Space Efficient Linear Time Lempel-Ziv Factorization for Small Alphabet

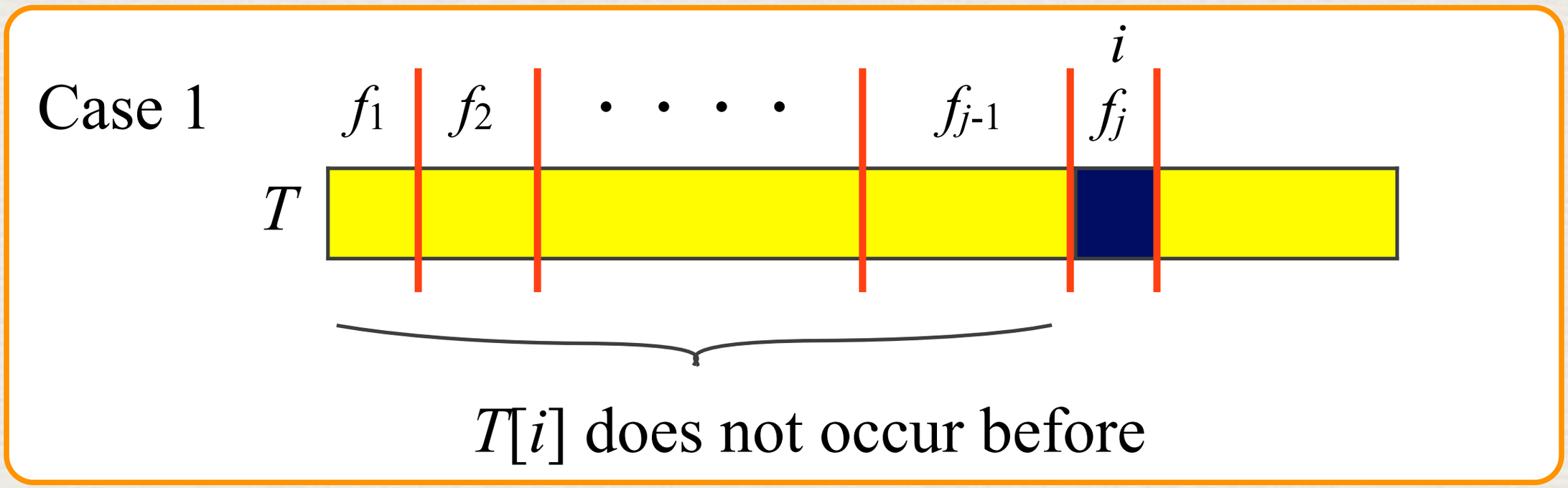
Kyushu University
Keisuke Goto, Hideo Bannai

LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies the following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before position i, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

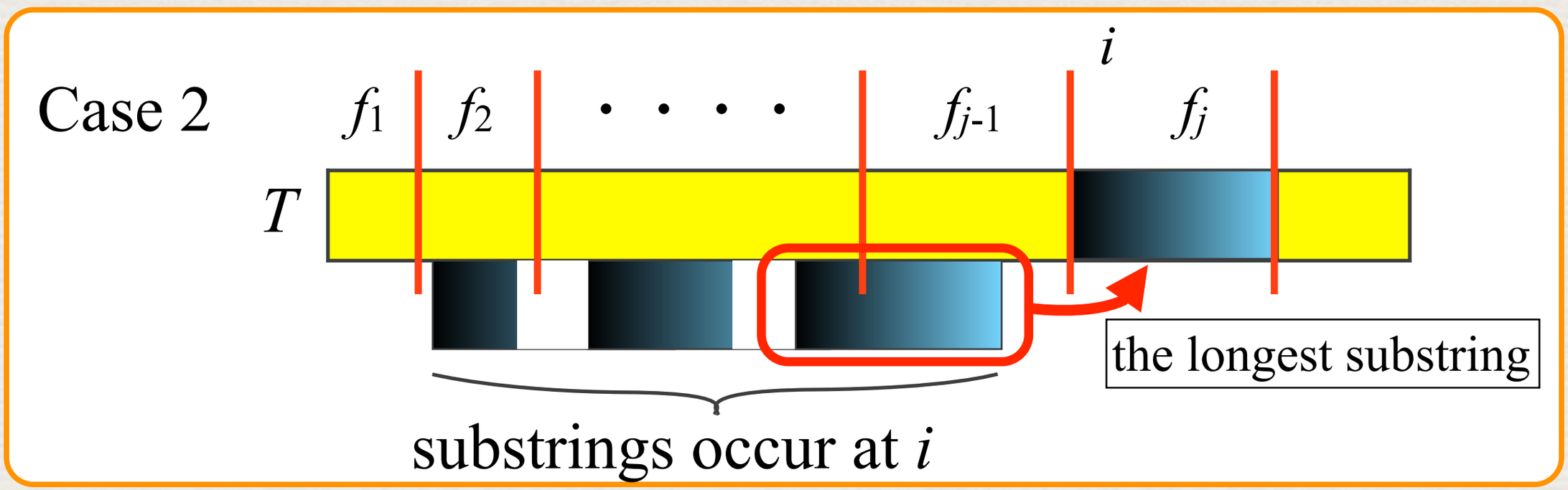


LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies the following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before position i, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i



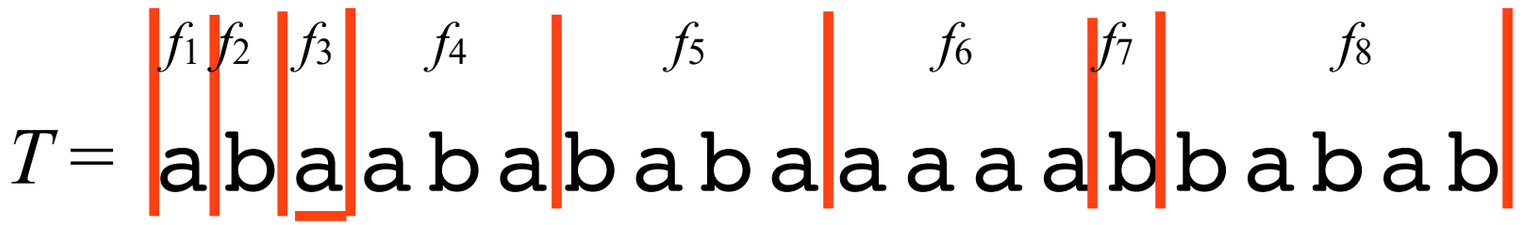
LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example



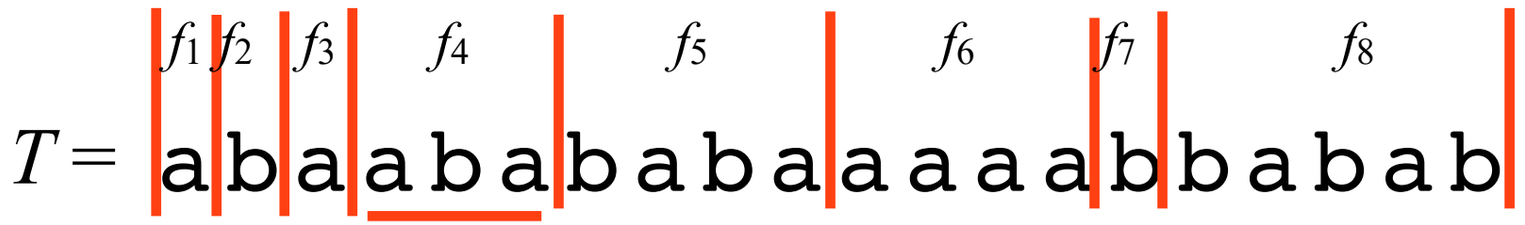
LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example



LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example

$T =$ | f_1 a | f_2 b | f_3 a | f_4 a b a | f_5 b a b a | f_6 a a a | f_7 b | f_8 b a b a b |

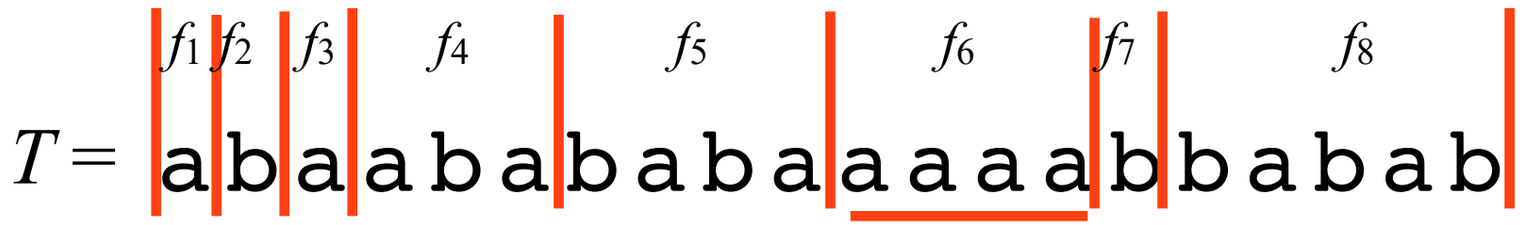
LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example



