

省領域な
lexicographic parse
構築アルゴリズム

カップル ドミニク
東京医科歯科大学

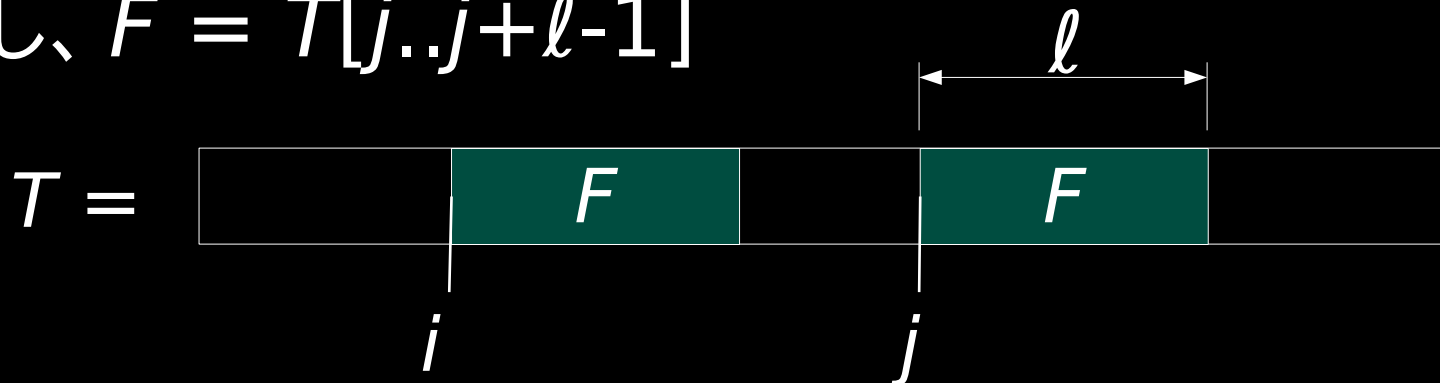
コンピューテーション研究会
令和3年 12 月 3 日

lexparse

- テキストデータの可逆圧縮の方法
- テキストの接尾辞の辞書式順序に基づいた反復を利用する
- bidirectional parse の一種
- [Navarro+ '21] によって提案された
(2018年に初めて arXiv に投稿された)

bidirectional parse

- テキスト T を項に分割する
- 各項 $F = T[i..i+l-1]$ は
 - 1つの文字 ($l=1$)、もしくは
 - 2つの組 (参照先 j 、長さ l)で表現する
- ただし、 $F = T[j..j+l-1]$



bidirectional parse の例

- 部分文字列を参照に置換 \Rightarrow 項を作る

$T =$

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

bidirectional parse の例

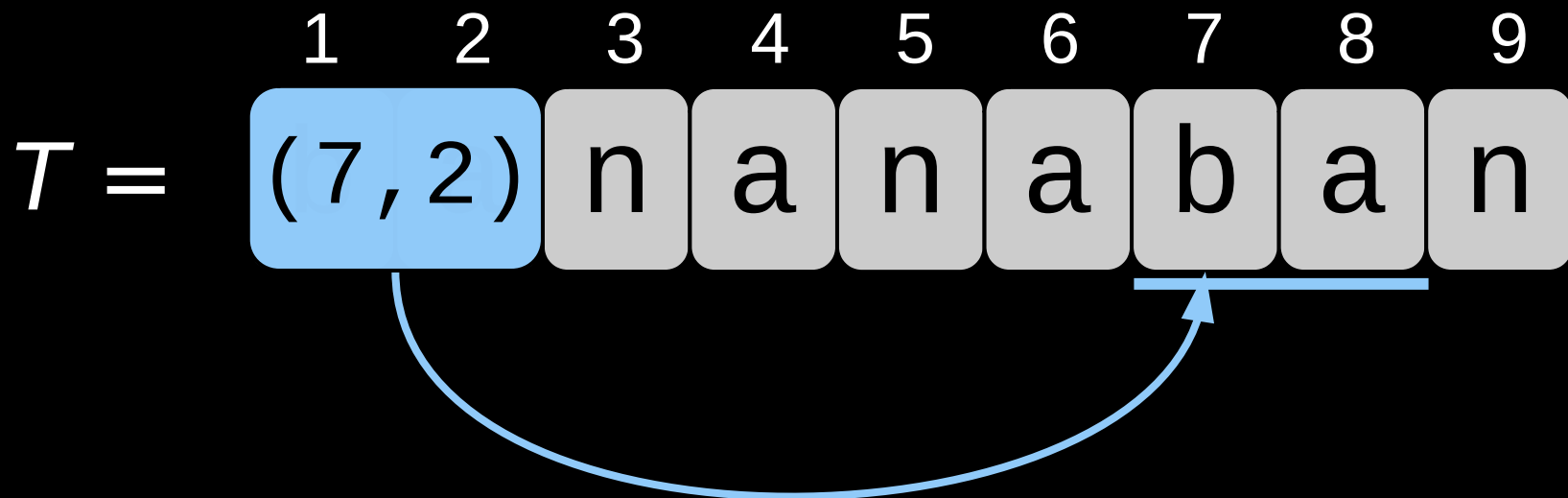
- 部分文字列を参照に置換 \Rightarrow 項を作る

$T =$

1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	<u>b</u>	a	n

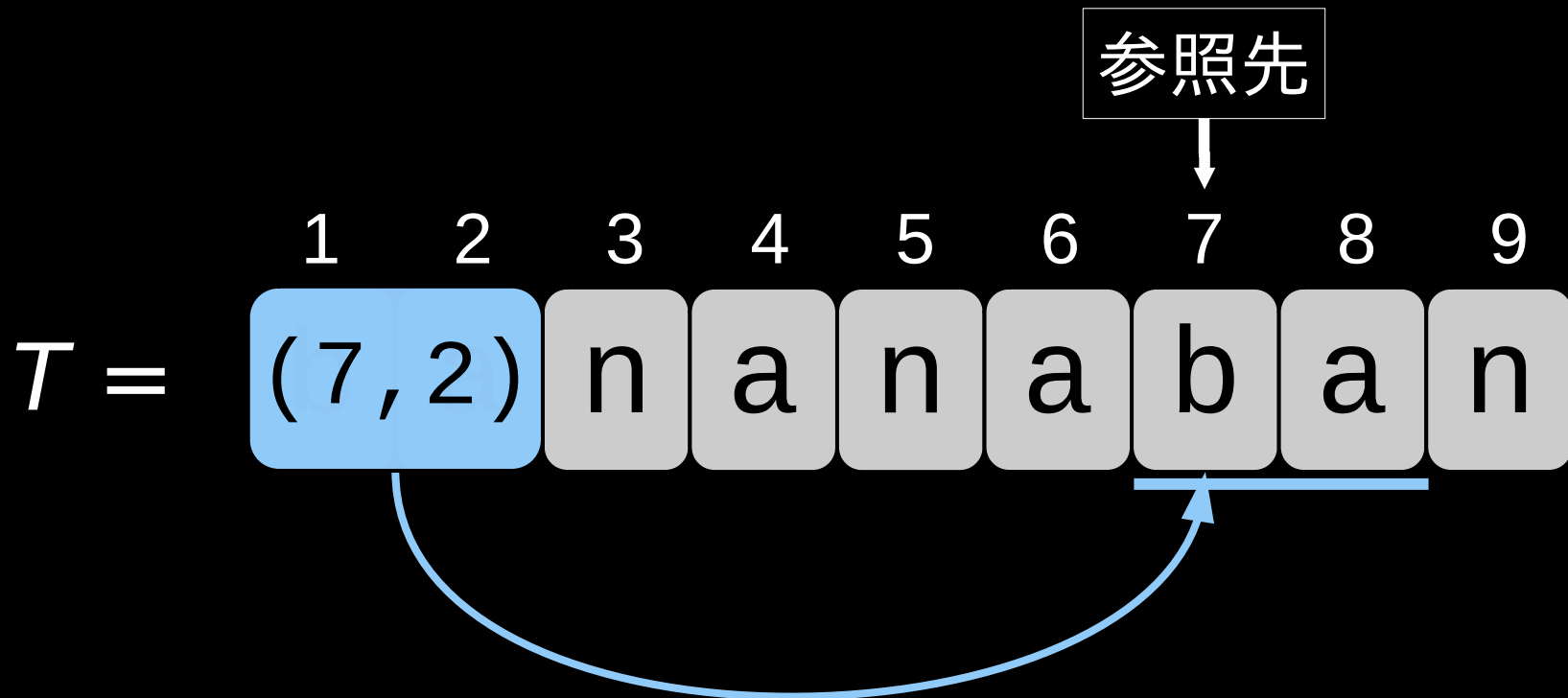
bidirectional parse の例

- 部分文字列を参照に置換 \Rightarrow 項を作る



bidirectional parse の例

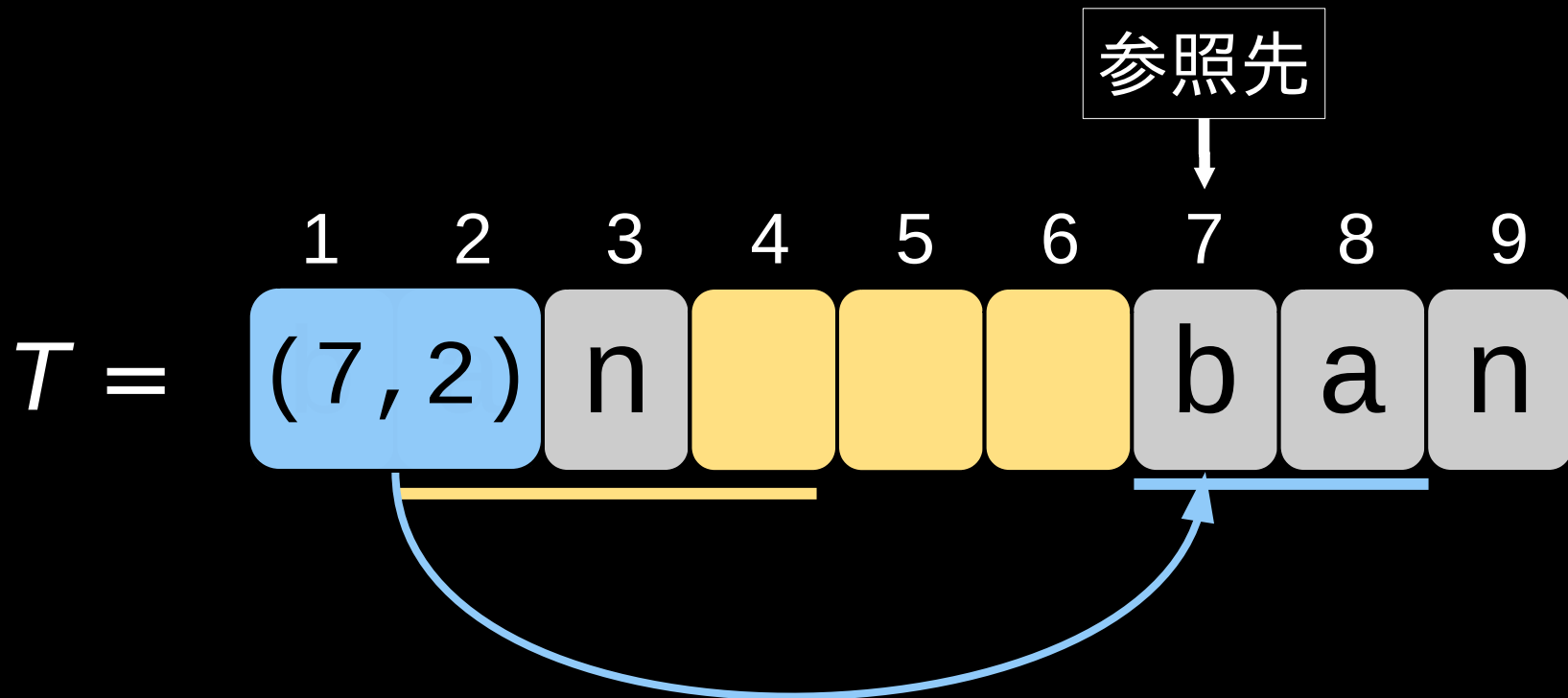
- 部分文字列を参照に置換 ⇒ 項を作る



符号: 2つの組 (参照先, 長さ)

