Edit and Alphabet-Ordering Sensitivity of Lex-Parse

Yuto Nakashima
Kyushu University

Dominik Köppl
University of Yamanashi

Mitsuru Funakoshi
NTT Comm. Sci. Lab.

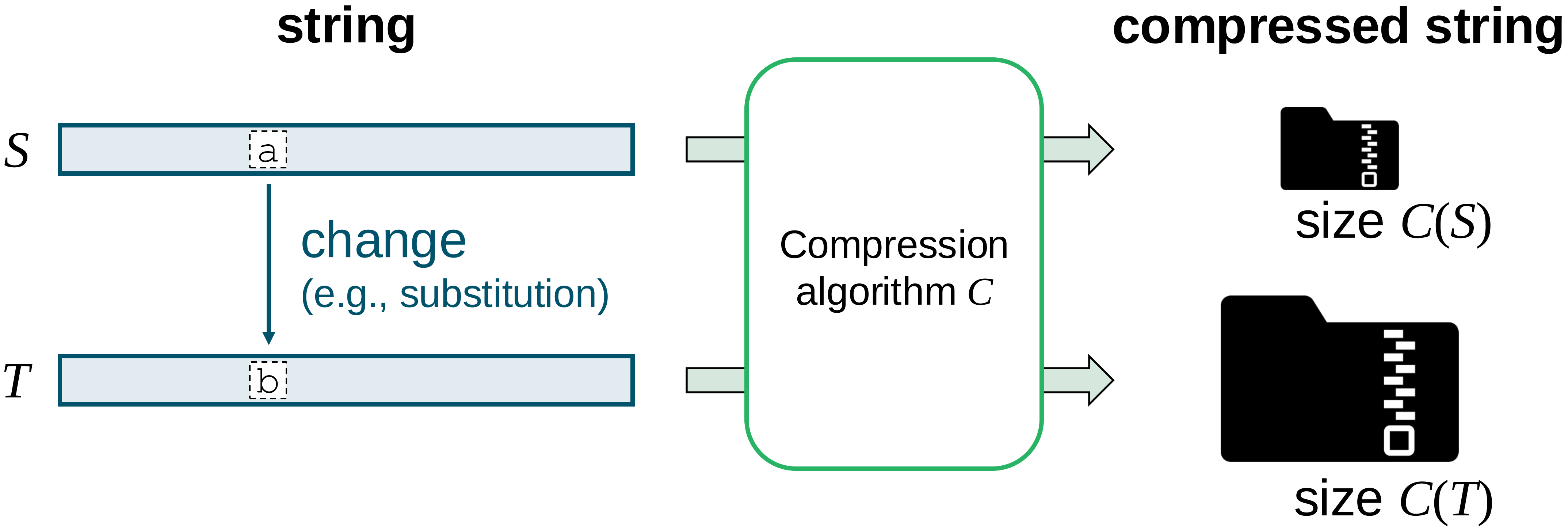
Shunsuke Inenaga
Kyushu University

Hideo Bannai
TMDU

2024/8/30

Sensitivity of compressors [Akagi et al., 2023]

The sensitivity of a string compression algorithm/scheme C is defined as the maximum difference between the size $C(S)$ for a string S and the size $C(T)$ for a single-character edited string T.



How much can the sizes differ in the worst case?

Sensitivity of compressors [Akagi et al., 2023]

The sensitivity of a string compression algorithm/scheme C is defined as the maximum difference between the size $C(S)$ for a string S and the size $C(T)$ for a single-character edited string T.

There are three edit operations.

- substitution `abac` → `acac`
- insertion `ab-ac` → `abcac`
- deletion `abac` → `aba-`

In this work,
we study the multiplicative sensitivity.

Multiplicative sensitivity

$$\max_{S\in\Sigma^n}\left\{\frac{\boldsymbol{C(T)}}{\boldsymbol{C(S)}}\,\middle|\,\mathrm{ED}(S,T)=1\right\}$$

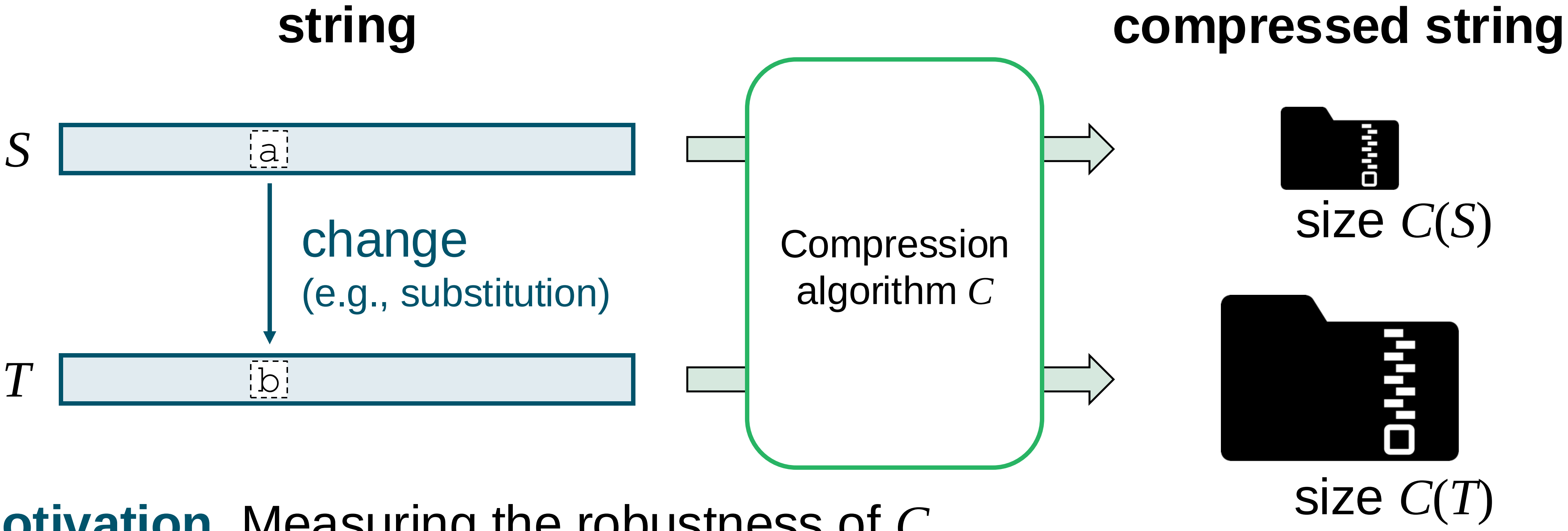
edit distance

Additive sensitivity

$$\max_{S\in\Sigma^n}\{\boldsymbol{C(T)}-\boldsymbol{C(S)}\mid \mathrm{ED}(S,T)=1\}$$

Sensitivity of compressors [Akagi et al., 2023]

The sensitivity of a string compression algorithm/scheme C is defined as the maximum difference between the size $C(S)$ for a string S and the size $C(T)$ for a single-character edited string T.



Motivation. Measuring the robustness of C for errors and/or dynamic changes occurring in the input string.

Sensitivity of compressors

Multiplicative sensitivity

Measures	Edit operations	Upper bounds	Lower bounds
Smallest bidirectional scheme (b)	all	2	2
Smallest grammar (g^*)	all	2	–
RePair (g_{rpair})	all	$O((n / \log n)^{2/3})$	–
Run-length BWT (r)	all	$O(\log r \log n)$	$\Omega(\log n)$ [Giuliani et al., 2023]
LZ77 (z_{77})	all	2	2
LZSS (z_{SS})	ins.	2	2
	sub./del.	3	3
LZ78 (z_{78})	all	–	$\Omega(n^{1/4})$ [Lagarde & Perifel, 2018]

See the following paper for more results.

