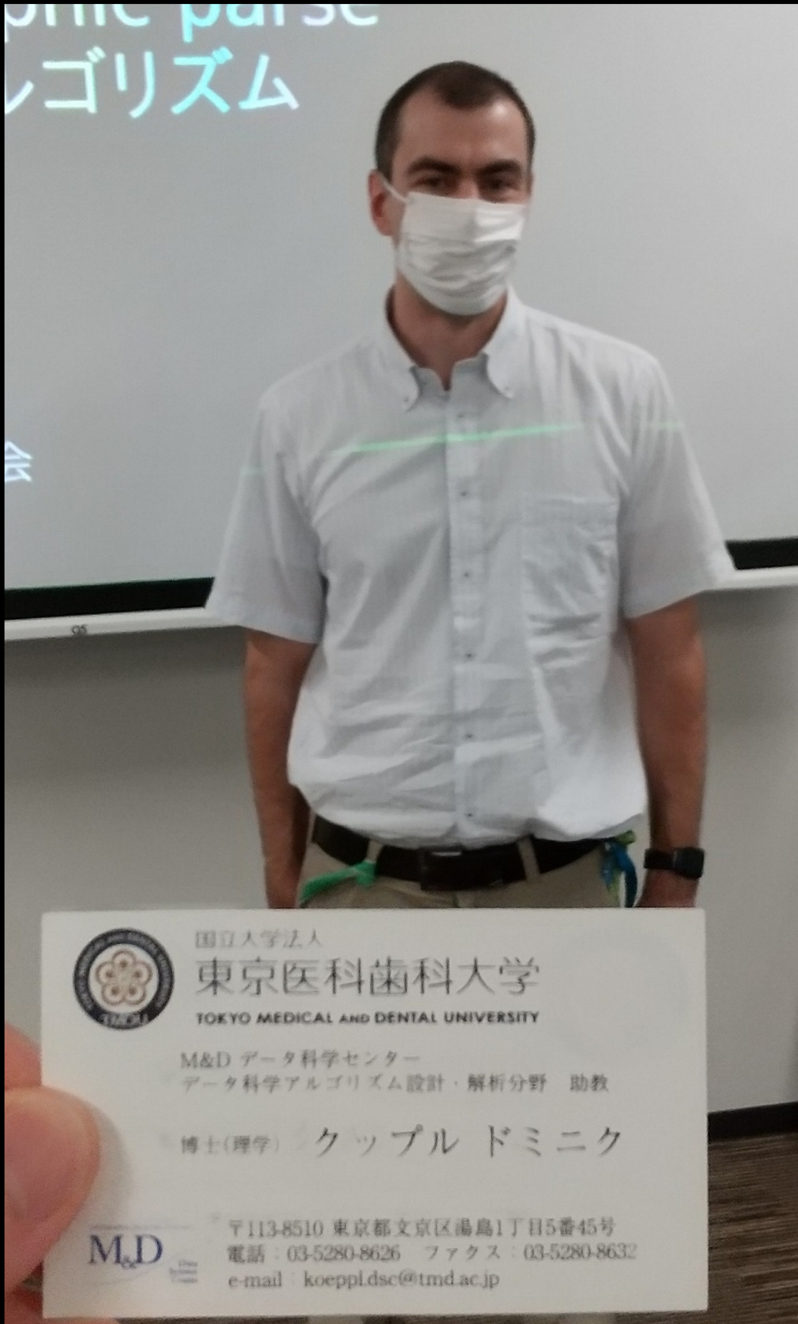
managing text data

Dominik Köppl
Tokyo Medical and
Dental University



why text data?

text data (strings) are ubiquitous:

- natural texts
- source code

```
namespace C = packed::character;
void print_packed(uint64_t packed) {
   for(size_t i = 0; i < C::FIT_CHARS; ++i) {
      std::cout << C::character(packed, i);
   }
}
```

- DNA

GCTACGT...

- binary code

0110011110101100

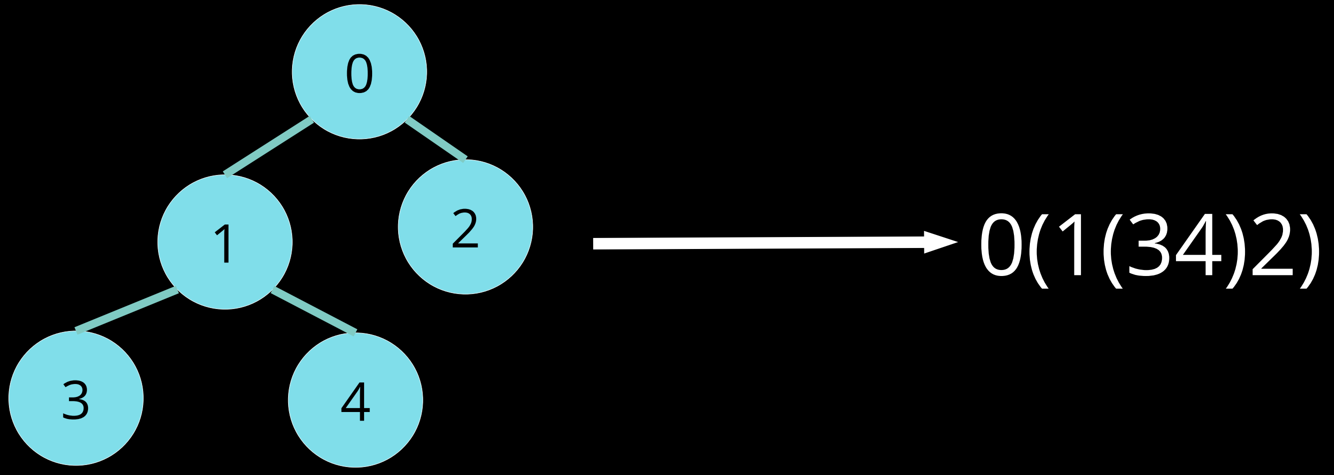
- etc.

why text data?

text data (strings) are ubiquitous:

we can interpret all data one-dimensionally by serialization

- graphs: adjacency lists
- trees: depths first search (mentioned later)



why strings?

string is a simple data type

⇒ already exhaustively treated?

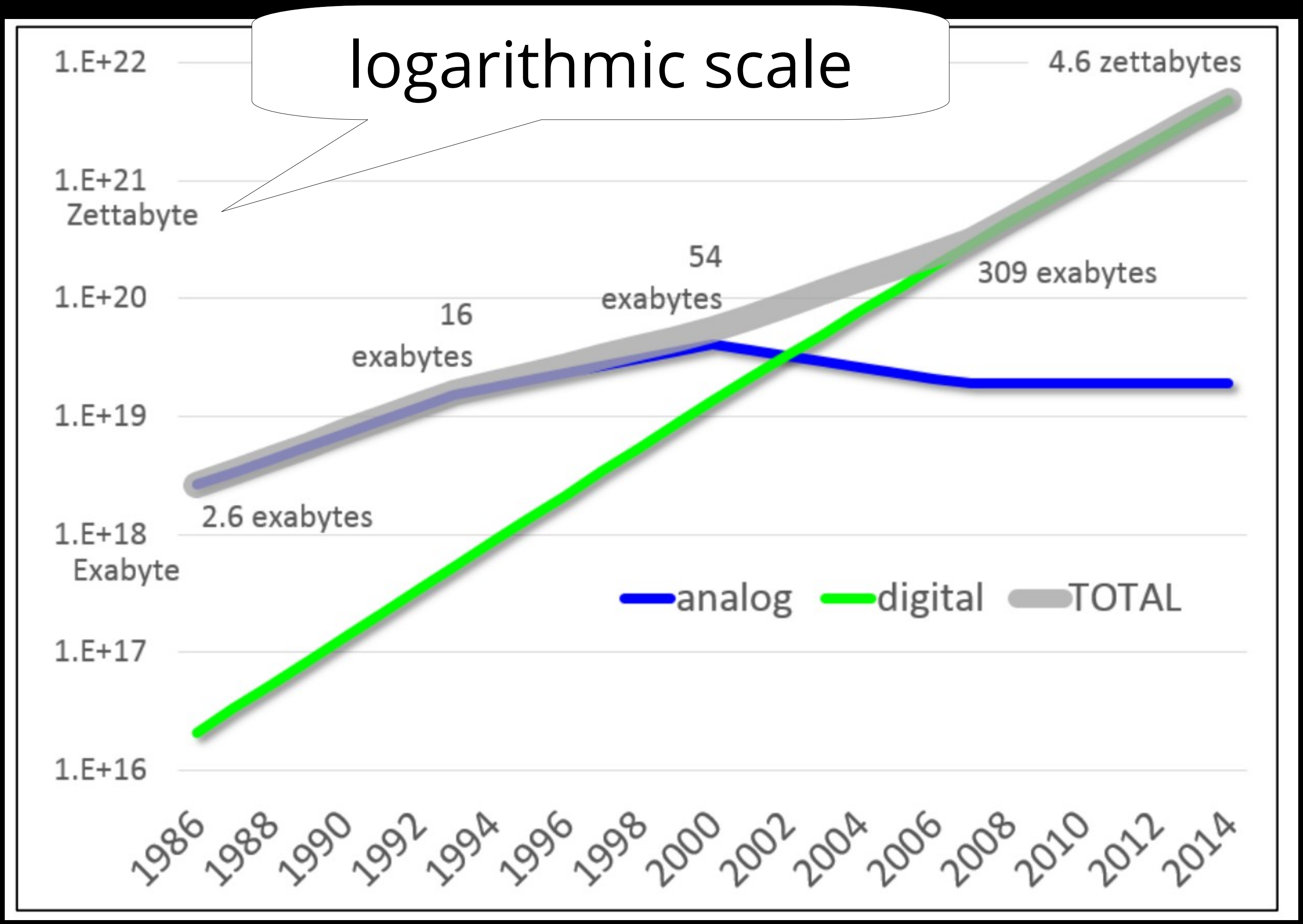
why strings?

string is a simple data type

⇒ already exhaustively treated?

many problems occur with big data sizes!

estimated information capacity of the world



[Hilbert, Lopez '11]

big data

- data collections in the web

  Wikipedia, Google Books, etc.

- version control systems

  git, SVN, etc.

- biological data

  1000 Genome Project:

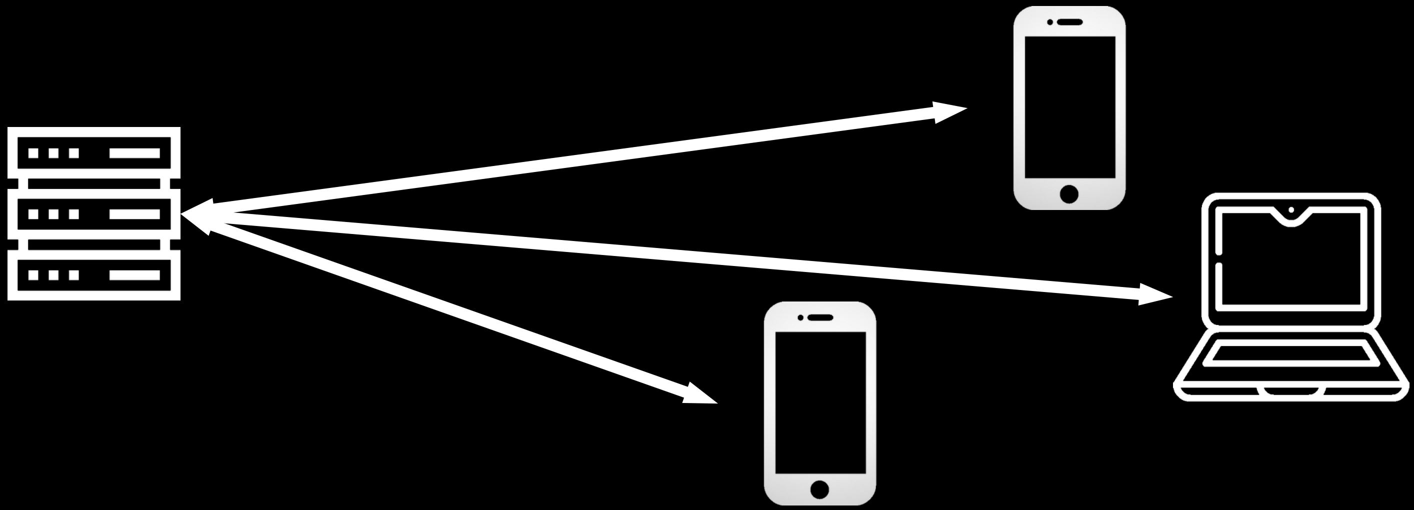
  human genomes, each ~ $3.2 \cdot 10^9$ characters

big data

to manage big data we need

- memory- and
- time-efficient

solutions, in particular for data transfer!



big data

- the problems to solve are often easily describable

    like pattern matching

big data

- the problems to solve are often easily describable

    like pattern matching

but:

- naive solutions are often
    - too slow
    - use too much space

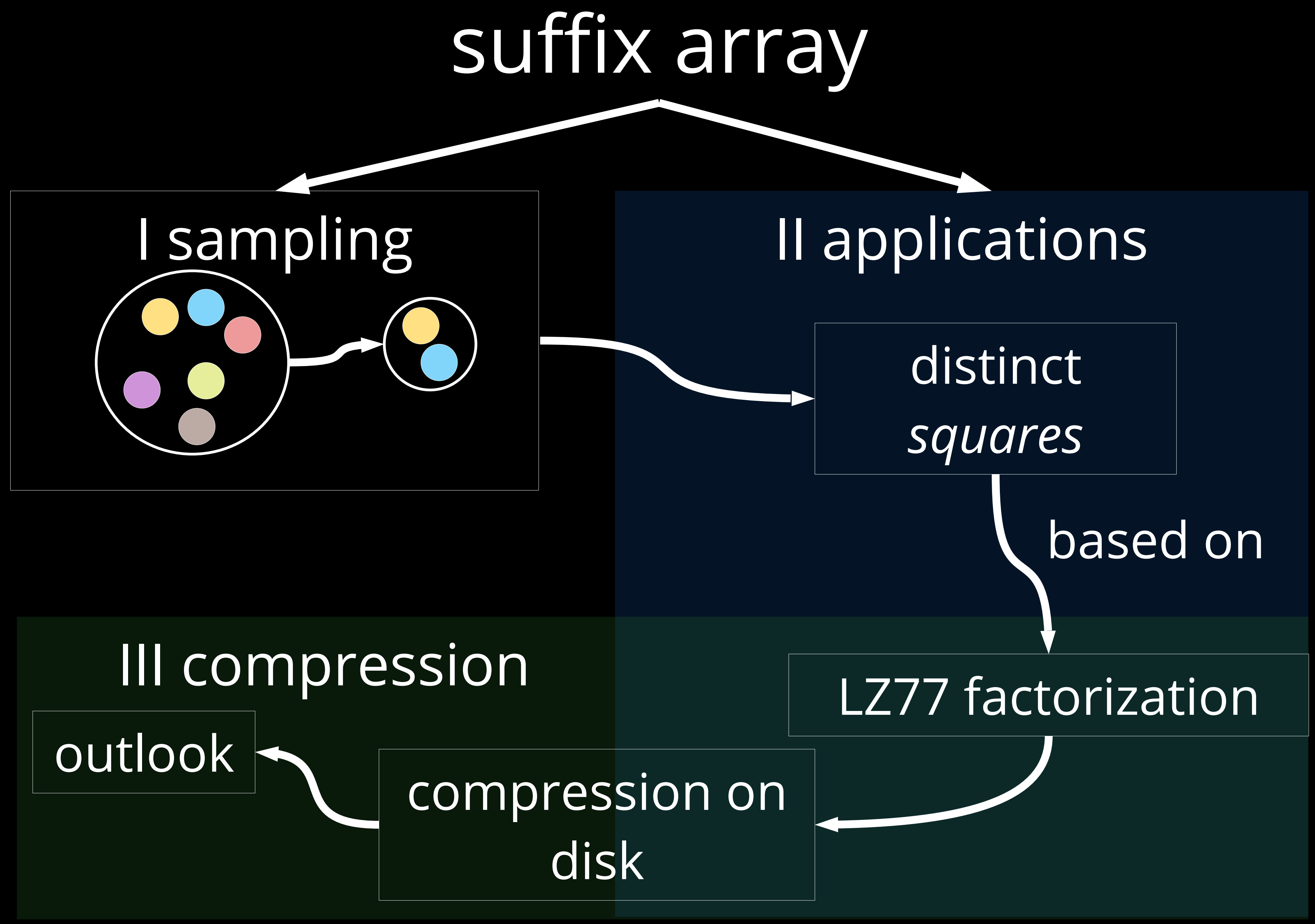
scalability

- how good performs a data structure or an algorithm when scaling the data size?
- goal: given a text with n characters
  - number of steps of an algorithm should be at most linear to n (linear-time algorithm).
    We say: $O(n)$ time
  - same for space (linear-space consumption)
    We say: $O(n)$ space

example text

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

- text length: n = |T| = 9
- alphabet Σ = { a , b }
- alphabet size σ = |Σ| = 2


suffix array
I sampling
II applications
distinct
squares
based on
III compression
LZ77 factorization
outlook
compression on
disk

suffix array

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

T =

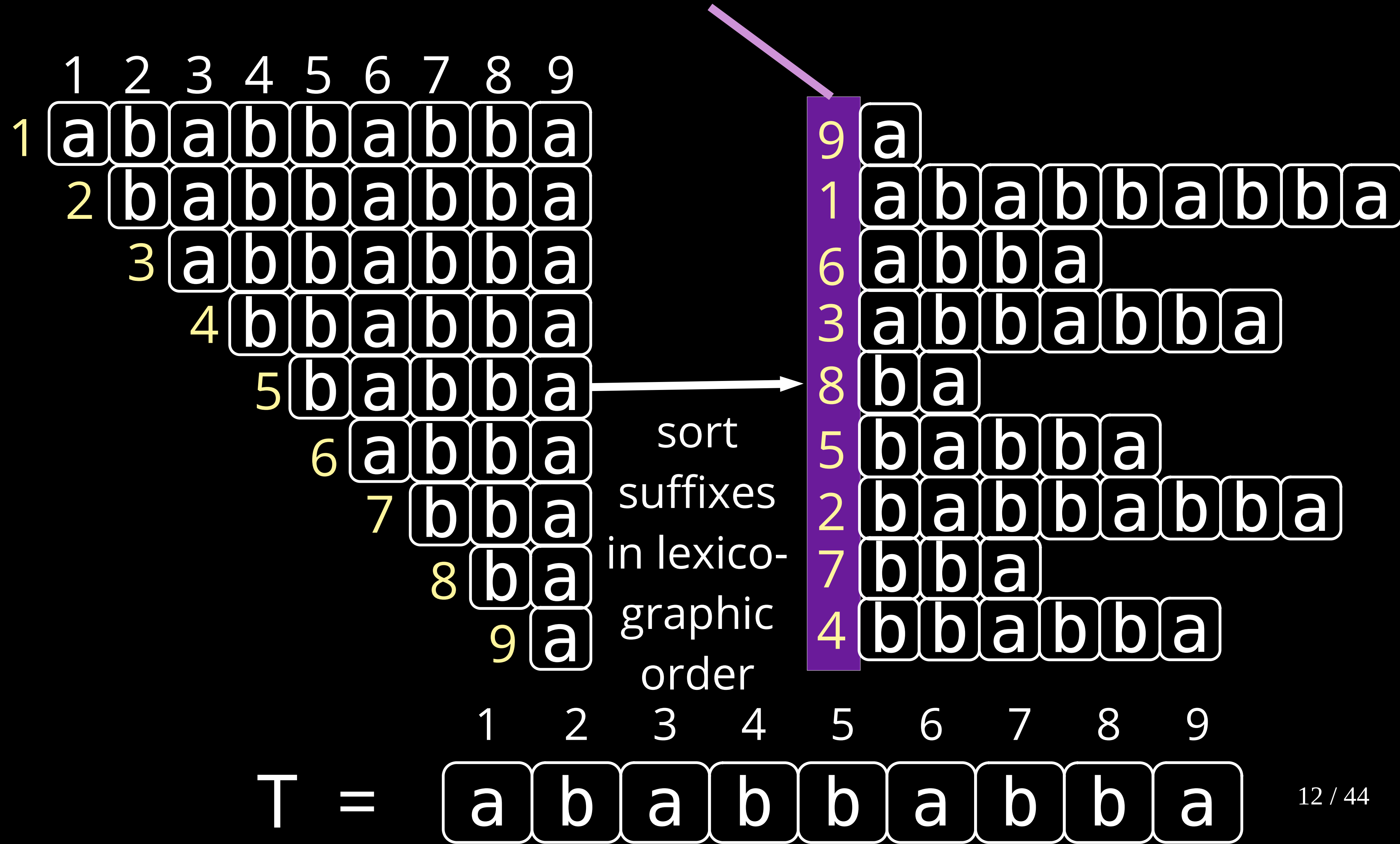
1	2	3	4	5	6	7	8	9
a	b	a	b	b	a	b	b	a

suffix array

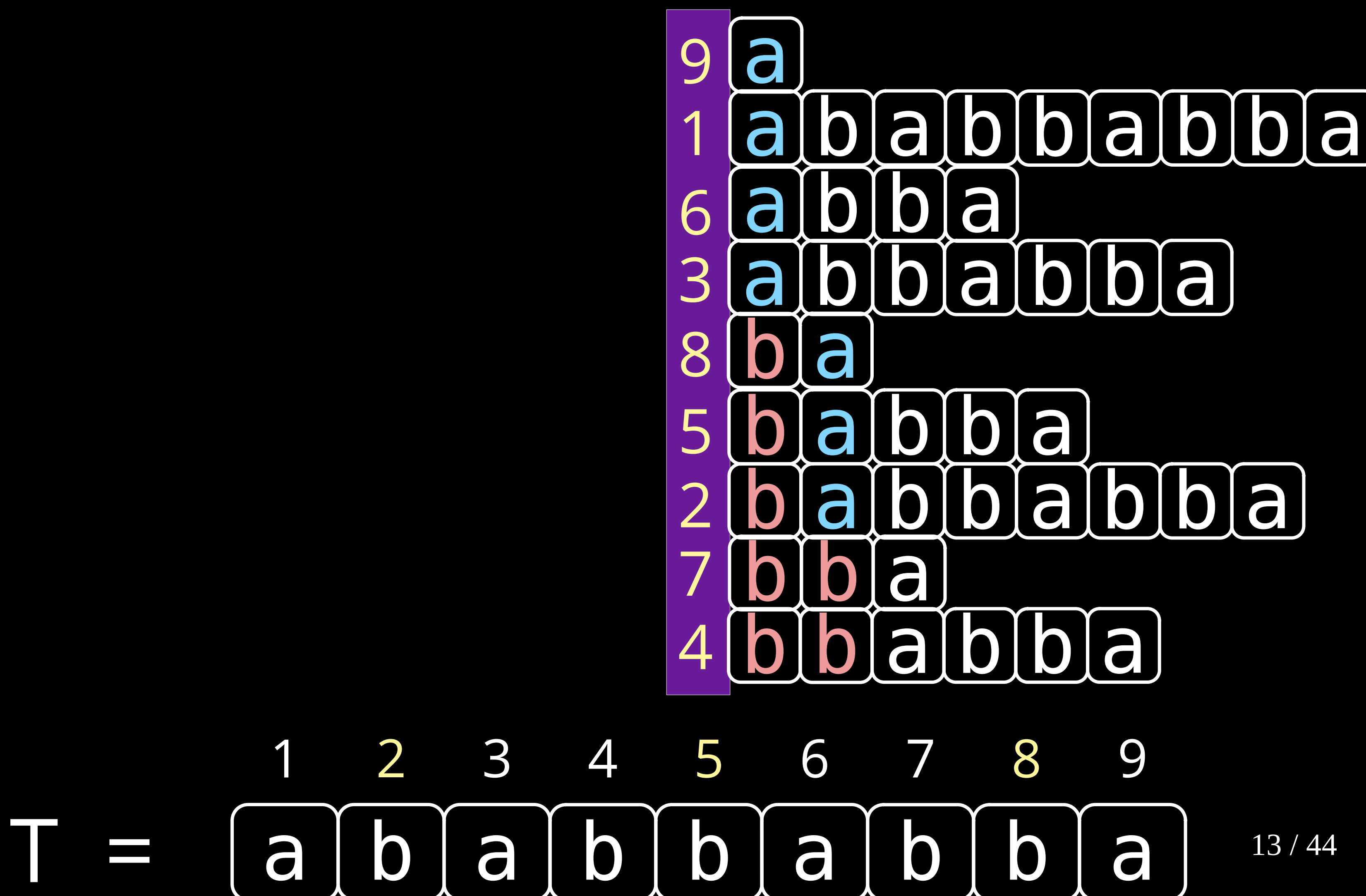
	1	2	3	4	5	6	7	8	9
1	a	b	a	b	b	a	b	b	a
2		b	a	b	b	a	b	b	a
3			a	b	b	a	b	b	a
4				b	b	a	b	b	a
5					b	a	b	b	a
6						a	b	b	a
7							b	b	a
8								b	a
9									a

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

suffix array

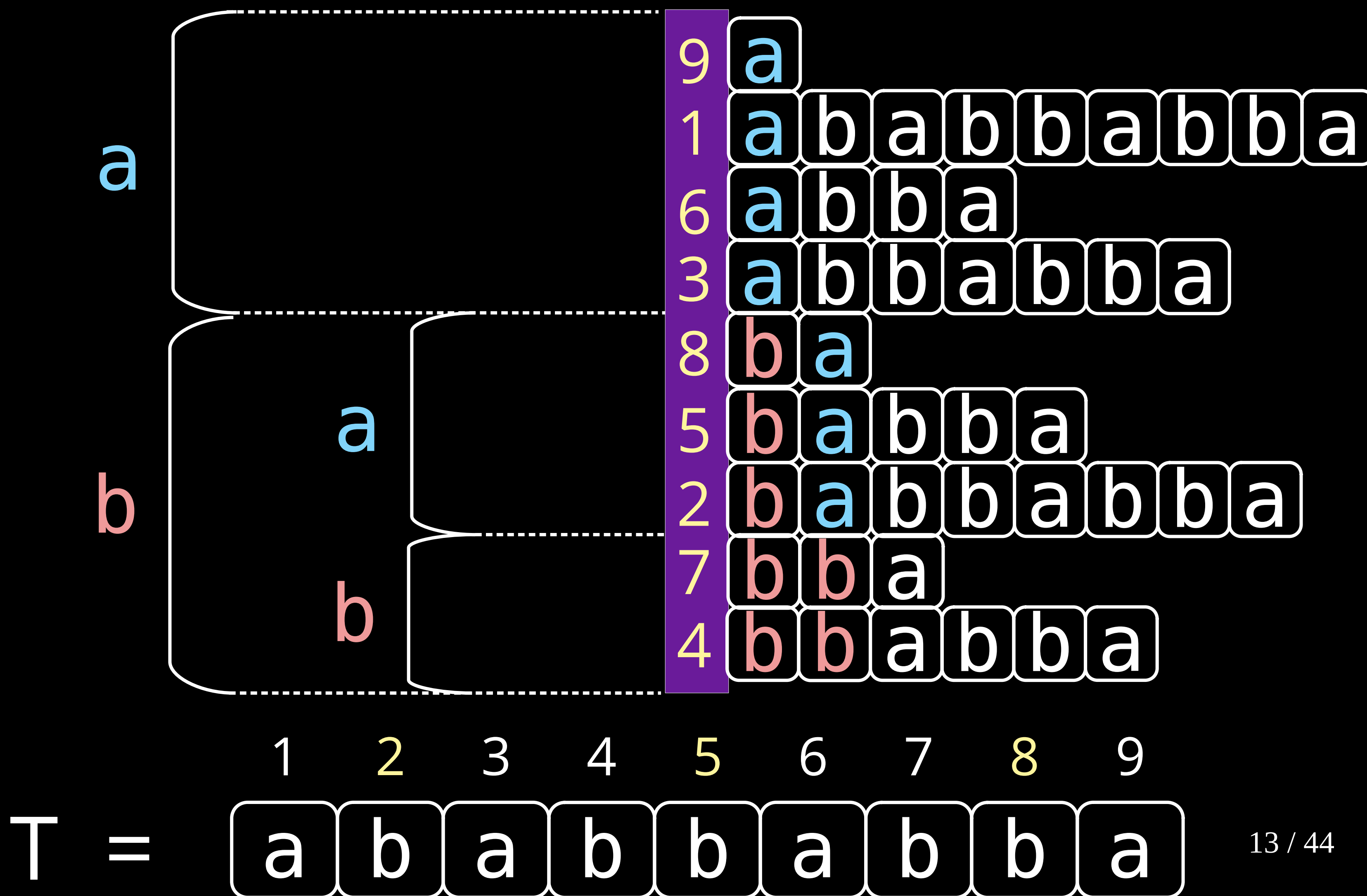


pattern matching : # ba = ?

