This is a slide for introducing my work to my ex-supervisors, which was made on August 15, 2016. This slide implies that core idea in my paper had been developed at this time.
Note that green dialogs are added for notes.
Note that this slide contains English and also Japanese.

FUJITSU
shaping tomorrow with you

In-Place Linear Time Suffix Array Construction

2016/09/15

August 15, 2016

後藤啓介

Keisuke Goto

INTERNAL USE ONLY

研究の動機

Only this slide is converted to an image for masking

FUJITSU

- [redacted]

  A topic which is not related to the research

- 6月[redacted]とBBQへ
  - [redacted]「いまゴトーさんのLZ77をもっと省領域に計算する方法を研究してまーす」
  - 後藤「へー（いやー、あれはSAのアルゴリズムを元にしてるし、まずはそっちの問題を解決しなきゃだし、SAのin-place計算なんて無理そうだし、、、あれもしかして解ける？）」
  - 後々考えると、[redacted]が言っていたのは線形時間じゃないアルゴリズムだった気がする

This is explaining why I started this research in Japanese

Suffix Array

FUJITSU

For a string **T** of length N, the Suffix Array **SA** is an array that stores all suffixes of **T** in lexicographic order.
SA[i] = j indicate the suffix $\mathbf{T}_j$ starting at j is the i-th lexicographically smallest suffix.

T=

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

i	**SA**[i]	$\mathbf{T}_{\mathbf{SA}[i]}$
1	7	$
2	6	a$
3	4	ana$
4	2	anana$
5	1	banana$
6	5	na$
7	3	nana$

Suffix $\mathbf{T}_2$ is 4-th smallest suffix,
and it is stored at **SA**[4]

Issue of Suffix Array

FUJITSU

- Though **SA** only requires $N \log N$ bits, most linear time construction algorithms require the double space of **SA**

Theorem [Nong, 2013]

The suffix array of a string of length N can be computed in linear time and $\sigma \log N + O(\log N)$ bits of working space, where σ is the number of distinct characters in the string

The algorithm requires close to constant space when σ is small, but it still requires close to the double space of **SA** when σ is large

関連研究

- 昨日軽く調べたら色々見つかりました(汗
  - [Rahmann and Smula, ？？？]バイオ系の人？出典分からず
    - Simple In-Place Suffix Array Construction by Walking along Burrows-Wheeler Transforms
    - BWTがあればSAとLCPを線形時間in-placeに計算できるよ(たぶんgeneral alphabet)
  - [Franceschini1 and Muthukrishnan, 2007]
    - General alphabetについてO(N log N)時間でin-placeにSAを計算できるよ
  - [Nong, 2013]
    - Constant alphabetについて線形時間でin-placeにSAを計算できるよ
  - [Chrochemore+, 2013]
    - O(n^2)時間でin-placeにBWTが計算できるよ

This is captured from someone's slide

Computing BWT

	time	space (bits)
Suffix array construction		
many algorithms	$\mathcal{O}(n)$	$> n \log n + n \log \sigma$
Franceschini & Muthu, 2007	$\mathcal{O}(n \log n)$	$n \log n + n \log \sigma$ $+\mathcal{O}(\log n)$
Direct BWT construction		
K, 2007	$\mathcal{O}(n/\epsilon^2)$	$2n \log \sigma + \mathcal{O}(\epsilon n \log n)$
Okanohara & Sadakane, 2009	$\mathcal{O}(n)$	$\mathcal{O}(n \log \sigma \log \log_\sigma n)$
Succinct index construction		
Hon & al., 2007	$\mathcal{O}(n \log n)$	$n \log \sigma + \mathcal{O}(nH_0)$
Hon & al., 2009	$\mathcal{O}(n \log \log \sigma)$	$\mathcal{O}(n \log \sigma)$
In-place BWT construction		
This talk	$\mathcal{O}(n^2)$	$n \log \sigma + \mathcal{O}(\log n)$

text on input
BWT on output

$\mathcal{O}(1)$
words

Our Contribution

- We propose the first algorithm to construct the suffix array in linear time and in-place
  - Our algorithm is based on Nong’s algorithm
  - Our strategy is first construct some suffix arrays for subset of suffixes, and then merge them by any stable in-place merge algorithm for two sorted arrays

Preliminaries of Nong's Algorithm

FUJITSU

- We call a suffix $\mathbf{T}_i$ is small type suffix (S-suffix) if $i = N$ or $\mathbf{T}_i < \mathbf{T}_{i+1}$, and large type suffix (L-suffix) otherwise
- We call $\mathbf{T}_i$ also left most S-suffix (LMS-suffix) if $\mathbf{T}_i$ is S-suffix and i=1 or $\mathbf{T}_{i-1}$ is L-suffix, and similarly call $\mathbf{T}_i$ left most L-suffix (LML-suffix) if $\mathbf{T}_i$ is L-suffix

T=

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

T[N]=$ is the most smallest character and does not appear in **T**[1..N-1]

Property

- If $\mathbf{T}[i] \neq \mathbf{T}[i+1]$, the type of $\mathbf{T}_i$ equals to $\mathbf{T}_{i+1}$
- By a right-to-left scan on **T**, each type of **T**[i] can be obtained

Preliminaries of Nong's Algorithm

FUJITSU

- We call a suffix $\mathbf{T}_i$ is small type suffix (S-suffix) if $i = N$ or $\mathbf{T}_i < \mathbf{T}_{i+1}$, and large type suffix (L-suffix) otherwise
- We call $\mathbf{T}_i$ also left most S-suffix (LMS-suffix) if $\mathbf{T}_i$ is S-suffix and i=1 or $\mathbf{T}_{i\text{-}1}$ is L-suffix, and similarly call $\mathbf{T}_i$ left most L-suffix (LML-suffix) if $\mathbf{T}_i$ is L-suffix

For a character *c*, we call the interval of suffixes starting with *c* as *c*-interval

T=

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

-interval

a-interval

b-interval

n-interval

i	SA[*i*]	$\mathbf{T}_{\mathbf{SA}[i]}$
1	7	$
2	6	a$
3	4	ana$
4	2	anana$
5	1	banana$
6	5	na$
7	3	nana$

L-suffixes must be smaller than S-suffixes in each interval

Overview of Nong's Algorithm

- Sort LMS-suffixes
  - Sort LMS-substring
  - If they are not unique
    - Make **T**' from **T** by replacing all LMS-substring with their ranks
    - Constructs **SA**' of **T**' recursively
- Sort L-suffixes
- Sort S-suffixes