“Sensitivity of string compressors and repetitiveness measures”
T. Akagi, M. Funakoshi, S. Inenaga - Information & Computation, 2023

Sensitivity of compressors

Multiplicative sensitivity

Measures	Edit operations	Upper bounds	Lower bounds
Smallest bidirectional scheme (b)	all	2	2
Smallest grammar (g^*)	all	2	–
RePair (g_{rpair})	all	$O((n / \log n)^{2/3})$	–
Run-length BWT (r)	all	$O(\log r \log n)$	$\Omega(\log n)$ [Giuliani et al., 2023]
LZ77 (z_{77})	all	2	2
LZSS (z_{SS})	ins.	2	2
	sub./del.	3	3
LZ78 (z_{78})	all	–	$\Omega(n^{1/4})$ [Lagarde & Perifel, 2018]
Lex-parse (v)	Open		

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \ldots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

1	2	3	4	5	6	7	8	9
a	a	b	a	b	a	a	b	a

The lex-parse of a string is a string parsing which is defined by the lex. ordering of suffixes of the input string.

We can explain by SA and LCP array.

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	a a b a
1	4	a a b a b a a b a
7	1	a b a
4	3	a b a a b a
2	3	a b a b a a b a
8	0	b a
5	2	b a a b a
3	2	b a b a a b a

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \ldots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

1	2	3	4	5	6	7	8	9
a	a	b	a	b	a	**a**	**b**	**a**

Suffix array:
i-th element represents the starting position of the lex. i-th smallest suffix.

LCP array: i-th element represents the length of the longest common prefix between the lex. i-th & $(i-1)$-th smallest suffix.

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	a a b a
1	4	a a b a b a a b a
7	1	**a b a**
4	3	a b a a b a
2	3	a b a b a a b a
8	0	b a
5	2	b a a b a
3	2	b a b a a b a

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \ldots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

1	2	3	4	5	6	7	8	9
a	**a**	**b**	**a**	b	a	a	b	a

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	**a a b a**
1	**4**	**a a b a** b a a b a
7	1	a b a
4	3	a b a a b a
2	3	a b a b a a b a
8	0	b a
5	2	b a a b a
3	2	b a b a a b a

Phrase starting at position 1

The length of the LCP between the suffix at position 1 and the its predecessor at position 6 is 4.

We choose “`aaba`” as the phrase.

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \ldots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

<u>**1**</u> 2 3 4 | <u>**5**</u> 6 | 7 8 9

a a b a | **b** **a** | a b a

<u>Phrase starting at position 5</u>

The length of the LCP between the suffix at position 5 and the its pred. at position 8 is 2.

We choose “`ba`” as the phrase.

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	a a b a
1	4	a a b a b a a b a
7	1	a b a
4	3	a b a a b a
2	3	a b a b a a b a
8	0	**b a**
<u>**5**</u>	2	**b a** a b a
3	2	b a b a a b a

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \ldots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

$$\begin{array}{cccc|cc|c|cc} \underline{1} & 2 & 3 & 4 & \underline{5} & 6 & \underline{7} & 8 & 9 \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathbf{a} & \mathtt{b} & \mathtt{a} \end{array}$$

Phrase starting at position 7

The length of the LCP between the suffix at position 7 and the its pred. at position 1 is 1.

We choose “a” as the phrase.

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	a a b a
1	4	**a** a b a b a a b a
$\underline{7}$	1	**a** b a
4	3	a b a a b a
2	3	a b a b a a b a
8	0	b a
5	2	b a a b a
3	2	b a b a a b a

Lex-parse [Navarro et al., 2021]

Def. The lex-parse of a string S is a parsing $s_1, \dots, s_v$, s.t. each phrase s_j starting at position $i = 1 + \sum_{k<j} |s_k|$ is $S[i..i + \max\{1, \ell\} - 1]$, where ℓ is the length of the longest common prefix between $S[i..]$ and its lex. predecessor $S[i'..]$.

input string

1	2	3	4	**5**	6	**7**	**8**	**9**
a	a	b	a	b	a	a	**b**	**a**

Phrases starting at positions 8 & 9

The length of the LCP is 0
(since it is the smallest suffix
that begins with the character).

We choose the character as the phrase.

Suffix array	LCP array	Suffixes sorted in lex. order
9	**0**	a
6	1	a a b a
1	4	a a b a b a a b a
7	1	a b a
4	3	a b a a b a
2	3	a b a b a a b a
8	**0**	b a
5	2	b a a b a
3	2	b a b a a b a

Compression by lex-parse

The bidirectional macro scheme is a compression scheme as follows.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

	1	2	3	4	5	6	7	8	9
$S =$	a	a	b	a	b	a	a	b	a

A BMS of S

$[4, 4] \leftarrow$ a

$[1, 3] \leftarrow [6, 8]$

$[6, 6] \leftarrow [4, 4]$

$[5, 5] \leftarrow$ b

$[7, 9] \leftarrow [4, 6]$

Compression by lex-parse

The bidirectional macro scheme is a compression scheme as follows.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

	1	2	3	4	5	6	7	8	9
S =	a	a	b	a	b	a	a	b	a

A BMS of S

$[4, 4] \leftarrow$ a

$[5, 5] \leftarrow$ b

$[1, 3] \leftarrow [6, 8]$

$[7, 9] \leftarrow [4, 6]$

$[6, 6] \leftarrow [4, 4]$

Decoding

1	2	3	4	5	6	7	8	9
			a	b				

Compression by lex-parse

The bidirectional macro scheme is a compression scheme as follows.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

	1	2	3	4	5	6	7	8	9
S =	a	a	b	a	b	a	a	b	a

A BMS of S

$[4, 4] \leftarrow$ a

$[5, 5] \leftarrow$ b

$[1, 3] \leftarrow [6, 8]$

$[7, 9] \leftarrow [4, 6]$

$[6, 6] \leftarrow [4, 4]$

Decoding

1	2	3	4	5	6	7	8	9
			$\underline{\text{a}}$	b	a			

Compression by lex-parse

The bidirectional macro scheme is a compression scheme as follows.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

	1	2	3	4	5	6	7	8	9
S =	a	a	b	a	b	a	a	b	a

A BMS of S

$[4, 4] \leftarrow$ a

$[5, 5] \leftarrow$ b

$[1, 3] \leftarrow [6, 8]$

$[7, 9] \leftarrow [4, 6]$

$[6, 6] \leftarrow [4, 4]$

Decoding

1	2	3	4	5	6	7	8	9
			a	b	a	a	b	a

Compression by lex-parse

The bidirectional macro scheme is a compression scheme as follows.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

	1	2	3	4	5	6	7	8	9
S =	a	a	b	a	b	a	a	b	a

A BMS of S

$[4, 4] \leftarrow$ a

$[5, 5] \leftarrow$ b

$[1, 3] \leftarrow [6, 8]$

$[7, 9] \leftarrow [4, 6]$

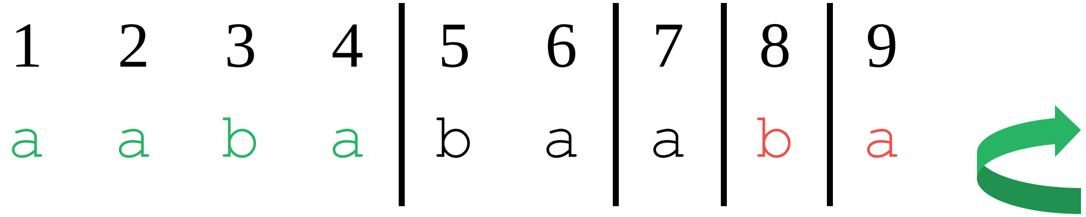
$[6, 6] \leftarrow [4, 4]$

Decoding

1	2	3	4	5	6	7	8	9
a	a	b	a	b	a	a	b	a

Compression by lex-parse

- The lex-parse gives a bidirectional macro scheme.
  - LCP means that there is another occ. in the text.
  - The position that has no common prefix is a single character phrase.



