Combinatorial algorithms for grammar-based text compression

Shunsuke Inenaga
Kyushu University, Japan

Agenda

- Highly Repetitive Strings
- Grammar-based String Compression
- Straight-Line Program (SLP)
- Compressed String Processing (CSP) on SLP
- Reviews on CSP Algorithms and Related Results
- Conclusions and Future Work

Highly Repetitive Strings (HRSs)

HRSs are strings that contain a lot of repeats.
Repeats may occur separately (not necessarily tandem).

Examples of HRSs are:

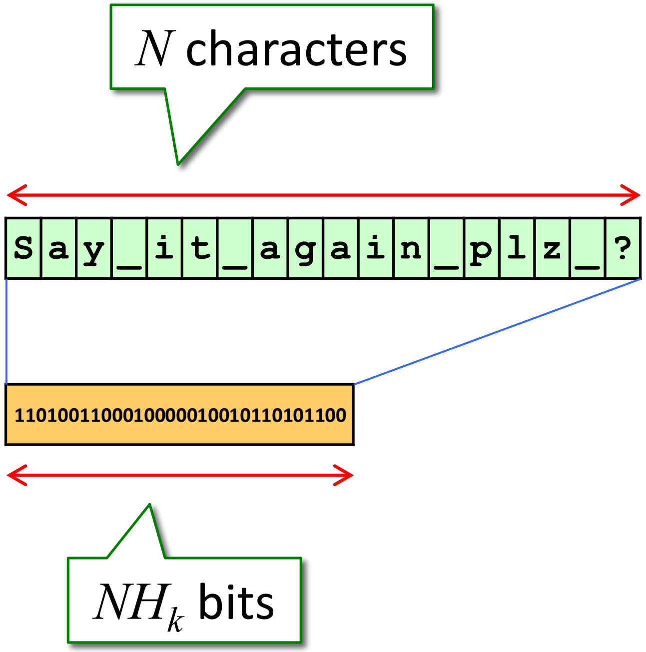
- Collection of DNA sequences of same species (two human genomes are 99.9% same)
- Software repositories (GitHub)
- Versioned documents (Wikipedia)
- Transaction logs

ID	DNA sequences
1	CATCTCCATCATCACCACCCTCCTCCTCAT...
2	CATCCCCATCATCACCACCCTCCTCCTCAT...
3	CATCTCCATCATCACTACCCTCCTCCTCAT...
4	CATCTCCATAATCACCACCCTCCTCCTCAT...
5	CATCTCCATCATCACCACCCTCCTACTCAT...
6	CATCTCCATCAACACCACCCTCCTCCTTAT...
...	...

Statistical Compressors vs. HRS

H_k : k-th order empirical entropy ($k < N$)

The k-th order empirical entropy H_k captures the dependence of symbols upon their k-long context.

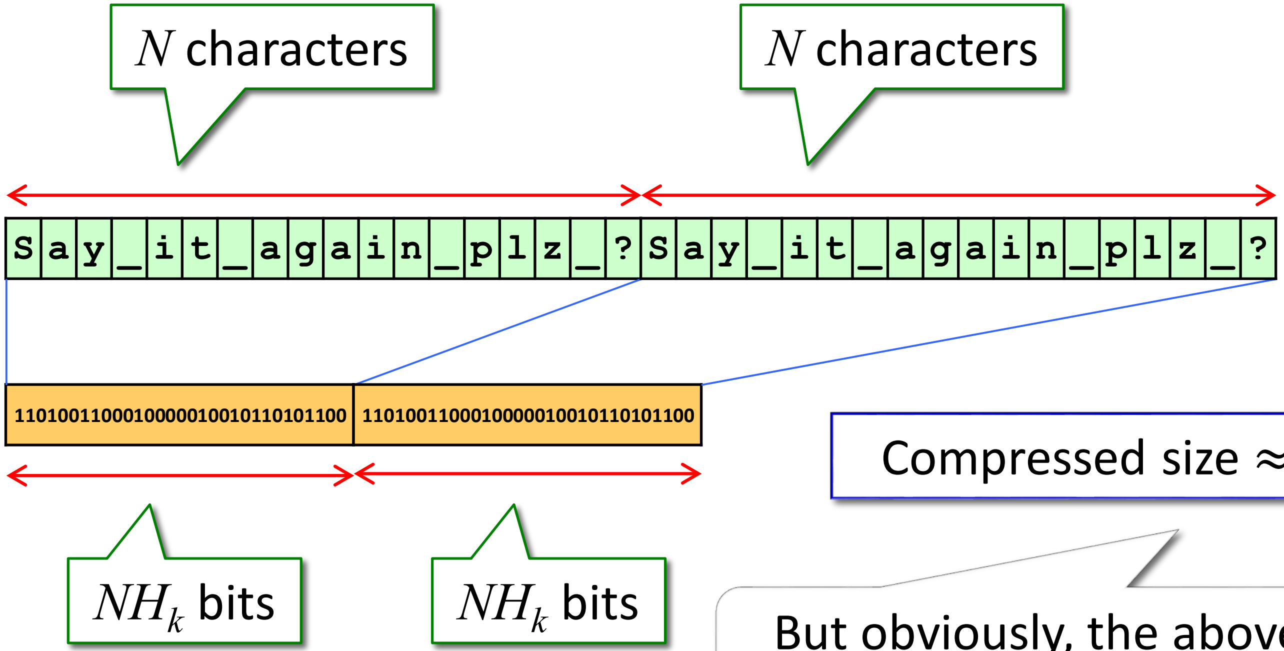


N characters

NH_k bits

Statistical Compressors vs. HRS

The k-th order entropy model does not capture repetitiveness very well [Kreft & Navarro 2013].



Compressed size ≈ $2NH_k$ bits

But obviously, the above repetitive string can be represented with $NH_k+O(1)$ bits!

Grammar-based Compression [Keiffer & Yang 2000]

String w

`abbbaabbaabbbaabbbbbabbb`

Grammar Transform

CFG G

Derives only w

$S \rightarrow ACBBEA$
$A \rightarrow D$**`b`**
$B \rightarrow C$**`b`**
$C \rightarrow$ **`a`**D
$D \rightarrow$ **`a`**E
$E \rightarrow$ **`bb`**

Grammar-compression in the Chomsky normal form is called an SLP (Straight Line Program)

Encoding

Encoding for G

0100010100010110101011101010110

Grammar-based Compression [Keiffer & Yang 2000]

String w

abbbaabbaabbbaabbbbbabbb

Grammar Transform

CFG G

Derives only w

$S \rightarrow ACBBEA$
$A \rightarrow D\mathtt{b}$
$B \rightarrow C\mathtt{b}$
$C \rightarrow \mathtt{a}D$
$D \rightarrow \mathtt{a}E$
$E \rightarrow \mathtt{bb}$

Grammar-compression in the Chomsky normal form is called an SLP (Straight Line Program)

This Talk

Encoding

Encoding for G

0100010100010110101011101010110

Straight Line Program (SLP)

An SLP is a sequence of n productions

$$X_1 \rightarrow expr_1,\ X_2 \rightarrow expr_2,\ \cdots,\ X_n \rightarrow expr_n$$

- $expr_i = a \qquad (a \in \Sigma)$
- $expr_i = X_l X_r \quad (l, r < i)$

SLP is grammar-compression in the Chomsky normal form.

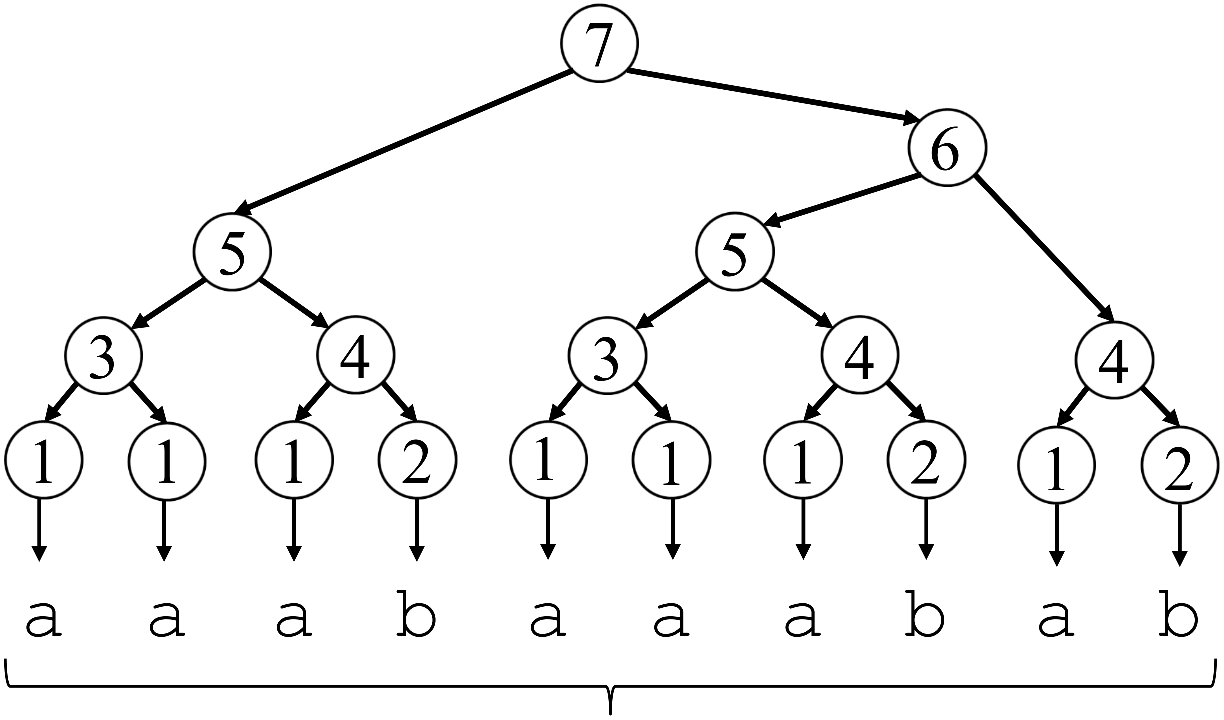
- ✓ SLP is a widely accepted model for the outputs of grammar-based compressors.
- ✓ The size of the SLP is the number n of productions.

Example of SLP

SLP S

$$X_7 \rightarrow X_5\, X_6$$
$$X_6 \rightarrow X_5\, X_4$$
$$X_5 \rightarrow X_3\, X_4$$
$$X_4 \rightarrow X_1\, X_2$$
$$X_3 \rightarrow X_1\, X_1$$
$$X_2 \rightarrow \mathtt{b}$$
$$X_1 \rightarrow \mathtt{a}$$

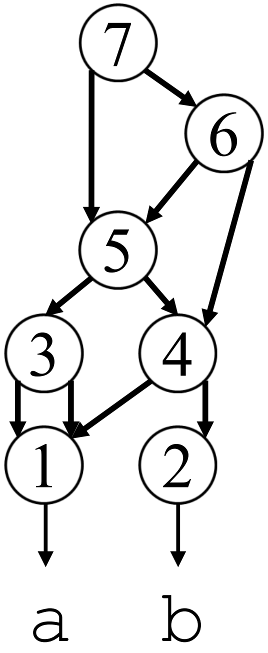
Derivation tree T of SLP S



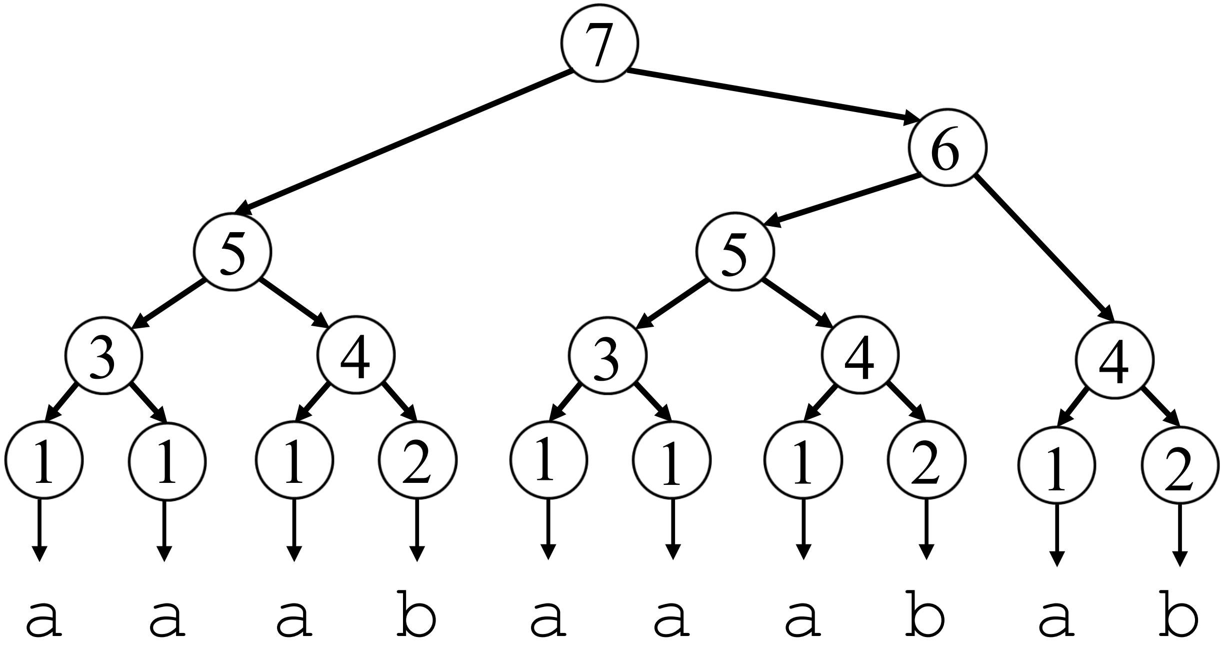
string represented by SLP S

DAG view of SLP

DAG for SLP S



Derivation tree T of SLP S

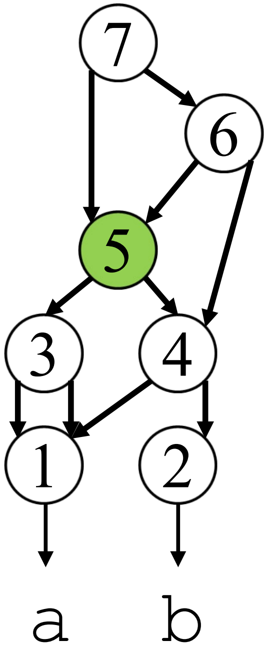


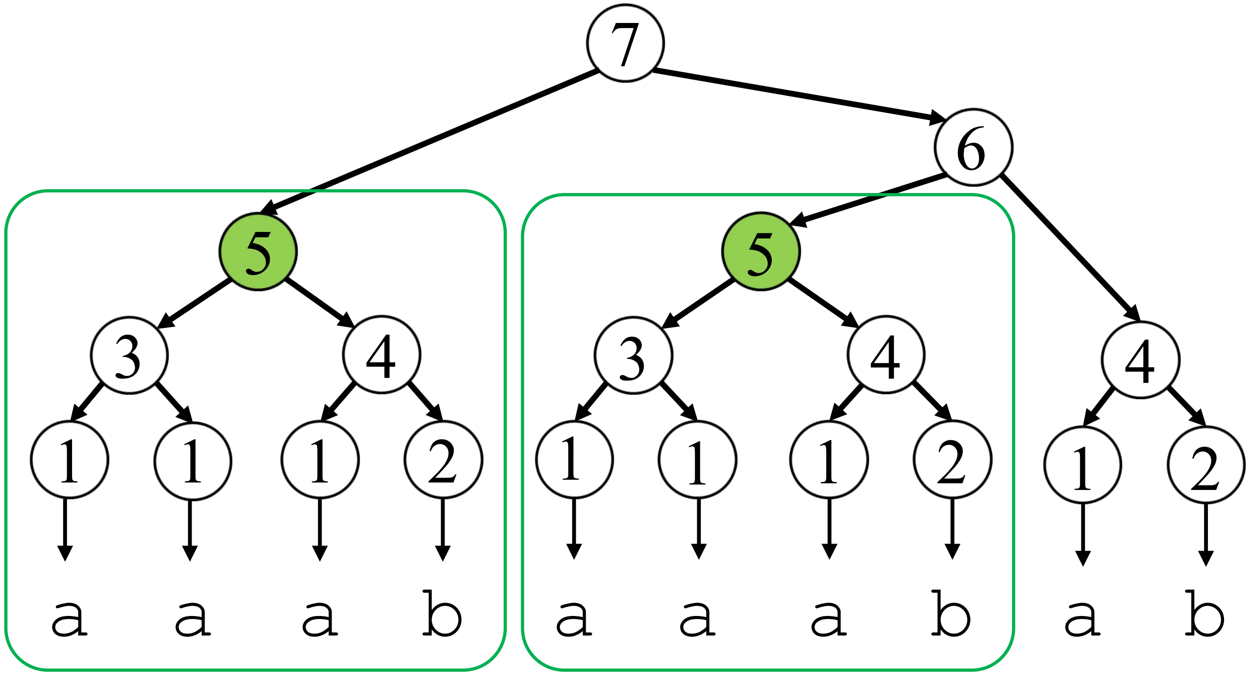
- ✓ This DAG is equivalent to the set of productions.

DAG view of SLP

DAG for SLP S

Derivation tree T of SLP S

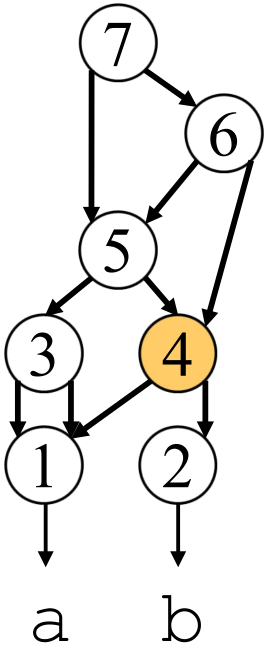




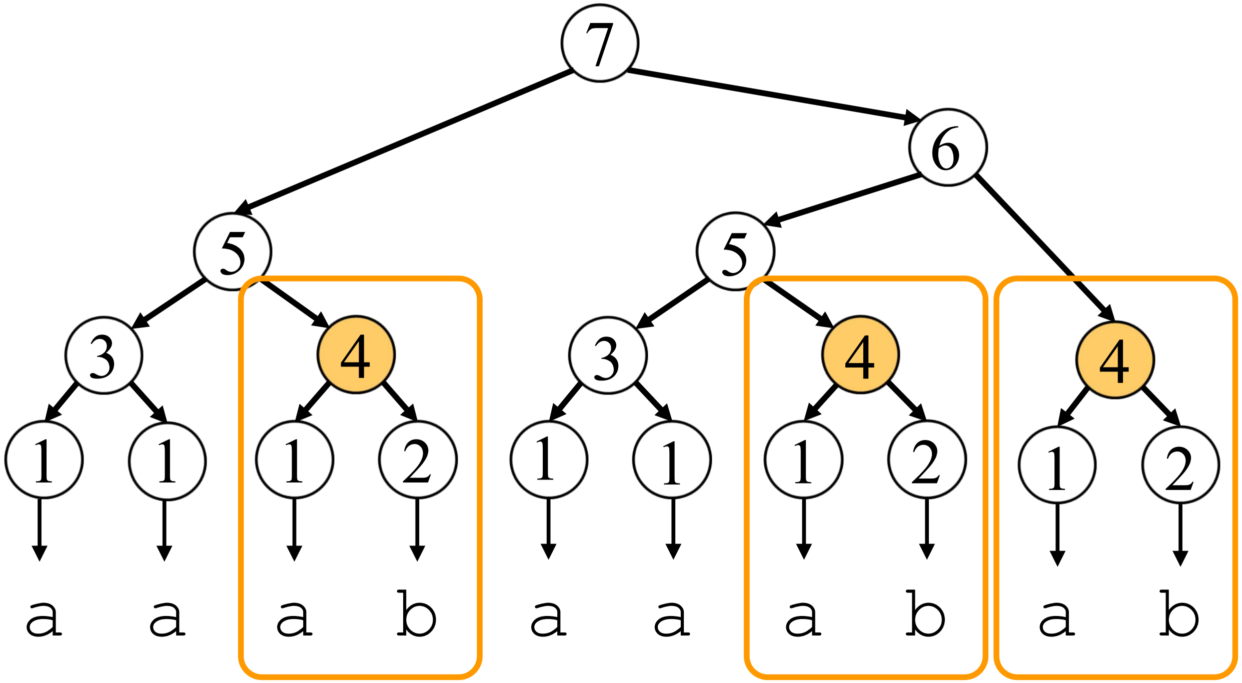
- ✓ Grammar-based compression captures repetitiveness in the string.

DAG view of SLP

DAG for SLP S



Derivation tree T of SLP S



- ✓ Grammar-based compression captures repetitiveness in the string.

Grammar-based Compressors

Computing the *smallest grammar* is NP-hard [Storer 1978].

$O(\log(N/g))$ approximation (g is the smallest grammar size)

- AVL grammar [Rytter 2003]
- α-balanced grammar [Charikar et al. 2005]
- Recompression [Jez 2015]

Greedy Algorithms

- LZ78 [Ziv & Lempel 1978]
- Re-Pair [Larsson & Moffat 2000]
- Longest Match [Nakamura et al. 2009]

Locally Consistent Parsing

- ESP-grammar [Sakamoto et al. 2009]
- Recompression [Jez 2015]

Note: This list is far from being comprehensive.

Grammar-based Compressors

Computing the *smallest grammar* is NP-hard [Storer 1978].

$O(\log(N/g))$ approximation (g is the smallest grammar size)
- AVL grammar [Rytter 2003]
- α-balanced grammar [Charikar et al. 2005]
- Recompression [Jez 2015]

Greedy Algorithms
- LZ78 [Ziv & Lempel 1978]
- **Re-Pair [Larsson & Moffat 2000]**
- Longest Match [Nakamura et al. 2009]

Best compression ratio in practice

Locally Consistent Parsing
- ESP-grammar [Sakamoto et al. 2009]
- Recompression [Jez 2015]

Note: This list is far from being comprehensive.

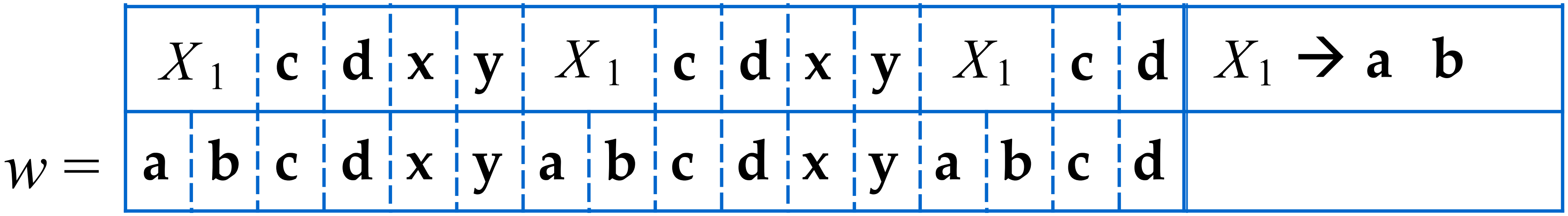
Re-Pair [Larsson & Moffat 2000]

$w =$

a	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]



X_1 c d x y X_1 c d x y X_1 c d $\quad X_1 \rightarrow \mathbf{a}\ \mathbf{b}$

$w =$ a b c d x y a b c d x y a b c d

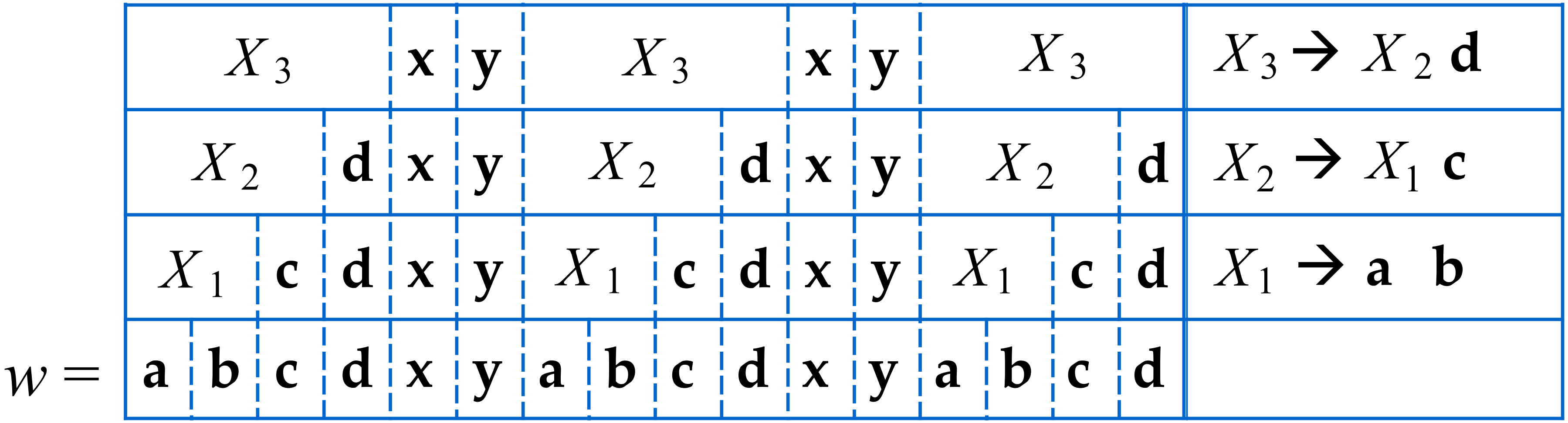
Recursively replaces the <u>most frequently occurring bigram</u> with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]

	X_2			**d**	**x**	**y**	X_2			**d**	**x**	**y**	X_2			**d**	$X_2 \rightarrow X_1$ **c**
	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	$X_1 \rightarrow$ **a** **b**
$w =$	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	

Recursively replaces the <u>most frequently occurring bigram</u> with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]



	X_3				**x**	**y**	X_3				**x**	**y**	X_3				$X_3 \rightarrow X_2$ **d**
	X_2			**d**	**x**	**y**	X_2			**d**	**x**	**y**	X_2			**d**	$X_2 \rightarrow X_1$ **c**
	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	$X_1 \rightarrow$ **a b**
$w =$	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]

<table>
<tr><td></td><td colspan="5">X_4</td><td>y</td><td colspan="5">X_4</td><td>y</td><td colspan="4">X_3</td><td>$X_4 \rightarrow X_3$ x</td></tr>
<tr><td></td><td colspan="4">X_3</td><td>x</td><td>y</td><td colspan="4">X_3</td><td>x</td><td>y</td><td colspan="4">X_3</td><td>$X_3 \rightarrow X_2$ d</td></tr>
<tr><td></td><td colspan="3">X_2</td><td>d</td><td>x</td><td>y</td><td colspan="3">X_2</td><td>d</td><td>x</td><td>y</td><td colspan="3">X_2</td><td>d</td><td>$X_2 \rightarrow X_1$ c</td></tr>
<tr><td></td><td colspan="2">X_1</td><td>c</td><td>d</td><td>x</td><td>y</td><td colspan="2">X_1</td><td>c</td><td>d</td><td>x</td><td>y</td><td colspan="2">X_1</td><td>c</td><td>d</td><td>$X_1 \rightarrow$ a b</td></tr>
<tr><td>$w =$</td><td>a</td><td>b</td><td>c</td><td>d</td><td>x</td><td>y</td><td>a</td><td>b</td><td>c</td><td>d</td><td>x</td><td>y</td><td>a</td><td>b</td><td>c</td><td>d</td><td></td></tr>
</table>

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]

	X_5						X_5						X_3				$X_5 \rightarrow X_4$ **y**
	X_4					**y**	X_4					**y**	X_3				$X_4 \rightarrow X_3$ **x**
	X_3				**x**	**y**	X_3				**x**	**y**	X_3				$X_3 \rightarrow X_2$ **d**
	X_2			**d**	**x**	**y**	X_2			**d**	**x**	**y**	X_2			**d**	$X_2 \rightarrow X_1$ **c**
	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	**x**	**y**	X_1		**c**	**d**	$X_1 \rightarrow$ **a b**
$w =$	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	**x**	**y**	**a**	**b**	**c**	**d**	

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

Re-Pair [Larsson & Moffat 2000]

<table>
<tr><td></td><td colspan="16">S</td><td>$S \rightarrow X_5 X_5 X_3$</td></tr>
<tr><td></td><td colspan="6">X_5</td><td colspan="6">X_5</td><td colspan="4">X_3</td><td>$X_5 \rightarrow X_4$ **y**</td></tr>
<tr><td></td><td colspan="5">X_4</td><td>**y**</td><td colspan="5">X_4</td><td>**y**</td><td colspan="4">X_3</td><td>$X_4 \rightarrow X_3$ **x**</td></tr>
<tr><td></td><td colspan="4">X_3</td><td>**x**</td><td>**y**</td><td colspan="4">X_3</td><td>**x**</td><td>**y**</td><td colspan="4">X_3</td><td>$X_3 \rightarrow X_2$ **d**</td></tr>
<tr><td></td><td colspan="3">X_2</td><td>**d**</td><td>**x**</td><td>**y**</td><td colspan="3">X_2</td><td>**d**</td><td>**x**</td><td>**y**</td><td colspan="3">X_2</td><td>**d**</td><td>$X_2 \rightarrow X_1$ **c**</td></tr>
<tr><td></td><td colspan="2">X_1</td><td>**c**</td><td>**d**</td><td>**x**</td><td>**y**</td><td colspan="2">X_1</td><td>**c**</td><td>**d**</td><td>**x**</td><td>**y**</td><td colspan="2">X_1</td><td>**c**</td><td>**d**</td><td>$X_1 \rightarrow$ **a b**</td></tr>
<tr><td>$w =$</td><td>**a**</td><td>**b**</td><td>**c**</td><td>**d**</td><td>**x**</td><td>**y**</td><td>**a**</td><td>**b**</td><td>**c**</td><td>**d**</td><td>**x**</td><td>**y**</td><td>**a**</td><td>**b**</td><td>**c**</td><td>**d**</td><td></td></tr>
</table>

Recursively replaces the most frequently occurring bigram with a new non-terminal (ties are broken arbitrarily).