Combination of sorting L, S-suffixes

Run in-place

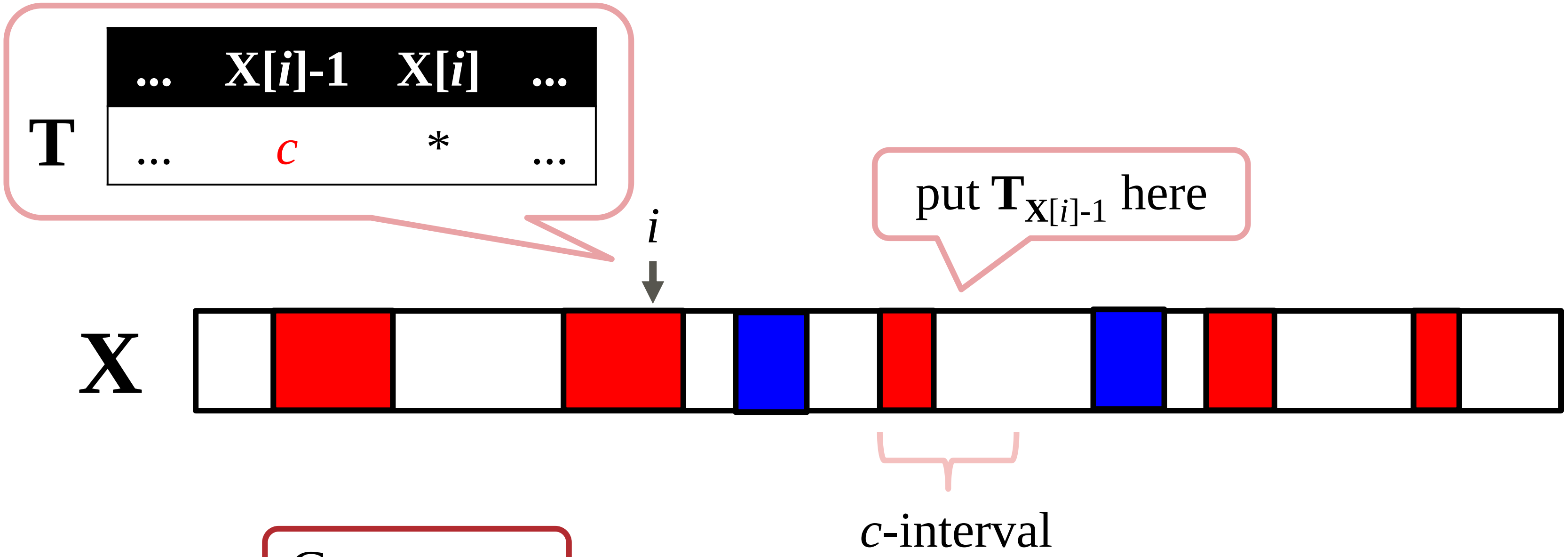
Use σ log N + O(log N) bits of space

Almost same of sorting L-suffixes

Sorting L-suffixes is the core of algorithm, and also the most space bottleneck part

Sketch of Sorting L-suffixes

- Initially sorted LMS-suffixes are stored in the tail of each *c*-interval
- By a left-to-right scan on **X**,
  - If **X**[*i*] is not EMPTY and $\mathbf{T}_{\mathbf{X}[i]-1}$ starting with *c* is L-suffix, put $\mathbf{T}_{\mathbf{X}[i]-1}$ to the left-most-empty position of *c*-interval on **SA**



Correctness

For a L-suffix $\mathbf{T}_j$ stored before $\mathbf{T}_{\mathbf{x}[i]-1}$, $\mathbf{T}_j < \mathbf{T}_{\mathbf{x}[i]-1}$ since $\mathbf{T}_j$ must be put before accessing *i*, and so $\mathbf{T}_{j+1} < \mathbf{T}_{\mathbf{x}[i]}$

Space Usage of Sorting L-suffixes

FUJITSU

- Initially sorted LMS-suffixes are stored in the tail of each *c*-interval
- By a left-to-right scan on **X**,
  - If **X**[*i*] is not EMPTY and $\mathbf{T}_{\mathbf{X}[i]-1}$ starting with *c* is L-suffix, put $\mathbf{T}_{\mathbf{X}[i]-1}$ to the left-most-empty position of *c*-interval on **SA**

Judge by **bkt**[*c*]

Mange by **bkt**[*c*] in $\sigma \log N$ bits

Our algorithm conceptually run in the same way,
but we store $\mathbf{SA}_{s(L)}$ in continuous space of **X**,
and use the remained continuous empty space in **X** as **bkt**

Notation

Notation

- s(L): set of all L-suffixes
- s(S): set of all S-suffixes
- $\mathbf{SA}_M$: suffix array of any set of suffixes M

We firstly explain in the case $\sigma \leqq N/2$ and |s(L)| $\leqq$ |s(S)|, and other case later

mean also |s(L)| $\leqq N/2$
since |s(L)| + |s(S)|=N

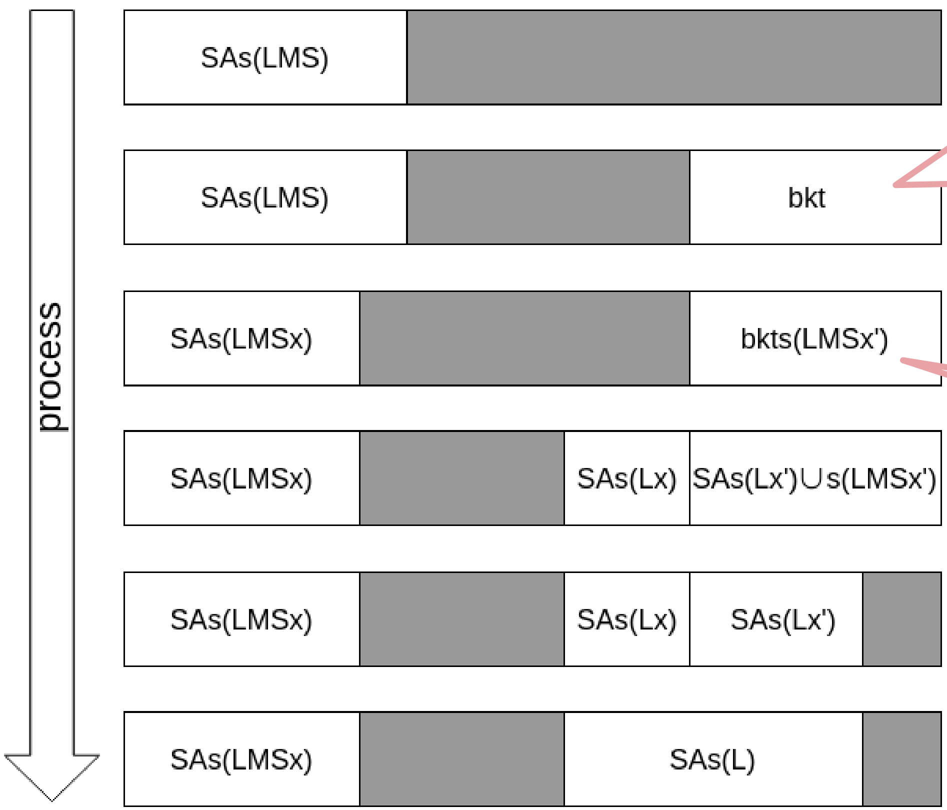
Sketch of Our Algorithm

FUJITSU

- s(Lx’): the set of largest suffixes starting with *c* for all characters *c*
  s(Lx)=s(L)-s(Lx’)
- s(LMSx’): the set of smallest suffixes starting with *c* such that there are no L-suffixes starting with *c*
  s(LMSx)=s(LMS)-s(LMSx’)