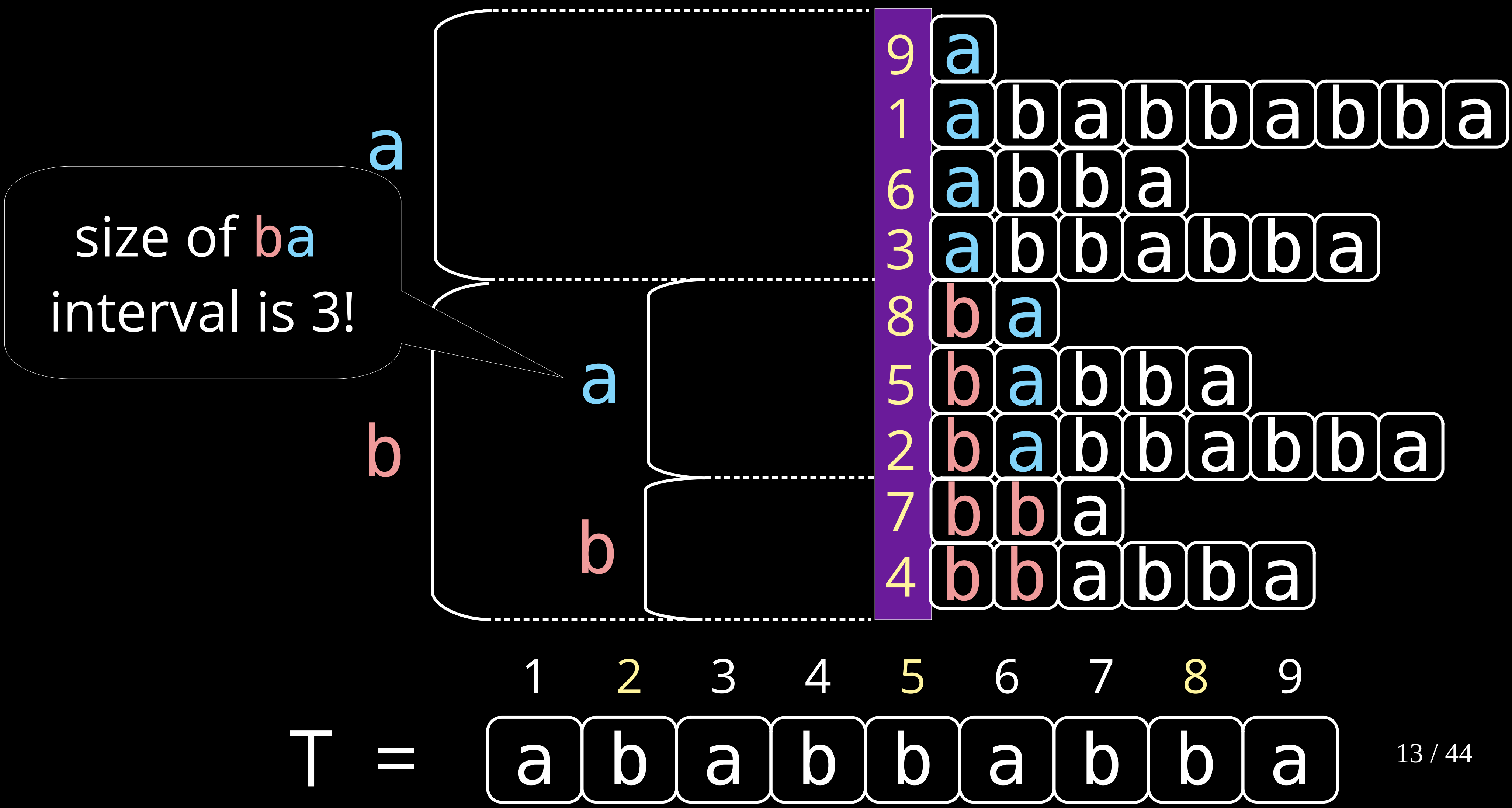


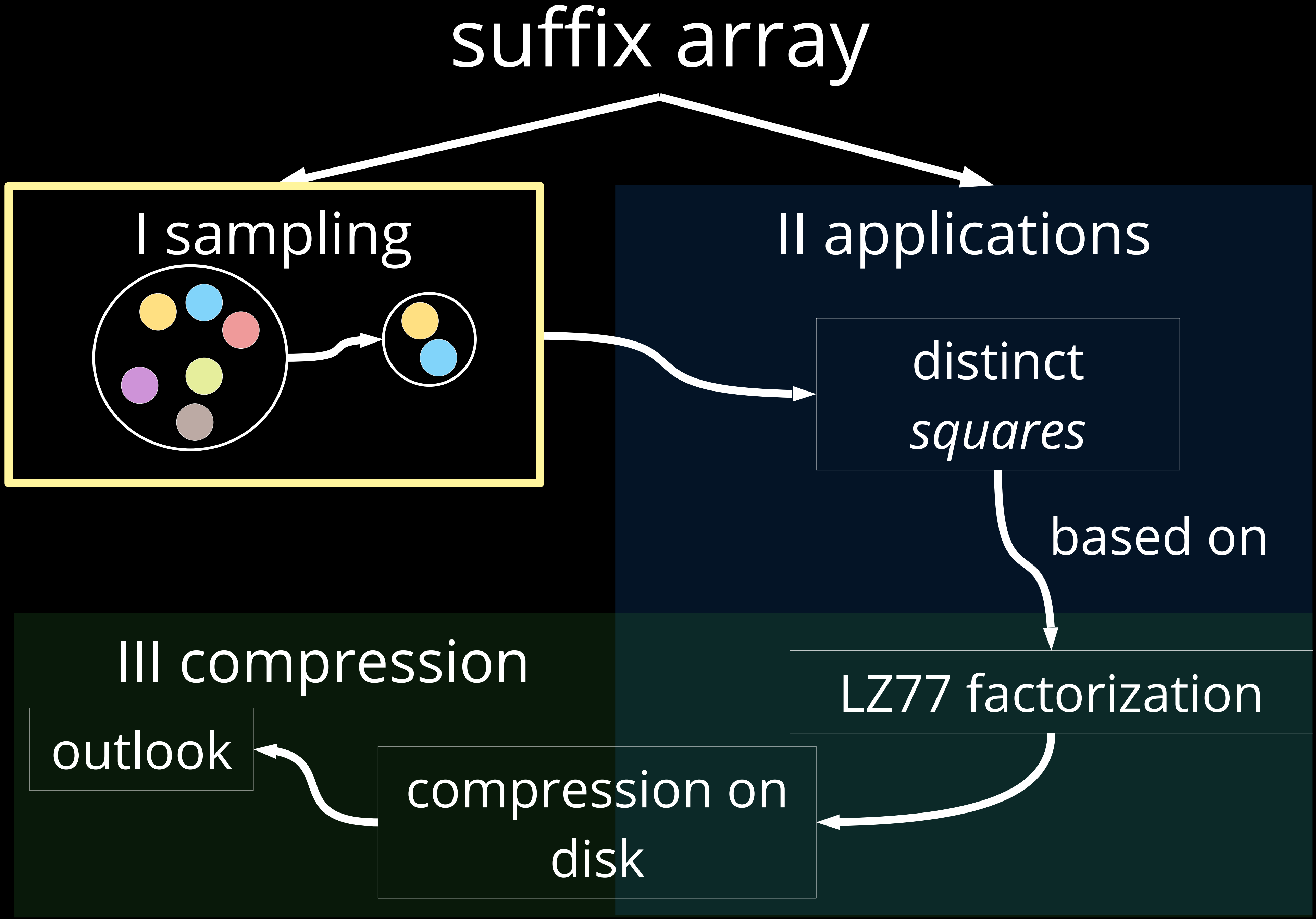
pattern matching : # ba = ?

pattern matching : # ba = ?



pattern matching : # ba = ?




suffix array
I sampling
II applications
distinct
squares
based on
III compression
LZ77 factorization
outlook
compression on
disk

I. sampling

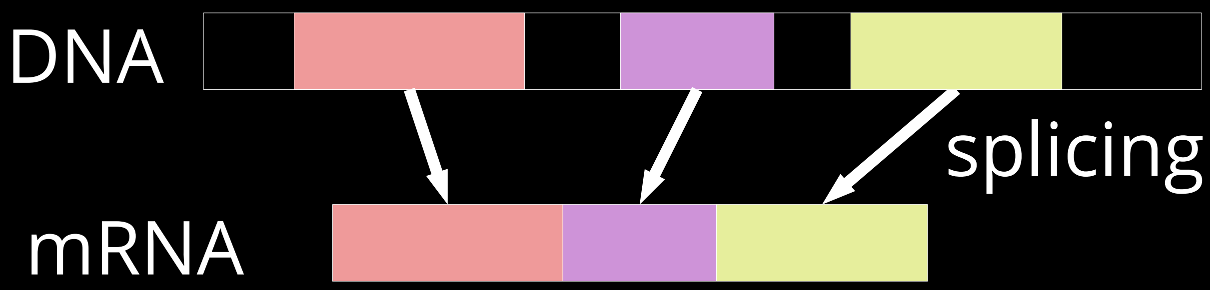
- assume we are only interested in indexing m text positions
- such a suffix array should
  - take $O(n)$ time and
  - $O(m)$ space, also during the construction
    ⇒ *sparse* suffix array

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

sparse suffix array

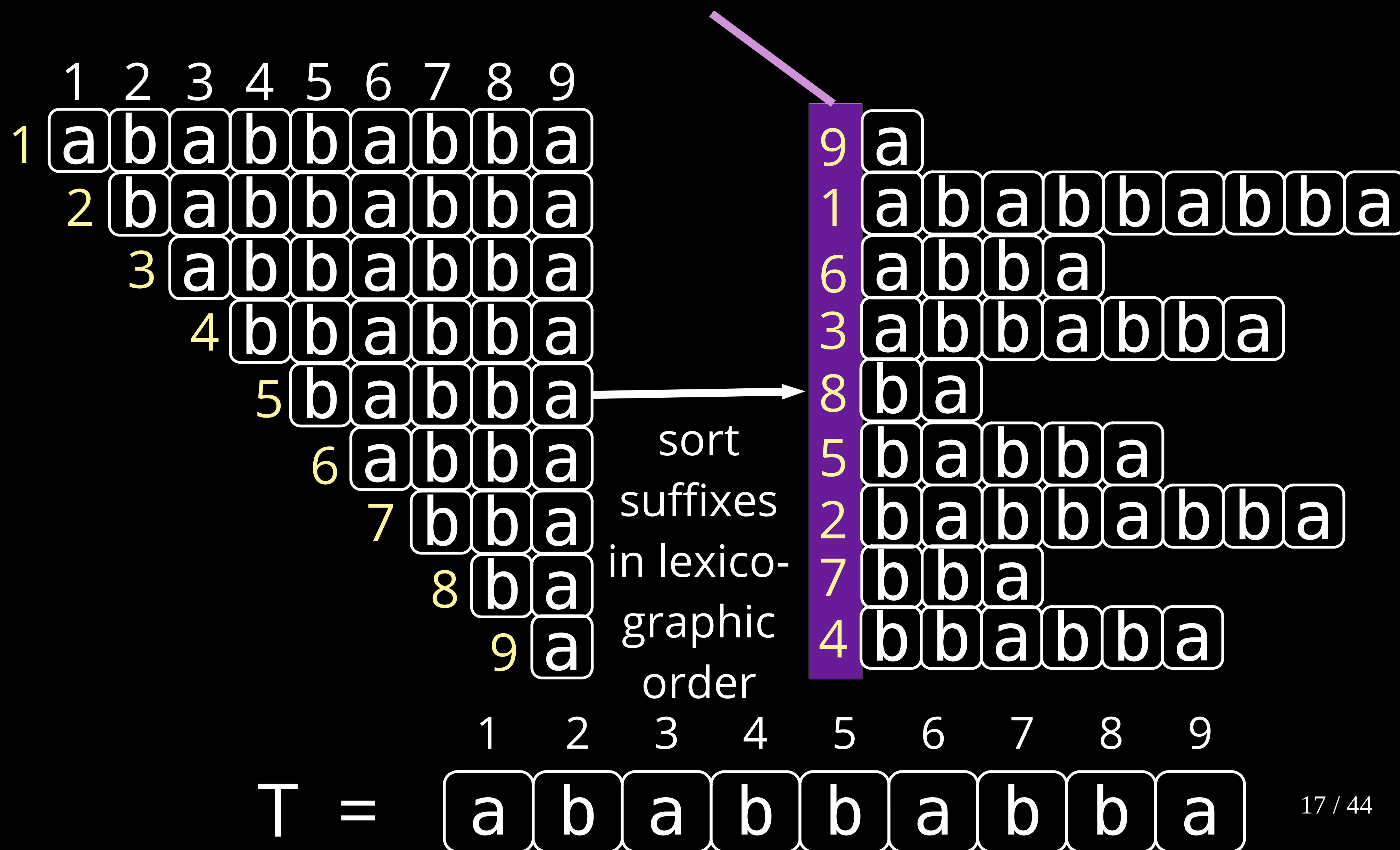
- sort only *m* suffixes
- but why?
  - only interested in word beginnings
  - coding sections of DNA sequences

Once upon a time ....

