Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets

PSC 2019

Keisuke Goto
Fujitsu Laboratories Ltd.

Suffix Arrays and LCP Arrays

- Suffix arrays sort all suffixes and store their starting positions
- LCP arrays store the length of longest common prefix of the consecutive suffixes in the suffix array

	1	2	3	4	5	6	7
T	b	a	n	a	n	a	$

suffix array and LCP array of **T**

i	LCP	SA	$T_{SA[i]}$
1	0	7	$
2	0	6	a$
3	1	4	ana$
4	3	2	anana$
5	0	1	banana$
6	0	5	na$
7	2	3	nana$

Problems

Input

T

Output

Problem 1 **SA**

Problem 2 **SA** **LCP**

Assumption

- T is read-only string of length N
- Word RAM mode of word size log N
- T consists of an integer alphabet $[1\ldots\sigma]$
- all σ characters appear in **T**

Stronger assumption than previous research

Example

T = banana$ from

{$←1, a←2, b←3, n←4}

Our Contributions

Problem 1: Construction of **SA**

	Time	**Extra Words**
[Manber and Mayers,1990]	$O(N \log N)$	$O(N)$
[Kim+, 2003], [Ko and Aluru, 2003], [Karkkainen Sanders, 2003]	$O(N)$	$O(N)$
[Franceschini and Muthukrishnan, 2007]	$O(N \log N)$	$O(1)$
[Nong, 2013]	$O(N)$	$\sigma + O(1)$
Ours	$O(N)$	$O(1)$

Space except for input and output space

Recent and Independent Works

- [Li et al., 2018] also proposed an optimal time and space algorithm for Problem 1 (Construction of SA)

	[Li et al., 2018]	Ours
Alphabet size	$\sigma \in O(N)$	$\sigma \leqq N$
All characters appear in T?	May not	Must
Framework	Induced sorting	Induced sorting
Main complex external tools	In-place Merging for two sorted arrays [Chen 2003] Succinct data structures for select queries [Jacobson, 1989]	In-place Merging for two sorted arrays [Chen 2003]

Recent and Independent Works

Our work may contribute to develop practical time and space efficient implementations for Problem 1

	[Li et al., 2018]	Ours
Alphabet size	$\sigma \in O(N)$	$\sigma \leqq N$
All characters appear in T?	May not	Must
Framework	Induced sorting	Induced sorting
Main complex external tools	In-place Merging for two sorted arrays [Chen 2003] Succinct data structures for select queries [Jacobson, 1989]	In-place Merging for two sorted arrays [Chen 2003]

Our Contributions

Problem 1: Construction of **SA**

	Time	Extra Words
[Manber and Mayers,1990]	$O(N \log N)$	$O(N)$
[Kim+, 2003], [Ko and Aluru, 2003], [Karkkainen Sanders, 2003]	$O(N)$	$O(N)$
[Franceschini and Muthukrishnan, 2007]	$O(N \log N)$	$O(1)$
[Nong, 2013]	$O(N)$	$\sigma + O(1)$
Ours	$O(N)$	$O(1)$

Space except for input and output space

Focus on Problem 1 in this talk

Problem 2: Construction of **SA + LCP**

	Time	Extra Words
[Kasai+, 2001]	$O(N)$	$N + O(1)$
[Manzini, 2004]	$O(N)$	$\sigma + O(1)$
[Nong, 2013] + [Manzini, 2004]	$O(N)$	$\sigma + O(1)$
Ours	$O(N)$	$O(1)$

Input: **T** and **SA**
Output: **LCP**

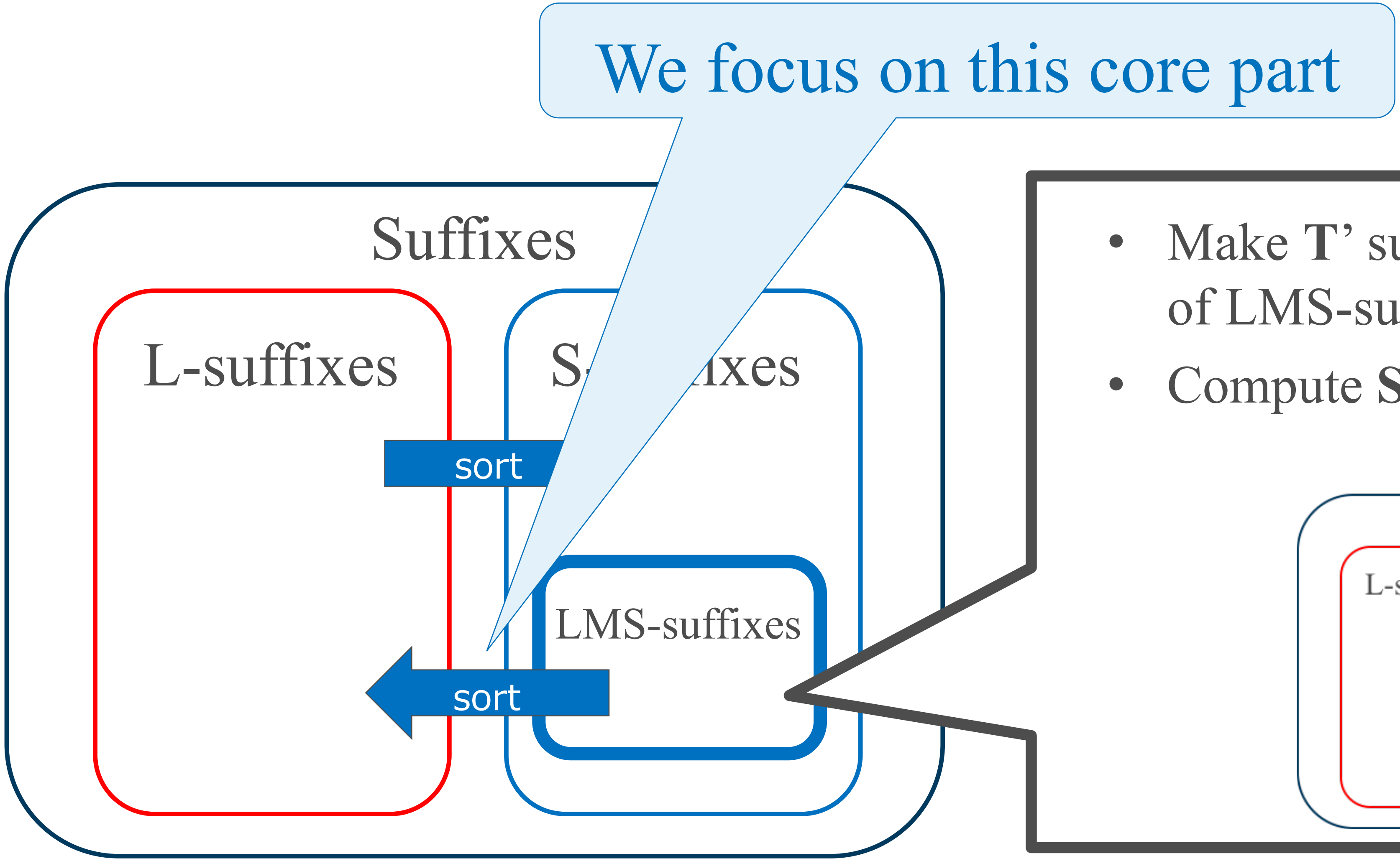
Input: **T**
Output: **SA** and **LCP**

- Problems
- Induced Sorting Framework
- Optimal Time and Space Algorithm
- Summary