Note that |s(Lx’)| + |s(LMSx’)|=σ

サイズと状態だけに注目



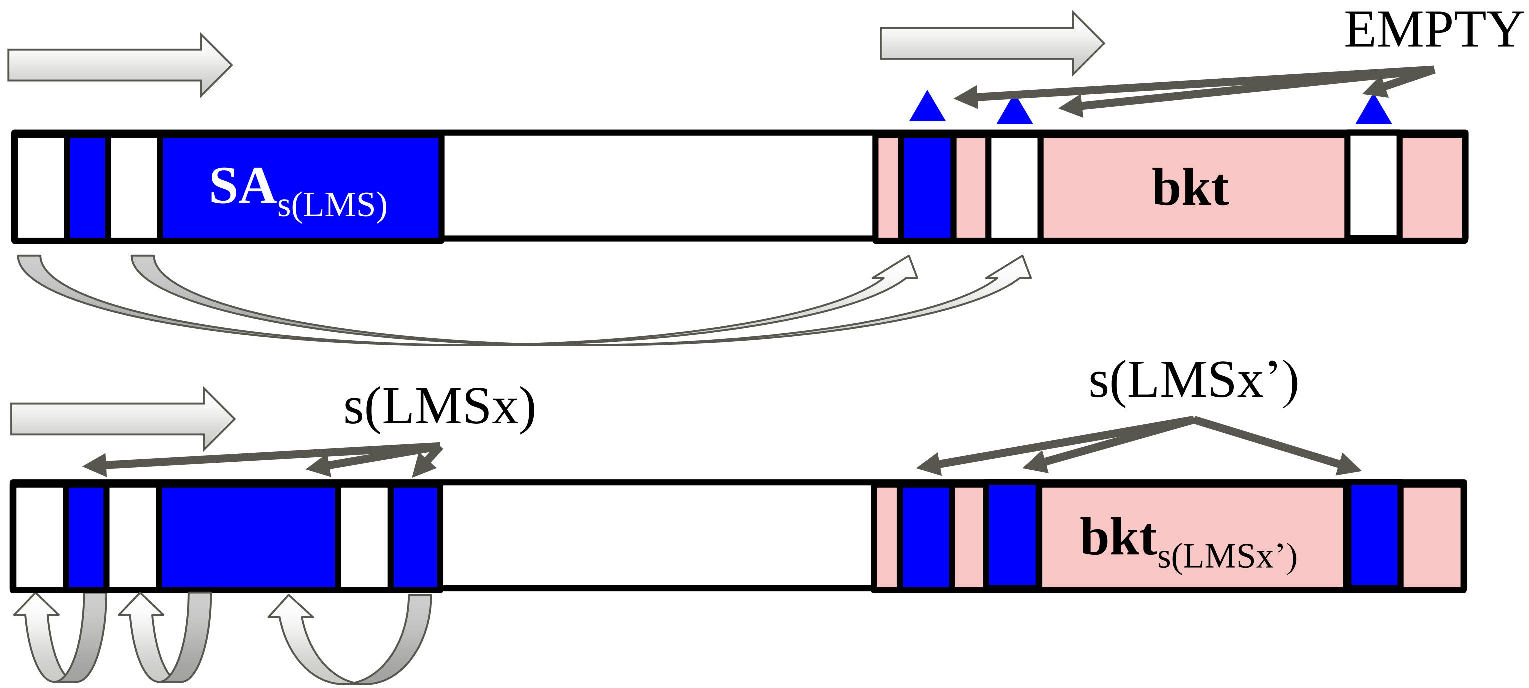
bkt[*c*]: left most empty position of *c*-interval in $\mathbf{SA}_{s(Lx)}$ if the corresponding L-suffix in s(LMSx’) exists, EMPTY otherwise

Note that |**SA**s(LMS)| + |**bkt**| ≦ *N* since |s(LMS)| ≦ *N*/2 and σ ≦ *N*/2

$\mathbf{bkt}_{s(LMSx')}[c]$: **bkt**[*c*] if **bkt**[*c*] ≠ EMPTY, a suffix of s(LMSx’) starting with *c* otherwise

How to Transit In-Place?

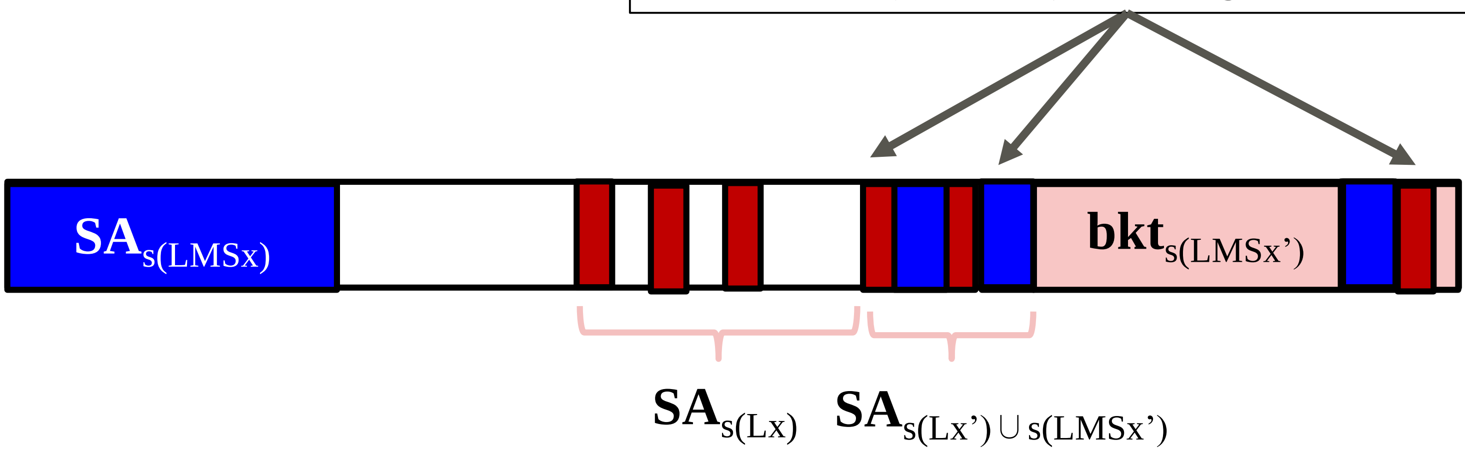
- **bkt**: Easy, we omit
- **bkt**$_{\mathrm{s(LMSx')}}$ and **SA**$_{\mathrm{s(LMSx)}}$:
  - By a left-to-right scan on **SA**$_{\mathrm{s(LMS)}}$, move the smallest LMS-suffix starting with *c* to **bkt**[*c*] if **bkt**[*c*] is EMPTY
  - By a left-to-right scan on **SA**$_{\mathrm{s(LMS)}}$, shift and make **SA**$_{\mathrm{s(LMSx)}}$



How to Transit In-Place?