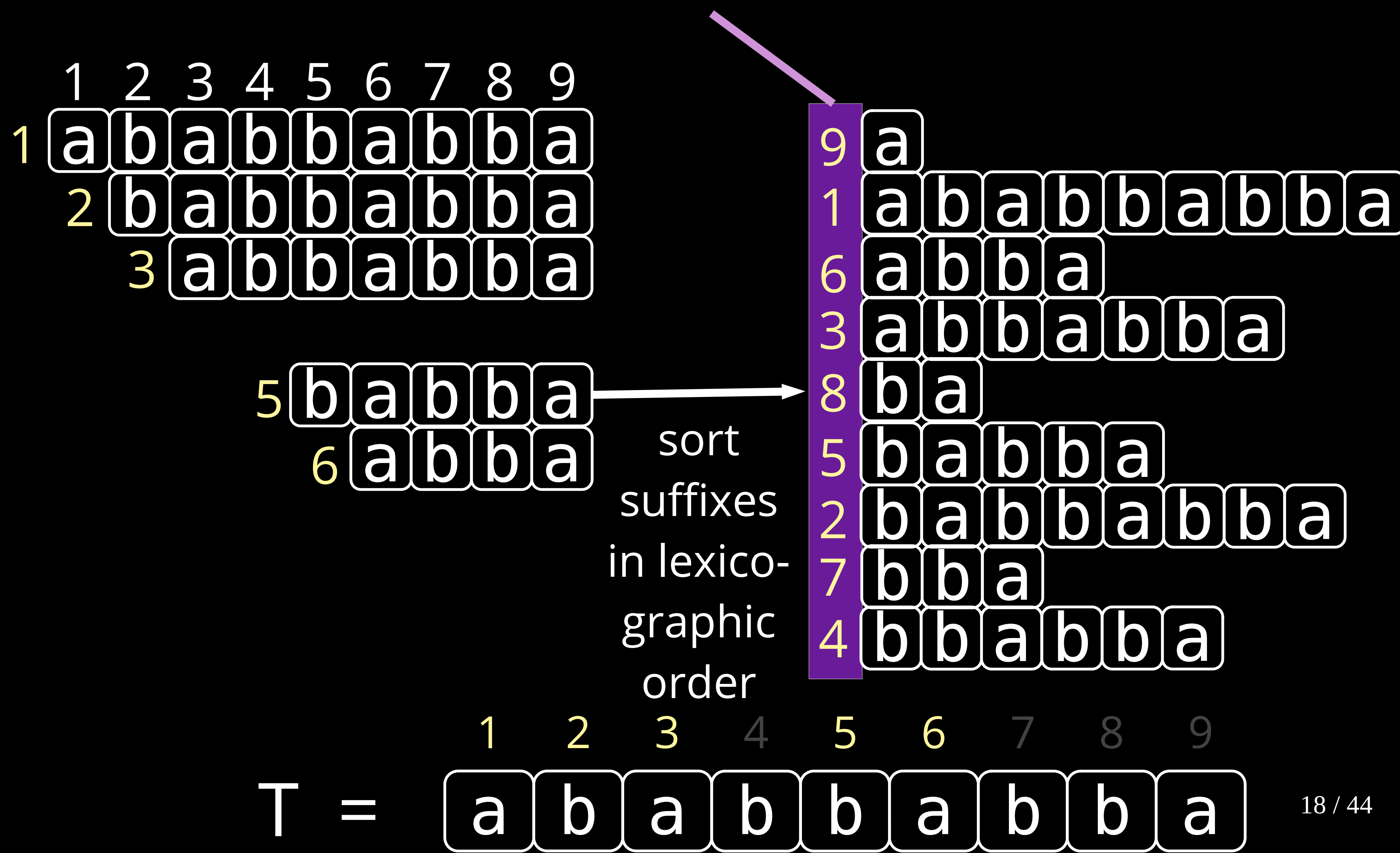


	1	2	3	4	5	6	7	8	9
T =	a	b	a	b	b	a	b	b	a

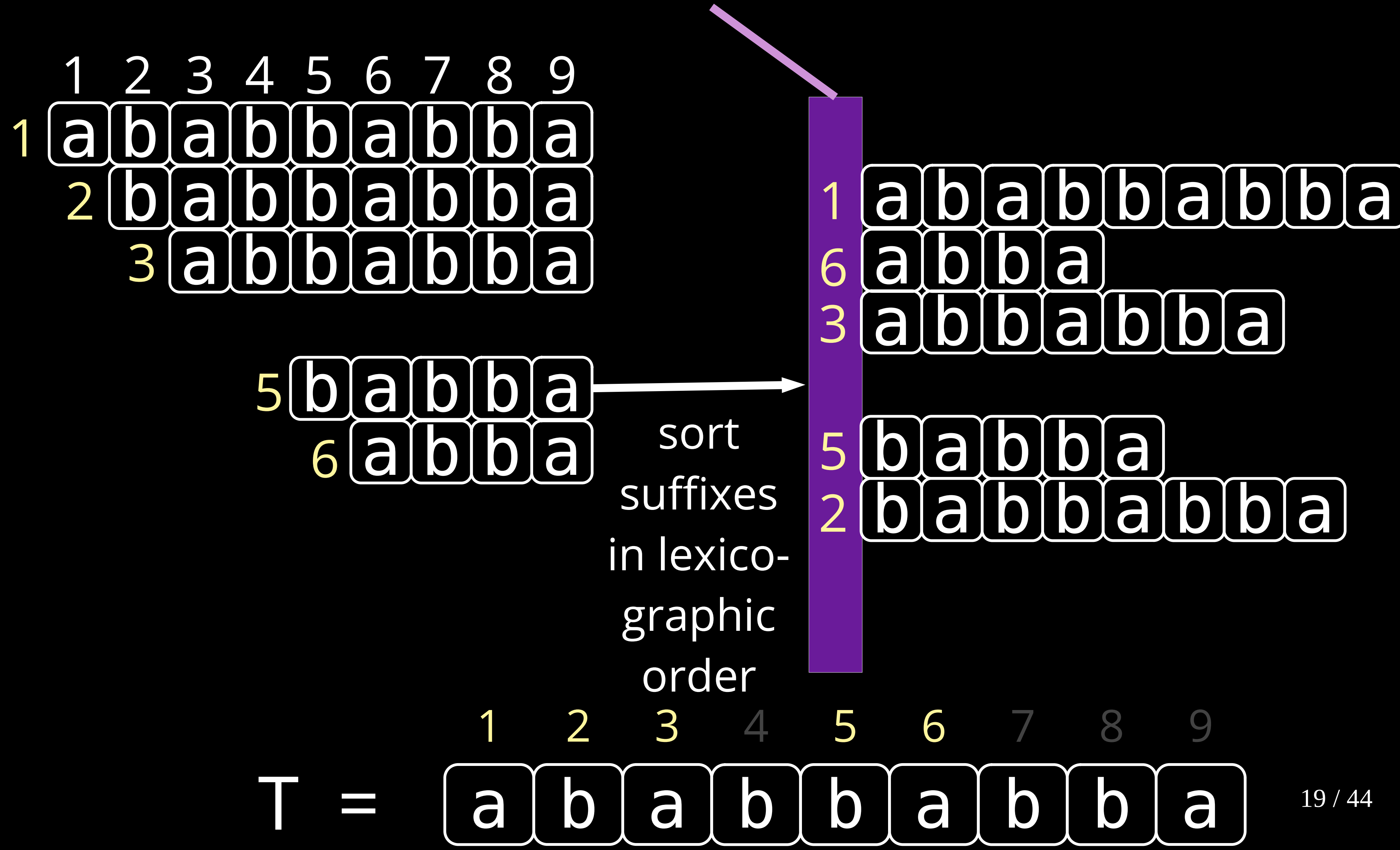
suffix array



suffix array



sparse suffix array



sparse suffix array

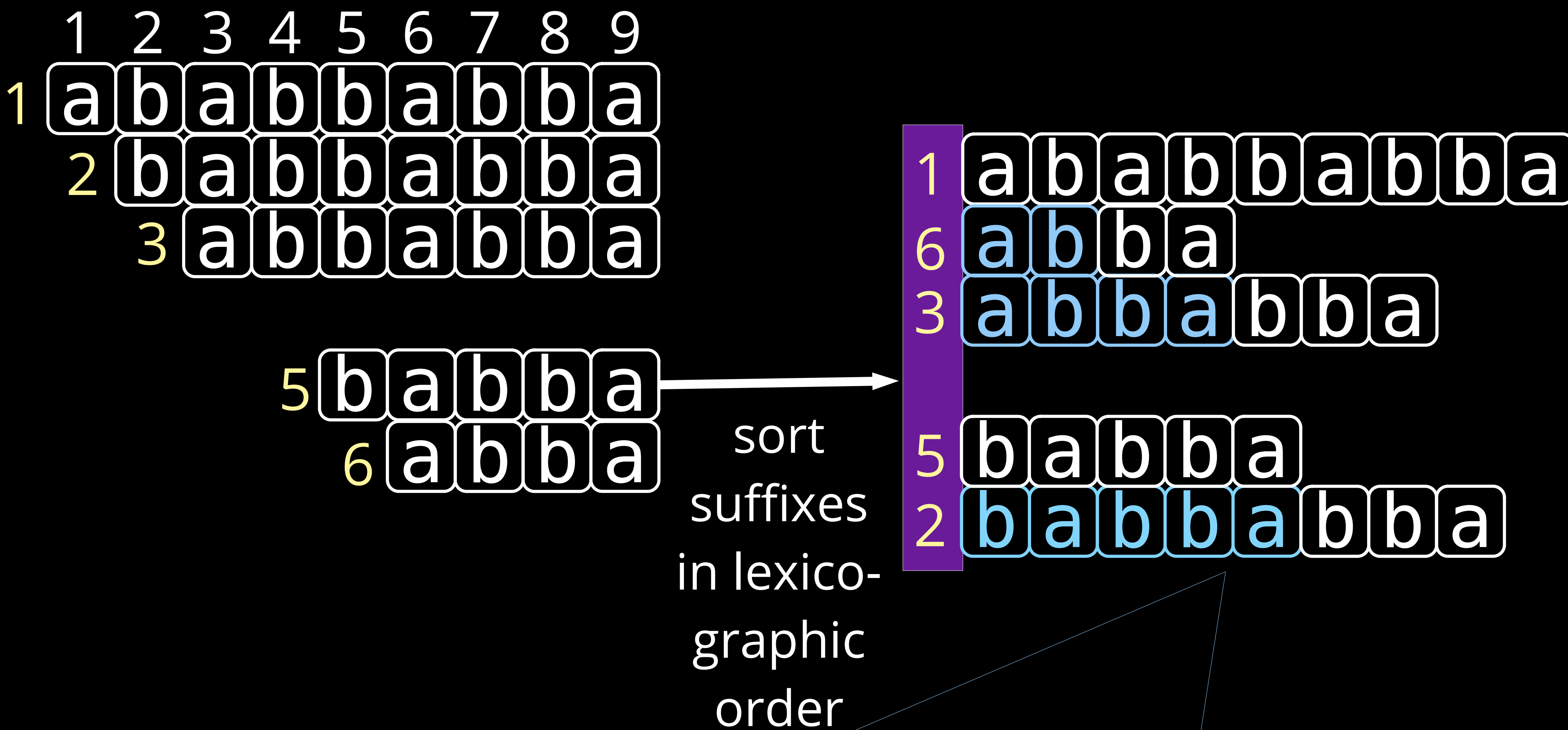
- can construct suffix array in $O(n)$ time and $O(n)$ space [Nong+ '11]
- how much time do we need to construct the sparse suffix array in $O(m)$ space?

naive: $O(mn)$ time (+ sorting integers)

- each suffix has a length of at most n
- sort all m suffixes ⇒ $O(mn)$ character comparisons

2 suffixes

sparse suffix array



LCP: length of the longest common prefix
with its lexicographic predecessor

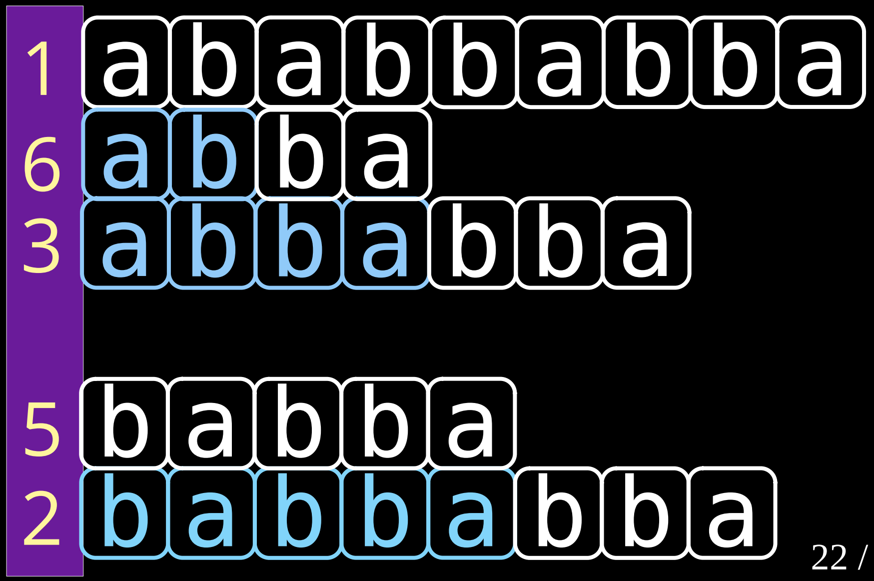
construction of the sparse suffix array

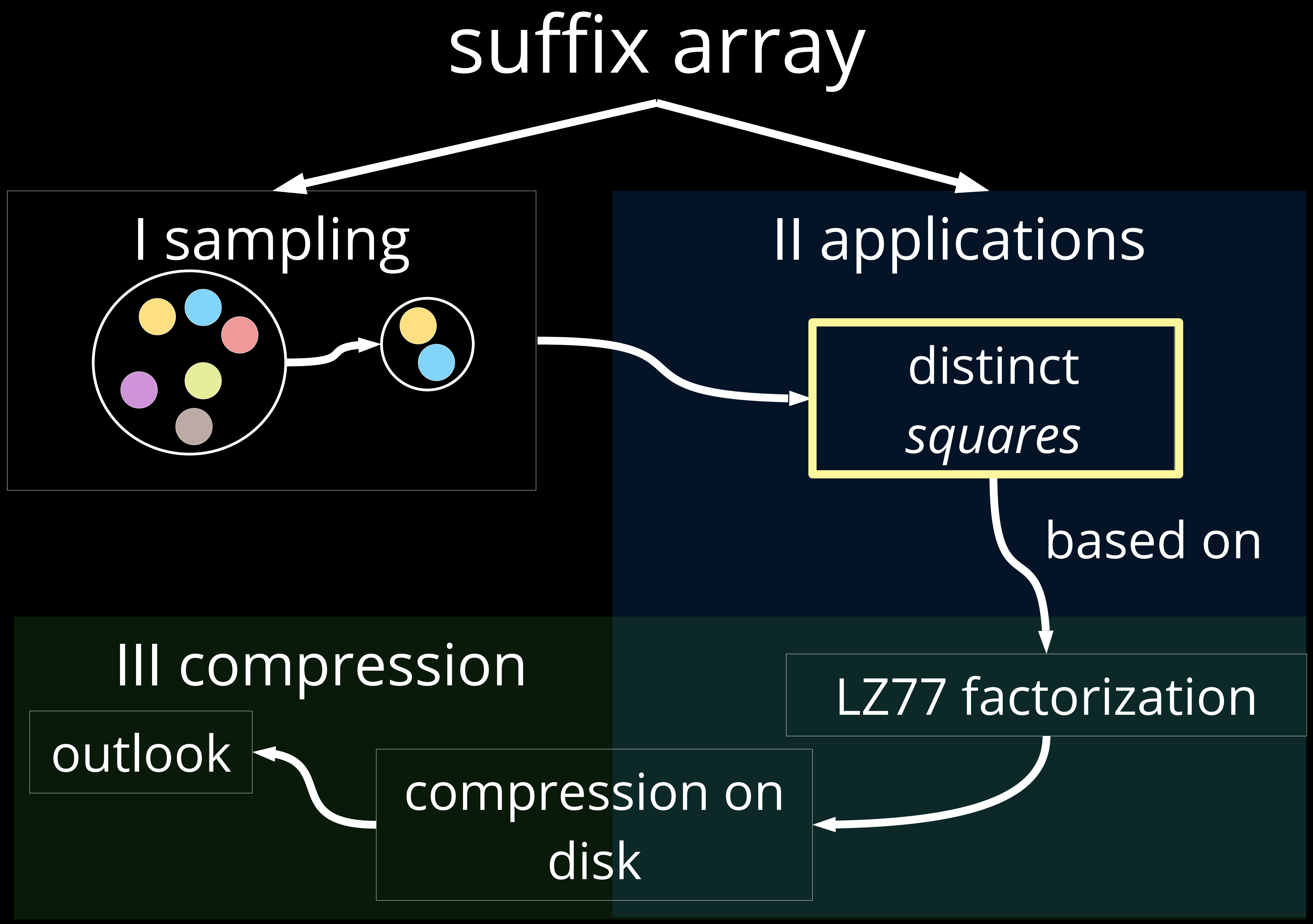
- c : sum of all LCPs
- need to compare c characters to determine the order of the suffixes