Re-Pair vs Empirical Entropy H_k

Re-Pair outperforms H_k on real-world repetitive strings.

File	Re-Pair	H_0	H_4	H_8
DNA sequences (influenza)	**3.31%**	24.63%	23.88%	13.25%
Source Codes (kernel)	**1.13%**	67.25%	19.25%	7.75%
Wikipedia (Einstein)	**0.10%**	62.00%	13.25%	3.50%
Documents (CIA world leaders)	**1.78%**	43.38%	7.63%	3.13%

This is a part of results reported in the statistics on Pizza and Chile highly-repetitive corpus.
http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf

Approximation to Smallest Grammar

Approximation ratios of grammar-based compressors to the smallest grammar of size g

Compressors	Upper bound	Lower bound
Re-Pair	$O((N / \log N)^{2/3})$ [1]	$\Omega(\log N / \log\log N)$ [2]
LongestMatch	$O((N / \log N)^{2/3})$ [1]	$\Omega(\log\log N)$ [1]
Greedy	$O((N / \log N)^{2/3})$ [1]	$> 1.37\ldots$ [1]
LZ78	$O((N / \log N)^{2/3})$ [1]	$\Omega((N / \log N)^{2/3})$ [2]
AVL-grammar	$O(\log(N / g))$ [3]	-
α-balanced grammar	$O(\log(N / g))$ [2]	-
Recompression	$O(\log(N / g))$ [4]	-
ESP-grammar	$O(\log^2 N \log^* N)$ [5][6]	-

[1] Charikar et al. 2005, [2] Bannai et al. 2020, [3] Rytter 2003, [4] Jez 2015, [5] Sakamoto et al. 2009, [6] I & Takabatake (personal communication)

Approximation to Smallest Grammar

Approximation ratios of grammar-based compressors
to the smallest grammar of size g

Compressors	Upper bound	Lower bound
Re-Pair	$O((N / \log N)^{2/3})$ [1]	$\Omega(\log N / \log\log N)$ [2]
LongestMatch	$O((N / \log N)^{2/3})$ [1]	$\Omega(\log\log N)$ [1]
Greedy	$O((N / \log N)^{2/3})$ [1]	$> 1.37...$ [1]
LZ78	$O((N / \log N)^{2/3})$ [1]	$\Omega((N / \log N)^{2/3})$ [2]
AVL-grammar	$O(\log(N / g))$ [3]	-
α-balanced grammar	$O(\log(N / g))$ [2]	
Recompression	$O(\log(N / g))$ [4]	
ESP-grammar	$O(\log^2 N \log^* N)$ [5][6]	

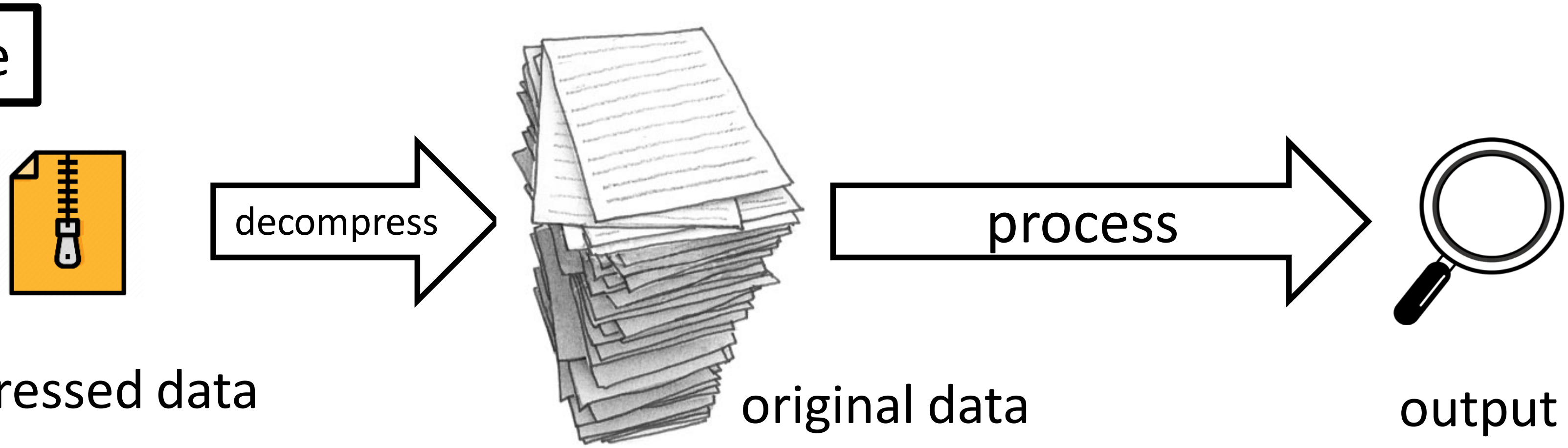
We still do not know which one is the best in theory.

[1] Charikar et al. 2005, [2] Bannai et al. 2020, [3] Rytter 2003, [4] Jez 2015, [5] Sakamoto et al. 2009, [6] I & Takabatake (personal communication)

Compressed String Processing (CSP)

Techniques that perform various operations on compressed data without decompression.

Naive



compressed data

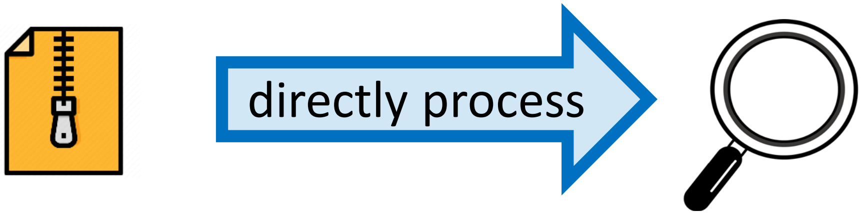
decompress

original data

process

output

CSP



compressed data

directly process

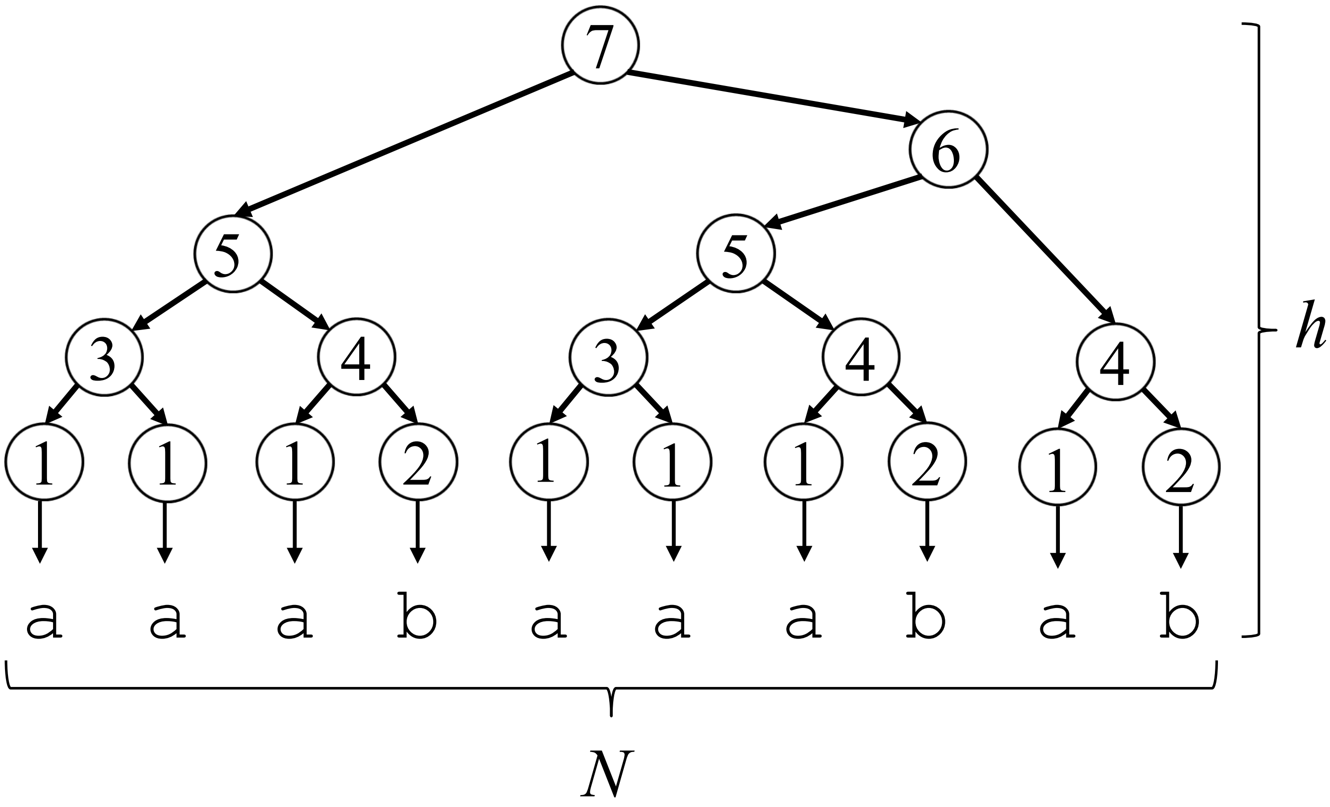
output

Merits of CSP

- ➢ Memory saving
- ➢ Faster computation

CSP on SLP

$$n\left\{\begin{array}{l} X_7 \rightarrow X_5\, X_6 \\ X_6 \rightarrow X_5\, X_4 \\ X_5 \rightarrow X_3\, X_4 \\ X_4 \rightarrow X_1\, X_2 \\ X_3 \rightarrow X_1\, X_1 \\ X_2 \rightarrow \mathtt{b} \\ X_1 \rightarrow \mathtt{a} \end{array}\right.$$

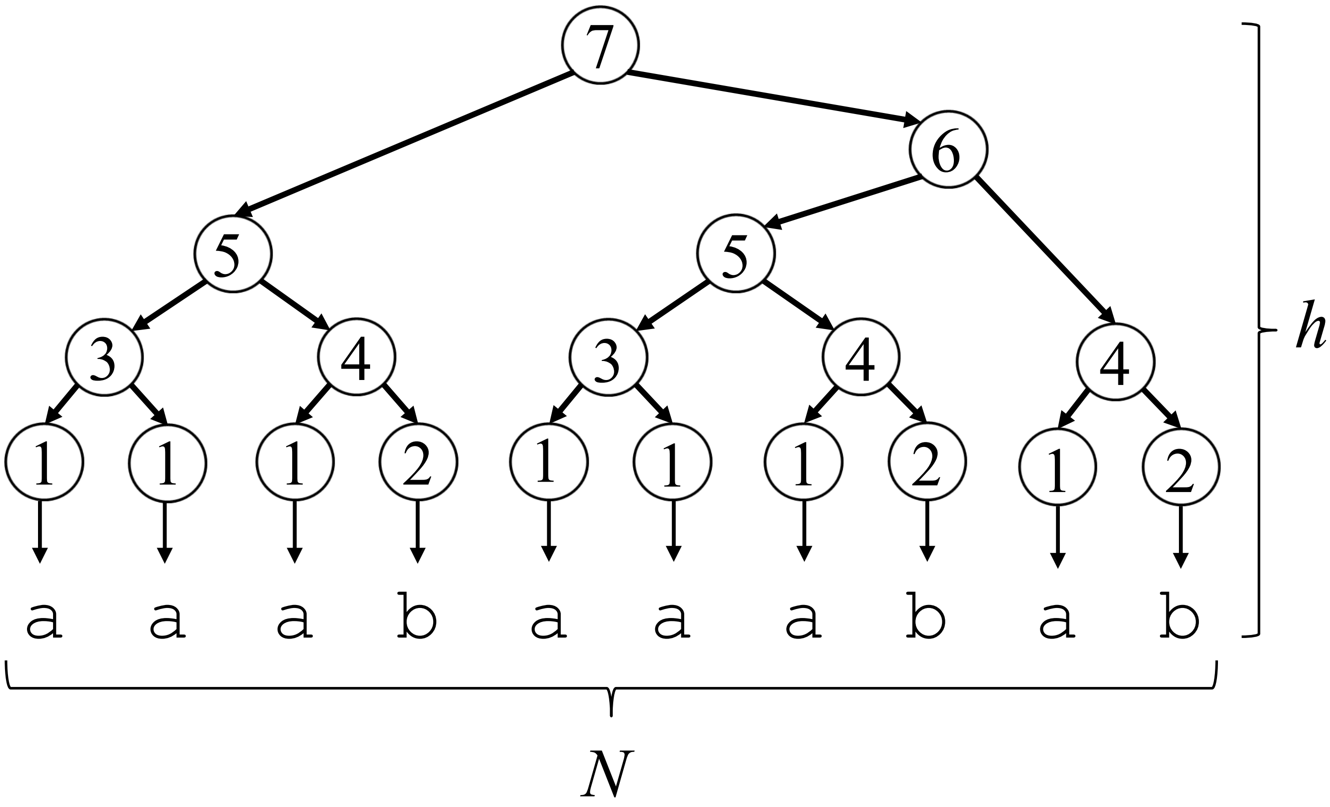


The values of n and h vary depending on the grammar compressor that was used for producing SLP.

➔ We wish to design CSP algorithms that can work on any SLP, independently of the compressor.

CSP on SLP

$$n\left\{\begin{array}{l} X_7 \to X_5\,X_6 \\ X_6 \to X_5\,X_4 \\ X_5 \to X_3\,X_4 \\ X_4 \to X_1\,X_2 \\ X_3 \to X_1\,X_1 \\ X_2 \to \mathtt{b} \\ X_1 \to \mathtt{a} \end{array}\right.$$



For any SLP, $\log_2 N \leq h \leq n$ always holds. ➔ $N \in O(2^n)$.

Therefore, CSP algorithms that run in time & space $O(\mathrm{poly}(n)\mathrm{polylog}(N))$ are interesting and can be useful.

(Incomplete) List of Known Results on CSP for SLPs

Fully Compressed Pattern Matching	[Karpinski et al. 1997]; [Miyazaki et al. 1997] [Lifshits 2007]; [Jez 2015]
Text Indexing	[Claude & Navarro 2012]; [Maruyama et al. 2013]; [Tsuruta et al. 2020];
Subsequence / VLDC Pattern Matching	[Cegielski 2000]; [Tiskin 2009]; [Yamamoto et al. 2011]
Random Access / Substring Extraction	[Belazzougui et al. 2013]; [Bille et al. 2015]
Longest Common Extension	[Karpinski et al. 1997]; [Miyazaki et al. 1997]; [Bille et al. 2016]; [Nishimoto et al. 2016]; [I 2017]
Longest Common Subsequence / Edit Distance	[Tiskin 2007, 2008]; [Hermelin et al. 2009, 2011]; [Gawrychowski 2012]
Longest Common Substring	[Matsubara et al. 2009]
String Regularities (palindromes, repetitions etc.)	[Matsubara et al. 2009]; [Inenaga & Bannai 2012]; [I et al. 2015]
q-gram frequencies	[Goto et al. 2012]; [Goto et al. 2013]; [Bille et al. 2014]

String Primitives

Space complexities are evaluated by the number of words (not bits) unless otherwise stated.

algorithm	query time	preprocess. time	space
random access [Bille et al. 2015]	$O(\log N)$	$O(n)$	$O(n)$
substring extraction [Bille et al. 2015]	$O(m + \log N)$	$O(n)$	$O(n)$
LCE queries [I 2017]	$O(\log N)$	$O(n + z \log\frac{N}{z})$	$O(n+z \log\frac{N}{z})$

- m is the length of the substring to extract.
- z is # of phrases in the LZ77 factorization.

Text Mining / String Comparison

algorithm	time	space
q-gram frequencies [Goto et al. 2013]	$O(qn)$	$O(qn)$
most frequent substring [Goto et al. 2013]	$O(n)$	$O(n)$
longest repeating substring [Inenaga & Bannai 2012]	$O(n^4 \log n)$	$O(n^3)$
longest common substring [Matsubara et al. 2009]	$O(n^4 \log n)$	$O(n^3)$

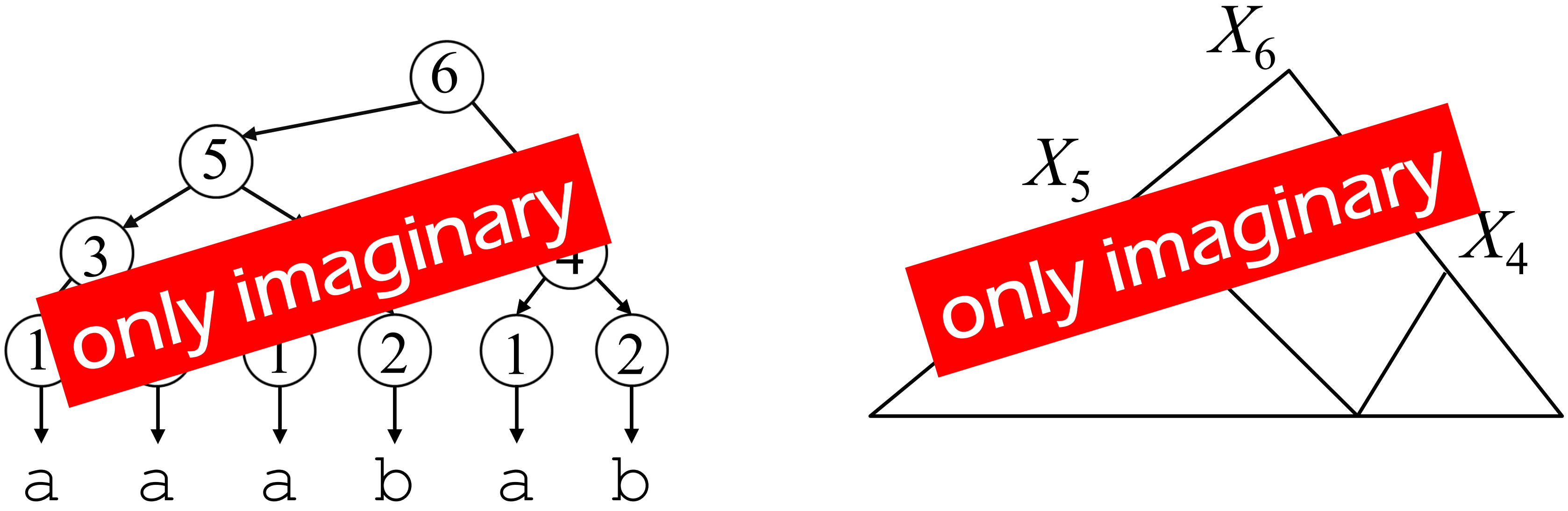
- q-gram is a string of length q.

String Regularities

algorithm	time	space
repetitions (runs) [I et al. 2015]	$O(n^3h)$	$O(n^2)$
palindromes [Matsubara et al. 2009]	$O(nh(n+h \log N))$	$O(n^2)$
gapped palindromes [I et al. 2015]	$O(nh(n^2 + g \log N))$	$O(n(n+g))$
periods [I et al. 2015]	$O(n^2h)$	$O(n^2)$
covers [I et al. 2015]	$O(nh\,(n + \log^2 N))$	$O(n^2)$

- g is the fixed gap length (usually a constant).

Important Remark



✓ Derivation trees are used only for illustrative purposes, and are not explicitly constructed in CPS algorithms.

✓ CSP on SLPs can be seen as algorithmic technique that performs various kinds of operations on the DAG for SLP, not on the derivation tree.

q-gram Frequency on SLP

Problem (q-gram frequencies on SLP)

Given an SLP S which represents a string w and a positive integer q, compute the number of occurrences of all substrings of length q in w.

Uncompressed q-gram Frequencies

Compute # occurrences of
each length-q substring in string w.

Eg) 3-gram frequencies ($q = 3$)

input $w = \underline{a\,b\,a}\,b\,b\,b\,b\,b\,\underline{a\,b\,a}\,b$

output

aba	2
abb	1
bab	3
bba	1
bbb	3

Lots of applications
for q-gram frequencies:
NLP, Bioinformatics,
Text Mining, etc.

Solution for Uncompressed String

✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$

SA	LCP	
8	–	$
7	0	a$
5	1	aba$
3	3	ababa$
1	5	abababa$
6	0	ba$
4	2	baba$
2	4	bababa$

Solution for Uncompressed String

✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$

SA	LCP	
8	–	$
7	0	a$
5	1	aba$
3	3	ababa$
1	5	abababa$
6	0	ba$
4	2	baba$
2	4	bababa$

Solution for Uncompressed String

- ✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$

SA	LCP		
8	–		`$`
7	0		`a$`
5	1	< 3	`aba$`
3	3	≥ 3	`ababa$`
1	5		`abababa$`
6	0		`ba$`
4	2		`baba$`
2	4		`bababa$`

Solution for Uncompressed String

✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$

SA	LCP		
8	–		$
7	0		a$
5	1	< 3	aba$
3	3	≥ 3	ababa$
1	5	≥ 3	abababa$
6	0		ba$
4	2		baba$
2	4		bababa$

Solution for Uncompressed String

- ✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

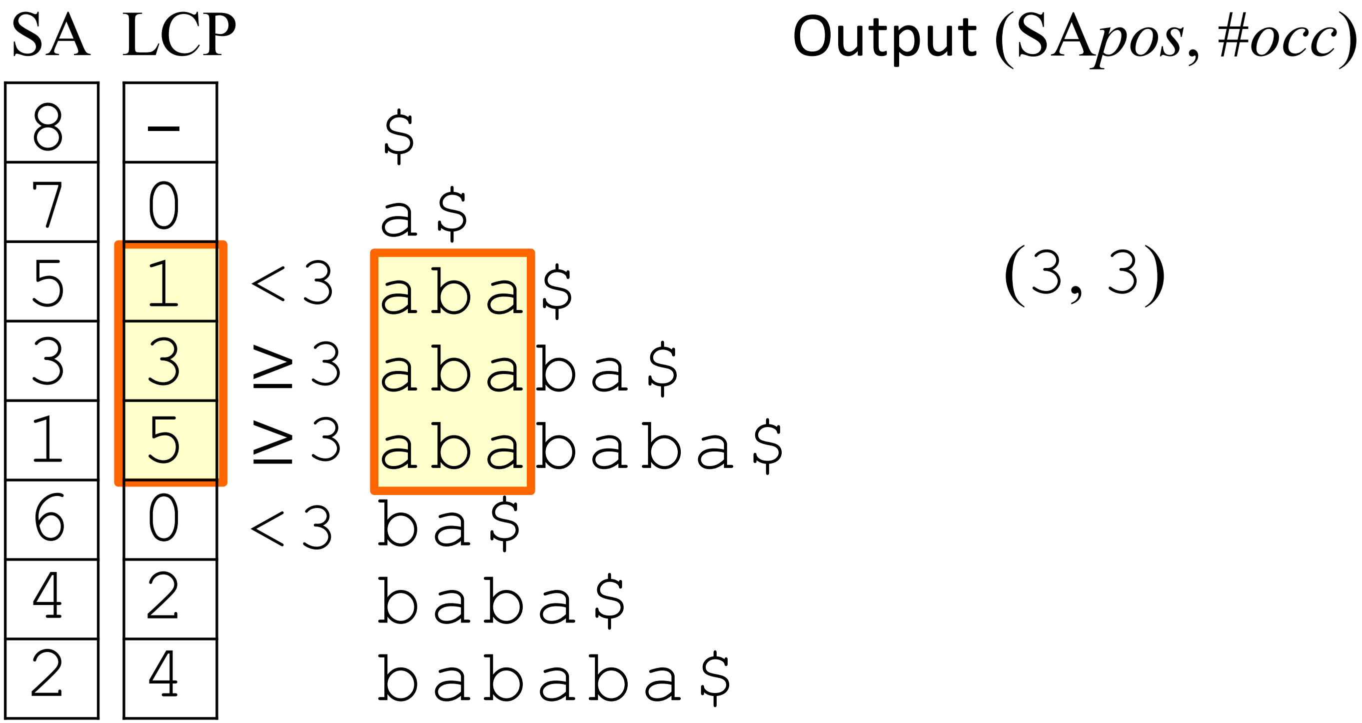
$q = 3$

SA	LCP			Output (SA*pos*, #*occ*)
8	–		$	
7	0		a$	
5	1	< 3	aba$	(3, 3)
3	3	≥ 3	ababa$	
1	5	≥ 3	abababa$	
6	0	< 3	ba$	
4	2		baba$	
2	4		bababa$	

Solution for Uncompressed String

- ✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$

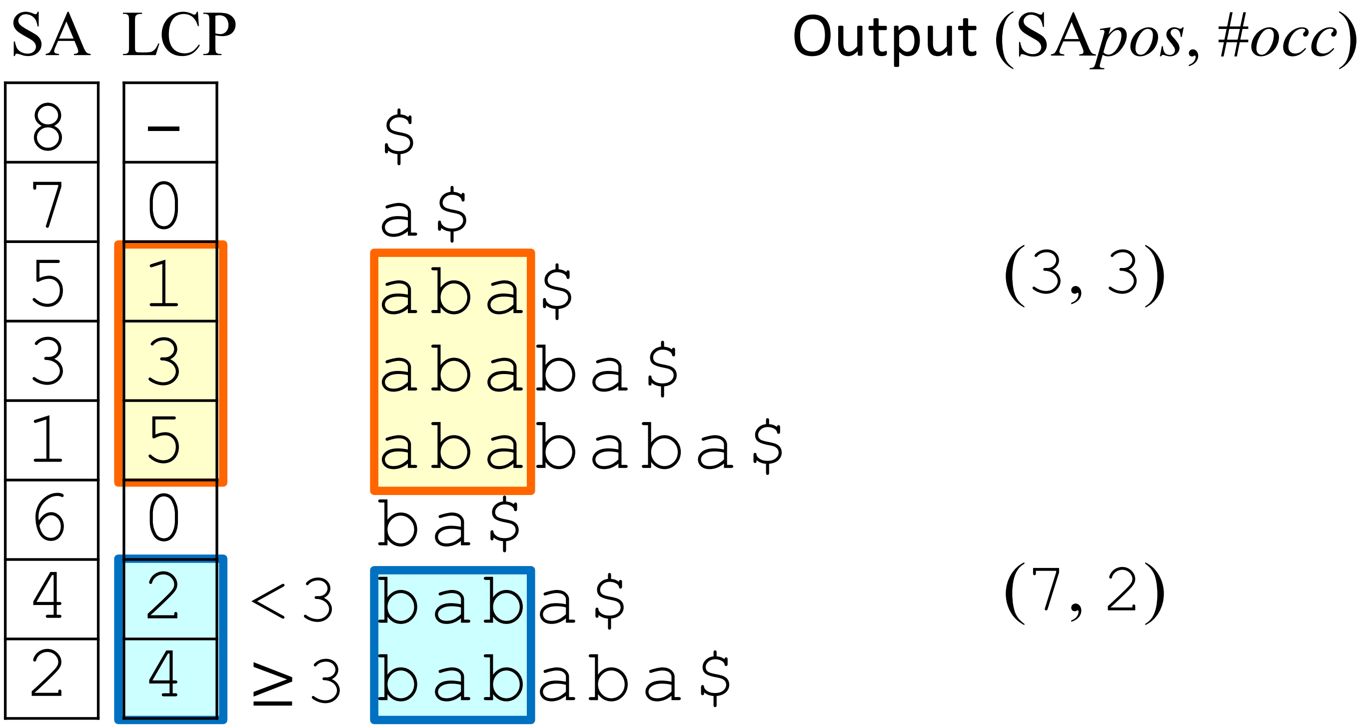


SA	LCP			Output (SA*pos*, #*occ*)
8	–		$	
7	0		a$	
5	1	< 3	aba$	(3, 3)
3	3	≥ 3	ababa$	
1	5	≥ 3	abababa$	
6	0	< 3	ba$	
4	2		baba$	
2	4		bababa$	

Solution for Uncompressed String

- ✓ Given an uncompressed string w , we can solve the q-gram frequencies problem in $O(N)$ time, by e.g. using the suffix array and LCP array of w.