- $\mathbf{SA}_{s(Lx)}$ and $\mathbf{SA}_{s(Lx)\cup s(LMSx')}$:

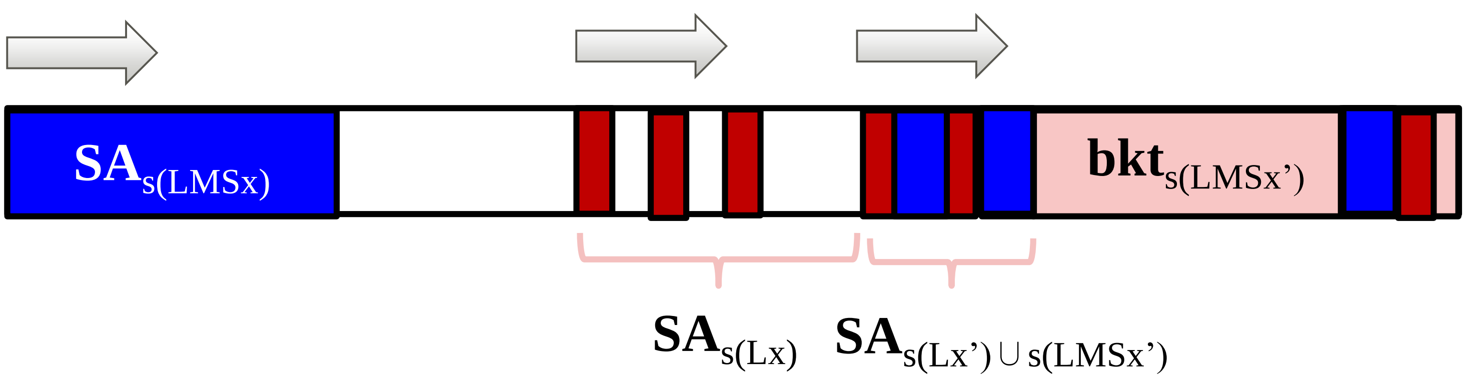
After storing all L-suffixes starting with *c*, **bkt**[*c*] never be used in future, we store the suffix of s(LMSx') starting with c at **bkt**[*c*]



How to Transit In-Place?

$\mathbf{SA}_{s(Lx)}$ and $\mathbf{SA}_{s(Lx)\cup s(LMSx')}$:

- By a left-to-right scan on $\mathbf{SA}_{s(Lx)}$, $\mathbf{SA}_{s(Lx')\cup s(LMSx')}$, $\mathbf{SA}_{s(LMSx)}$, access all LMS, L-suffixes $\mathbf{T}_i$ starting with c lexicographically
  - Judge whether $\mathbf{T}_{i\text{-}1}$ is L-suffix, if not do nothing
  - We try to put L-suffix $\mathbf{T}_{i\text{-}1}$ starting with c to $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$

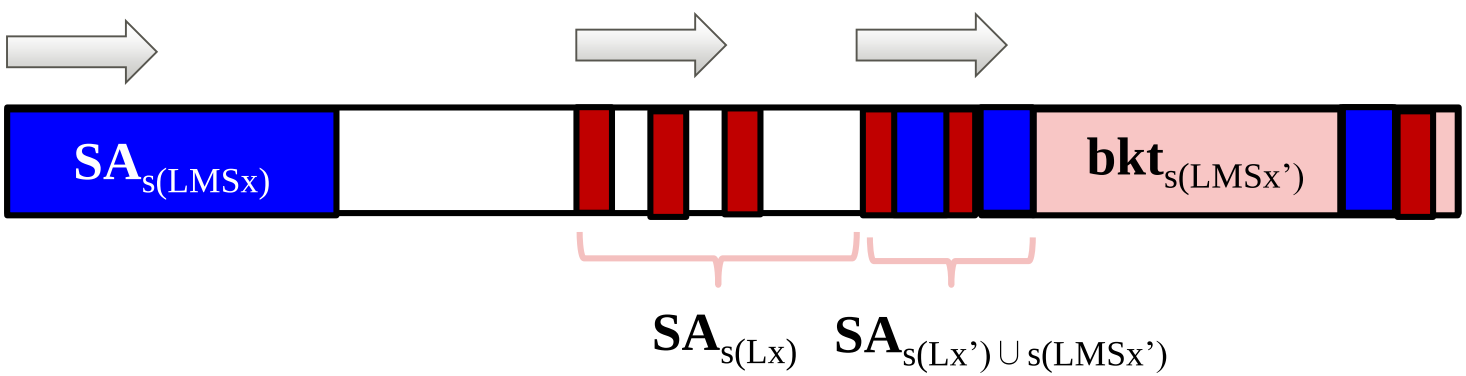


How to Transit In-Place?

$\mathbf{SA}_{s(Lx)}$ and $\mathbf{SA}_{s(Lx)\cup s(LMSx')}$:

Judge whether $\mathbf{T}_{i\text{-}1}$ is L-suffix, if not do nothing

- If $\mathbf{T}_i$ is of $\mathbf{SA}_{s(Lx)}$ or $\mathbf{SA}_{s(LMSx)}$, we can determine the type of $\mathbf{T}_{i\text{-}1}$ by comparing heading characters of **c** of $\mathbf{T}_i$
- If $\mathbf{T}_i$ is of $\mathbf{SA}_{s(Lx')\cup s(LMSx')}$, the heading characters of $\mathbf{T}_{i\text{-}1}$ of $\mathbf{T}_i$ must be different, so we can determine the type of $\mathbf{T}_{i\text{-}1}$ by comparing heading characters



How to Transit In-Place?

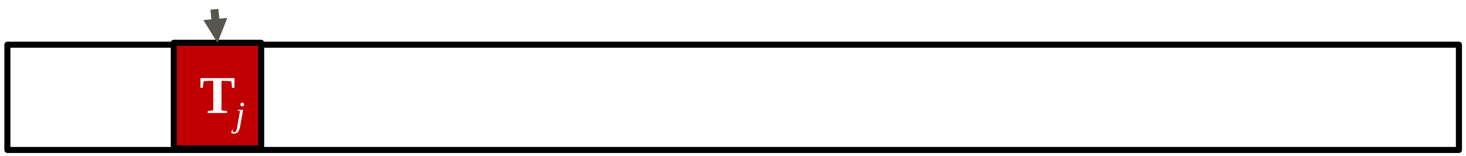
FUJITSU

$\mathbf{SA}_{s(Lx)}$ and $\mathbf{SA}_{s(Lx) \cup s(LMSx')}$:

We try to put L-suffix $\mathbf{T}_{i\text{-}1}$ starting with c to $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$

- If $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$ is EMPTY, we put $\mathbf{T}_{i\text{-}1}$ there
- Otherwise

$\mathbf{SA}[\mathbf{bkt}[c]]$
cannot be stored $\mathbf{T}_{i\text{-}1}$



At least $\mathbf{T}_{i\text{-}1}$ or $\mathbf{T}_j$ (j=$\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$) is of s(Lx'), and heading characters of them must be different since $\mathbf{bkt}[c]$ must indicate different positions for each L-suffix starting with c

Conflict occurs between a largest suffix starting with $c1$ and a smallest suffix starting with $c2>c1$

How to Transit In-Place?

