e]$ is a run with period p.
10: Repeat lines 2–9 with next-larger-suffix edges.

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

```
1:  Precompute all next-smaller-suffix edges
    in O(n) time [Bille et al. 2020].
2:  for i ∈ {1, ..., n} do
3:      j ← nss[i]
4:      if j < n + 1 then
5:          p ← j − i
6:          s ← i − LCE_ℓ(i, j) + 1
7:          e ← j + LCE_r(i, j) − 1
8:          if e − s + 1 ≥ 2p then
9:              S[s..e] is a run with period p.
10: Repeat lines 2–9 with next-larger-suffix edges.
```

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

1: Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].
2: **for** $i \in \{1, \ldots, n\}$ **do**
3:     $j \leftarrow \text{nss}[i]$
4:     **if** $j < n + 1$ **then**
5:         $p \leftarrow j - i$
6:         $s \leftarrow i - \text{LCE}_\ell(i, j) + 1$
7:         $e \leftarrow j + \text{LCE}_r(i, j) - 1$
8:         **if** $e - s + 1 \geq 2p$ **then**
9:             $S[s..e]$ is a run with period p.
10: Repeat lines 2–9 with next-larger-suffix edges.

$\mathcal{O}(n \lg^{2/3} n)$
[Kosolobov 2016]

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

1: Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].
2: **for** $i \in \{1, \ldots, n\}$ **do**
3: $\quad j \leftarrow \text{nss}[i]$
4: $\quad$ **if** $j < n + 1$ **then**
5: $\quad\quad p \leftarrow j - i$
6: $\quad\quad s \leftarrow i - \text{LCE}_\ell(i, j) + 1$
7: $\quad\quad e \leftarrow j + \text{LCE}_r(i, j) - 1$
8: $\quad\quad$ **if** $e - s + 1 \geq 2p$ **then**
9: $\quad\quad\quad S[s..e]$ is a run with period p.
10: Repeat lines 2–9 with next-larger-suffix edges.

$\mathcal{O}(n \lg^{2/3} n)$
[Kosolobov 2016]

$\mathcal{O}(n \lg \lg n)$
[Gawrychowski et al. 2016]

A Simple Runs Algorithm

Require: Text of length n over general ordered alphabet.
Ensure: All runs in the text.

1: Precompute all next-smaller-suffix edges in $\mathcal{O}(n)$ time [Bille et al. 2020].
2: **for** $i \in \{1, \ldots, n\}$ **do**
3: $\quad j \leftarrow \text{nss}[i]$
4: $\quad$ **if** $j < n + 1$ **then**
5: $\quad\quad p \leftarrow j - i$
6: $\quad\quad s \leftarrow i - \text{LCE}_\ell(i, j) + 1$
7: $\quad\quad e \leftarrow j + \text{LCE}_r(i, j) - 1$
8: $\quad\quad$ **if** $e - s + 1 \geq 2p$ **then**
9: $\quad\quad\quad S[s..e]$ is a run with period p.
10: Repeat lines 2–9 with next-larger-suffix edges.

$\mathcal{O}(n \lg^{2/3} n)$ [Kosolobov 2016]

$\mathcal{O}(n \lg \lg n)$ [Gawrychowski et al. 2016]

$\mathcal{O}(n\alpha(n))$ [Crochemore et al. 2016]

- A Note on Alphabet Types
- Reduction of Runs to Next Smaller Suffixes and LCEs
- **Linear Time Next Smaller Suffixes**
- Linear Time LCEs
- Practical Aspects & Conclusion

Computing Next Smaller Suffixes

Computing Next Smaller Suffixes Values

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

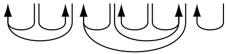
```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10



Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

for $i = 1$ **to** n **do**
  $j \leftarrow i - 1$
  while $j > 0 \wedge A[j] > A[i]$ **do**
    $\text{nsv}[j] \leftarrow i$
    $j \leftarrow \text{psv}[j]$
  $\text{psv}[i] \leftarrow j$

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

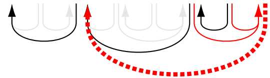
1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

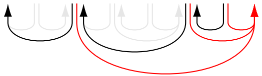


Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

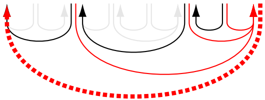


Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
for i = 1 to n do
    j ← i − 1
    while j > 0 ∧ A[j] > A[i] do
        nsv[j] ← i
        j ← psv[j]
    psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

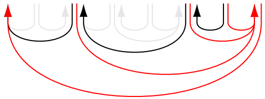


Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

for $i = 1$ **to** n **do**
  $j \leftarrow i - 1$
  while $j > 0 \wedge A[j] > A[i]$ **do**
    $\mathrm{nsv}[j] \leftarrow i$
    $j \leftarrow \mathrm{psv}[j]$
  $\mathrm{psv}[i] \leftarrow j$

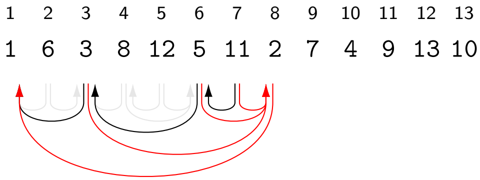
1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10



Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ A[j] > A[i] do
4:         nsv[j] ← i
5:         j ← psv[j]
6:     psv[i] ← j
```



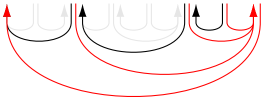
- after every iteration of line 3, we assign either nsv$[j]$ or psv$[i]$

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ A[j] > A[i] do
4:         nsv[j] ← i
5:         j ← psv[j]
6:     psv[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10



- after every iteration of line 3, we assign either nsv$[j]$ or psv$[i]$
- thus at most $2n$ element comparisons