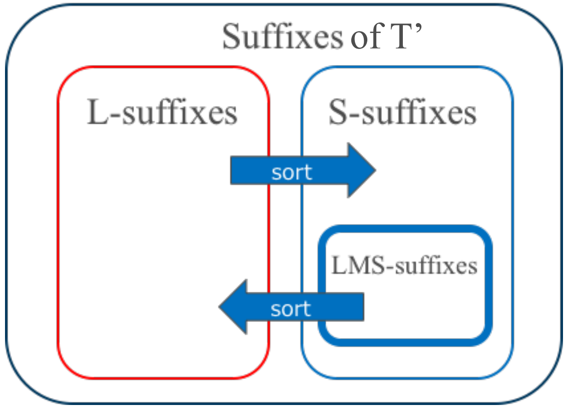
Induced Sorting Frameworks

[Ko and Aluru, 2003]
[Nong et al., 2011]

- Sort suffixes from sorted suffixes of smaller size



- Make **T'** such that **SA** of **T'** equals **SA** of LMS-suffixes
- Compute **SA** of **T'** recursively



Type of Suffixes

- Suffix $\mathbf{T}_i$ ($\mathbf{T}[i..N]$) is an L(arger)-suffix if $\mathbf{T}_i > \mathbf{T}_{i+1}$
- Suffix $\mathbf{T}_i$ ($\mathbf{T}[i..N]$) is an S(maller)-suffix if $\mathbf{T}_i < \mathbf{T}_{i+1}$

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

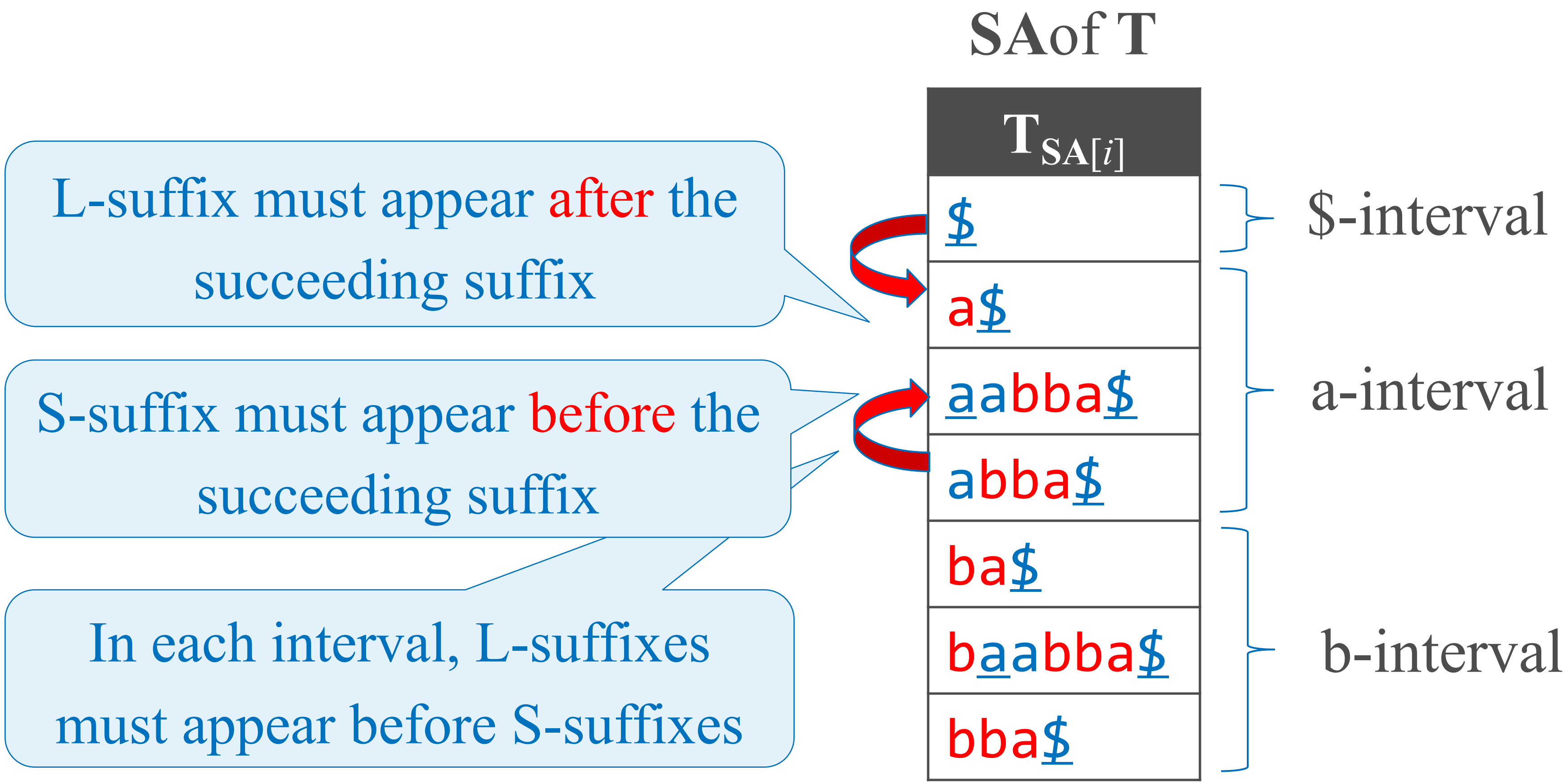
Left-most S-suffix (LMS-suffix)