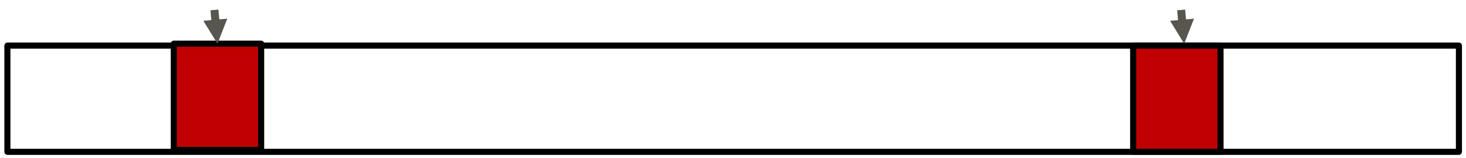
$\mathbf{SA}_{s(Lx)}$ and $\mathbf{SA}_{s(Lx)\cup s(LMSx')}$:

We try to put L-suffix $\mathbf{T}_{i\text{-}1}$ starting with c to $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$

- If $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$ is EMPTY, we put $\mathbf{T}_{\text{i-1}}$ there
- Otherwise, we put the smallest of $\mathbf{T}(\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]])$ and $\mathbf{T}_{\text{i}}$ to $\mathbf{SA}_{s(Lx')\cup s(LMSx')}[c']$ and the other into $\mathbf{SA}_{s(Lx)}[\mathbf{bkt}[c]]$, where c' is the starting character of smallest one

$\mathbf{SA}[\mathbf{bkt}[c]]$
store the largest one

$\mathbf{SA}_{s(Lx')\cup s(LMSx')}[c']$
store the smallest one starting with c'

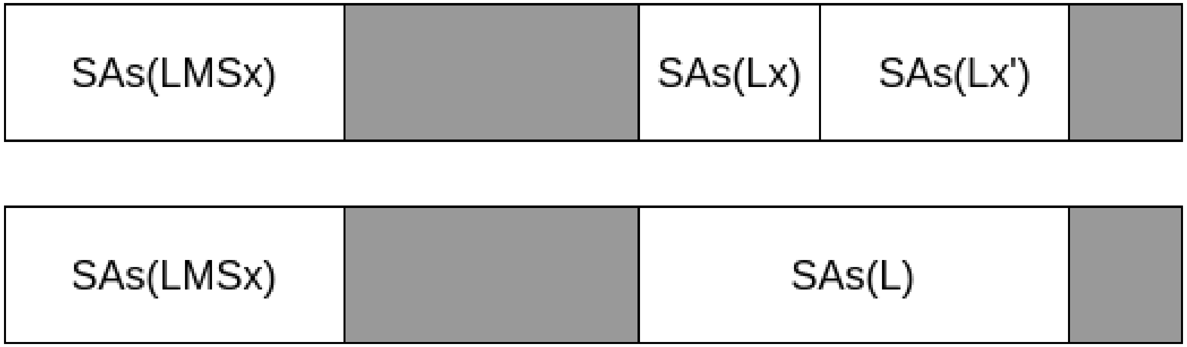


Merge Two Suffix Arrays

SA$_{s(L)}$:

> Observation: In each *c*-interval in **SA**$_{s(L)}$, Each L-suffix of **SA**$_{s(Lx')}$ must be stored at the right side of all L-suffixes of **SA**$_{s(Lx)}$

- We can make **SA**$_{s(L)}$ by a stable merge of **SA**$_{s(Lx)}$ and **SA**$_{s(Lx')}$ with **T**[**SA**$_{s(Lx)}$[i]] and **T**[**SA**$_{s(Lx')}$[*i*]] as keys



Merge Two Suffix Arrays

SA$_{s(L)}$:

> Observation: In each c-interval in **SA**$_{s(L)}$, Each L-suffix of **SA**$_{s(Lx')}$ must be stored at the right side of all L-suffixes of **SA**$_{s(Lx)}$

- We can make **SA**$_{s(L)}$ by a stable merge of **SA**$_{s(Lx)}$ and **SA**$_{s(Lx')}$ with **T**[**SA**$_{s(Lx)}$[i]] and **T**[**SA**$_{s(Lx')}$[i]] as keys

Theorem [Huang and Langston, 1998]

For two integer arrays $\mathbf{Y}_1$ of length N_1 and $\mathbf{Y}_2$ of length N_2, which are stored in $\mathbf{Y}$ of length $N_1 + N_2$ in this order, there is an algorithm to sort the values in $\mathbf{Y}$ stably in $O(N_1 + N_2)$ and in-place

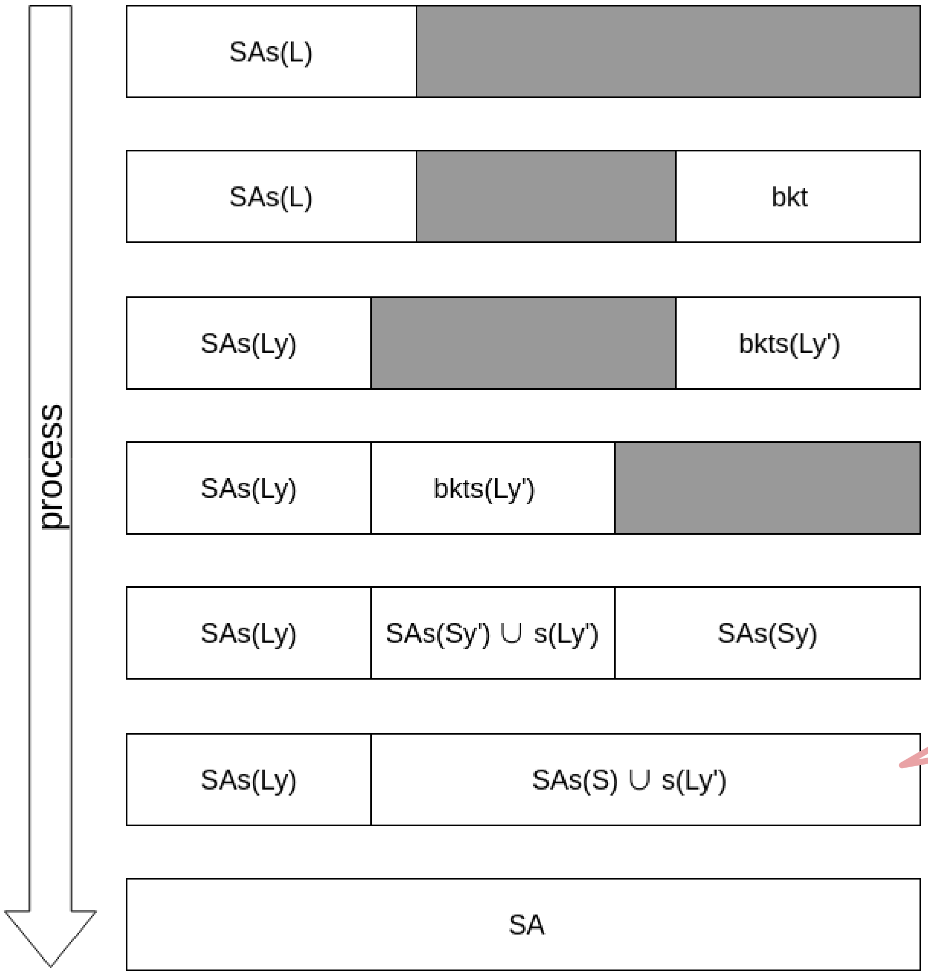
Lemma

There is an algorithm to compute **SA**$_{s(L)}$ linear time and in-place when $\sigma \leqq N/2$

Sorting S-suffixes

FUJITSU

- Almost same of sorting L-suffixes



Note that |**SA**s(L)| + |**bkt**| $\leqq$ N since we assumed |s(L)| $\leqq$ $N/2$ and $\sigma \leqq N/2$