Computing Next Smaller ~~Suffixes~~ Values

Require: Array $A[1..n]$
Ensure: Arrays nsv and psv

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ A[j] > A[i] do
4:         nsv[j] ← i
5:         j ← psv[j]
6:     psv[i] ← j
```

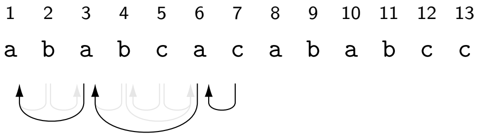
1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	8	12	5	11	2	7	4	9	13	10

- after every iteration of line 3, we assign either nsv[j] or psv[i]
- thus at most $2n$ element comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

1: **for** $i = 1$ **to** n **do**
2: $\quad j \leftarrow i - 1$
3: $\quad$ **while** $j > 0 \land S_j \succ S_i$ **do**
4: $\quad\quad \text{nss}[j] \leftarrow i$
5: $\quad\quad j \leftarrow \text{pss}[j]$
6: $\quad \text{pss}[i] \leftarrow j$



- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	b	c	a	c	a	b	a	b	c	c

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	b	c	a	c	a	b	a	b	c	c

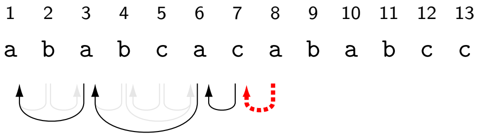
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} =$$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



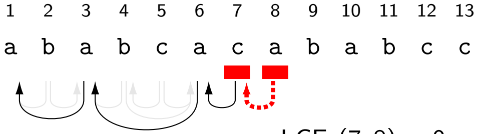
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} =$$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

1: **for** $i = 1$ **to** n **do**
2: $\quad j \leftarrow i - 1$
3: $\quad$ **while** $j > 0 \wedge S_j \succ S_i$ **do**
4: $\quad\quad \text{nss}[j] \leftarrow i$
5: $\quad\quad j \leftarrow \text{pss}[j]$
6: $\quad \text{pss}[i] \leftarrow j$



$\text{LCE}_r(7, 8) = 0$

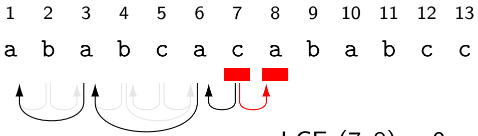
$\# \binom{\text{comparisons}}{\text{for } i=8} =$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
    - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



$\mathrm{LCE}_r(7, 8) = 0$

$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 +$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	b	c	a	c	a	b	a	b	c	c

$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 +$$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	b	c	a	c	a	b	a	b	c	c

$\mathrm{LCE}_r(6, 8) = 1$

$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 +$

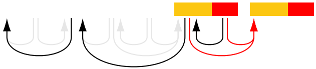
- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

1: **for** $i = 1$ **to** n **do**
2: $j \leftarrow i - 1$
3: **while** $j > 0 \wedge S_j \succ S_i$ **do**
4: $\mathsf{nss}[j] \leftarrow i$
5: $j \leftarrow \mathsf{pss}[j]$
6: $\mathsf{pss}[i] \leftarrow j$

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	b	c	a	c	a	b	a	b	c	c



$\mathrm{LCE}_r(6, 8) = 1$

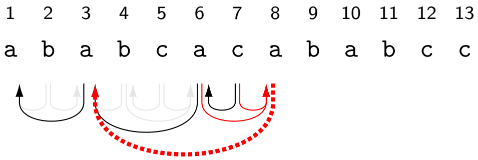
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 +$$

- after every iteration of line 3, we assign either $\mathsf{nss}[j]$ or $\mathsf{pss}[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



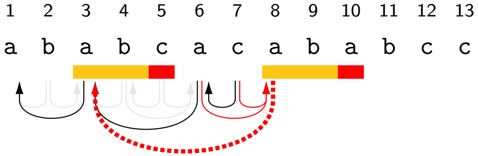
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 +$$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



$$\mathsf{LCE}_r(3,8) = 2$$

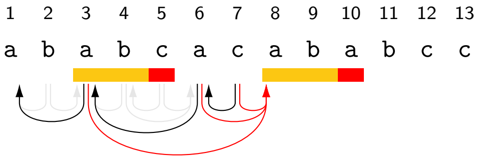
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 +$$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



$\mathrm{LCE}_r(3, 8) = 2$

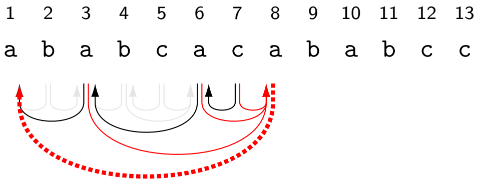
$\#\binom{\text{comparisons}}{\text{for } i=8} = 1 + 2 + 3 +$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



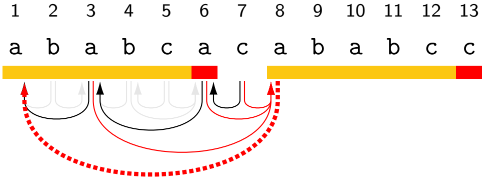
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 + 3 +$$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



$$\mathrm{LCE}_r(1, 8) = 5$$

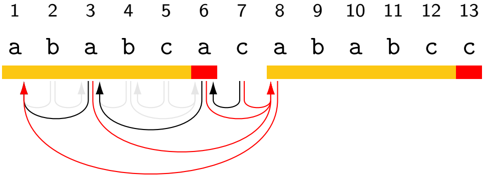
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 + 3 +$$

- after every iteration of line 3, we assign either nss$[j]$ or pss$[i]$
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```



$$\mathrm{LCE}_r(1,8) = 5$$

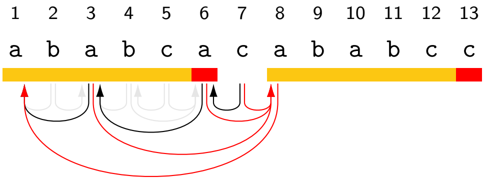
$$\#\binom{\text{comparisons}}{\text{for } i=8} = 1 + 2 + 3 + 6 = 12$$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
  - but possibly many more symbol comparisons

Computing Next Smaller Suffixes

Require: String $S[1..n]$
Ensure: Arrays nss and pss