$q = 3$



SA	LCP			Output (SA*pos*, #*occ*)
8	–		`$`	
7	0		`a$`	
5	1		`aba$`	(3, 3)
3	3		`ababa$`	
1	5		`abababa$`	
6	0		`ba$`	
4	2	<3	`baba$`	(7, 2)
2	4	≥ 3	`bababa$`	

Compressed q-gram Frequencies

Compute # occurrences of each length-q substring in grammar-compressed string w.

input

CFG
deriving only w

$S \rightarrow BCCB$
$C \rightarrow bb$
$B \rightarrow AA$
$A \rightarrow ab$

$q = 3$

Directly compute

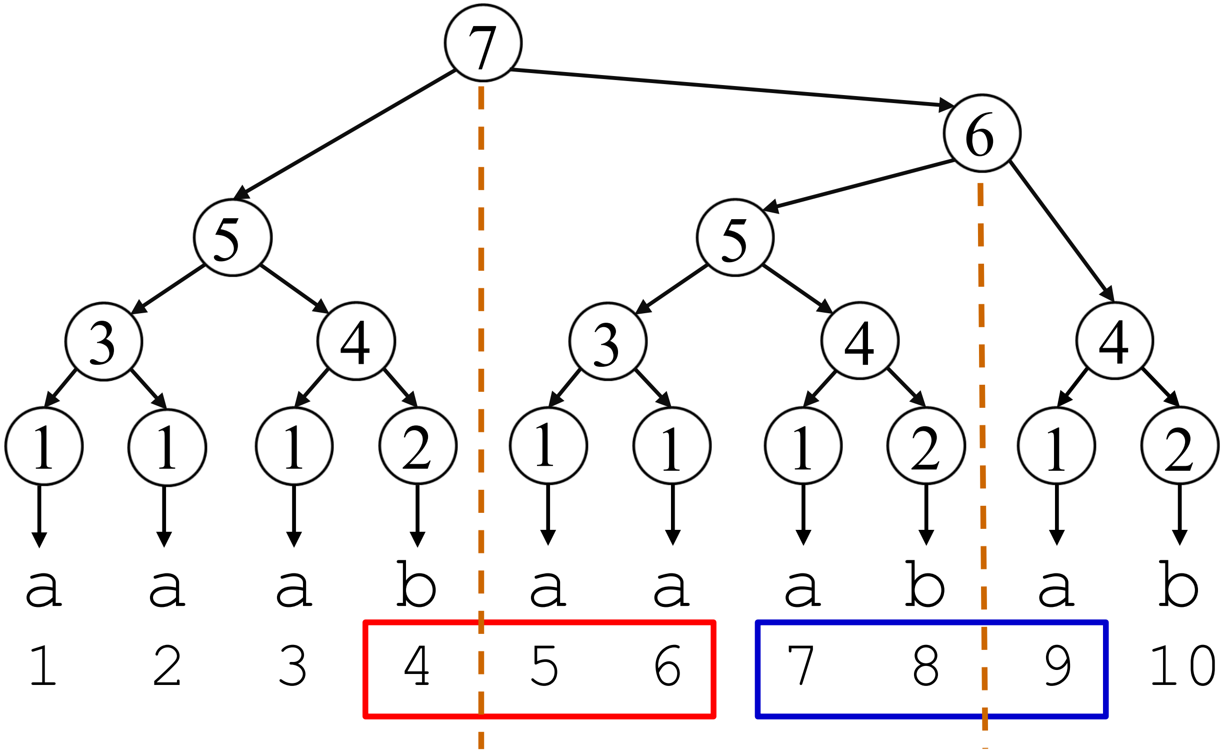
output

3-gram frequencies

aba	2
abb	1
bab	3
bba	1
bbb	3

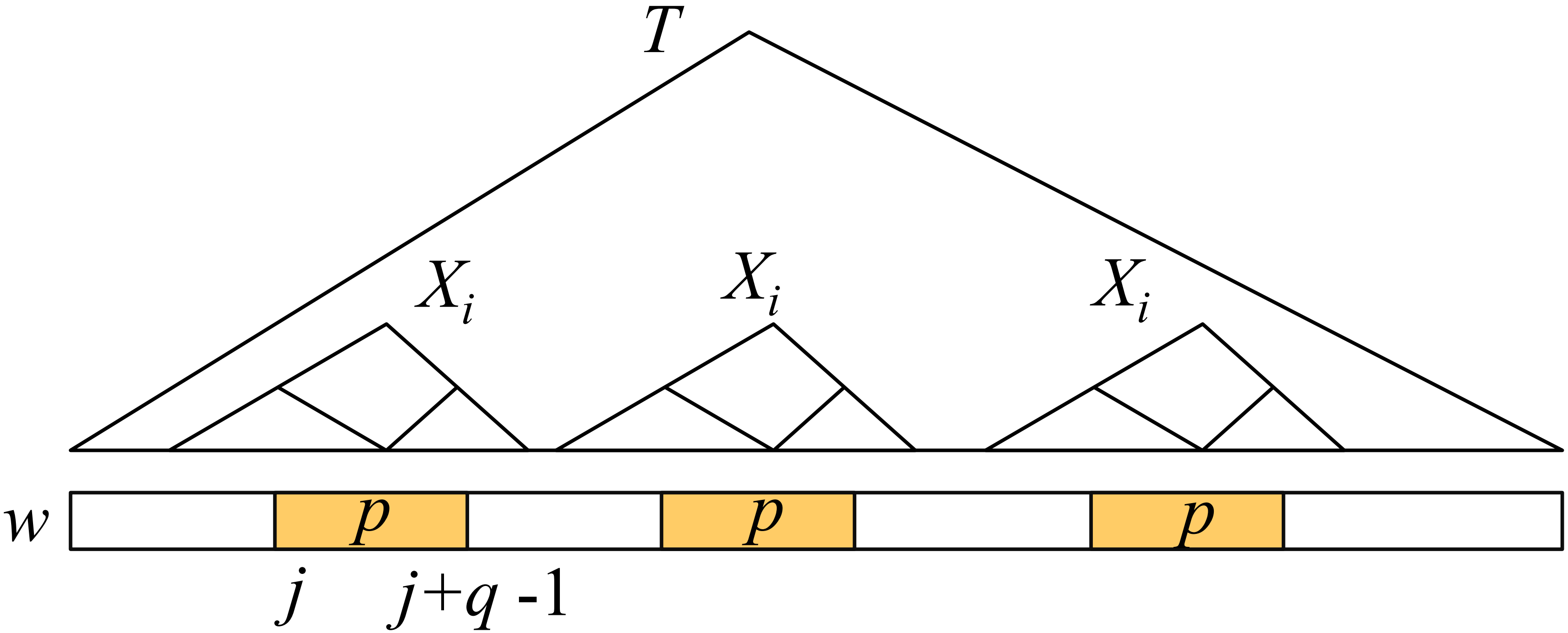
Stabbing

An integer interval $[b, e]$ $(1 \leq b \leq e \leq N)$ is said to be stabbed by a variable X_i, if the LCA of the bth and eth leaves of the derivation tree T is labeled by X_i.



Observation

- ✓ Assume that the occurrence of a q-gram p starting at position j is stabbed by an occurrence of X_i in T.
- ✓ Then clearly, in any other occurrence of X_i in T, there is another stabbed occurrence of p.



Sub-problems

- ✓ Hence, the q -gram frequencies problem on SLP reduces to the following sub-problems:

Sub-Problem 1

For each variable X_i , count the number of occurrences of X_i in the derivation tree T.

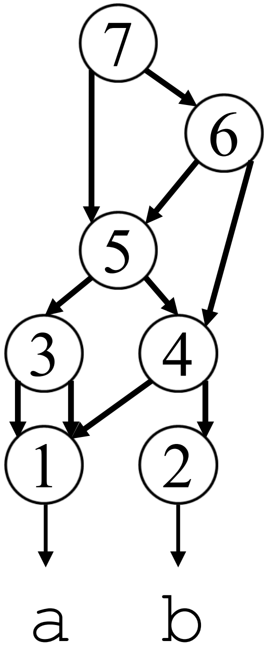
Sub-Problem 2

For each variable X_i , count the number of occurrences of every q-gram stabbed by X_i.

Solving Sub-Problem 1

Lemma

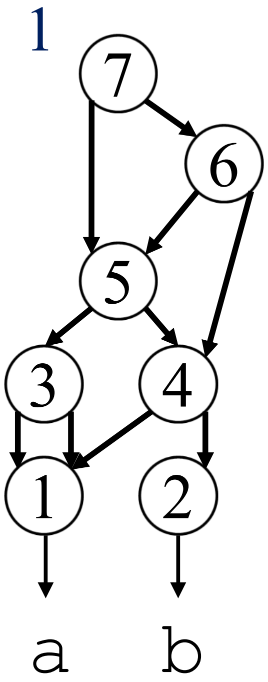
We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.



Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

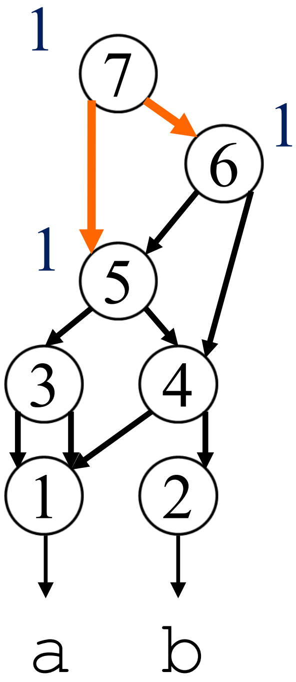


✓ The root occurs exactly once.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

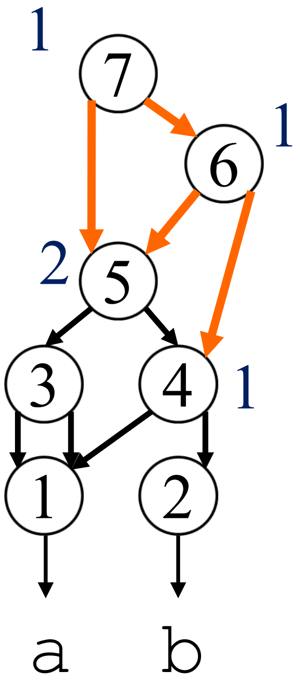


- ✓ For each node in a topological order, propagate its number of occurrences to its children.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

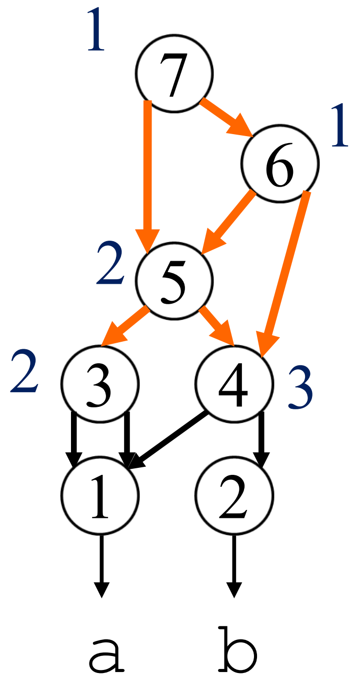


- ✓ For each node in a topological order, propagate its number of occurrences to its children.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

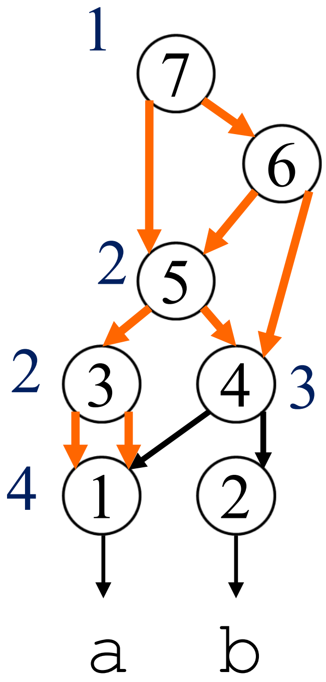


- ✓ For each node in a topological order, propagate its number of occurrences to its children.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

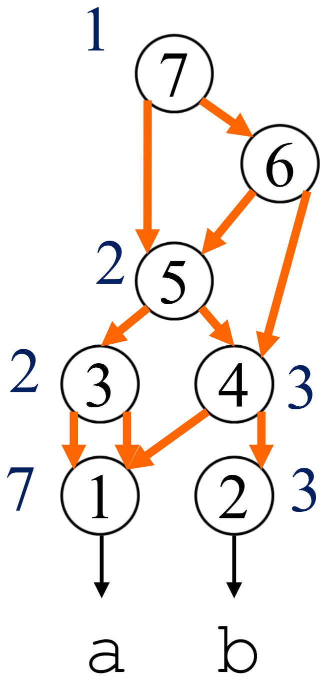


- ✓ For each node in a topological order, propagate its number of occurrences to its children.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

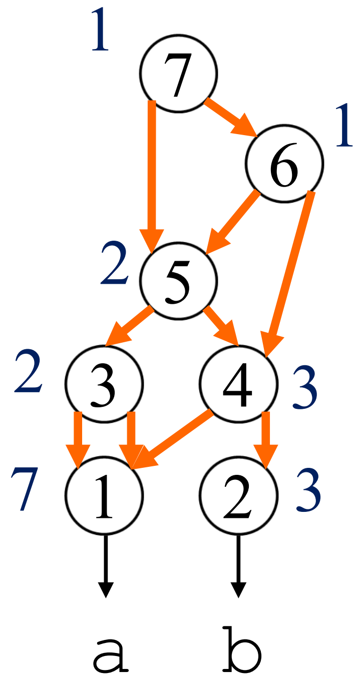


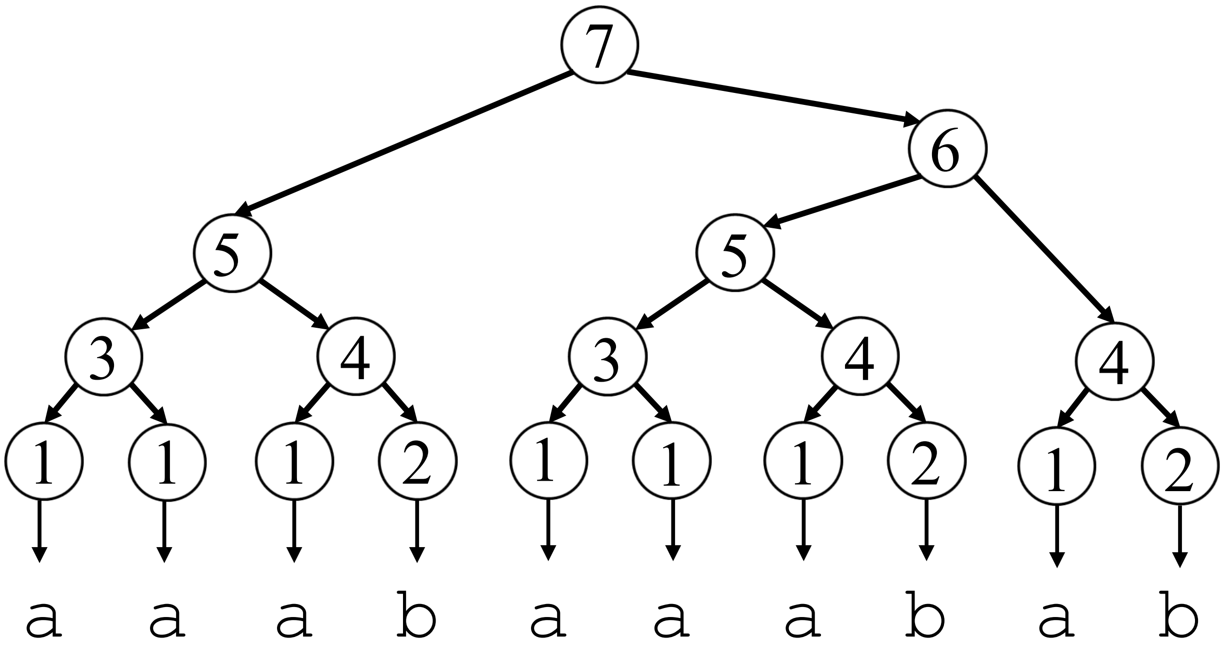
- ✓ For each node in a topological order, propagate its number of occurrences to its children.

Solving Sub-Problem 1

Lemma

We can count # of occurrences of every variable X_i in the derivation tree T in $O(n)$ time.

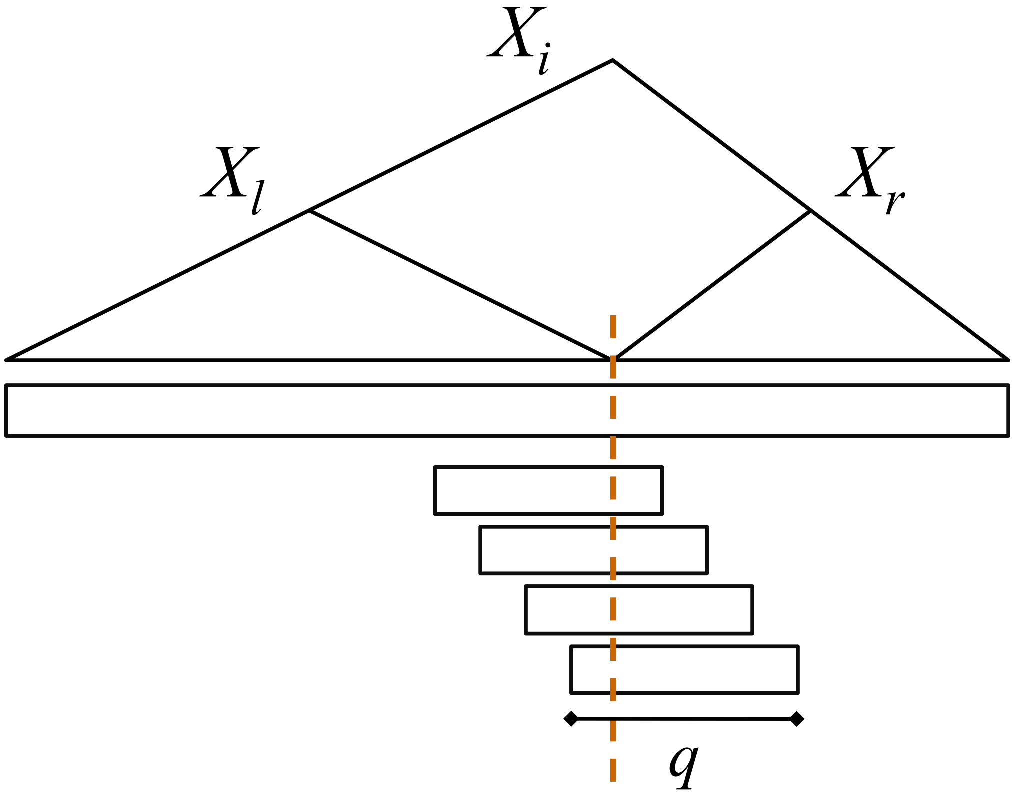




Solving Sub-Problem 2

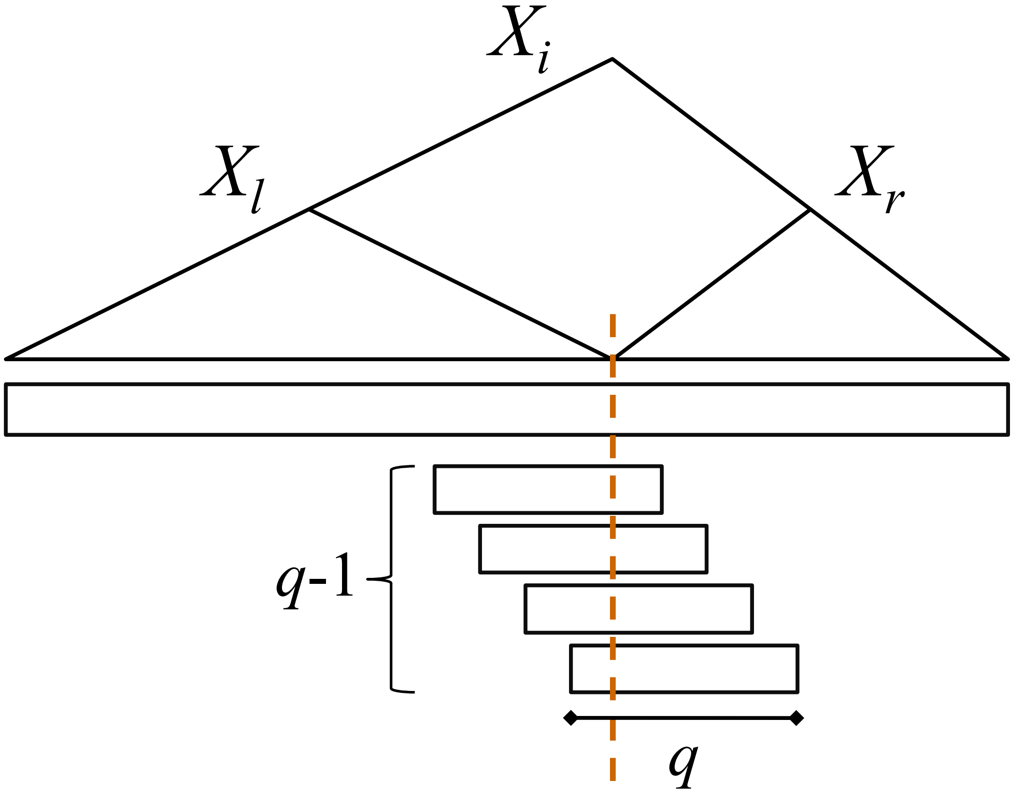
Sub-Problem 2

For each variable X_i, count the number of occurrences of every q-gram stabbed by X_i.



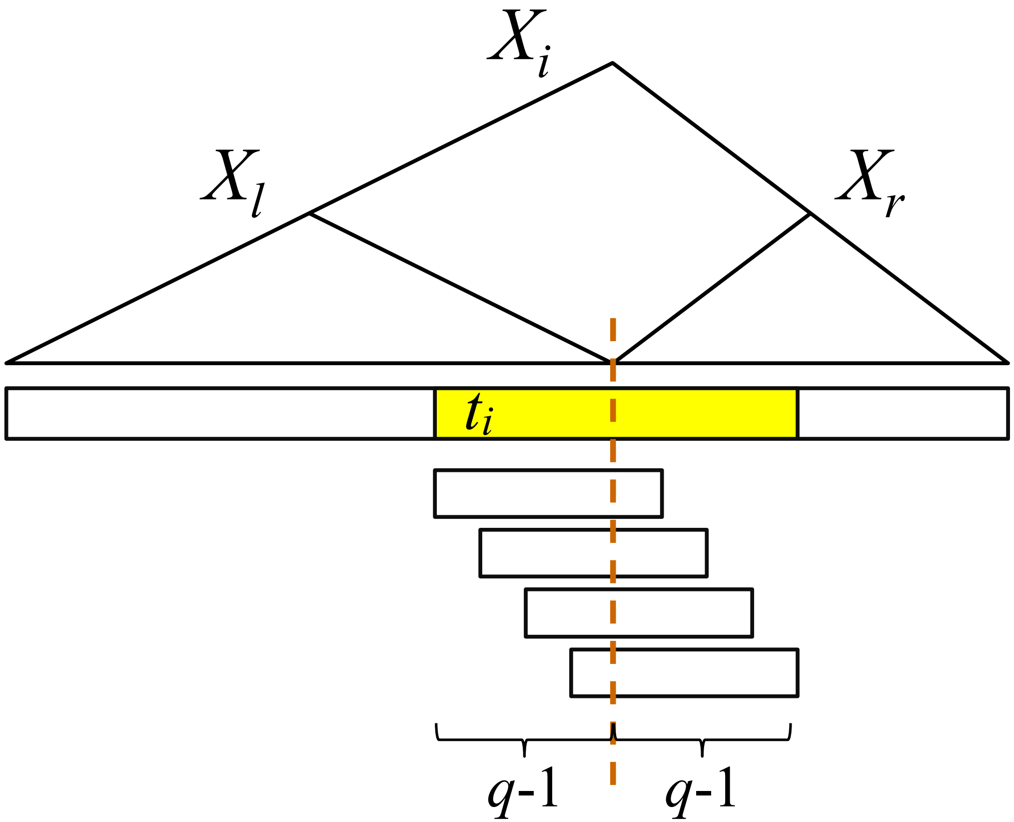
Solving Sub-Problem 2

Key Observation: Each variable X_i can stab at most q-1 occurrences of q-grams.



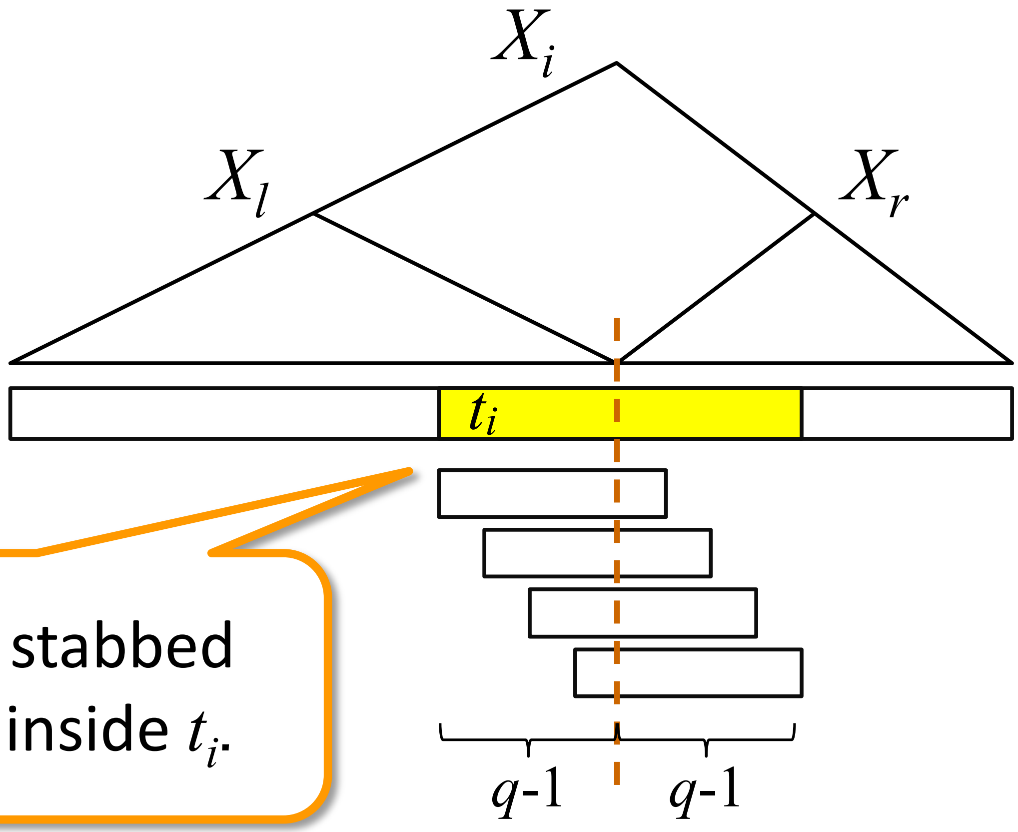
Solving Sub-Problem 2

- ✓ We "partially" decompress the substring $t_i = X_l[|X_l|-q+2..|X_l|]\ X_r[1..q-1]$ of length $2q-2$.



Solving Sub-Problem 2

- ✓ We "partially" decompress the substring $t_i = X_l[|X_l|-q+2..|X_l|]\ X_r[1..q-1]$ of length $2q-2$.



All q-grams stabbed by X_i occur inside t_i.

Solving Sub-Problem 2

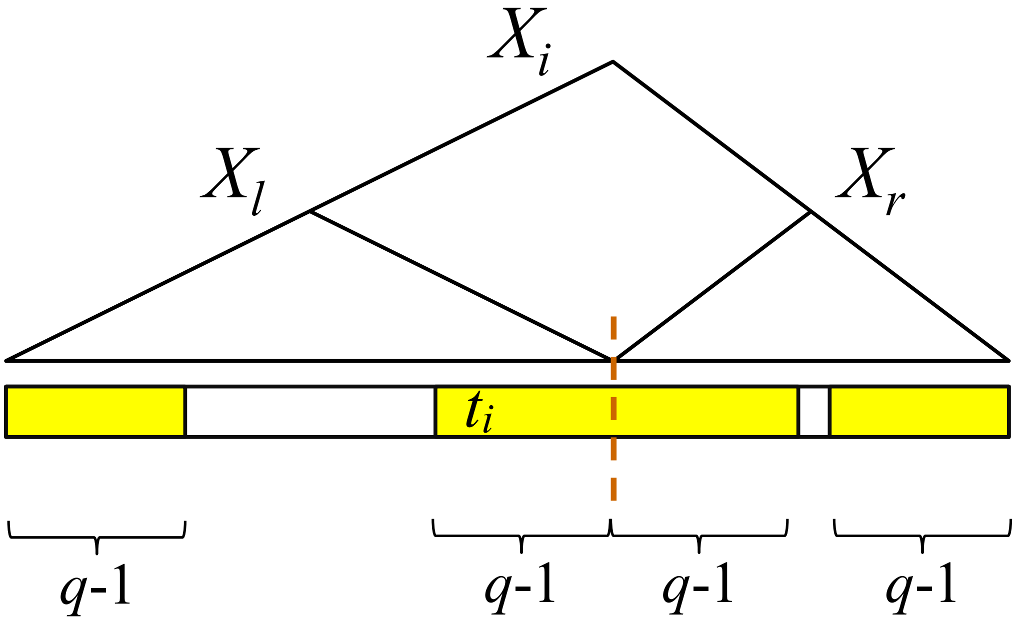
Lemma

For all variables X_i, we can count the number of occurrences of every q-gram stabbed by X_i in $O(qn)$ time and space.

- ✓ For every variable X_i, the substring t_i can be computed in a total of $O(qn)$ time, by a simple DP (to be explained in the next slide).

Solving Sub-Problem 2

- ✓ To compute t_i, it is enough to compute the prefix and suffix of length q-1 of each variable.



Solving Sub-Problem 2

Lemma

For all variables X_i, we can count the number of occurrences of every q-gram stabbed by X_i in $O(qn)$ time and space.

✓ Then, we construct the suffix array and LCP array for strings $t_1, \ldots, t_n$ in $O(|t_1 \ldots t_n|) \subseteq O(qn)$ time.

q-gram Frequency on SLP

Theorem [Goto et al. 2013]

The problem of computing q-gram frequencies on SLP can be solved in $O(qn)$ time and space.

- ☐ Usually q is a small constant (from 2 to 4)
  → In most practical situations,
  this algorithm works in $O(n)$ time & space.
- ☐ Decompression-then-compute method takes $O(N) \subseteq O(2^n)$ time in the worst case.