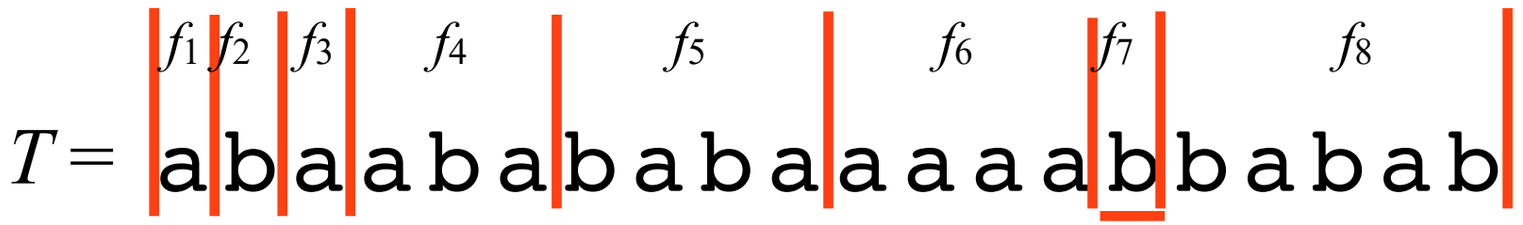
LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example



LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Example

$T =$ | f_1 a | f_2 b | f_3 a | f_4 a b a | f_5 b a b a | f_6 a a a | f_7 b | f_8 b a b a b |

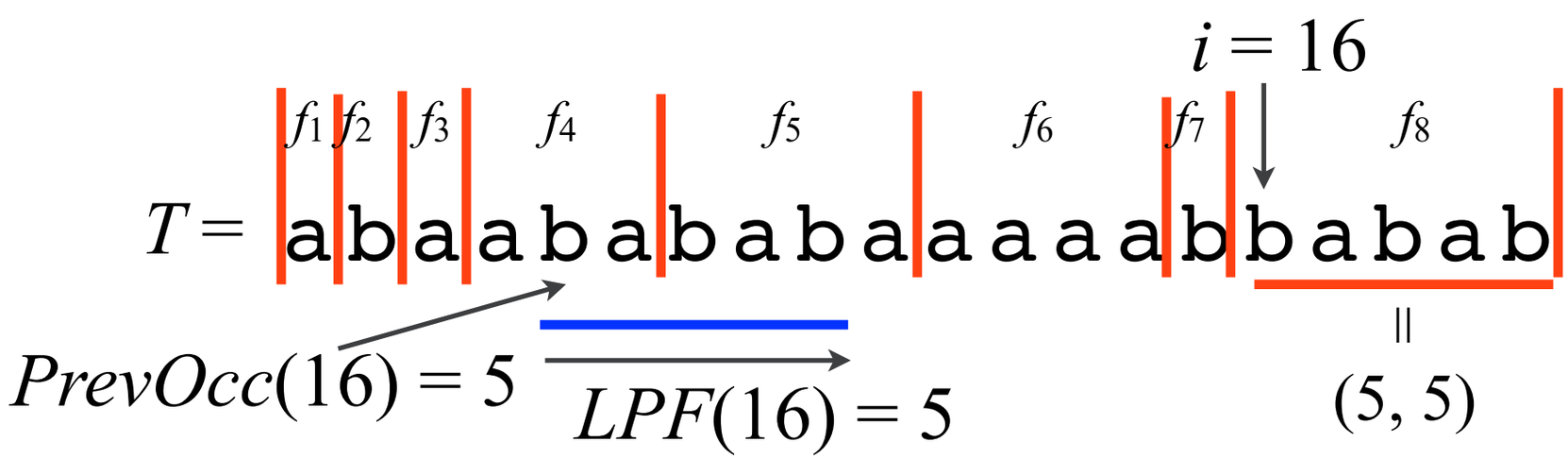
LZ77 Factorization

- For a string T of length N, the LZ77 factorization of T is a factorization $T = f_1 f_2 \ldots f_z$ such that each factor satisfies following conditions
- For a factor f_j, let $i = |f_1 f_2 .. f_{j-1}| + 1$,

  1) if $T[i]$ does not occur before, $f_j = T[i]$

  2) otherwise, f_j is the longest substring that occurs at i and somewhere before i

Each factor f_j starting at i can be represented as [L, SEP]
a single character (case 1), or a pair ($PrevOcc(i)$, $LPF(i)$)(case 2)



Algorithm by using *PrevOcc*, *LPF* Arrays

- Consider the arrays that store *PrevOcc*(i) and *LPF*(i) for all postions i

① For a factor f_j starting at i, output *PrevOcc*(i) and *LPF*(i)

② Update $i = i + LPF(i)$, and repeat ① when $i < N$

	f_1	f_2	f_3	f_4			f_5				f_6				f_7	f_8				
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

$PrevOcc(16) = 5$ $LPF(16) = 5$

||

$(5, 5)$

Previous Works Before 2013

- Most previous works focus on how to efficiently compute the *LPF* and *PrevOcc* arrays

	Space		required Integer Arrays						
Algorithm	Stack	# of Integer Arrays of	*LCP*	*LPF*	*Prev Occ*	*SA*	*PSV*	*NSV*	SA^{-1}
Crochemore & Ilie, 2008		5		✔	✔	✔	✔	✔	
	✔	4	✔	✔	✔	✔			
Chen+, 2008	✔	4	✔	✔	✔	✔			
	✔	3	✔	✔	✔				
	✔	2	✔		✔				
Crochemore+, 2008	✔	4	✔	✔	✔	✔			
Crochemore+, 2009	✔	4	✔	✔	✔	✔			
Ohlenbusch & Gog, 2011		3		✔	✔	✔			

Previous Works in 2013

- In 2013, algorithms not based on *LPF* and *PrevOcc* were proposed.
- Karkkainen+ reduced integer arrays of length *N* from three to two.

		Space		required Integer Arrays						
Algorithm	Name	Stack	# of Integer Arrays of length *N*	*LPF*	*Prev Occ*	*SA*	*PSV*	*NSV*	*Φ*	*SA*$^{-1}$
Ohlenbusch & Gog,2011	OG		3	✔	✔	✔				
Goto & Bannai, 2013	BG4	✔	4			✔	✔	✔		✔
	BG3		3			✔		✔	✔	
Kärkkäinen+, 2013	KKP3		3			✔	✔	✔		
	KKP2		2			✔		✔		

Our Contribution

We propose a linear time algorithm that uses a single integer array of length N in addition to $O(\sigma \log N)$ bits, where σ is alphabet size

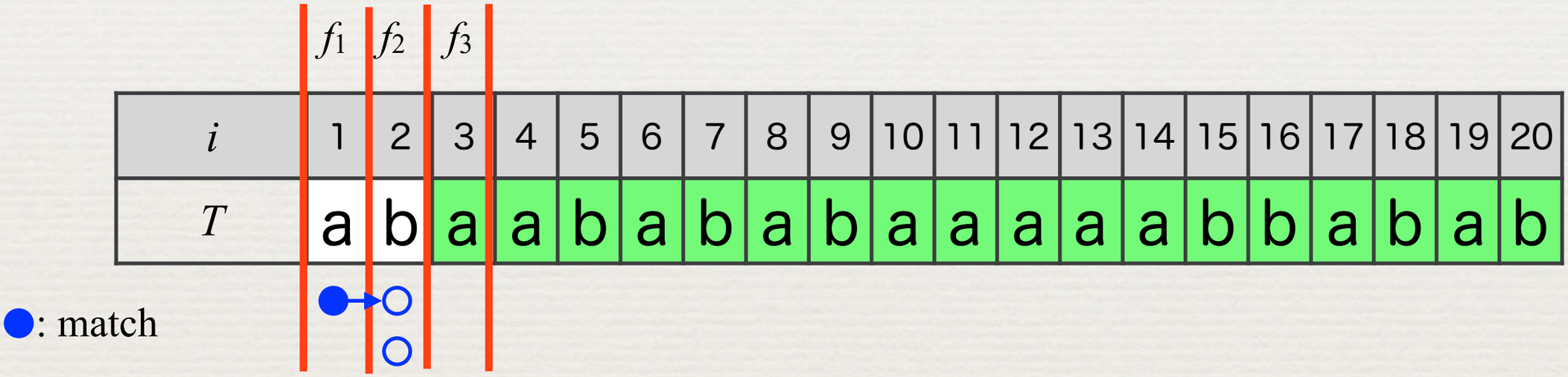
		Space		required Integer Arrays						
Algorithm	Name	Stack	# of Integer Arrays of length N	*LPF*	*Prev Occ*	*SA*	*PSV*	*NSV*	*Φ*	SA^{-1}
Ohlenbusch & Gog,2011	OG		3	✔	✔	✔				
Goto & Bannai, 2013	BG4	✔	4			✔	✔	✔		✔
	BG3		3			✔		✔	✔	
Kärkkäinen+, 2013	KKP3		3			✔	✔	✔		
	KKP2		2			✔		✔		
Goto & Bannai, 2014	BG2		2			✔		✔		
	BG1		1*					✔		