```
1: for i = 1 to n do
2:     j ← i − 1
3:     while j > 0 ∧ S_j ≻ S_i do
4:         nss[j] ← i
5:         j ← pss[j]
6:     pss[i] ← j
```

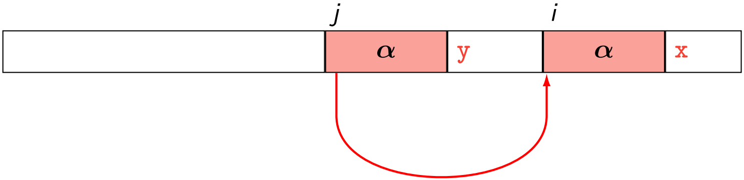


$$\mathrm{LCE}_r(1,8) = 5$$

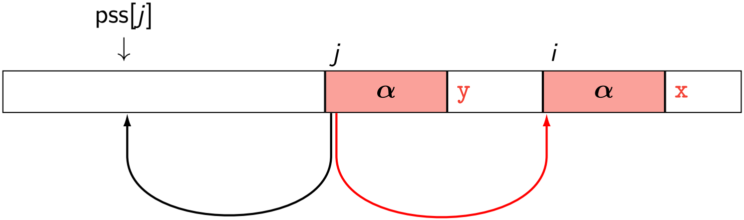
$$\#\binom{\text{comparisons}}{\text{for } i{=}8} = 1 + 2 + 3 + 6 = 12$$

- after every iteration of line 3, we assign either nss[j] or pss[i]
- thus at most $2n$ suffix comparisons
    - but possibly many more symbol comparisons
    - some strings require $\Omega(n^2)$ symbol comparisons

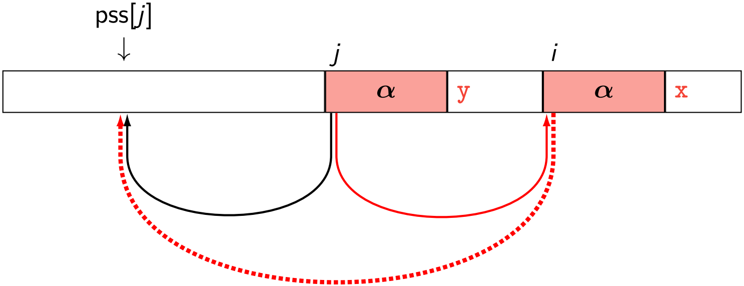
Skipping Symbol Comparisons (fixed *i*)



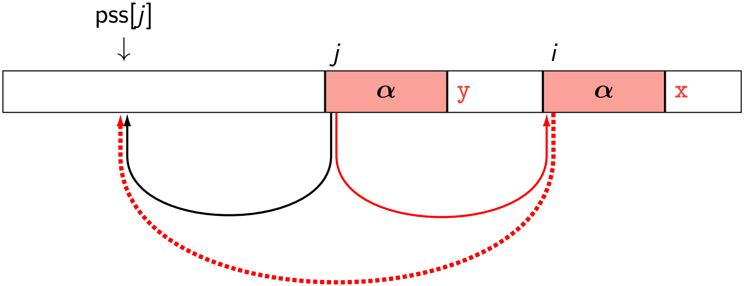
Skipping Symbol Comparisons (fixed *i*)



Skipping Symbol Comparisons (fixed *i*)

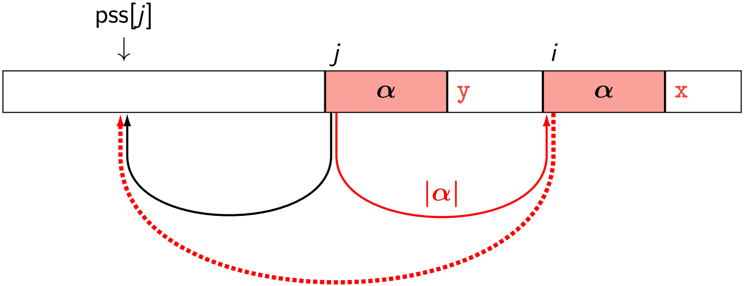


Skipping Symbol Comparisons (fixed i)



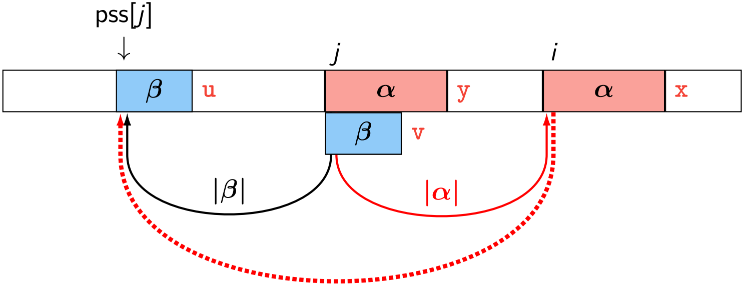
- store the computed LCEs together with edges

Skipping Symbol Comparisons (fixed *i*)



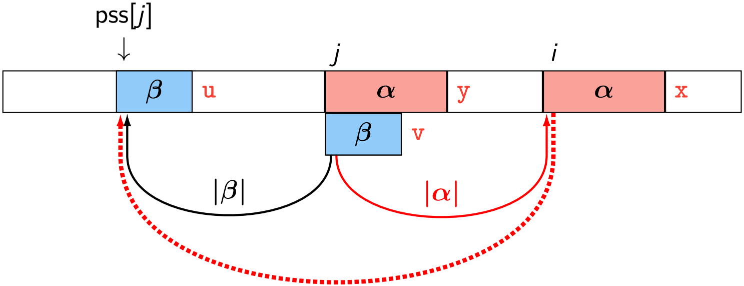
- store the computed LCEs together with edges

Skipping Symbol Comparisons (fixed *i*)



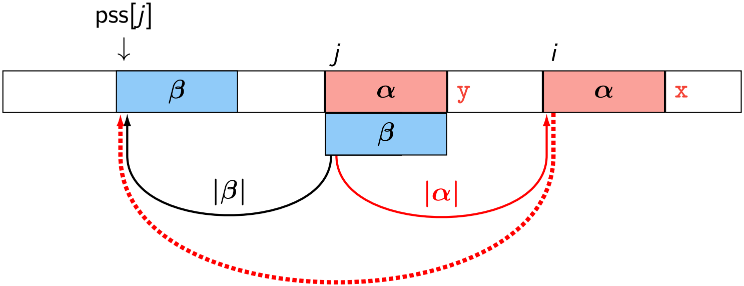
- store the computed LCEs together with edges

Skipping Symbol Comparisons (fixed *i*)



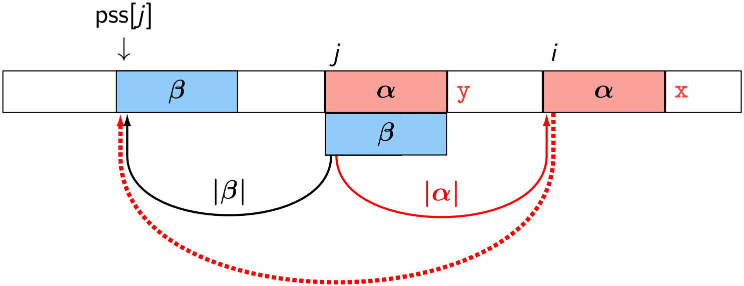
- store the computed LCEs together with edges
  - if $|\beta| < |\alpha|$, then $\mathrm{LCE}_r(pss[j], i) = |\beta|$ and $\mathrm{pss}[i] = \mathrm{pss}[j]$

Skipping Symbol Comparisons (fixed *i*)



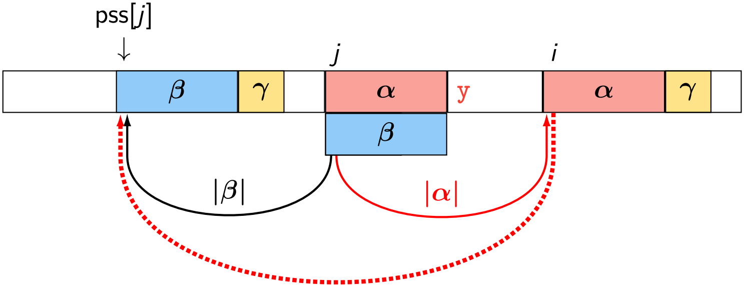
- store the computed LCEs together with edges
  - if $|\beta| < |\alpha|$, then $\text{LCE}_r(pss[j], i) = |\beta|$ and $\text{pss}[i] = \text{pss}[j]$

Skipping Symbol Comparisons (fixed i)



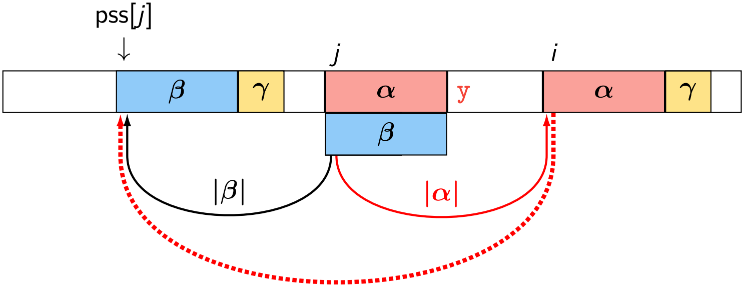
- store the computed LCEs together with edges
  - if $|\beta| < |\alpha|$, then $\mathrm{LCE}_r(pss[j], i) = |\beta|$ and $\mathrm{pss}[i] = \mathrm{pss}[j]$
  - if $|\beta| \geq |\alpha|$, then $\mathrm{LCE}_r(pss[j], i) \geq |\alpha|$

Skipping Symbol Comparisons (fixed *i*)



- store the computed LCEs together with edges
  - if $|\beta| < |\alpha|$, then $\text{LCE}_r(pss[j], i) = |\beta|$ and $\text{pss}[i] = \text{pss}[j]$
  - if $|\beta| \geq |\alpha|$, then $\text{LCE}_r(pss[j], i) \geq |\alpha|$

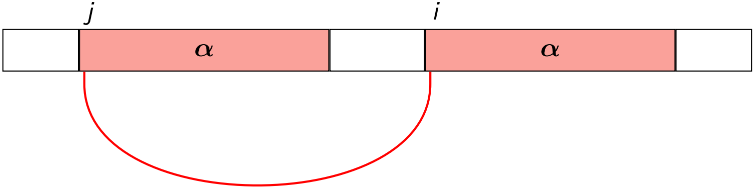
Skipping Symbol Comparisons (fixed *i*)



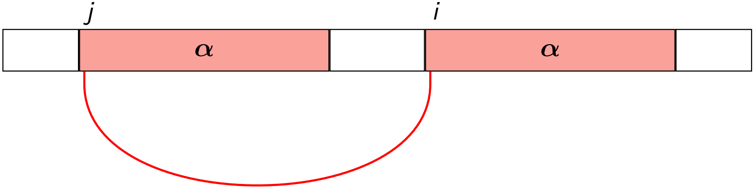
- store the computed LCEs together with edges
  - if $|\beta| < |\alpha|$, then $\mathrm{LCE}_r(pss[j], i) = |\beta|$ and $\mathrm{pss}[i] = \mathrm{pss}[j]$
  - if $|\beta| \geq |\alpha|$, then $\mathrm{LCE}_r(pss[j], i) \geq |\alpha|$
- still $\Omega(n^2)$ comparisons in the worst case

Skipping Symbol Comparisons (generally)