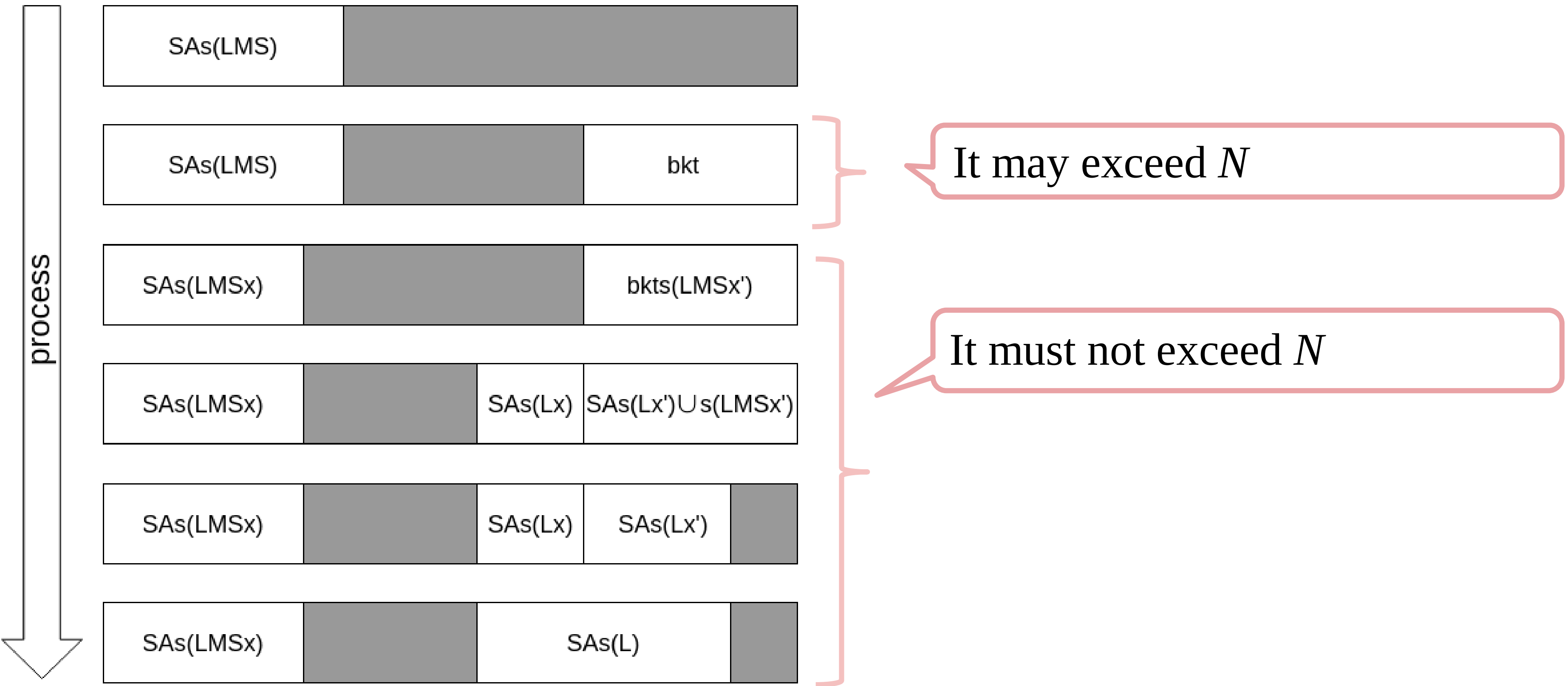
Make **SA** rather than $\mathbf{SA}_{s(S)}$

- If $\sigma > N/2$ or $|s(L)| > |s(S)|$, there may not be sufficient space to store $\mathbf{SA}_{s(LMS)}/\mathbf{SA}_{s(L)}$ and **bkt**
- Previous assumption:
  - $\sigma > N/2$ and $|s(L)| \leqq |s(S)|$
- We next consider the case $\sigma > N/2$ and $|s(L)| \leqq |s(S)|$

Observation

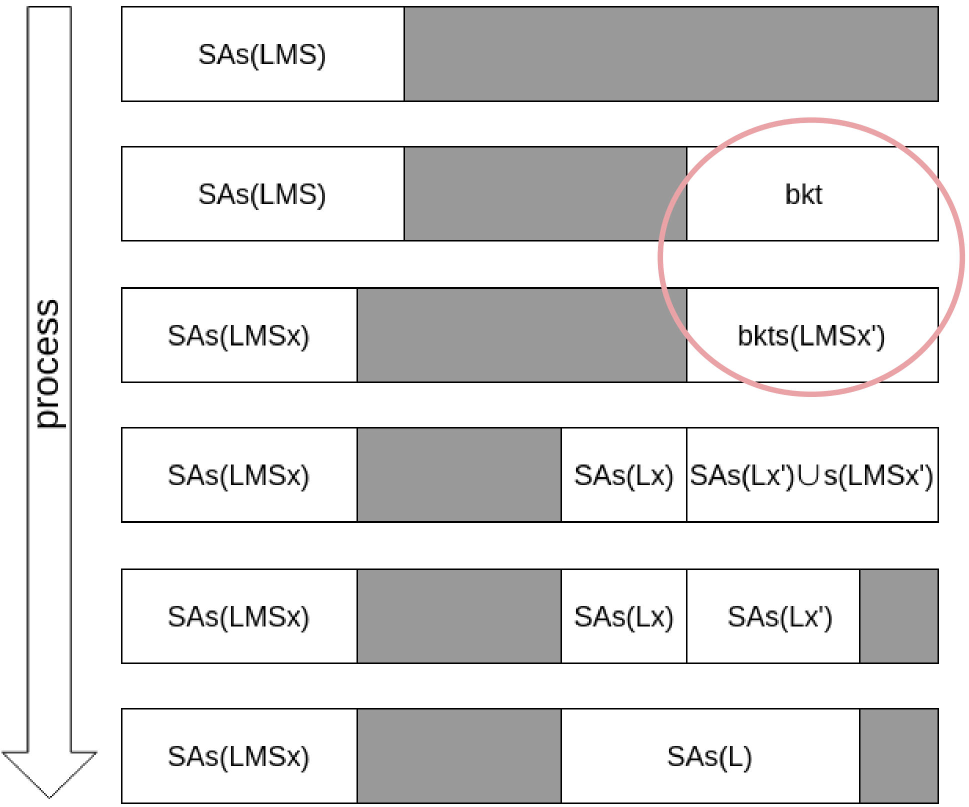
FUJITSU

- We can compute $\mathbf{SA}_{s(L)}$ in-place if we can compute $\mathbf{SA}_{s(LMSx)}$ and $\mathbf{bkt}_{s(LMSx')}$ in-place



