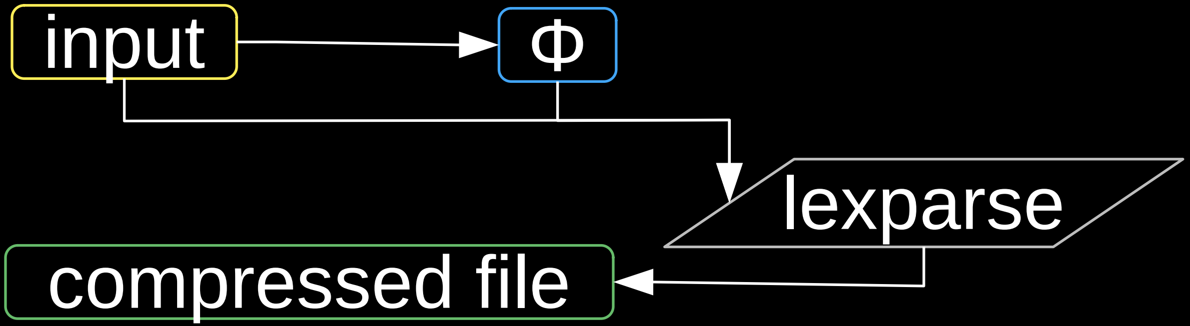


computing lexicographic parsings

Dominik Köppl
Tokyo Medical and Dental University

DCC 2022

lexparse

- text factorization $T = \boxed{F_1}\boxed{F_2}\boxed{\ldots}$
- usable for lossless text compression
- uses lexicographic order of suffixes of input text
- special kind of bidirectional parse [Storer, Szymanski 1978]
- introduced by Navarro+ ’21 (arXiv preprint: ‘18)

bidirectional parse

- factorizes T
- represent a factor $F = T[i..i+\ell\text{-}1]$ as
  - a single character ($\ell=1$), or
  - a pair (reference j 、length ℓ) where $F = T[j..j+\ell\text{-}1]$

example text

text T = bananaban 

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

bidirectional parse: example

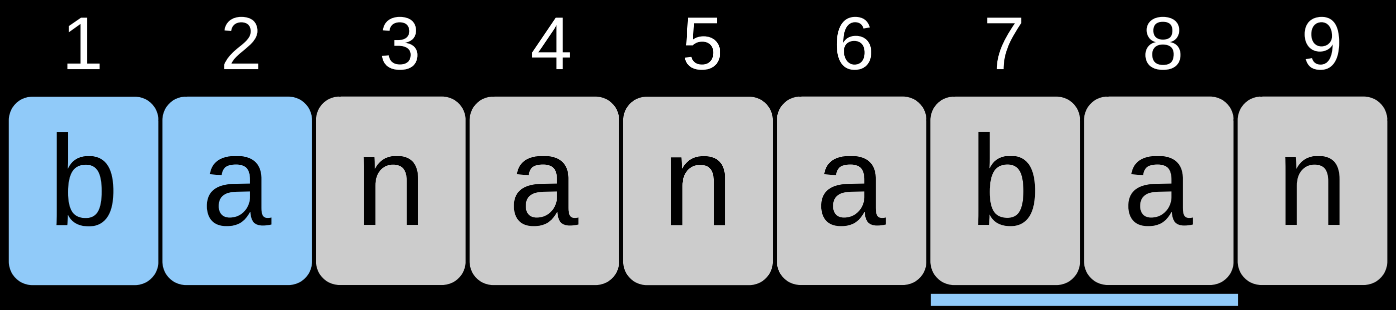
- replace factors by pair-representation

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

bidirectional parse: example

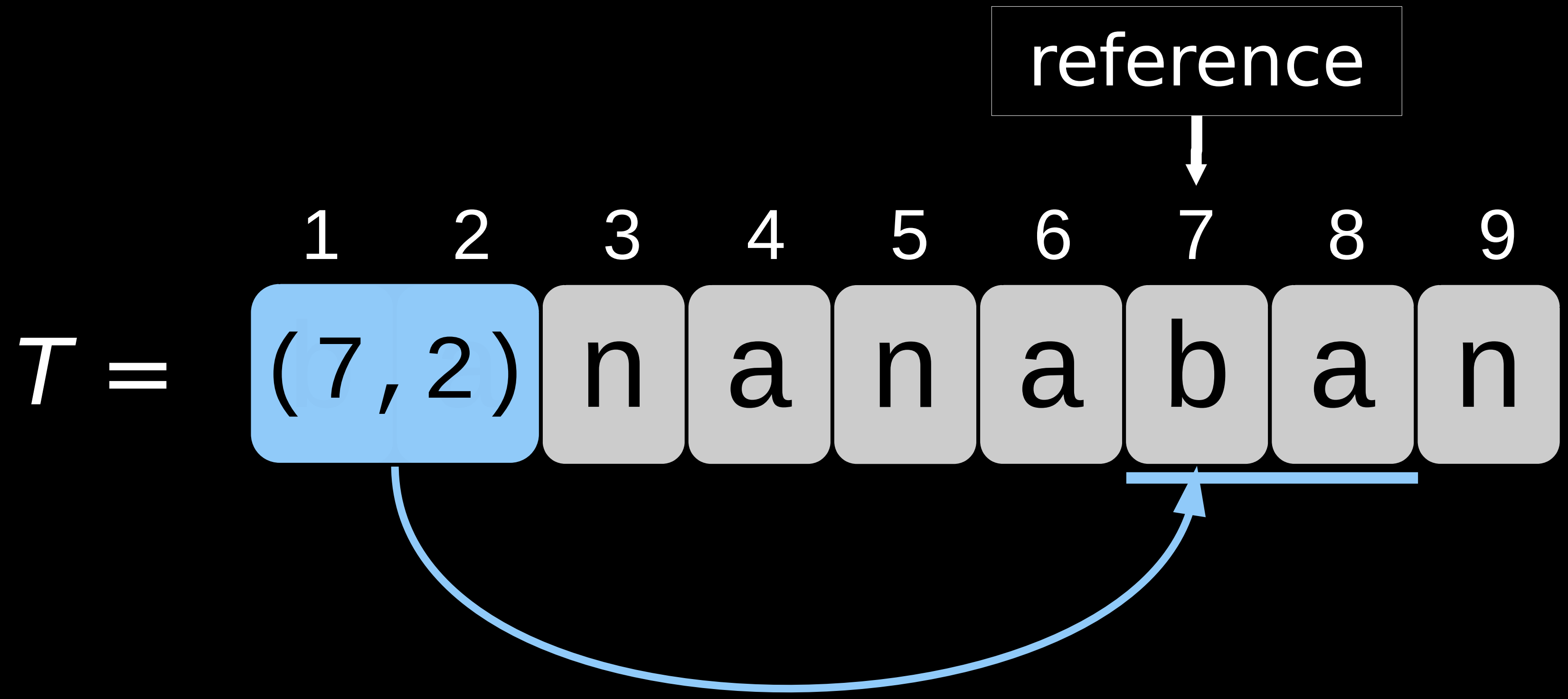
- replace factors by pair-representation

$T =$



bidirectional parse: example

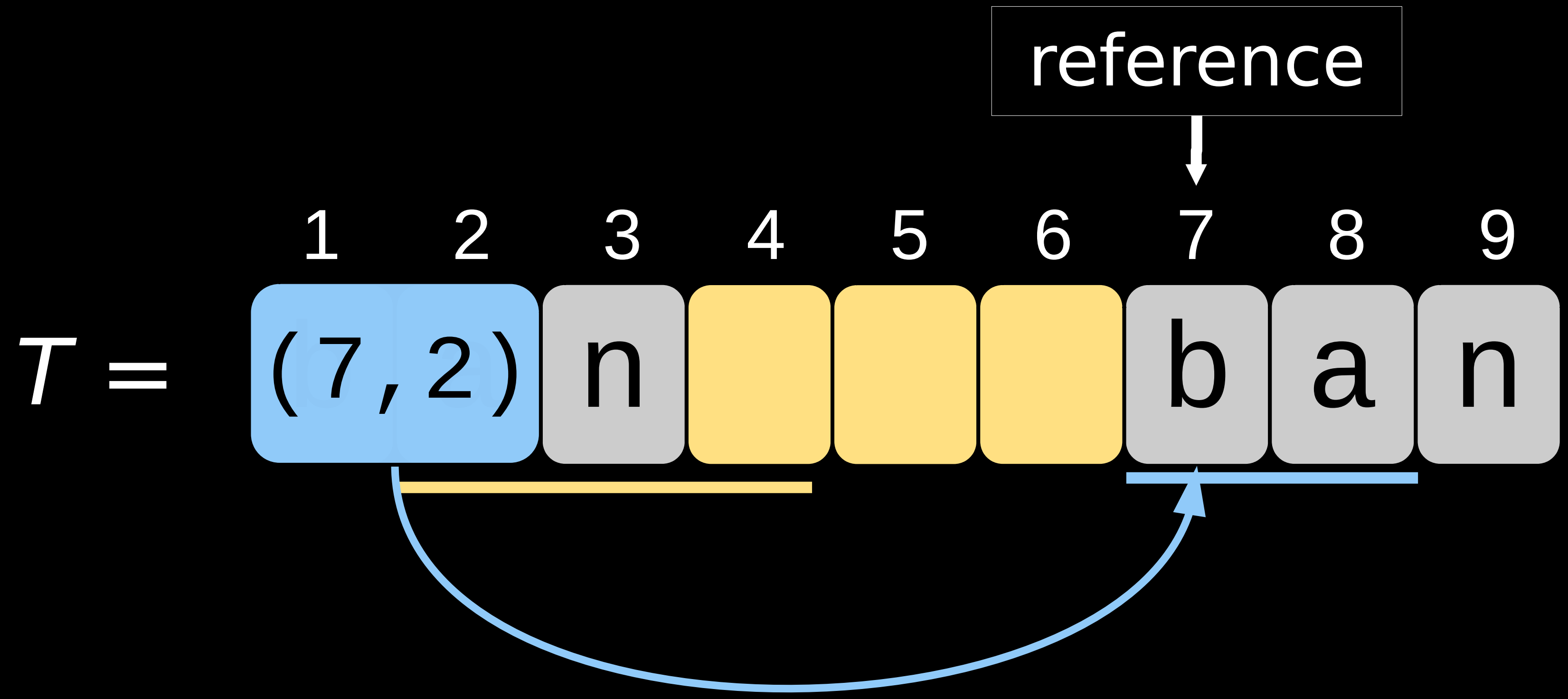
- replace factors by pair-representation



pairs (reference, length)