bidirectional parse の例

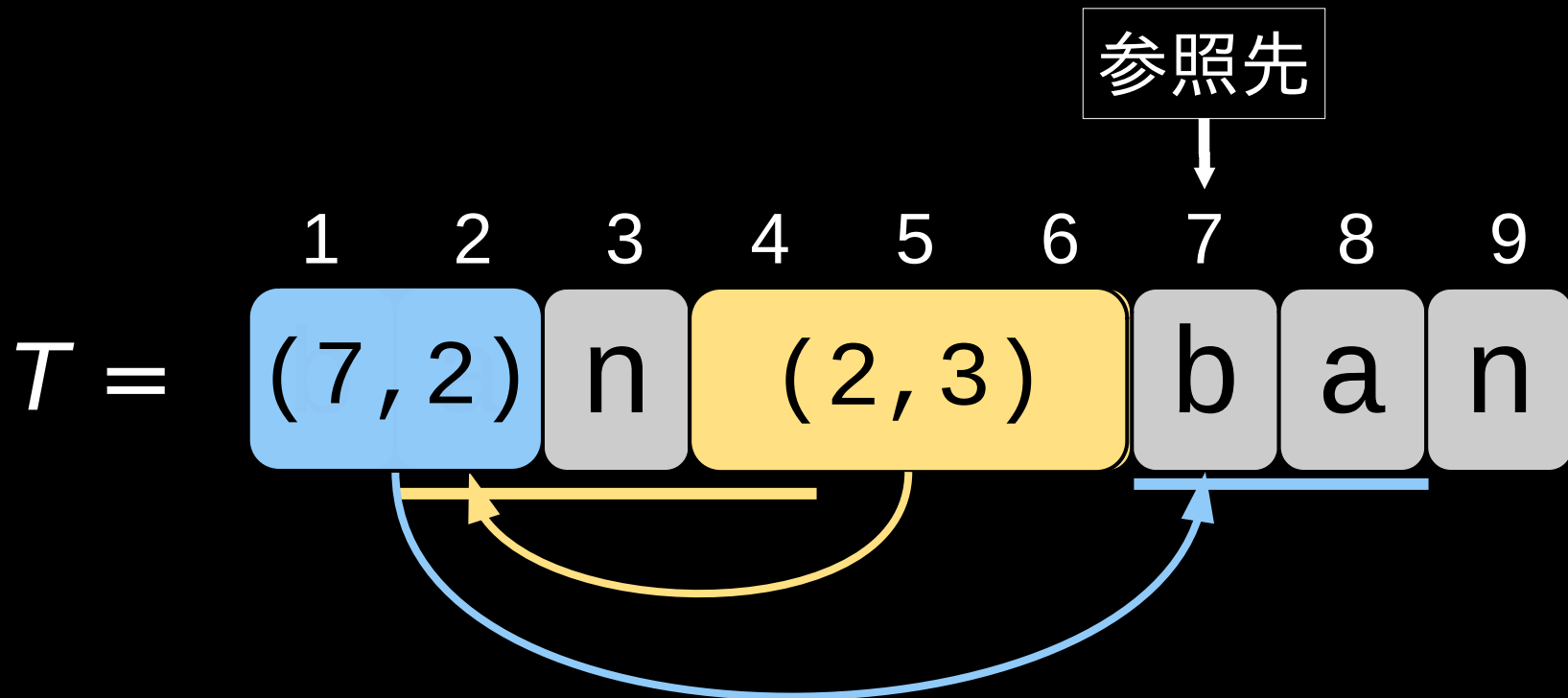
- 部分文字列を参照に置換 ⇒ 項を作る
- 自己参照も OK



符号: 2つの組 (参照先, 長さ)

bidirectional parse の例

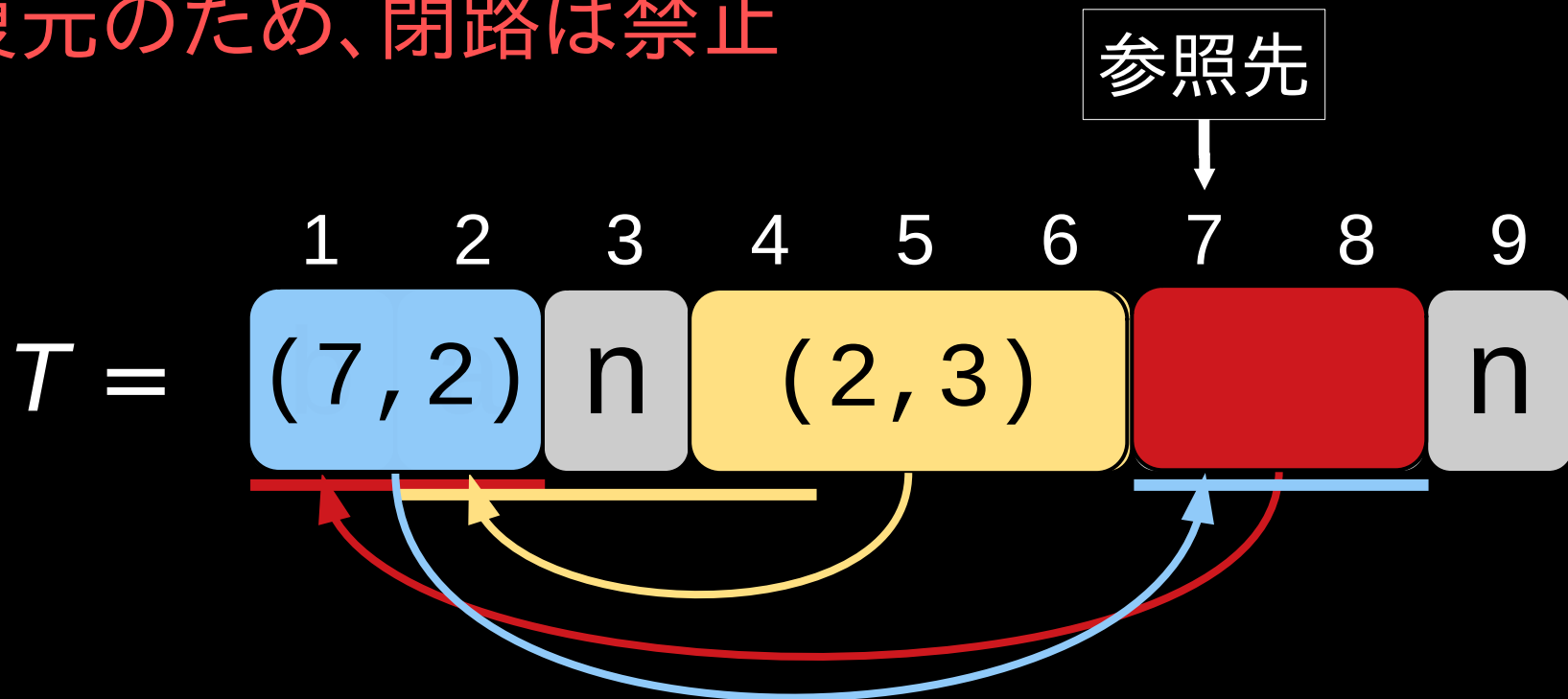
- 部分文字列を参照に置換 ⇒ 項を作る
- 自己参照も OK



符号: 2つの組 (参照先, 長さ)

bidirectional parse の例

- 部分文字列を参照に置換 \Rightarrow 項を作る
- 自己参照も OK
- 復元のため、閉路は禁止

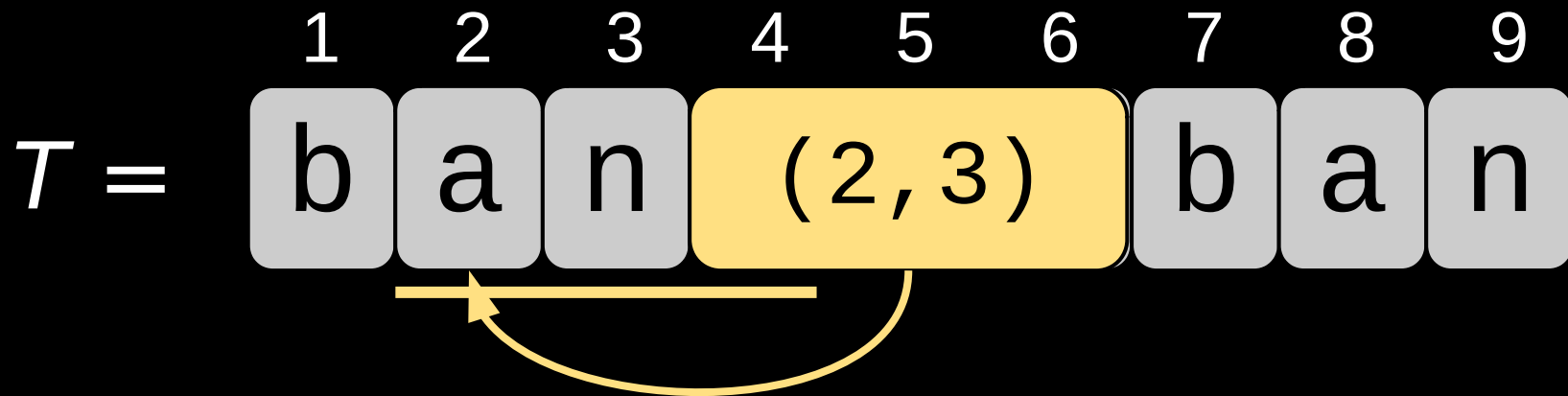


符号: 2つの組 (参照先, 長さ)