a $ > $

aabba$ < abba$

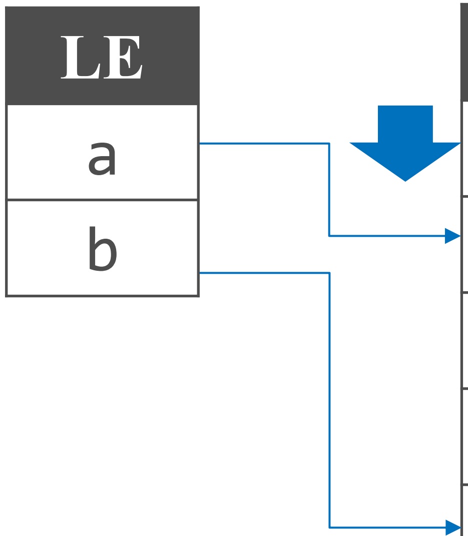
Type of Suffixes

- Suffix $\mathbf{T}_i$ ($\mathbf{T}[i..N]$) is an L(arger)-suffix if $\mathbf{T}_i > \mathbf{T}_{i+1}$
- Suffix $\mathbf{T}_i$ ($\mathbf{T}[i..N]$) is an S(maller)-suffix if $\mathbf{T}_i < \mathbf{T}_{i+1}$



Sorting L-suffixes from sorted LMS-suffixes

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S



A	$T_{SA[i]}$
$	$
	a$
	aabba$
aabba$	abba$
	ba$
	baabba$
	bba$

Use three arrays

- **A**: will be **SA**
- **LE**: indicate the leftmost empty position of each interval — σ extra words
- **type**: store the type of each suffix — $N / \log N$ extra words

Preliminary, we store sorted LMS-suffixes in the tail of each interval

Sorting L-suffixes from sorted LMS-suffixes

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

LE
a
b

A	$\mathbf{T}_{\mathbf{SA}[i]}$
$	$
	a$
	aabba$
aabba$	abba$
	ba$
	baabba$
	bba$

With a left-to-right scan on **A**

- Read a suffix $\mathbf{A}[i]=\mathbf{T}_j$, lexicographically
- Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not
- If so, we store $\mathbf{T}_{j\text{-}1}$ at the leftmost empty position $\mathbf{LE}[t_{j\text{-}1}]$ of $t_{j\text{-}1}$-interval

$t_{j\text{-}1}$: Starting character of $\mathbf{T}_{j\text{-}1}$

Sorting L-suffixes from sorted LMS-suffixes

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

LE
a
b

A	$\mathbf{T}_{\mathbf{SA}[i]}$
$	$
a$	a$
	aabba$
aabba$	abba$
	ba$
	baabba$
	bba$

With a left-to-right scan on **A**

- Read a suffix $\mathbf{A}[i]=\mathbf{T}_j$, lexicographically
- Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not
- If so, we store $\mathbf{T}_{j\text{-}1}$ at the leftmost empty position $\mathbf{LE}[t_{j\text{-}1}]$ of $t_{j\text{-}1}$-interval

$t_{j\text{-}1}$: Starting character of $\mathbf{T}_{j\text{-}1}$

Sorting L-suffixes from sorted LMS-suffixes

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

LE
a
b

A	$T_{SA[i]}$
$	$
a$	a$
	aabba$
aabba$	abba$
	ba$
	baabba$
	bba$

With a left-to-right scan on **A**

- Read a suffix **A**[*i*]=$\mathbf{T}_j$, lexicographically
- Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not
- If so, we store $\mathbf{T}_{j\text{-}1}$ at the leftmost empty position **LE**[$t_{j\text{-}1}$] of $t_{j\text{-}1}$-interval

$t_{j\text{-}1}$: Starting character of $\mathbf{T}_{j\text{-}1}$

Sorting L-suffixes from sorted LMS-suffixes

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

LE
a
b

A	$\mathbf{T}_{\mathbf{SA}[i]}$
$	$
a$	a$
	aabba$
aabba$	abba$
ba$	ba$
baabba$	baabba$
bba$	bba$

With a left-to-right scan on **A**

- Read a suffix $\mathbf{A}[i]=\mathbf{T}_j$, lexicographically
- Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not
- If so, we store $\mathbf{T}_{j\text{-}1}$ at the leftmost empty position $\mathbf{LE}[t_{j\text{-}1}]$ of $t_{j\text{-}1}$-interval

$t_{j\text{-}1}$: Starting character of $\mathbf{T}_{j\text{-}1}$

Correctness

	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

LE
a
b

A	$T_{SA[i]}$
$	$
a$	a$
	aabba$
aabba$	abba$
ba$	ba$
baabba$	baabba$
bba$	bba$

- We don't miss any L-suffixes
- We keep an invariant that suffixes in **A** are always sorted during the step

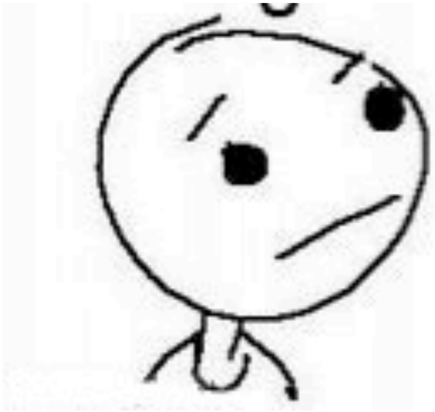
Induced sorting framework runs in $O(N)$ time and uses $\sigma + N / \log N$ extra words

- Problems
- Induced Sorting Framework
- Optimal Time and Space Algorithm
- Summary

Observations

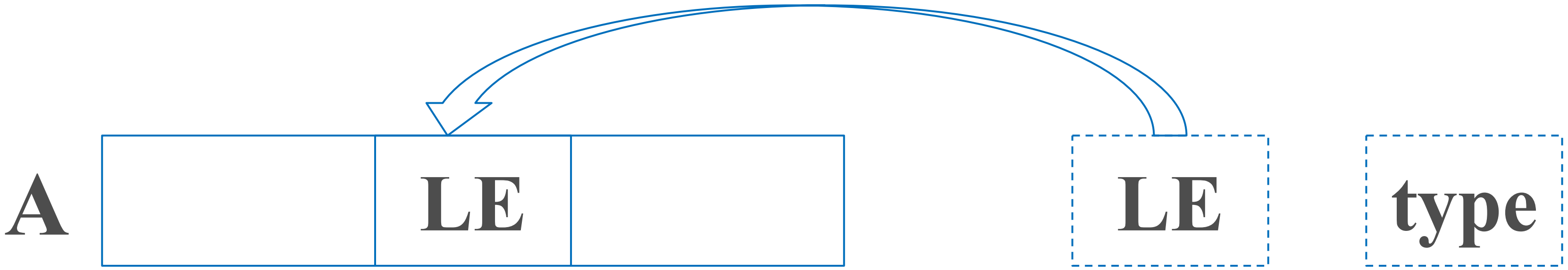
- Induced sorting framework
  - Good: run in $O(N)$ time
  - Bad: use $\sigma + N / \log N$ extra words for **LE** and **type**