bidirectional parse: example

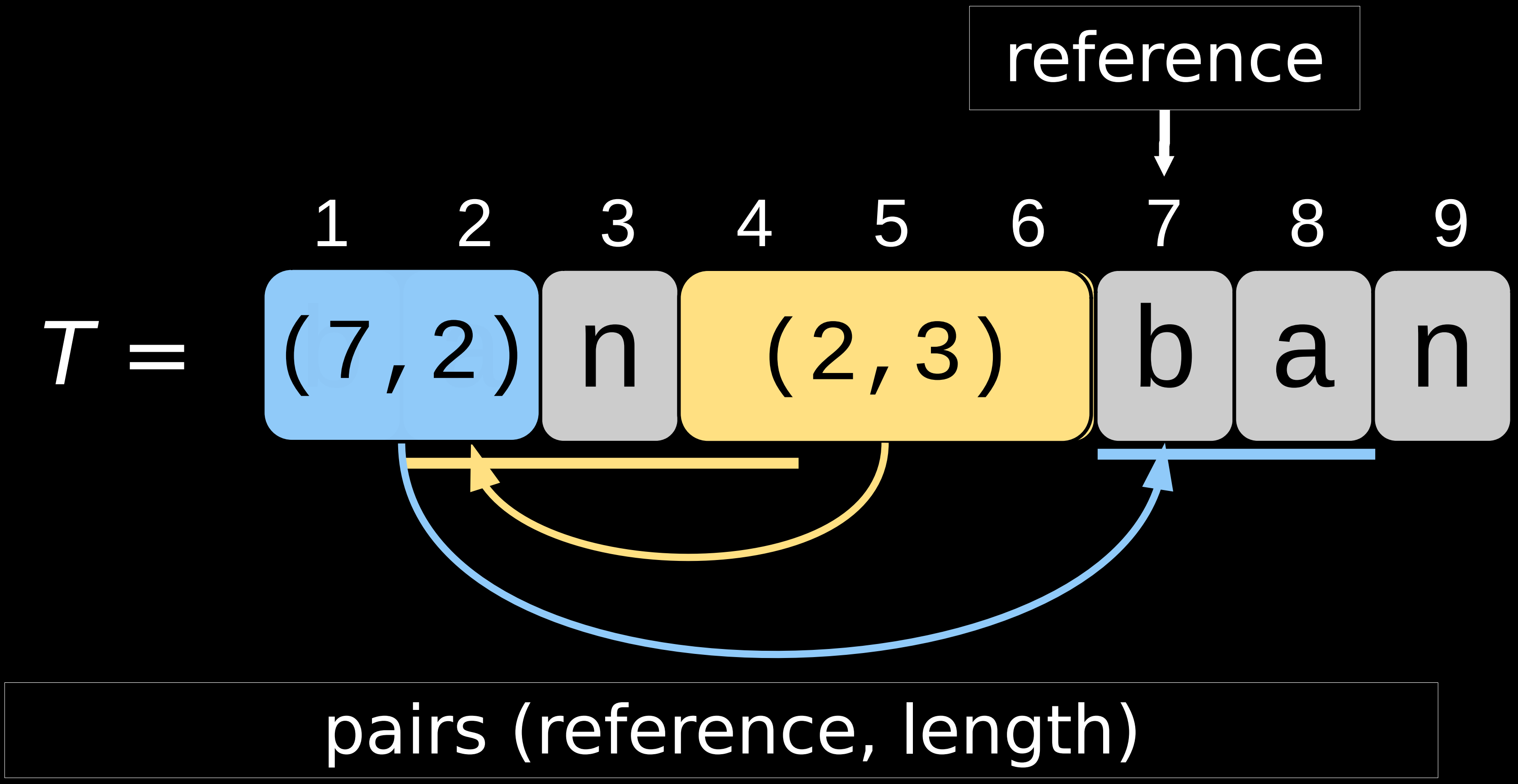
- replace factors by pair-representation
- self-references are allowed



pairs (reference, length)

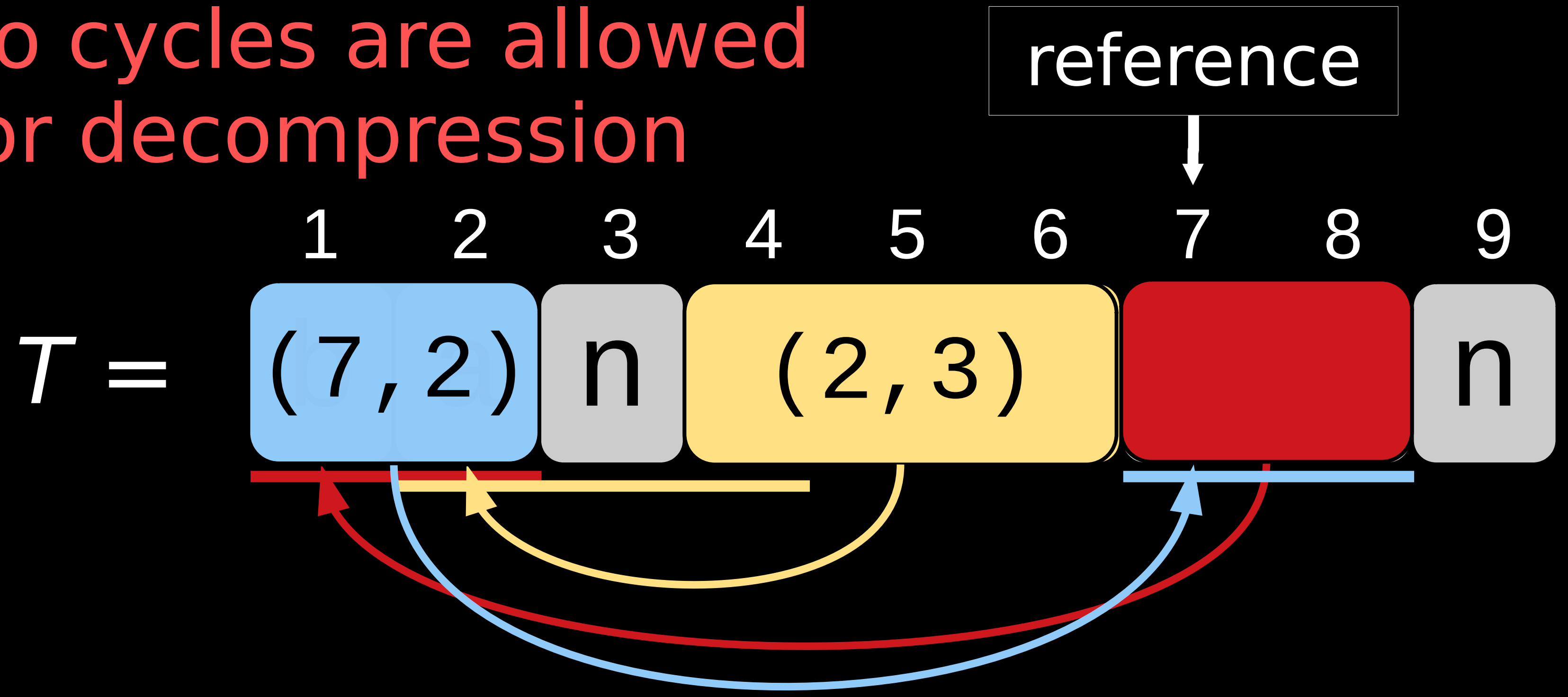
bidirectional parse: example

- replace factors by pair-representation
- self-references are allowed



bidirectional parse: example

- replace factors by pair-representation
- self-references are allowed
- no cycles are allowed for decompression

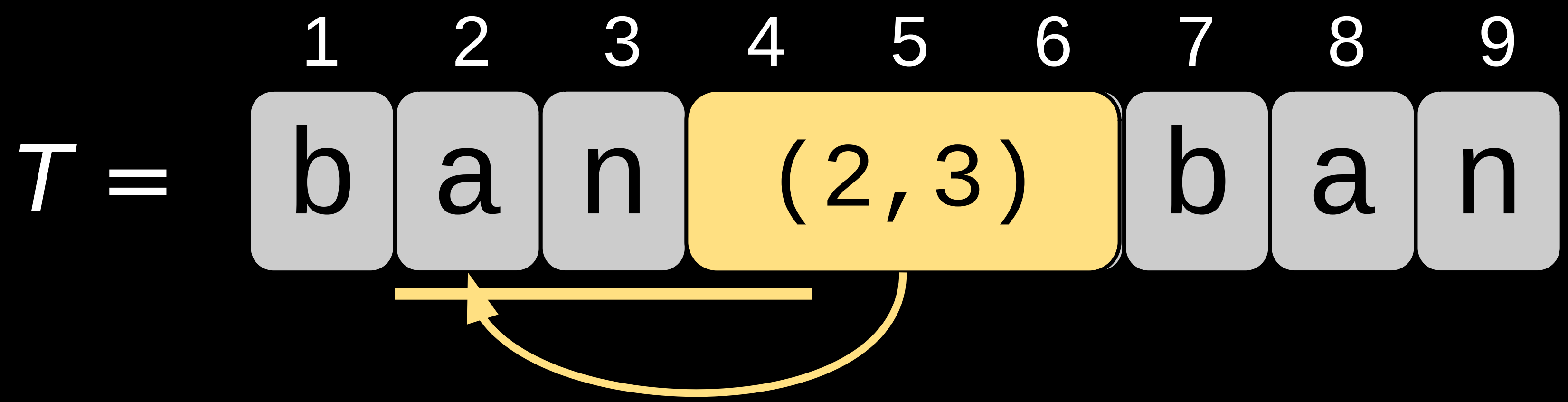


pairs (reference, length)

decompressing self-references

why are self-references allowed?

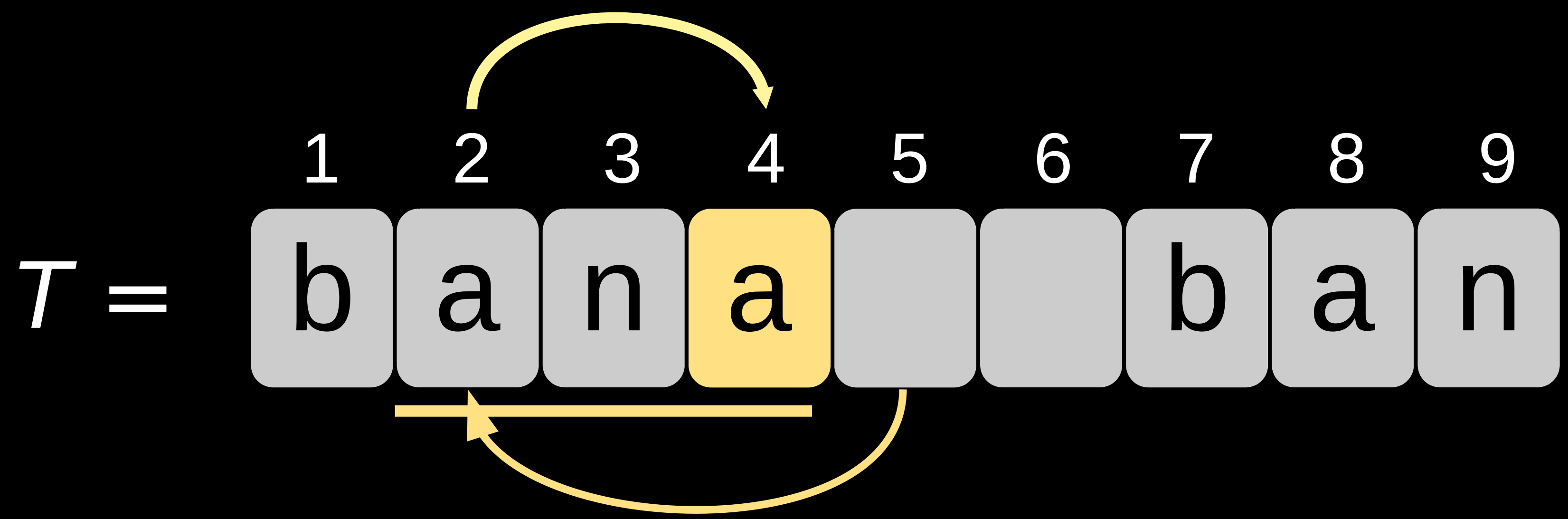
on decompression, copy characterwise



decompressing self-references

why are self-references allowed?