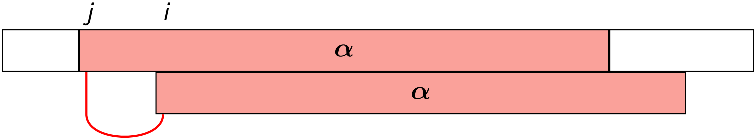
Skipping Symbol Comparisons (generally)



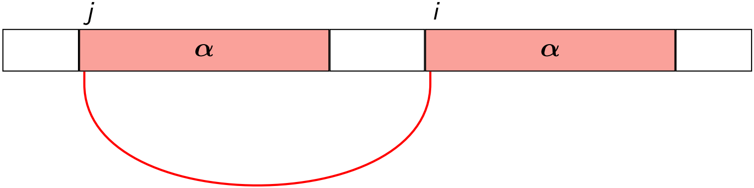
Skipping Symbol Comparisons (generally)



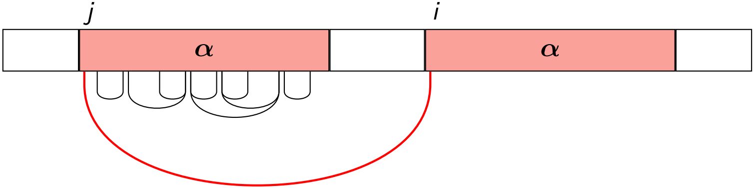
- possible, but not a problem:



Skipping Symbol Comparisons (generally)

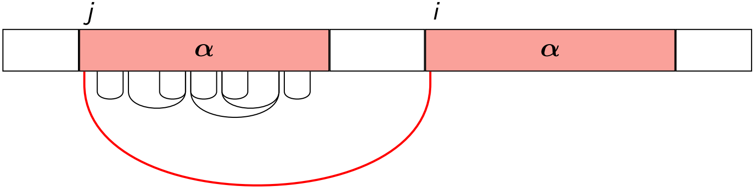


Skipping Symbol Comparisons (generally)



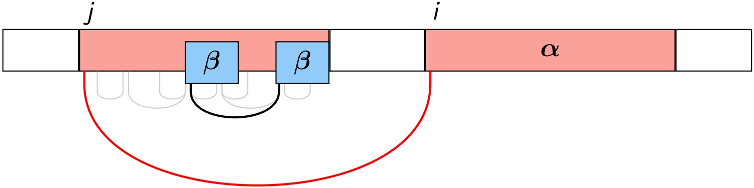
- consider edges in left occurrence of α in computational order

Skipping Symbol Comparisons (generally)



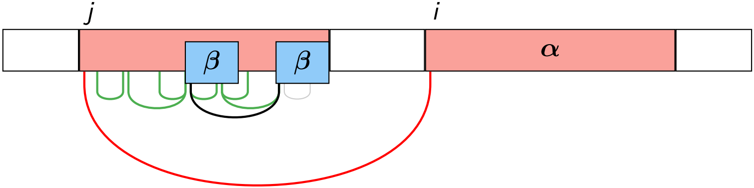
- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE

Skipping Symbol Comparisons (generally)



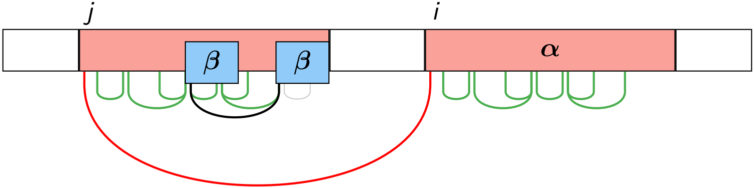
- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE

Skipping Symbol Comparisons (generally)



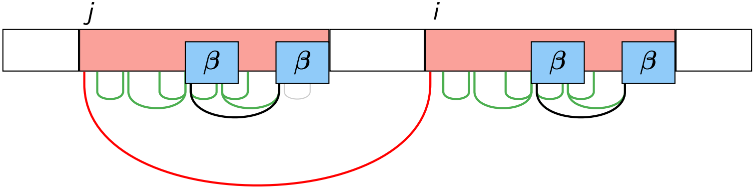
- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE
- transfer all previous edges to right occurrence of α

Skipping Symbol Comparisons (generally)



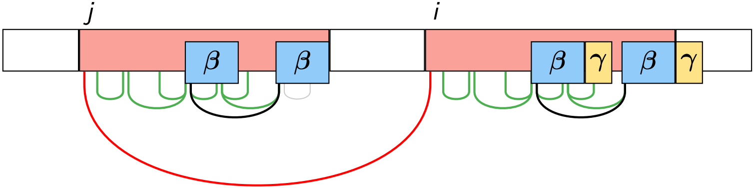
- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE
- transfer all previous edges to right occurrence of α

Skipping Symbol Comparisons (generally)



- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE
- transfer all previous edges to right occurrence of α

Skipping Symbol Comparisons (generally)



- consider edges in left occurrence of α in computational order
- find first edge with "long" LCE
- transfer all previous edges to right occurrence of α

Computing the LCEs in the Left Direction

Computing the LCEs in the Left Direction

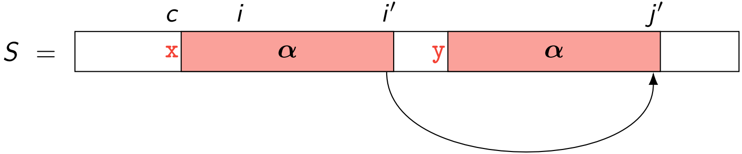
- consider the NSS edges in decreasing order of start positions

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

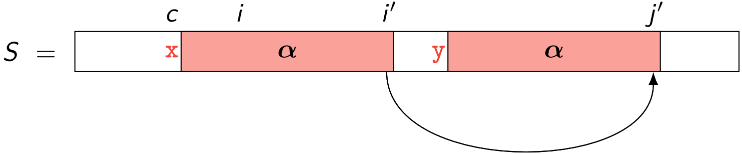
Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

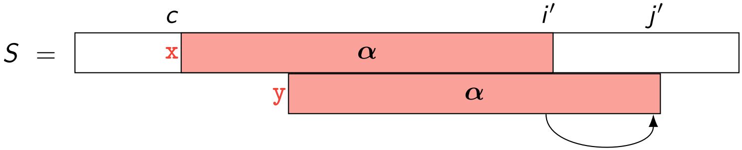


Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

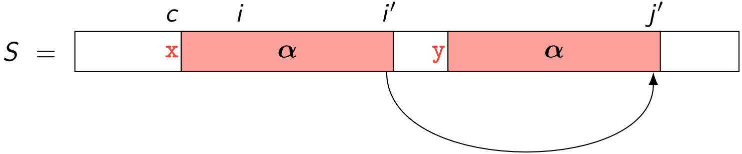


- in the paper:



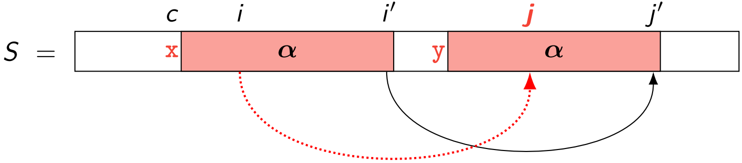
Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



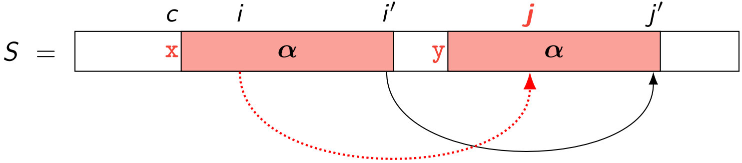
Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



Computing the LCEs in the Left Direction

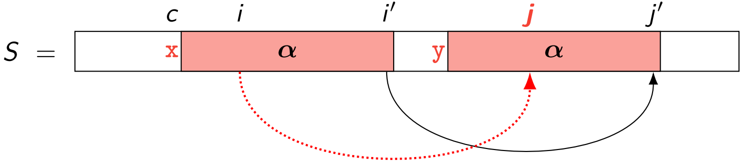
- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



$S_j \prec S_i$

Computing the LCEs in the Left Direction

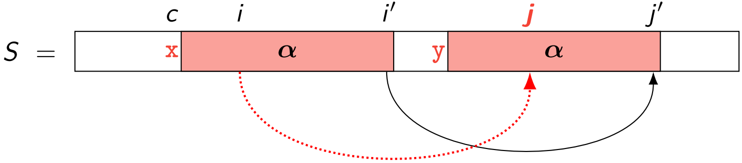
- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



$$S_j \prec S_i \prec S_{i'}$$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



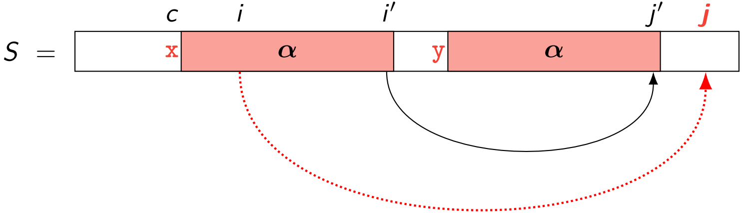
$$S_j \prec S_i \prec S_{i'} \implies \mathsf{nss}[i'] \leq j$$ ↯