I was thinking …

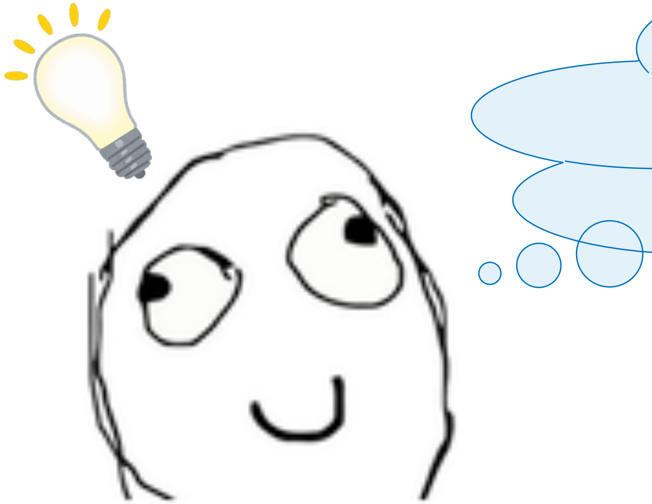


I'd like to remove **LE** and **type**,
but constructing SA without them
seems TOO difficult

Observations



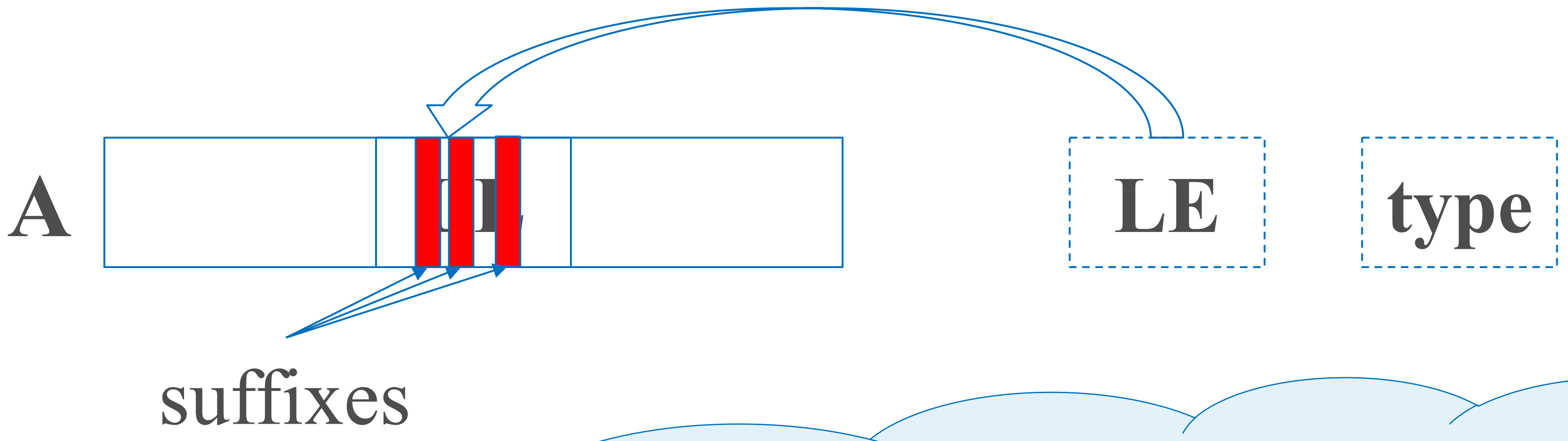
One day,
I came up with a good idea!