1	2	3	4	5	6	7	8	9
a	a	b	a	b	a	a	b	a

Suffix array	LCP array	Suffixes sorted in lex. order
9	0	a
6	1	a a b a
1	4	a a b a b a a b a
7	1	a b a
4	3	a b a a b a
2	3	a b a b a a b a
8	0	b a
5	2	b a a b a
3	2	b a b a a b a

BMS by lex-parse

- $[9, 9] \leftarrow$ a
- $[8, 8] \leftarrow$ b
- $[1, 4] \leftarrow [6, 9]$
- $[5, 6] \leftarrow [8, 9]$
- $[7, 7] \leftarrow [1, 1]$

Compression by lex-parse

The lex-parse gives a bidirectional macro scheme.

Def. [Storer & Szymanski, 1982] A bidirectional macro scheme of a string S is a set of directives of the following two types:

- $S[i..j] \leftarrow S[i'..j']$ (i.e. copy $S[i'..j']$ in $S[i..j]$), or
- $S[i] \leftarrow c$, with $c \in \Sigma$ (i.e. assign character c to $S[i]$).

- $b(S)$ denotes the minimum number of directives in BMS of a string S.
- $v(S)$ denotes the size of the lex-parse.
- For any string S, $b(S) \leq v(S)$ holds [Navarro et al., 2021].

Lex-parse gives a BMS.

Studies on lex-parse

- Lex-parse was proposed as a new variant in a family of ordered parsings that is considered as a generalization of the LZ parsing and a subset of bidirectional macro schemes.
  - LZ parsing: Each phrase refers **a smaller suffix w.r.t. the text position**.
  - Lex-parse: Each phrase refers **a smaller suffix w.r.t. the lex. order**.
- Combinatorial studies on lex-parse can lead us to further understanding in string repetitiveness measures and compressors.
  - A direct relation $v \in O(r)$ between the size of the lex-parse v and the size r of the RLBWT, one of the most important dictionary compressors, holds [Navarro et al., 2021].
  - A direct relation between the lex-parse v and the LZ parsing z is open.

Our contribution (1) – Edit-sensitivity

Multiplicative edit-sensitivity

Measures	Edit operations	Upper bounds	Lower bounds
Lex-parse (*v*)	all	$O(\log (n/b))$	$\Omega(\log n)$

Since we use a family of strings s.t. $b = O(1)$,
this bound does not contradict the upper bound.

Overview

- The upper bound can be obtained by combining some known results (regarding the lex-parse and the bidirectional macro scheme).
- The lower bound can be obtained by the Fibonacci word.
  - Combinatorial properties on the Fibonacci word
  - Characterizing the lex-parse by Lyndon factorization

A new sensitivity – Alphabet ordering sensitivity

We introduce a new sensitivity of compressors by alphabet orderings.

- If a compressor C is defined by using the lexicographic order, the size $C(S)$ depends on alphabet orderings.

For instance,

- **Lex-parse**
- (Run-length) Burrows-Wheeler transform
- GCIS (a grammar comp. algorithm based on induced sorting)

Multiplicative sensitivity

$$\max_{S\in\Sigma^n}\left\{\frac{C(S,<_2)}{C(S,<_1)}\,\middle|\,<_1,<_2\in A\right\}$$

Additive sensitivity

$$\max_{S\in\Sigma^n}\{C(S,<_2)-C(S,<_1) \mid <_1,<_2\in A\}$$

In this work,
we investigate the <u>multiplicative sensitivity</u>.

Studies on alphabet orderings

Optimization problems of these kinds of structures have been studied.

- Complexity (NP-hardness)
  - Minimization for the RLBWT [Bentley et al., 2020]
  - Minimization/Maximization for the Lyndon factorization [Gibney & Thankachan, 2021]
  - Minimization for the minimizer [Verbeek et al., 2024]
- Exact algorithms/heuristics
  - for RLBWT and Lyndon factorization [Cenzato et al., 2023] [Clare & Daykin, 2019] [Clare et al., 2019] [Major et al., 2020]

Our contribution (2) – AO-Sensitivity

Multiplicative AO-sensitivity

Measures	Upper bounds	Lower bounds	
Run-length BWT (r)	$O(\log^2 n)$	$\Omega(\log n)$	from known results
Lyndon factorization†	$O(n)$	$\Omega(n)$	obvious
Lex-parse	$O(\log (n/b))$	$\Omega(\log n)$	our paper

$b = O(1)$

† Lyndon factorization is not a compressor, but a popular object depending on the lex. order.

Overview

- The upper bound can be obtained by combining some known results (regarding the lex-parse and the bidirectional macro scheme).
- The lower bound can be obtained by the Fibonacci word over $\{a, b\}$.
  - Although the results for “$a < b$” have been proven [Navarro et al., 2021], we give alternative proofs for this case that leads us to the proof for the case when “$b < a$”.

Our contribution – summary

Multiplicative Edit-sensitivity (for all operations)

Measures	Upper bounds	Lower bounds
Lex-parse (v)	$O(\log(n/b))$	$\Omega(\log n)$

Multiplicative AO-sensitivity

Measures	Upper bounds	Lower bounds
Lex-parse (v)	$O(\log(n/b))$	$\Omega(\log n)$

In this talk, I will explain

- the upper bounds for both sensitivities, and
- the lower bound for the edit-sensitivity.

Upper bounds for both problems

The upper bounds can be obtained by combining some known results.

Lemma. For any string S and T such that $\mathrm{ED}(S, T) = 1$,

- $b(S) \leq v(S)$ [Navarro et al., 2021],
- $v(S) \in O(b(S) \log(n/b(S)))$ [Navarro et al., 2021],
- $b(T) \leq 2b(S)$ [Akagi et al., 2023].

- Multiplicative Edit-sensitivity

$$\frac{v(T)}{v(S)} \leq \frac{v(T)}{b(S)} \in O\left(\frac{b(T)\log\frac{n}{b(T)}}{b(S)}\right) \subseteq O\left(\frac{b(S)\log\frac{n}{b(S)}}{b(S)}\right) = O\left(\log\frac{n}{b(S)}\right)$$

- Multiplicative AO-sensitivity

$$\frac{v(S,<_2)}{v(S,<_1)} \leq \frac{v(S,<_2)}{b(S)} \in O\left(\frac{b(S)\log\frac{n}{b(S)}}{b(S)}\right) = O\left(\log\frac{n}{b(S)}\right)$$

Lower bound for Edit-sensitivity

In the rest of this talk,
I will explain our ideas for the lower bound for edit-sensitivity.

The formal statement for our result is given as follows.

Theorem. There exists a family of strings α_k, β_k that satisfies $\mathrm{ED}(\alpha_k, \beta_k) = 1$ and $v(\beta_k) / v(\alpha_k) \in \Omega(\log n)$.

- We use (finite) Fibonacci words F_{2k} as the base strings α_k.