Experimental Results

Running time (sec.) on XML data (200MB)

q	(1) Naive $O(qN)$ time	(2) SA $O(N)$ time	(3) Goto et al. $O(qn)$ time
2	22.9	41.7	**6.5**
3	55.7	41.7	**11.0**
4	93.3	41.7	**16.3**
5	129.3	41.7	**21.3**
6	158.7	41.7	**25.8**
7	181.1	41.7	**30.1**
8	198.3	41.9	**34.2**

Goto et al.'s method is by far the fastest for important values of $q = 2..4$.

Note: CSP algorithm by Goto et al. is the fastest even if we subtract decompression times (3.6 sec.) from (1) and (2), for all values of q tested here.

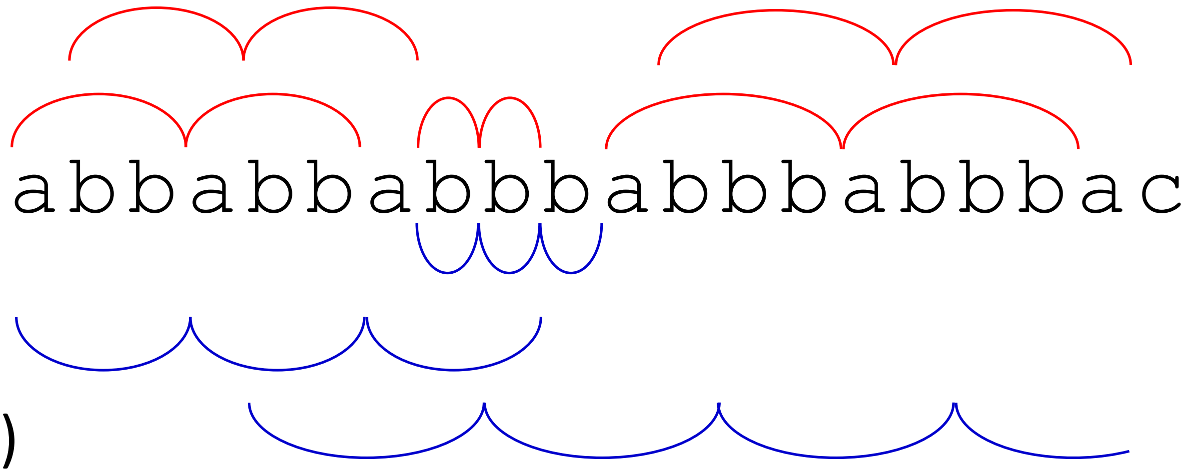
Finding Repetitions from SLP

Problem (finding repetitions from SLP)

Given an SLP S which represents a string w, compute all squares and runs that occur in w.

Note: There are more squares

squares
(of form xx)

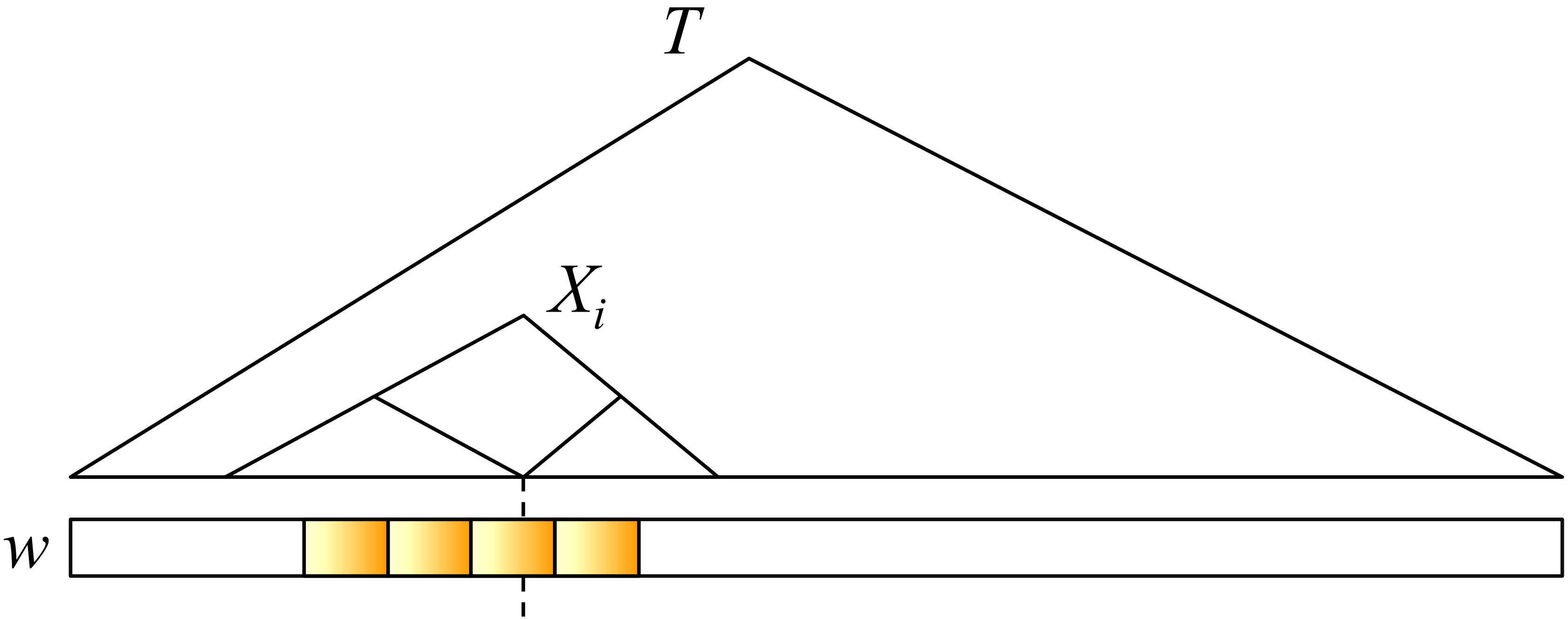


a b b a b b a b b b a b b b a b b b a c

runs
(maximal repetition $x^k x'$)

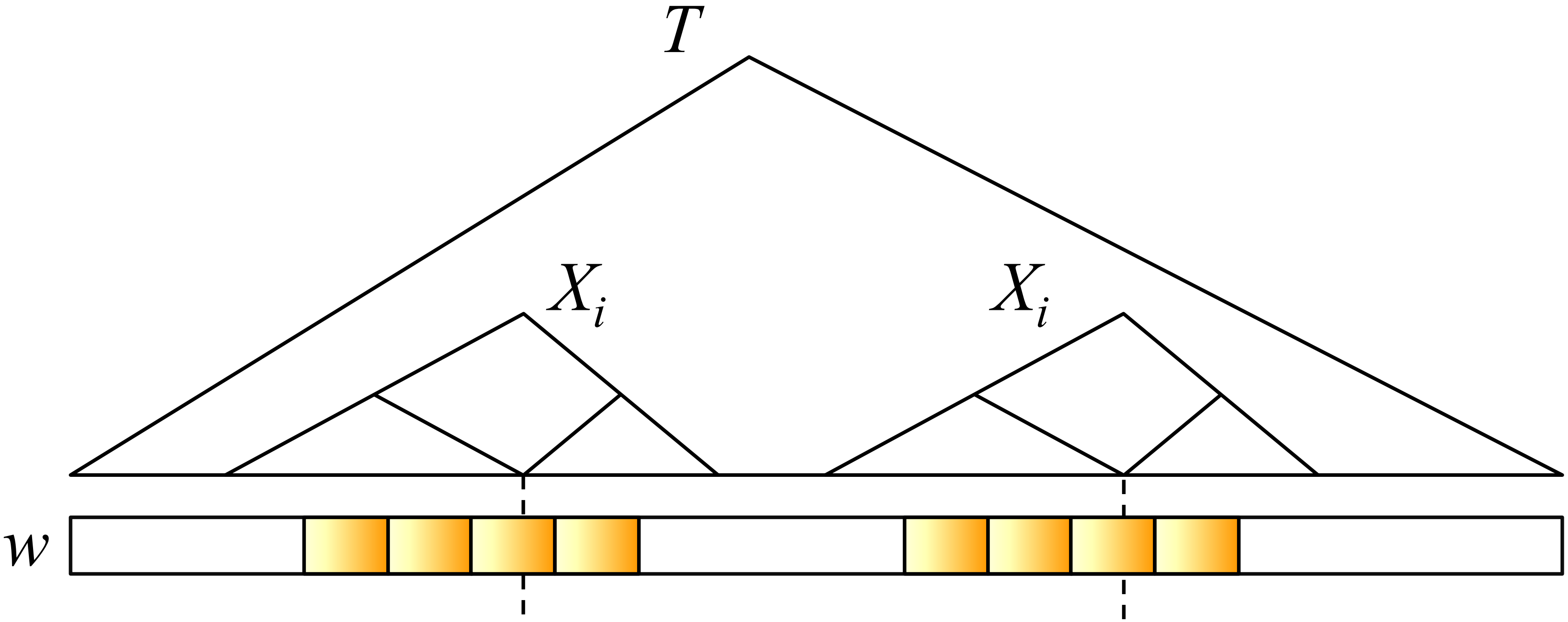
Stabbed Runs

- ✓ For each run in the string w, there is a unique variable X_i that stabs the run.



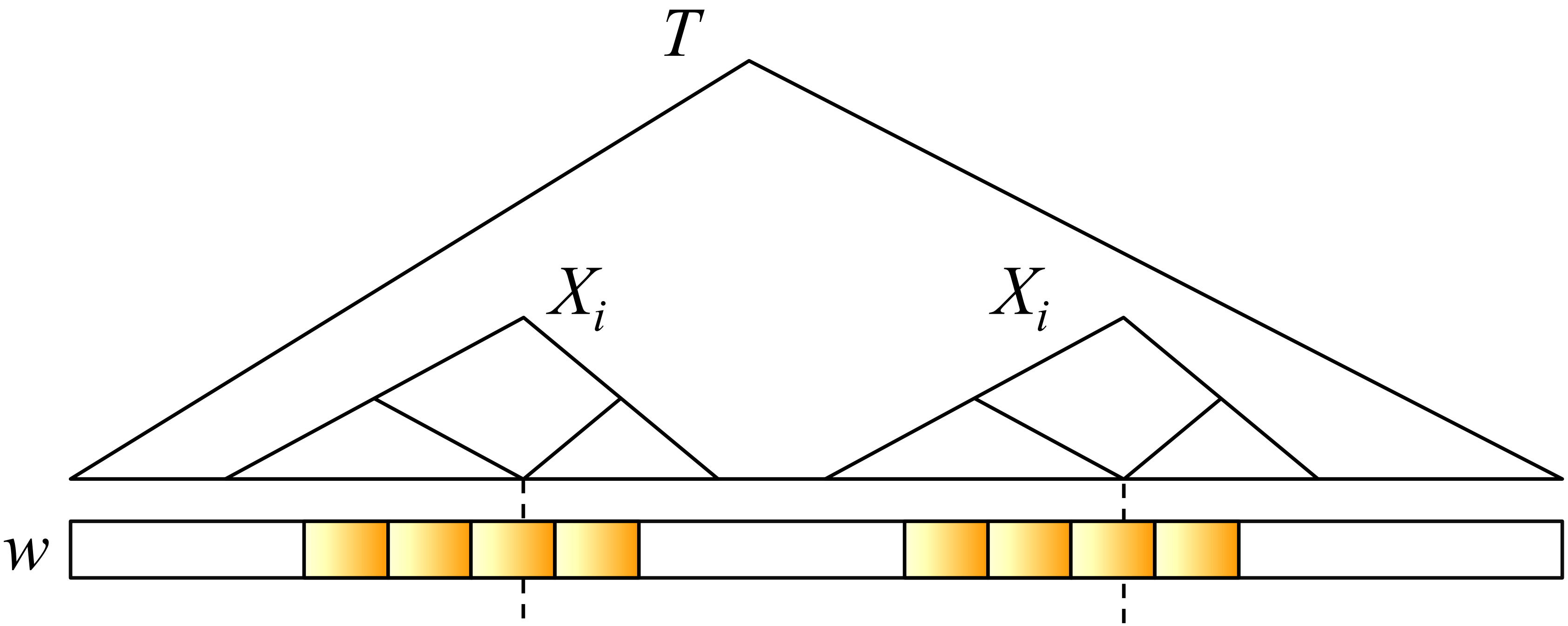
Stabbed Runs [Cont.]

- ✓ In any other occurrences of X_i in the derivation tree, the same run is stabbed by X_i.



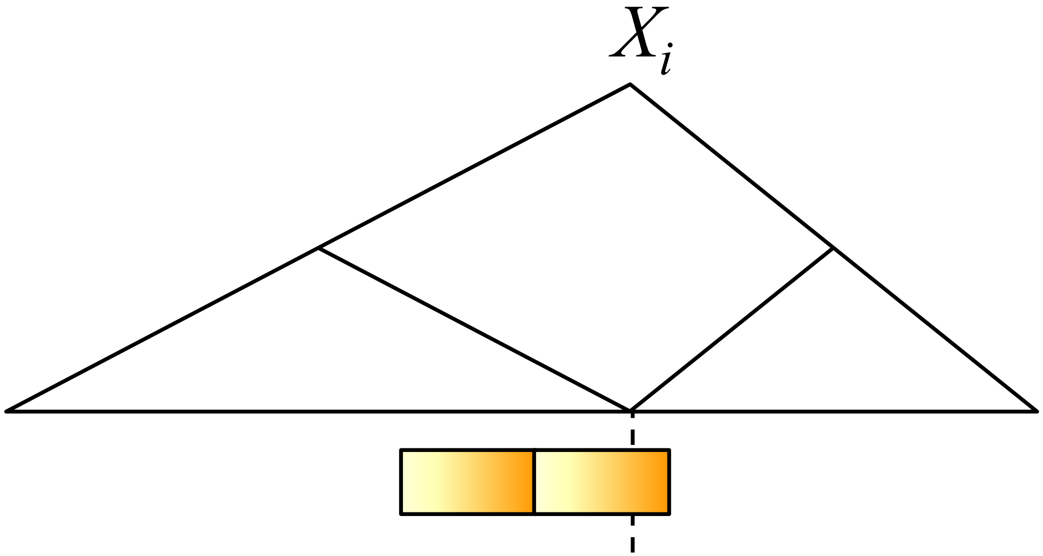
Stabbed Runs [Cont.]

- ✓ Computing runs in string w reduces to computing the stabbed runs for each variable X_i.



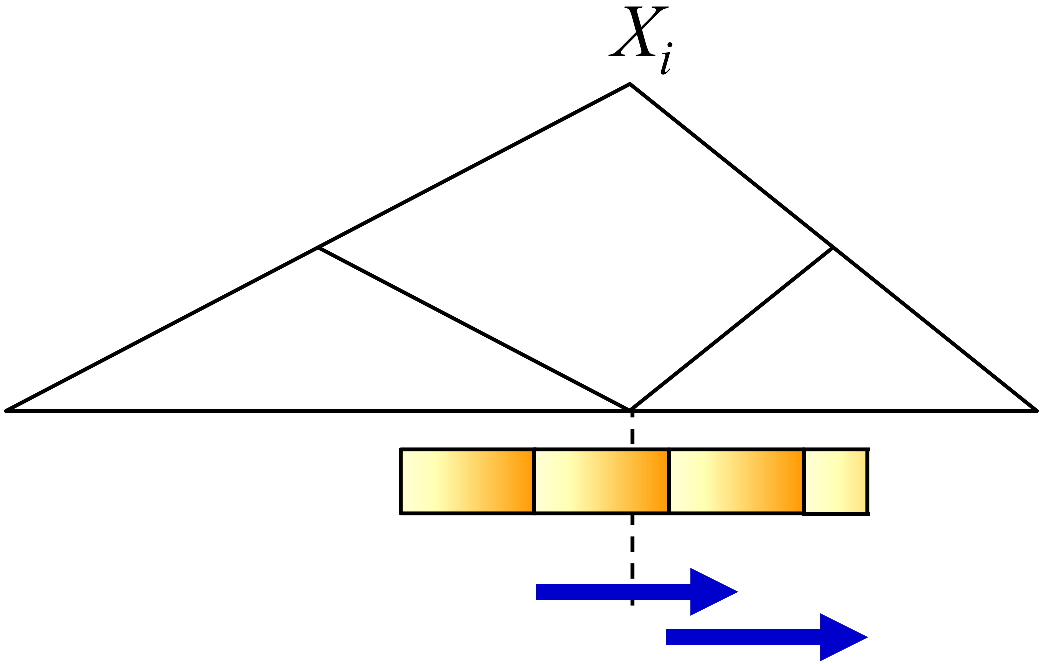
Stabbed Runs [Cont.]

- ✓ For each variable X_i, we first compute (the beginning and ending positions of) the stabbed squares.



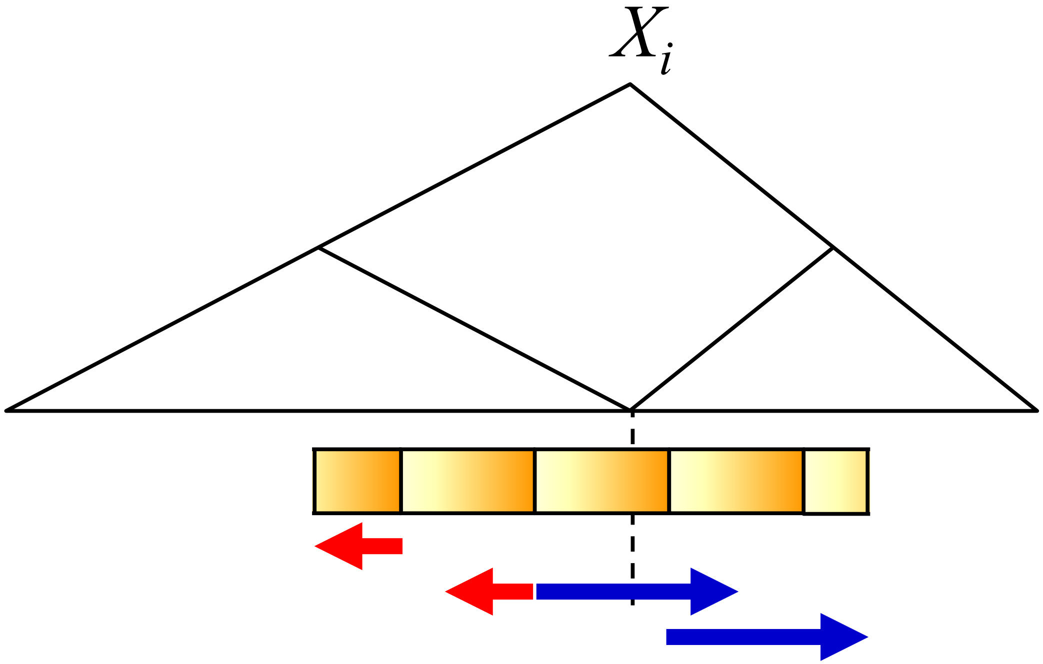
Stabbed Runs [Cont.]

- ✓ We then determine how long the periodicity continues to the right and to the left, using LCE.
  - ➢ We can efficiently perform LCE without expanding X_i.



Stabbed Runs [Cont.]

- ✓ We then determine how long the periodicity continues to the right and to the left, using LCE.
  - ➢ We can efficiently perform LCE without expanding X_i.



Finding Repetitions on SLP

Theorem [I et al. 2015]

$O(n \log N)$-size representation of all runs and squares can be computed in $O(n^3 h)$ time with $O(n^2)$ working space.

- ✓ There are at most N-1 runs [Bannai et al. 2017].
  ➔ Naive representation of runs requires
  $O(N) \subseteq O(2^n)$ space in the worst case.
- ✓ We can compactly represent all runs within $O(n \log N)$ space using periodicities.

Finding Palindromes from SLP

Problem 5 (finding palindromes on SLP)

Given an SLP S which represents a string w, compute all maximal palindromes of w.

maximal
palindromes

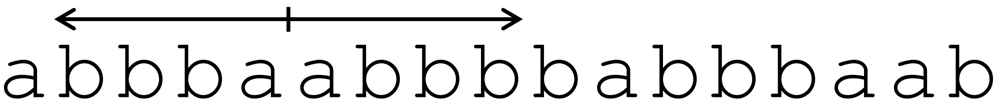
`a b b b a a b b b b a b b b a a b`

Finding Palindromes from SLP

Problem 5 (finding palindromes on SLP)

Given an SLP S which represents a string w, compute all maximal palindromes of w.

maximal palindromes



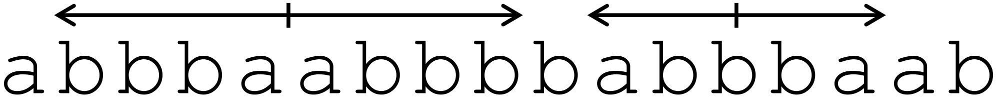
a b b b a a b b b b a b b b a a b

Finding Palindromes from SLP

Problem 5 (finding palindromes on SLP)

Given an SLP S which represents a string w, compute all maximal palindromes of w.

maximal palindromes



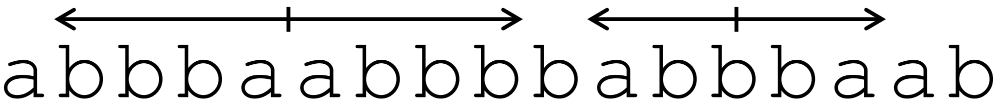
a b b b a a b b b b a b b b a a b

Finding Palindromes from SLP

Problem 5 (finding palindromes on SLP)

Given an SLP S which represents a string w, compute all maximal palindromes of w.

maximal palindromes

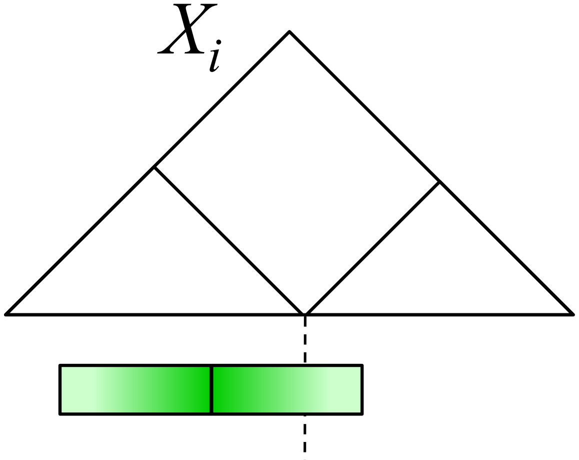


a b b b a a b b b b a b b b a a b

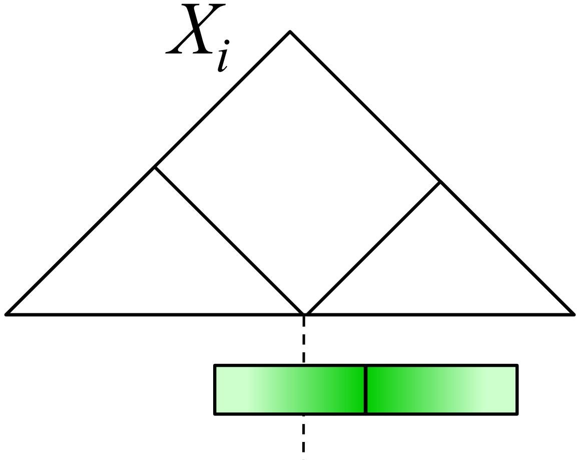
There are N integer positions and N-1 half-integer positions.
→ There are $2N$-1 maximal palindromes in a string of length N.

Stabbed Palindromes

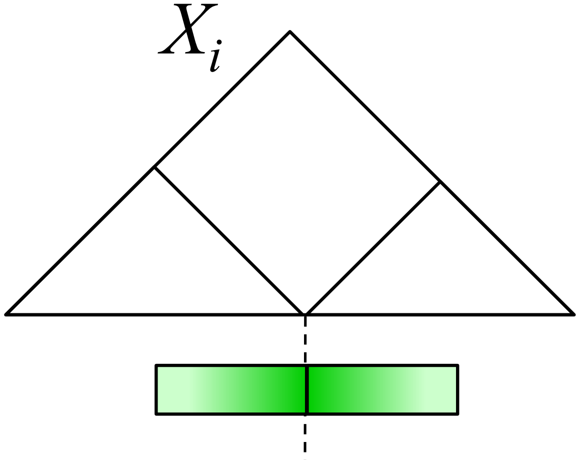
- For each variable X_i, there can be 3 different types of stabbed maximal palindromes.



Type 1



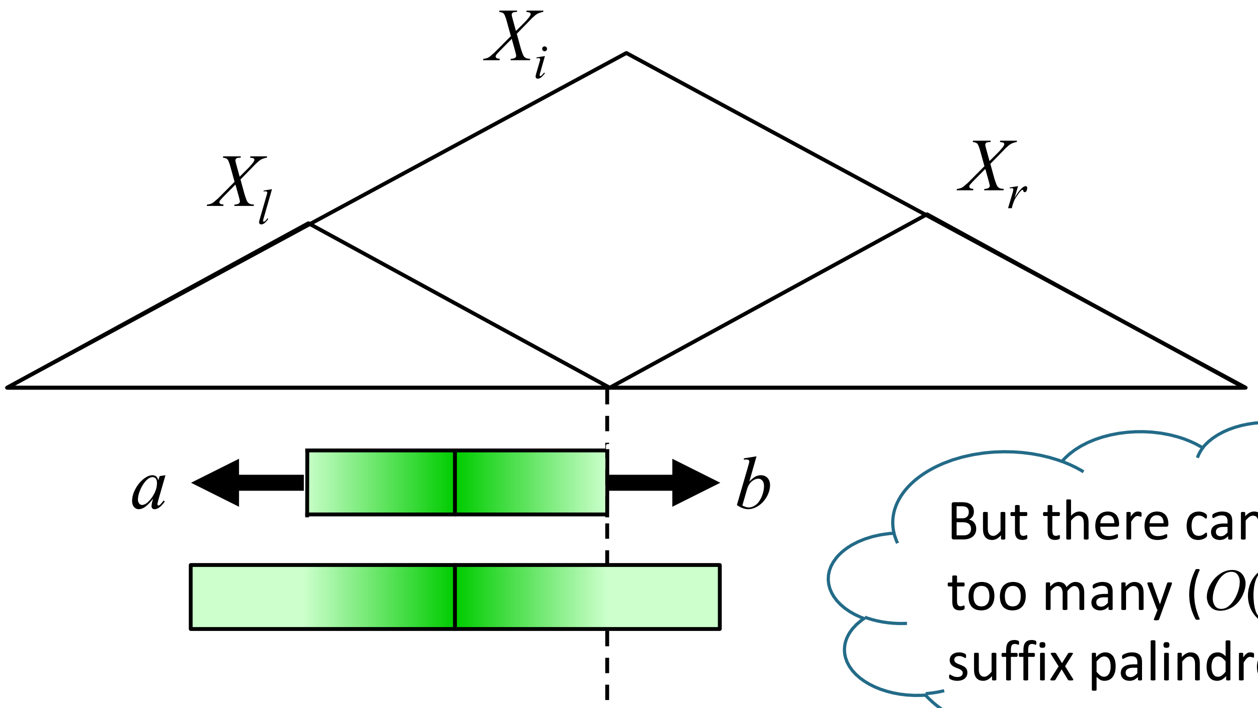
Type 2



Type 3

Computing Type 1 Palindromes

- ✓ Type 1 maximal palindromes of X_i can be computed by extending the arms of the suffix palindromes of X_l.



But there can be too many ($O(N)$) suffix palindromes...

Suffix Palindromes

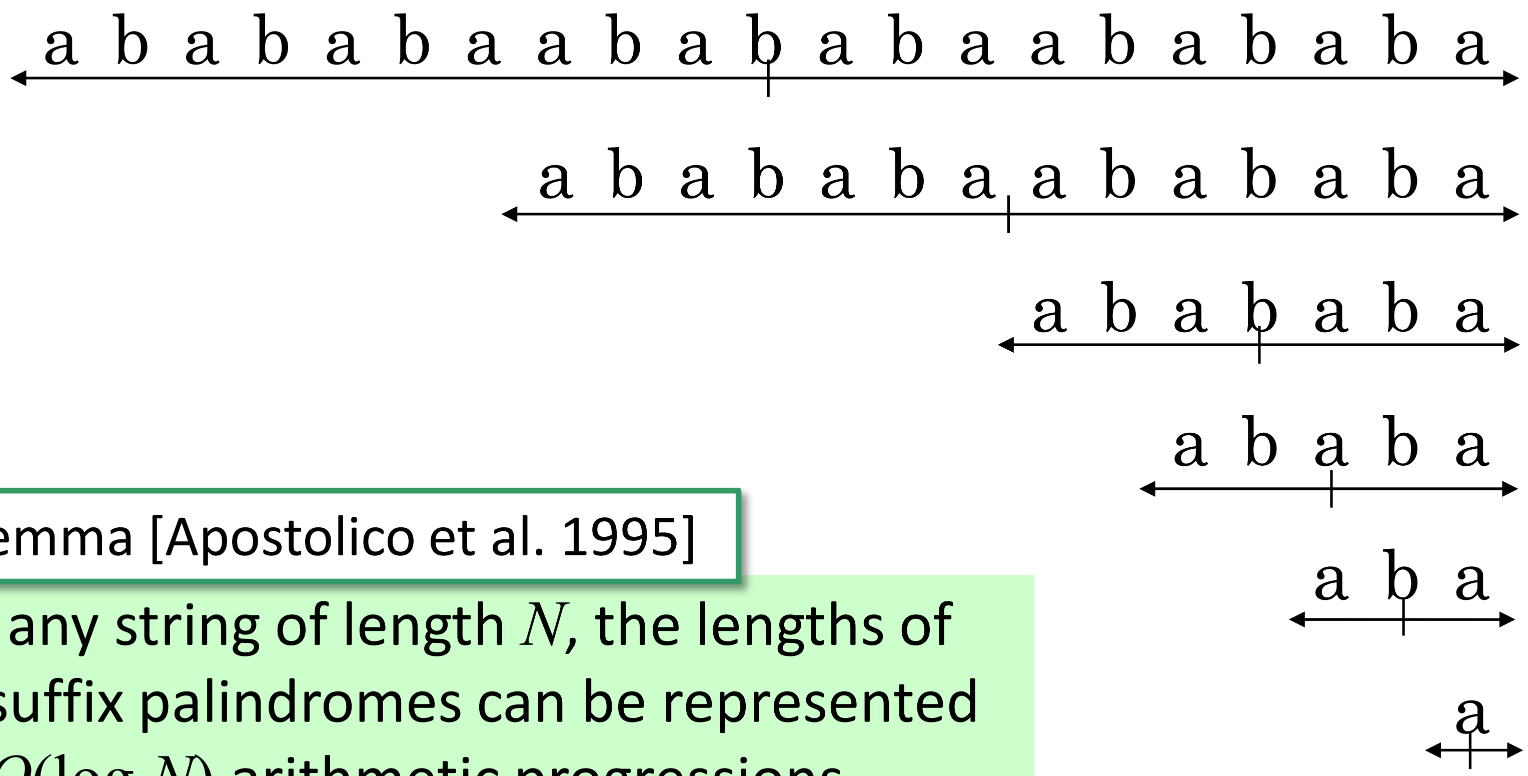
Lemma [Apostolico et al. 1995]

For any string of length N, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

- ✓ We can extend the suffix palindromes belonging to the same arithmetic progression <u>in a batch</u>.
- ✓ This batched LCE can be performed efficiently using the <u>periodicity of suffix palindromes</u>.