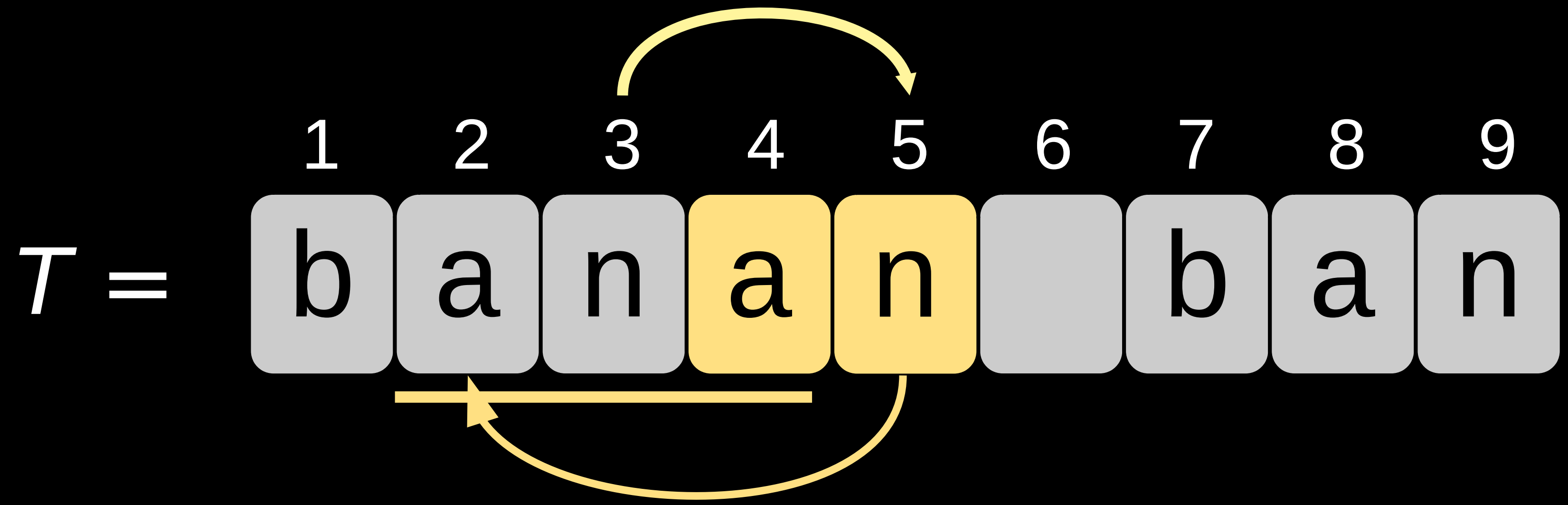
on decompression, copy characterwise



decompressing self-references

why are self-references allowed?

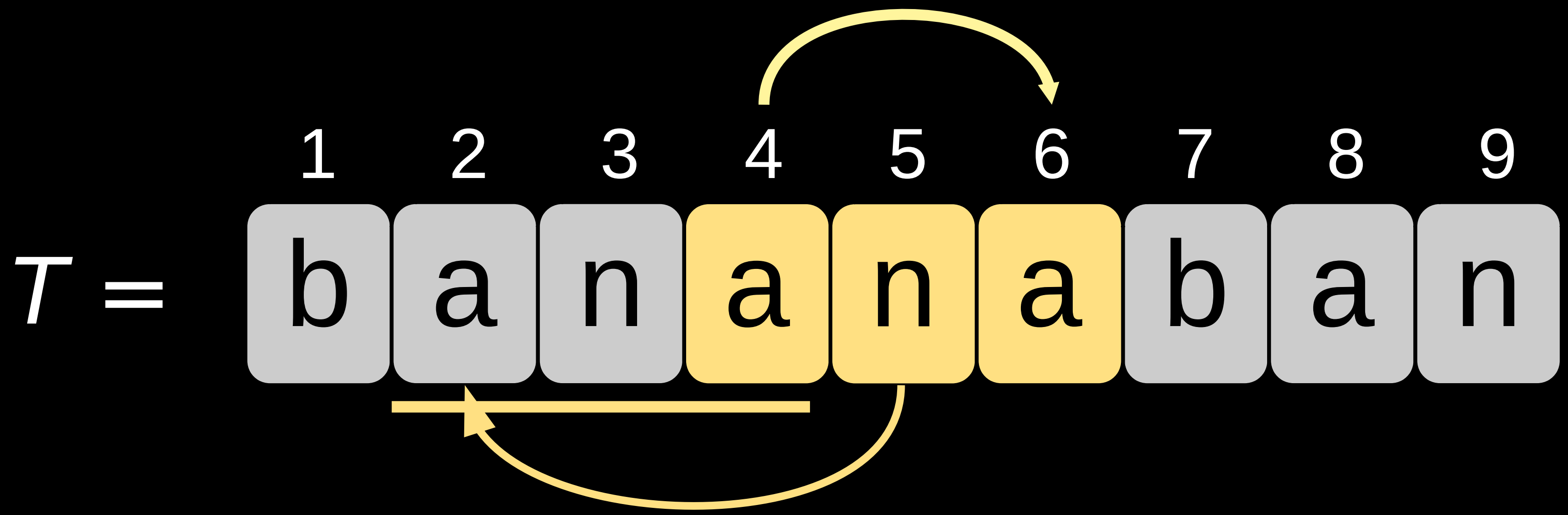
on decompression, copy characterwise



decompressing self-references

why are self-references allowed?

on decompression, copy characterwise



notation

- T : input text
- n : length of T , i.e., $n := |T|$
- σ : alphabet size
- $T[i..]$: suffix of T starting at position i

lexparse

- process T from left to right
- when computing factor starting at $T[i]$: select suffix $T[j..]$ directly lexicographically preceding $T[i..]$,
  - j becomes reference,
  - the factor length is the longest common prefix of $T[i..]$ and $T[j..]$

lexparse

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

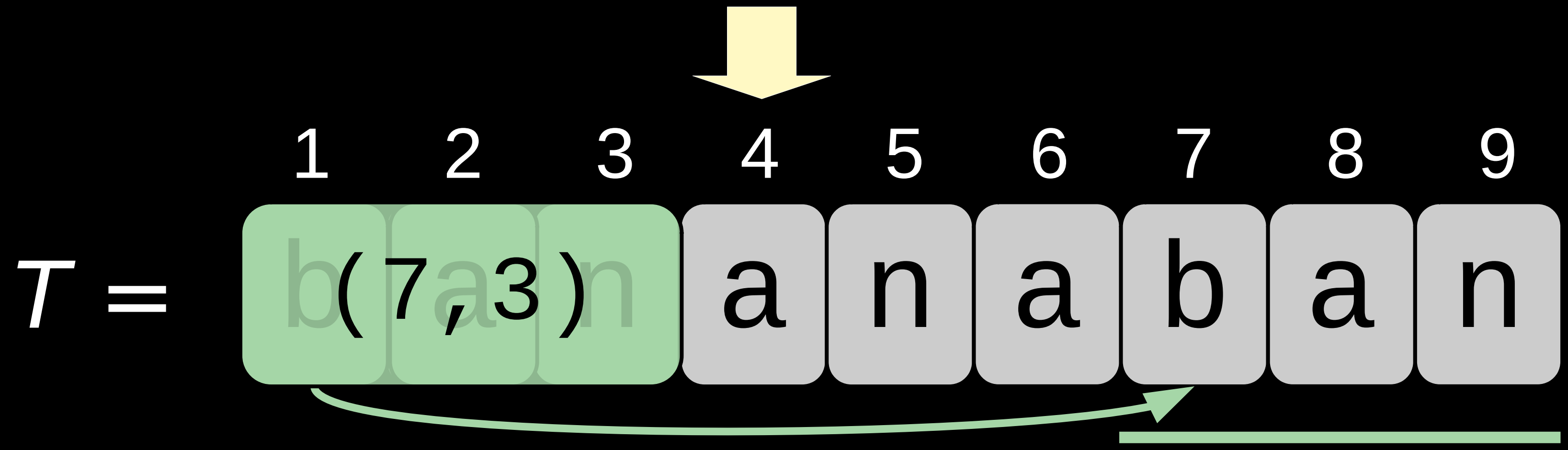
lexparse

- $T[7..] \prec T[1..]$ and
- there is no j with
  $T[7..] \prec T[j..] \prec T[1..]$