⇒ at least c time necessary

[TALG' 20]:

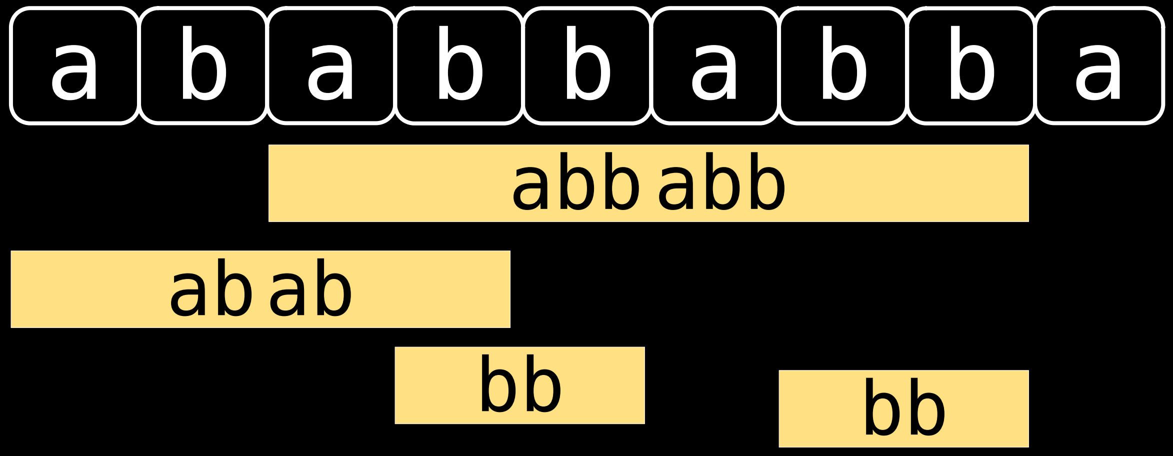
- O(m) space
- O($c \lg n + m \lg^2 n$) time
- fastest (deterministic) solution for small c




suffix array
I sampling
II applications
distinct
squares
based on
III compression
LZ77 factorization
outlook
compression on
disk

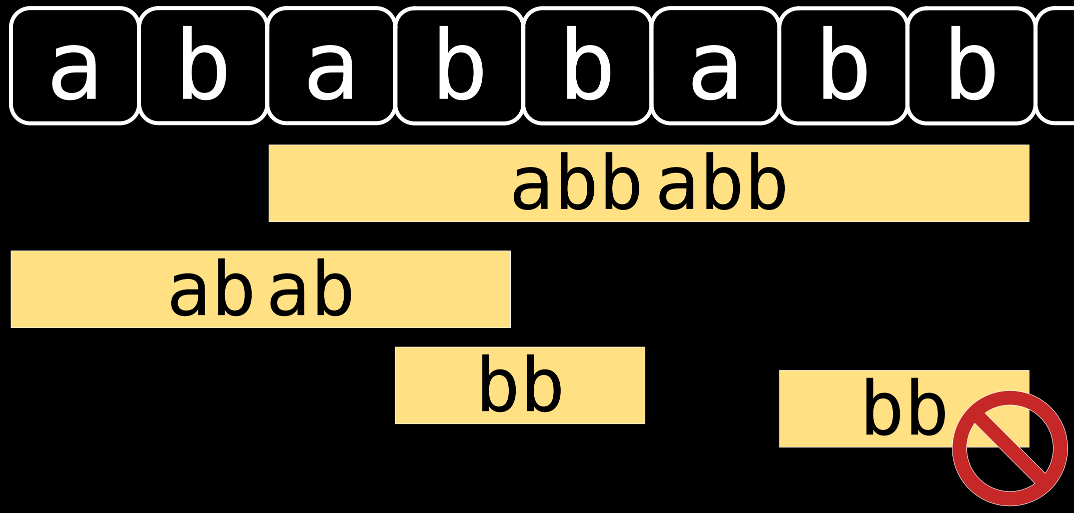
all distinct squares

- square: substring of the form A A



all distinct squares

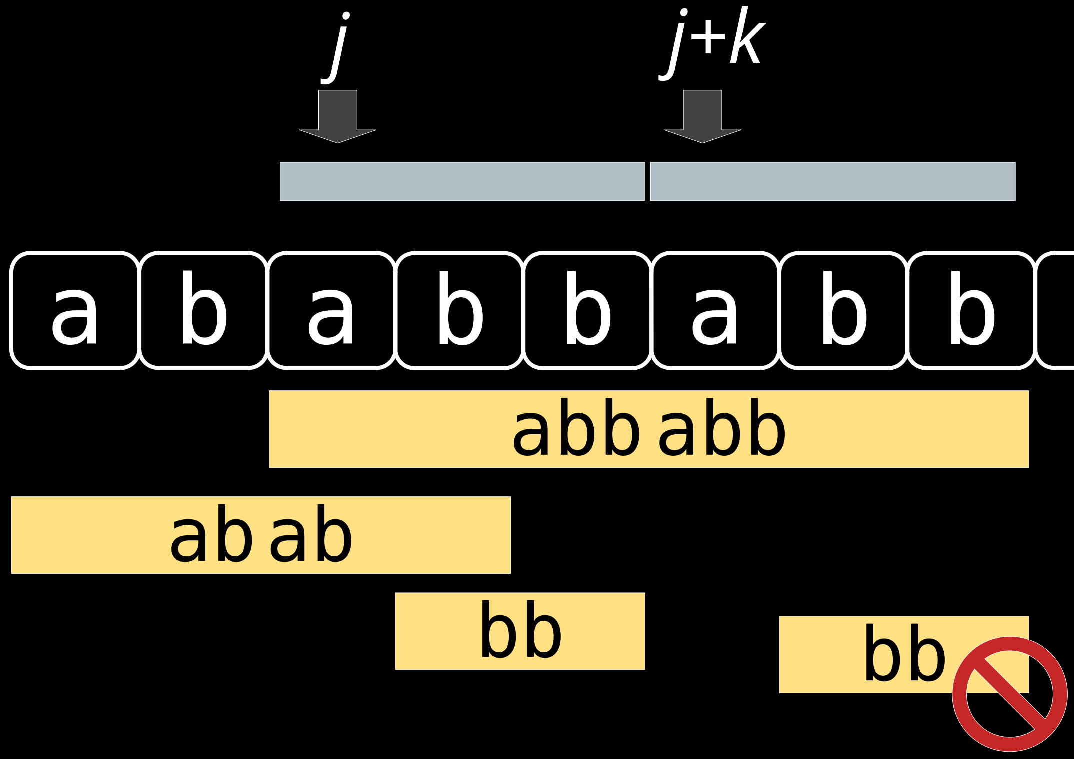
- square: substring of the form A A
- aim: list only all different squares (useful for DNA fingerprinting, etc.)
- at most $2n$ many different squares



naive approach

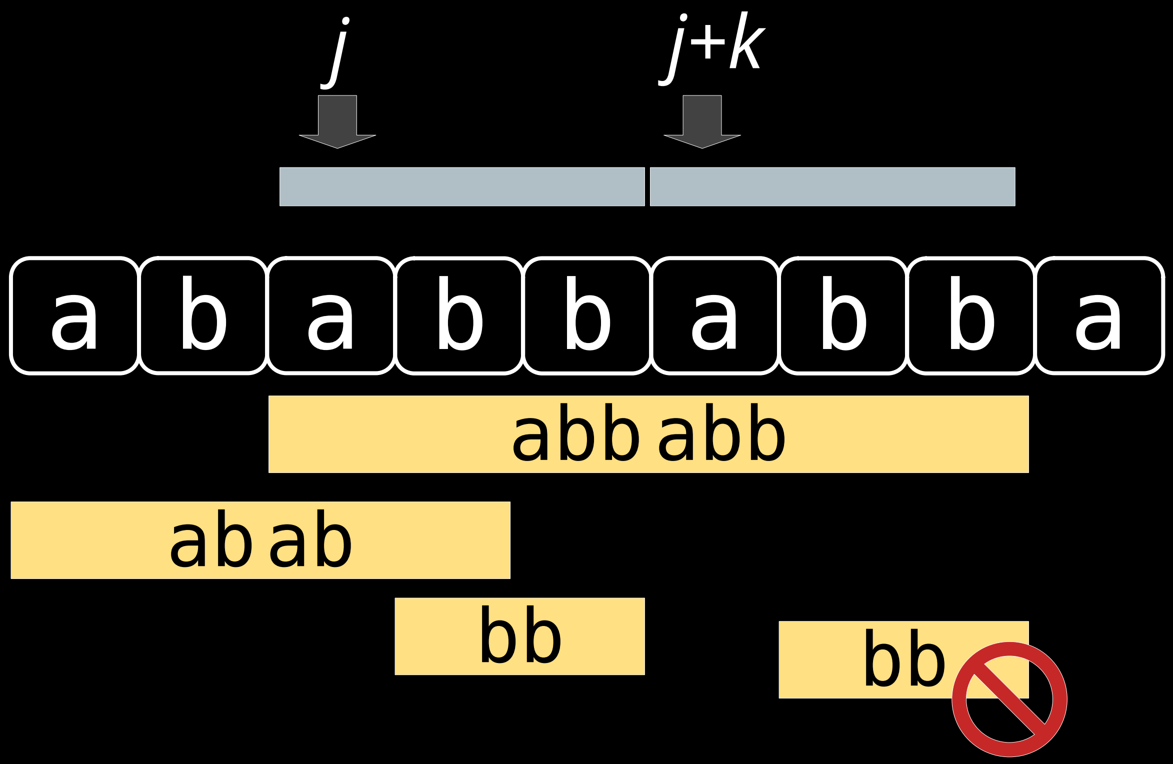
- test for all text positions j and lengths k if $T[j..j+k-1]$ and $T[j+k..j+2k-1]$ make a square