Batched LCE for Suffix Palindromes

a a b a b a b a a b a b a b a a b a b a b a



a b a b a b a a b a b a b a a b a b a b a

a b a b a b a a b a b a b a

a b a b a b a

a b a b a

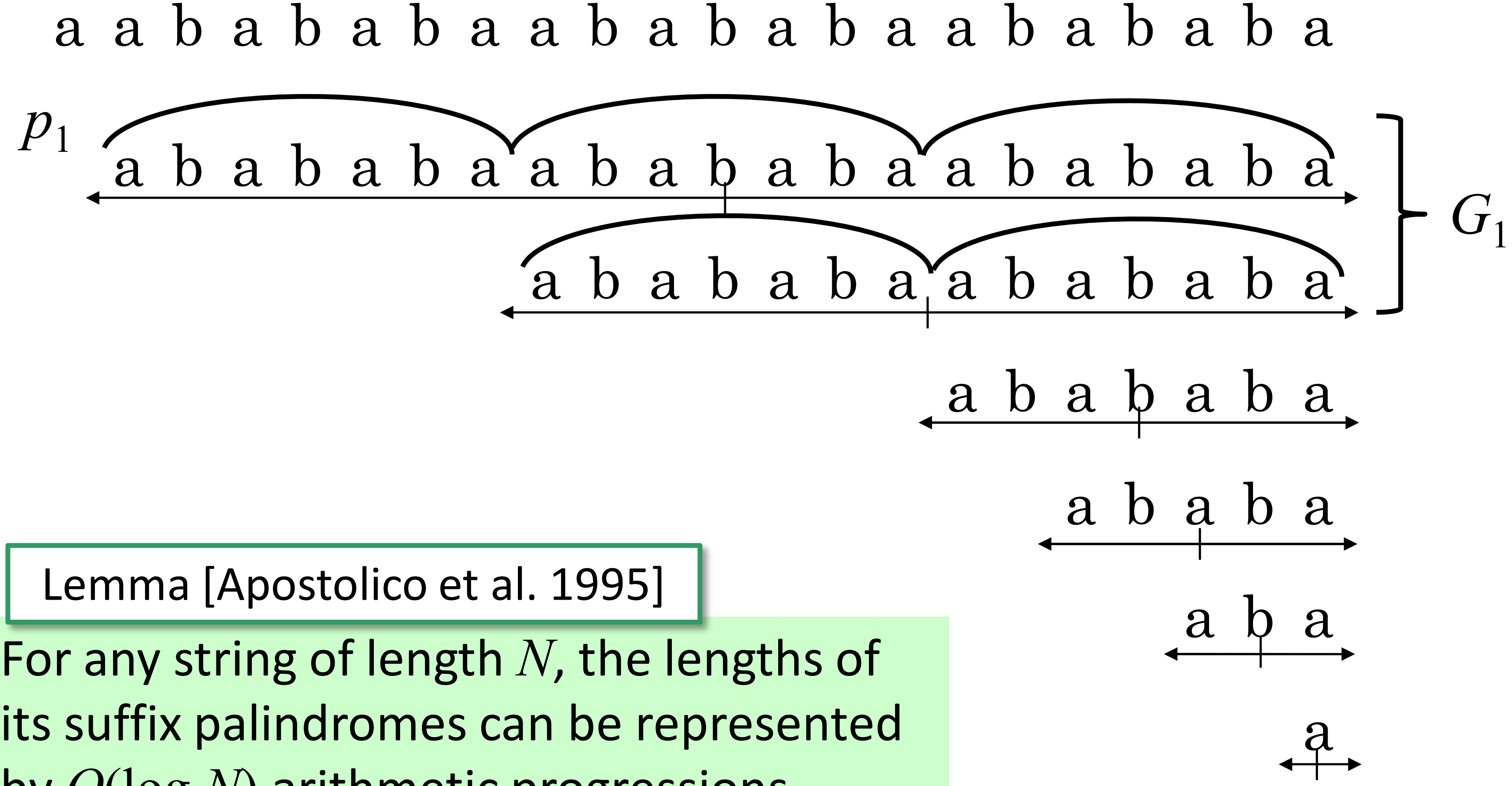
a b a

a

Lemma [Apostolico et al. 1995]

For any string of length N, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

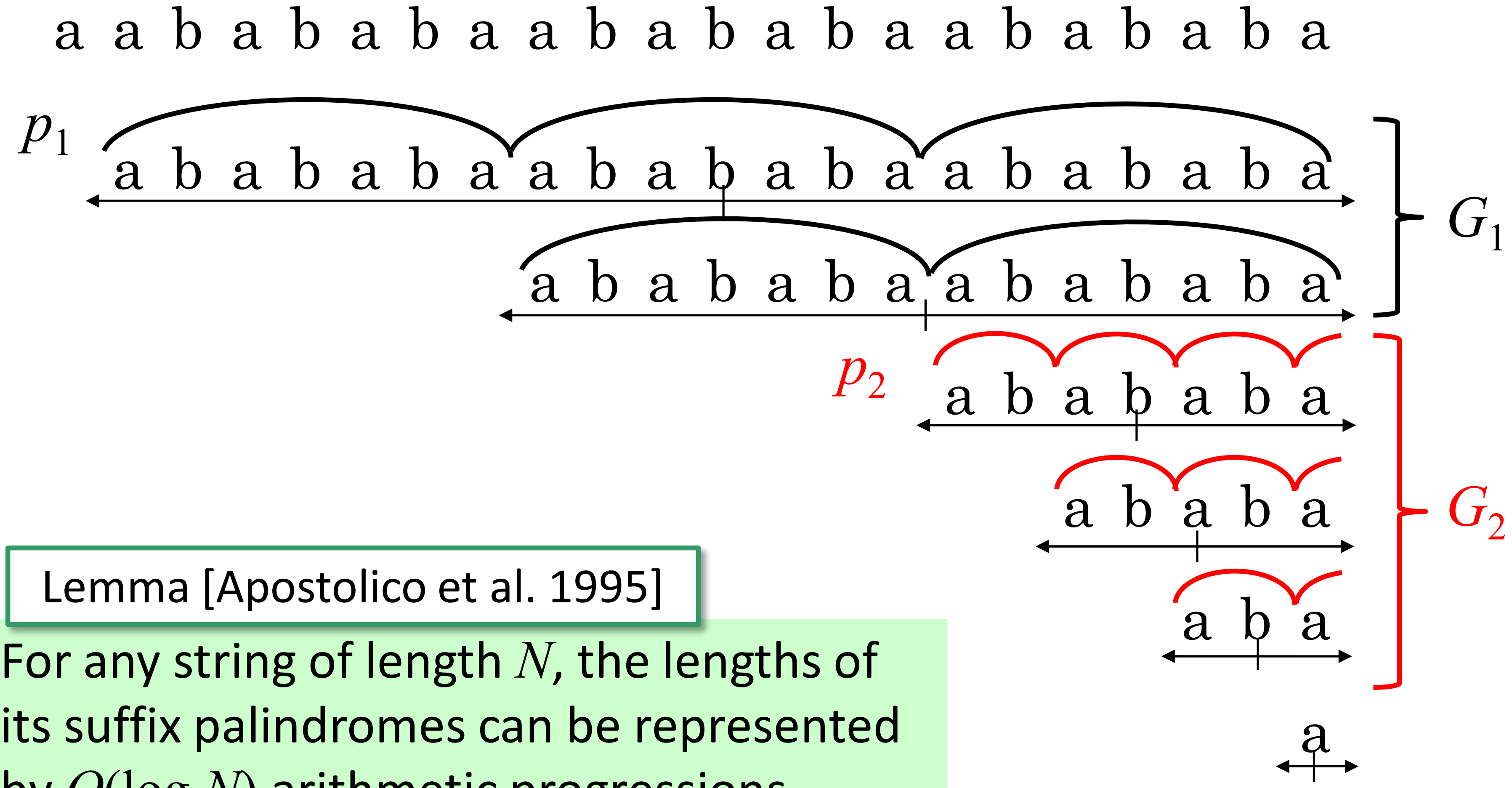
Batched LCE for Suffix Palindromes



Lemma [Apostolico et al. 1995]

For any string of length N, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

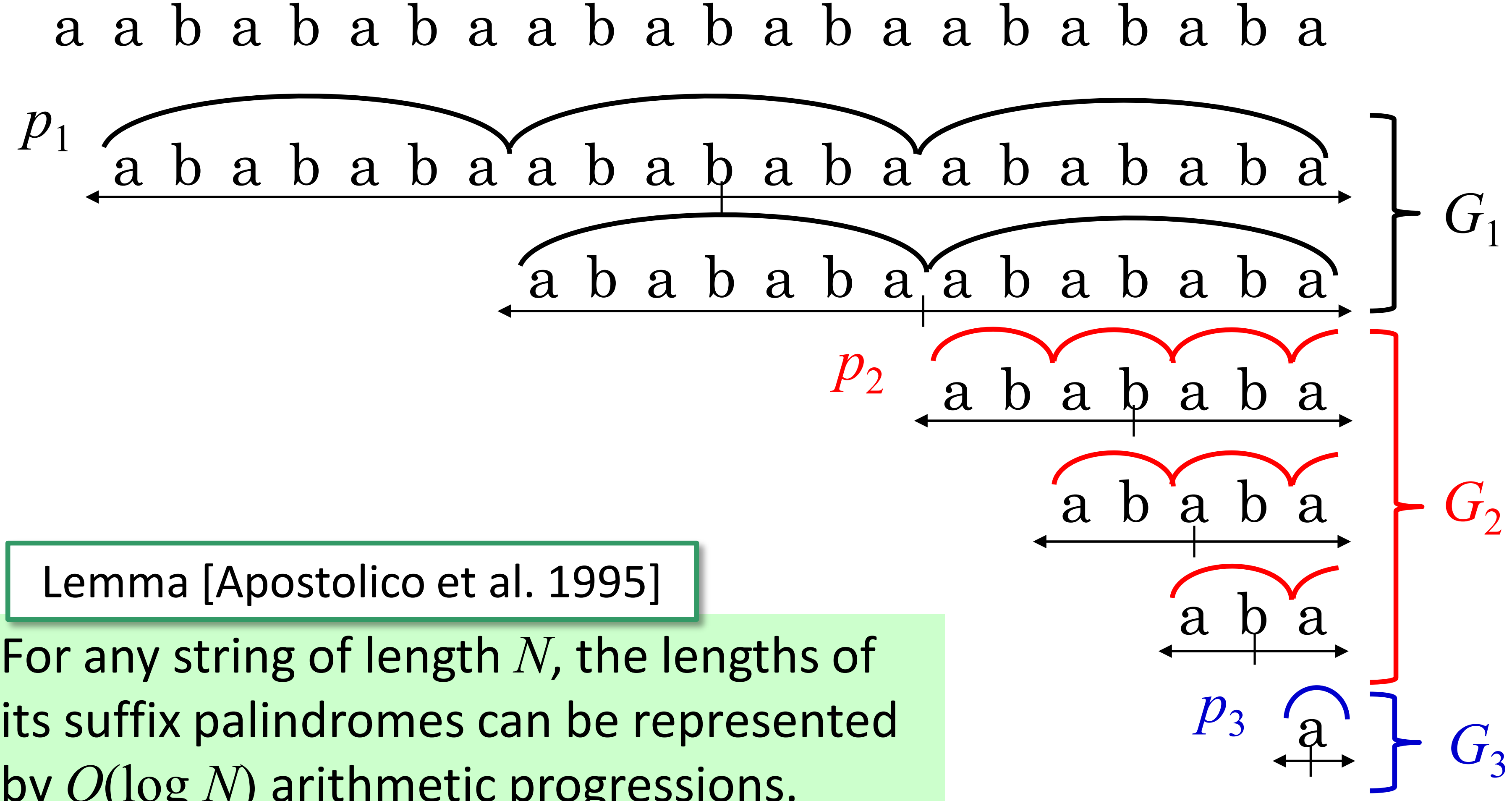
Batched LCE for Suffix Palindromes



Lemma [Apostolico et al. 1995]

For any string of length N, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

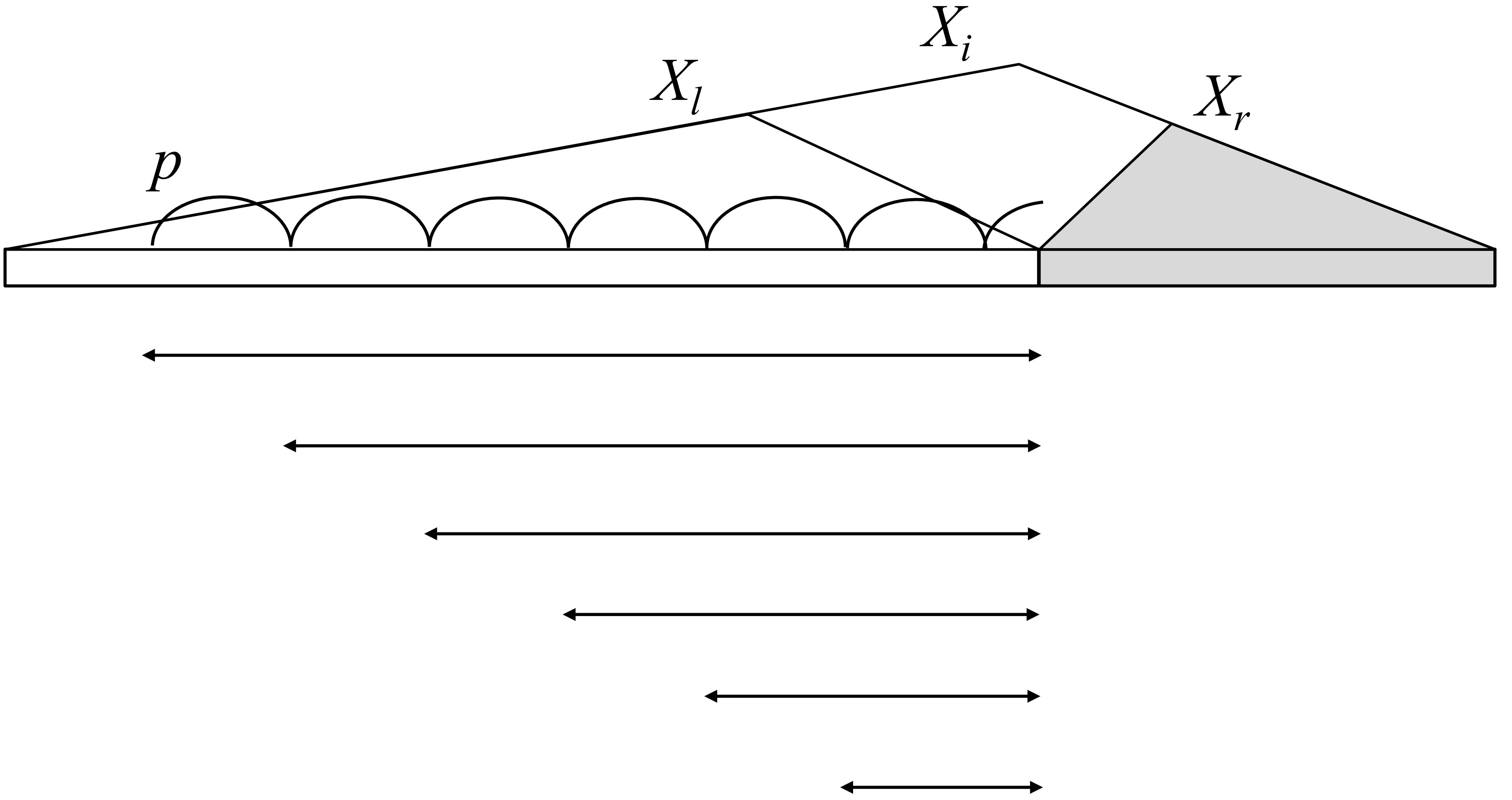
Batched LCE for Suffix Palindromes



Lemma [Apostolico et al. 1995]

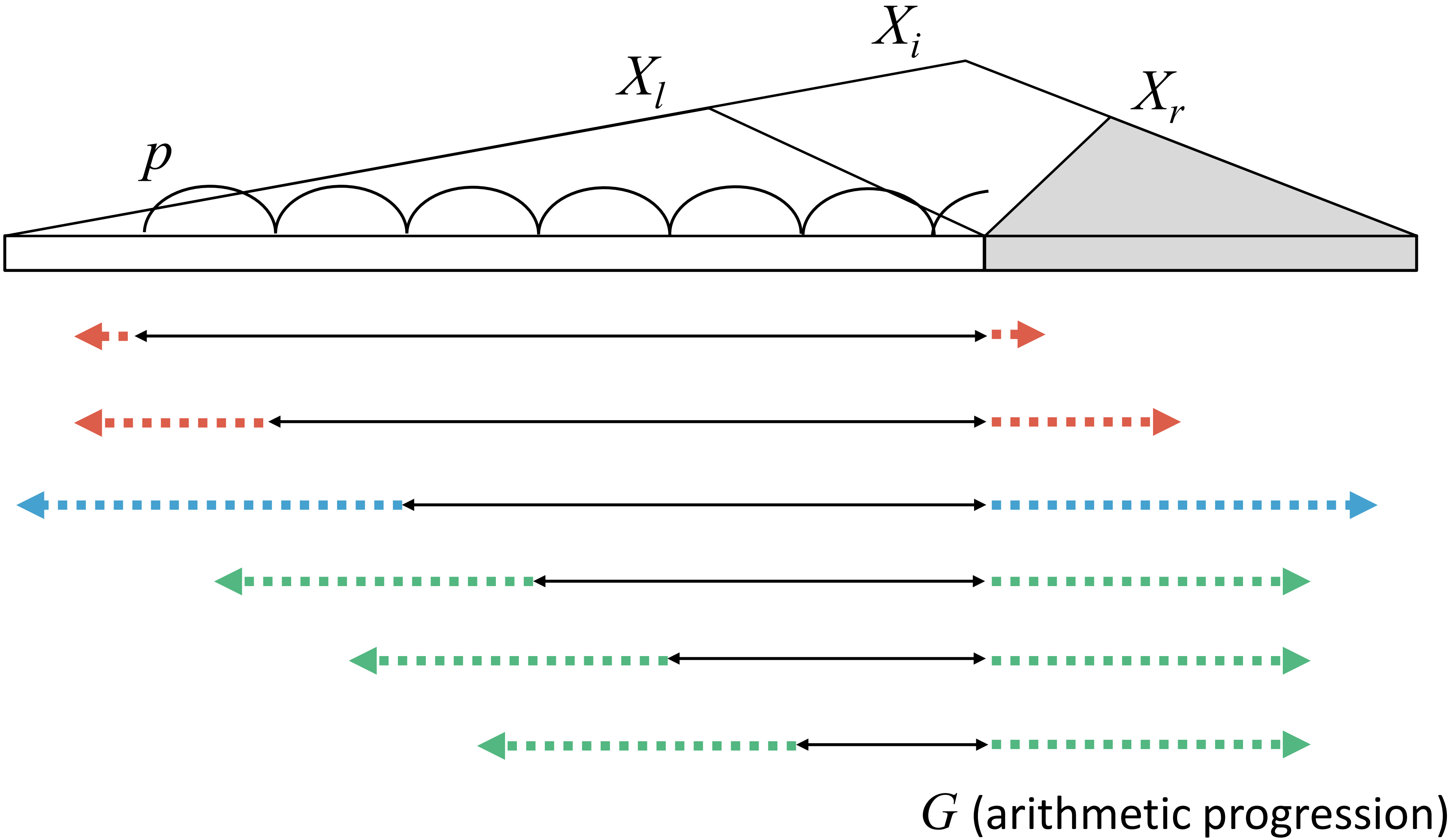
For any string of length N, the lengths of its suffix palindromes can be represented by $O(\log N)$ arithmetic progressions.

Batched LCE for Suffix Palindromes

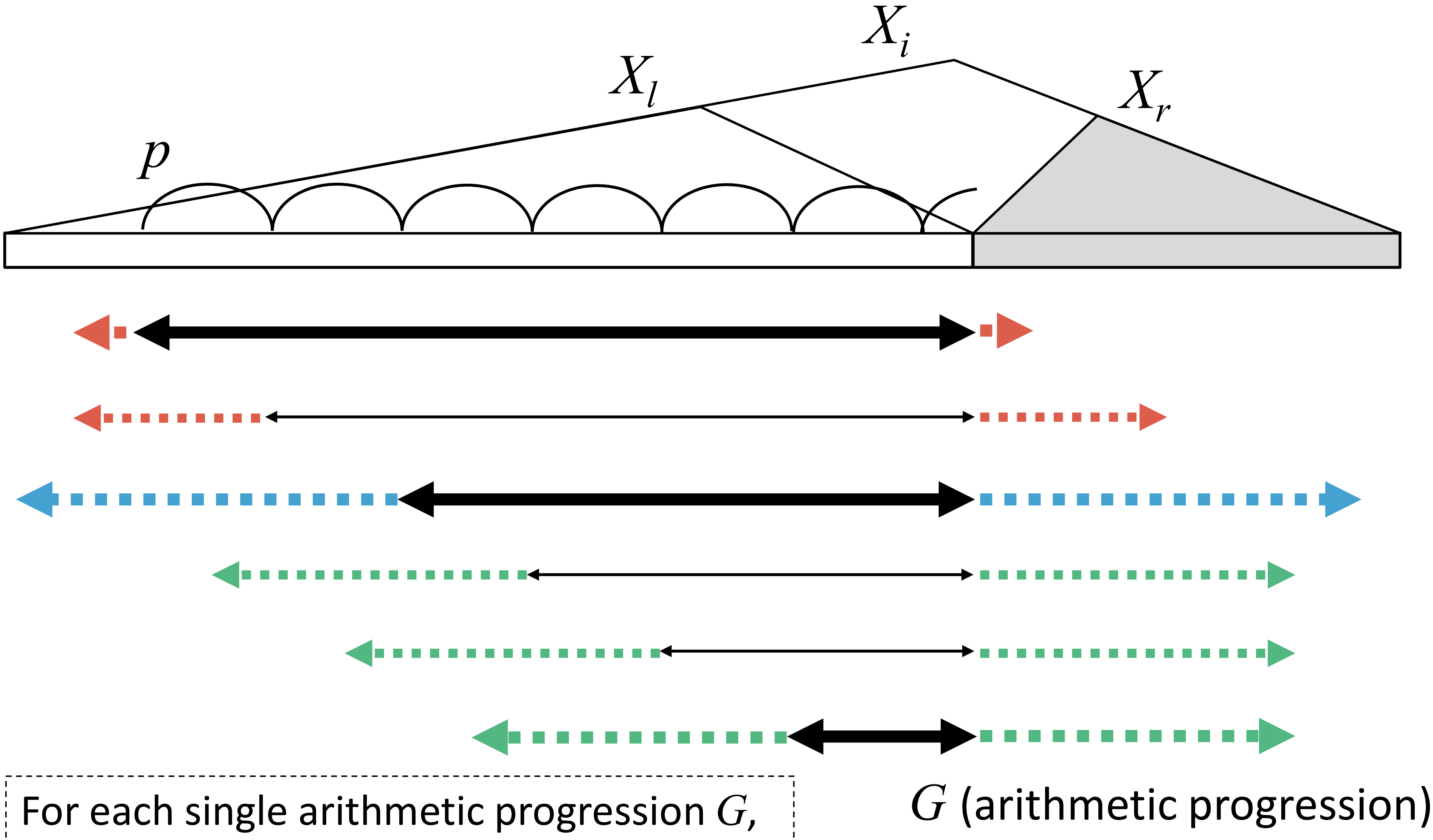


G (arithmetic progression)

Batched LCE for Suffix Palindromes



Batched LCE for Suffix Palindromes



For each single arithmetic progression G, three LCE queries are sufficient.

Finding Palindromes on SLP

Theorem [Matsubara et al. 2009]

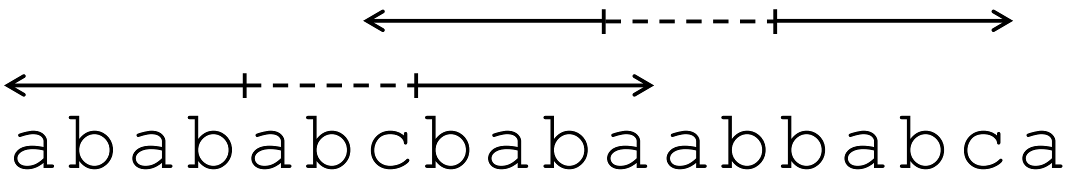
$O(n \log N)$-size representation of all maximal palindromes can be computed in $O(nh\,(n + h \log N))$ time using $O(n^2)$ space.

Finding Gapped Palindromes on SLP

Problem (finding gapped palindromes on SLP)

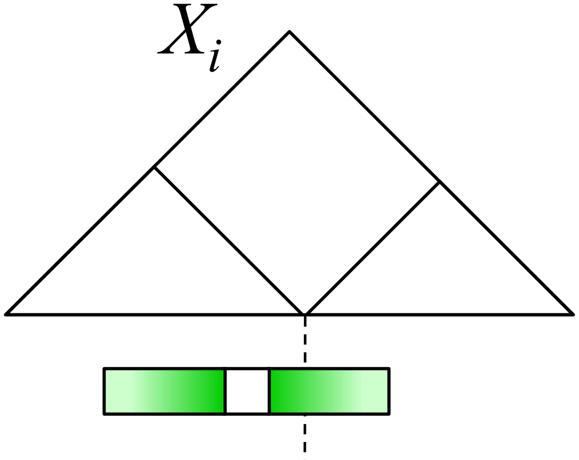
Given an SLP S which represents a string w and a positive integer g, compute all g-gapped palindromes that occur in w.

3-gapped palindromes (g = 3)

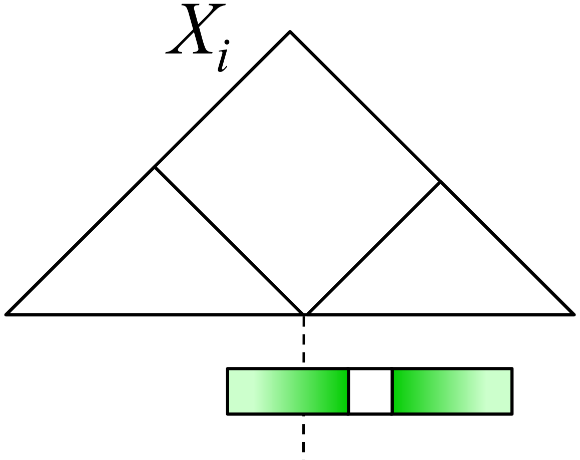

abababcbabaabbabca

Stabbed g-gapped Palindromes

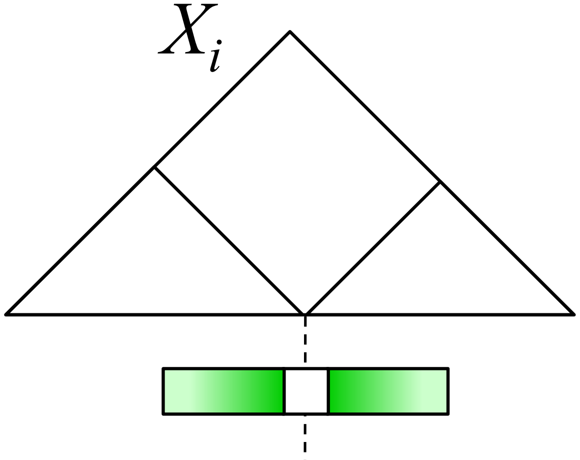
- ✓ There are 3 types of g-gapped palindromes which are stabbed by each variable X_i.



Type 1



Type 2



Type 3

Finding Gapped Palindromes on SLP

Theorem [I et al. 2015]

$O(n\,(\log N + g))$-size representation of all g-gapped palindromes can be computed in $O(nh\,(n^2 + g \log N))$ time using $O(n^2)$ space.

- ✓ Unfortunately, Apostolico et al.'s lemma does not hold for gapped palindromes.
- ✓ Instead, we can use a similar technique to the solution for computing stabbed squares.