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

lexparse

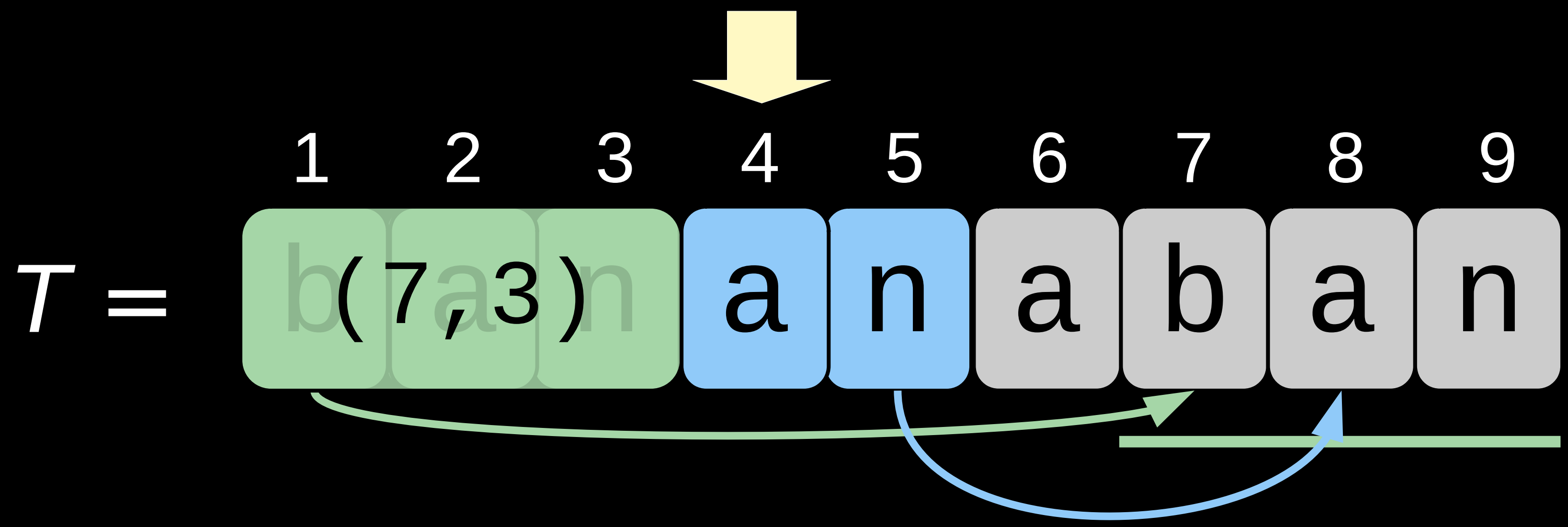
- $T[7..] \prec T[1..]$ and
- there is no j with $T[7..] \prec T[j..] \prec T[1..]$



copy 3 characters from position 7

lexparse

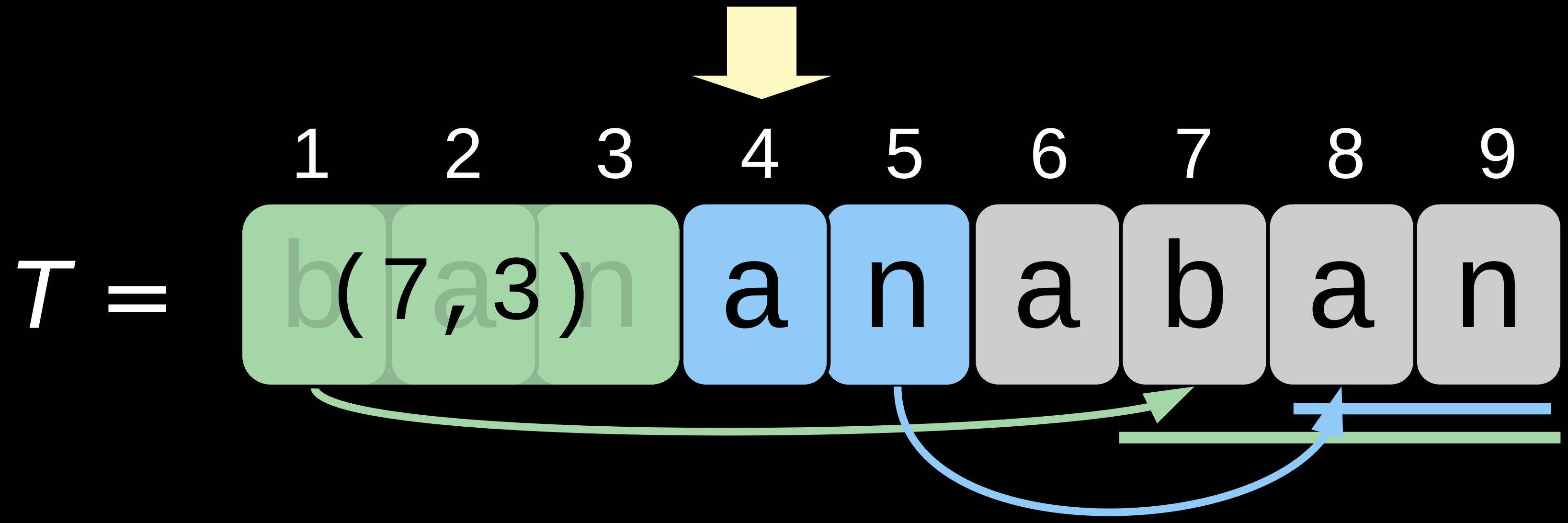
- $T[7..] \prec T[1..]$ and
- there is no j with $T[7..] \prec T[j..] \prec T[1..]$



copy 3 characters from position 7

lexparse

- $T[7..] \prec T[1..]$ and
- there is no j with $T[7..] \prec T[j..] \prec T[1..]$



copy 3 characters from position 7

copy 2 characters from position 8

decompressible

lexparse does not produce cycles

- reference is always the starting position of a lexicographically preceding suffix
- the lexicographic order induces a ranking (= total order) on all suffixes
- total orders are transitive

[Dinklage+ ‘17]

aim of this talk

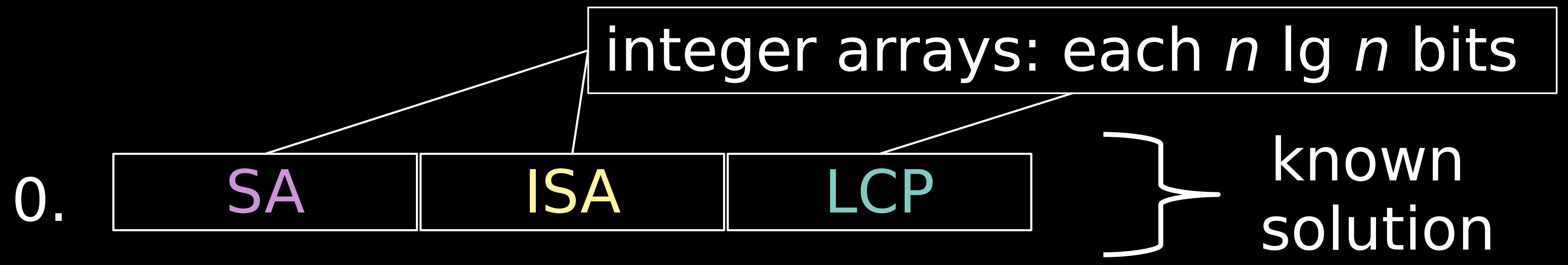
question:

Within O(n) time,
in what space can we compute lexparse ?

known solution:

- O(n) time and
- O(n log n) bits of space

aim of this talk



aim of this talk

integer arrays: each $n \lg n$ bits

0. SA | ISA | LCP — known solution

1. SA | ISA

2. Φ

3. Φ’ | B — compressed Φ + bit vector

$r \lg n + n + o(n)$ bits, where
r : #character runs in the
Burrows-Wheeler transform

(1.–3.: this talk)