⇒ at least n^2 time



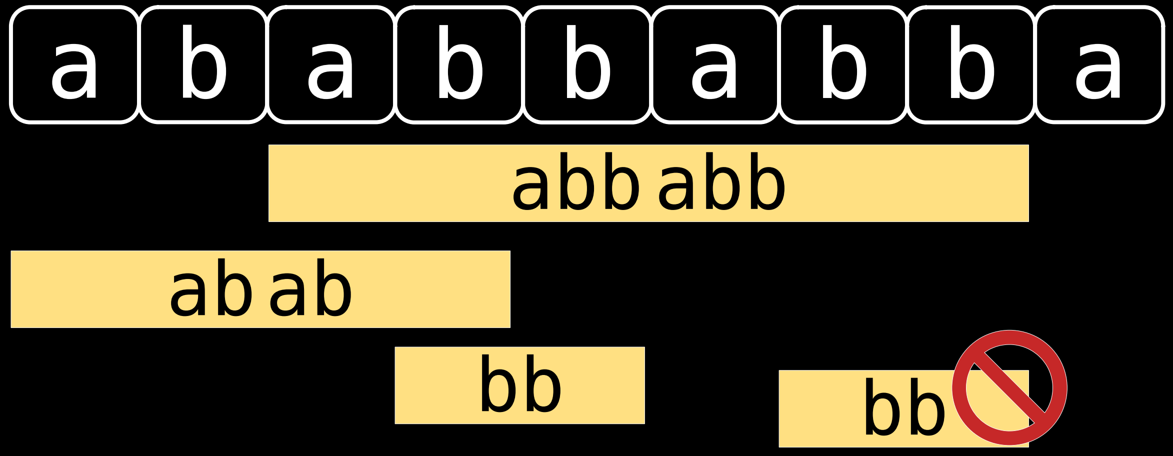
linear time algorithm

- [CPM '17] : O(n) time algorithm
- only choose specific j und k



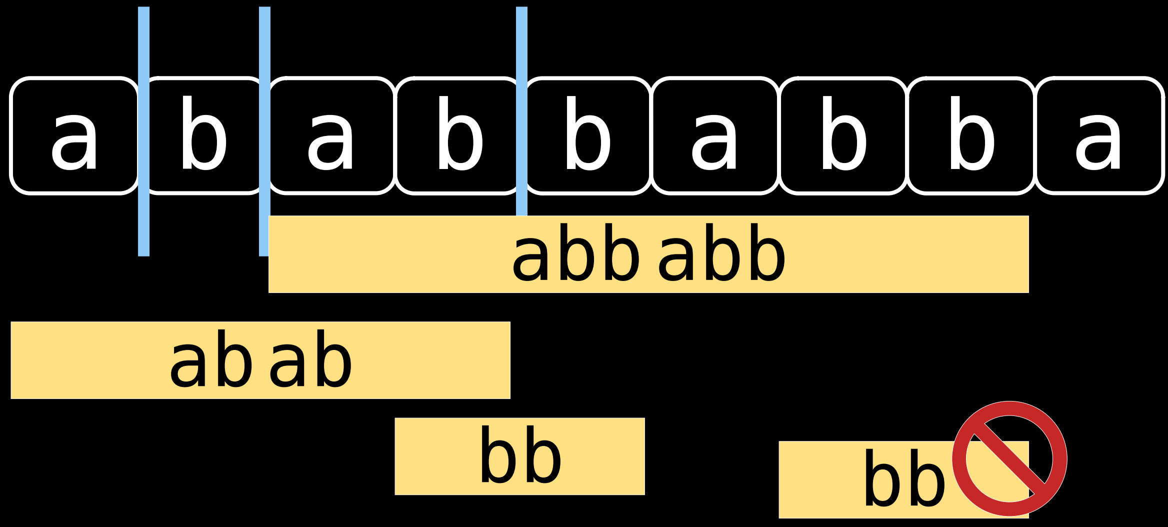
linear time algorithm

- idea: only report the leftmost ones



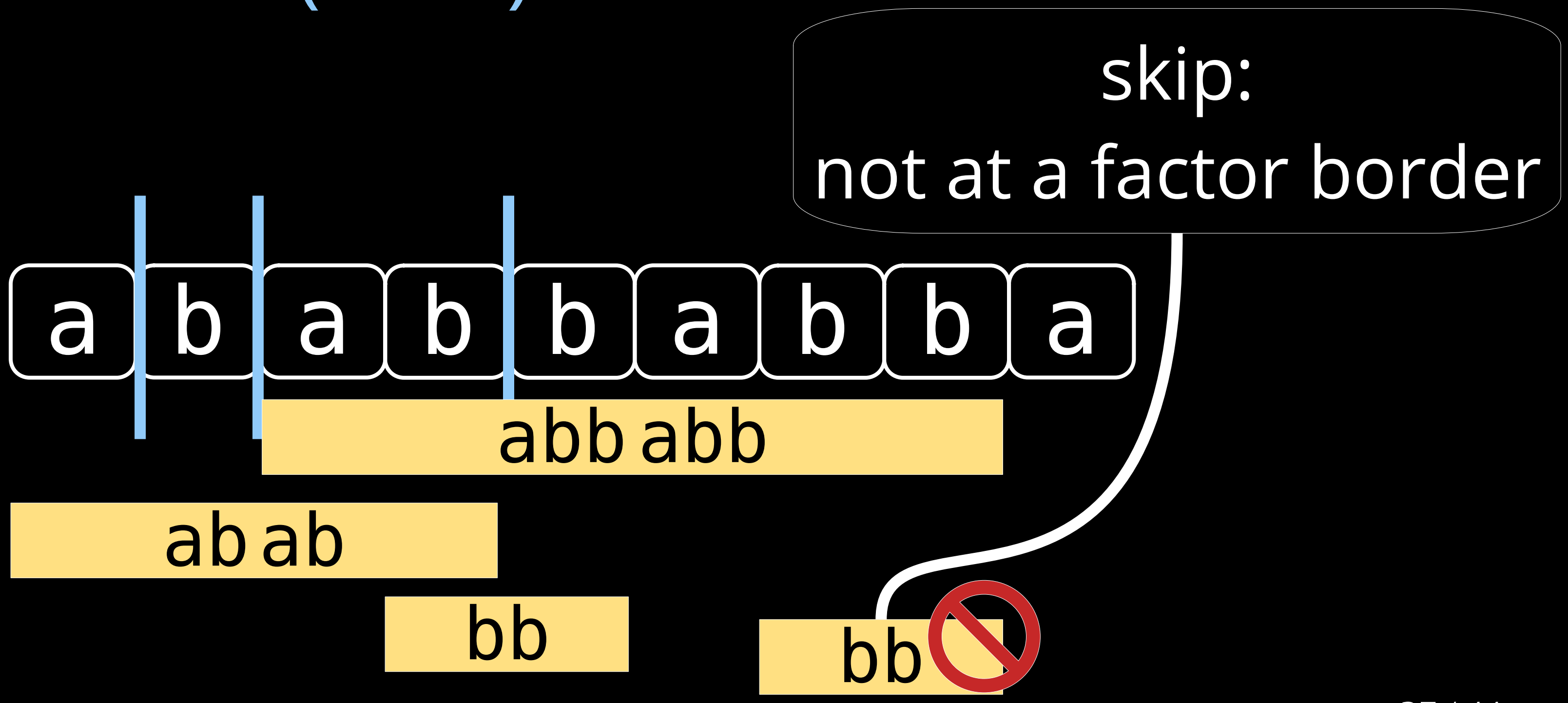
linear time algorithm

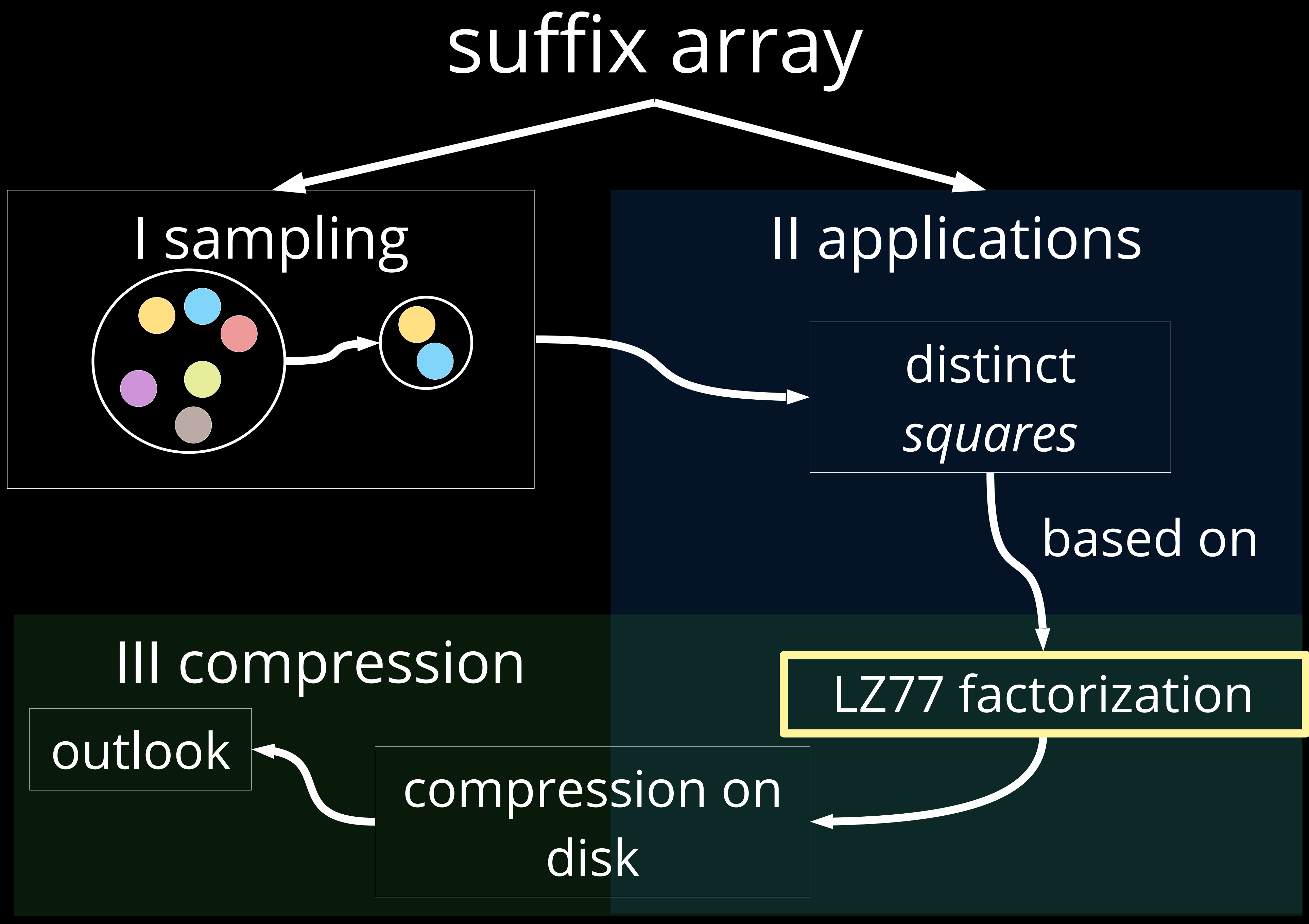
- idea: only report the leftmost ones
- find them with the
  Lempel-Ziv 77 (LZ77) factorization



linear time algorithm

- idea: only report the leftmost ones
- find them with the Lempel-Ziv 77 (LZ77) factorization




suffix array
I sampling
II applications
distinct
squares
based on
III compression
LZ77 factorization
outlook
compression on
disk

LZ77

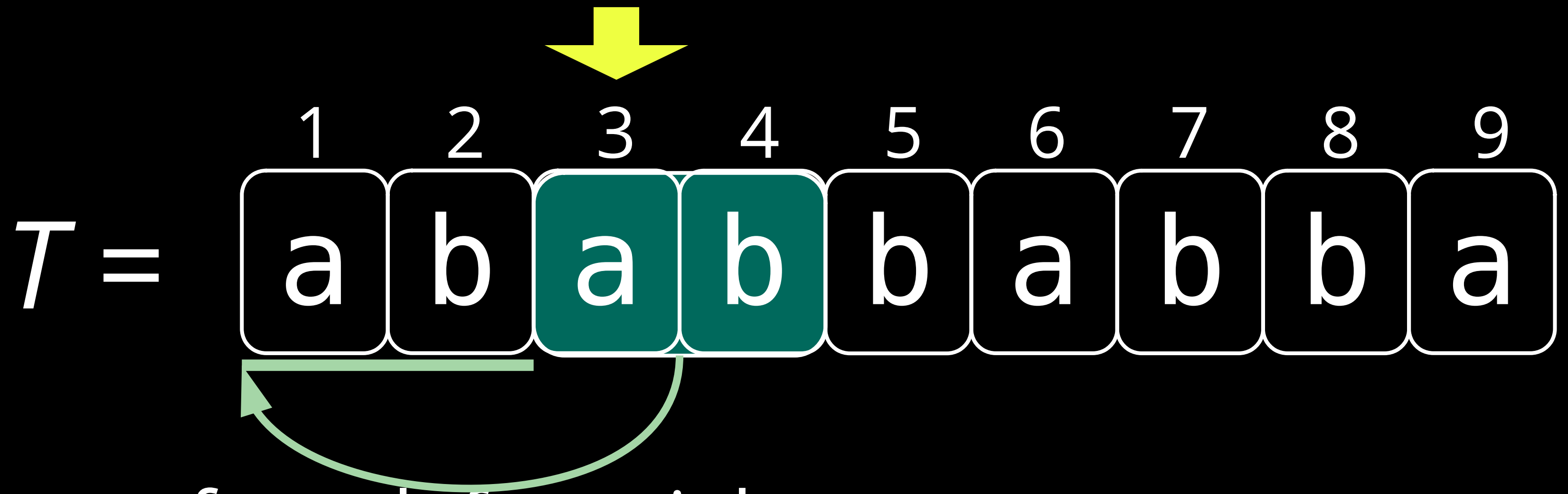
	1	2	3	4	5	6	7	8	9
T =	a	b	a	b	b	a	b	b	a

LZ77

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

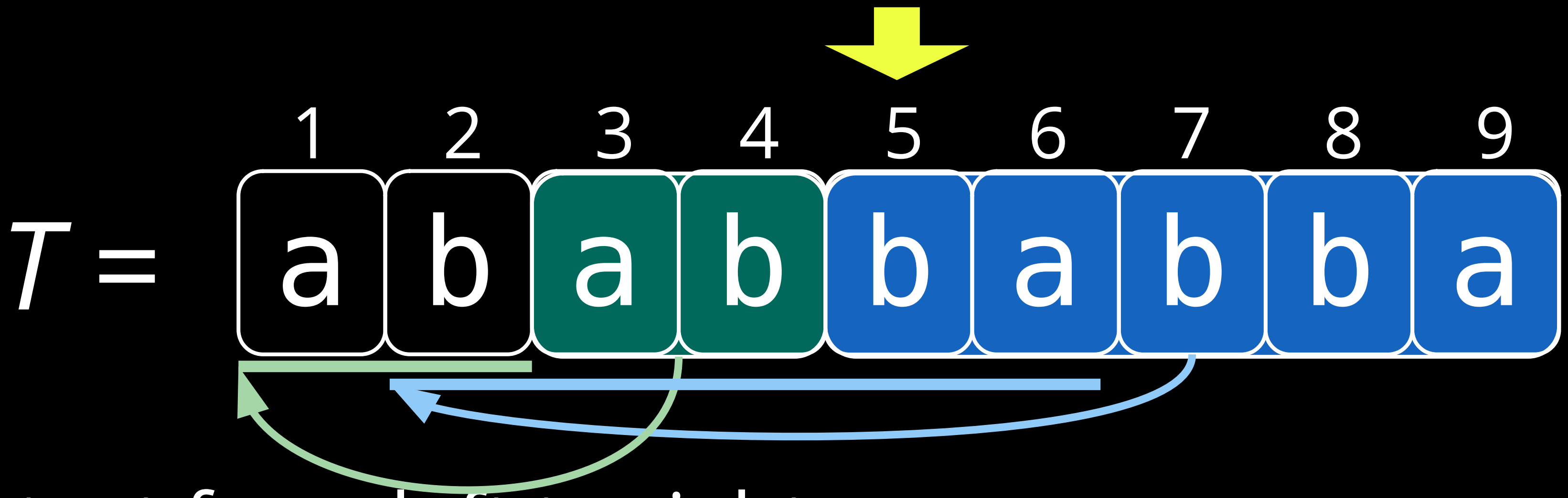
- read text from left to right

LZ77



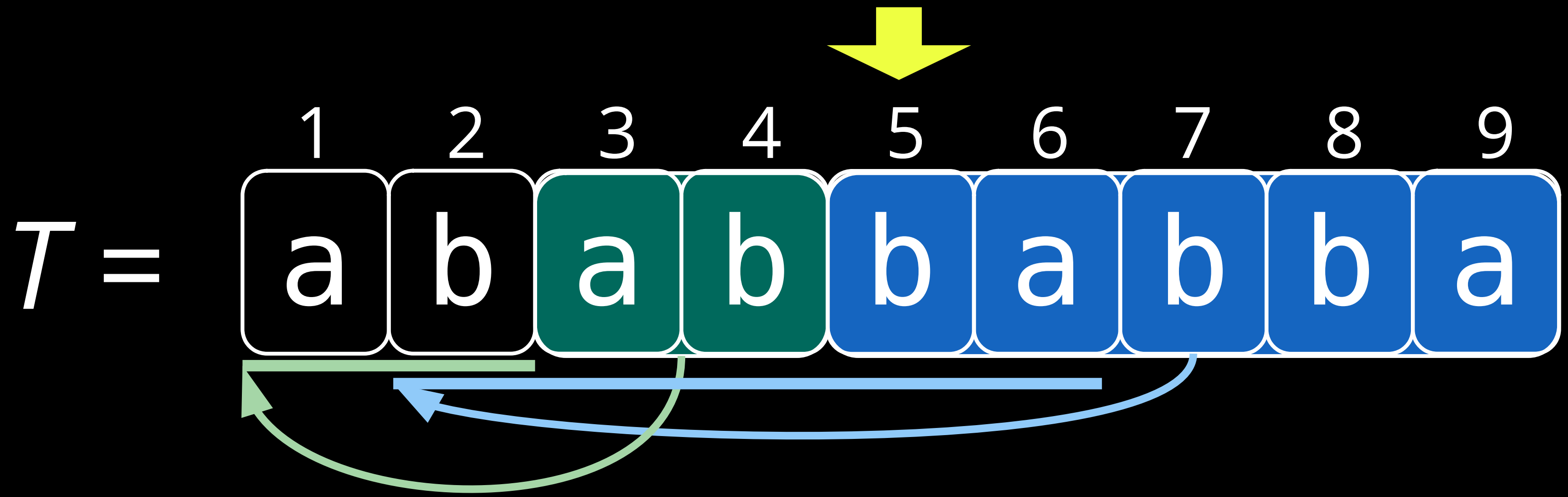
- read text from left to right
- replace longest repetition occurring in read part

LZ77



- read text from left to right
- replace longest repetition occurring in read part
- support overlapping