Property 1:

$\mathsf{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

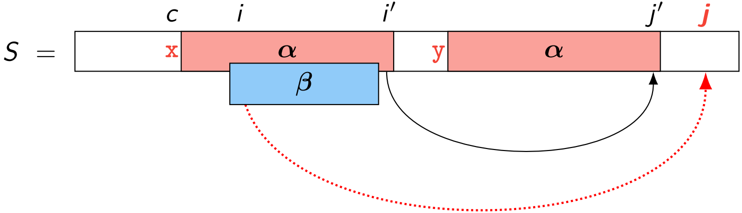


Property 1:

$\mathsf{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

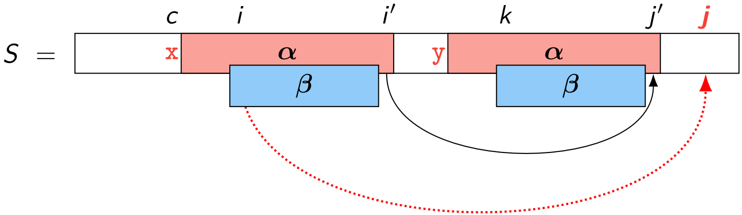


Property 1:

$\mathsf{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

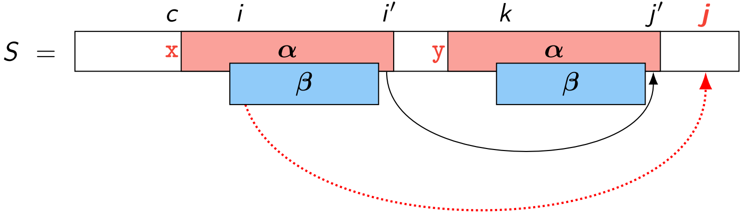


Property 1:

$\mathsf{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



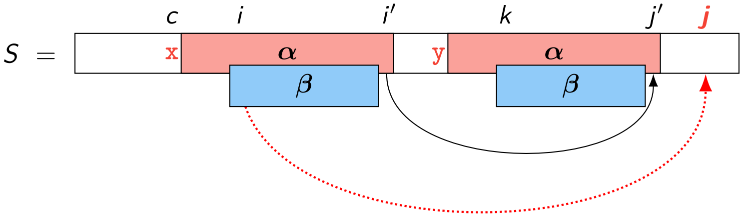
$S_{i'} \succ S_{j'}$

Property 1:

$\mathsf{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



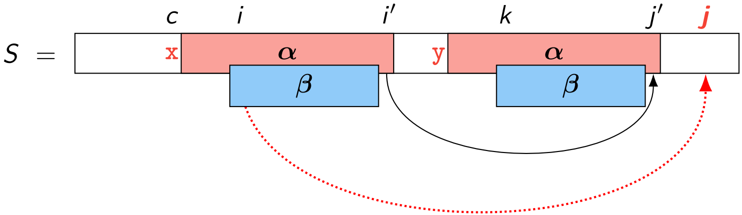
$$S_{i'} \succ S_{j'}$$
$$\iff \beta S_{i'} \succ \beta S_{j'}$$

Property 1:

$\mathrm{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



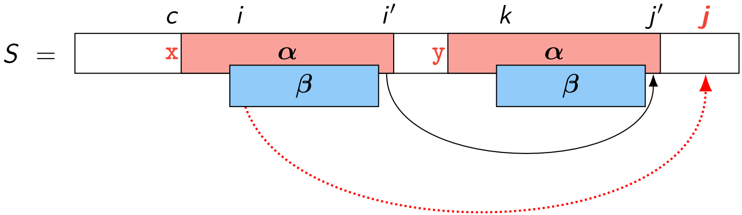
$$S_{i'} \succ S_{j'}$$
$$\iff \beta S_{i'} \succ \beta S_{j'}$$
$$\iff S_i \succ S_k$$

Property 1:

$\mathrm{nss}[i] \notin (i', j')$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



$$
\begin{aligned}
& S_{i'} \succ S_{j'} \\
\iff & \beta S_{i'} \succ \beta S_{j'} \\
\iff & S_i \succ S_k \\
\implies & \mathrm{nss}[i] \leq k \ \lightning
\end{aligned}
$$

Property 1:

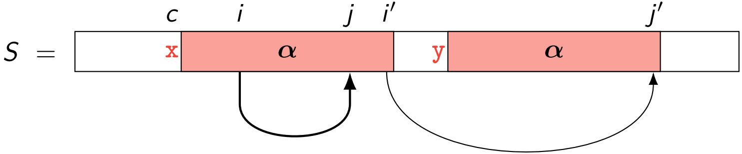
$\mathrm{nss}[i] \notin (i', j')$

Property 2:

$\mathrm{nss}[i] \notin [j', n]$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

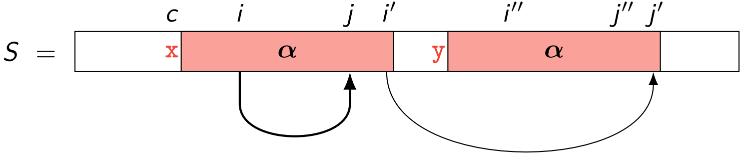


Property 1:
$\mathsf{nss}[i] \notin (i', j')$

Property 2:
$\mathsf{nss}[i] \notin [j', n]$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c

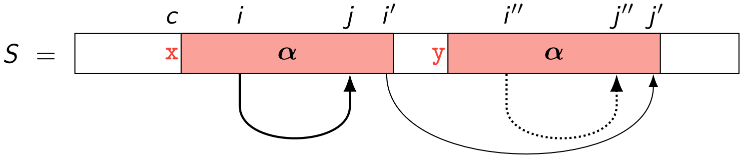


Property 1:
$\mathsf{nss}[i] \notin (i', j')$

Property 2:
$\mathsf{nss}[i] \notin [j', n]$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



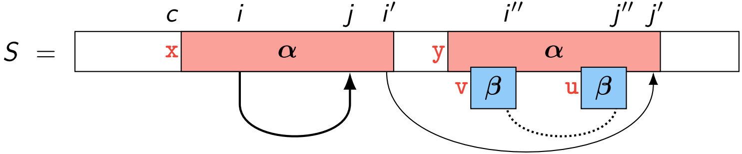
Property 1:
$\mathrm{nss}[i] \notin (i', j')$

Property 2:
$\mathrm{nss}[i] \notin [j', n]$

Property 3:
$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



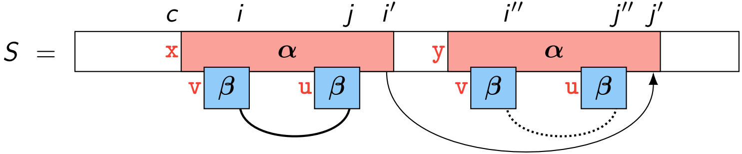
Property 1:
$\mathsf{nss}[i] \notin (i', j')$

Property 2:
$\mathsf{nss}[i] \notin [j', n]$

Property 3:
$\mathsf{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



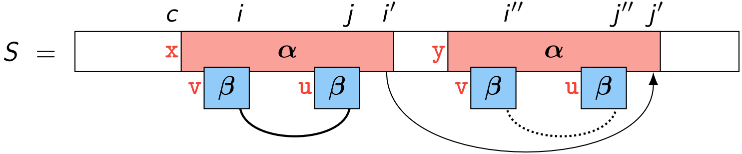
Property 1:
$\mathsf{nss}[i] \notin (i', j')$

Property 2:
$\mathsf{nss}[i] \notin [j', n]$

Property 3:
$\mathsf{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



- "short" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = \mathrm{LCE}_\ell(i'',j'')$

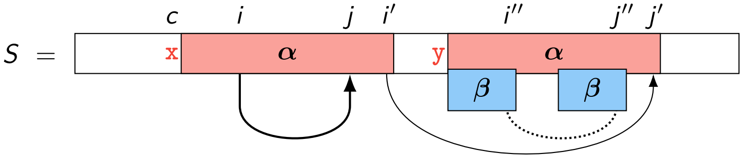
Property 1:
$\mathrm{nss}[i] \notin (i', j')$

Property 2:
$\mathrm{nss}[i] \notin [j', n]$

Property 3:
$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