Definition of SA

order of suffixes

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

order of suffixes

	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
```

order of suffixes

	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

for visualization,
left-align all suffixes

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

order of suffixes

sort lexicographically

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

suffix array SA

store starting positions
of the suffixes → suffix array 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

construction of SA

- enumerating all suffixes takes $\Omega(n^2)$ time
- however, there are $O(n)$-time algorithms constructing SA with enumeration

[Ko, Aluru '05]

suffix array SA

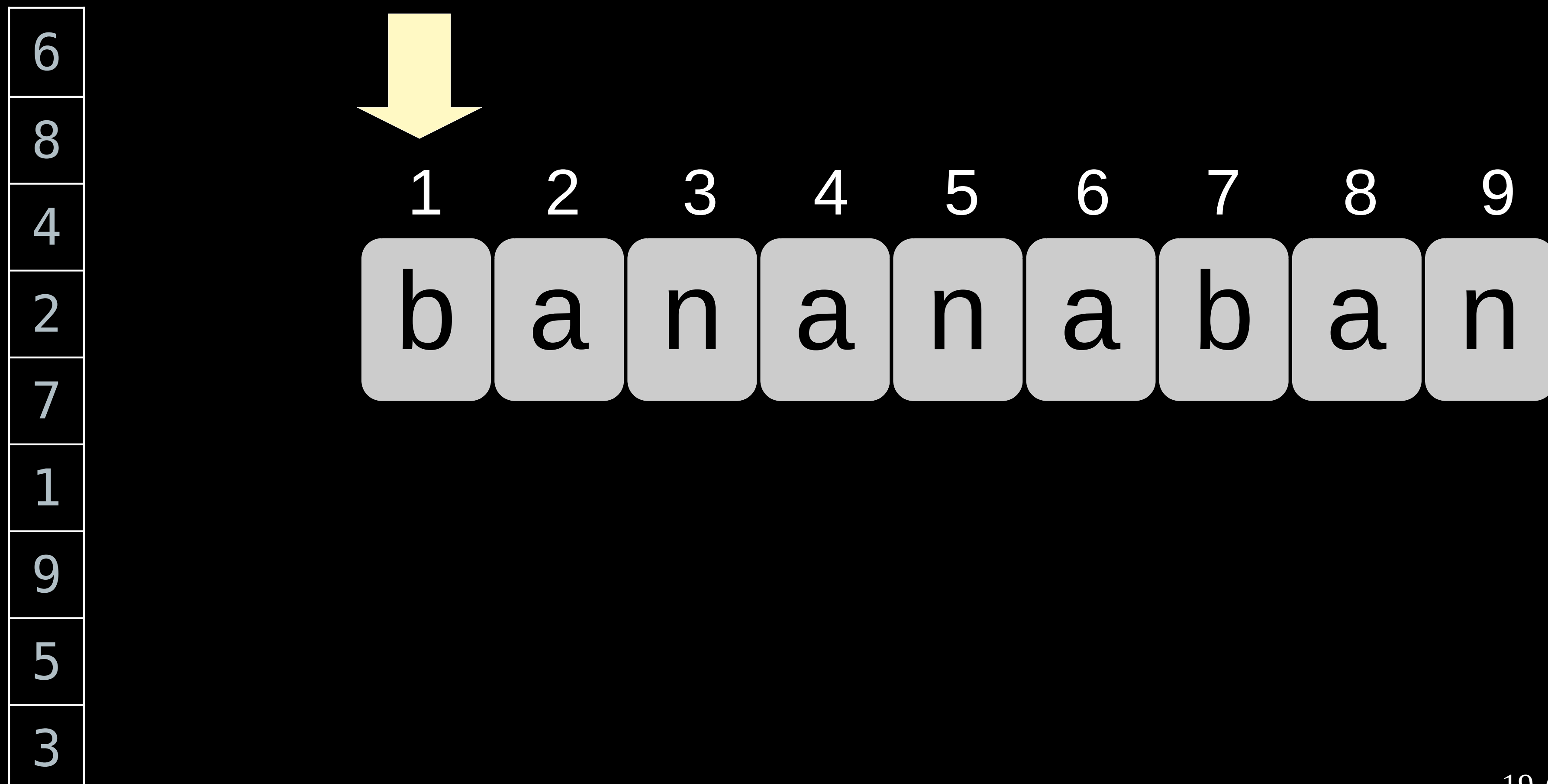
6
8
4
2
7
1
9
5
3

known solution:
compute lexparse

- in O(n) time
- with O(n log n) bits

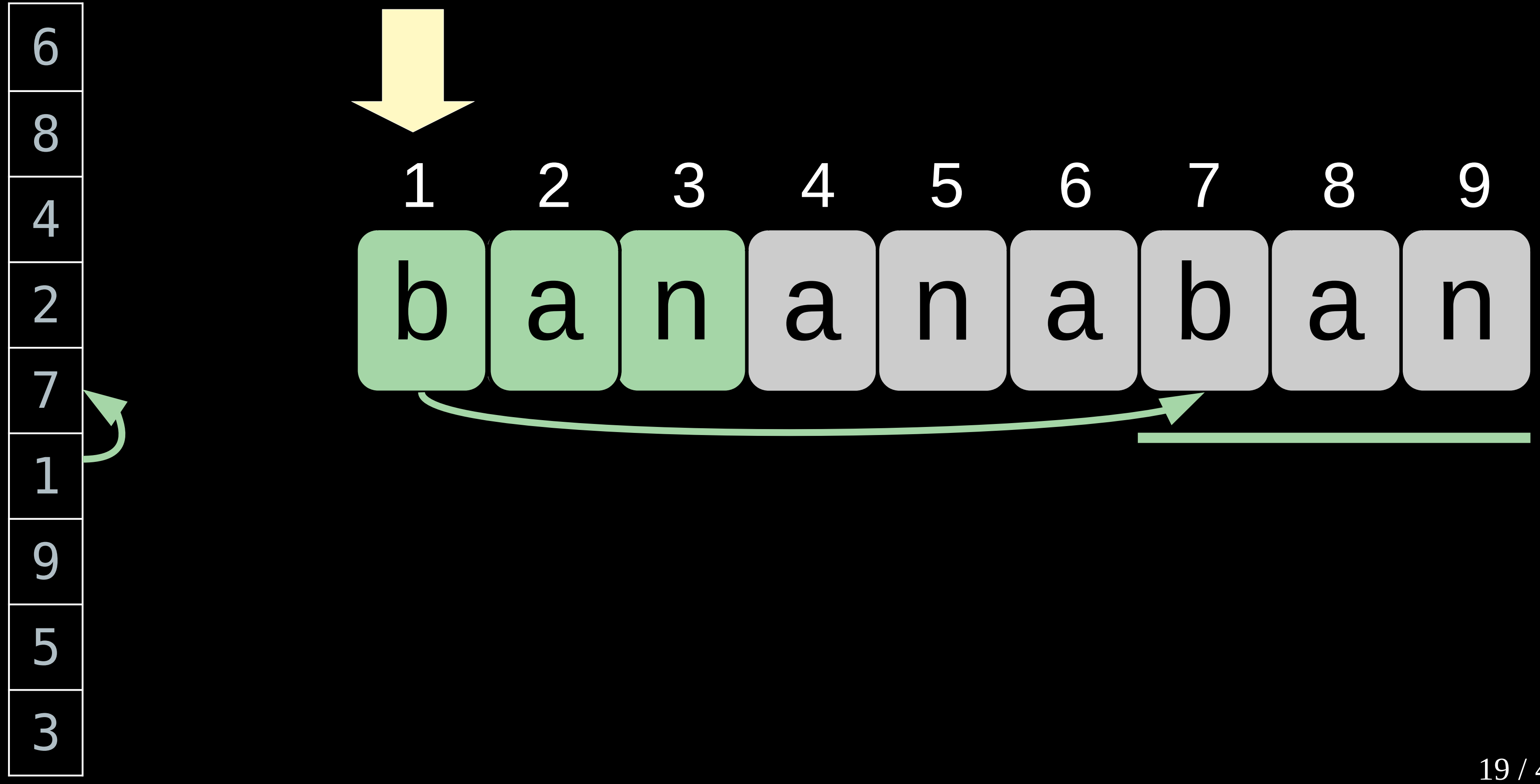
SA-based computation of lexparse

suffix array SA



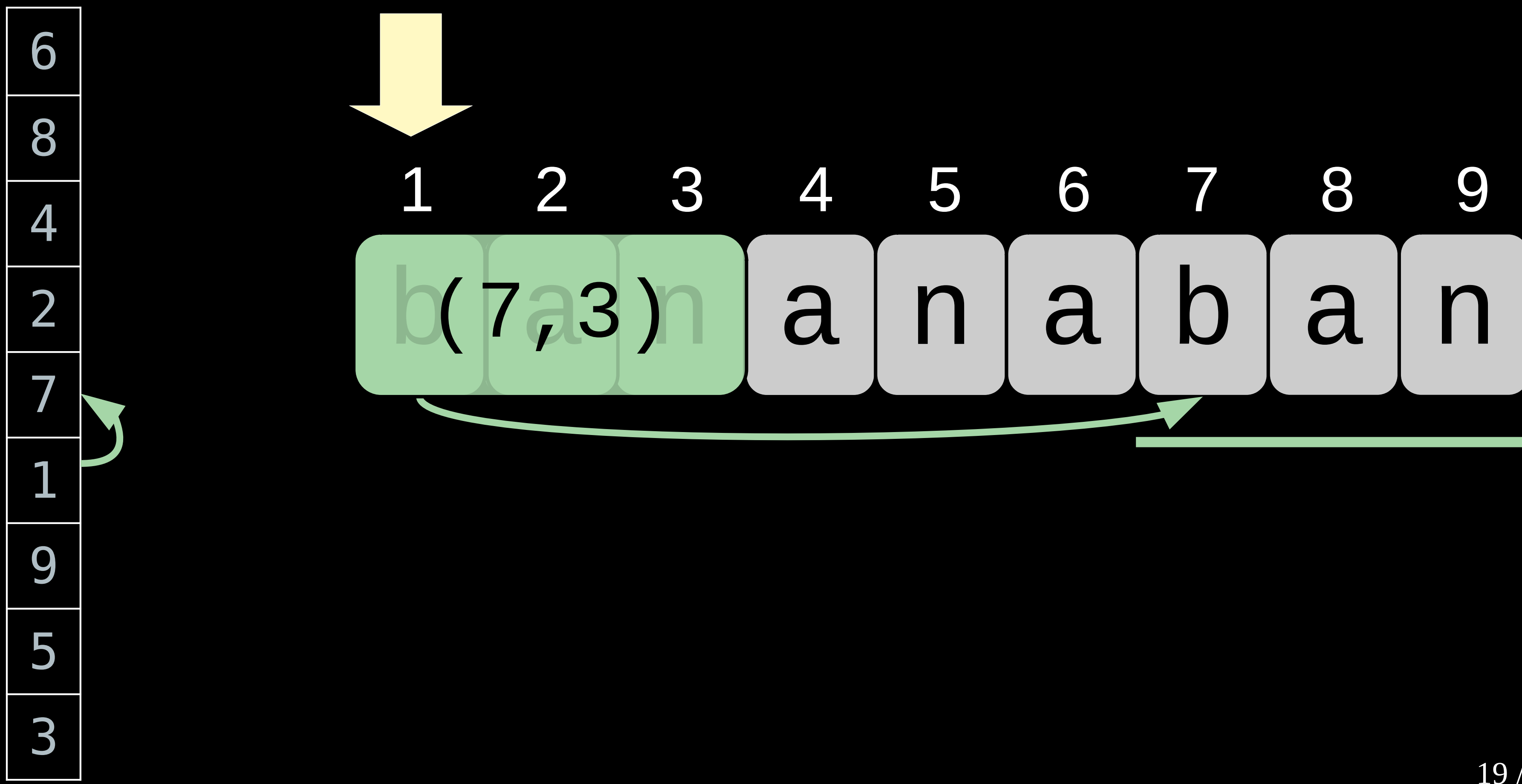
SA-based computation of lexparse

suffix array SA



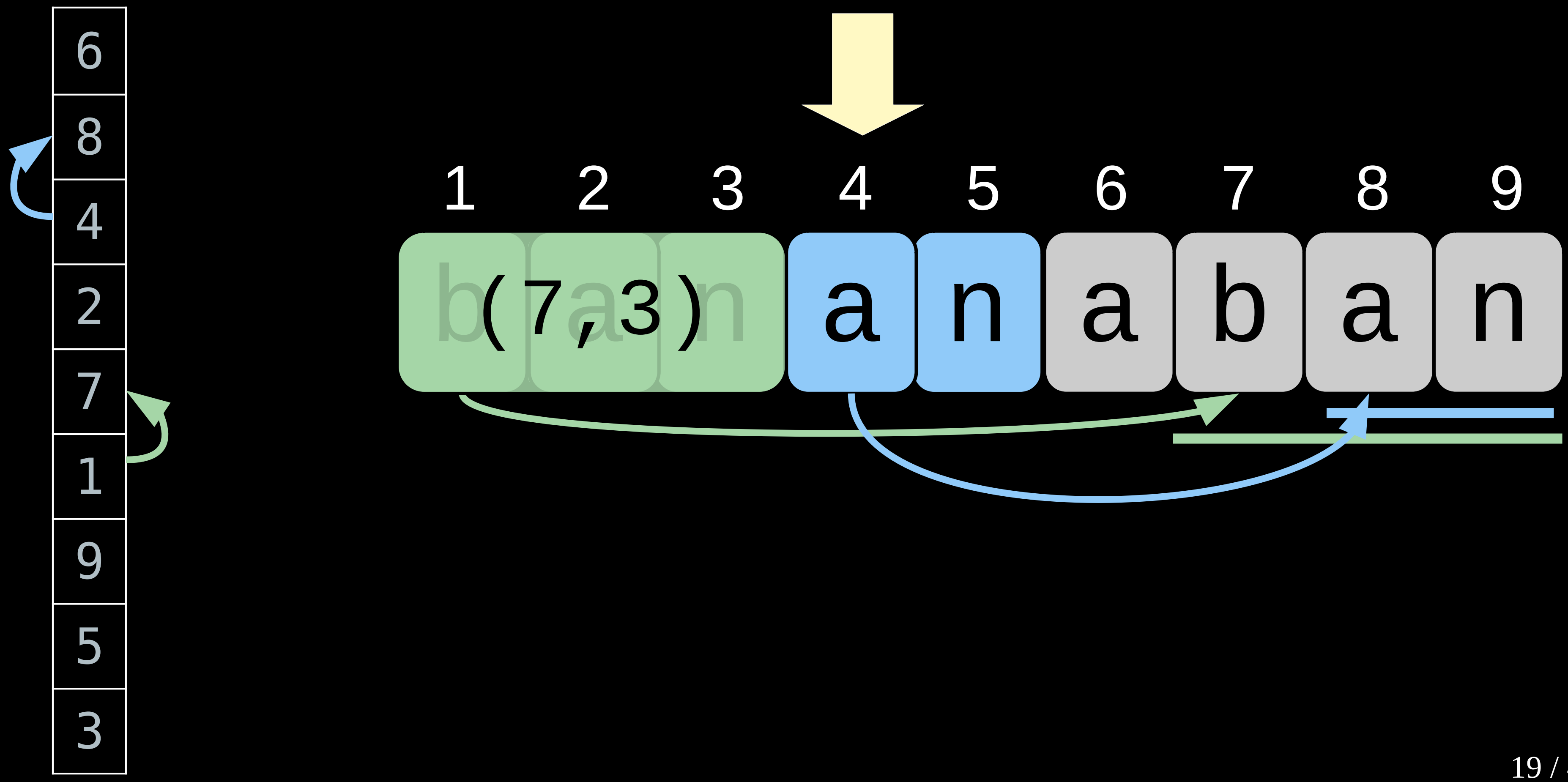
SA-based computation of lexparse

suffix array SA



SA-based computation of lexparse

suffix array SA



ISA / LCP

- to compute factor F starting at $T[i]$, we need to know index p with i = SA[p]
- for that use inverse suffix array ISA with SA[ISA[i]] = i such that ISA[i]= p

  ⇒ reference of F is SA[p-1] = SA[ISA[i]-1]
- length of reference given by LCP array storing, for every p, the longest common prefix of T[SA[p] ..] and T[SA[p-1]..] in LCP[p]

  ⇒LCP[ISA[i]] = LCP[p] is the length of F

known algorithm

- construct SA, ISA, LCP in O(n) time
- compute factor starting at $T[i]$ in constant time:
  - reference: SA[ISA[i] - 1]
  - length : max(LCP[i], 1)
- O(n) total time
- pseudo code :
  i = 1; while $i < n$:
  - if LCP[i] = 0: report $T[i]$; $i \leftarrow i + 1$
  - else: report pair (SA[ISA[i]-1], LCP[i]); $i \leftarrow i$ + LCP[i]

[Navarro+ ‘21]

known algorithm

- [Navarro+ ‘21]: O(n) time,

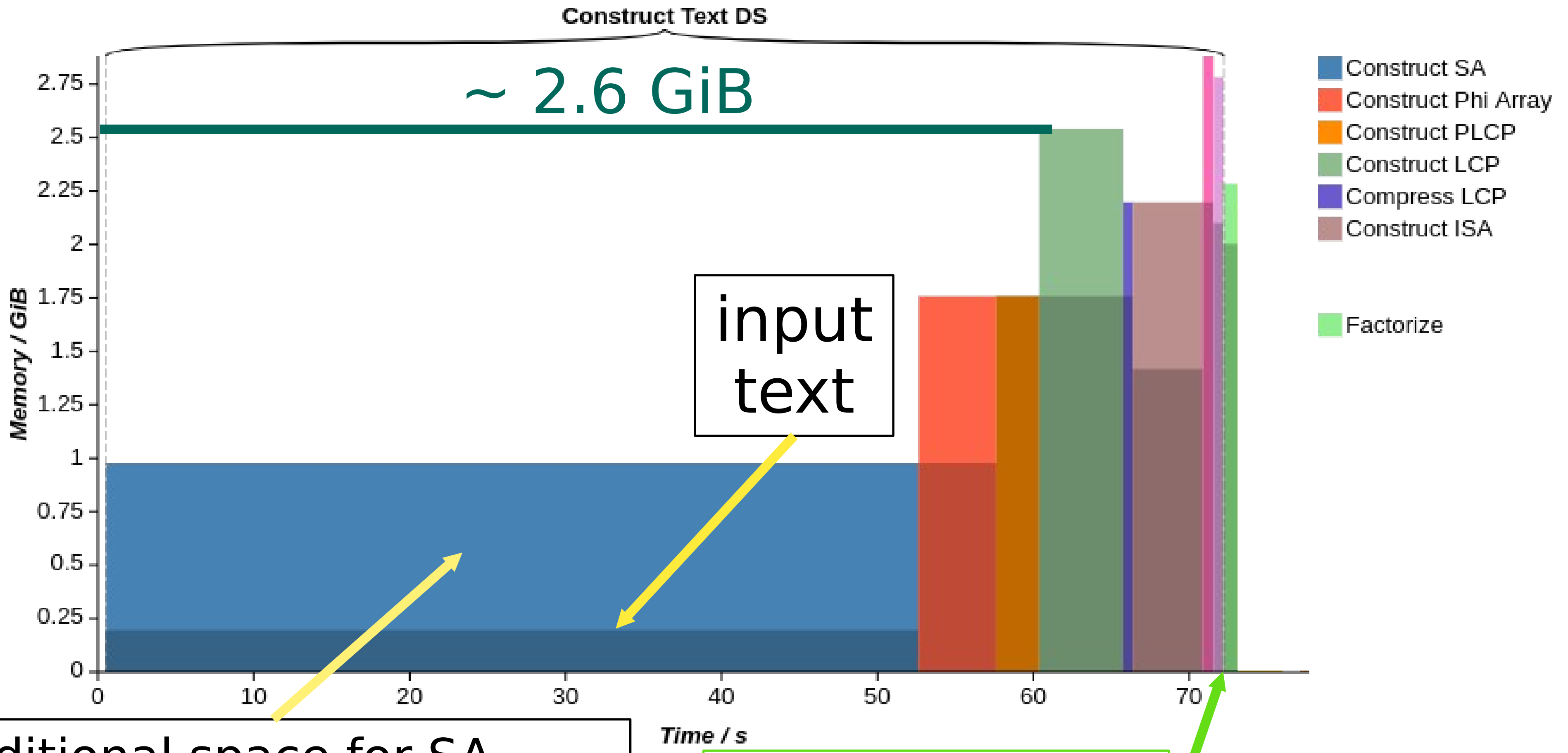
    3 integer arrays: SA, ISA, LCP

concrete example

- byte alphabet (1 byte = 8 bits)
- entry of an integer array: 4 bytes (32 bits)
- for 200 MiB of input:
  2.6 GiB RAM are necessary

(1 MiB = 1024^2 byte)

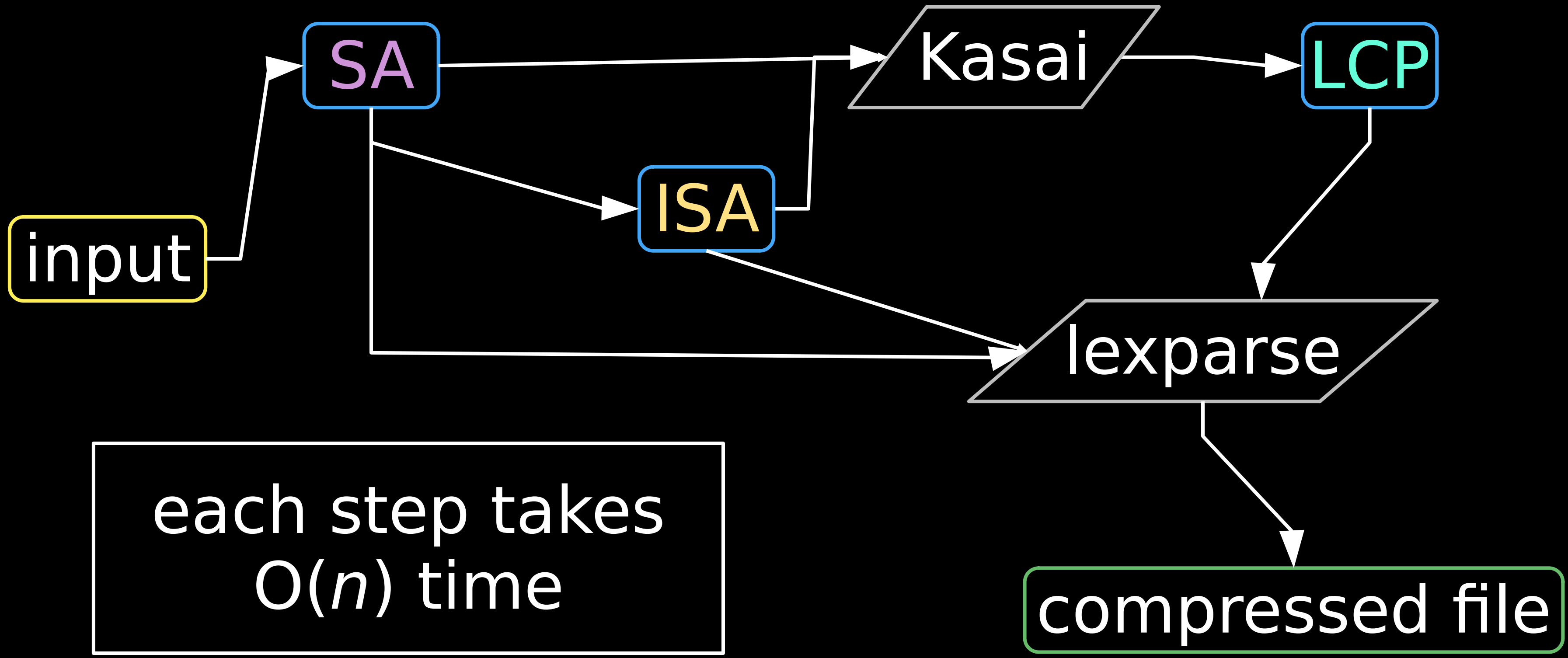
200 MiB
ASCII web pages



additional space for SA construction including space for SA

lexparse computation

algorithmic flow chart



each step takes
O(n) time

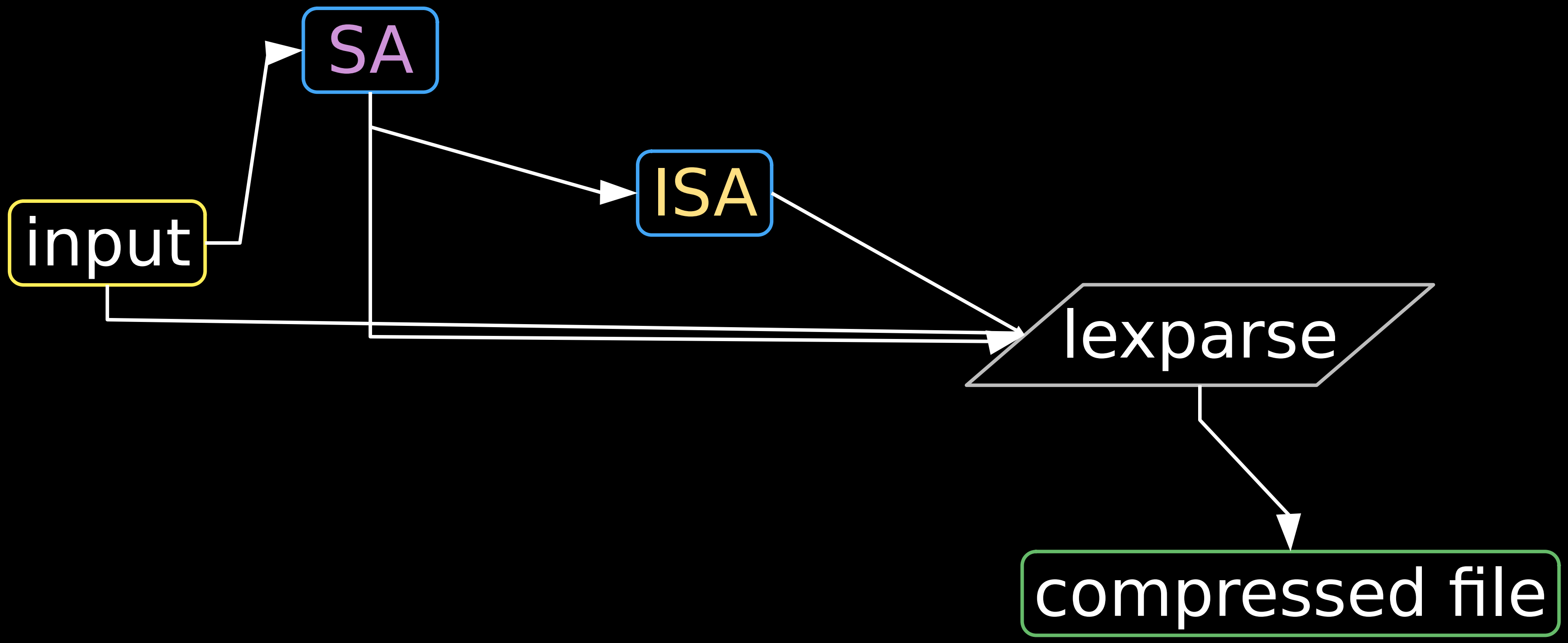
Kasai+ '01: LCP array construction algorithm

towards small memory

- drop LCP
- compute longest common prefix of $T[i..]$ and $T[\mathrm{SA}[\mathrm{ISA}[i]-1]..]$ naively
- given factor F_x has length $|F_x|$, then $\sum_x |F_x| = n$

$\Rightarrow$ O(n) time is needed

algorithmic flow chart



- are SA / ISA necessary?

Φ array

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

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 array

Φ[*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 array

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

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 array

Φ[*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 array

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

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 array

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