LZ77



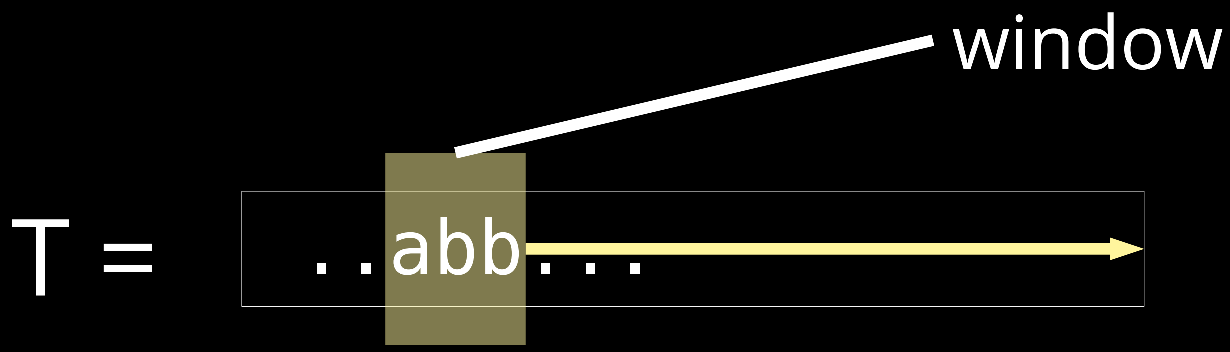
- read text from left to right
- replace longest repetition occurring in read part
- support overlapping

⇒ finds repetitions (and leftmost squares)

LZ77 compression

[Lempel, Ziv '77]

- high compression ratios
- popular: zip, gzip, 7zip, png, ...
- technique: window search



fixed window

search for repetitions only within a fixed window width

- memory-efficient
- but: far redundancies cannot be found

  e.g. compress 100 *identical* human genomes

```
repetitive data with a gap are overlooked repetitively
```

fixed window

search for repetitions only within a fixed window width

- memory-efficient
- but: far redundancies cannot be found

e.g. compress 100 *identical* human genomes

already forgotten

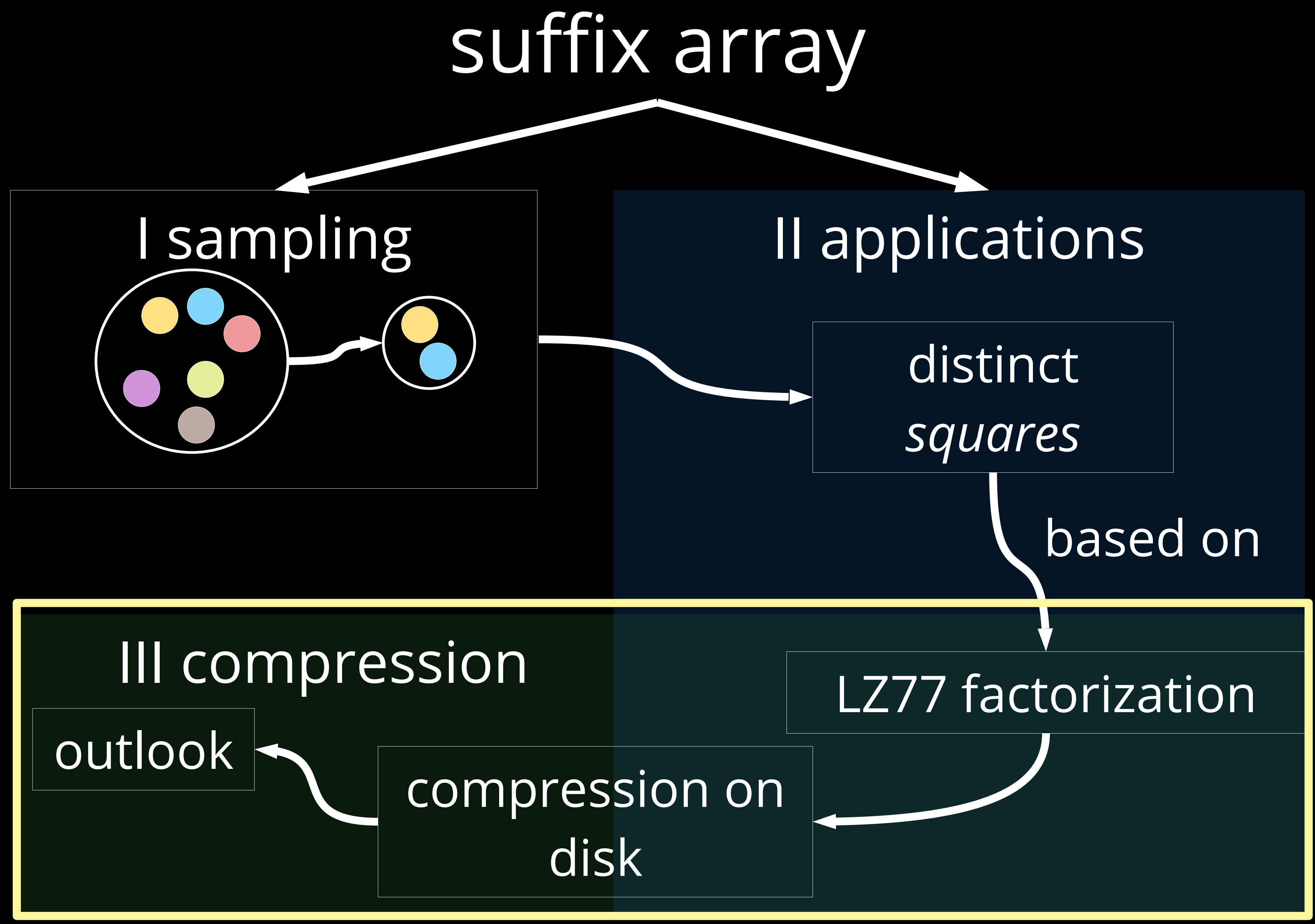
`repetitive data with a gap are overlooked repetitively`

without window

solution: work on the entire input!

- improves compression rate
- for 200 MiB repetitive DNA:

compressor	ratio (less better)	memory
`gzip -1`	30,73%	7 MiB
`gzip -9`	26,22%	7 MiB
LZ77 without window	4,05%	2900 MiB


suffix array
I sampling
II applications
distinct
squares
based on
III compression
LZ77 factorization
outlook
compression on
disk

III. compress big data

if the data does not fit into memory, work

1) with compact data structures for LZ77 [Algorithmica '18]

2) on hard disk [ESA '19]

III. compress big data

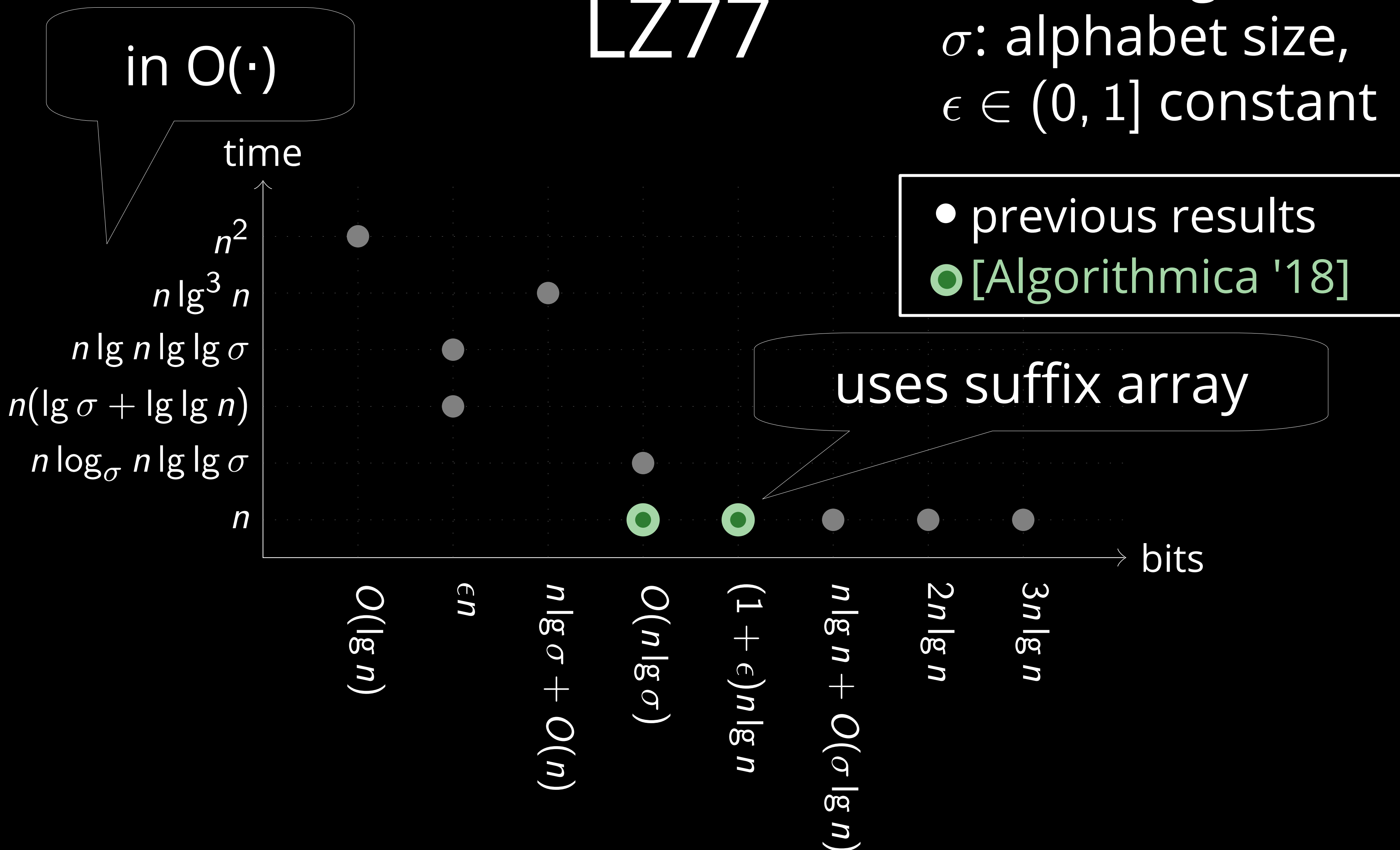
if the data does not fit into memory, work

1) with compact data structures for LZ77
[Algorithmica '18]

2) on hard disk [ESA '19]

LZ77

n: text length,
σ: alphabet size,
$\epsilon \in (0,1]$ constant



smallest space among linear-time algorithms!

III. compress big data

if the data does not fit into memory, work

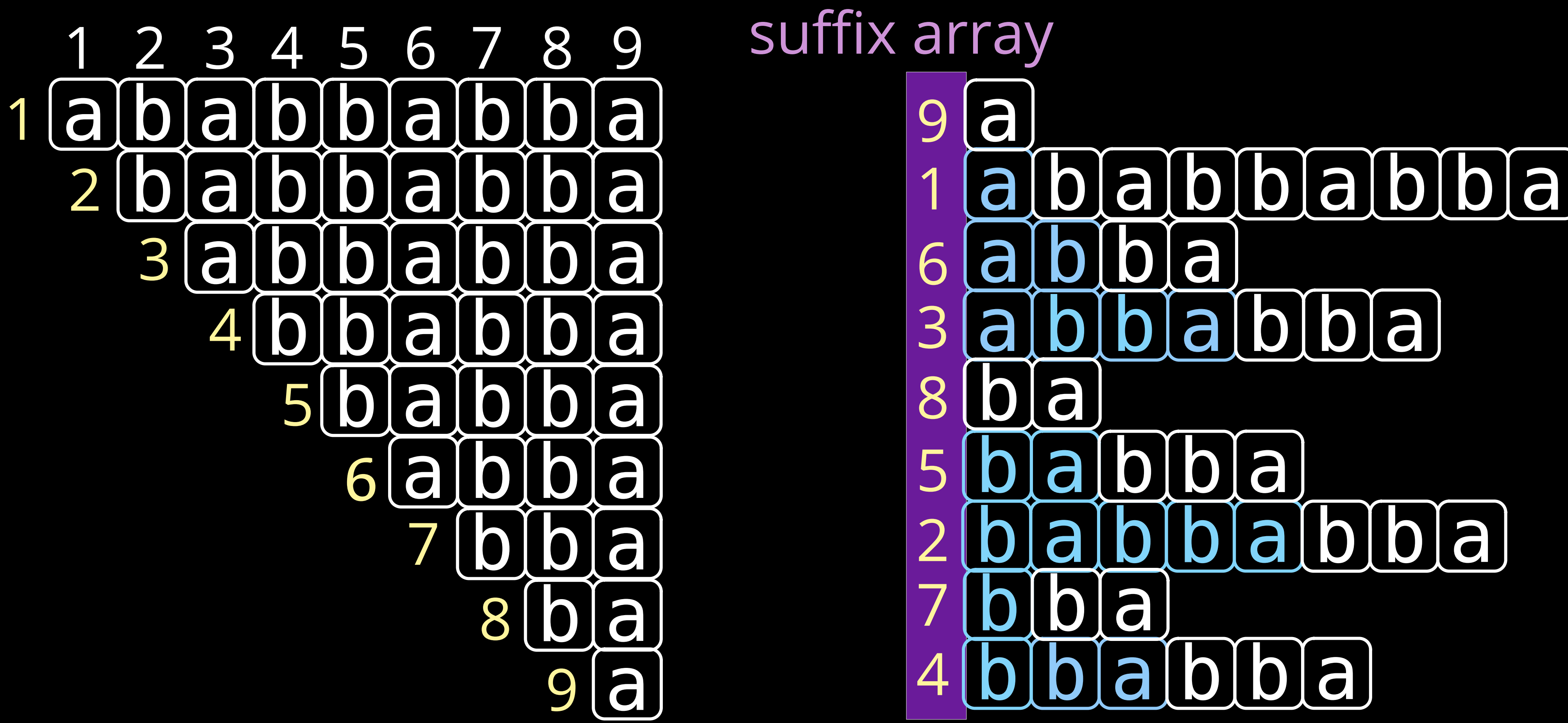
1) with compact data structures for LZ77 [Algorithmica '18]

2) on hard disk [ESA '19]