* require additional space of $O(\sigma \log N)$ bits

KKP3 / BG3

Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat

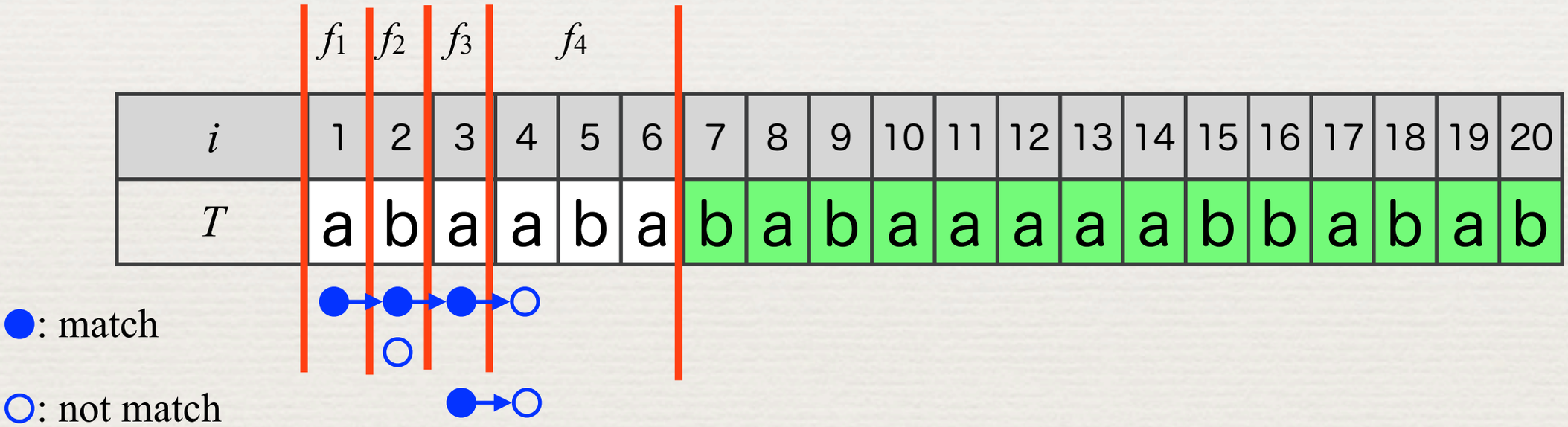


●: match

○: not match

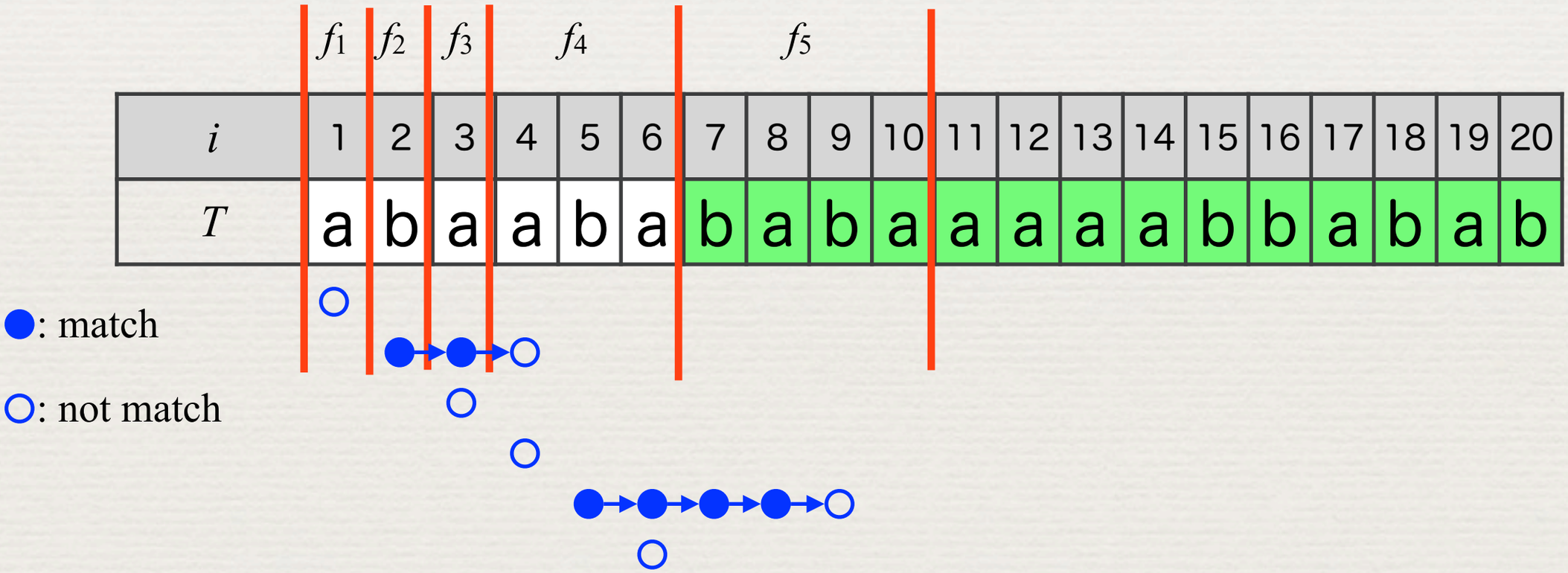
Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



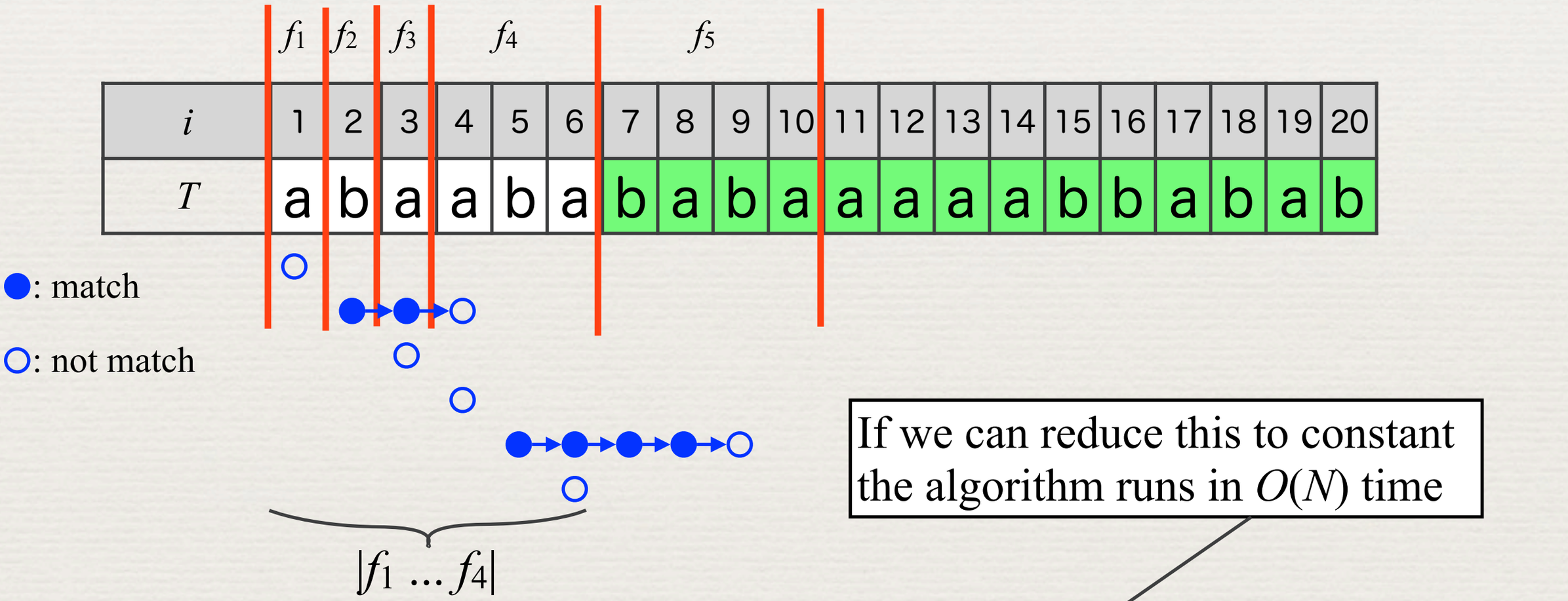
Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



If we can reduce this to constant the algorithm runs in $O(N)$ time

For a factor f_j, naive algorithm needs at most $|f_1 \ldots f_{j-1}| \cdot |f_j|$ comparisons

Total # of comparison is $\Sigma\, |f_1 \ldots f_{j-1}| \cdot |f_j| \leqq N \Sigma\, |f_j| = O(N^2)$

Idea of KKP3 / BG3

Lemma [Crochemore and Ilie, 2008]

The candidates of *PrevOcc*(i) can be reduced to 2 positions; the lexicographic predecessor and successor of suffix $i_{\$}$ in suffixes that start at previous positions

i	$SA[i]$	$T[SA[i]..N]$
⋮	⋮	⋮
15	2	baababababaaaaabbbabab
16	18	bab
17	7	babaaaaabbbabab
18	16	babab
19	5	bababaaaaabbbabab
20	15	bbabab

occur before 7

Idea of KKP3 / BG3

Lemma [Crochemore and Ilie, 2008]

$$PrevOcc(SA[i]) = PSV(SA[i]) \text{ OR } NSV(SA[i])$$

$$PSV(SA[i]) = \max\{SA[j] \mid j < i, SA[j] < SA[i]\}$$

$$NSV(SA[i]) = \min\{SA[j] \mid j > i, SA[j] < SA[i]\}$$

i	$SA[i]$	$T[SA[i]..N]$
⋮	⋮	⋮
15	2	baababababaaaaabbbabab
18	18	bab
17	7	babaaaaabbbabab
16	16	babab
19	5	bababaaaaaabbbabab
20	15	bbabab