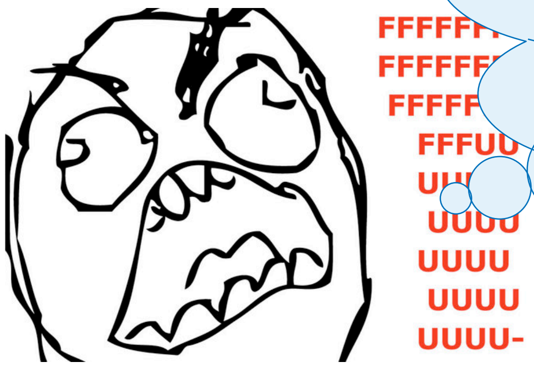


Use only **LE**, BUT we store it in **A**, so we require no extra space

Is it so easy? Of course not

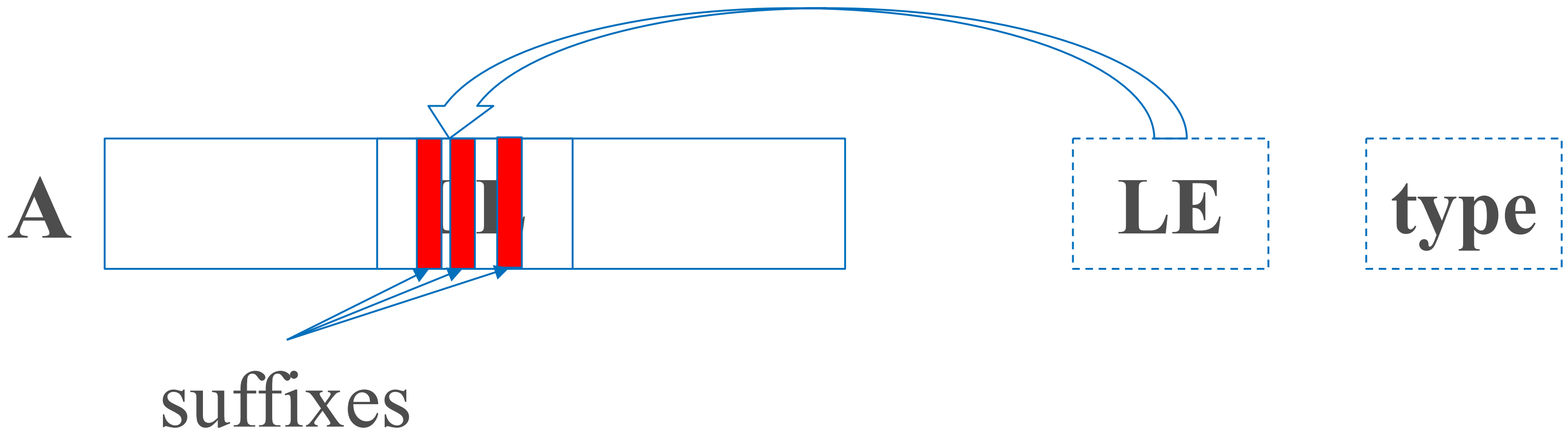
Observations





NOOO! some LE-values, which will be needed, are overwritten by induced suffixes

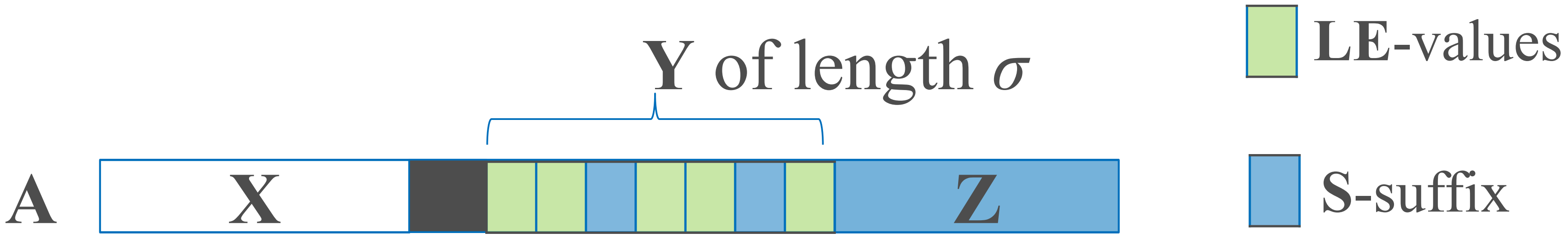
Observations



Our algorithm store **LE** in **A** and overwrite **LE**-values *only when* they will be no longer used. It runs in $O(N)$ time and uses $O(1)$ extra words space!

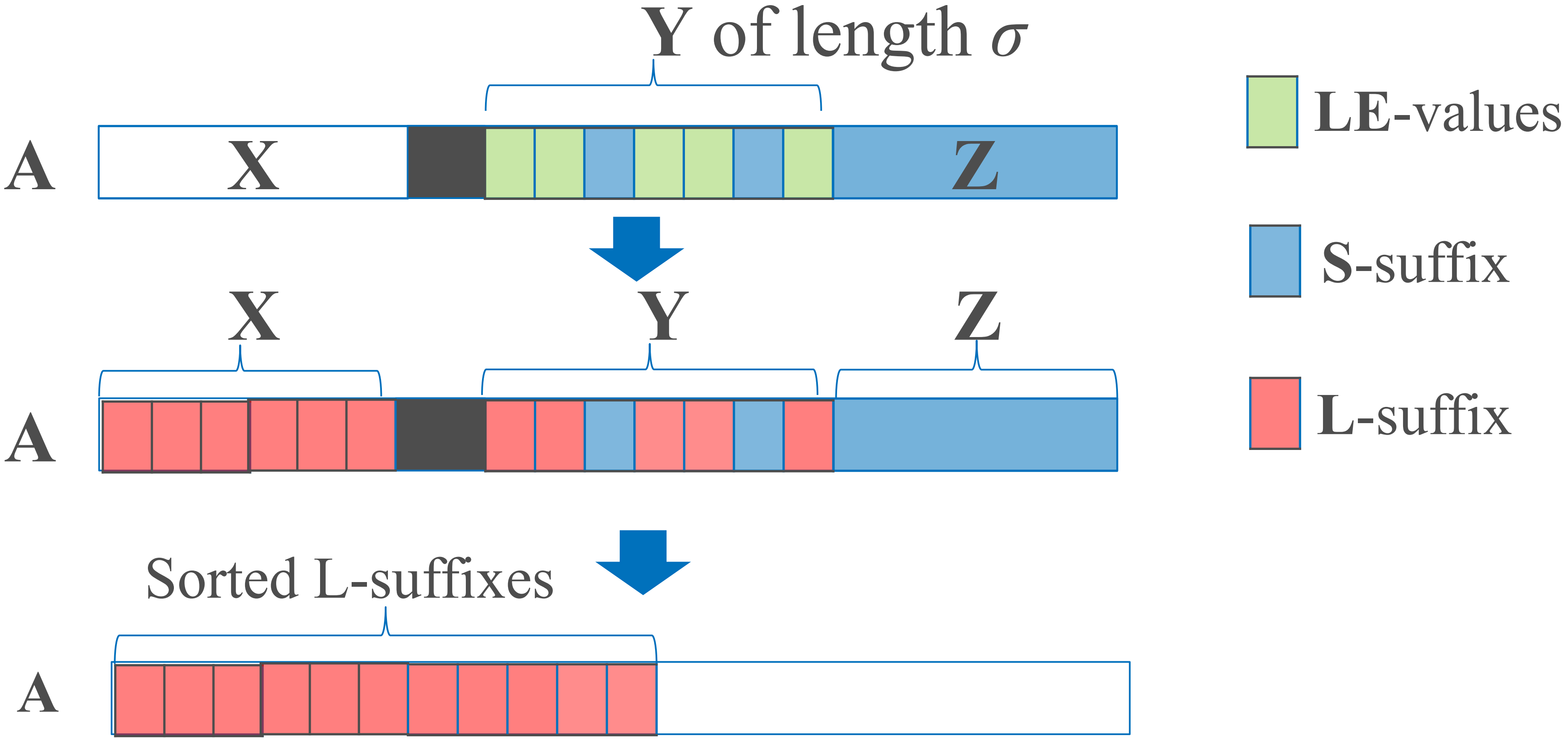
Overview of Our Algorithm

- We use three internal sub-arrays in **A**
- Preliminary, **Y** store **LE**-values and some LMS-suffixes
- **Z** stores the other LMS-suffixes