Fibonacci word

Def. The i-th Fibonacci word over $\{\mathtt{a}, \mathtt{b}\}$ is defined as follows:

$$F_1 = \mathtt{b},\ F_2 = \mathtt{a},\ F_i = F_{i-1} \cdot F_{i-2}.$$

f_i	F_i		
1	F_1	=	b
1	F_2	=	a
2	F_3	=	ab
3	F_4	=	aba
5	F_5	=	abaab
8	F_6	=	**abaababa**
13	F_7	=	**abaababaabaab**
21	F_8	=	**abaababaabaababaababa**
34	F_9	=	abaababaabaababaababaabaababaabaab
55	F_{10}	=	abaababaabaababaababaabaababaabaababaababaabaababaababa

f_i denotes the length $|F_i|$.
(= f_i corresponds the Fibonacci sequence.)

Fibonacci word

Def. The i-th Fibonacci word over $\{\mathtt{a}, \mathtt{b}\}$ is defined as follows:

$$F_1 = \mathtt{b},\ F_2 = \mathtt{a},\ F_i = F_{i-1} \cdot F_{i-2}.$$

f_i	F_i		
1	F_1	=	b
1	F_2	=	a
2	F_3	=	ab
3	F_4	=	a**ba**
5	F_5	=	aba**ab**
8	F_6	=	abaaba**ba**
13	F_7	=	abaababaaba**ab**
21	F_8	=	abaababaabaababaaba**ba**
34	F_9	=	abaababaabaababaababaabaababaaba**ab**
55	F_{10}	=	abaababaabaababaababaabaababaabaababaababaabaababaaba**ba**

Property. For every even (resp. odd) i, the suffix of length 2 is **ba** (resp. ab).

Property.
aaa and bb cannot appear.

Lower bound for Edit-sensitivity

Theorem. There exists a family of strings α_k, β_k that satisfies $\mathrm{ED}(\alpha_k, \beta_k) = 1$ and $v(\beta_k)$ / $v(\alpha_k) \in \Omega(\log n)$.

- We use (finite) Fibonacci words F_{2k} as the base strings α_k.
  - ▻ It is known that $v(F_{2k}) \in \mathrm{O}(1)$ [Navarro et al., 2021].
- Let $\beta_k = T_{2k}$ be the string obtained from F_{2k} by substituting the rightmost `b` of F_{2k} with `a`.

Here, we focus on the substitution.

Example for $k = 5$

F_{10} = abaababaabaababaababa|abaababaabaababaababaabaababaaba|**b**|a

substitution

T_{10} = abaababaabaababaababaabaababaabaa|babaaba|baabaa|ba|baa|ba|**a**|a

Lower bound for Edit-sensitivity

Theorem. There exists a family of strings α_k, β_k that satisfies $\mathrm{ED}(\alpha_k, \beta_k) = 1$ and $v(\beta_k) / v(\alpha_k) \in \Omega(\log n)$.

- We use (finite) Fibonacci words F_{2k} as the base strings α_k.
  - ▻ It is known that $v(F_{2k}) \in \mathrm{O}(1)$ [Navarro et al., 2021].
- Let $\beta_k = T_{2k}$ be the string obtained from F_{2k} by substituting the rightmost `b` of F_{2k} with `a`.

Here, we focus on the substitution.

Theorem. $v(T_{2k}) = 2k - 2$ holds for every $k \geq 6$.

$k \in \Theta(\log n)$

Example for $k = 5$

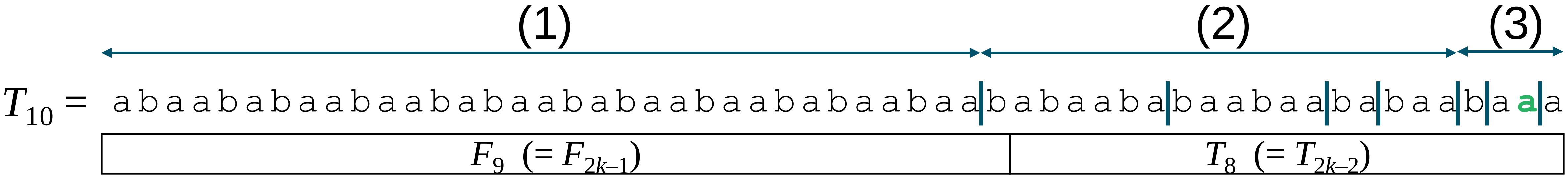
T_{10} = a b a a b a b a a b a a b a b a a b a b a a b a a b a b a a b a a | b a b a a b a | b a a b a a | b a | b a a | b | a **a** | a

Sketch of proof

Lemma. The lengths of the phrases of the lex-parse of T_{2k} are

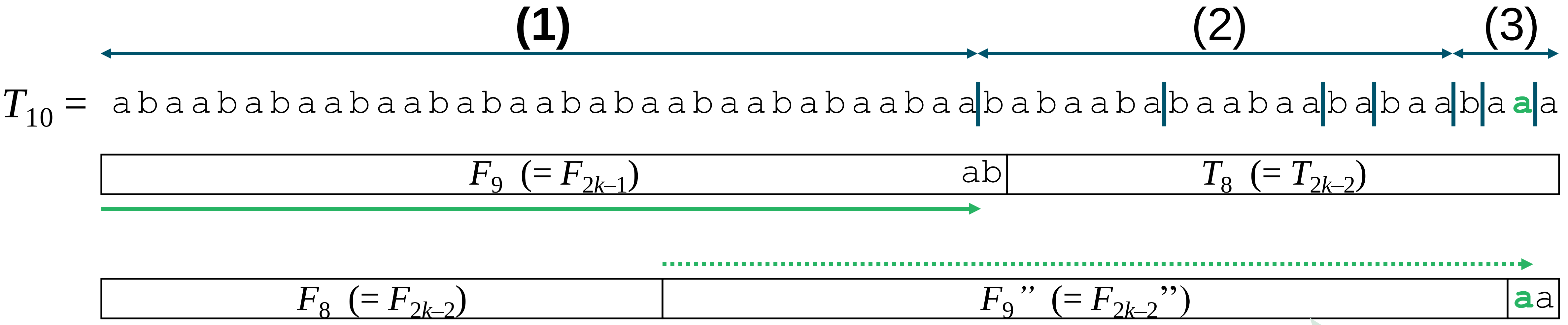
$$f_{k-1}-1,\ (f_{k-4}-1,\ f_{k-5}+1,\ \ldots,\ f_4-1,\ f_3+1),\ 1,\ 2,\ 1.$$

There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.



Sketch of proof

There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.