自己参照の復元

自己参照も OK

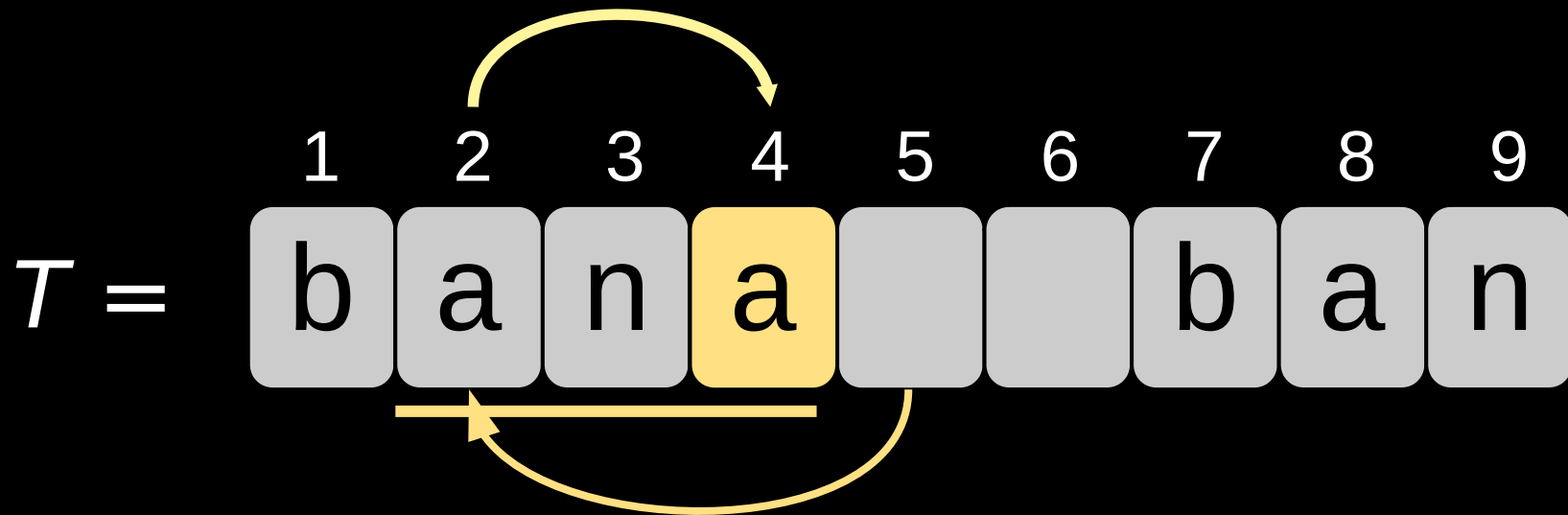
- 復元するとき、文字それぞれをコピーできる



自己参照の復元

自己参照も OK

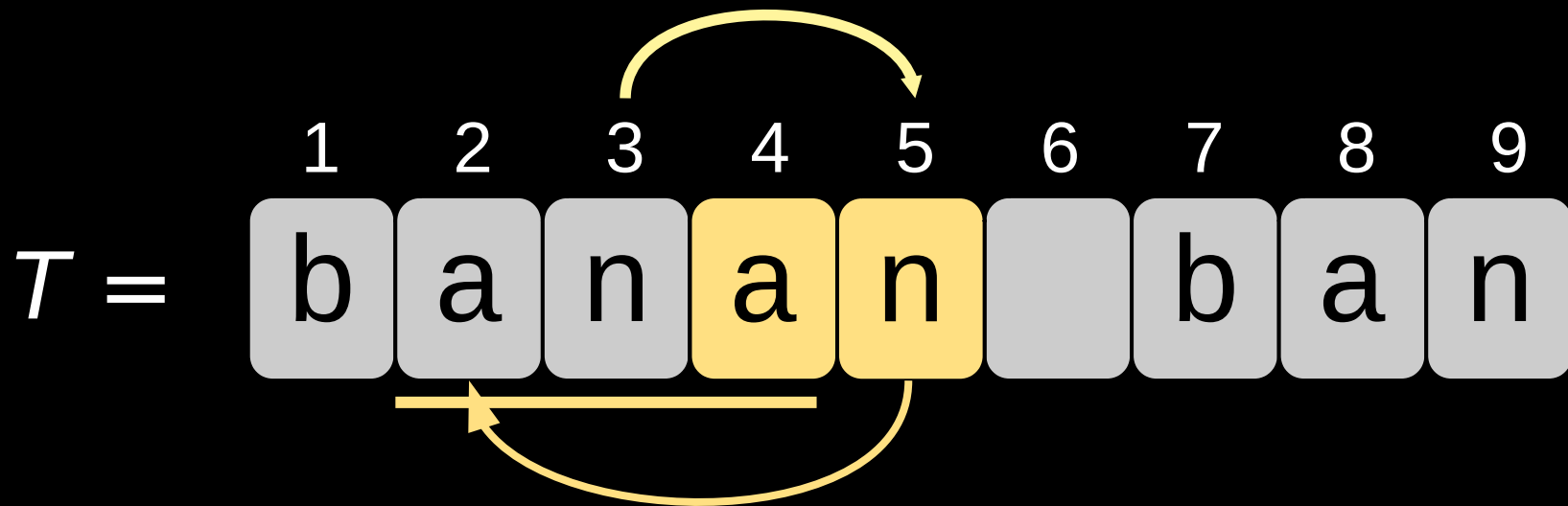
- 復元するとき、文字それぞれをコピーできる



自己参照の復元

自己参照も OK

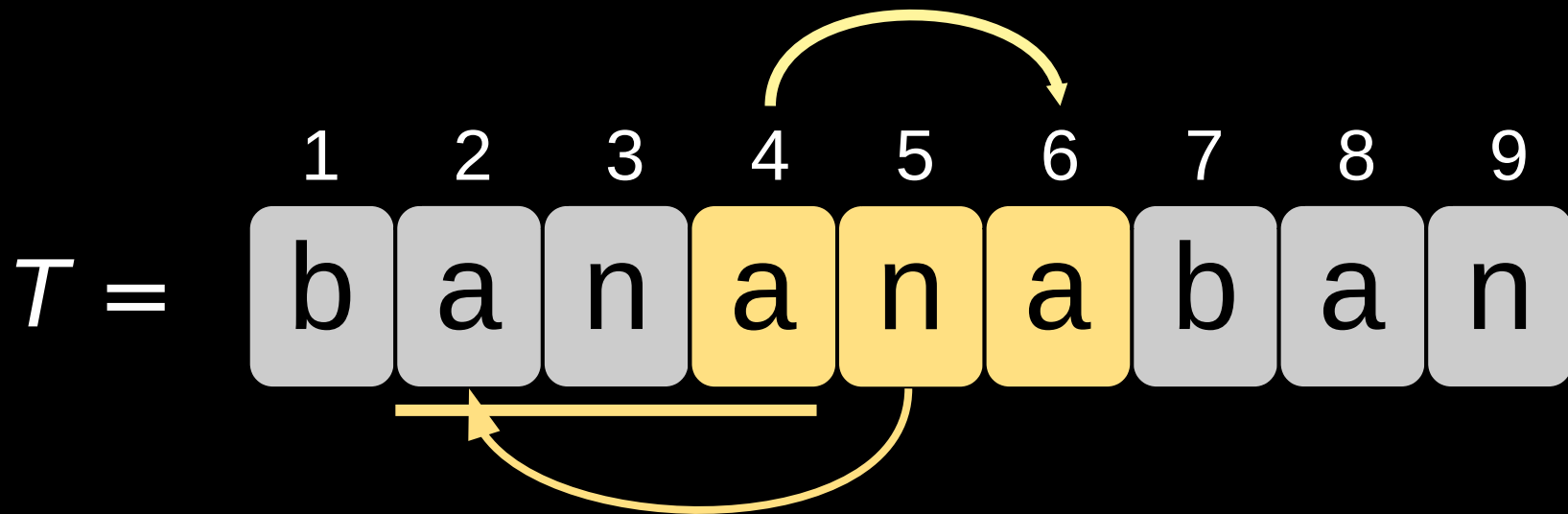
- 復元するとき、文字それぞれをコピーできる



自己参照の復元

自己参照も OK

- 復元するとき、文字それぞれをコピーできる



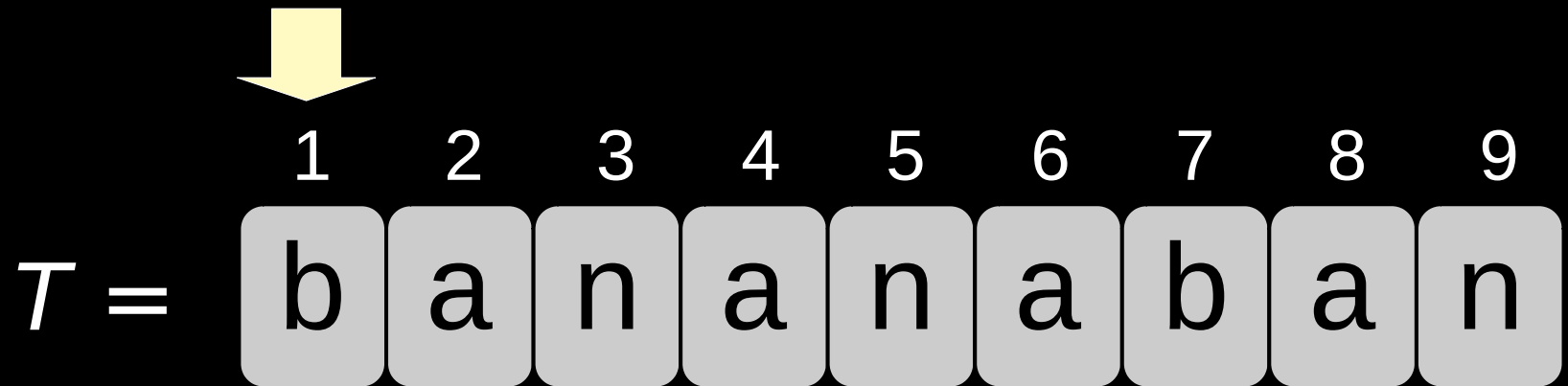
用語集

- T : 入力のテキスト
- n : T の長さ、すなわち $n := |T|$
- σ : アルファベットサイズ
- $T[i..]$: T の i 番目の文字から始まる接尾辞

lexparse

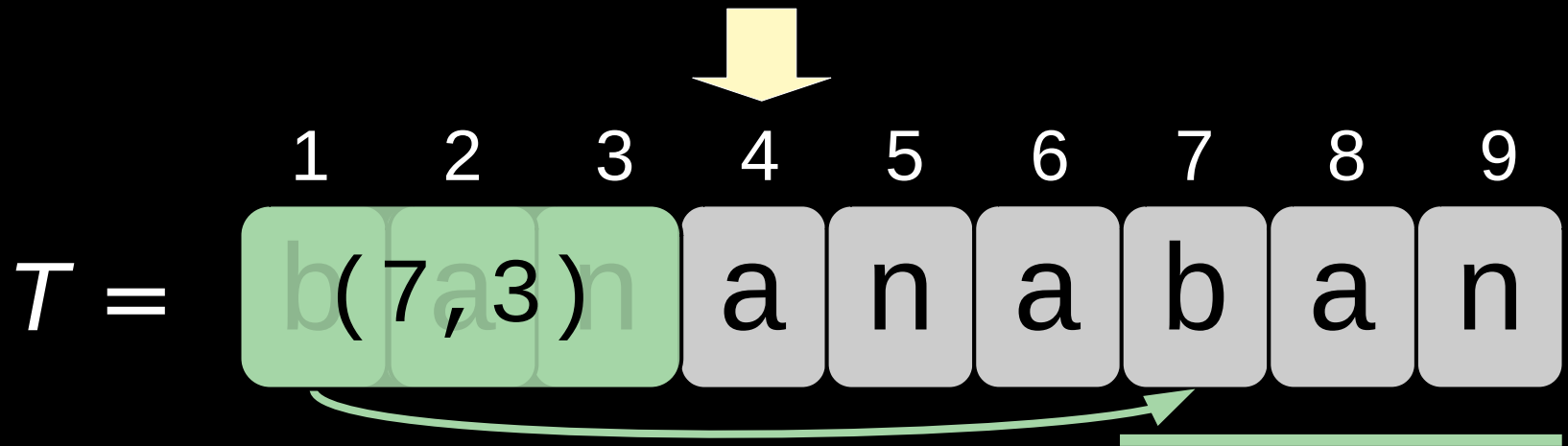
- T を左から右までスキャンする
- $\pi[i]$ から始まる項を計算するとき、
 - $\pi[i..]$ の辞書式順序で直前にある接尾辞 $\pi[j..]$ を選び、
 - j を参照先にし、
 - $\pi[i..]$ と $\pi[j..]$ の最長共通接頭辞を項の長さとする

lexparse



lexparse

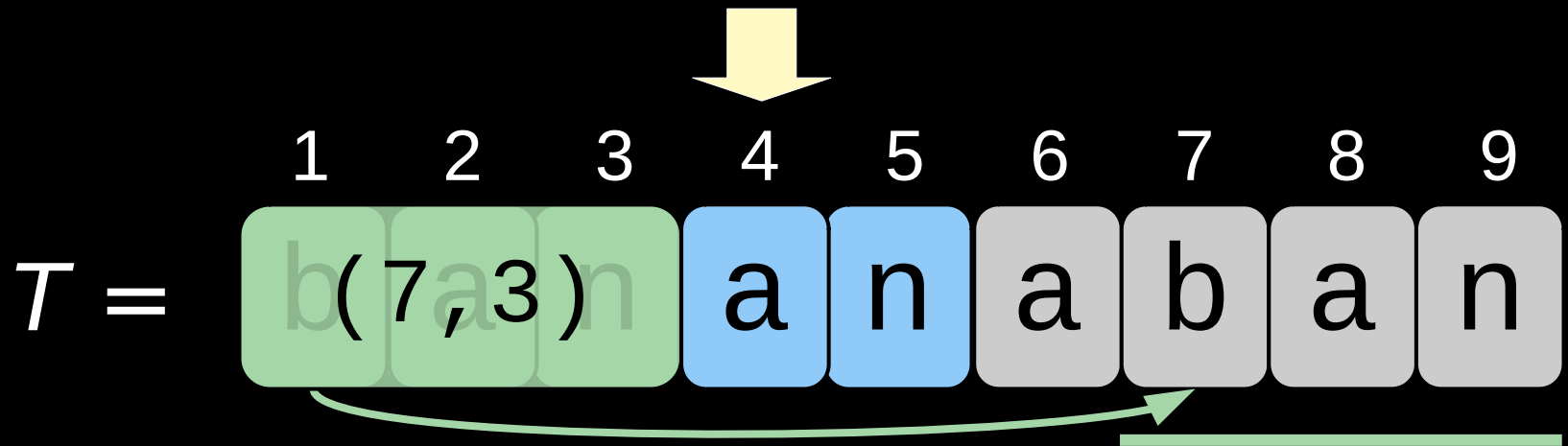
- $T[1..] < T[7..]$ かつ
- $T[1..] < T[j..] < T[7..]$
を満たす j が存在しない



位置 7 から 3 文字をコピーし、

lexparse

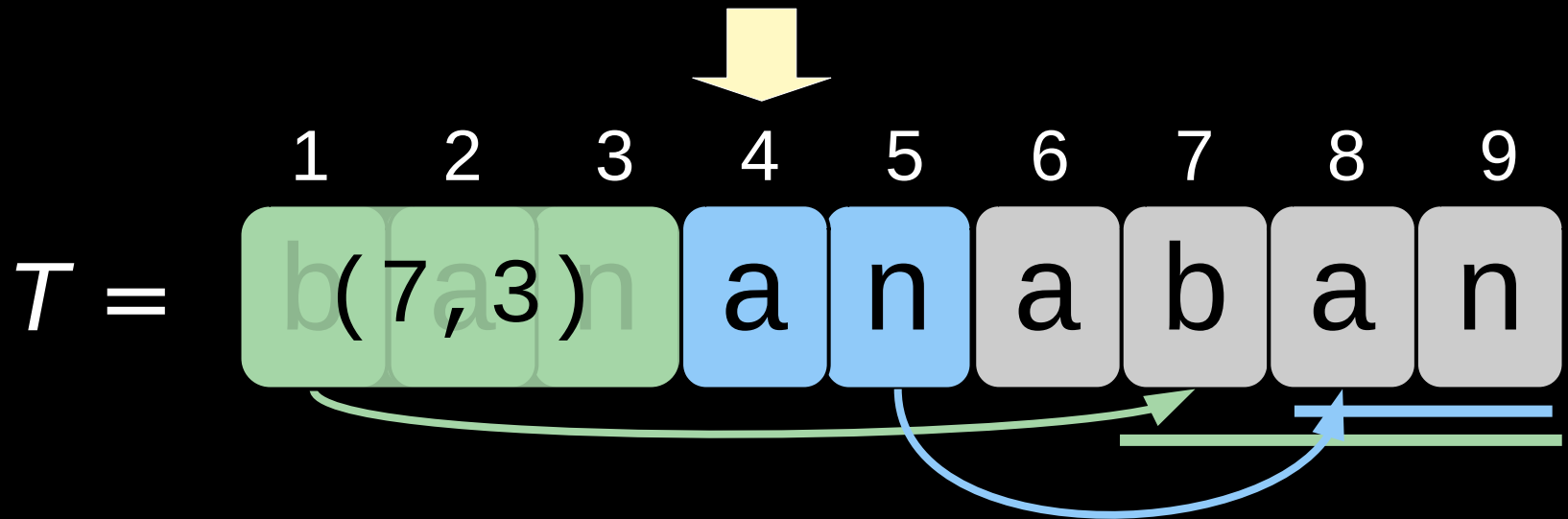
- $T[1..] < T[7..]$ かつ
- $T[1..] < T[j..] < T[7..]$
を満たす j が存在しない