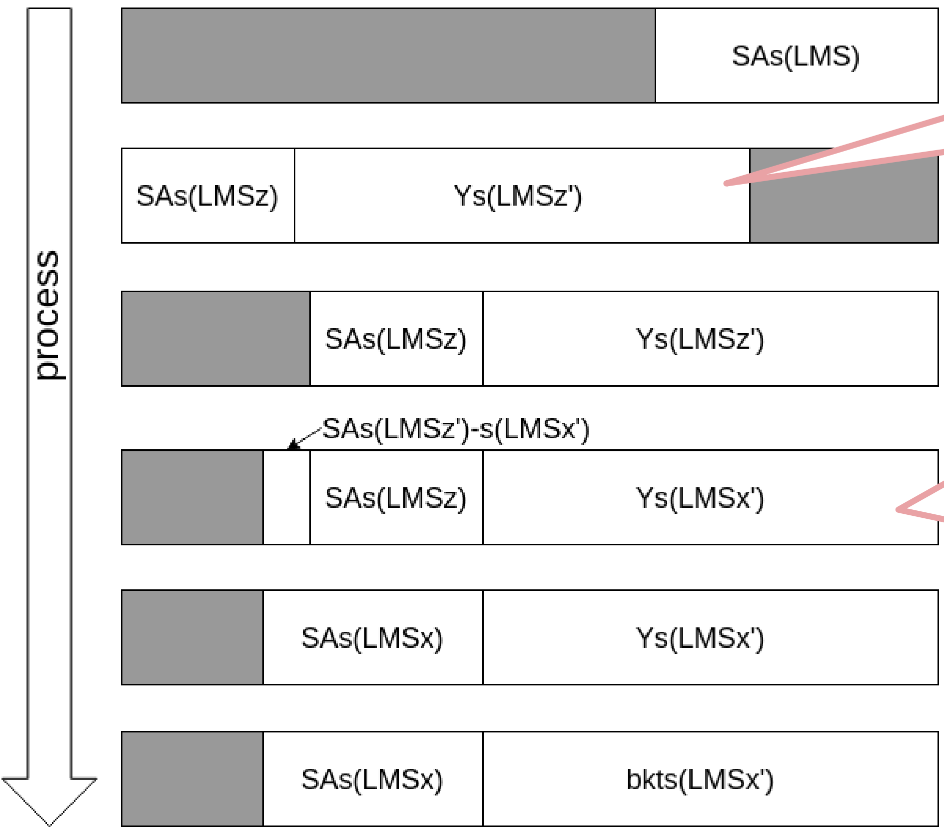
Observation

- We can compute $\mathbf{SA}_{s(L)}$ in-place if we can compute $\mathbf{SA}_{s(LMSx)}$ and $\mathbf{bkt}_{s(LMSx')}$ in-place



New algorithm splits s(LMSx) and s(LMSx') first, and computes **bkt** later

- s(LMSz’): the set of smallest suffixes starting with *c* for all characters *c*
  s(LMSz)=s(LMS)-s(LMSz’)



$\mathbf{Y}_{s(LMSz')}[c]$: a suffix of s(LMSz’) starting with *c*

Remember that s(LMSx’) is the set of smallest LMS-suffixes starting with *c* for all characters *c* such that there are no L-suffixes starting with *c*

Y[c]の空き領域は少なくともcから始まるL-suffixが1以上あることを示す。そのため*N*を超えない

Tool

I used this notation as the ceil function

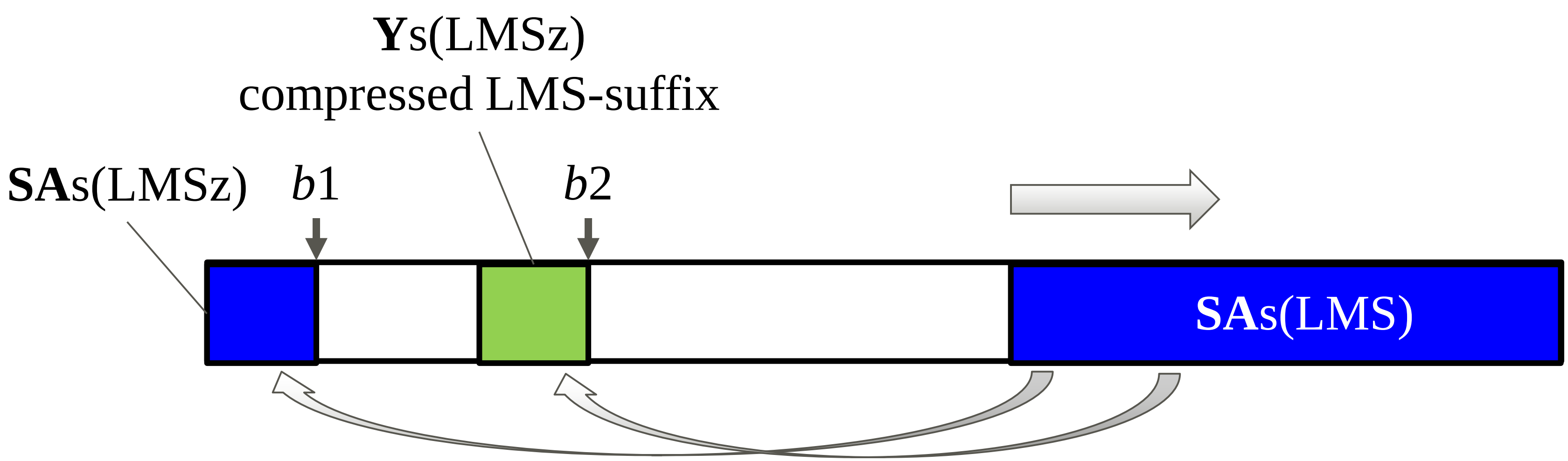
Lemma

If $\mathbf{T}[i] \neq \mathbf{T}[i\text{-}1]$, i can be represented by ⌈log N⌉ -1 bits, and back again if we know $\mathbf{T}[i]$ and $\mathbf{T}$

- Proof
  - Let j=⌈i/2⌉ and c=$\mathbf{T}[i]$
    Note that i is $2j$ or $2j$+1
  - If $\mathbf{T}[2j] \neq \mathbf{T}[2j+1]$, the position of either which equals to $\mathbf{T}[i]$ is i
  - Otherwise $i = 2j$ since if not, it contradicts the definition

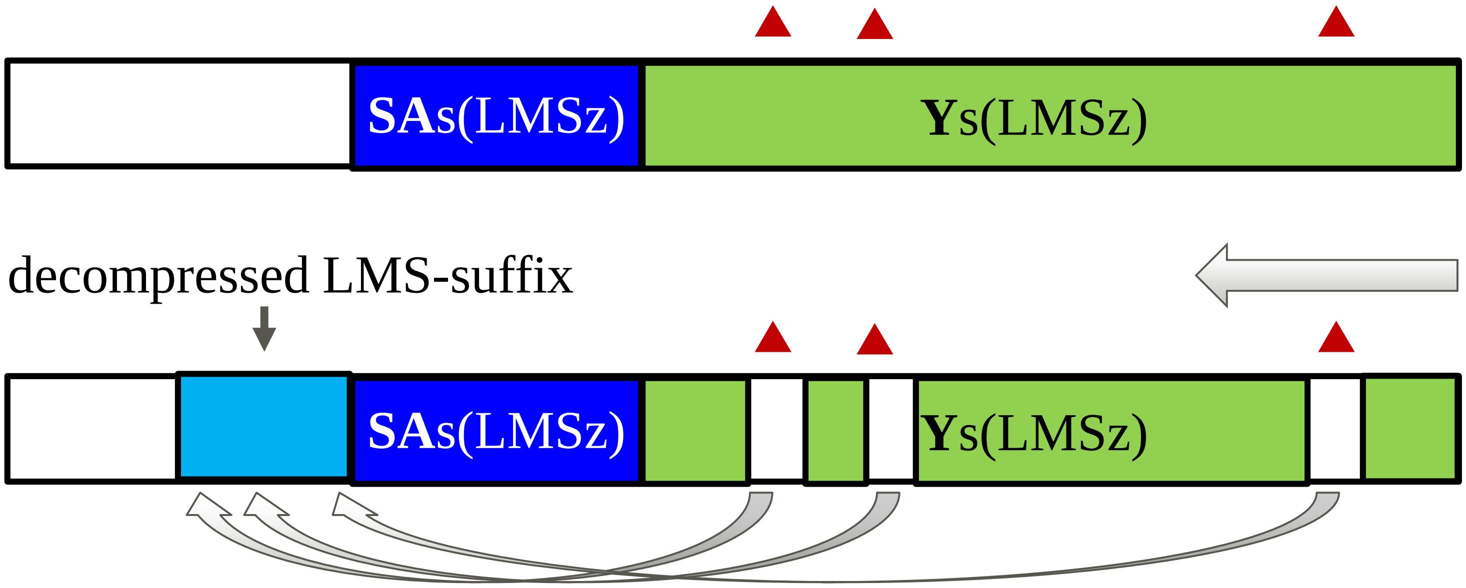
How to Transit In-Place?