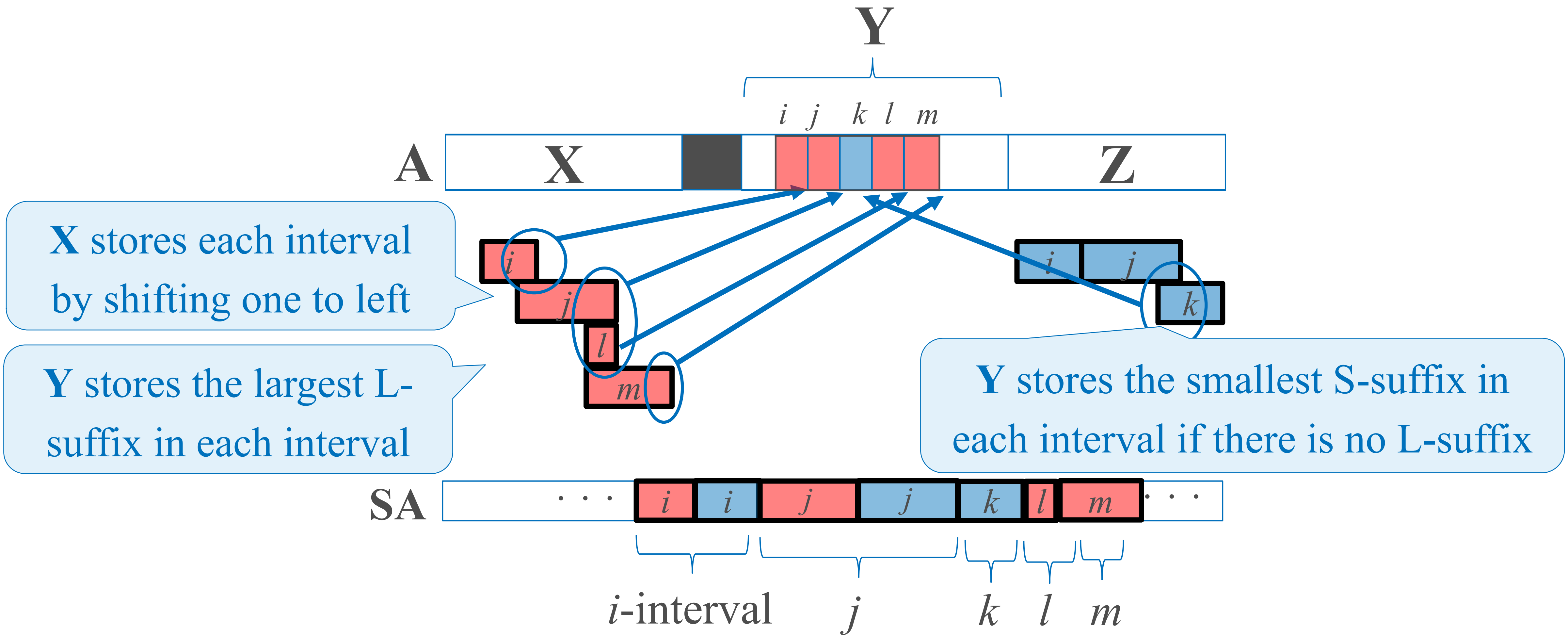


Overview of Our Algorithm

- Our goal is to store sorted L-suffixes separatory in **X** and **Y**
- Finally, we merge them