位置 7 から 3 文字をコピーし、

lexparse

- $T[1..] < T[7..]$ かつ
- $T[1..] < T[j..] < T[7..]$
を満たす j が存在しない



位置 7 から 3 文字をコピーし、
位置 8 から 2 文字をコピーする

復元

lexparse が閉路を作らない理由

- 参照先は必ず辞書式順序でより早い接尾辞の開始位置を選ぶ
- 辞書式順序はすべての接尾辞に一意の順位を割り合てる
(ように、全順序;さらに推移律を利用できる)

[Dinklage+ '17]

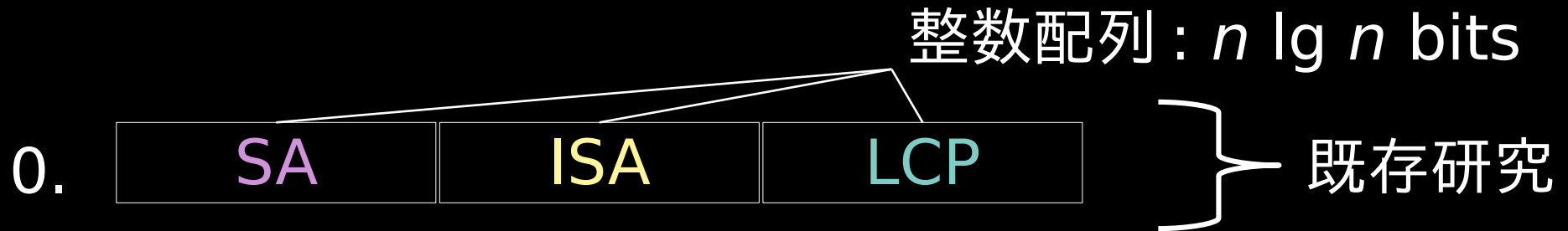
発表の目的

質問

$O(n)$ 時間で、
lexparse をどのぐらいの領域で計算できる？

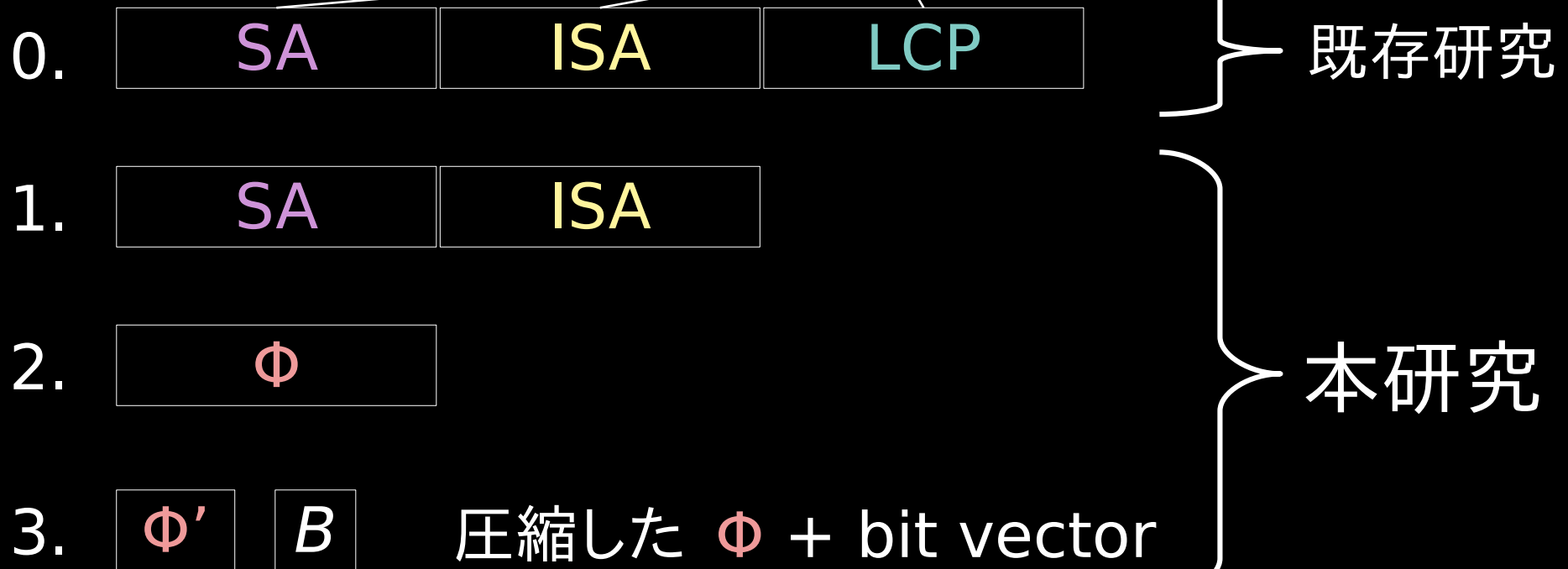
既存研究: $O(n)$ 時間 $O(n \log n)$ bits 領域

発表の目的



発表の目的

整数配列 : $n \lg n$ bits



$r \lg n + n + o(n)$ bits。ただし、 r は Burrows-Wheeler transform の文字の連の個数を示す

SA の定義

接尾辞の順序

1 2 3 4 5 6 7 8 9

$T = b a n a n a b a n$

接尾辞の順序

1 2 3 4 5 6 7 8 9

$T =$ b a n a n a b a n

b a n a n a b a n

a n a n a b a n

n a n a b a n

a n a b a n

n a b a n

a b a n

b a n

a n

n

接尾辞の順序

1 2 3 4 5 6 7 8 9

$T =$ b a n a n a b a n

表示のため、すべての接
尾辞を左に揃える

b a n a n a b a n

1 b a n a n a b a n

a n a n a b a n

2 a n a n a b a n

n a n a b a n

3 n a n a b a n

a n a b a n

4 a n a b a n

n a b a n

5 n a b a n

a b a n

6 a b a n

b a n

7 b a n

a n

8 a n

n

9 n

接尾辞の順序

辞書式順序でソート



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

接尾辞配列 SA

接尾辞の開始位置を格納する

接尾辞配列 SA

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



6
8
4
2
7
1
9
5
3

SA の構築

- すべての接尾辞の列挙は $\Omega(n^2)$ 時間が必要
- 列挙せず、 $O(n)$ 時間で SA を構築できるアルゴリズムもある
[Ko, Aluru '05]