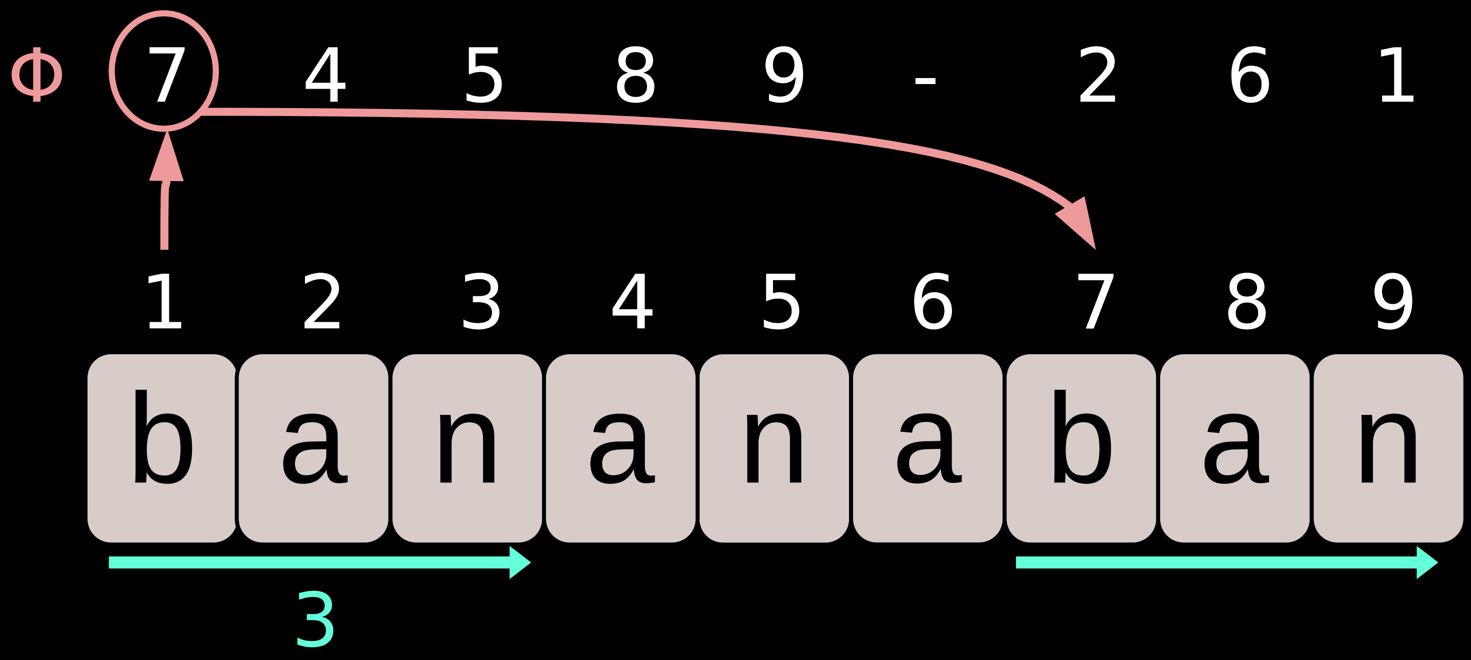
Φ	7	4	5	8	9	-	2	6	1
SA	6	8	4	2	7	1	9	5	3
ISA	6	4	9	3	8	1	5	2	7
	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n

application of Φ

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

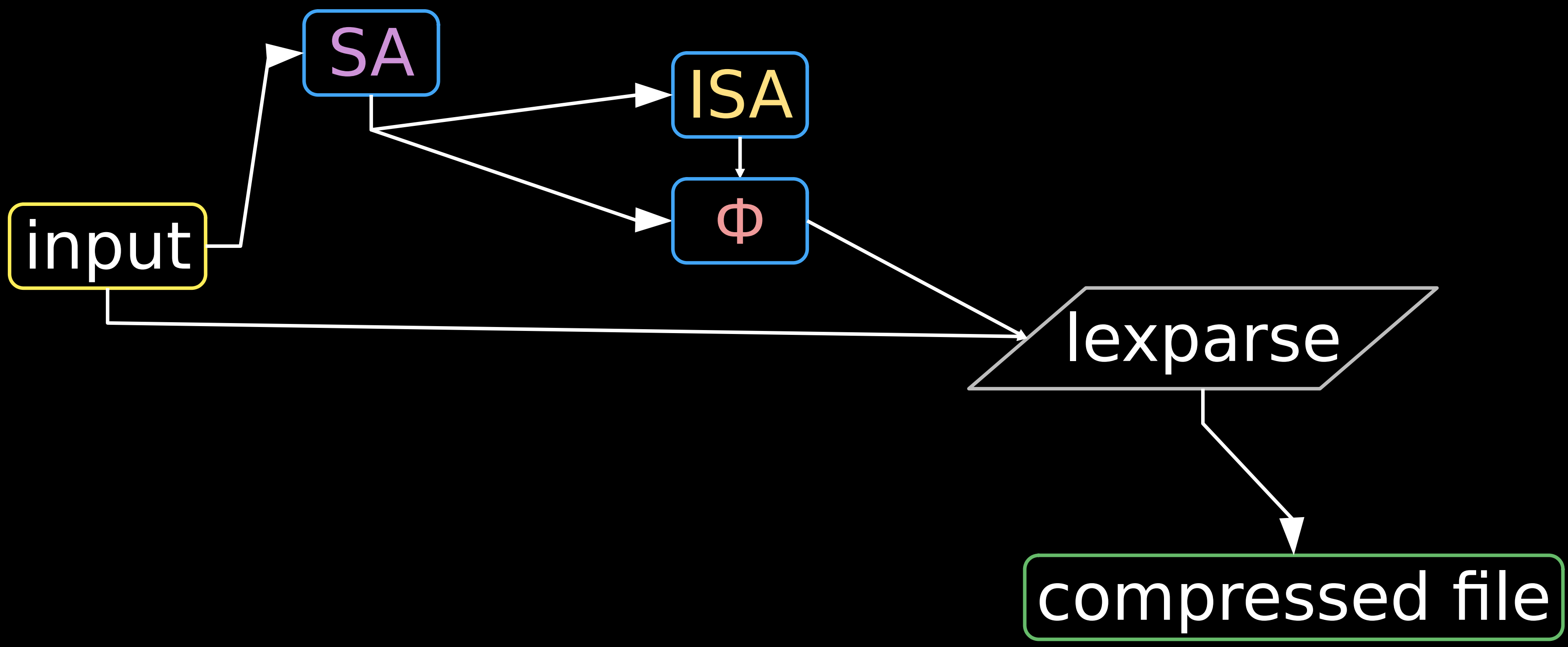
- Φ[i]: reference
- factor length computed naively

application of Φ

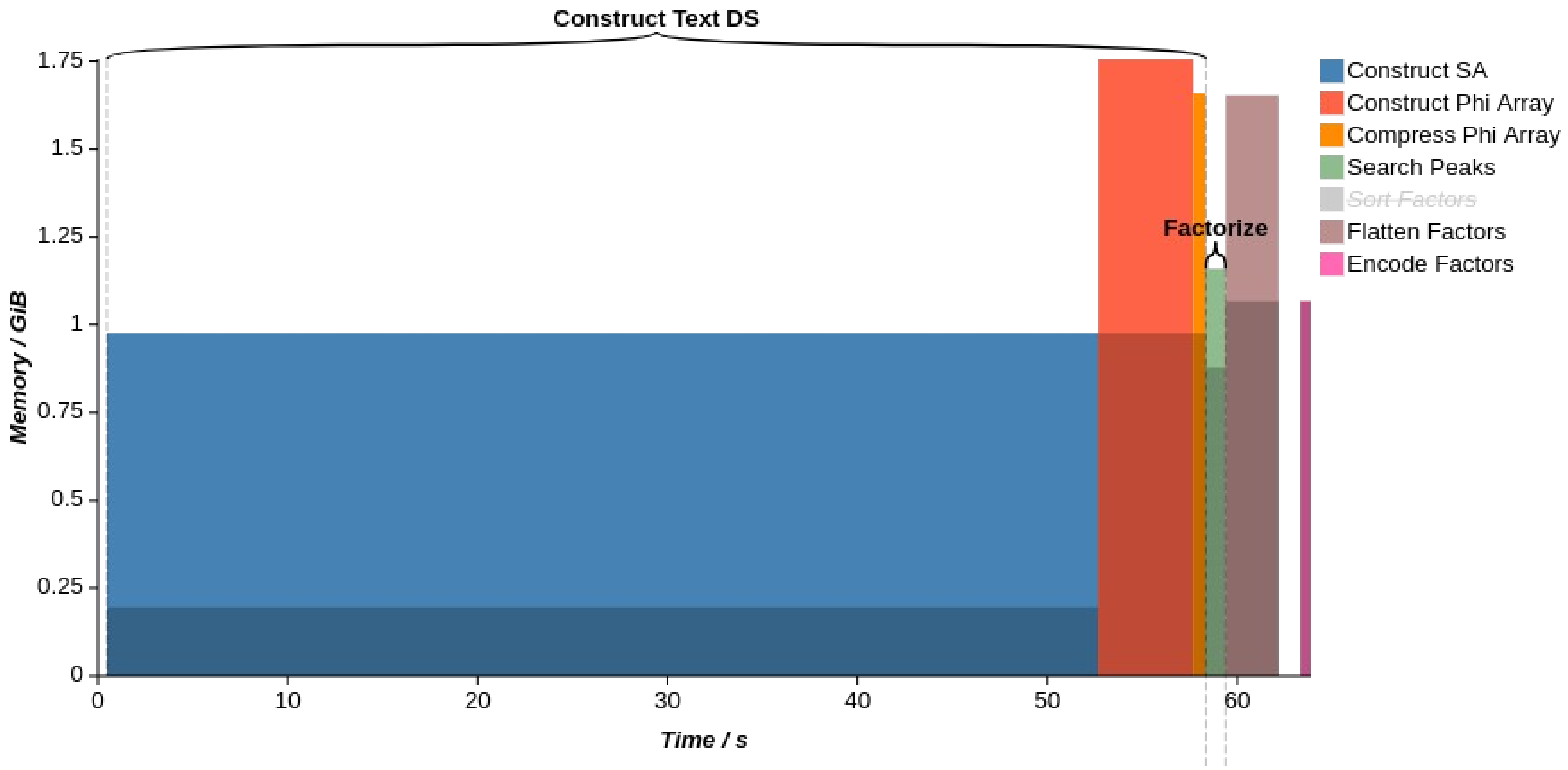


- Φ[*i*]: reference
- factor length computed naively

algorithmic flow chart

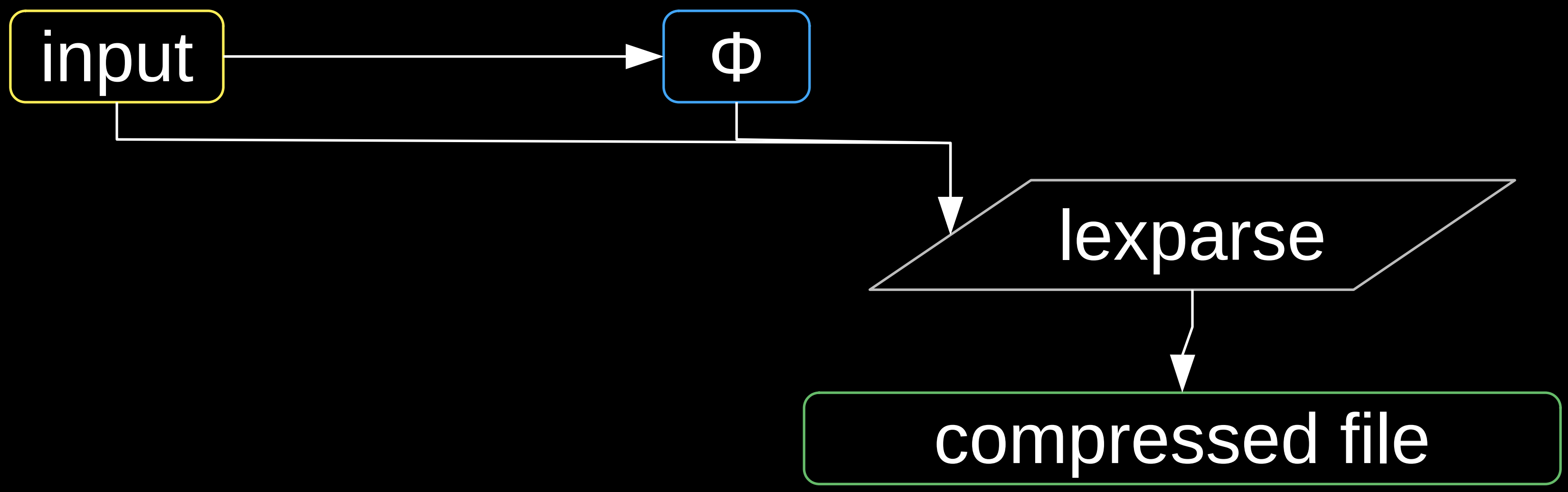


200 MiB
ASCII web pages



39% of memory reduced

algorithmic flow chart



[Goto, Bannai '14]:
construct Φ from input text directly with

- O(n) time and
- O(σ lg n) bits of additional working space

precomputation : max. memory usage

0) SA + ISA + LCP : 2.88 GiB

1) SA + ISA → Φ: 1.76 GiB

2) only Φ: $\sim$1 GiB

all methods are linear time,
but 2) only needs 35% of the memory of 0)

compressed Φ representation

entries with $\Phi[i] = \Phi[i-1]$ + 1 are prevalent for highly repetitive texts

⇒ allows for compression

+1 (from $\Phi[2]$ to $\Phi[3]$) +1 (from $\Phi[4]$ to $\Phi[5]$)

Φ	7	4	5	8	9	-	2	6	1
i	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n

Φ’: sparse Φ

+1 (from position 2 to 3)   +1 (from position 4 to 5)

Φ’	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

compressed Φ

bit vector

left-align

B	1	1	0	1	0	1	1	1	1
Φ’	7	4	8	-	2	6	1		
	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

compressed Φ

query Φ[*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

if $B[j] = 1$, $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
where $\text{rank}_1(j)$ counts the ‘1’s in $B[1..j]$

compressed Φ

query Φ[j],
$j = 4$

B[1..4] has 3 '1's