More on LCE, height h, and Stabbing

We have seen that many of the CSP algorithms on SLPs use

- **LCE (Longest Common Extension) queries,**

their efficiency depends on

- **the height h of the derivation tree,**

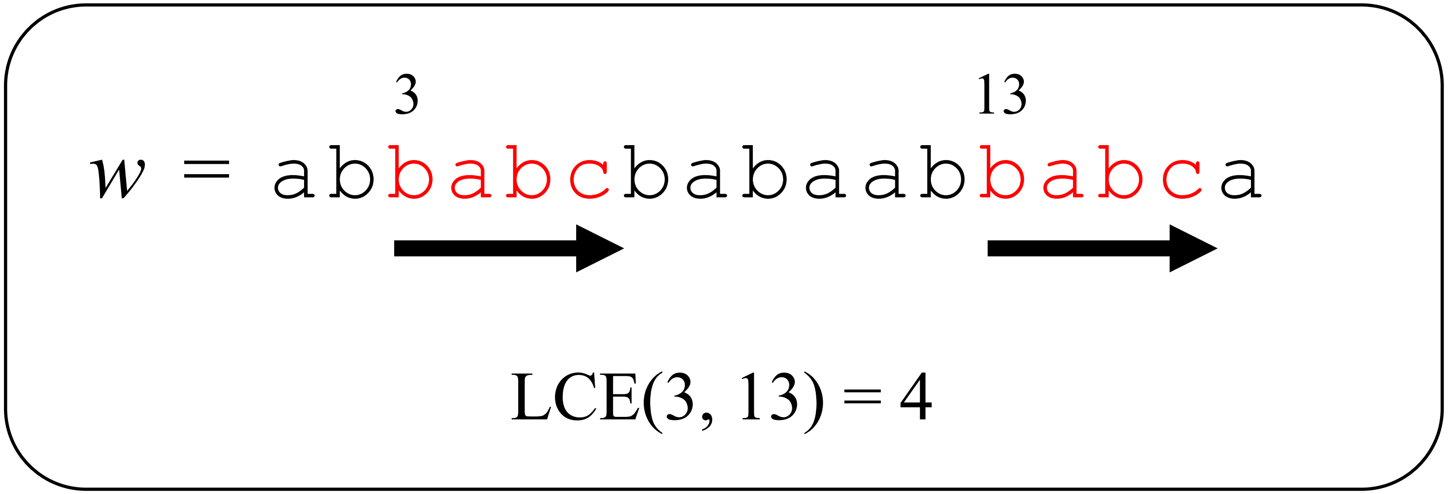
and their key concepts are

- **stabbed occurrences at variable boundaries.**

In the next slides, we will briefly review
the recent developments on these three concepts.

LCE (longest common extension) queries

- ✓ Recall that extension of periodicities (for runs) and extension of arms (for palindromes) can be efficiently computed by an LCE query on SLP.
- ✓ LCE(i, j) for string w returns the length of the longest common prefix of $w[i..N]$ and $w[j..N]$.



LCE(3, 13) = 4

LCE on grammar-compressed string

Theorem [I 2017]

A data structure of size $O(n + z \log (N / z))$ which answers LCE queries on SLP in $O(\log N)$ time can be constructed in $O(n \log (N / n))$ time.

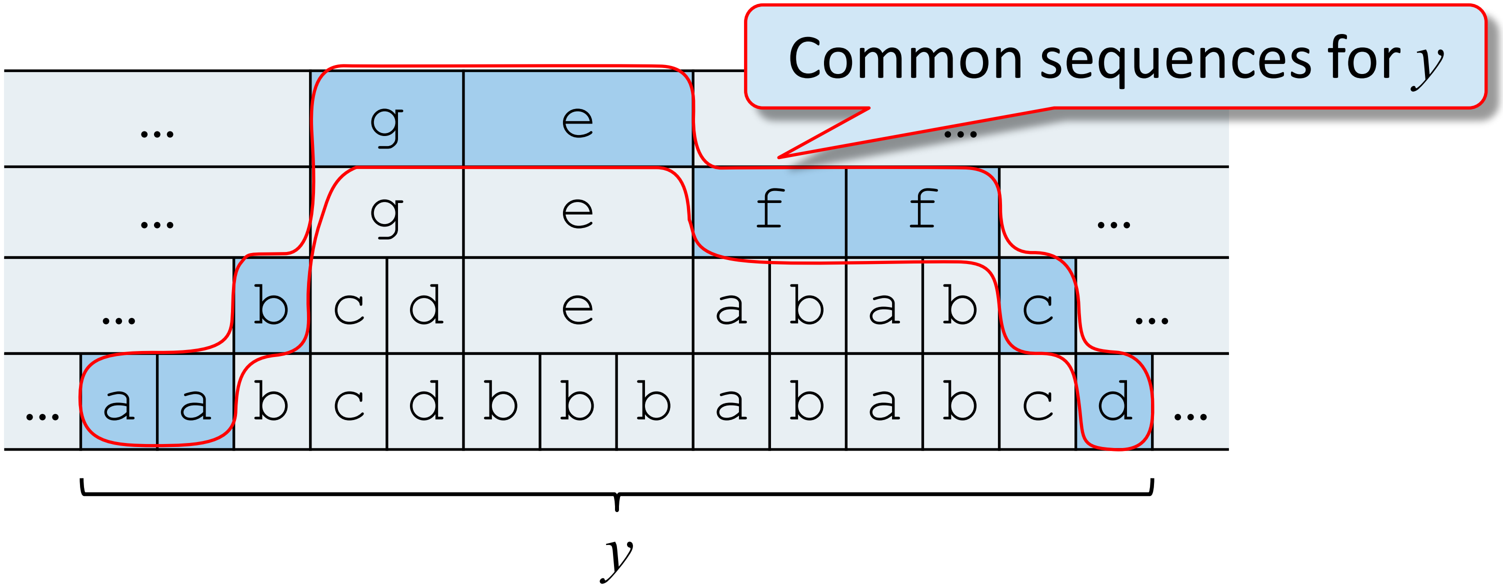
z = size of LZ77 factorization

I's algorithm above uses a kind of locally consistent parsing called *Recompression* [Jez 2015] that transforms a given SLP into another small SLP of size $O(z \log (N / z))$.

LCE with Recompression

In the grammar produced by Recompression, the occurrences of the same substring are compressed "almost" in the same way.

In each occurrence of a substring y, there is a unique sequence of symbols called "*common sequences*".

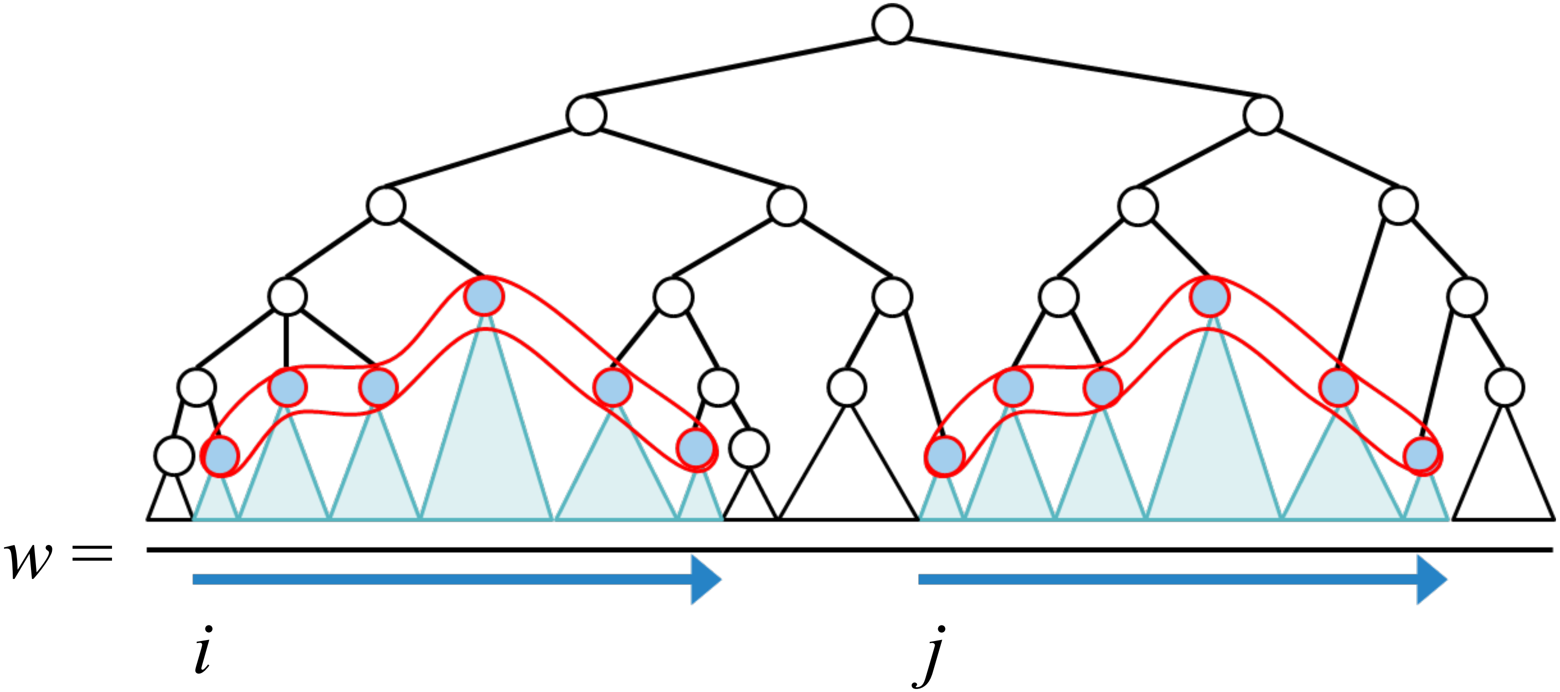


Common sequences for y

LCE with Recompression

$\mathrm{LCE}(i, j)$ can be computed by matching the common sequences of the LCE sub-strings that begin at positions i and j.

# of traversed nodes is bounded by $O(\log N)$.



Balancing SLPs

- ☐ An SLP is said to be *balanced* if its height $h = O(\log N)$.
- ☐ Given an SLP of size n, all existing approximation algorithms to the smallest grammar
  - AVL grammar [Rytter 2003];
  - α-balanced grammar [Charikar et al. 2005];
  - Recompression [Jez 2015];

  are able to produce a balanced SLP.
  But their grammar sizes blow up to $O(g \log(N/g))$, which can be larger than n when n is quite small.

Balancing SLPs [cont.]

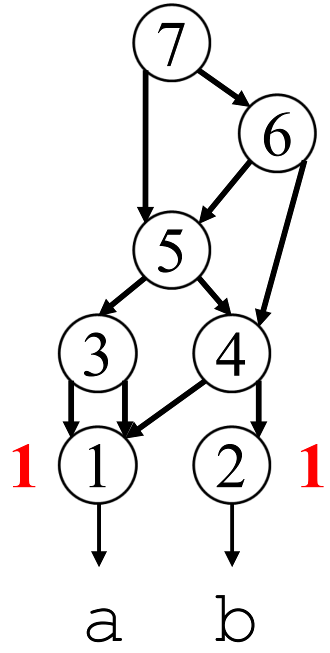
- Recently, Ganardi et al. (FOCS 2019) showed how to transform a given SLP of size n into a balanced SLP of size $O(n)$.

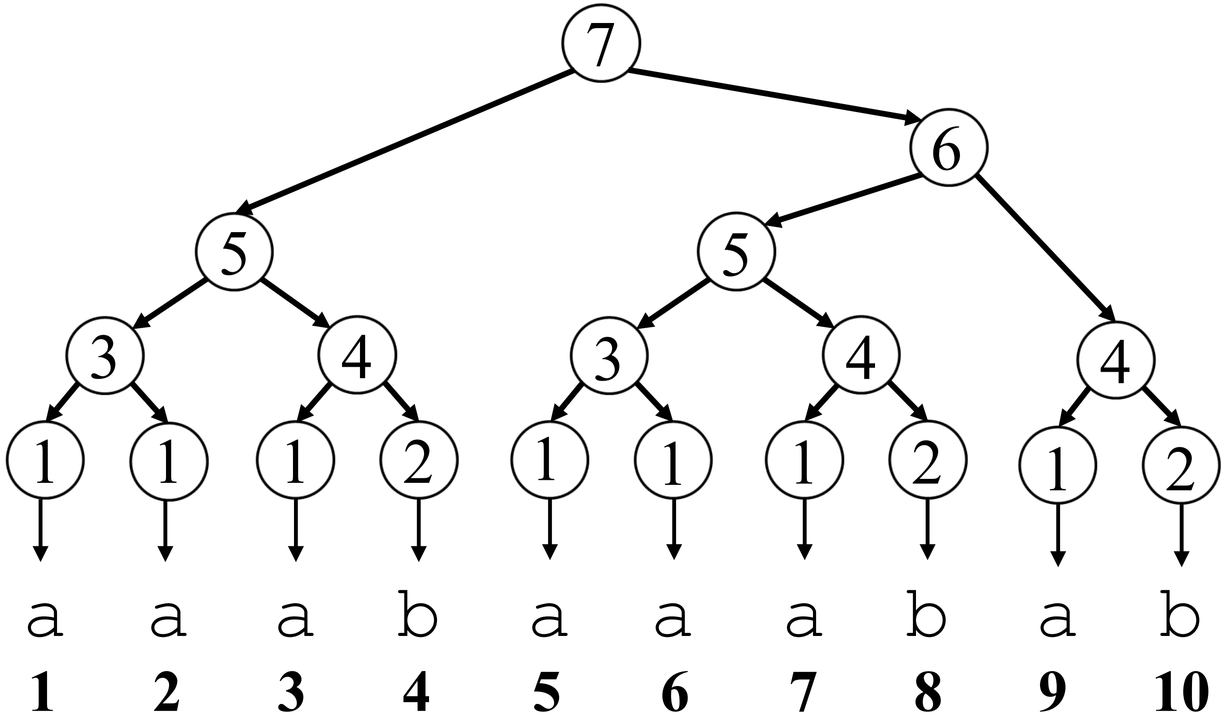
➔ New $O(\log N)$-time random access and $O(m + \log N)$-time substring extraction algorithms with $O(n)$ space, which alternate Bille et al.'s previous algorithms.

$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree

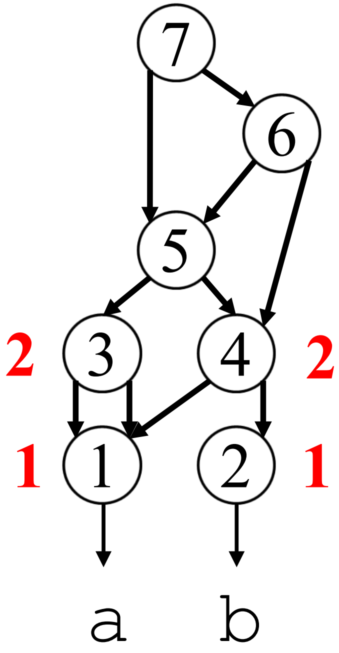




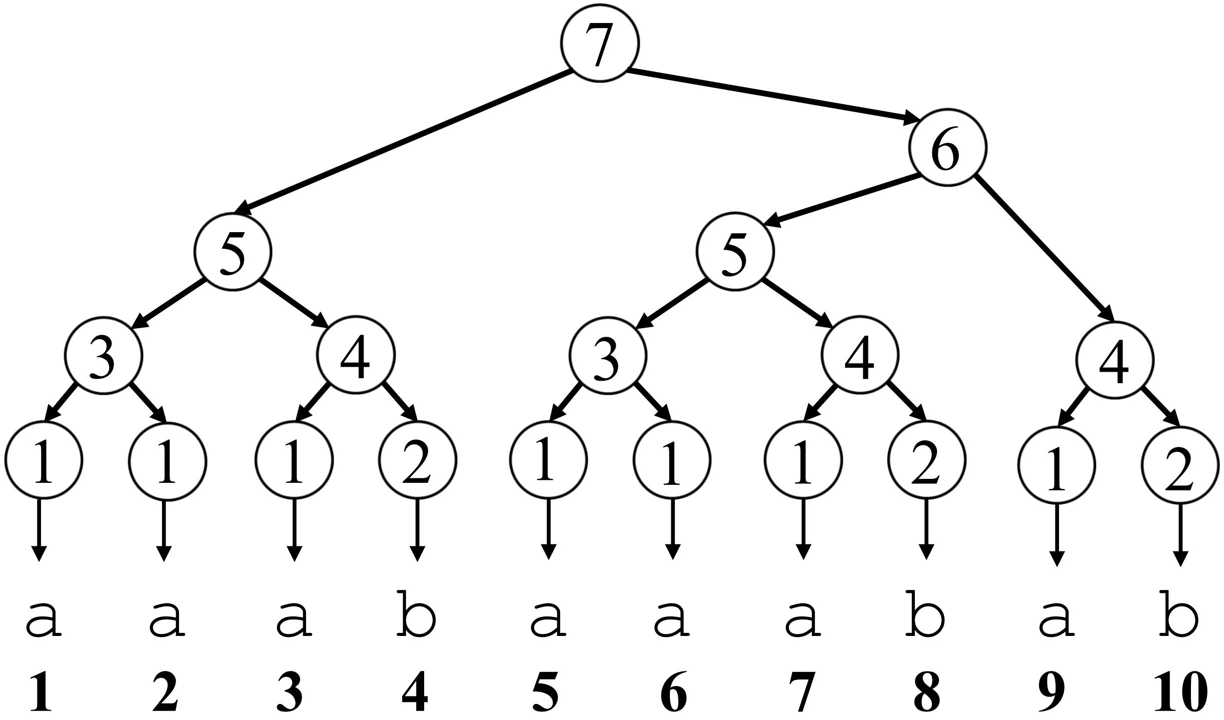
We precompute the **decompression length** of every variable in a bottom-up manner.

$O(\log N)$-time Random Access

Balanced SLP



Derivation Tree

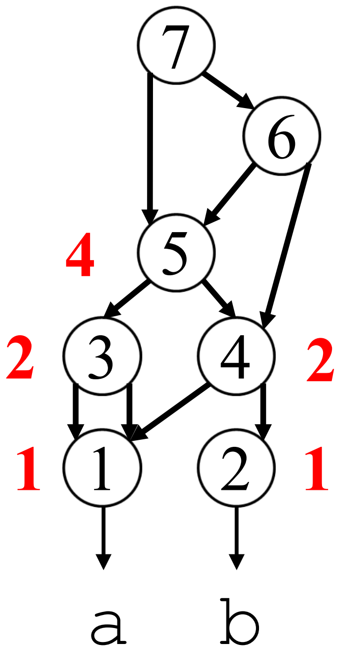


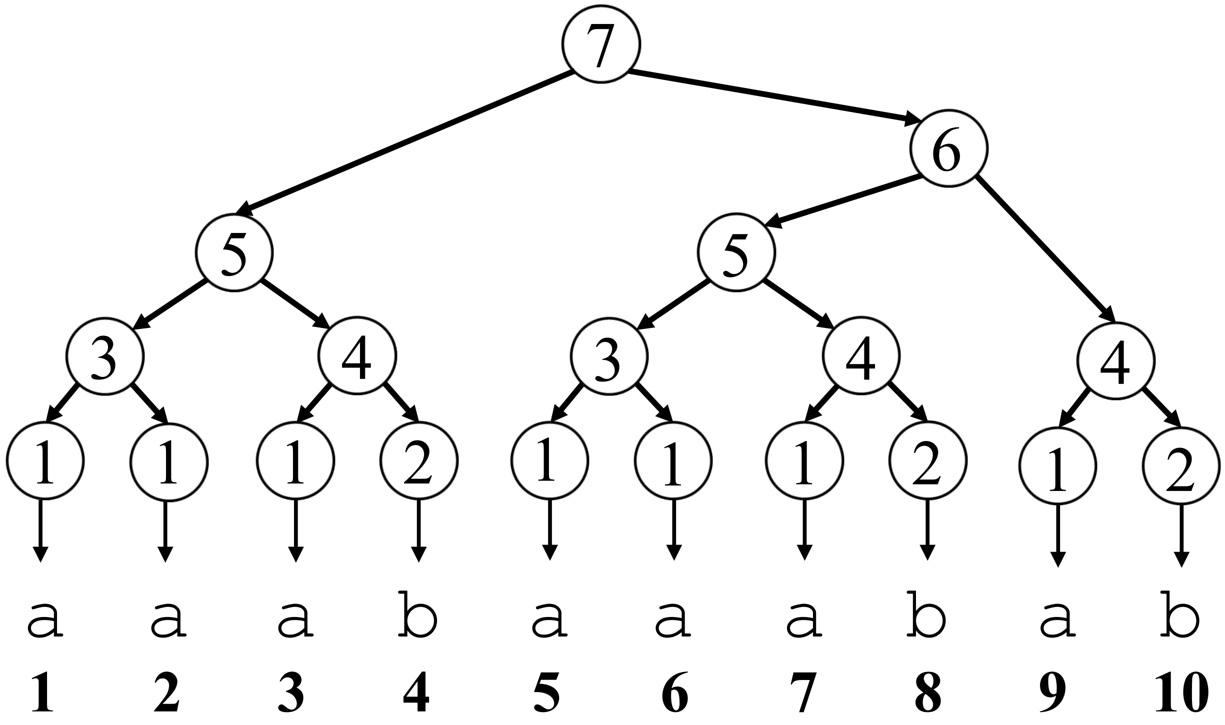
We precompute the **decompression length** of every variable in a bottom-up manner.

$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree



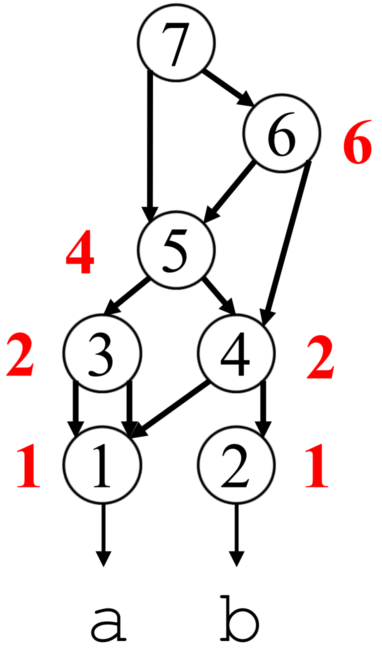


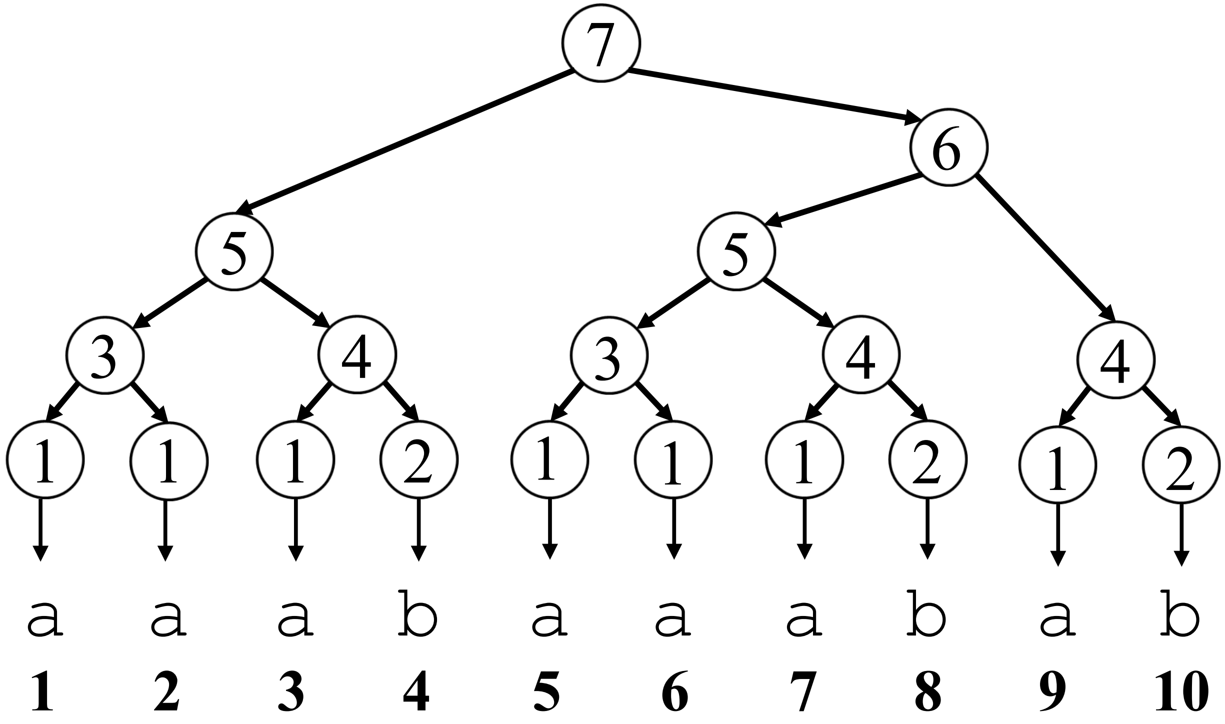
We precompute the **decompression length** of every variable in a bottom-up manner.

$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree



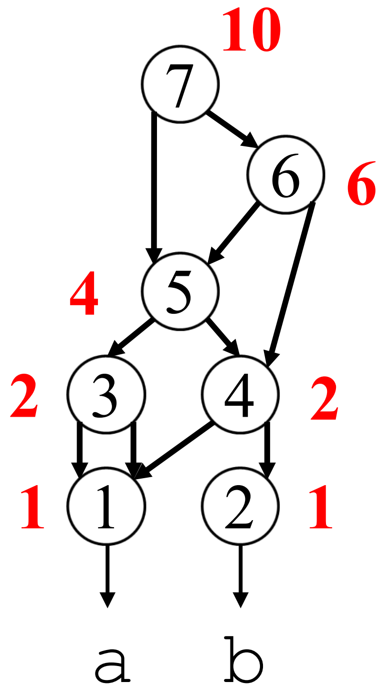


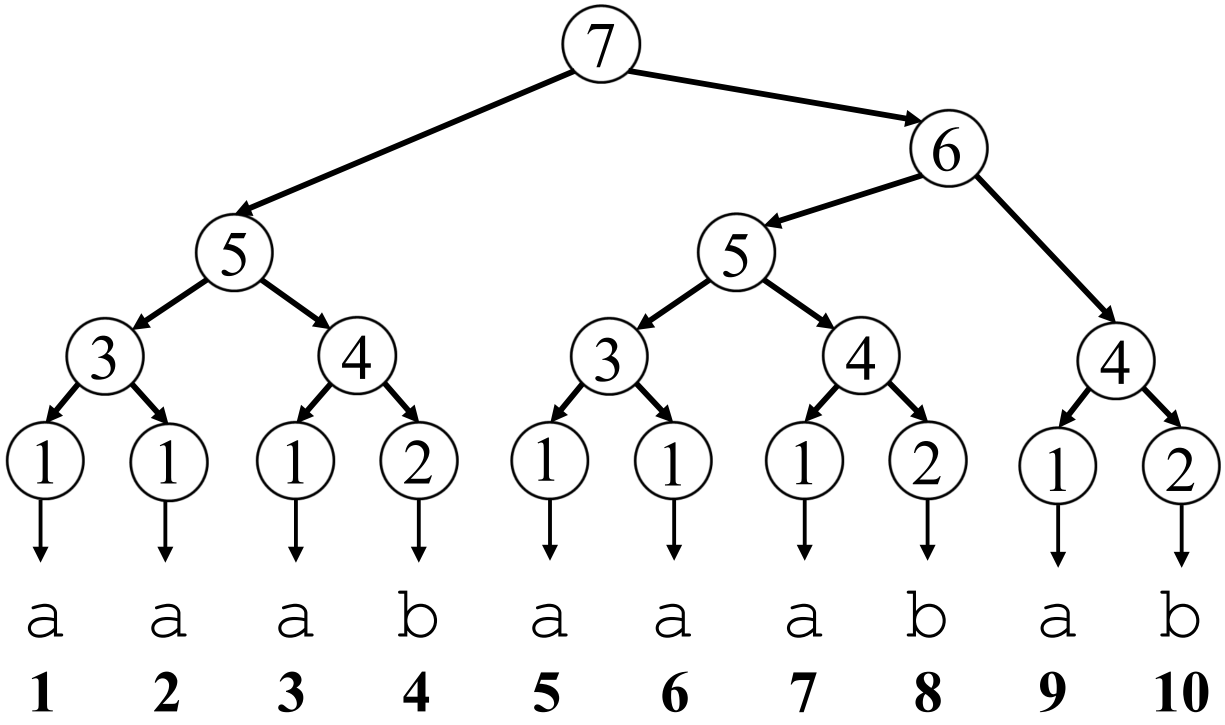
We precompute the **decompression length** of every variable in a bottom-up manner.