$PSV(7)$

$NSV(7)$

occur before 7

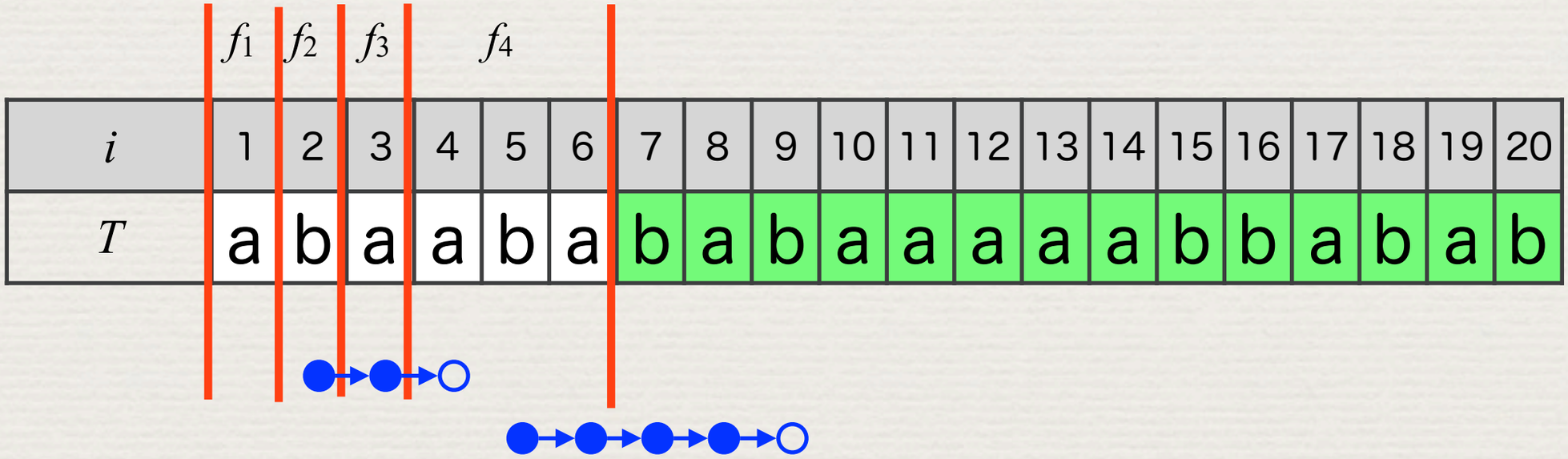
Idea of KKP3 / BG3

Lemma [Crochemore and Ilie, 2008]

$$PrevOcc(SA[i]) = PSV(SA[i]) \text{ OR } NSV(SA[i])$$

$$PSV(SA[i]) = \max\{SA[j] \mid j < i,\ SA[j] < SA[i]\}$$

$$NSV(SA[i]) = \min\{SA[j] \mid j > i,\ SA[j] < SA[i]\}$$



For a factor f_j, the number of comparison is $2\,|f_j|$

Total # of comparison is $\Sigma\ 2\,|f_j| = O(N)$

Linear Time Computation of *PSV* and *NSV* Arrays

Lemma [Crochemore and Ilie, 2008]

All *PSV* and *NSV* values can be obtained in linear time
by sequentially scanning *SA* from left to right OR right to left

lexicographic rank

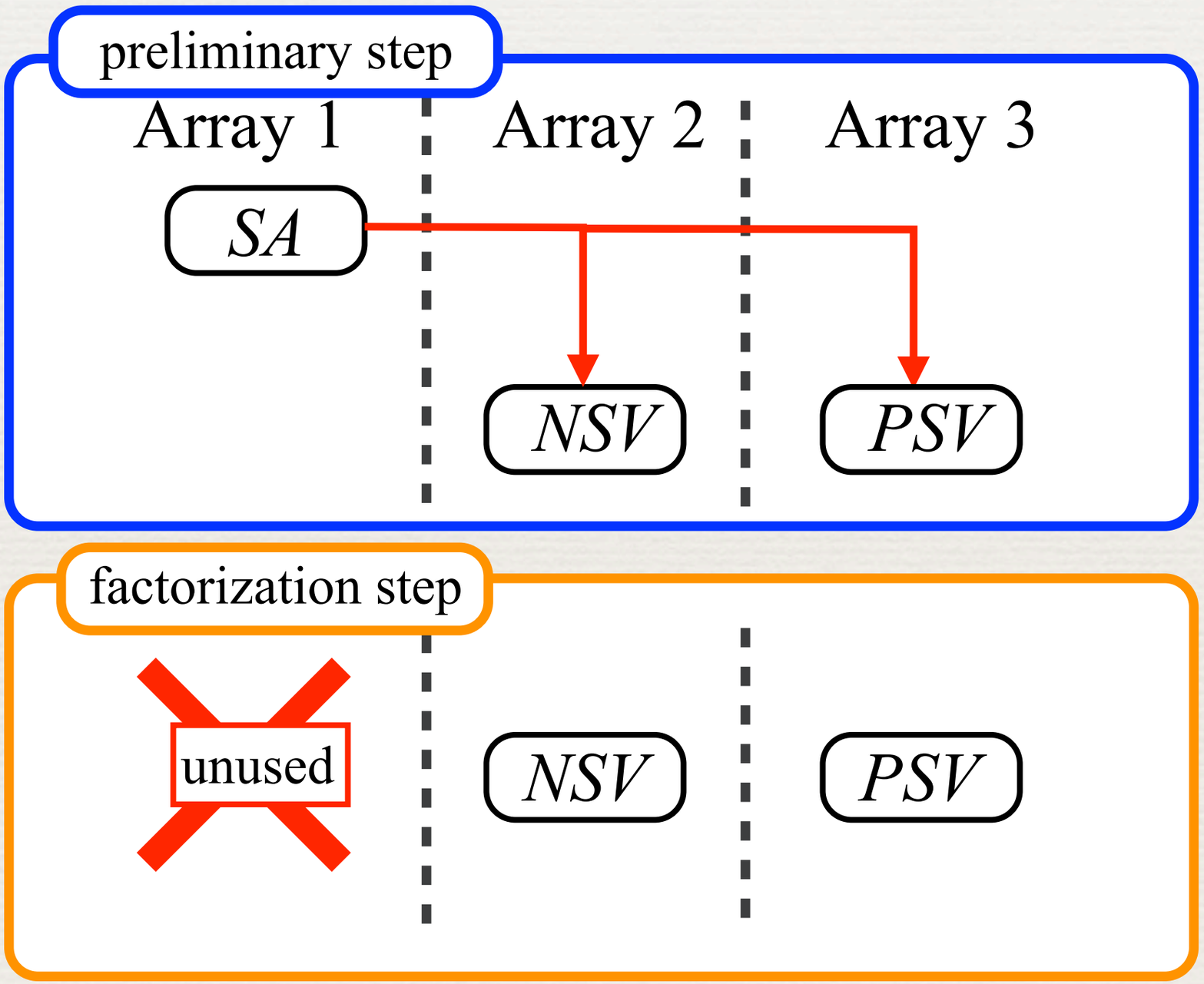
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	12	7	15	4	9	11	2	8	13	1	14	6	3	10	16	5

text positions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PSV																
NSV																

Overview of KKP3

KKP3 run in linear time and $3N \log N$ bits space



KKP2

Φ Array [Kärkkäinen+, 2009]

Φ array is an array that stores each lexicographic predecessor of each suffix in text order.

$\Phi[SA[i]] = SA[i\text{-}1]$ $(i > 1)$, $\Phi[SA[1]] = N$

NOTE: all $SA[i]$ can be obtained from right to left by Φ

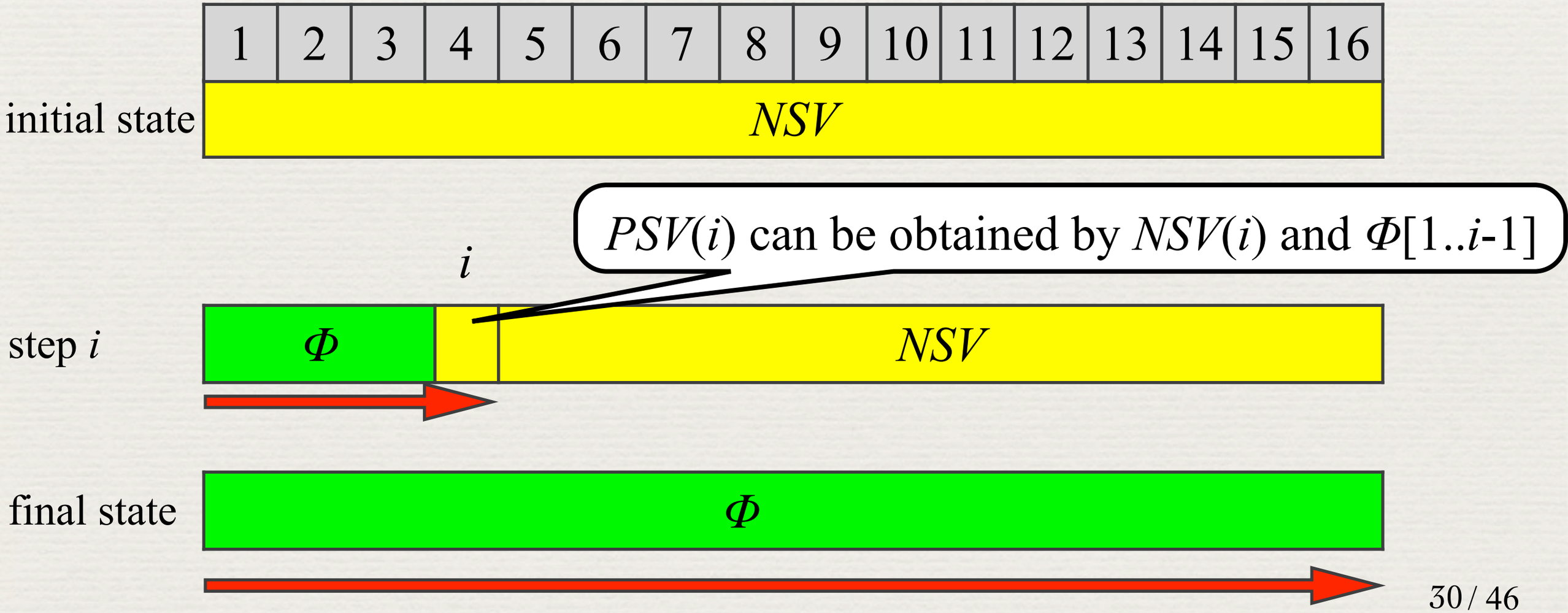
T = acaaacatat

i	$\Phi[i]$	$SA[i]$	$T[SA[i]]$
1	4	3	aaacatat
2	7	4	aacatat
3	0	1	acaaacatat
4	3	5	acatat
5	1	9	at
6	2	7	atat
7	9	2	caaacatat
8	10	6	catat
9	5	10	t
10	6	8	tat