(1) (2) (3)

T_{10} = `abaababaabaababaababaabaababaabaa|babaaba|baabaa|ba|baa|b|aa|a`

F_9 $(= F_{2k-1})$ ab | T_8 $(= T_{2k-2})$

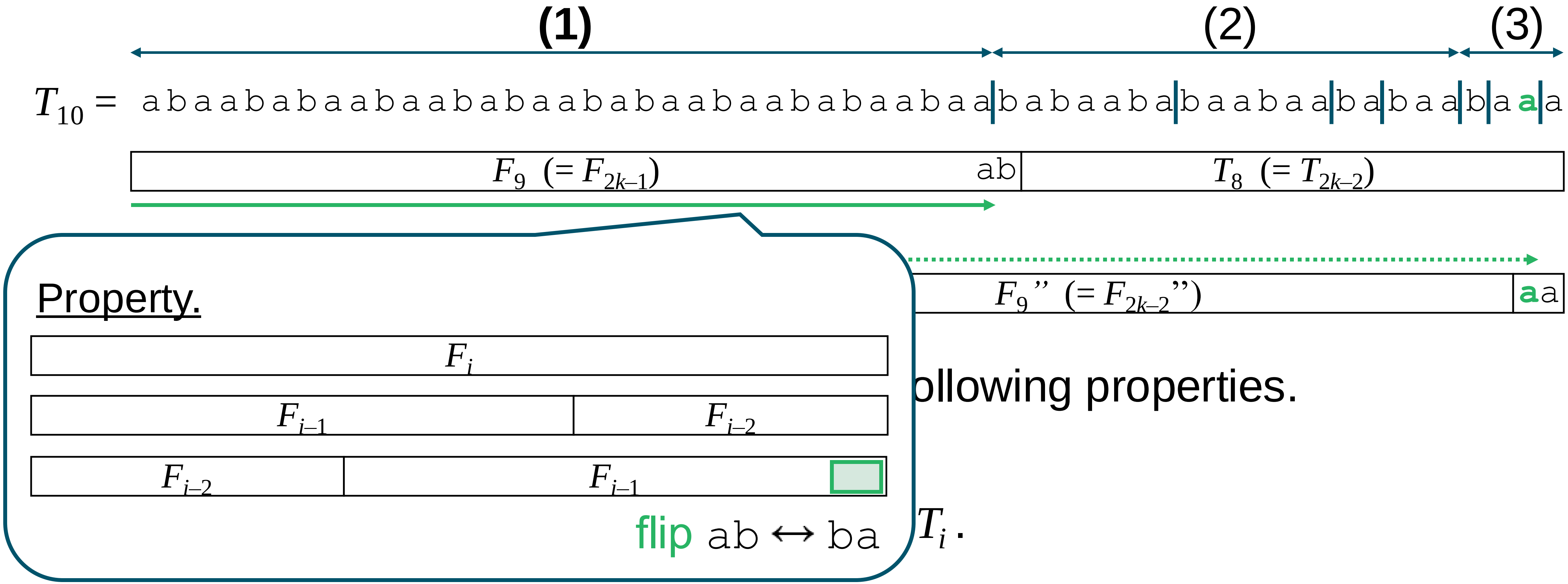
F_8 $(= F_{2k-2})$ | F_9'' $(= F_{2k-2}$'') | aa

- We can show the reference by using the following properties.
  - ▻ F_{i-1} only occurs as a prefix of F_i .
  - ▻ Suffix "`aaa`" cannot occur as an infix of T_i .

For any string S,
$S' = S[1..|S|-1]$, $S'' = S[1..|S|-2]$.

Sketch of proof

There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.

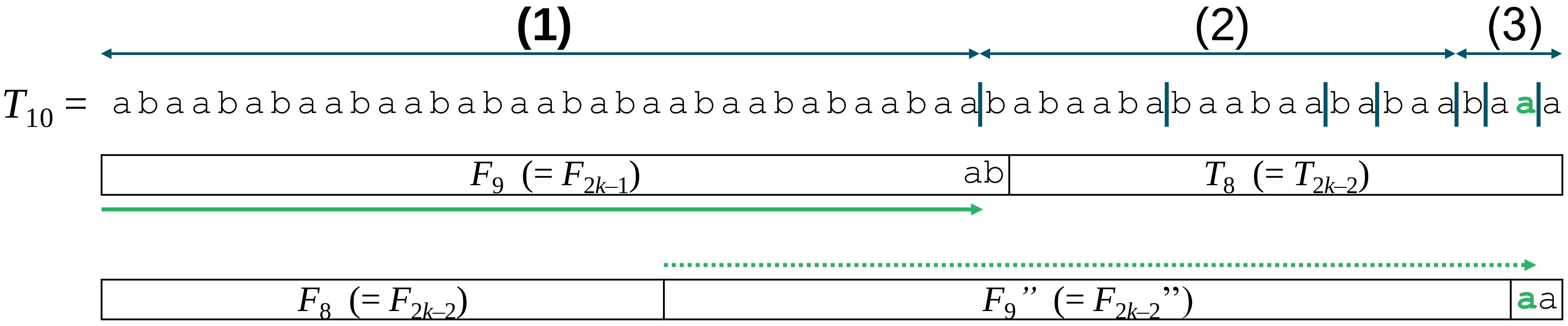


ollowing properties.

T_i .

Sketch of proof

There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.



- We can show the reference by using the following properties.
  - F_{i-1} only occurs as a prefix of F_i .
  - Suffix "`aaa`" cannot occur as an infix of T_i .

Sketch of proof

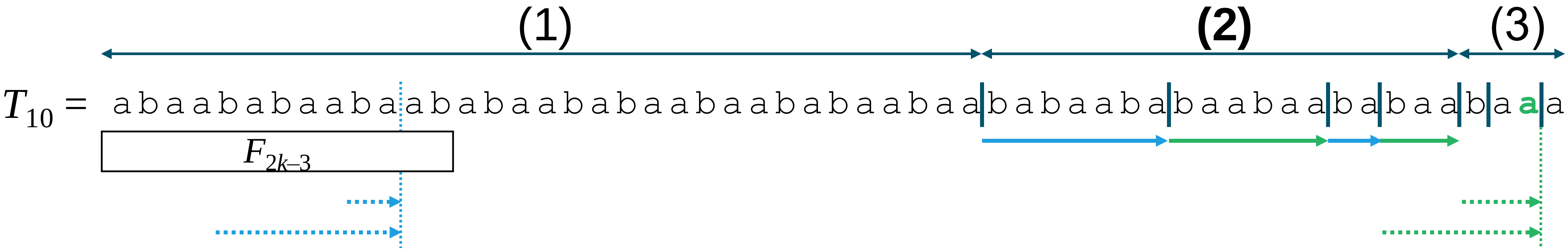
There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.

T_{10} = `abaababaabaababaababaabaababaabaa|babaaba|baabaa|ba|baa|`**`b`**`|a`**`a`**`|`**`a`**

- It is clear that
  - suffix ”`a`” is the smallest suffix of T_{2i} ,
  - suffix ”`aa`” is the second smallest suffix of T_{2i} ,
  - suffix ”`aaa`” is the third smallest suffix of T_{2i} ,
  - suffix ”`baaa`” is the smallest suffix of T_{2i} that begins with ”`b`”.
- These facts implies the last three phrases are ”**`b`**”, ”`aa`”, ”**`a`**”.

Sketch of proof

There are three types of phrases:
(1) first phrase, (2) inductive phrases, and (3) last three phrases.



Key observations

- The sources of odd phrases in (2) ends at position $f_{2k-3}-2$ (left-side).
- The sources of even phrases in (2) ends at position $f_{2k}-1$ (right-side).

We can characterize the structure by using the Lyndon factorization.

Lyndon word [Lyndon, 1958]

Def. A string λ is a Lyndon word,
if λ is lexicographically smaller than any of its non-empty proper suffixes.

proper suffixes

$$
\begin{aligned}
\lambda = \mathtt{a\,b\,a\,b\,b} \quad & \prec \quad \mathtt{b\,a\,b\,b} \\
& \prec \quad \mathtt{a\,b\,b} \\
& \prec \quad \mathtt{b\,b} \\
& \prec \quad \mathtt{b}
\end{aligned}
$$

`ababb` is a Lyndon word.