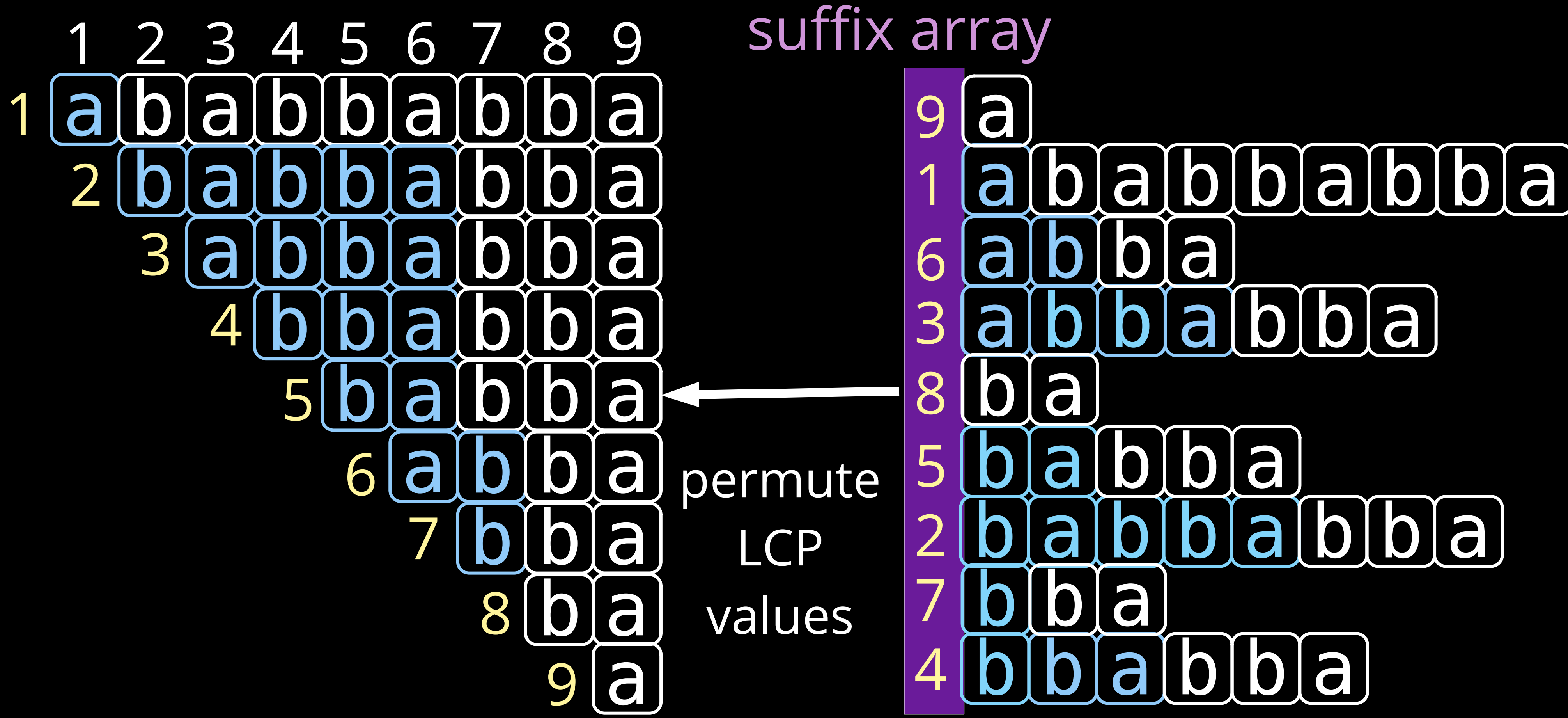
plcpcomp

- variation of LZ77
- works on hard drive for really large data
- idea:
  - search for longest factors first
  - permute LCPs of suffix array

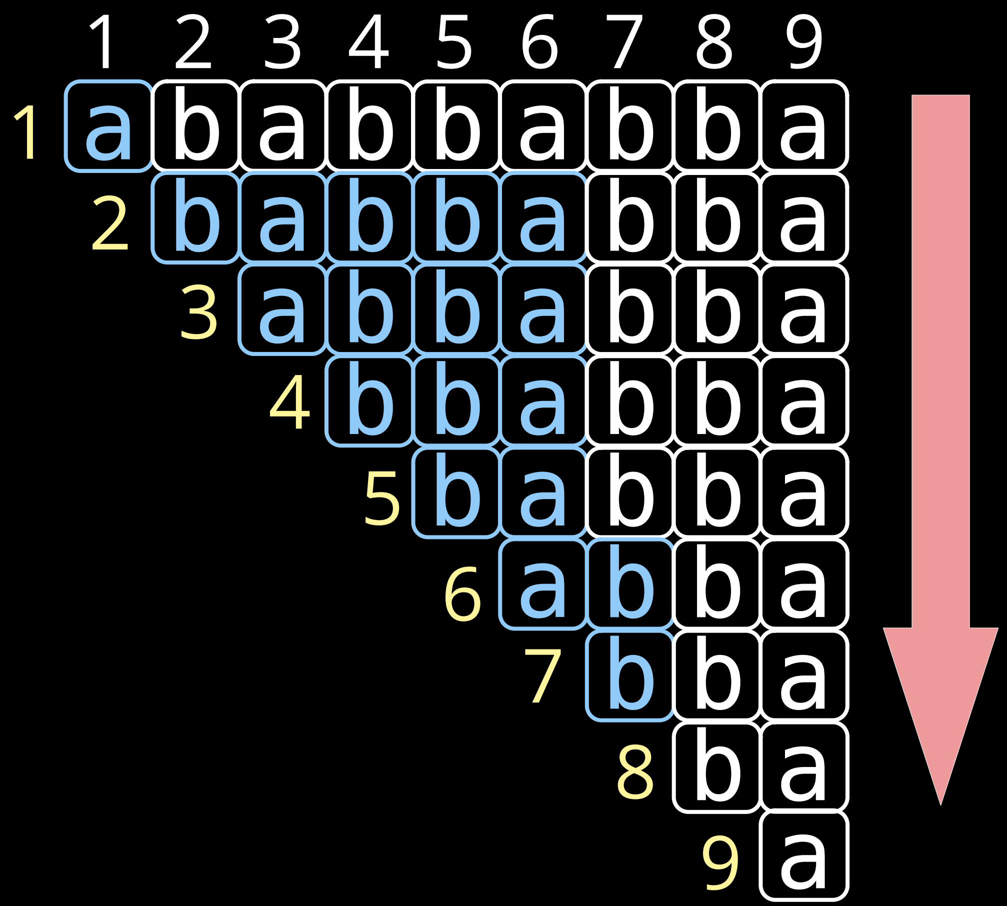
plcpcomp


1 2 3 4 5 6 7 8 9
1 a b a b b a b b a
2 b a b b a b b a
3 a b b a b b a
4 b b a b b a
5 b a b b a
6 a b b a
7 b b a
8 b a
9 a
suffix array
9 a
1 a b a b b a b b a
6 a b b a
3 a b b a b b a
8 b a
5 b a b b a
2 b a b b a b b a
7 b b a
4 b b a b b a

plcpcomp


1 2 3 4 5 6 7 8 9
1 a b a b b a b b a
2 b a b b a b b a
3 a b b a b b a
4 b b a b b a
5 b a b b a
6 a b b a
7 b b a
8 b a
9 a
permute
LCP
values
suffix array
9 a
1 a b a b b a b b a
6 a b b a
3 a b b a b b a
8 b a
5 b a b b a
2 b a b b a b b a
7 b b a
4 b b a b b a

plcpcomp



scan the permuted LCP values in text order

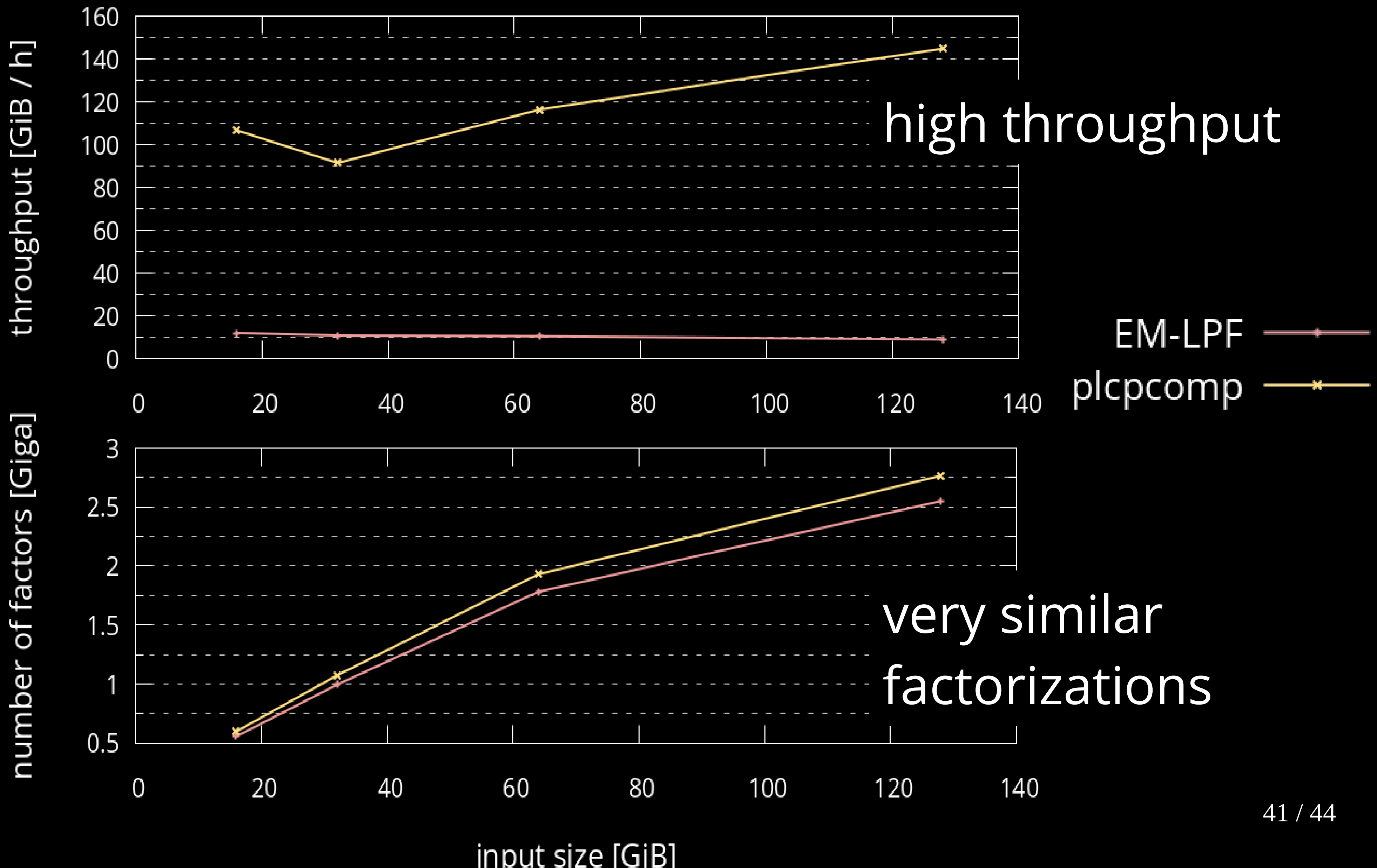
benchmark

1) plcpcomp [ESA '19]

2) EM-LPF [Kärkkäinen+ '14]

currently fastest LZ77 factorization algorithm
on disk space

web pages compression



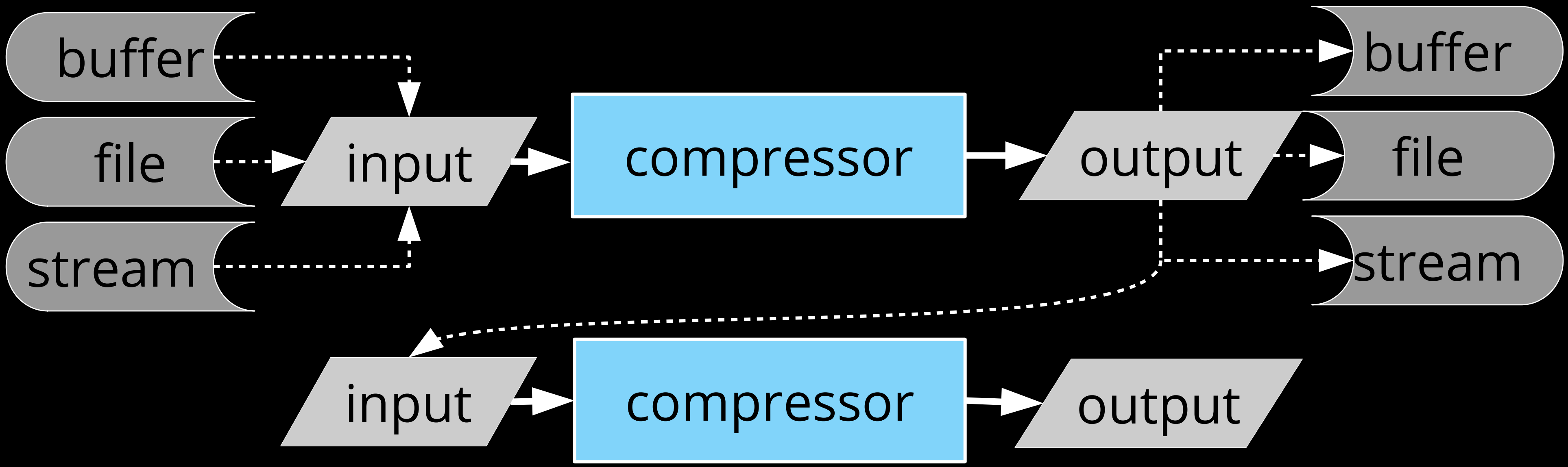
high throughput

very similar factorizations

http://tudocomp.org

C++ compression framework for

- benchmarks
- combinations of compressors



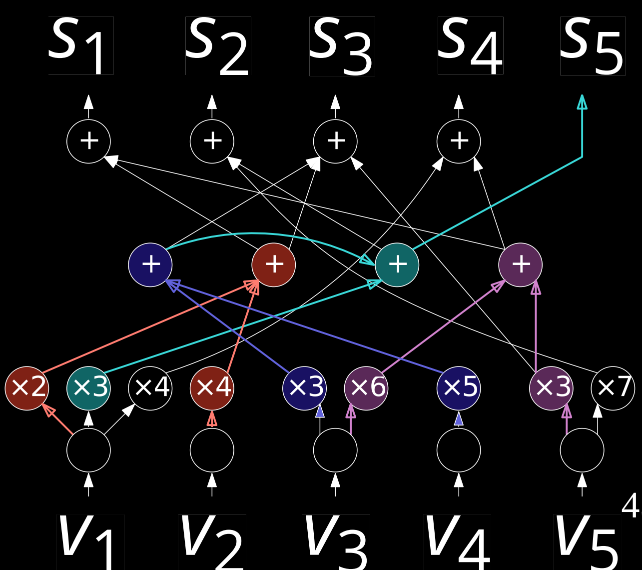
[SEA '17]

current research

compression not only for strings, but also for 2D structures like matrices

faster matrix multiplication
(multi-purpose: e.g., deep learning)

$$\begin{bmatrix} 2 & 4 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 7 \\ 2 & 4 & 3 & 5 & 3 \\ 4 & 0 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 0 \end{bmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{pmatrix}$$



why Japan?

- high number of collaborators
- good research network
- many domestic conferences