Lemma [Kärkkäinen+, 2013]

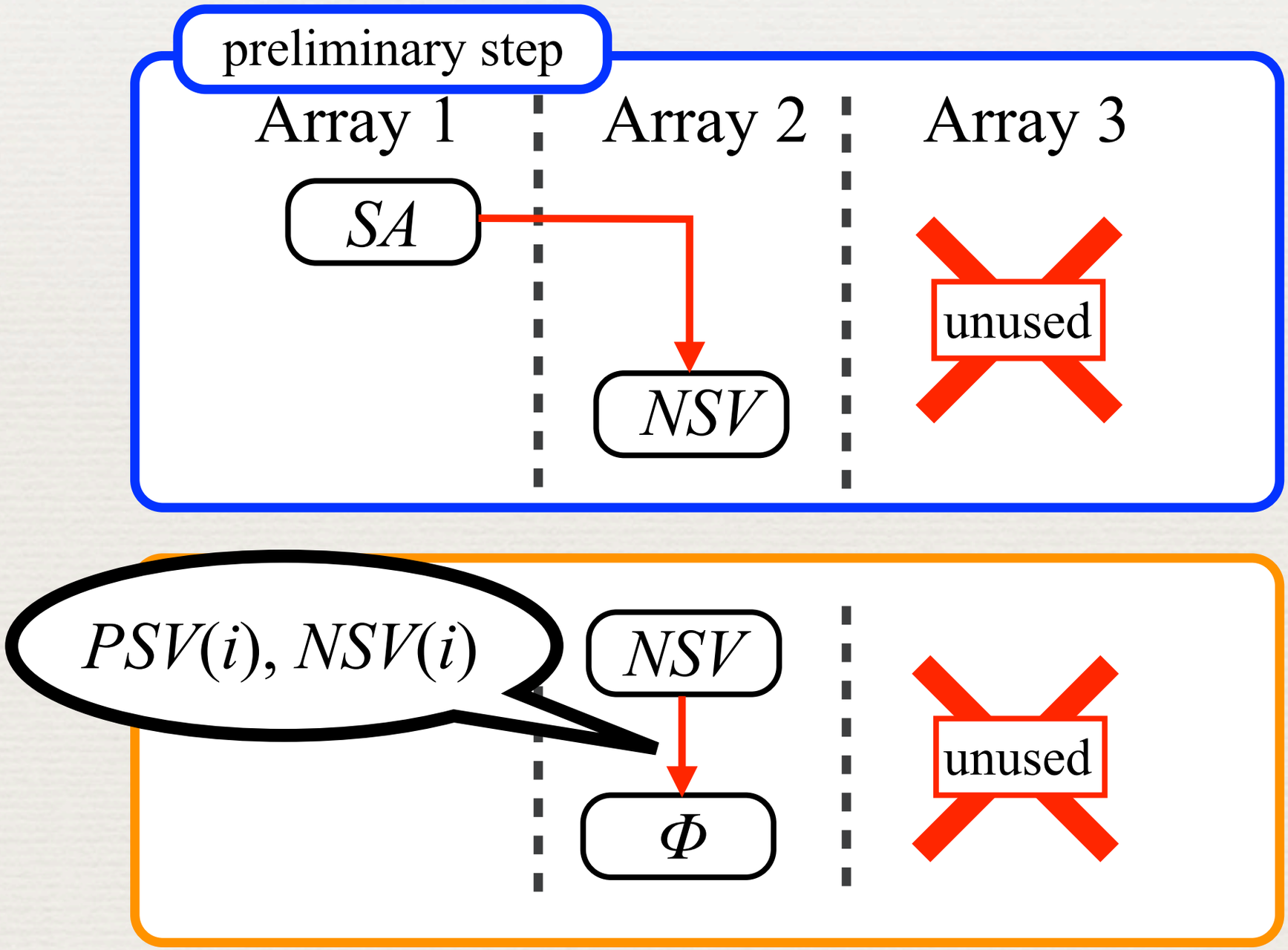
Φ array can be constructed in linear time and in-place from *NSV*(*PSV*) array by sequentially scanning from left to right

NOTE: for each step i, *PSV*(i), *NSV*(i) can be obtained



Overview of KKP2

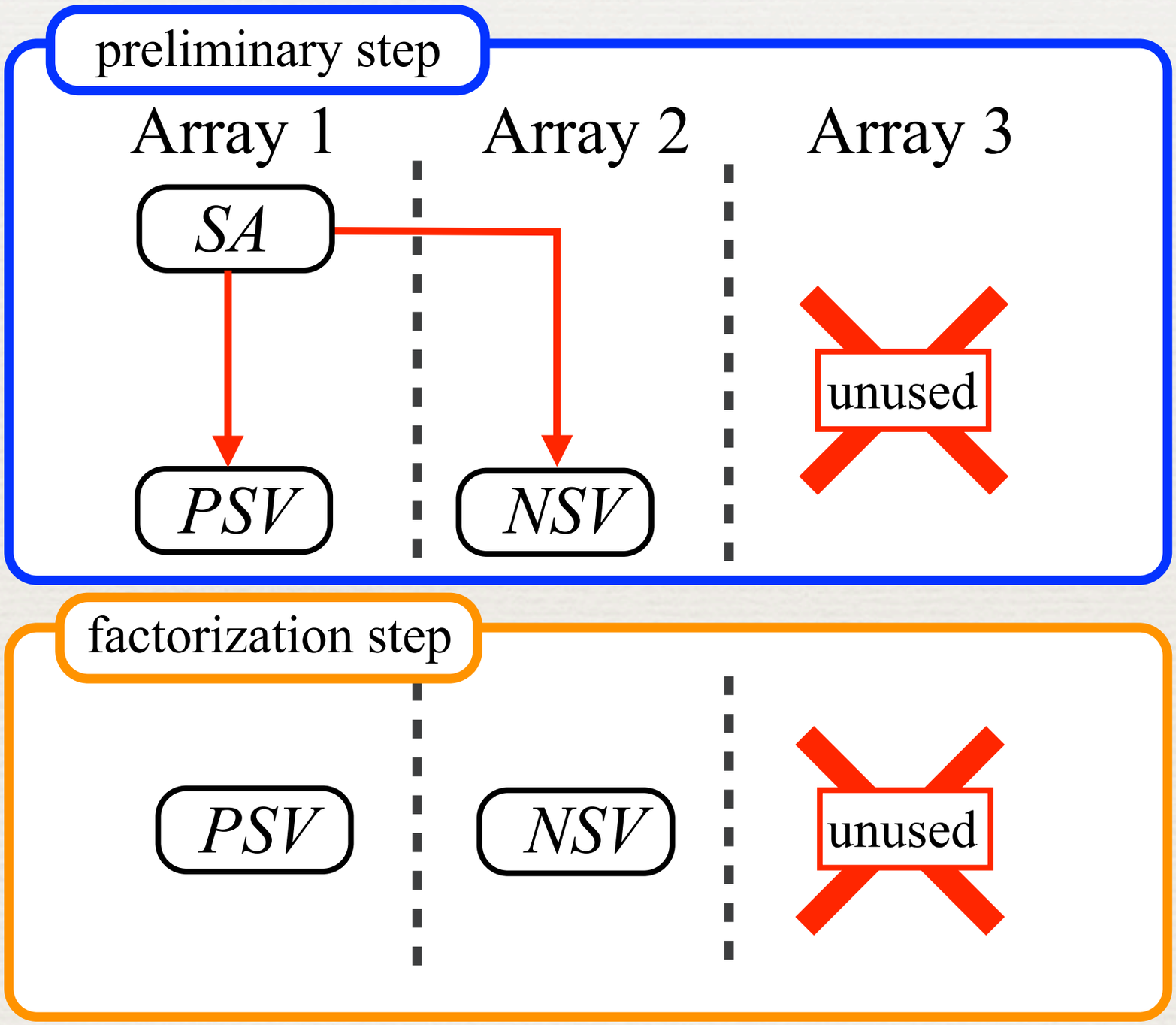
KKP2 runs in linear time and $2N \log N$ bits space