Lyndon factorization [Lyndon, 1958]

Def. A factorization $\lambda_1^{p_1}, \ldots, \lambda_m^{p_m}$ of S is the Lyndon factorization $LF(S)$, if

- $\lambda_1 \succ \ldots \succ \lambda_m$ are Lyndon words, and
- $p_i \geq 1$ for all $1 \leq i \leq m$.

$$\lambda = \mathtt{a\,b\,b} \vdots \mathtt{a\,b\,b} \mid \mathtt{a\,b\,a\,b\,b} \mid \mathtt{a\,b} \vdots \mathtt{a\,b} \vdots \mathtt{a\,b} \mid \mathtt{a} \vdots \mathtt{a}$$

$$\mathtt{abb} = \mathtt{abb} \succ \mathtt{ababb} \succ \mathtt{ab} = \mathtt{ab} = \mathtt{ab} \succ \mathtt{a} = \mathtt{a}$$

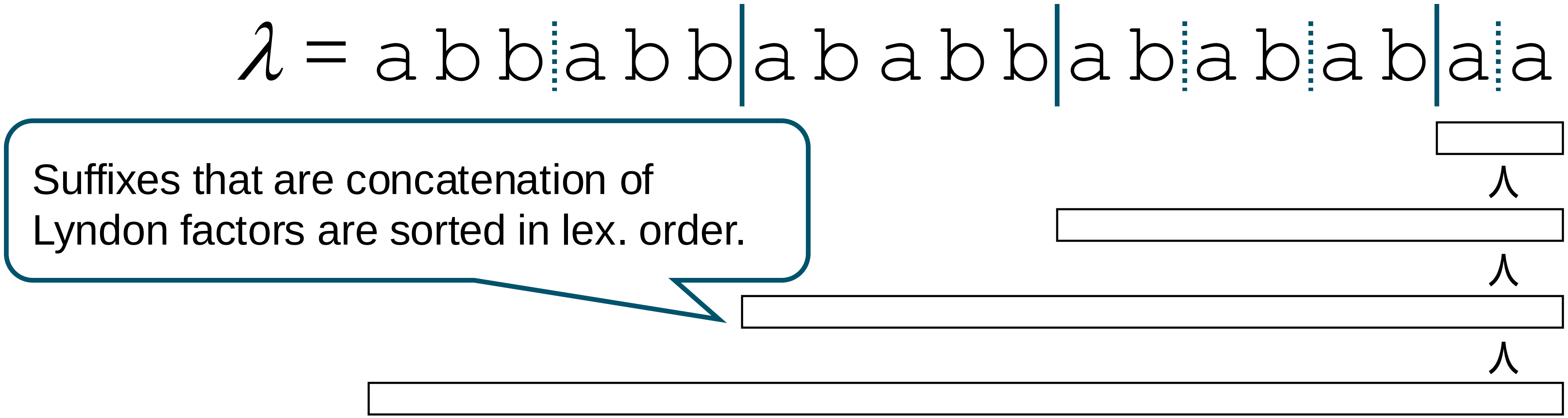
$$LF(\lambda) = (\mathtt{abb})^2 \quad \mathtt{ababb} \quad (\mathtt{ab})^3 \quad (\mathtt{a})^2$$

For any string S, the Lyndon factorization of S is unique.

Lyndon factorization [Lyndon, 1958]

Def. A factorization $\lambda_1^{p_1}, \ldots, \lambda_m^{p_m}$ of S is the Lyndon factorization $LF(S)$, if

- $\lambda_1 \succ \ldots \succ \lambda_m$ are Lyndon words, and
- $p_i \geq 1$ for all $1 \leq i \leq m$.



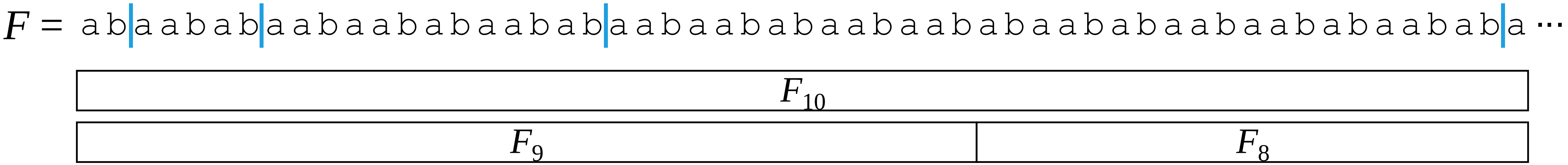
Suffixes that are concatenation of Lyndon factors are sorted in lex. order.

Property. For any integer $i \in [1, m\!-\!1]$, $\lambda_i^{p_i} \cdots \lambda_m^{p_m} \succ \lambda_{i+1}^{p_{i+1}} \cdots \lambda_m^{p_m}$ holds.

Lyndon factorization of infinite Fibonacci word

Def. Let $F = \lim_{i \to \infty} F_i$ be the infinite Fibonacci word, and $LF(F) = \ell_1, \ell_2, \ldots$ be the (infinite) Lyndon factorization of F.

Lemma. [Melançon, 2000] Let ϕ be a string morphism s.t. $\phi(\mathtt{a}) = \mathtt{aab}$, $\phi(\mathtt{b}) = \mathtt{ab}$. $\ell_1 = \mathtt{ab}$, $\ell_{i+1} = \phi(\ell_i)$, and $|\ell_i| = f_{2i+1}$ holds.



$F = \mathtt{ab|aabab|aabaababaabab|aabaababaabaababaababaabaababaabab|a} \cdots$

To show our result, we consider the Lyndon factorization of $F_i'' = T_i''$ (the string that can be obtained by removing the edited suffix "$\mathtt{aa}$").

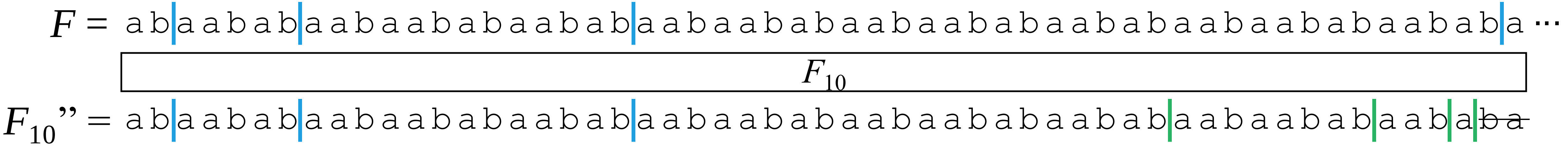
Lyndon factorization of a prefix of Fibonacci word

Lemma. Let L_i be the i-th Lyndon factor of F_{2k}''.

- $L_i = \phi^{i-1}(\mathtt{ab}) = \ell_i$ if $1 \leq i < k-1$,
- $L_i = \phi^{2k-i-3}(\mathtt{a})$ if $k-1 \leq i \leq 2k-3$.

- The total length of the first $k-1$ factors of F is $f_{2k}-1$.
- Thus, $LF(F_{2k}'') = \ell_1, \ldots, \ell_{k-2}, LF(\ell_{k-1}')$ holds.
  - Because $\phi^i(\lambda)$, for any Lyndon word λ, is also a Lyndon word, we can see that $LF(\ell_i') = \phi^{i-1}(\mathtt{a}), \ldots, \phi^0(\mathtt{a})$.

The last Lyndon factor of F which is in F_{2k} is factorized into shorter factors.