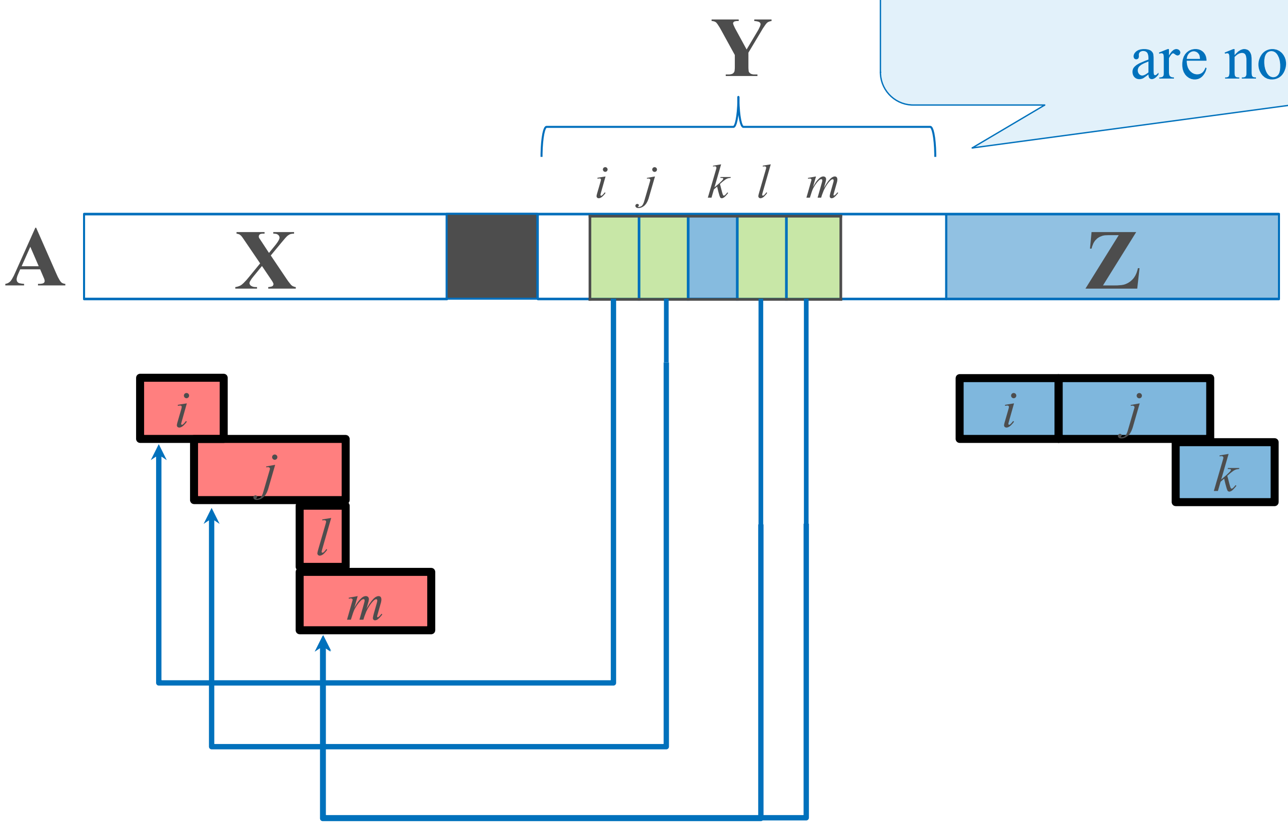


Detail Layout of Suffixes



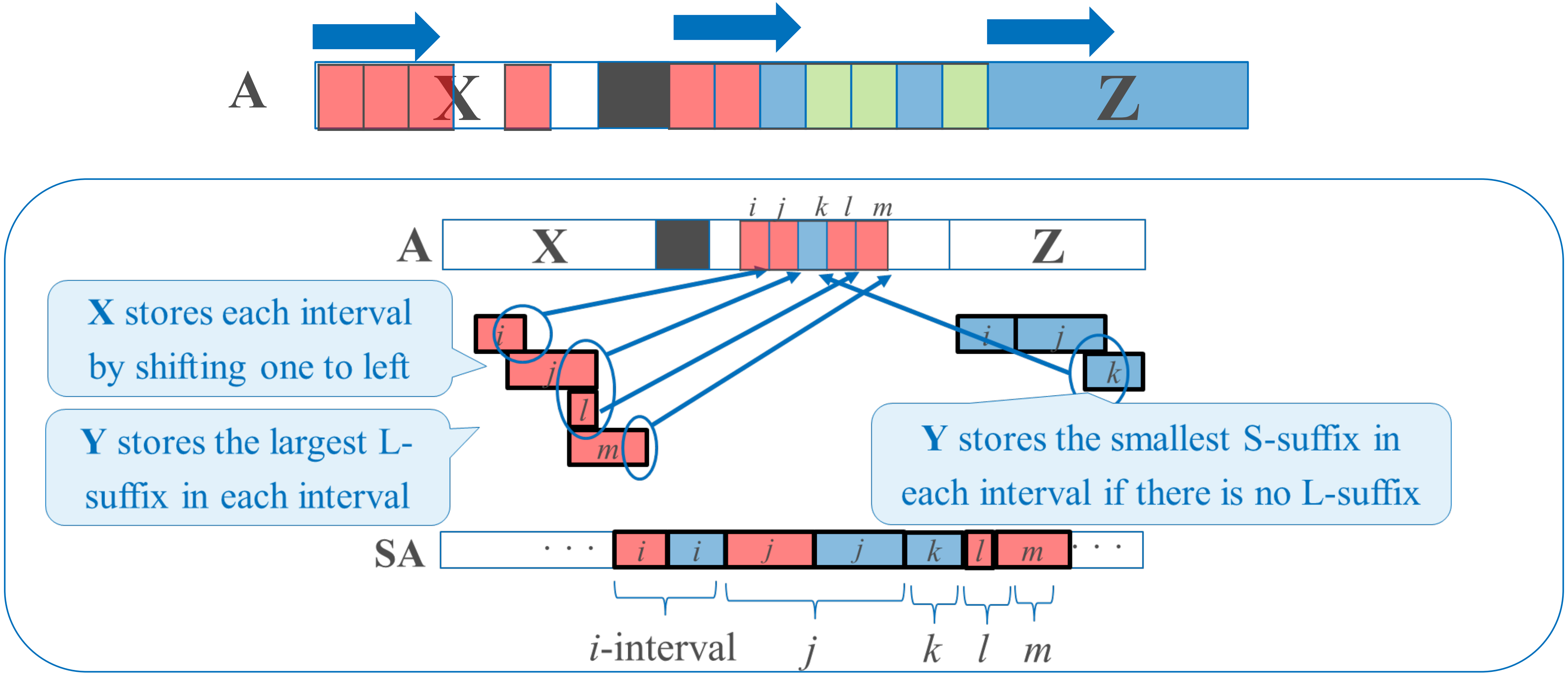
Initial State

LE-values, which will be needed, are not overwritten



Step 1: Lexicographically read L- and LMS-suffixes T_j

- Left-to-right scan on **X**, **Y**, and **Z**, respectively
- Compare their starting characters and choose the smallest one in priority over **X**, **Y**, and **Z**



Step2: Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not

Key Property [Nong et al., 2011]

For $\mathbf{T}_{j\text{-}1}$ and $\mathbf{T}_j$, if $t_{j\text{-}1} = t_j$, the type of $\mathbf{T}_{j\text{-}1}$ equals one of $\mathbf{T}_j$

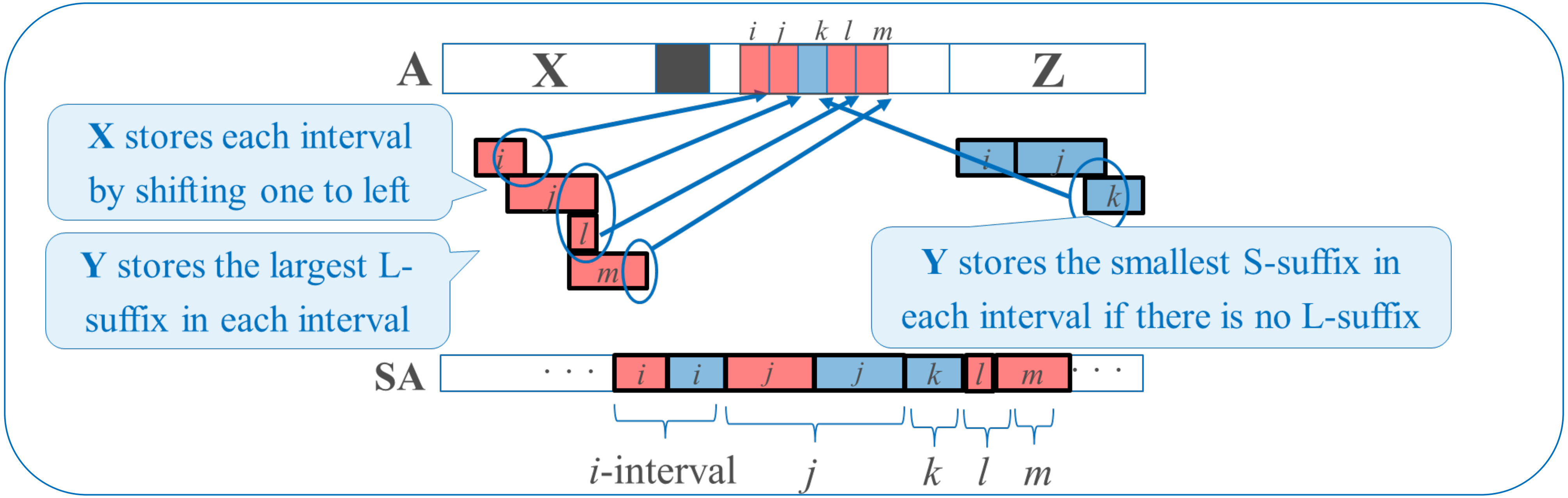
	1	2	3	4	5	6	7
T	b	a	a	b	b	a	$
type	L	S	S	L	L	L	S