- "short" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = \mathrm{LCE}_\ell(i'',j'')$

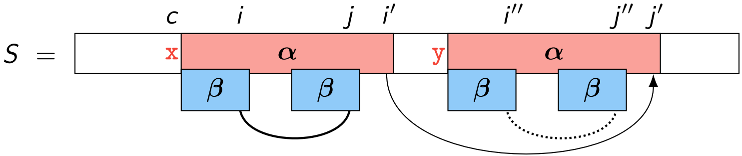
Property 1:
$\mathrm{nss}[i] \notin (i',j')$

Property 2:
$\mathrm{nss}[i] \notin [j',n]$

Property 3:
$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



- "short" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = \mathrm{LCE}_\ell(i'',j'')$

Property 1:

$\mathrm{nss}[i] \notin (i', j')$

Property 2:

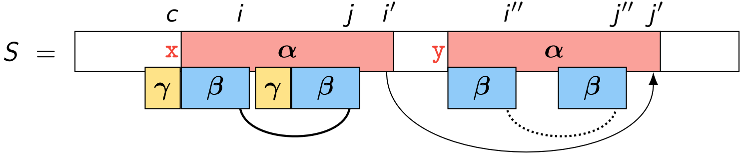
$\mathrm{nss}[i] \notin [j', n]$

Property 3:

$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



- "short" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = \mathrm{LCE}_\ell(i'',j'')$

Property 1:

$\mathrm{nss}[i] \notin (i', j')$

Property 2:

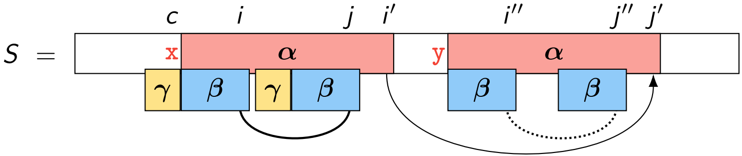
$\mathrm{nss}[i] \notin [j', n]$

Property 3:

$$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$$

Computing the LCEs in the Left Direction

- consider the NSS edges in decreasing order of start positions
- keep track of the leftmost inspected symbol at position c



- "short" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = \mathrm{LCE}_\ell(i'',j'')$
- "long" LCE between i'' and j'' $\implies$ $\mathrm{LCE}_\ell(i,j) = i - c + |\gamma|$

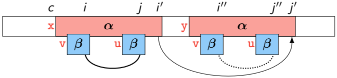
Property 1:
$\mathrm{nss}[i] \notin (i', j')$

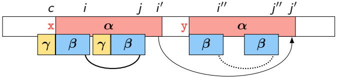
Property 2:
$\mathrm{nss}[i] \notin [j', n]$

Property 3:
$\mathrm{nss}[\underbrace{i - i' + j'}_{=i''}] = \underbrace{j - i' + j'}_{=j''}$

Computing the LCEs in the Left Direction

Require: Array nss$[1..n]$
Ensure: $\forall i$: LCE$_\ell(i, \text{nss}[i])$

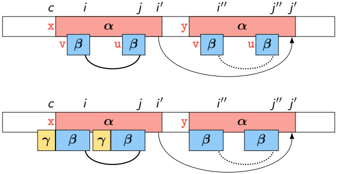




Computing the LCEs in the Left Direction

Require: Array nss[1..n]
Ensure: $\forall i$: $\mathrm{LCE}_\ell(i, \mathrm{nss}[i])$

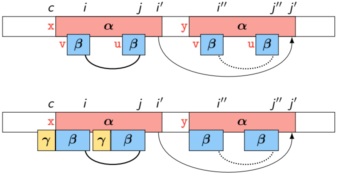
$c \leftarrow n$; $d \leftarrow \perp$.



Computing the LCEs in the Left Direction

Require: Array nss$[1..n]$
Ensure: $\forall i : \mathsf{LCE}_\ell(i, \mathsf{nss}[i])$

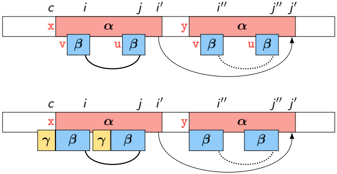
$c \leftarrow n$; $d \leftarrow \bot$.
for $i = n$ **down to** 1 **do**
  $j \leftarrow \mathsf{nss}[i]$
  if $j < n + 1$ **then**



Computing the LCEs in the Left Direction

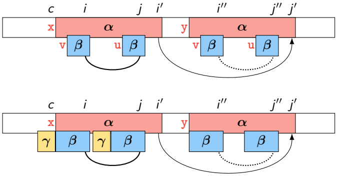
Require: Array nss$[1..n]$

Ensure: $\forall i$: LCE$_\ell(i, \text{nss}[i])$



$c \leftarrow n$; $d \leftarrow \bot$.
for $i = n$ **down to** 1 **do**
    $j \leftarrow \text{nss}[i]$
    if $j < n + 1$ **then**
        $i'' \leftarrow i + d$
        $j'' \leftarrow j + d$

Computing the LCEs in the Left Direction



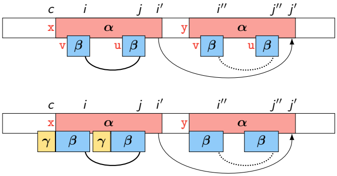
Require: Array nss$[1..n]$
Ensure: $\forall i$: $\mathrm{LCE}_\ell(i, \mathrm{nss}[i])$