F = ab|aabab|aabaababaabab|aabaababaabaababaababaabaababaabab|a ···

F_{10}

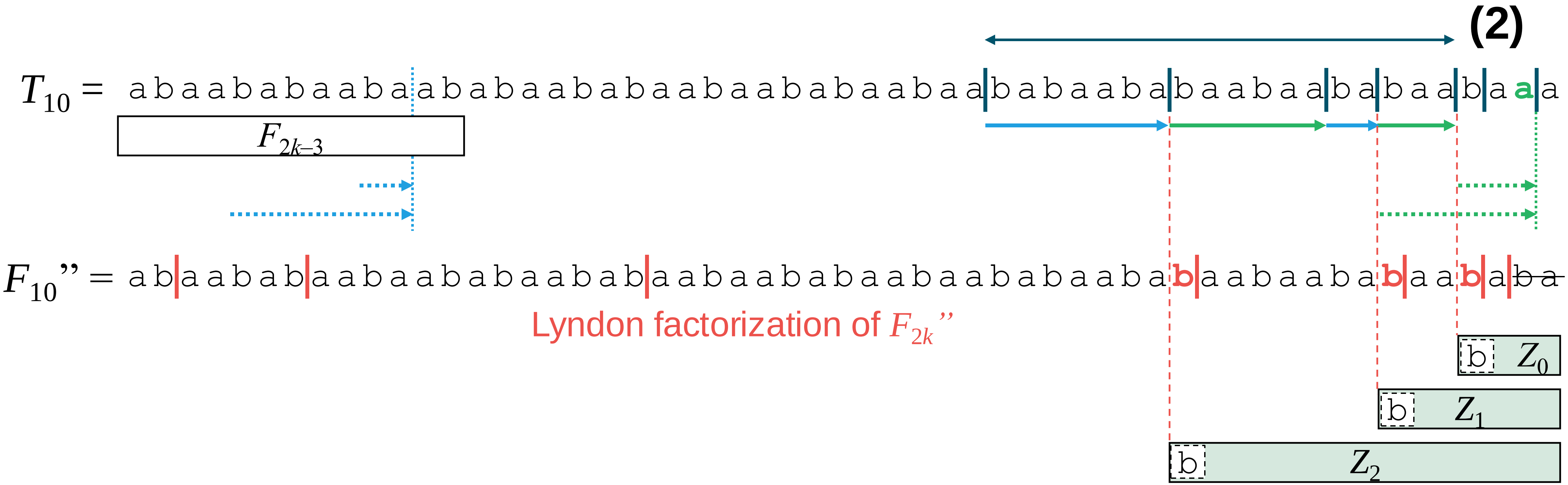
F_{10}'' = ab|aabab|aabaababaabab|aabaababaabaababaabab|aabaabab|aab|a|~~ba~~

How can we use the Lyndon factorization?

- A suffix that is “**b** + a concatenation of Lyndon factors” corresponds to the even phrase of the lex-parse.

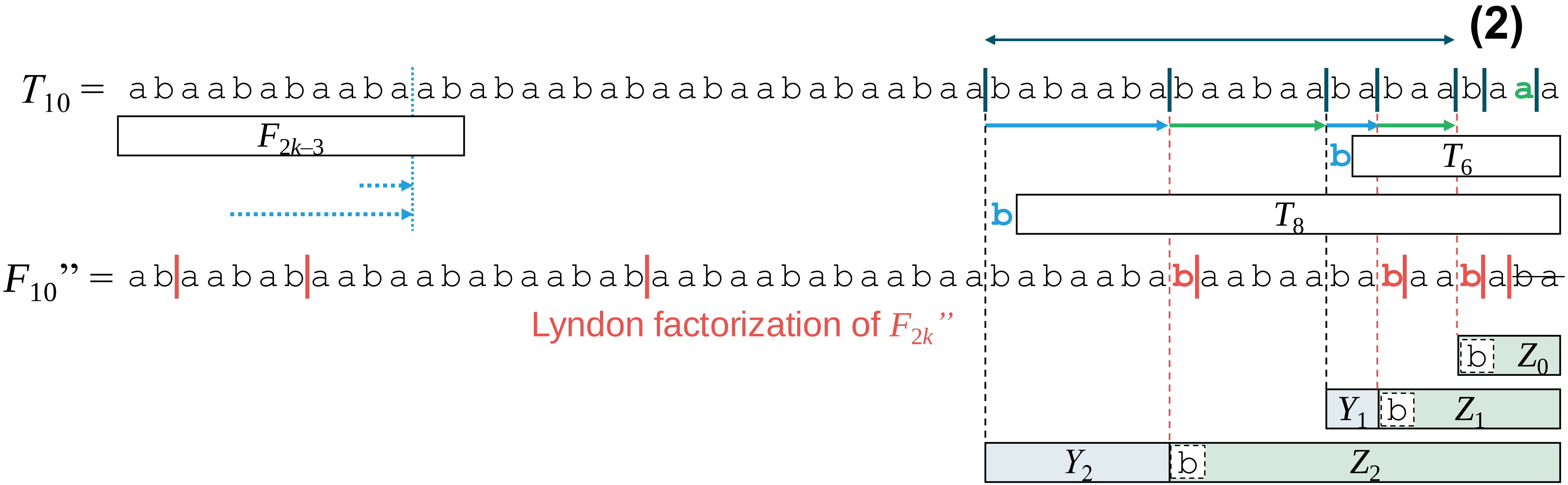
Define as Z_i

Lemma. The predecessor of Z_i among the suffixes of T_{2k} is Z_{i-1}.
Namely, Z_i refers Z_{i-1}in the lex-parse of T_{2k}.



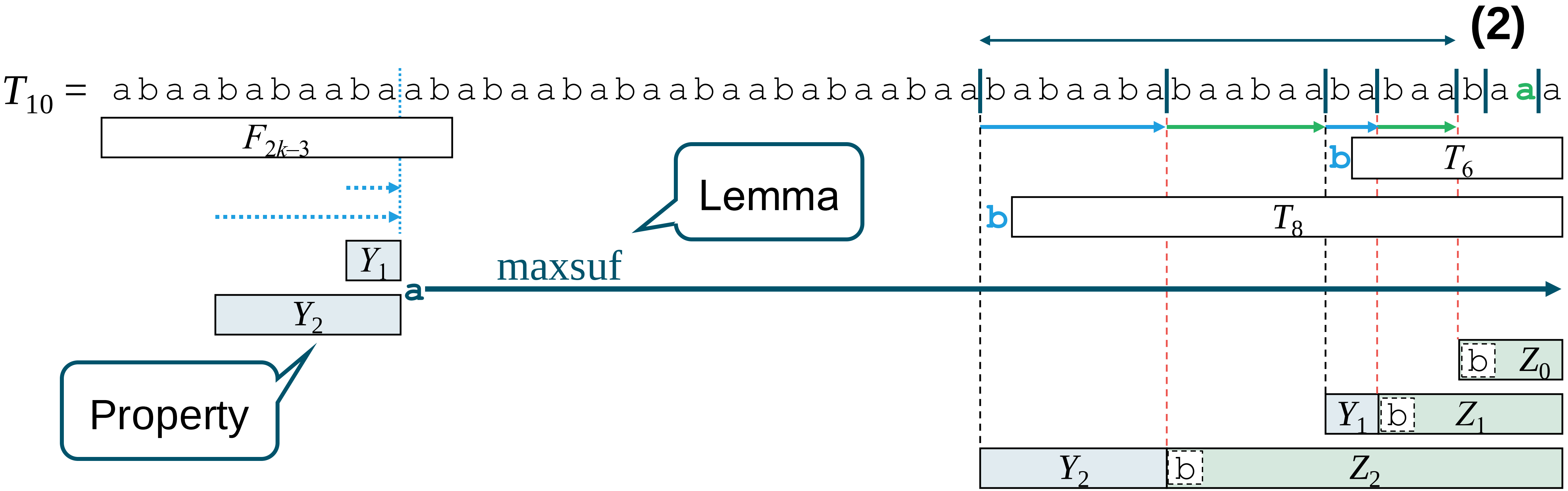
How can we use the Lyndon factorization?