- **SA**$_{s(LMSz)}$ and **Y**$_{s(LMSz')}$:
  - Let b1=0, b2=|s(LMSz)|
  - By a left-to-right scan on **SA**$_{s(LMS)}$, when accessing **SA**$_{s(LMS)}$ $[i]=j$
    - If $\mathbf{T}_j \in$ s(LMSz), move j to **X**[b1], and update b1+=1
    - otherwise, move ⌈ j/2 ⌈ to **X**[b2] and update b2+=1



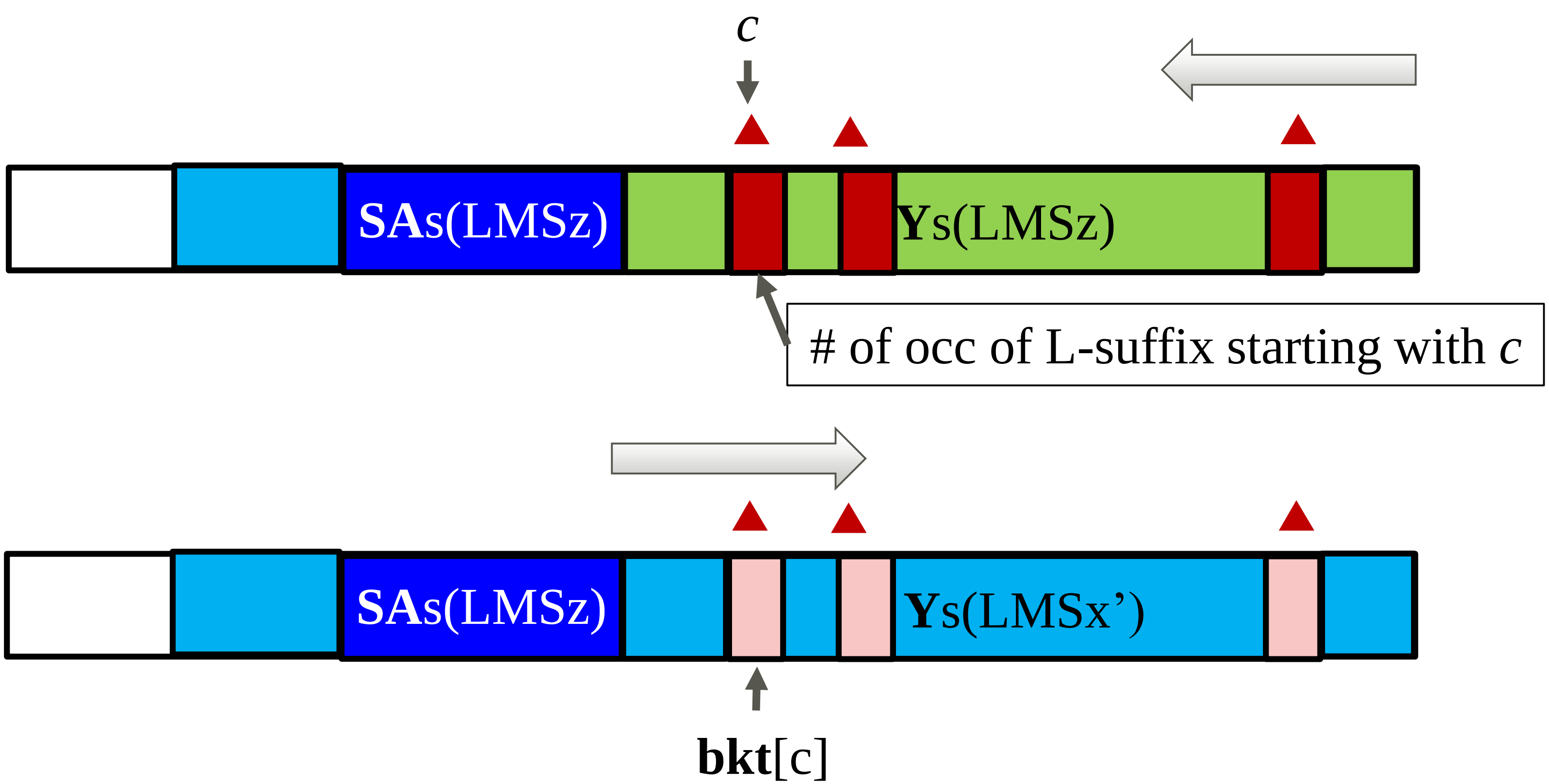
How to Transit In-Place?

- **SA**$_{s(LMSz')-s(LMSx)}$ and **Y**$_{s(LMSx)}$:
  - If a suffix starting with c is in s(LMSz')-s(LMSx), L-suffix starting with c exists
  - By a right-to-left scan on T,
    - We can determine Ys(LMSz')[T[i]]
  - We can move all suffixes of s(LMSz')-s(LMSx) but they are not sorted
  - We mark all all suffixes of s(LMSz')-s(LMSx) and move them later by a sequential scan



How to Transit In-Place?

- **bkt**$_{s(LMSx)}$:
  - By a right-to-left scan on **T**, count the number of L-suffixes starting with *c*, and store **Y**[*c*]
  - By a left-to-right scan on **Y**, If the highest bit is 1, accumulate value, and decompress the value otherwise



- If $|s(L)| > |s(S)|$,
  - Sort LML-suffixes
  - Sort S-suffixes
  - Sort L-suffixes

Summary

- Summary
  - We propose the first in-place suffix array construction algorithm
  - We also propose a algorithm not to use stack space for recursive steps

- Future Work
  - More simpler in-place algorithm
  - In-place linear time construction algorithm of Φ array
    - The most space efficient one is Goto and Bannai's algorithm which takes $\sigma \log N + O(\log N)$ bits of space
    - If it is possible, we can compute LZ77 in $N \log N$ bits + $O(\log N)$ bits of space. the most space efficient one takes $N \log N + \sigma \log N + O(\log N)$ bits of space

[Goto and Bannai, 2014]

shaping tomorrow with you