```
c ← n; d ← ⊥.
for i = n down to 1 do
    j ← nss[i]
    if j < n + 1 then
        i'' ← i + d
        j'' ← j + d
        if LCE_ℓ(i'', j'') < (i − c) then
            LCE_ℓ(i, j) ← LCE_ℓ(i'', j'')
```

Computing the LCEs in the Left Direction

Require: Array nss$[1..n]$
Ensure: $\forall i : \text{LCE}_\ell(i, \text{nss}[i])$

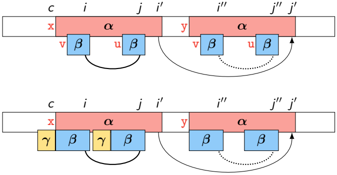
```
c ← n; d ← ⊥.
for i = n down to 1 do
    j ← nss[i]
    if j < n + 1 then
        i'' ← i + d
        j'' ← j + d
        if LCE_ℓ(i'', j'') < (i − c) then
            LCE_ℓ(i, j) ← LCE_ℓ(i'', j'')
```



Computing the LCEs in the Left Direction

Require: Array $\mathsf{nss}[1..n]$
Ensure: $\forall i : \mathrm{LCE}_\ell(i, \mathsf{nss}[i])$

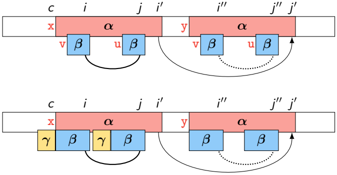
$c \leftarrow n$; $d \leftarrow \bot$.
for $i = n$ **down to** 1 **do**
  $j \leftarrow \mathsf{nss}[i]$
  if $j < n + 1$ **then**
    $i'' \leftarrow i + d$
    $j'' \leftarrow j + d$
    if $\mathrm{LCE}_\ell(i'', j'') < (i - c)$ **then**
      $\mathrm{LCE}_\ell(i, j) \leftarrow \mathrm{LCE}_\ell(i'', j'')$
    else
      $\mathrm{LCE}_\ell(i, j) \leftarrow (i - c) + \text{SCAN-LCE}_\ell(c, j - (i - c))$



Computing the LCEs in the Left Direction

Require: Array $\mathrm{nss}[1..n]$
Ensure: $\forall i : \mathrm{LCE}_\ell(i, \mathrm{nss}[i])$

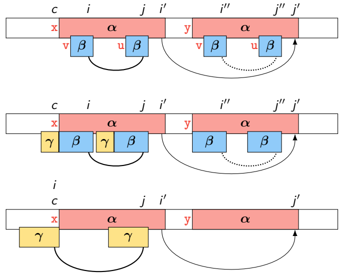
$c \leftarrow n$; $d \leftarrow \bot$.
for $i = n$ **down to** 1 **do**
  $j \leftarrow \mathrm{nss}[i]$
  if $j < n + 1$ **then**
    $i'' \leftarrow i + d$
    $j'' \leftarrow j + d$
    if $\mathrm{LCE}_\ell(i'', j'') < (i - c)$ **then**
      $\mathrm{LCE}_\ell(i, j) \leftarrow \mathrm{LCE}_\ell(i'', j'')$
    else
      $\mathrm{LCE}_\ell(i, j) \leftarrow (i - c) + \text{SCAN-LCE}_\ell(c, j - (i - c))$



Computing the LCEs in the Left Direction

Require: Array nss[1..n]
Ensure: $\forall i$: $\mathrm{LCE}_\ell(i, \mathrm{nss}[i])$

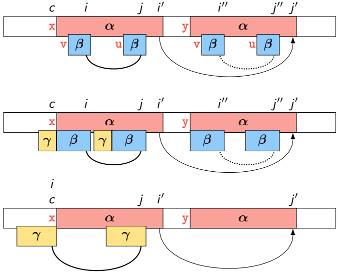
```
c ← n; d ← ⊥.
for i = n down to 1 do
    j ← nss[i]
    if j < n + 1 then
        i'' ← i + d
        j'' ← j + d
        if LCE_ℓ(i'', j'') < (i − c) then
            LCE_ℓ(i, j) ← LCE_ℓ(i'', j'')
        else
            LCE_ℓ(i, j) ← (i − c) + SCAN-LCE_ℓ(c, j − (i − c))
```



Computing the LCEs in the Left Direction

Require: Array $\mathsf{nss}[1..n]$
Ensure: $\forall i : \mathsf{LCE}_\ell(i, \mathsf{nss}[i])$