BG2

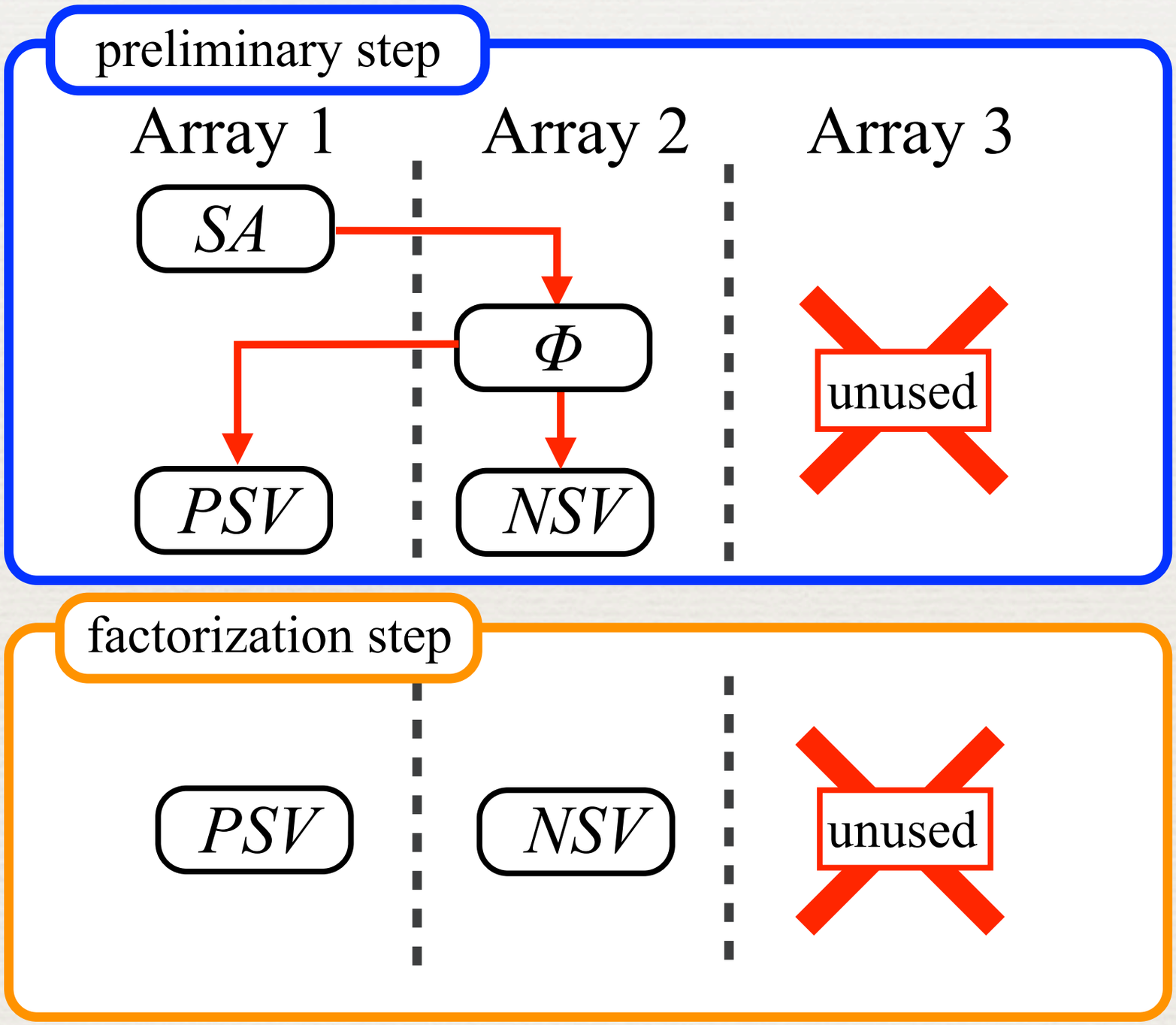
Idea of BG2

If we can rewrite *SA* to *PSV* array, LZ77 factorization can be computed by two integer arrays



Idea of BG2

It can be accomplished through Φ array



Observation

- By scanning $SA[i]$ from left to right, $PSV(SA[i])$, $NSV(SA[i])$ can be obtained
- $SA[i]$ never be read after read once

Is it possible to store $NSV[SA[i]]$ to $SA[i]$?

It seems IMPOSSIBLE because they have a different ordering

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	9	11	2	8	13	1	14	6	3	10	16	5	12	7	15	4

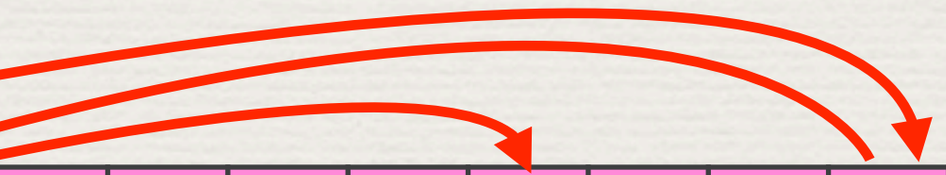
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
NSV																
PSV																

Observation

- All $SA[i]$ can be obtained from right to left by Φ
- $\Phi[SA[i]]$ never be used after read once

PSV and NSV arrays can be computed by Φ

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	9	11	2	8	13	1	14	6	3	10	16	5	12	7	15	4



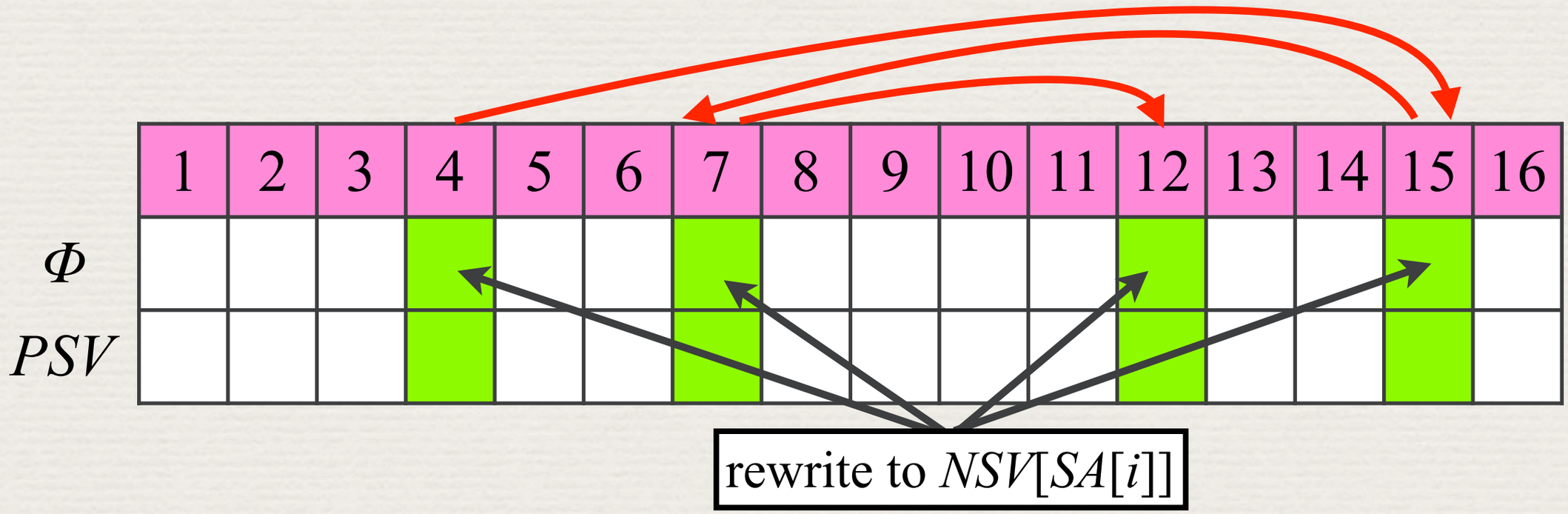
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Φ				15			12								7	
PSV																