接尾辞配列 SA

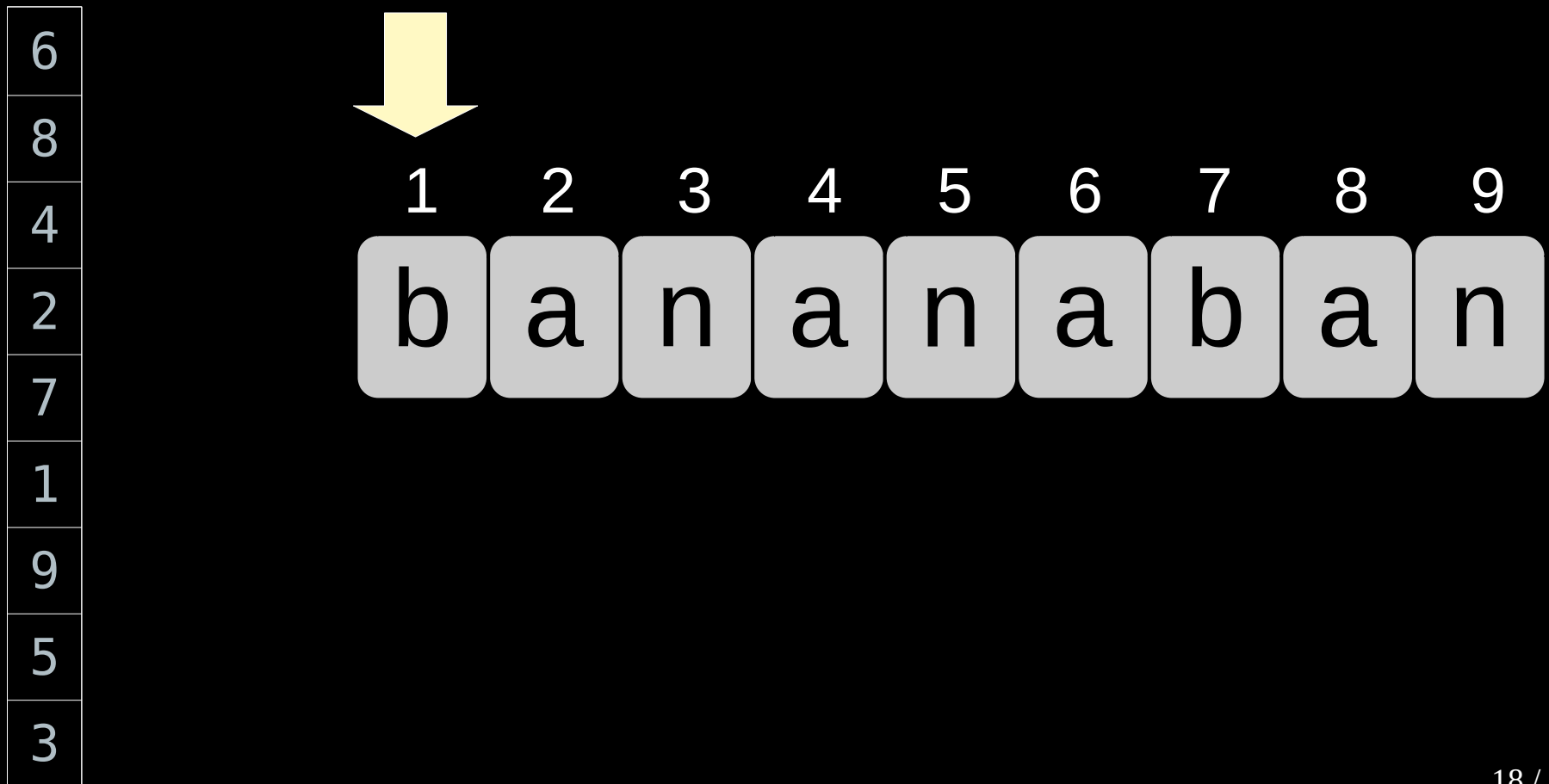
6
8
4
2
7
1
9
5
3

既存研究

$O(n)$ 時間 + $O(n \log n)$ bits で
lexparse の構築アルゴリズム

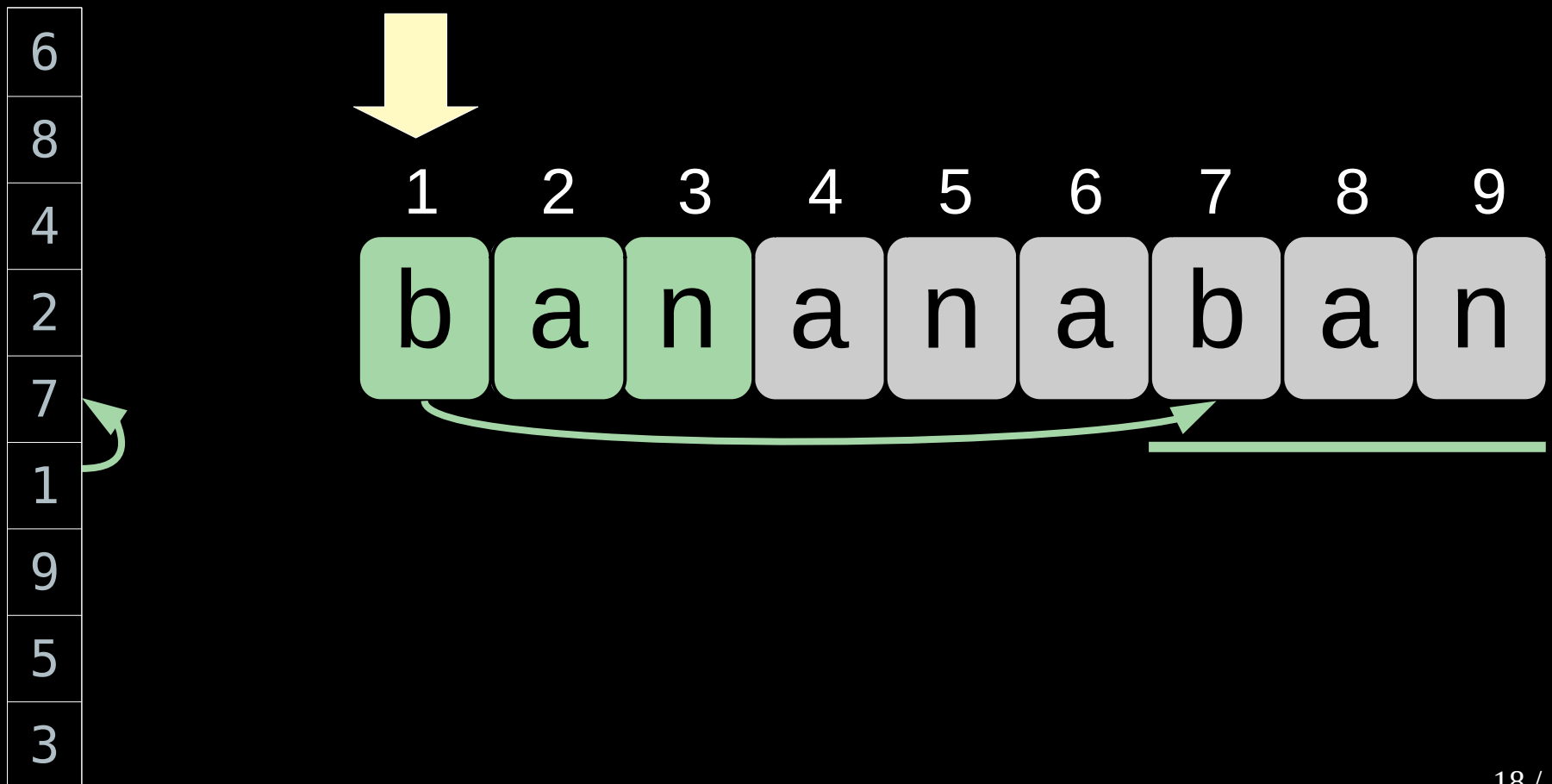
SA のもとで lexparse の計算

接尾辞配列 SA



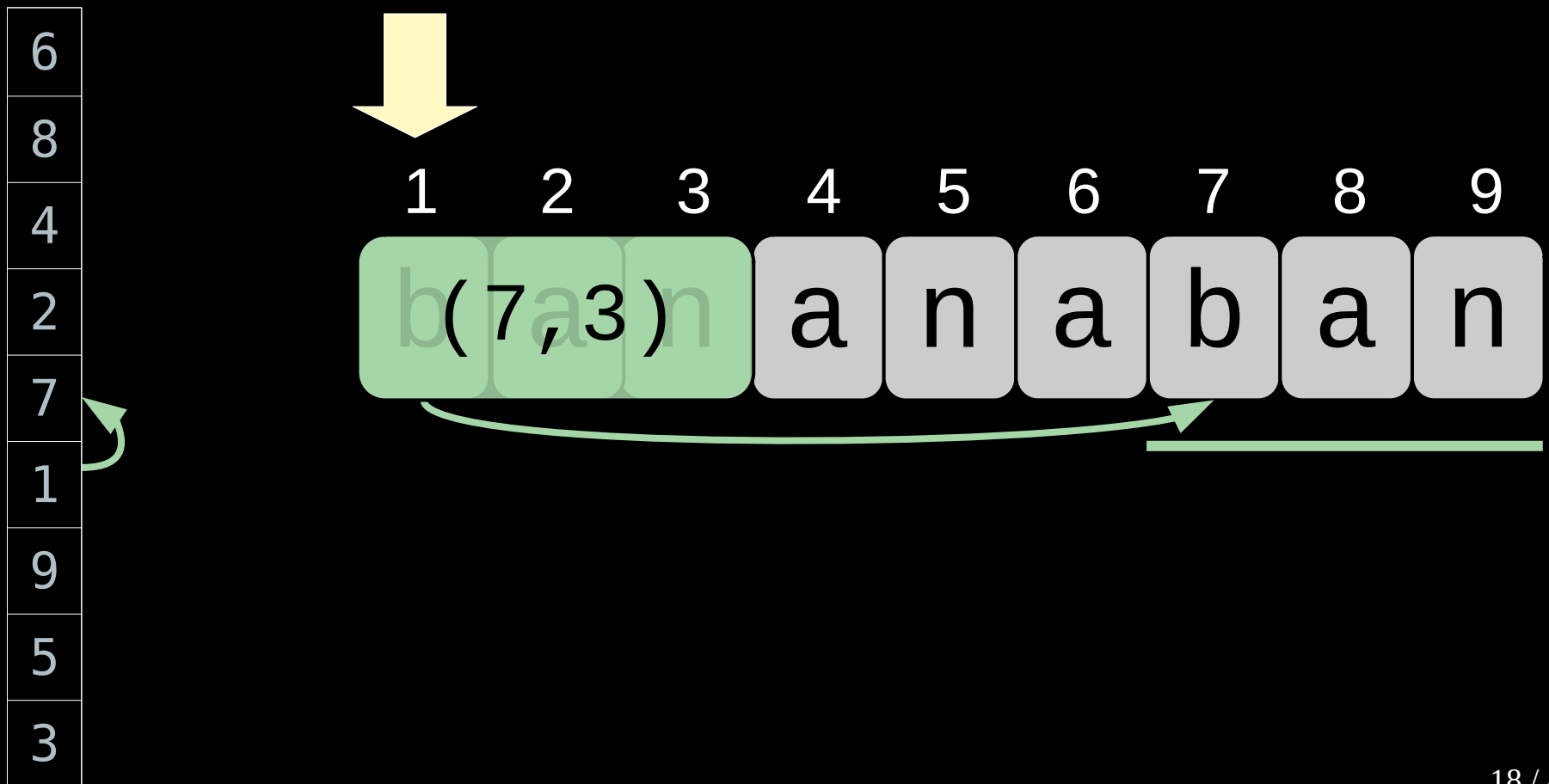
SA のもとで lexparse の計算

接尾辞配列 SA



SA のもとで lexparse の計算

接尾辞配列 SA



ISA / LCP

- $T[i]$ から始まる項を計算するため、 $i = SA[p]$ となる p が必要
- 逆接尾辞配列 **ISA** は $SA[ISA[i]] = i$ を満たすので、
 $ISA[i] = p$
⇒ 参照先は $SA[p-1] = SA[ISA[i]-1]$ になる
- 項の長さを **LCP** 配列で定める
LCP 配列は、任意の p に対し、 $LCP[p]$ に $T[SA[p]..]$ と $T[SA[p-1]..]$ の最長共通接頭辞の長さを格納する
⇒ $LCP[ISA[i]] = LCP[p]$ は項の長さとなる

既存のアルゴリズム

- SA, ISA, LCP を $O(n)$ 時間で構築できる
- i から始まる項を $O(1)$ 時間で定める:
 - 参照先: $SA[ISA[i] - 1]$
 - 長さ: $\max(LCP[i], 1)$
- 合計 $O(n)$ 時間
- 疑似コード:
 $i = 1; \text{while } i < n :$
 - if $LCP[i] = 0$: 文字 $T[i]$ を出力する; $i \leftarrow i + 1$
 - else: 項 $(SA[ISA[i]-1], LCP[i])$ を出力する; $i \leftarrow i + LCP[i]$

[Navarro+ '21]

既存のアルゴリズム

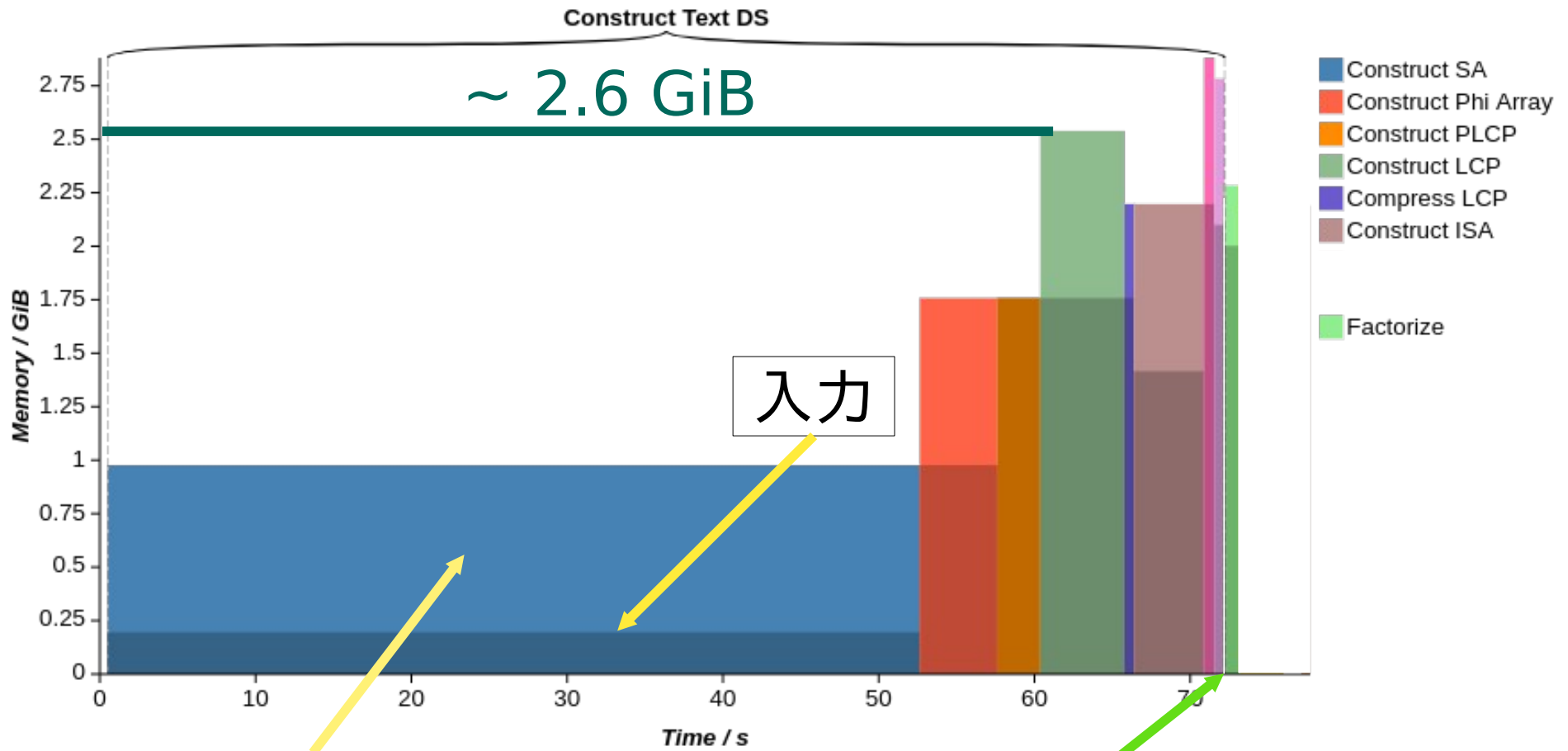
- [Navarro+ '21]: $O(n)$ 時間,
3つの整数配列 (SA, ISA, LCP)

具体的な設定:

- 入力の文字: 1 byte (8 bit)
- 整数配列の要素 : 4 byte (32 bit)
- 200 MiB 入力データに対し、2.6 GiB RAM が
必要

(1 MiB = 1024^2 byte)