```
c ← n; d ← ⊥.
for i = n down to 1 do
    j ← nss[i]
    if j < n + 1 then
        i'' ← i + d
        j'' ← j + d
        if LCE_ℓ(i'', j'') < (i − c) then
            LCE_ℓ(i, j) ← LCE_ℓ(i'', j'')
        else
            LCE_ℓ(i, j) ← (i − c) + SCAN-LCE_ℓ(c, j − (i − c))
            c ← i − LCE_ℓ(i, j)   // = c − LCE_ℓ(c, j − (i − c))
            d ← j − i
```



Preliminary Experiments

- AMD EPYC 7452 CPU
- repeat each experiment five times, use median as result
- Texts from `http://pizzachili.dcc.uchile.cl/`

Text n in MiB	**sources** 201 MiB	**pitches** 53 MiB	**proteins** 1024 MiB	**dna** 385 MiB	**english** 1024 MiB	**xml** 282 MiB	**fib41** 255 MiB	**tm29** 256 MiB	**rs.13** 206 MiB
runs/100n									
MiB/s									

Preliminary Experiments

- AMD EPYC 7452 CPU
- repeat each experiment five times, use median as result
- Texts from `http://pizzachili.dcc.uchile.cl/`

Text n in MiB	**sources** 201 MiB	**pitches** 53 MiB	**proteins** 1024 MiB	**dna** 385 MiB	**english** 1024 MiB	**xml** 282 MiB	**fib41** 255 MiB	**tm29** 256 MiB	**rs.13** 206 MiB
runs/100n	4.7	11.7	7.0	25.3	2.4	3.4			
MiB/s	11.4	11.0	10.9	8.8	10.5	12.8			

Preliminary Experiments

- AMD EPYC 7452 CPU
- repeat each experiment five times, use median as result
- Texts from `http://pizzachili.dcc.uchile.cl/`

Text n in MiB	**sources** 201 MiB	**pitches** 53 MiB	**proteins** 1024 MiB	**dna** 385 MiB	**english** 1024 MiB	**xml** 282 MiB	**fib41** 255 MiB	**tm29** 256 MiB	**rs.13** 206 MiB
runs/100n	4.7	11.7	7.0	25.3	2.4	3.4	76.3	83.3	92.7
MiB/s	11.4	11.0	10.9	8.8	10.5	12.8	15.4	15.6	15.1

Conclusion

Conclusion

- simple and practically fast algorithm (over 10MiB of text per second)

Conclusion

- simple and practically fast algorithm (over 10MiB of text per second)

- single header file C++ implementation:

  jonas-ellert/linear-time-runs

Conclusion

- simple and practically fast algorithm (over 10MiB of text per second)
- single header file C++ implementation:

  jonas-ellert/linear-time-runs
- first linear time algorithm for runs over general ordered alphabets

Conclusion

- simple and practically fast algorithm (over 10MiB of text per second)
- single header file C++ implementation:

  jonas-ellert/linear-time-runs

- first linear time algorithm for runs over general ordered alphabets
- exploits properties of the next-smaller/larger-suffix edges
  (also known as Lyndon array)

Conclusion

- simple and practically fast algorithm (over 10MiB of text per second)
- single header file C++ implementation:

  jonas-ellert/linear-time-runs
- first linear time algorithm for runs over general ordered alphabets
- exploits properties of the next-smaller/larger-suffix edges
  (also known as Lyndon array)

Thanks for your attention! Questions?