- We define Y_i as the odd phrases of the lex-parse (see figure).
- Then, we can see that $Y_i Z_i$ = “**b** + suffix T_{2i+4}”.



How can we use the Lyndon factorization?

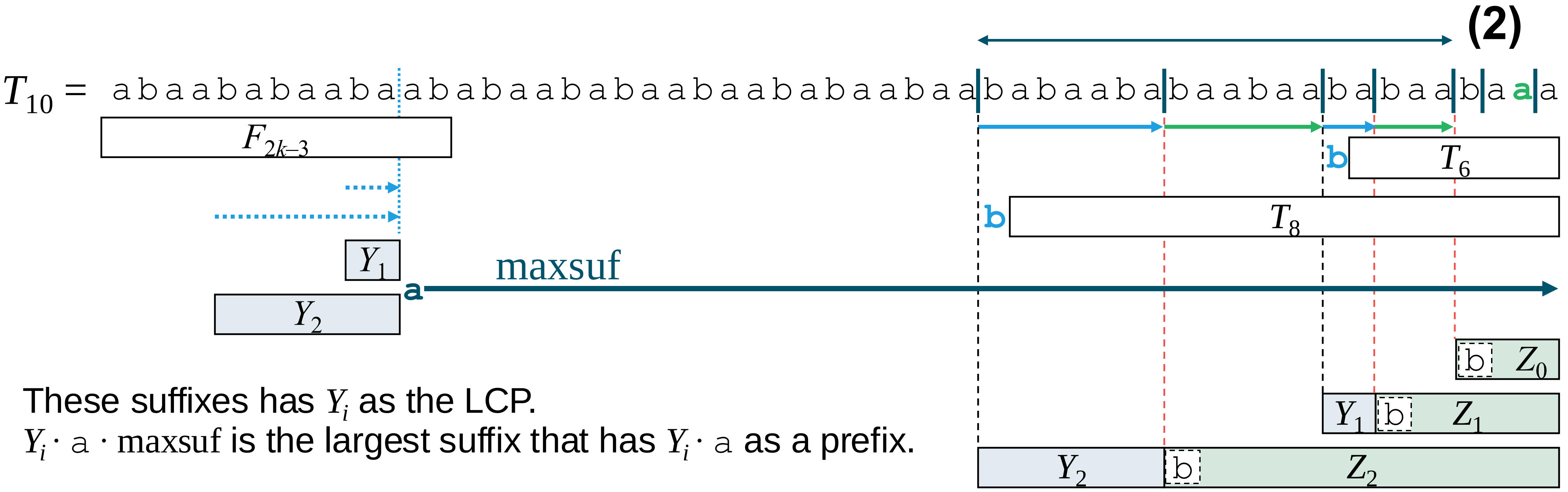
- We can also show
  - ▻ **Lemma.** The max. suffix (maxsuf) of T_{2k} is begins at position f_{2k-3}.
  - ▻ **Property.** Every Y_i has an occ. that ends at position $f_{2k-3}-2$.



How can we use the Lyndon factorization?

- Then, we can obtain the following lemma.

Lemma. The predecessor of $Y_i Z_i$ among the suffixes of T_{2k} is $Y_i \cdot \texttt{a} \cdot \mathrm{maxsuf}$.
Namely, $Y_i Z_i$ refers $Y_i \cdot \texttt{a} \cdot \mathrm{maxsuf}$ in the lex-parse of T_{2k}.



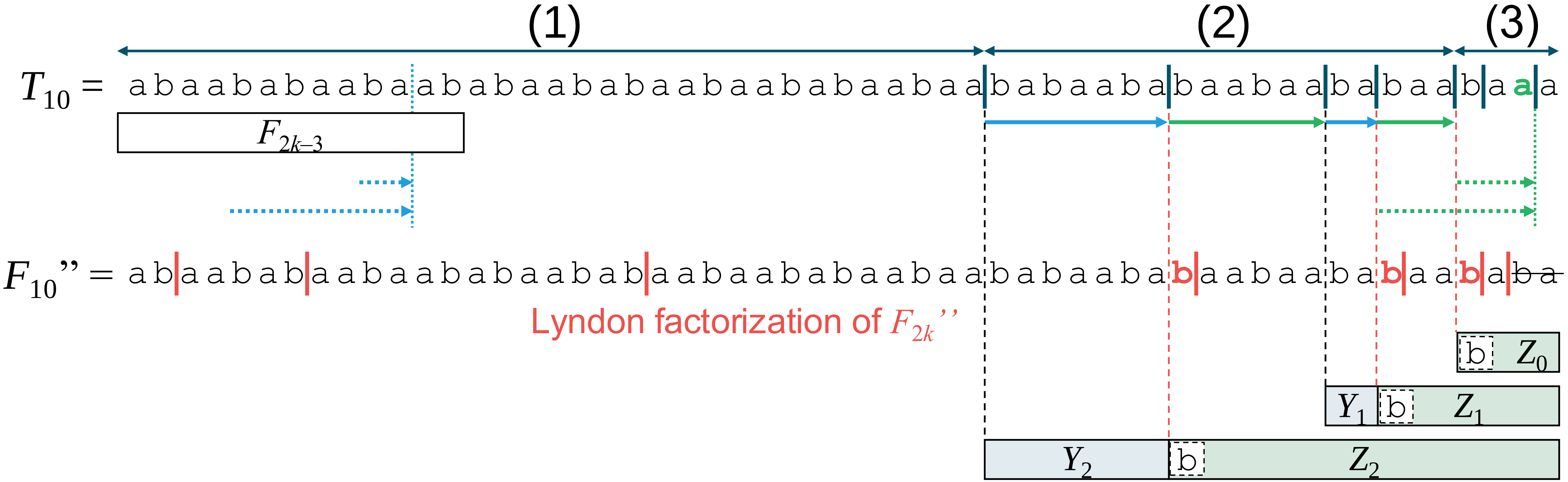
These suffixes has Y_i as the LCP.
$Y_i \cdot \texttt{a} \cdot \mathrm{maxsuf}$ is the largest suffix that has $Y_i \cdot \texttt{a}$ as a prefix.

Lower bound for Edit-sensitivity

Lemma. The lengths of the phrases of the lex-parse of T_{2k} are

$$f_{k-1}-1,\ (f_{k-4}-1,\ f_{k-5}+1,\ \ldots,\ f_4-1,\ f_3+1),\ 1,\ 2,\ 1.$$

Theorem. $v(T_{2k}) = 2k - 2$ holds for every $k \geq 6$.



Conclusion

Multiplicative Edit-sensitivity (for all operations)

Measures	Upper bounds	Lower bounds
Lex-parse (v)	$O(\log (n/b))$	$\Omega(\log n)$

Multiplicative AO-sensitivity

Measures	Upper bounds	Lower bounds
Lex-parse (v)	$O(\log (n/b))$	$\Omega(\log n)$

Further work (on alphabet–orderings)

- Problem of computing optimal alphabet orderings for the lex-parse.
  - The problems for the RLBWT [Bentley et al., 2020] and the Lyndon factorization [Gibney & Thankachan, 2021] are known to be NP-hard.