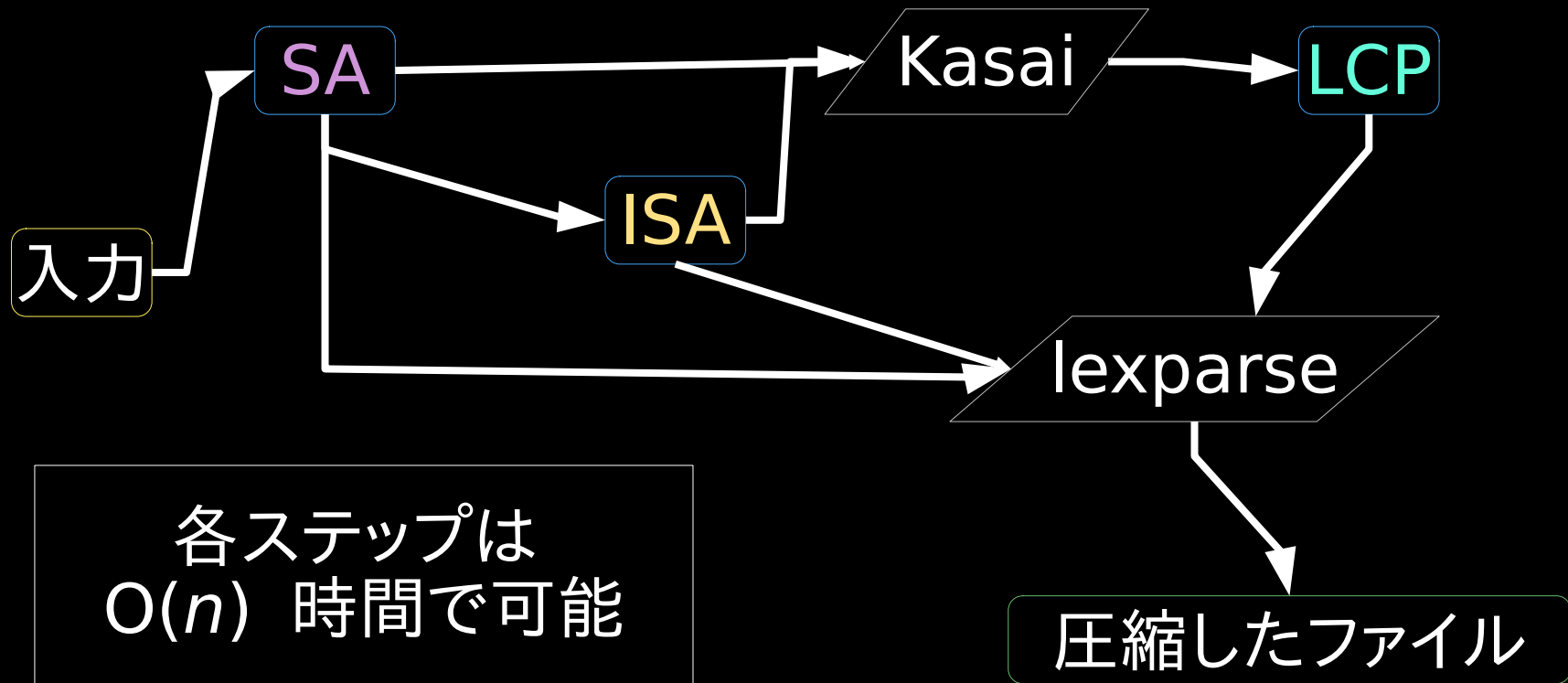
200 MiB ASCII web pages



SA 構築アルゴリズムの追加領域 (SA が含まれる)

lexparse のアルゴリズム

データ構造とアルゴリズム

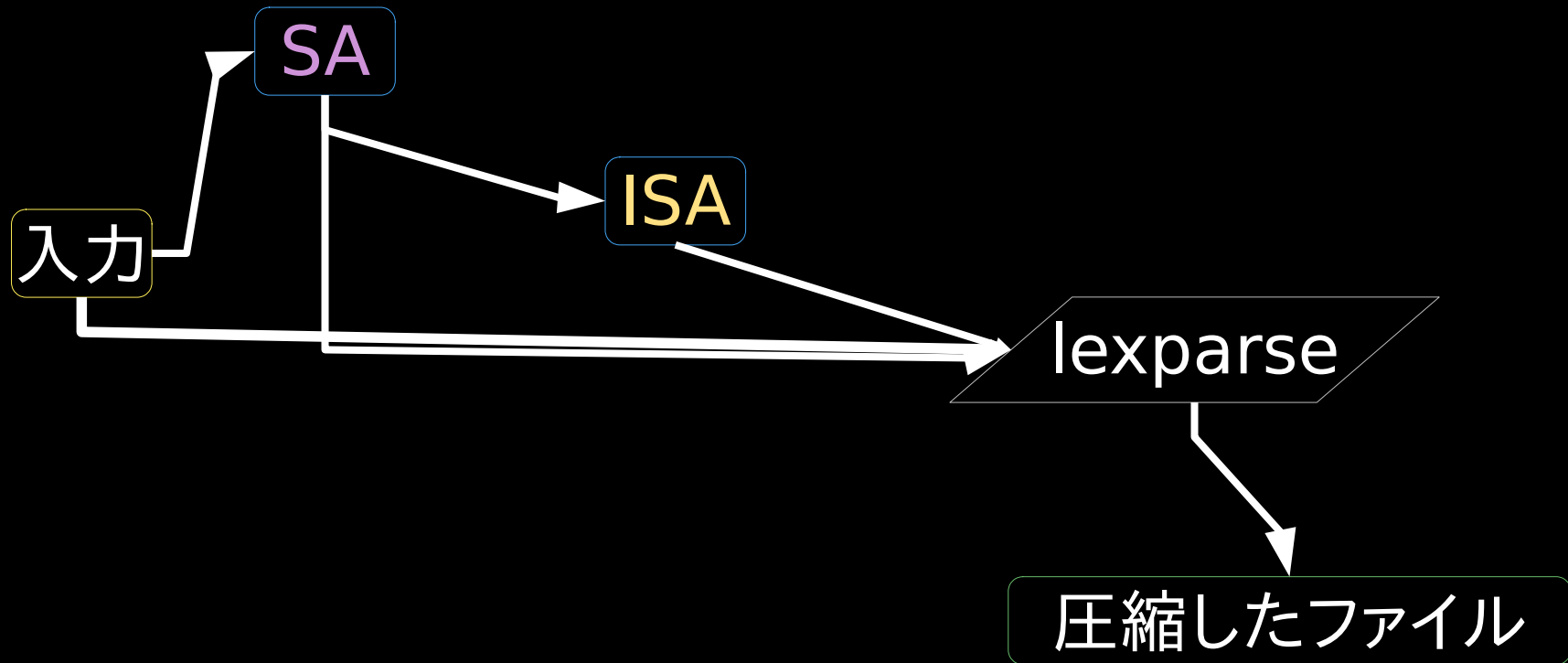


Kasai+ '01: LCP 配列の計算アルゴリズム

省メモリへ

- LCP 配列を利用せずに、素朴に $T[i..]$ と $T[SA[ISA[i]-1]..]$ との最長の接頭辞を計算できる
- 項 F_x の長さを $|F_x|$ とすると、 $\sum_x |F_x| = n$
 $\Rightarrow O(n)$ 時間を維持できる

データ構造とアルゴリズム



- SA / ISA はまだ必要？

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	↑								
	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n

The diagram illustrates the relationship between the ISA, SA, and Phi arrays for the string "banana". The ISA array (6, 4, 9, 3, 8, 1, 5, 2, 7) maps each character to its position in the SA array. The SA array (6, 8, 4, 2, 7, 1, 9, 5, 3) lists the sorted positions of the characters. The Phi array (7, 4, 5, 8, 9, -, 2, 6, 1) is derived from SA[ISA[i] - 1]. A yellow arrow points from the value 6 in the ISA array at index 1 to the value 6 in the SA array at index 6, which corresponds to the character 'a' at index 6 in the string "banana".

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9

b a n a n a b a n

Φ 配列

$$\Phi[i] := SA[ISA[i] - 1]$$

Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9

b a n a n a b a n

アルゴリズムの方針

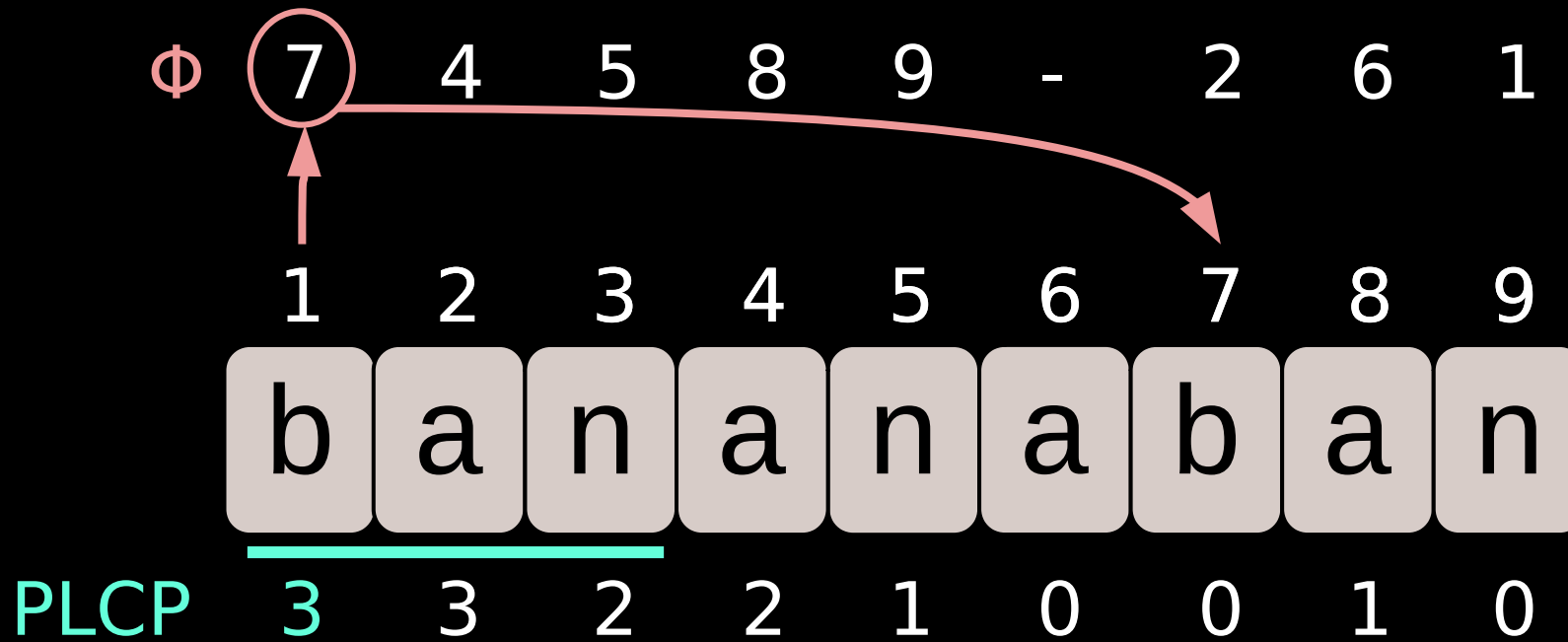
Φ 7 4 5 8 9 - 2 6 1

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

PLCP 3 3 2 2 1 0 0 1 0

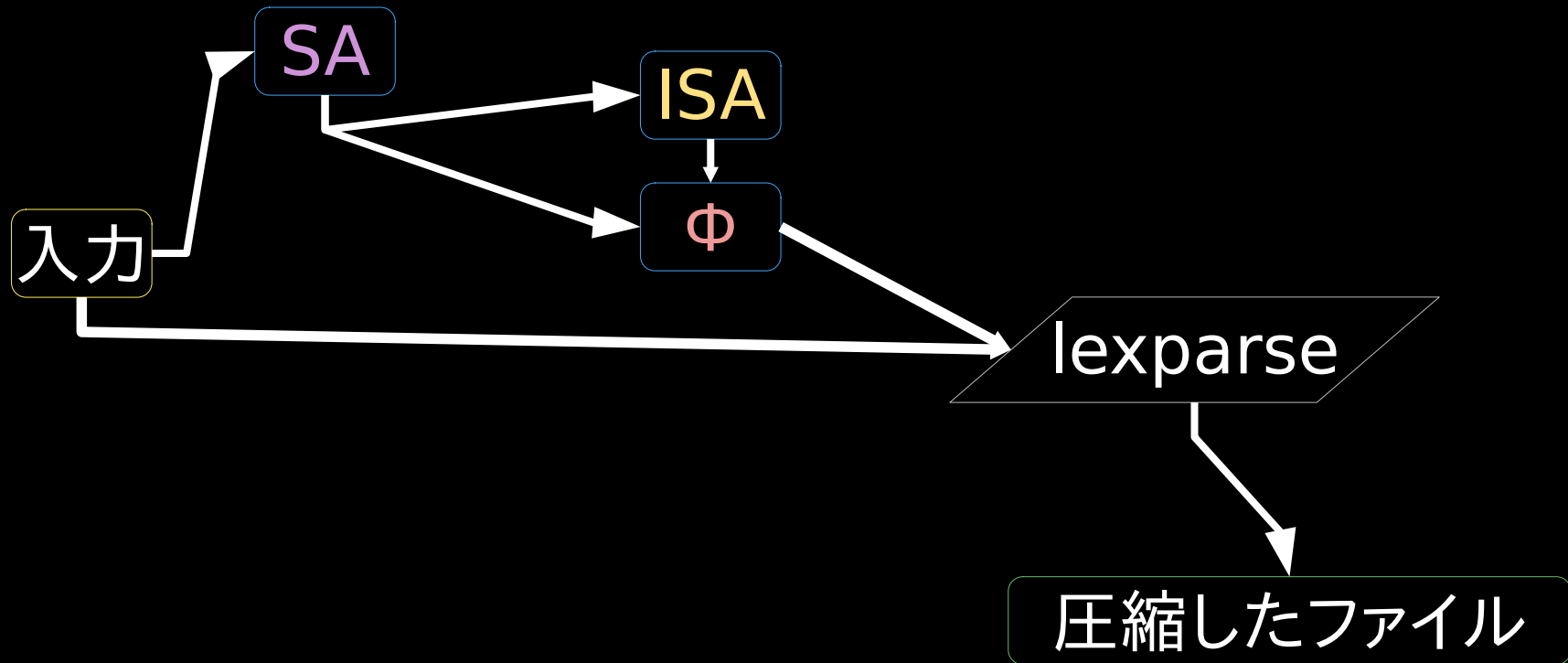
- $PLCP[i] = LCP[ISA[i]]$: 参照の長さ
- $\Phi[i]$: 参照先