$O(\log N)$-time Random Access

Balanced SLP

Derivation Tree

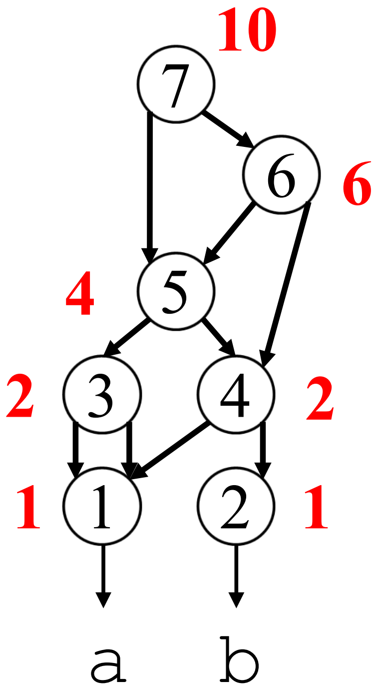




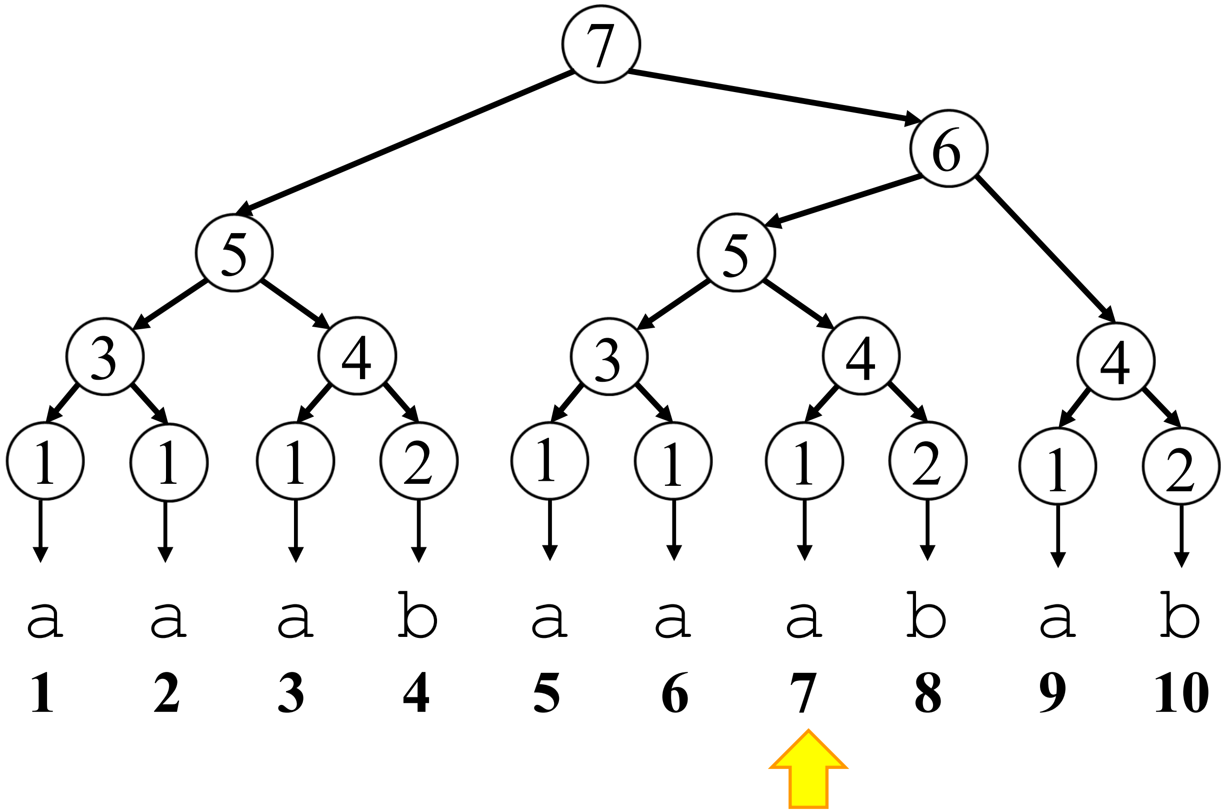
We precompute the **decompression length** of every variable in a bottom-up manner.

$O(\log N)$-time Random Access

Balanced SLP



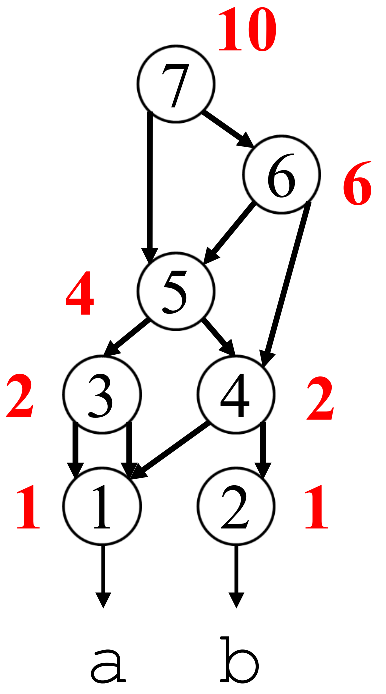
Derivation Tree



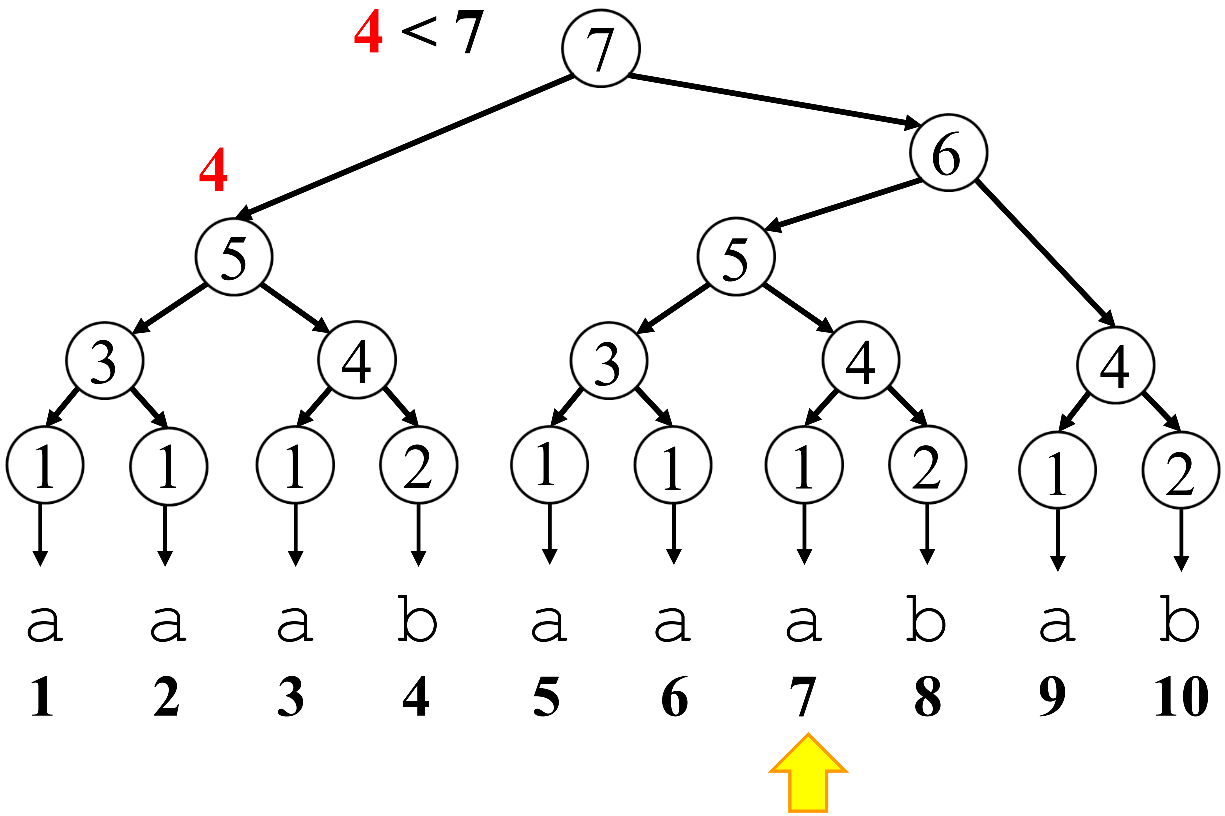
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



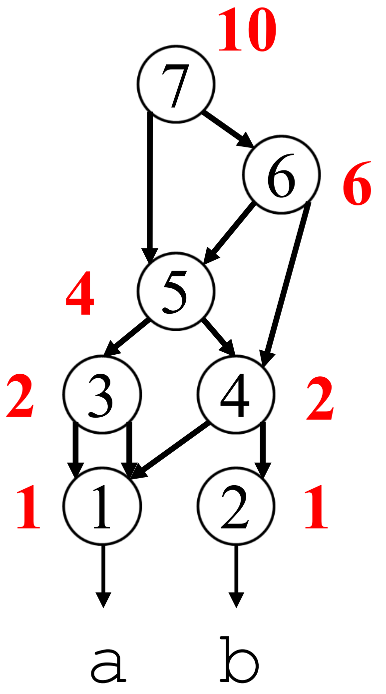
Derivation Tree



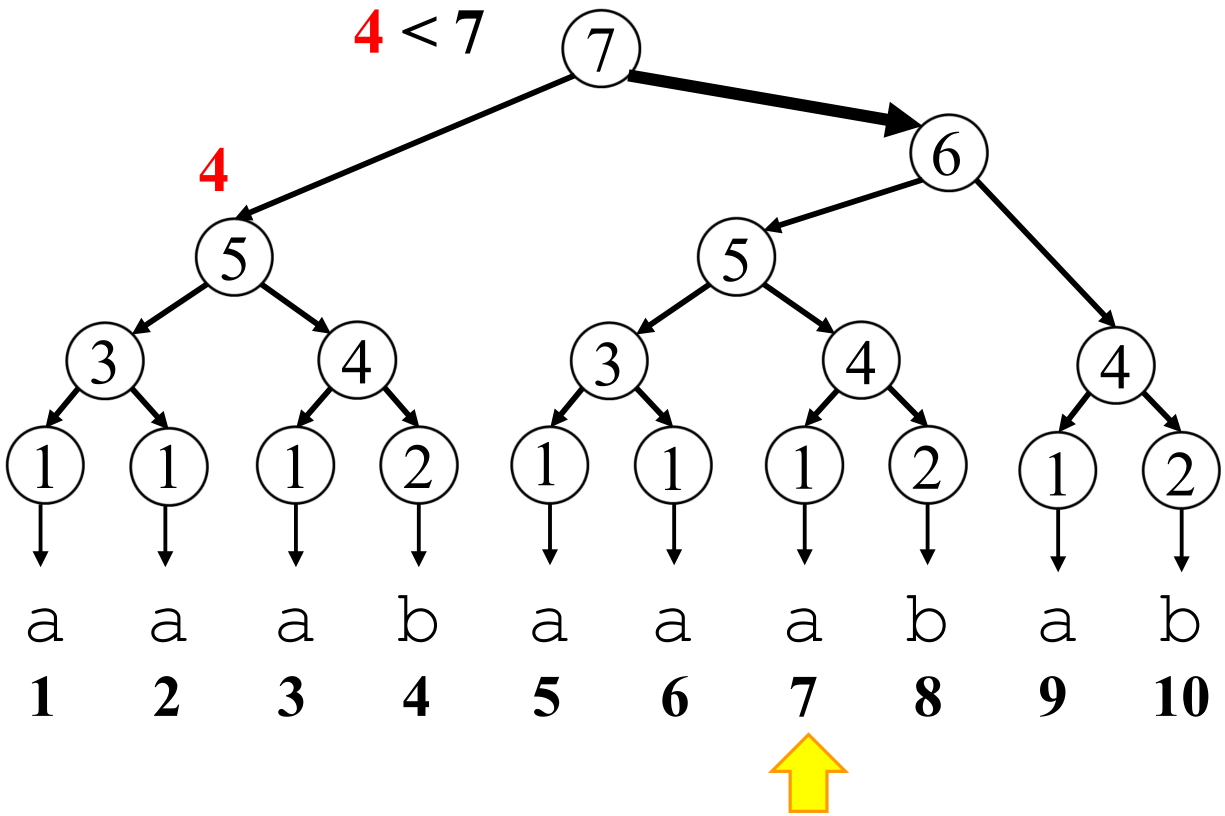
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



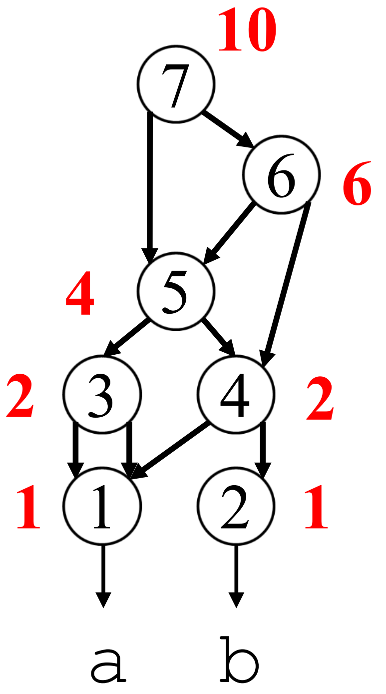
Derivation Tree



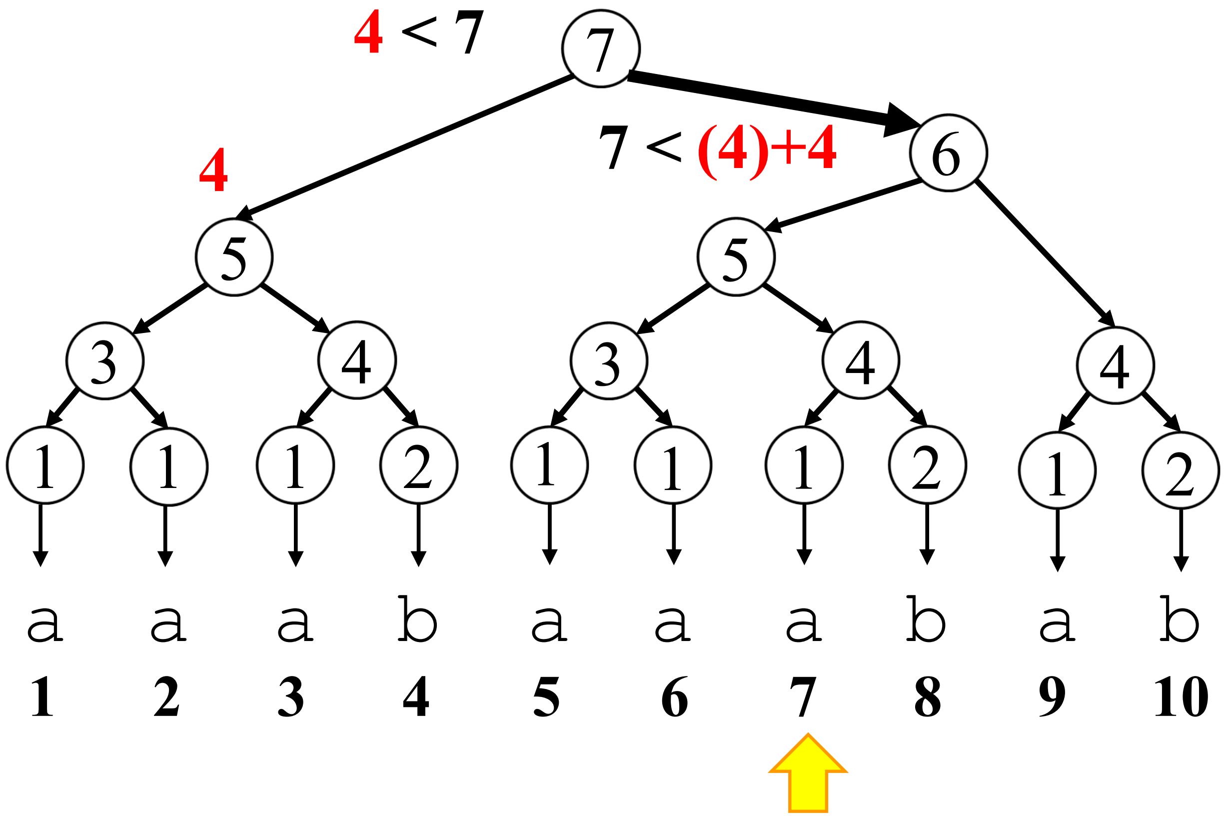
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



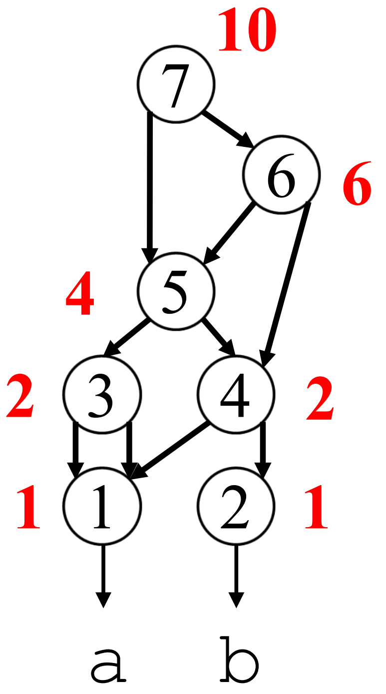
Derivation Tree



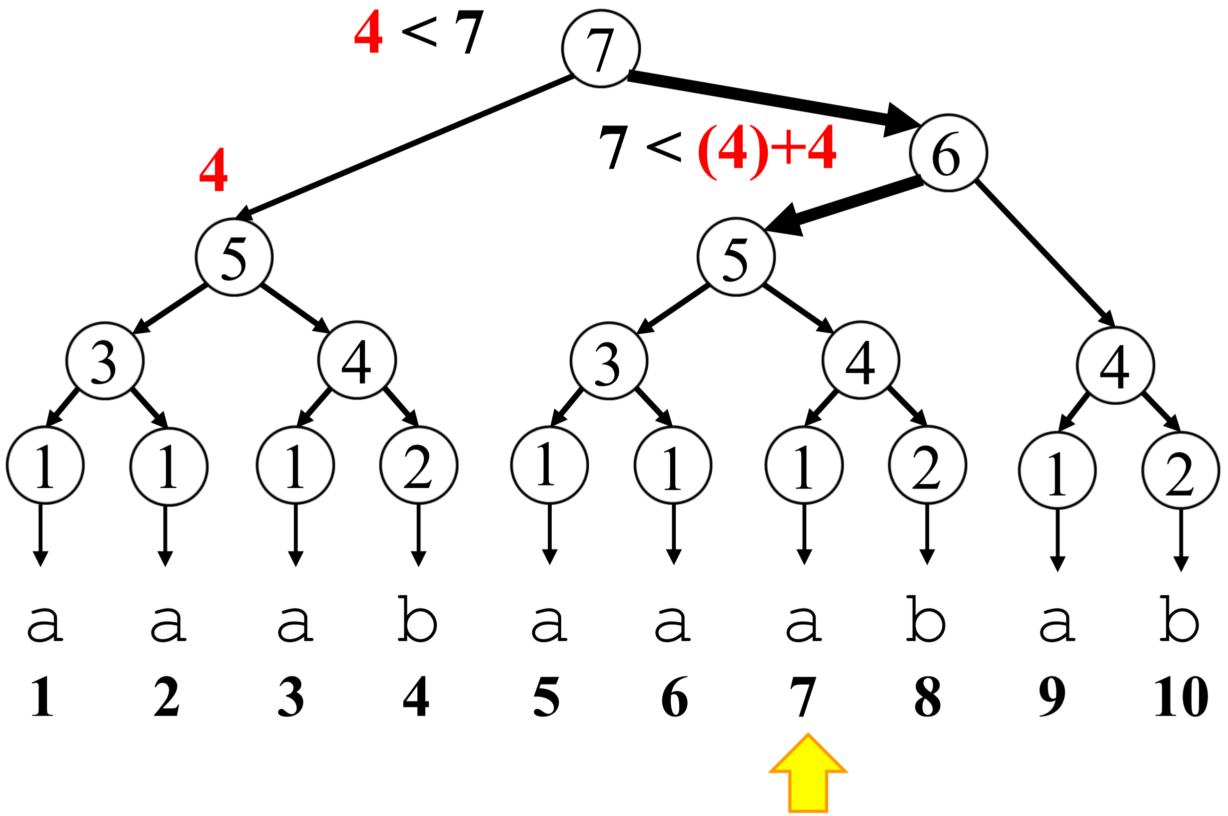
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



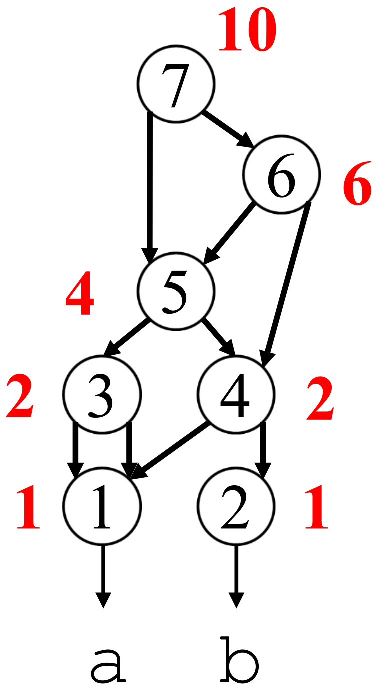
Derivation Tree



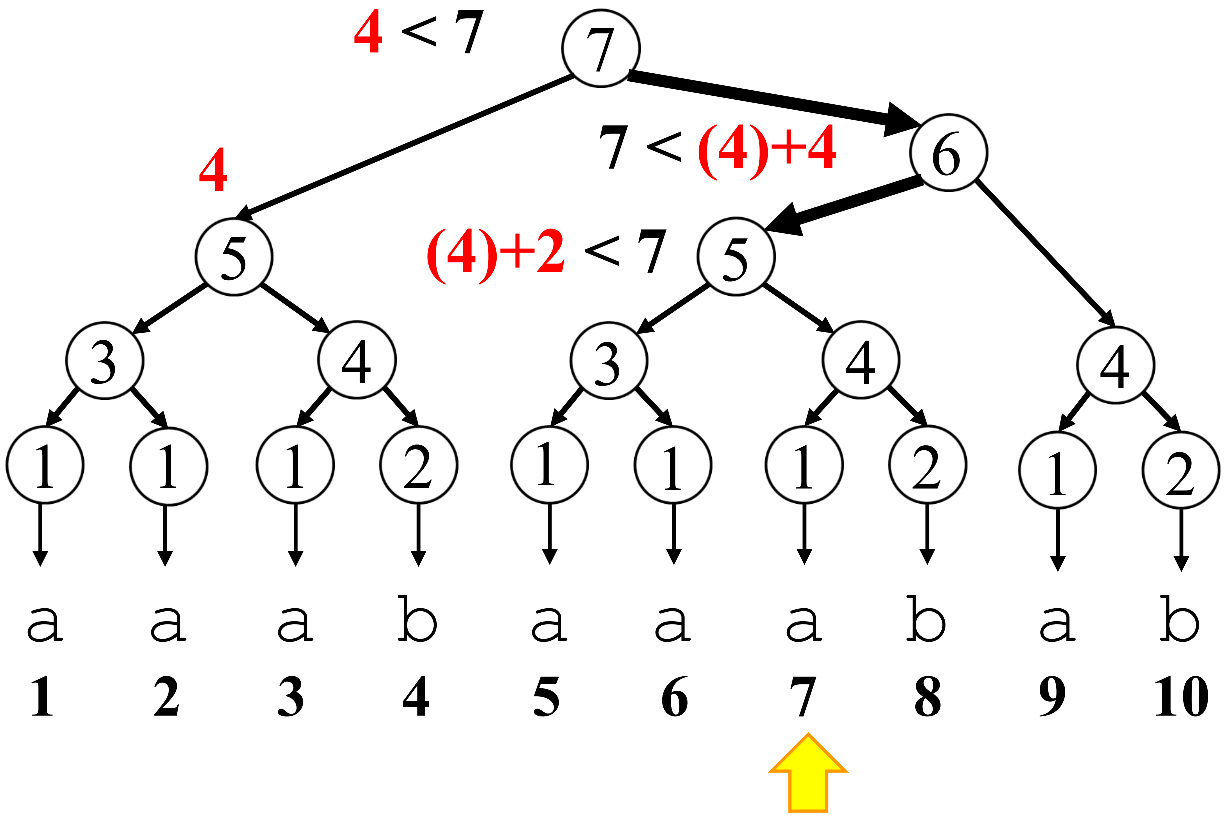
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



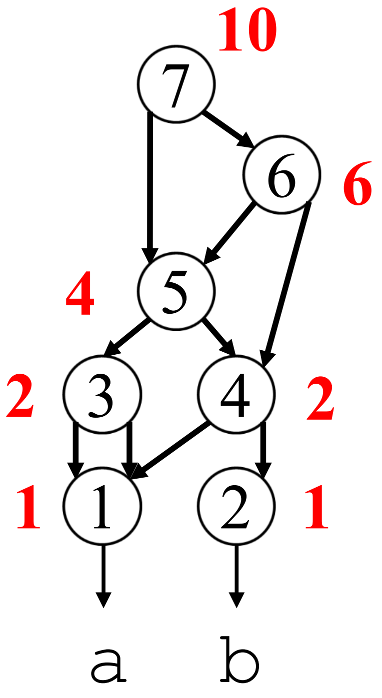
Derivation Tree



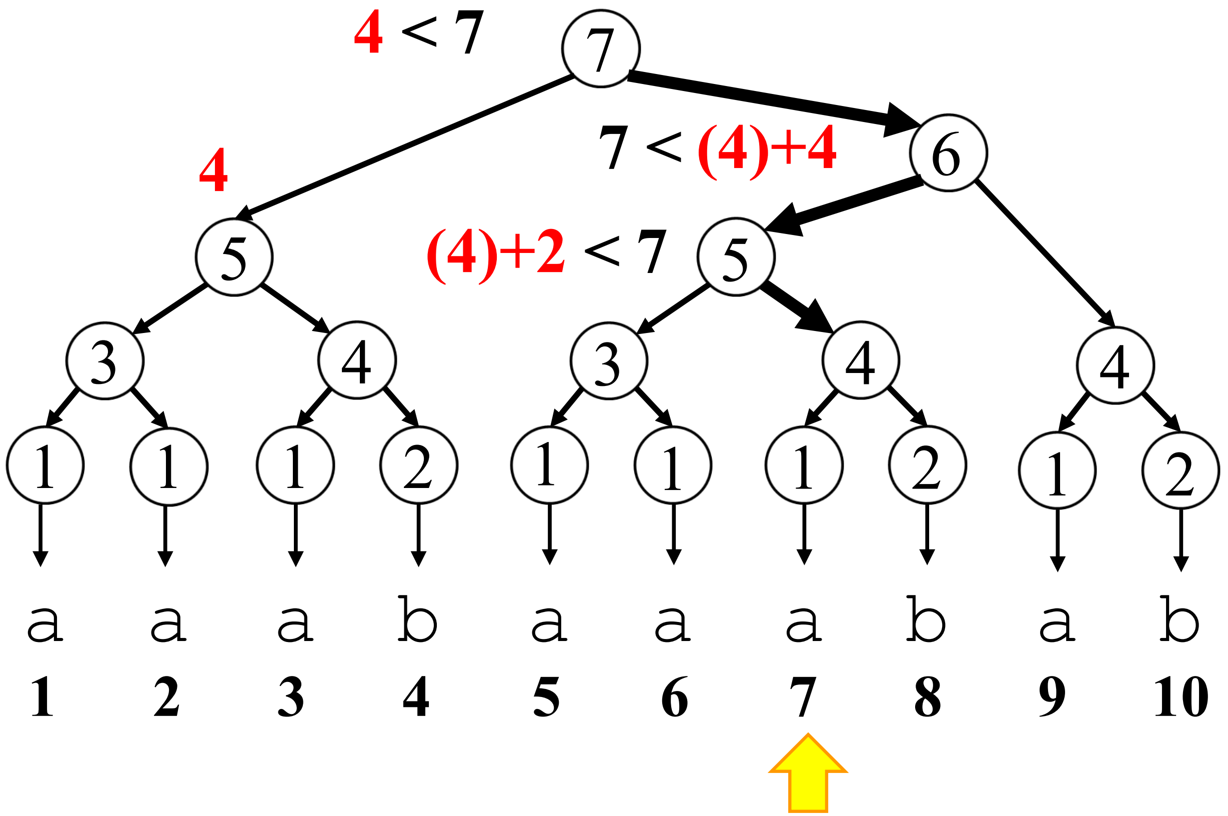
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP



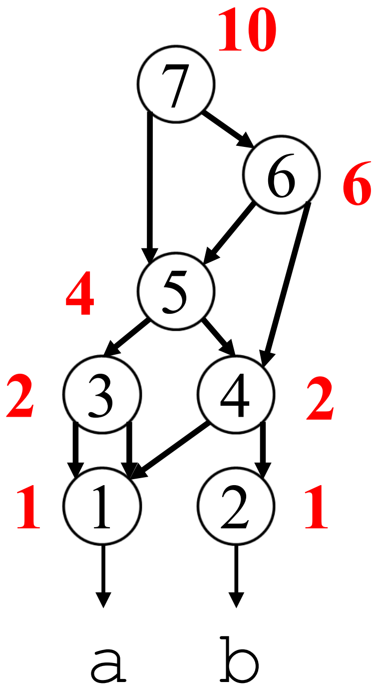
Derivation Tree



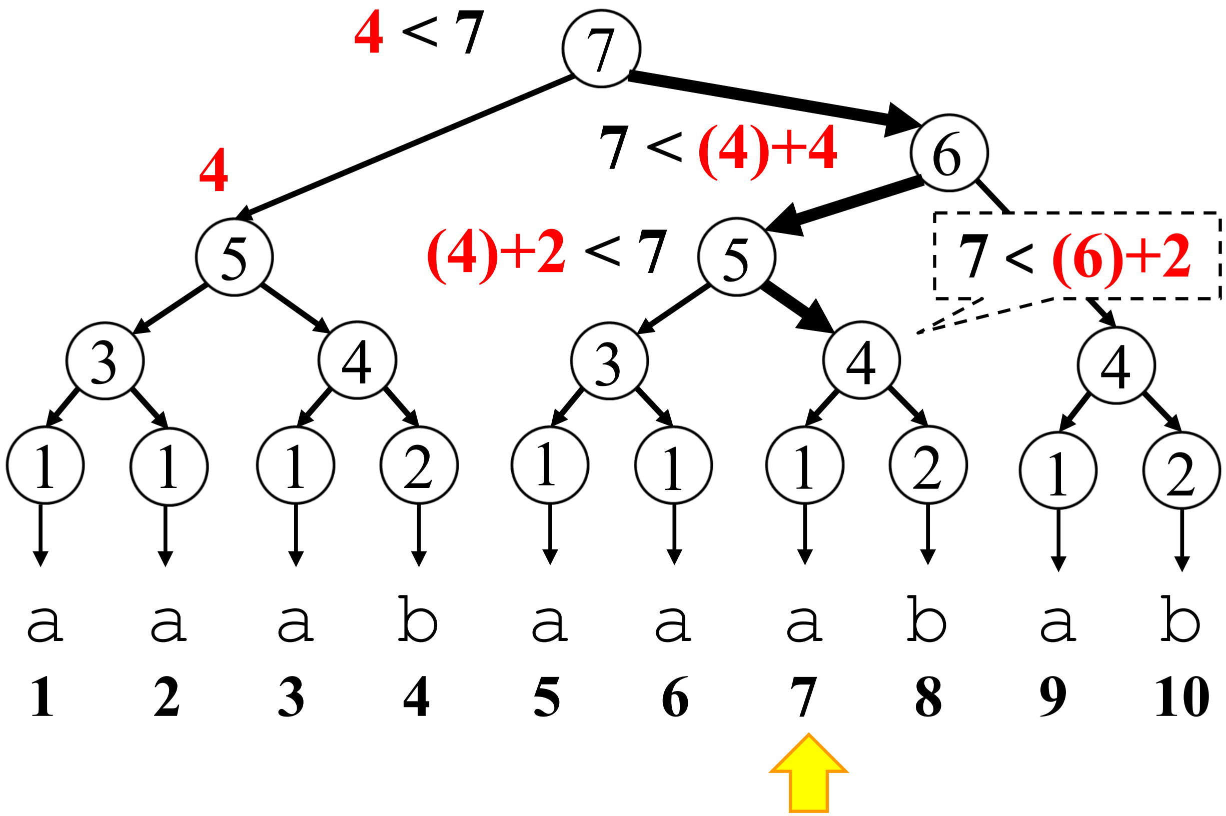
For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access

Balanced SLP

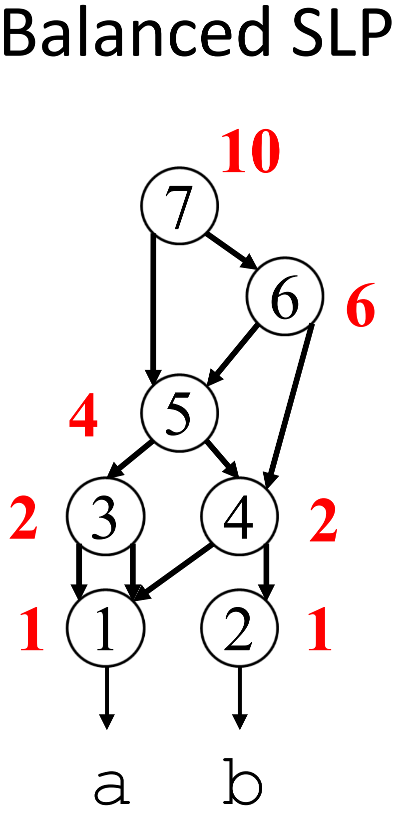


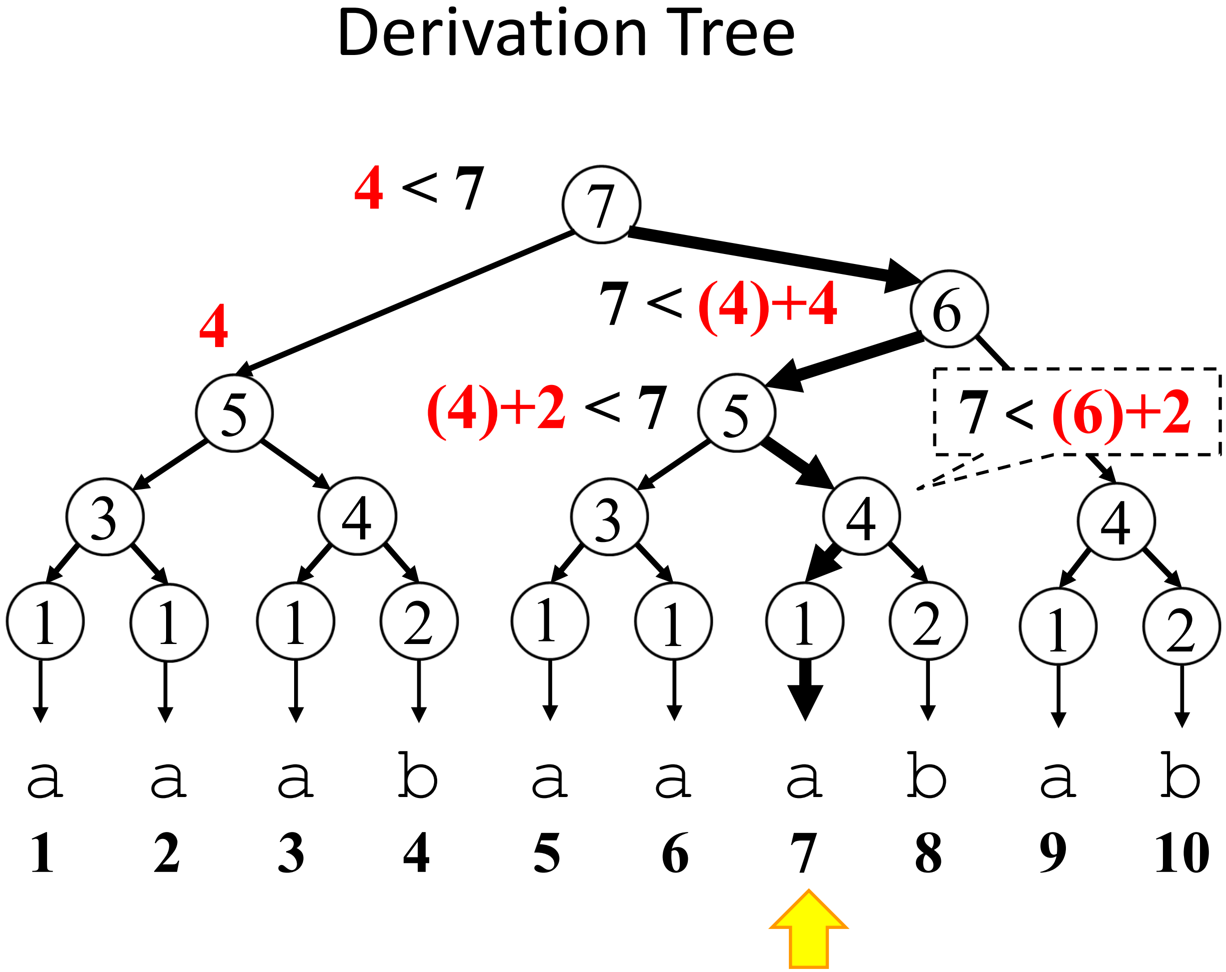
Derivation Tree



For a given position i in w, we can random access to the i-th char. in a top-down manner.

$O(\log N)$-time Random Access





For a given position i in w, we can random access to the i-th char. in a top-down manner.

Balancing SLPs [cont.]

- ☐ Recently, Ganardi et al. (FOCS 2019) showed how to transform a given SLP of size n into a balanced SLP of size $O(n)$.

➔ New $O(\log N)$-time random access and $O(m + \log N)$-time substring extraction algorithms with $O(n)$ space, which alternate Bille et al.'s previous algorithms.

➔ In addition, every h term in the time complexity for other operations on SLPs can be replaced with $\log N$.

String Attractors

String Attractors [Kempa & Prezza 2017]

A set $\Gamma \subseteq \{1, \ldots, N\}$ of positions in a string w of length N is called a *string attractor* of w, if any substring y of w has an occurrence $y = w[i..j]$ that contains an element k of Γ (i.e. $k \in [i..j]$).

String Attractors

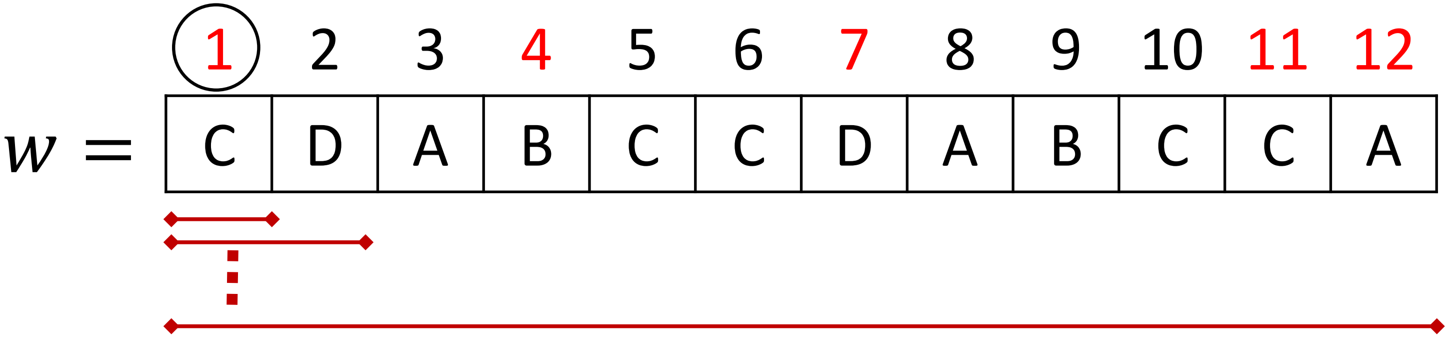
$\{1, 4, 7, 11, 12\}$

	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$



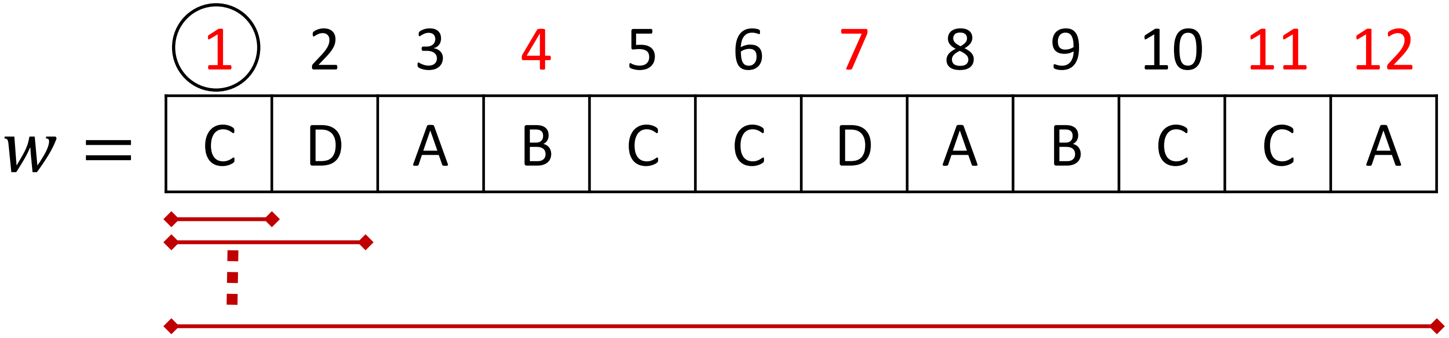
$w =$

1	2	3	4	5	6	7	8	9	10	11	12
C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$

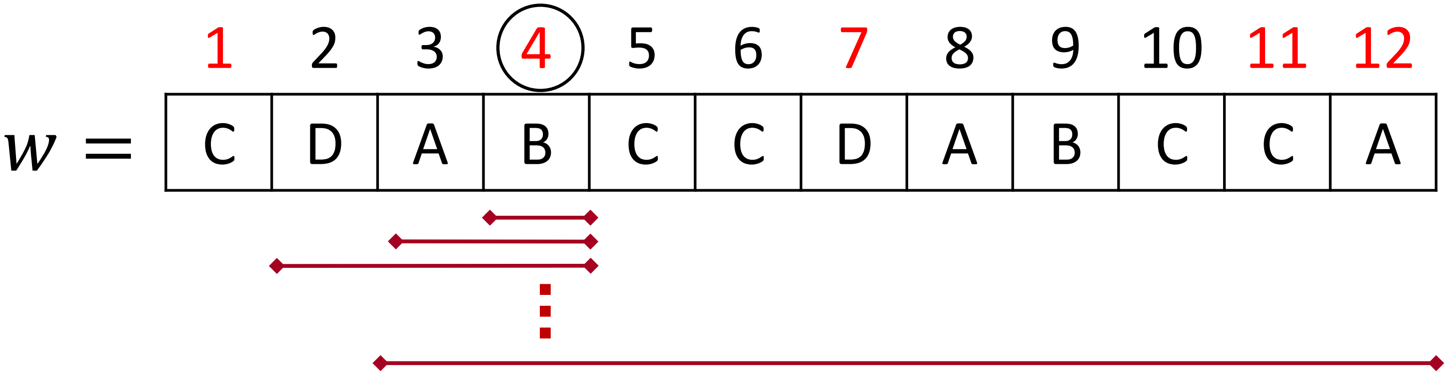


	①	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

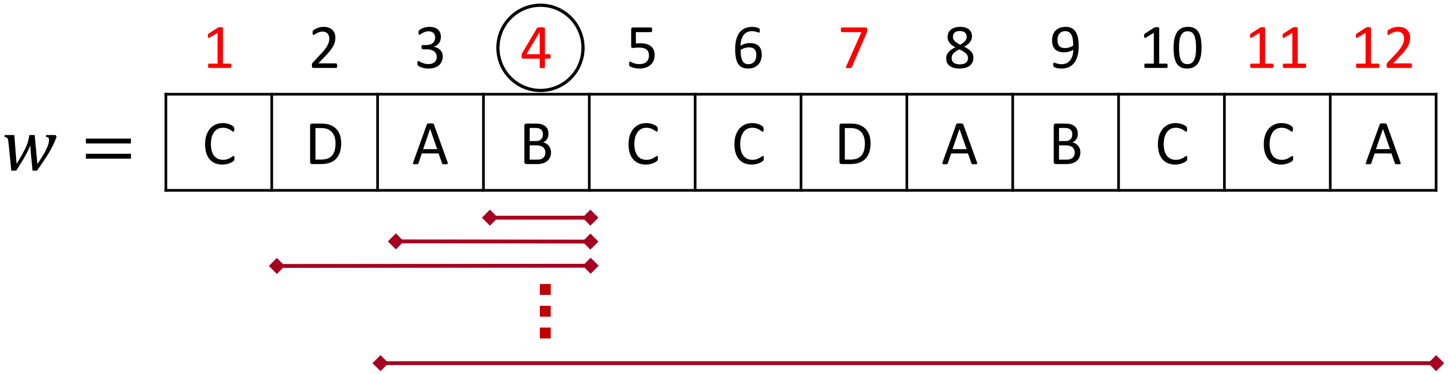
$\{1, 4, 7, 11, 12\}$



C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$



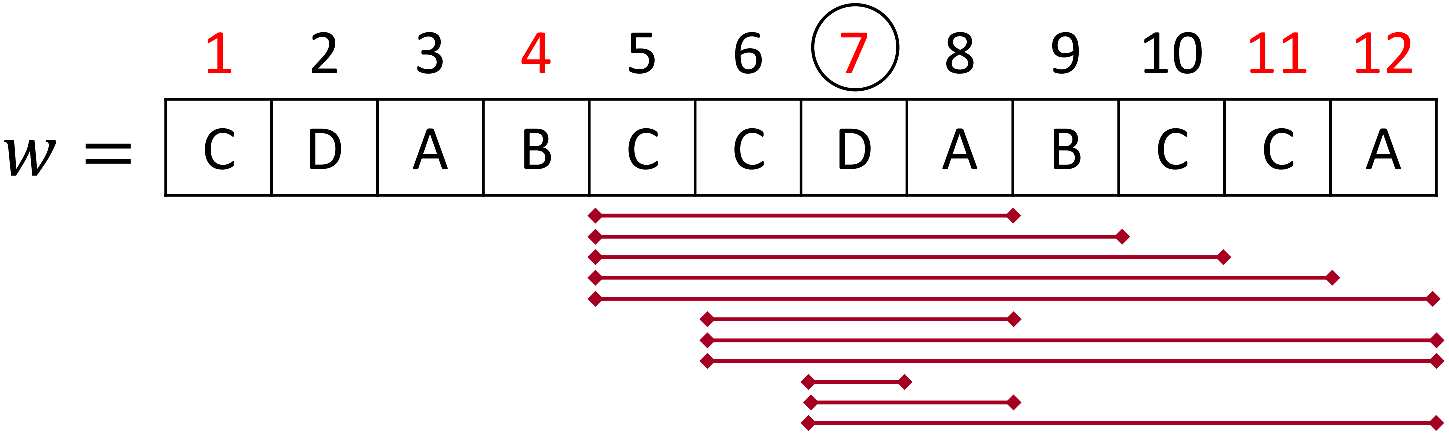
$w =$

1	2	3	4	5	6	7	8	9	10	11	12
C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$

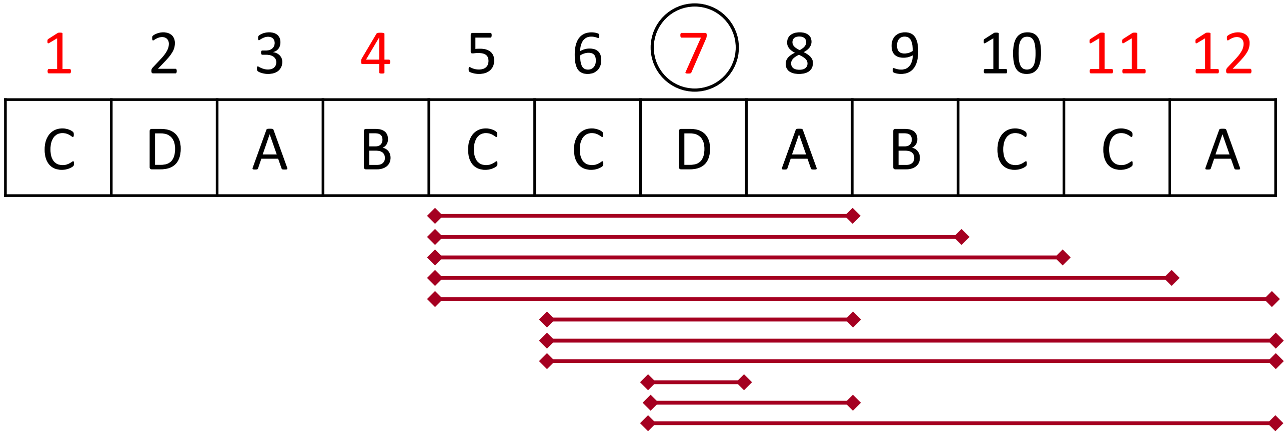


	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$



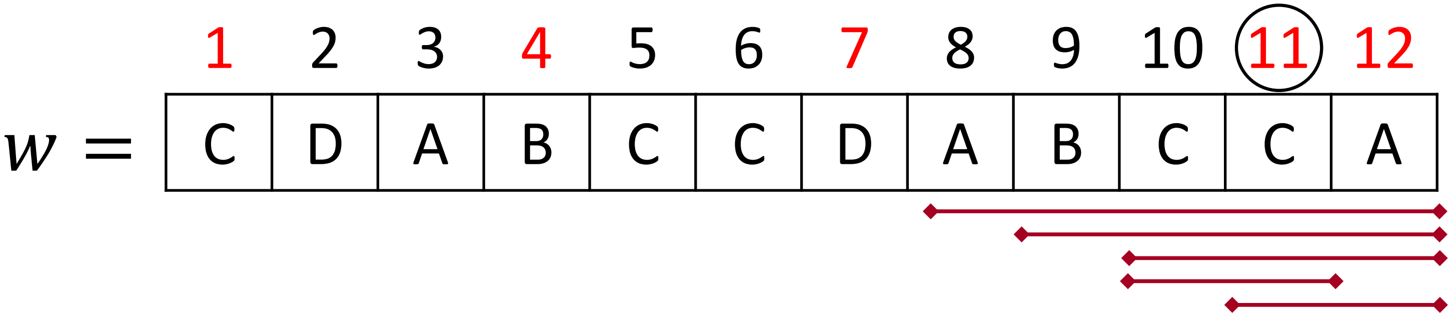
$w =$

1	2	3	4	5	6	7	8	9	10	11	12
C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$

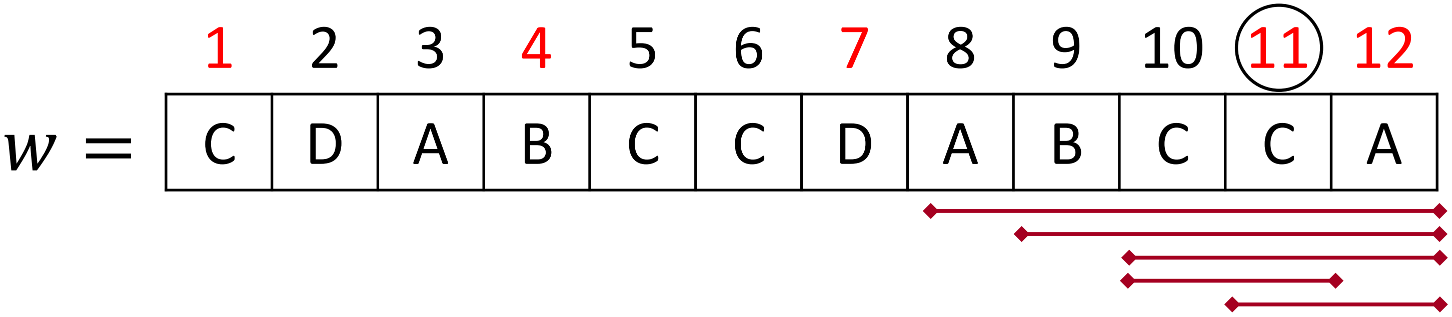


	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$



	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

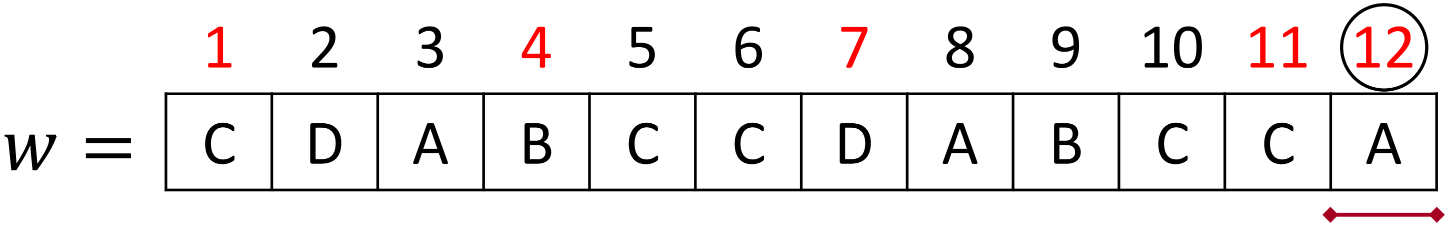
$\{1, 4, 7, 11, 12\}$

	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

$\{1, 4, 7, 11, 12\}$



	1	2	3	4	5	6	7	8	9	10	11	12
$w =$	C	D	A	B	C	C	D	A	B	C	C	A

C, D, A, B, CD, DA, AB, BC, CC, CA, CDA, DAB, ABC, BCC, CCD, CDA, CCA, CDAB, DABC, ABCC, BCCD, CCDA, BCCA, CDABC, DABCC, ABCCD, BCCDA, CCDAB, ABCCA, CDABCC, DABCCD, ABCCDA, BCCDAB, CCDABC, DABCCA, CDABCCD, DABCCDA, ABCCDAB, BCCDABC, CCDABCC, CDABCCA, CDABCCDA, DABCCDAB, ABCCDABC, BCCDABCC, CCDABCCA, CDABCCDAB, DABCCDABC, ABCCDABCC, BCCDABCCA, CDABCCDABC, DABCCDABCC, ABCCDACCA, CDABCCDABCC, DABCCDABCCA, CDABCCDABCCA

String Attractors

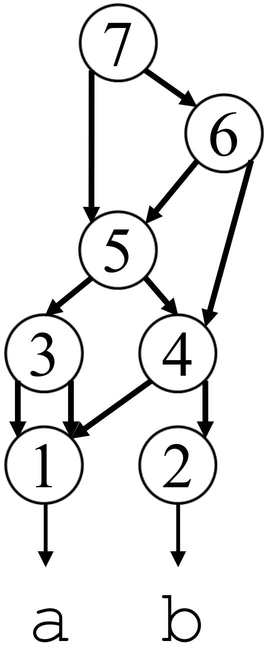
String Attractors [Kempa & Prezza 2017]

A set $\Gamma \subseteq \{1, \ldots, N\}$ of positions in a string w of length N is called a *string attractor* of w, if any substring y of w has an occurrence $y = w[i..j]$ that contains an element k of Γ (i.e. $k \in [i..j]$).

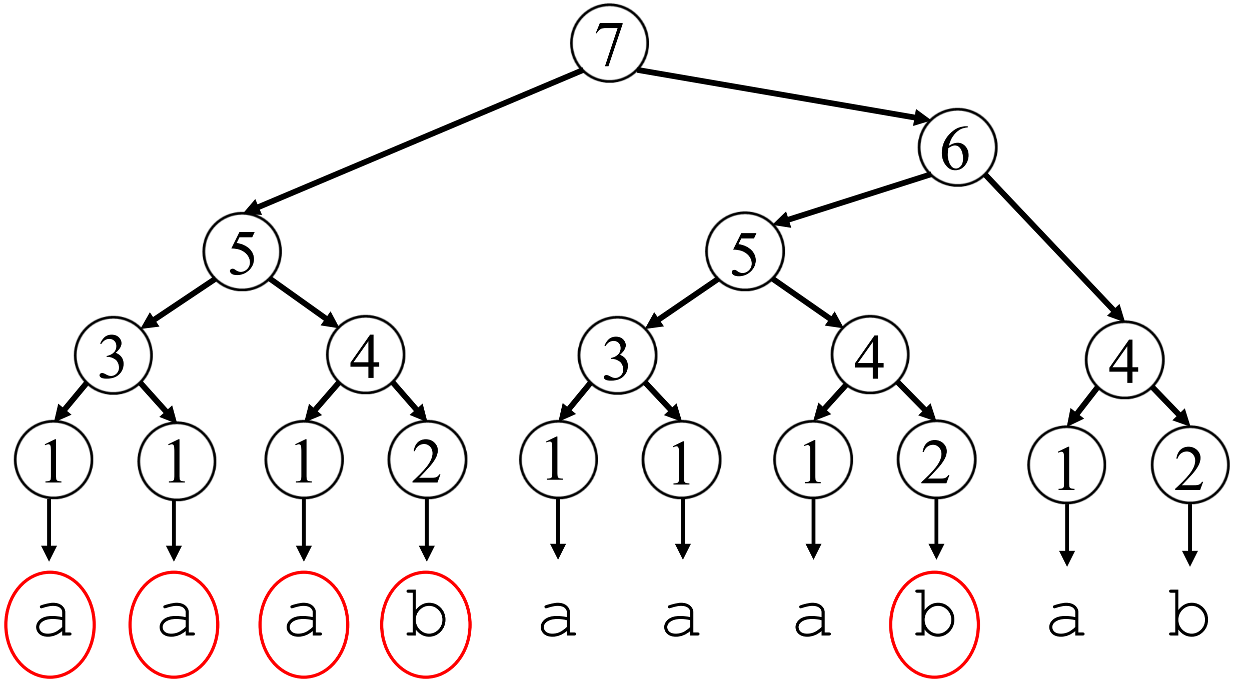
- ☐ Every string w has a string attractor since $\{1, \ldots, |w|\}$ is clearly a string attractor of w.
- ☐ String Attractors generalize to the notion of "stabbing" by SLP boundaries.

SLP as String Attractor

DAG for SLP S



Derivation tree T of SLP S



String Attractor as Lower Bound for SLP size

Theorem

Let γ = the smallest string attractor size,
z = # phrases in the LZ77 factorization,
g = the smallest SLP size, for the same string w.
Then, $\gamma \leq z \leq g$ holds.

- ✓ $\gamma \leq g$ because any substring of w must be stabbed by at least one variable of the SLP.
- ✓ $\gamma \leq z$ follows from similar arguments.
- ✓ $z \leq g$ was proved in the literature [Rytter 2003].

Indexing with String Attractor Space

Theorem [Ettienne et al. 2020 (arXiv)]

There exist compressed indexing structures that perform pattern matching queries in $O(m+(occ+1)\log^{\varepsilon} N)$ time with $O(\gamma \log(N/\gamma))$ space, or in $O(m+occ)$ time with $O(\gamma \log(N/\gamma) \log^{\varepsilon} N)$ space.

$\gamma =$ smallest attractor size; m = pattern length; occ = # pattern occurrences; $\varepsilon > 0$

- ✓ Computing the smallest string attractor is NP-hard [Kempa & Prezza 2017].
- ✓ However, these indexing structures can be built without knowing the smallest attractor size γ.

Conclusions and Future Work

- ☐ Grammar-based compression is a powerful compression scheme for highly repetitive strings.
- ☐ A variety of string processing can be performed directly on grammar-compressed strings.

- ◆ How can we close the gap for the upper and lower bounds of Re-Pair's approximation ratio to the smallest grammar?
- ◆ Can we perform various processing within space proportional to the smallest string attractor size γ?