Lemma

NSV array can be rewrited from Φ array in linear time and in-place

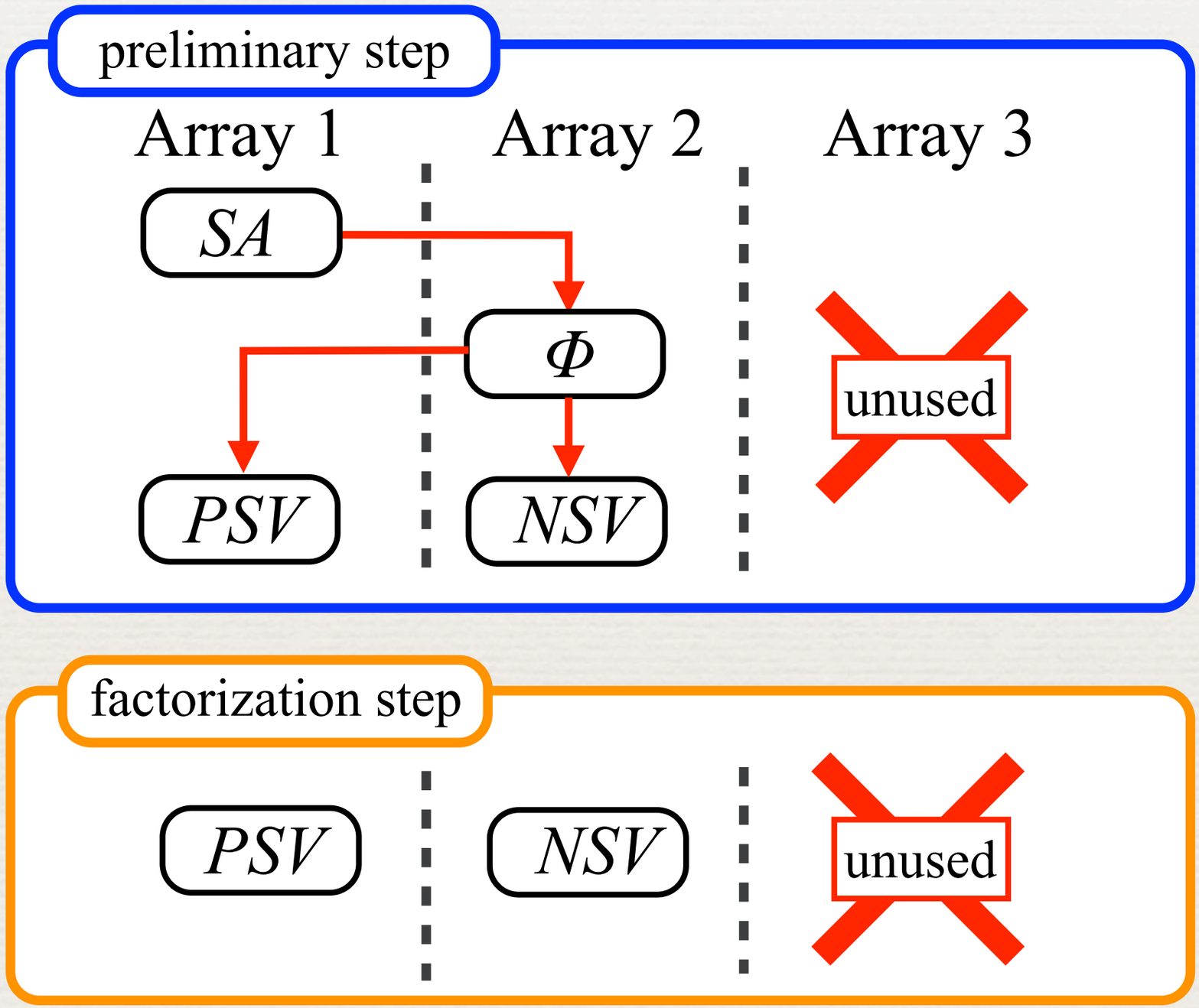
Since *PSV*, *NSV*, and Φ array are text ordering,
NSV[*SA*[*i*]] can be stored in Φ[*SA*[*i*]]

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	9	11	2	8	13	1	14	6	3	10	16	5	12	7	15	4



Overview of BG2

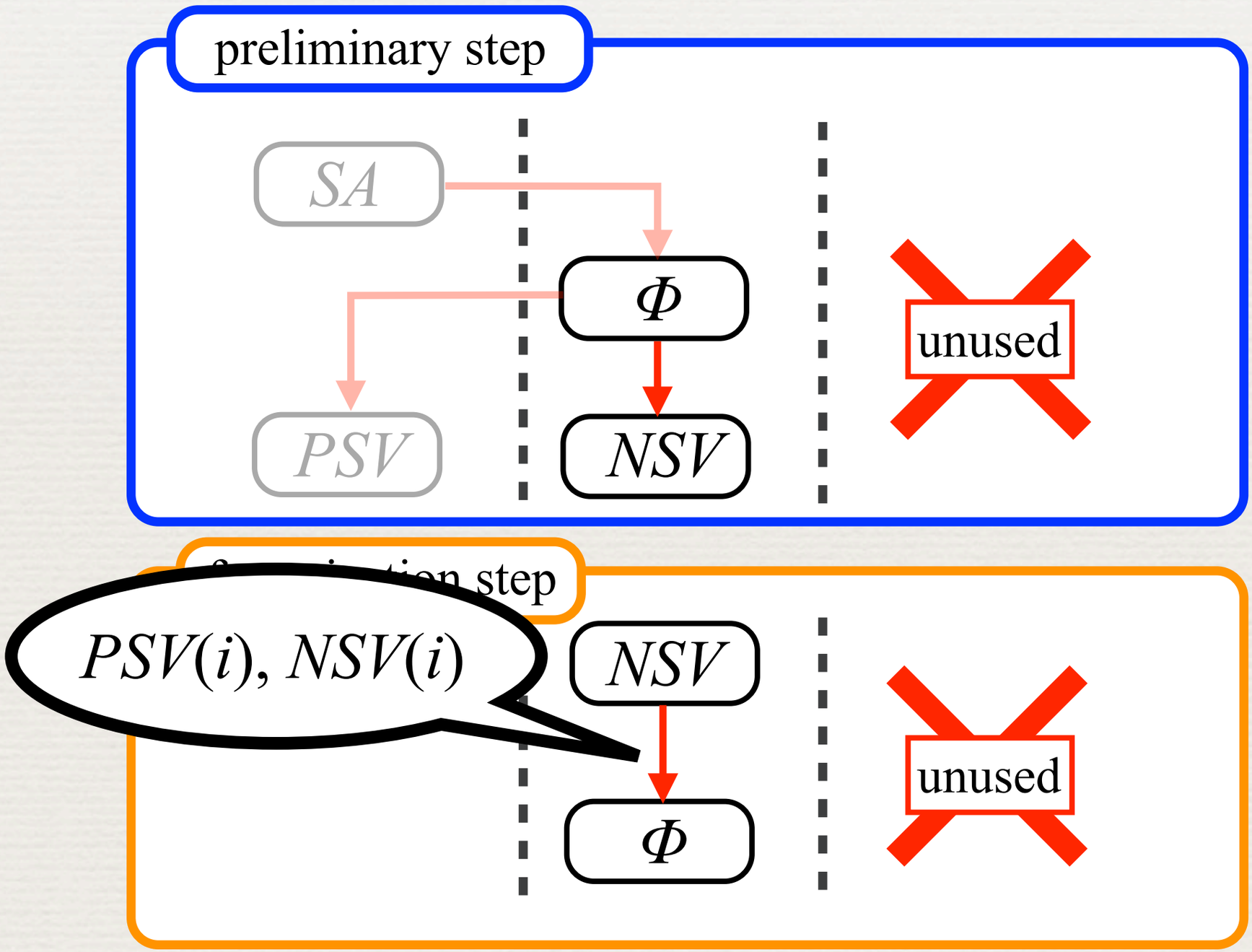
BG2 runs in linear time and $2N \log N$ bits space



BG1

Idea of BG1

- If given Φ not SA, LZ77 factorization can be computed in $N \log N$ bits space by using ideas of BG2 and KKP2



In-place computation of Φ array

Lemma

For a string T of length N, the Φ array of T can be computed in linear time and $O(\sigma \log N)$ bits additional space

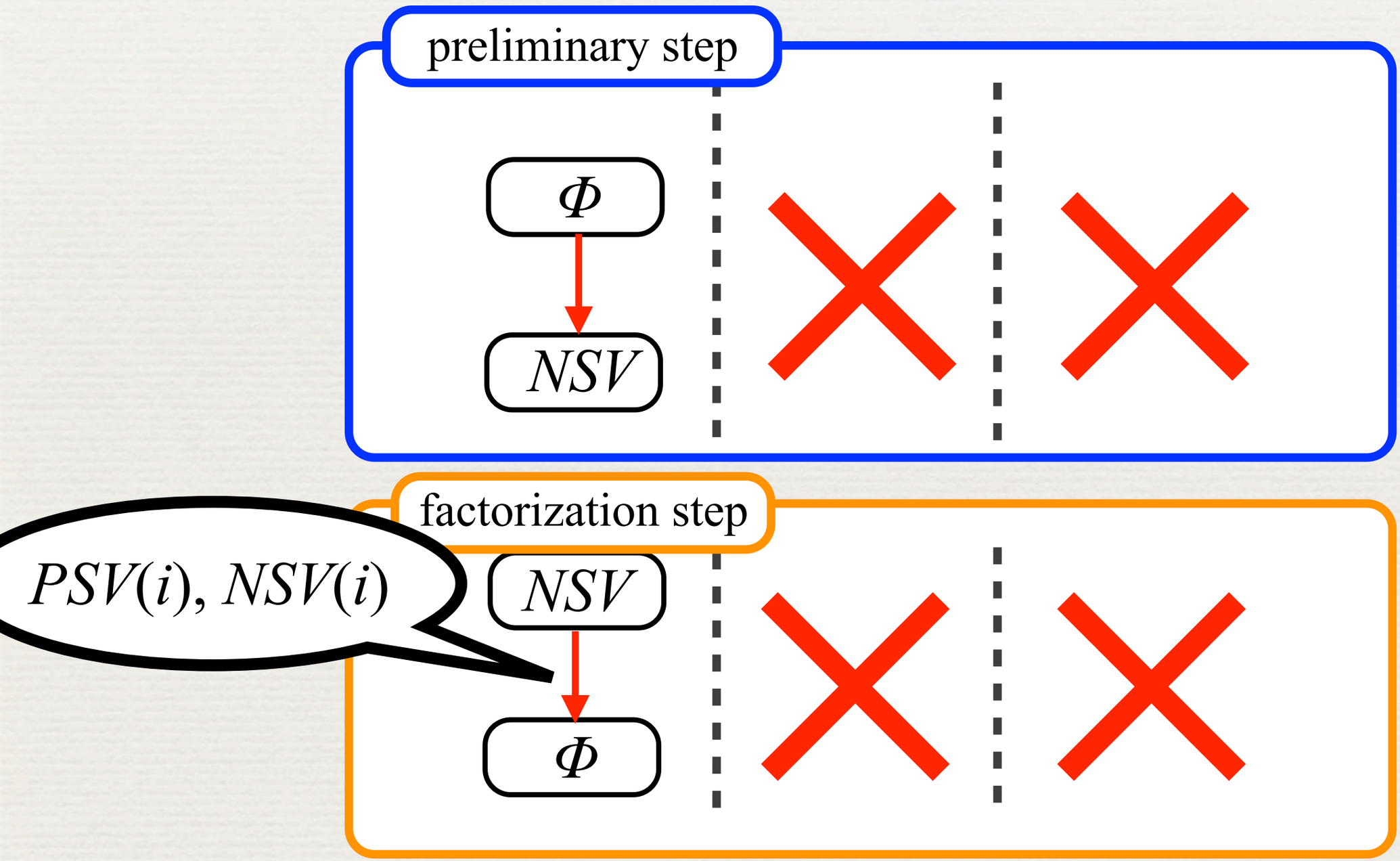
- Idea: simulating Nong's suffix array construction algorithm on Φ array