アルゴリズムの方針

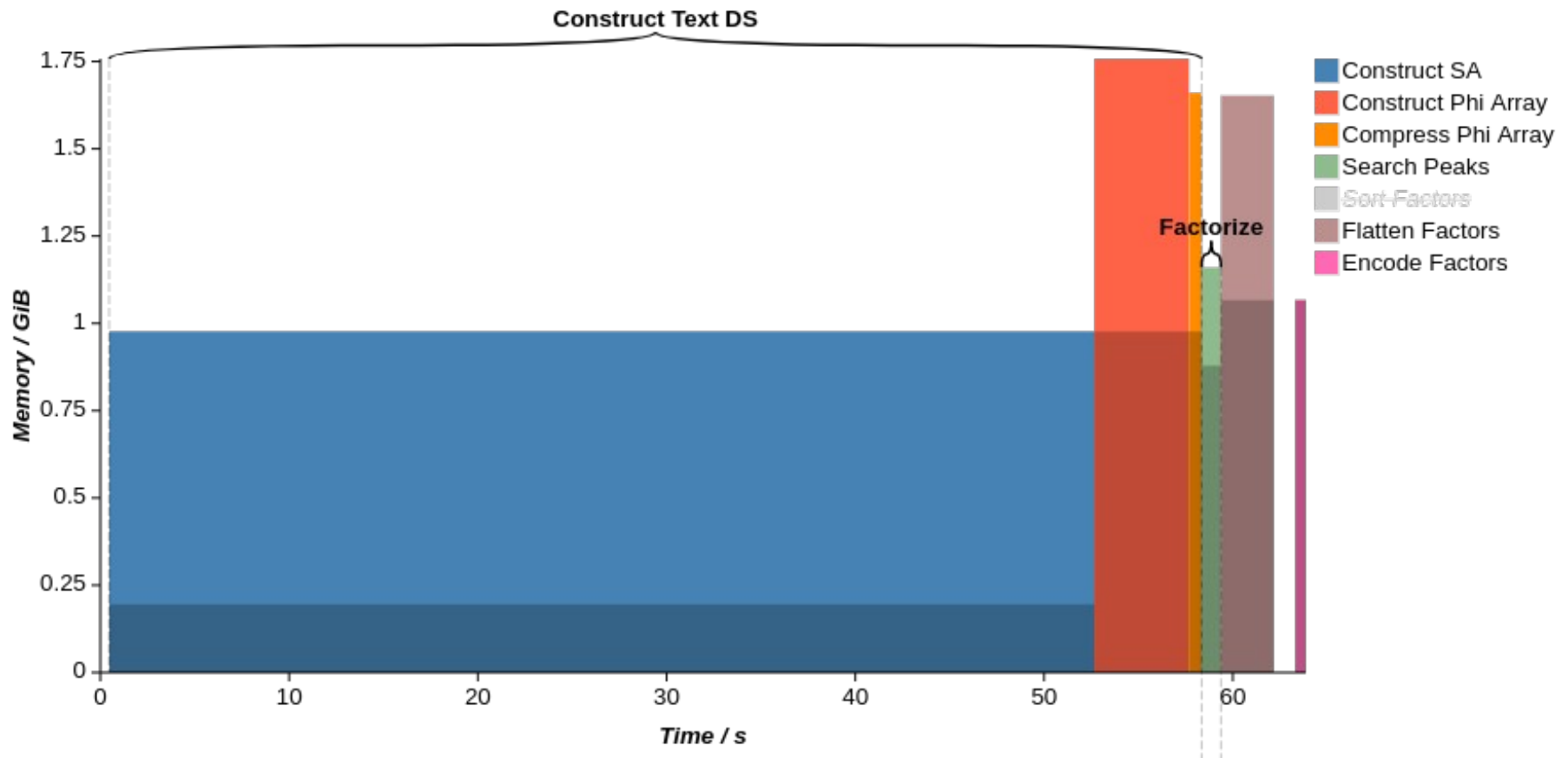


- $PLCP[i] = LCP[ISA[i]]$: 参照の長さ
- $\Phi[i]$: 参照先

データ構造とアルゴリズム

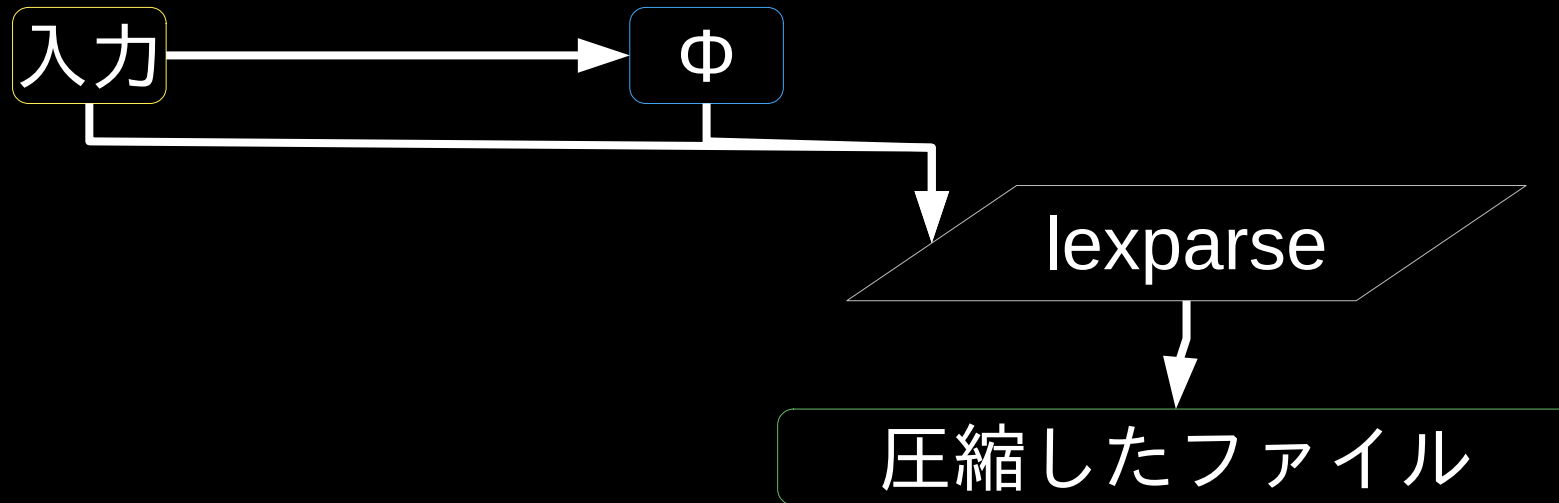


200 MiB ASCII web pages



39% のメモリーを節約できた

データ構造



[Goto, Bannai '14]: テキストから直接に Φ を

- $O(\sigma \lg n)$ 追加領域で
- $O(n)$ 時間で

計算できる

前処理：最大の領域

lexparse の前処理それぞれの最大実践的な領域：

0) SA+ISA+LCP : 2.88 GiB

1) SA+ISA→ Φ : 1.76 GiB

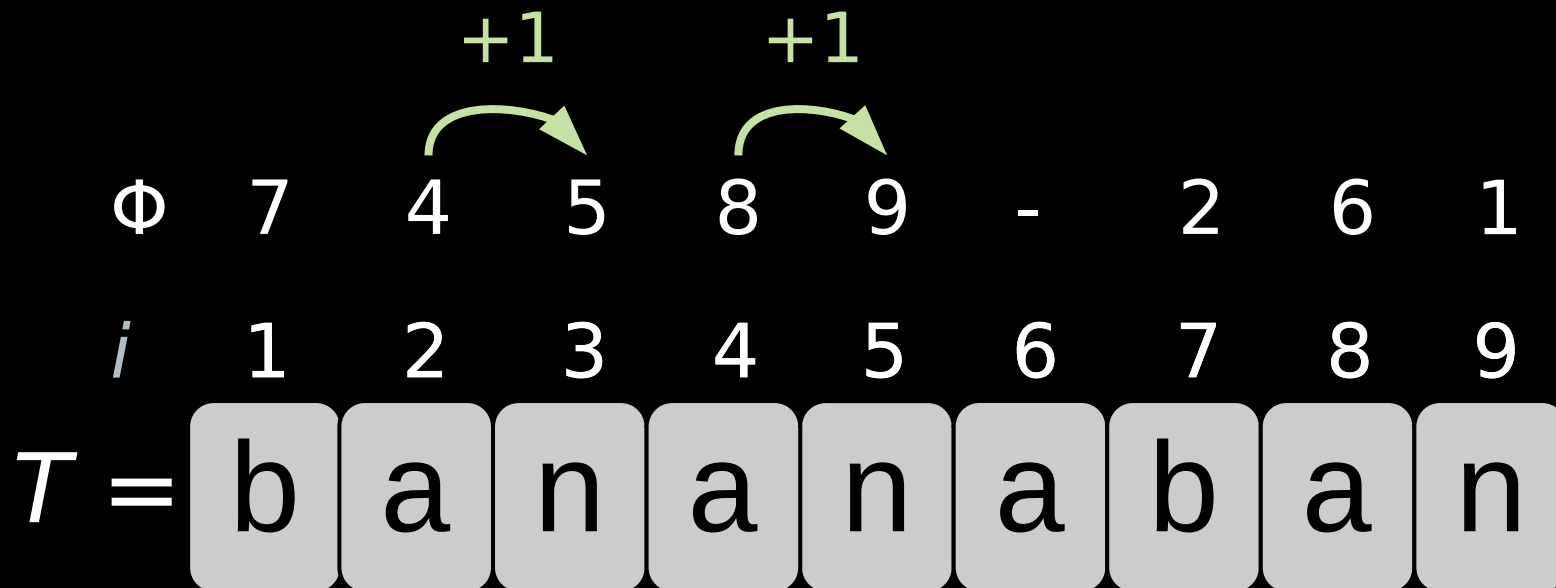
2) Φ のみ : ~ 1 GiB

すべては線形時間であるが、2) は 0) の 35% の領域で済む

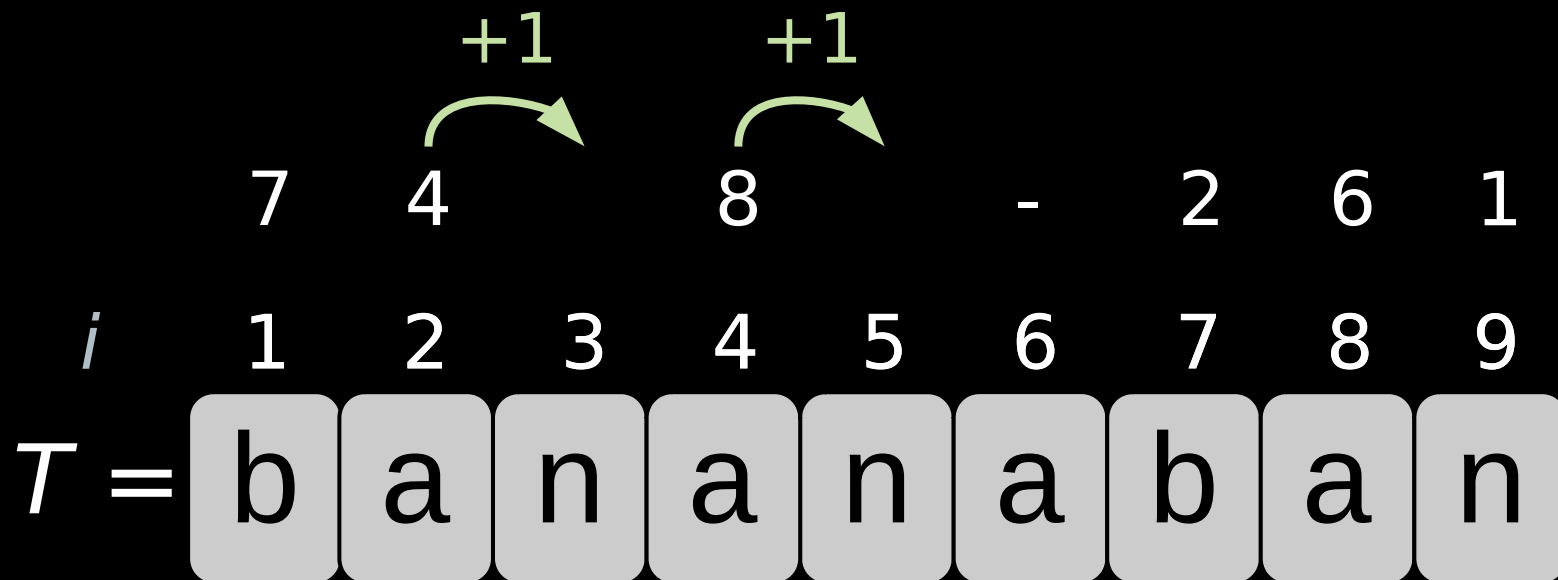
Φ の圧縮表現

$\Phi[i] = \Phi[i-1] + 1$ という要素が多い
(反復が原因)