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

if $B[j] = 1$, $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
where $\text{rank}_1(j)$ counts the '1's in $B[1..j]$

compressed Φ

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

$B[1..4]$ has 3 '1's

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

if $B[j] = 1$, $\Phi[j] = \Phi'[B.\text{rank}_1(j)]$
where $\text{rank}_1(j)$ counts the '1's in $B[1..j]$

compressed Φ

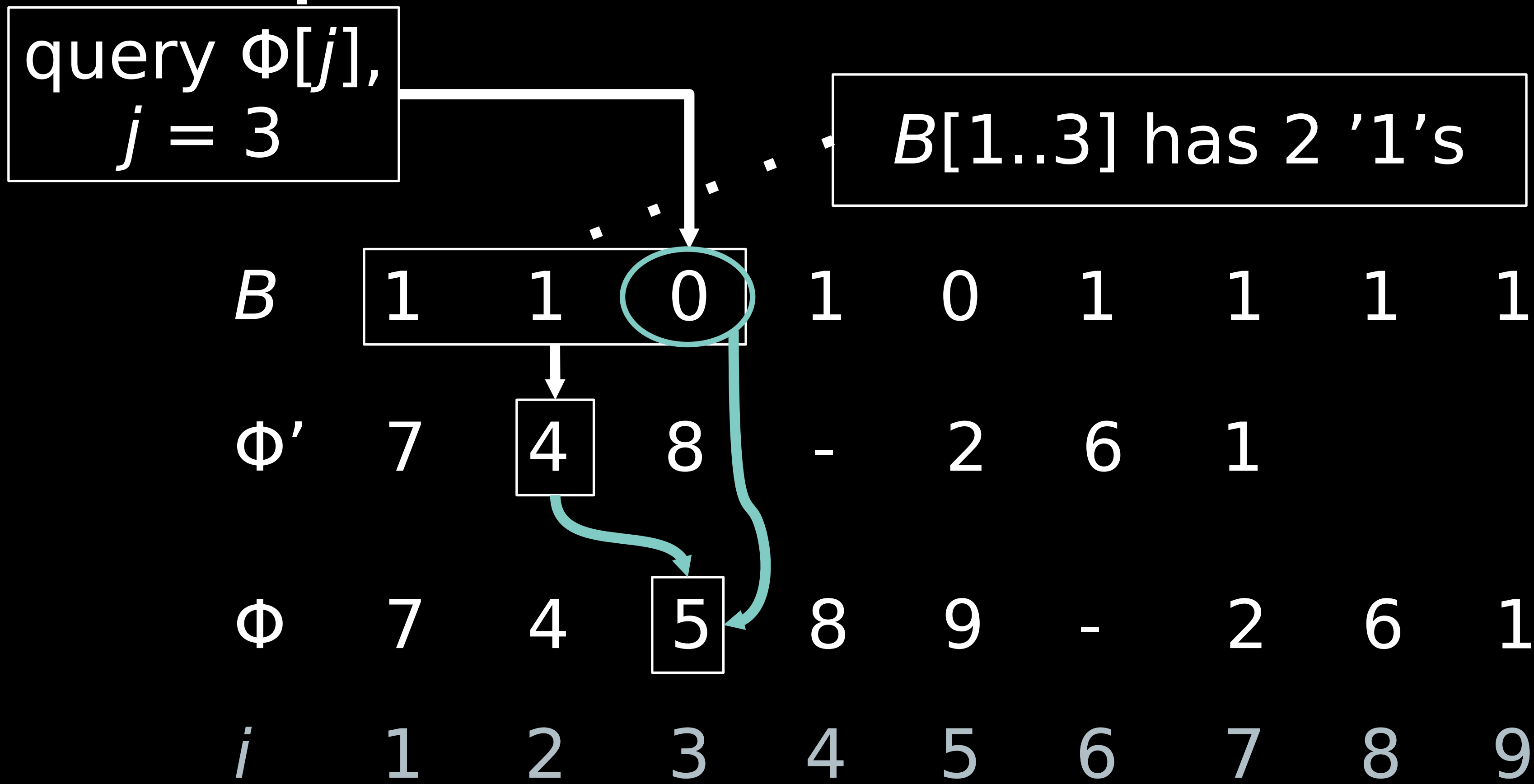
query $\Phi[j]$, $j = 3$

$B[1..3]$ has 2 '1's

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

if $B[j] = 0$:
$\Phi[j] = \Phi'[B.\text{rank}_1(j)] +$
$B.\text{rank}_0(j) - B.\text{rank}_0(B.\text{select}_1(B.\text{rank}_1(j)))$
where $B.\text{select}_1(k)$ gives the position of the k-th '1' in B

compressed Φ



if $B[j] = 0$:
$\Phi[j] = \Phi'[B.\mathrm{rank}_1(j)] +$
$B.\mathrm{rank}_0(j) - B.\mathrm{rank}_0(B.\mathrm{select}_1(B.\mathrm{rank}_1(j)))$
where $B.\mathrm{select}_1(k)$ gives the position of the k-th ‘1’ in B

rank / select

construct rank/select data structure on bit vector $B[1..n]$

- $O(n)$ construction time
- constant query time for rank / select
- $n + o(n)$ bits of space (including B)

[Jacobson ‘89, Clark ‘96]

space analysis

- number of entries i with $\Phi[i] \neq \Phi[i\text{-}1] + 1$ is bounded by r, where r is #character runs in the Burrows-Wheeler transform [Kärkkäinen+ '16]

$\Rightarrow$ $r \lg n + n + o(n)$ bits of total space for Φ

summary

construct lexparse in O(n) time:

- only with Φ array
- represent Φ in $r \lg n + n + o(n)$ bits

open problems:

can we compute compressed Φ directly from text in compressed space?

implementation: https://tudocomp.github.io

questions are welcome!