Theorem [Nong, 2013]

For a string T of length N, the SA of T can be computed in linear time and $O(\sigma \log N)$ bits additional space

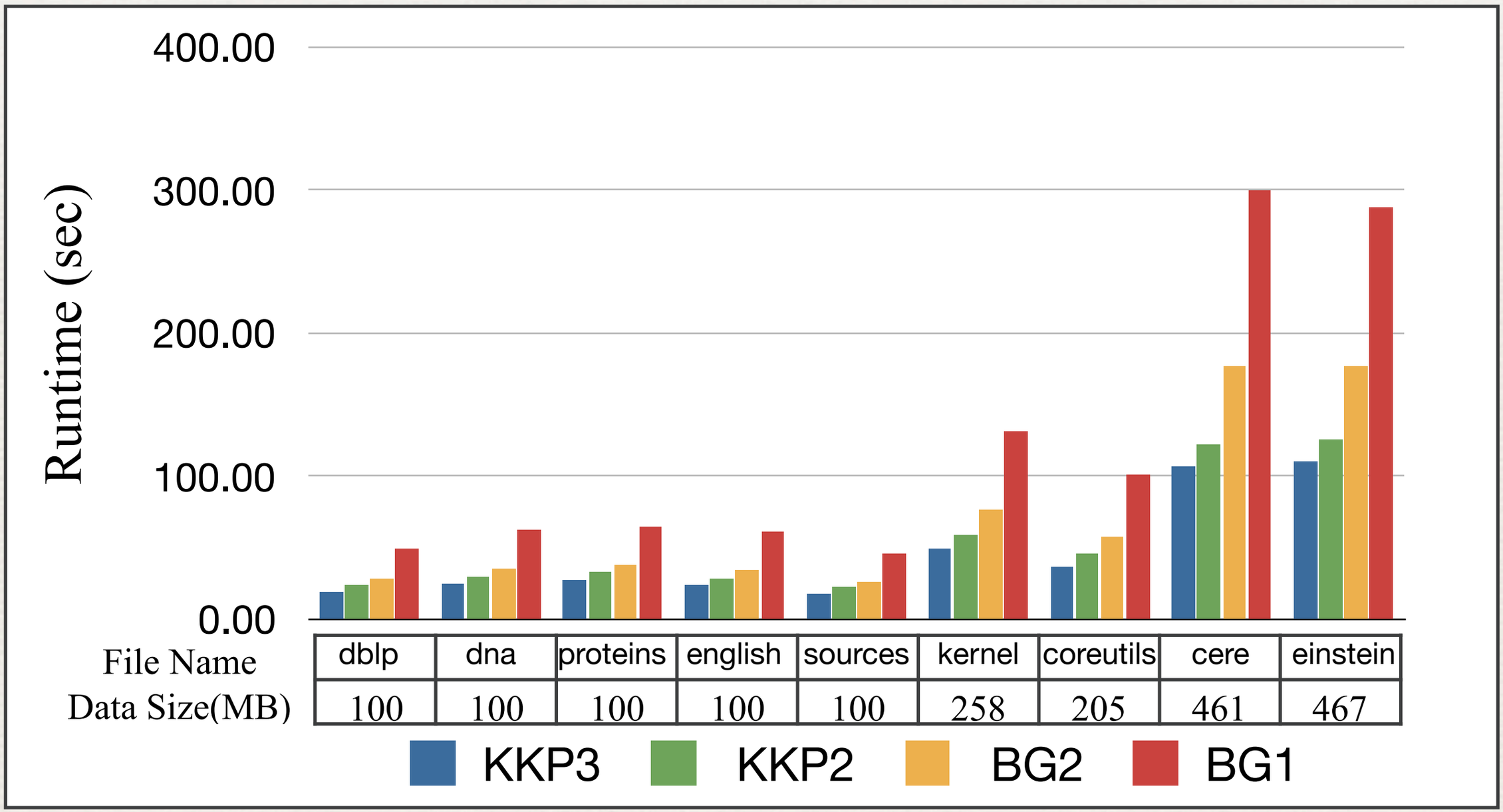
Overview of BG1

BG1 runs in linear time and $N \log N + O(\sigma \log N)$ bits space



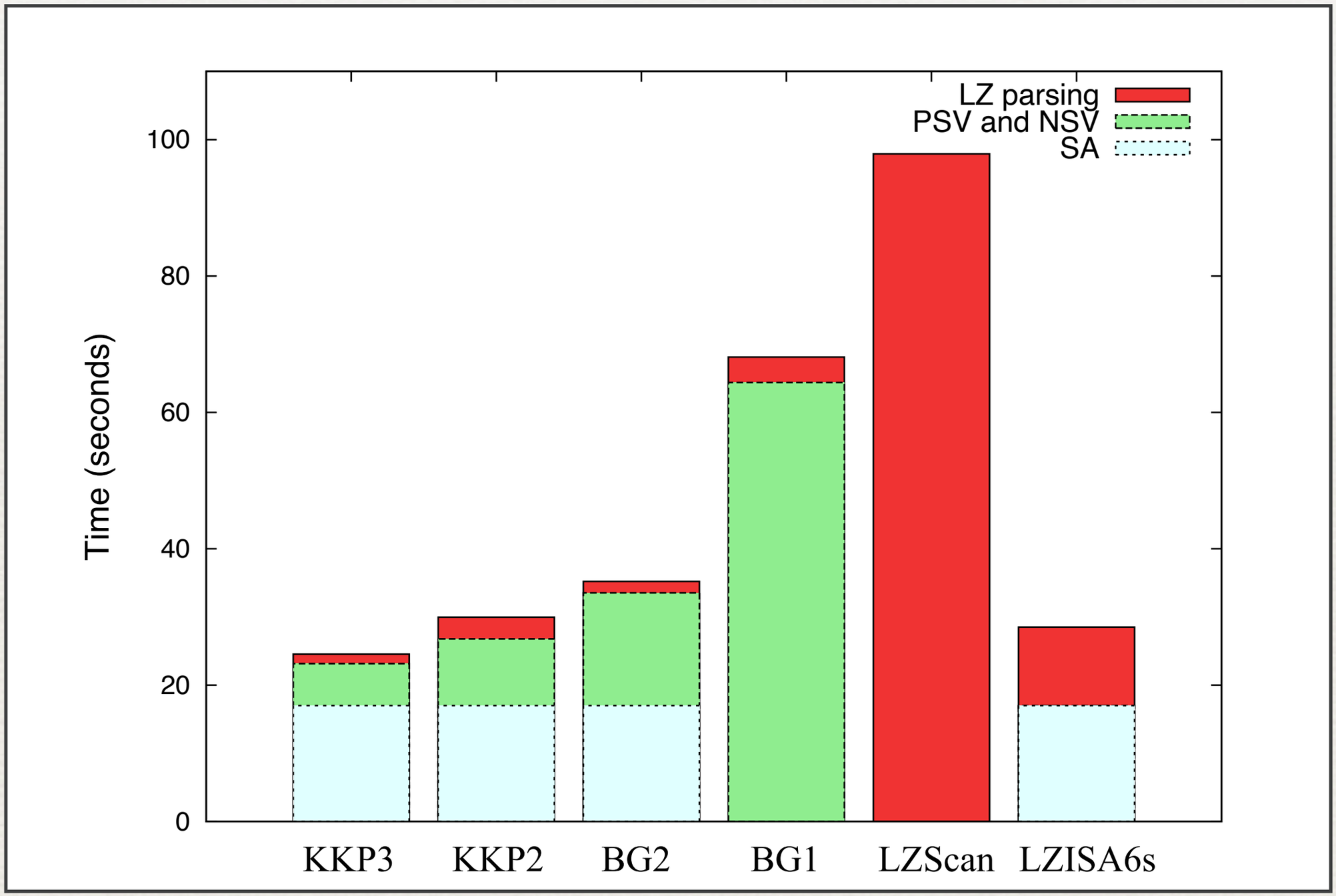
Computational Experiments

- Nevertheless BG1 uses a third space than KKP3,
  the runtime of BG1 does not over 2.5 times than that of KKP3
- Nevertheless BG1 uses a half space than KKP2,
  the runtime of BG1 does not over 2 times than that of KKP2



corpus: http://www.cas.mcmaster.ca/~bill/strings

DNA 100MB $\sigma = 16$



Summary

We proposed a space efficient linear time LZ77 factorization algorithm for small alphabets, which uses $N \log N$ bits + $O(\sigma \log N)$ bits of space

Future Perspective

- Can we reduce the space $O(\sigma \log N)$ bits of BG1?
  - It seems difficult since we have to compute Φ array truly in-place.
- We should perhaps shift our focus to practically fast non-linear time algorithms which use less space than BG1.