⇒ 別の表現で圧縮できる



Φ' : Φ の圧縮表現



Φ の圧縮表現

bit vector

B 1 1 0 1 0 1 1 1 1

Φ' 7 4 8 - 2 6 1 ← 左揃え


7 4 8 - 2 6 1

i 1 2 3 4 5 6 7 8 9

$T =$ **b** **a** **n** **a** **n** **a** **b** **a** **n**

Φ の圧縮表現

query $\Phi[j]$,
 $j = 4$



B	1	1	0	1	0	1	1	1	1
Φ'	7	4	8	-	2	6	1		
Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9

- $B[j] = 1$ なら、 $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
- ただし、 $B[1..j]$ の中の '1' を数える計算は $\text{rank}_1(j)$ 関数を呼ぶ

Φ の圧縮表現

query $\Phi[j]$,
 $j = 4$

$B[1..4]$ の中に3つ'1'がある

B	1	1	0	1	0	1	1	1	1
Φ'	7	4	8	-	2	6	1		
Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9

- $B[j] = 1$ なら、 $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
- ただし、 $B[1..j]$ の中の'1'を数える計算は $\text{rank}_1(j)$ 関数を呼ぶ

Φ の圧縮表現

query $\Phi[j]$,
 $j = 4$

$B[1..4]$ の中に3つ'1'がある

B	1	1	0	1	0	1	1	1	1
Φ'	7	4	8	-	2	6	1		
Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9

- $B[j] = 1$ なら、 $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
- ただし、 $B[1..j]$ の中の'1'を数える計算は $\text{rank}_1(j)$ 関数を呼ぶ

Φ の圧縮表現

query $\Phi[j]$,
 $j = 3$

$B[1..3]$ の中に2つ'1'がある

B	1	1	0	1	0	1	1	1	1
Φ'	7	4	8	-	2	6	1		
Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9

- $B[j] = 0$ なら、
 $\Phi[j] = \Phi'[B.\text{rank}_1(j)] +$
 $B.\text{rank}_0(j) - B.\text{rank}_0(B.\text{select}_1(B.\text{rank}_1(j)))$
- ただし、 $B.\text{select}_1(k)$ は B の k 番目の'1'の位置を示す

Φ の圧縮表現

query $\Phi[j]$,
 $j = 3$

$B[1..3]$ の中に2つ'1'がある

B	1	1	0	1	0	1	1	1	1
Φ'	7	4	8	-	2	6	1		
Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9

- $B[j] = 0$ なら、
 $\Phi[j] = \Phi'[B.rank_1(j)] +$
 $B.rank_0(j) - B.rank_0(B.select_1(B.rank_1(j)))$
- ただし、 $B.select_1(k)$ は B の k 番目の'1'の位置を示す

rank / select

bit vector $B[1..n]$ 上で rank/select データ構造を構築できる

- $O(n)$ 時間で構築できる
- rank / select を $O(1)$ 時間で答える
- $n + o(n)$ ビットを取る (B が含まれる)

[Jacobson '89, Clark '96]

領域の解析

- $\Phi[i] = \Phi[i-1] + 1$ を満たさない位置 i の個数は r
ただし、 r は Burrows-Wheeler transform の
文字の連の個数を示す
[Kärkkäinen+ '16]
- $r \lg n + n + o(n)$ bits で表現できる

まとめ

lexparse の新しい構築手法の考案:

- Φ 配列だけで、線形時間で計算できる
- $r \lg n + n + o(n)$ bits で表現できる

今後の課題

Φ の圧縮表現をテキストから直接計算できる?

実装: <https://tudocomp.github.io>

ご清聴ありがとうございました

v 对 z

file	n	r	z	v	z/v	r/v
fib41	267, 914, 296	4	41	4	> 10	1.000
rs.13	216, 747, 218	77	52	40	1.300	1.925
tm29	268, 435, 456	82	56	43	1.302	1.907
dblp.xml.00001.base	104, 857, 600	172, 489	59, 573	59, 821	0.996	2.883
dblp.xml.00001.prev	104, 857, 600	175, 617	59, 556	61, 580	0.967	2.852
dblp.xml.0001.base	104, 857, 600	240, 535	78, 167	83, 963	0.931	2.865
dblp.xml.0001.prev	104, 857, 600	270, 205	78, 158	100, 605	0.777	2.686
sources.001.prev	104, 857, 600	1, 213, 428	294, 994	466, 643	0.632	2.600
dna.001.base	104, 857, 600	1, 716, 808	308, 355	307, 329	1.003	5.586
proteins.001.base	104, 857, 600	1, 278, 201	355, 268	364, 093	0.976	3.511
english.001.prev	104, 857, 600	1, 449, 519	335, 815	489, 034	0.687	2.964
boost	500, 000, 000	61, 814	22, 680	22, 418	1.012	2.757
einstein.de	92, 758, 441	101, 370	34, 572	37, 721	0.917	2.687
einstein.en	467, 626, 544	290, 239	89, 467	97, 442	0.918	2.979
bwa	438, 698, 066	311, 427	106, 655	107, 117	0.996	2.907
sdsl	500, 000, 000	345, 325	113, 591	112, 832	1.007	3.061
samtools	500, 000, 000	458, 965	150, 988	150, 322	1.004	3.053
world_leaders	46, 968, 181	573, 487	175, 740	179, 696	0.978	3.191
influenza	154, 808, 555	3, 022, 822	769, 286	768, 623	1.001	3.933
kernel	257, 961, 616	2, 791, 368	793, 915	794, 058	1.000	3.515
cere	461, 286, 644	11, 574, 641	1, 700, 630	1, 649, 448	1.031	7.017
coreutils	205, 281, 778	4, 684, 460	1, 446, 468	1, 439, 918	1.005	3.253
escherichia_coli	112, 689, 515	15, 044, 487	2, 078, 512	2, 014, 012	1.032	7.470
para	429, 265, 758	15, 636, 740	2, 332, 657	2, 238, 362	1.042	6.986

v 対 plcpcomp

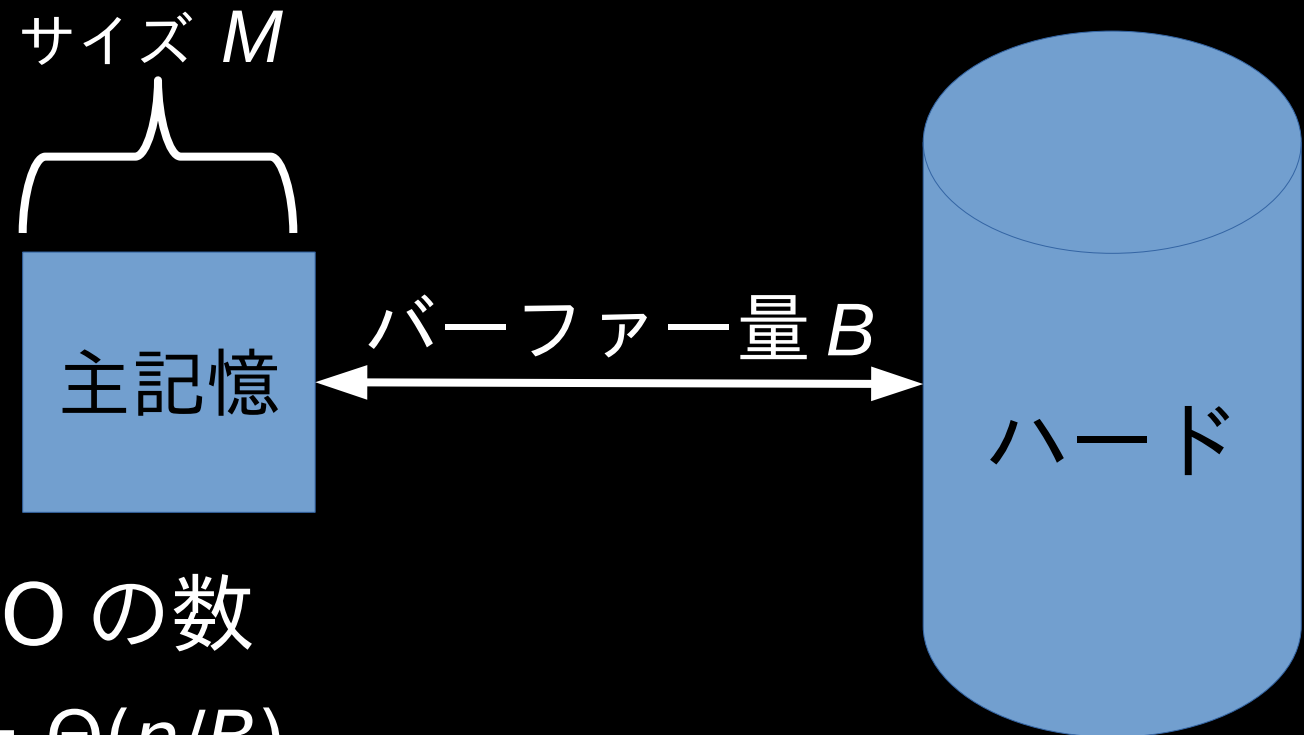
$$V = Z + Z_1$$

input		lex-parse				plcpcomp			
name	n [M]	z [M]	$\emptyset d$ [M]	$\emptyset l$	$\emptyset z_1$ [K]	z [M]	$\emptyset d$ [M]	$\emptyset l$	$\emptyset z_1$ [K]
ESCHERICHIA_COLI	112.7	2.01	33.4	12	0.1	2.02	33.4	13	287.8
CERE	461.3	1.65	132.2	12	0.0	1.63	132.3	13	245.1
COREUTILS	205.3	1.43	26.3	10	6.1	1.32	28.5	10	166.2
EINSTEIN.DE.TXT	92.8	0.04	19.0	6	1.3	0.03	18.0	7	6.5
EINSTEIN.EN.TXT	467.6	0.10	81.4	7	1.8	0.09	75.6	8	14.3
INFLUENZA	154.8	0.77	30.3	51	0.1	0.66	27.4	57	142.4
KERNEL	258.0	0.79	52.5	9	2.2	0.73	50.8	10	90.3
PARA	429.3	2.24	130.9	12	0.0	2.14	131.0	13	364.4
WORLD_LEADERS	47.0	0.18	5.7	11	0.9	0.17	5.7	12	19.5

plcpcomp: [Dinklage+ '17]

lexparse のように lexicographic parsing の一種

external memory



計算量: I/O の数

- $\text{scan}(n) = \Theta(n/B)$

- $\text{sort}(n) = \Theta\left(\frac{n}{B} \log_{M/B} \left(\frac{n}{B}\right)\right)$

EM での計算

- $\text{sort}(17n) + \text{scan}(9n)$ I/Os で SA を構築できる [Bingmann+ '13]
- SA $\rightarrow \Phi$
 - 各位置 j に対し、2つの組 $\langle \text{SA}[j], \text{SA}[j]-1 \rangle$ を格納し、
 - 一番目の成分をキーにとし、 $\text{sort}(n)$ でソートする
 - $\langle j, \text{ISA}[\text{SA}[j]-1] \rangle = \langle j, \Phi[j] \rangle$ を求める
- lexparse は Φ と T をスキャンする $\Rightarrow \text{scan}(2n)$

合計: $\text{sort}(18n) + \text{scan}(11n)$ I/Os

復元: [Dinklage' +19] によって、任意の bidirectional parse を EM で復元するアルゴリズムが提案された

LZ77 の関係

- v を lexparse の項の個数とする
- z : Lempel-Ziv 77 の項の個数
- b : 最小の bidirectional scheme の個数,
 $b \leq z, b \leq v$
- r : Burrows-Wheeler transform の文字の連の個数
 - $r = O(z \log^2 n)$ [Kempa, Kociumaka '21]
 - $v \leq 2r$ [Navarro+ '21]
- $z = O(r \log n)$ [Gagie+ '18]
- \exists 文字列集合 : $z = \Omega(v \log n)$
- \exists 文字列集合 : $z = o(v)$? [Navarro+ '21]