Step2: Judge $\mathbf{T}_{j\text{-}1}$ is L-suffix or not

- $\mathbf{T}_{j\text{-}1}$ is L-suffix only if
  - $\mathbf{T}_j$ is read from **X** and $t_{j\text{-}1} \geqq t_j$
  - Or, $\mathbf{T}_j$ is read from **Z** and $t_{j\text{-}1} > t_j$
  - Or, $\mathbf{T}_j$ is read from **Y** and $t_{j\text{-}1} > t_j$

We know the type of $\mathbf{T}_j$

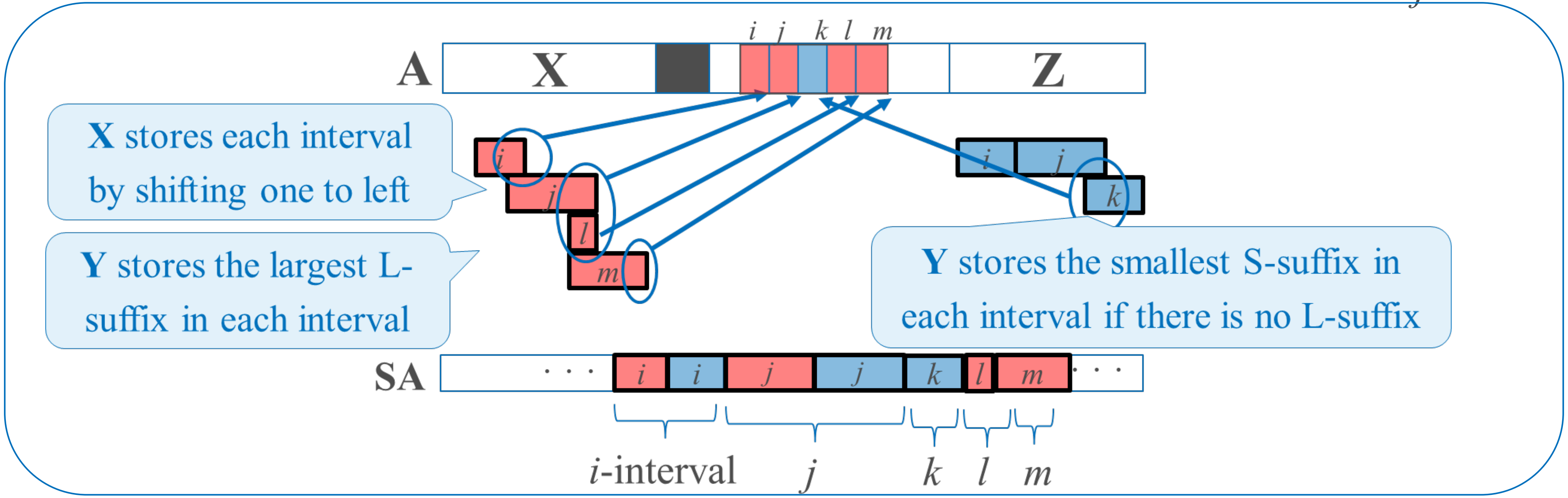
their starting characters must be different since $\mathbf{T}_j$ is the largest L-suffix or the smallest LMS-suffix



Step3: Store T_{j-1} If It is L-suffix

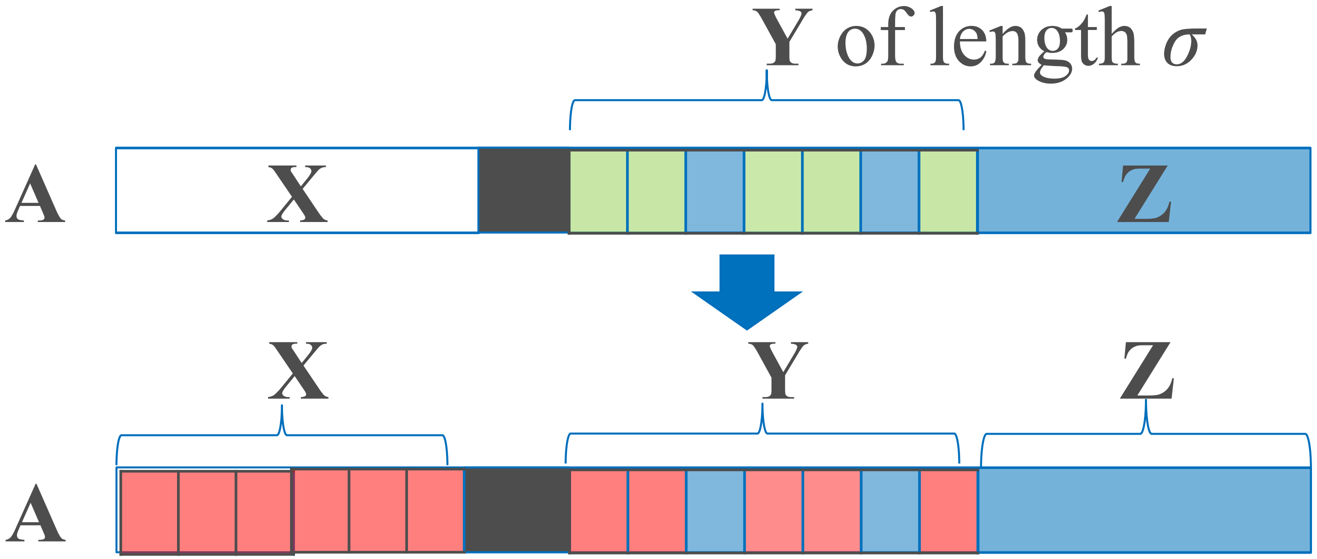
- We try to store $\mathbf{T}_{j-1}$ in $\mathbf{X}[\mathbf{LE}[t_{j-1}]]$
  - If $\mathbf{X}[\mathbf{LE}[t_{j-1}]]$ is EMPTY, then we store $\mathbf{T}_{j-1}$ in $\mathbf{X}[\mathbf{LE}[t_{j-1}]]$
  - otherwise, $\mathbf{X}[\mathbf{LE}[t_{j-1}]]$ has a suffix then we compare their starting characters, and store the smallest one in **Y** and store the other in $\mathbf{X}[\mathbf{LE}[t_{j-1}]]$

LE-value for the smallest one is no longer used since it is the largest one in its interval



Correctness

- Our algorithm simulates induced sorting framework without errors



Our algorithm runs in $O(N)$ time and uses $O(1)$ extra words space

Summary

- Proposed an algorithm for constructing **SA** in optimal time and space
- Proposed an algorithm for constructing both **SA** and **LCP** in optimal time and space (see our paper)

Future work?

- Using some techniques or observations in this work, develop practical implementations