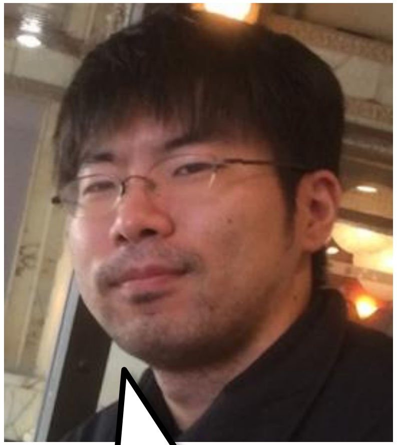
PSC 2015

# Faster Longest Common Extension on Compressed Strings and Applications

Shunsuke Inenaga
Kyushu University, Japan

# Collaborators

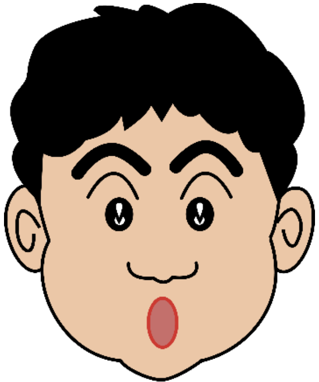
This work is a collaboration with:



Takaaki Nishimoto

Tomohiro I

Hideo Bannai



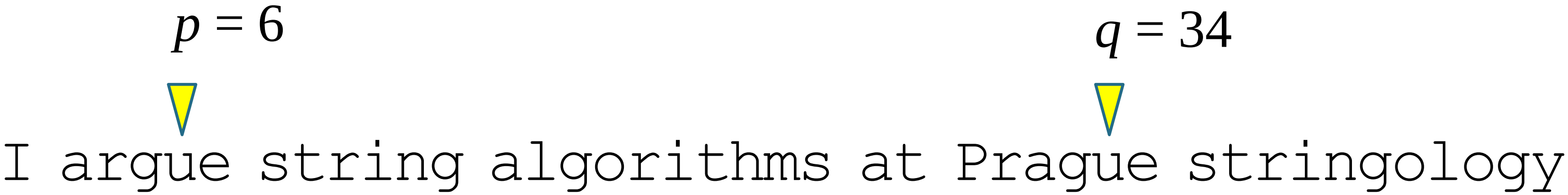
Masayuki Takeda

# Longest common extension (LCE)

**Longest common extension** (**LCE**) on string $T$ is a task such that, given two positions $p$ and $q$, compute the length of the longest common substring of $T$ starting at positions $p$ and $q$.

# Longest common extension (LCE)

**Longest common extension** (**LCE**) on string $T$ is a task such that, given two positions $p$ and $q$, compute the length of the longest common substring of $T$ starting at positions $p$ and $q$.



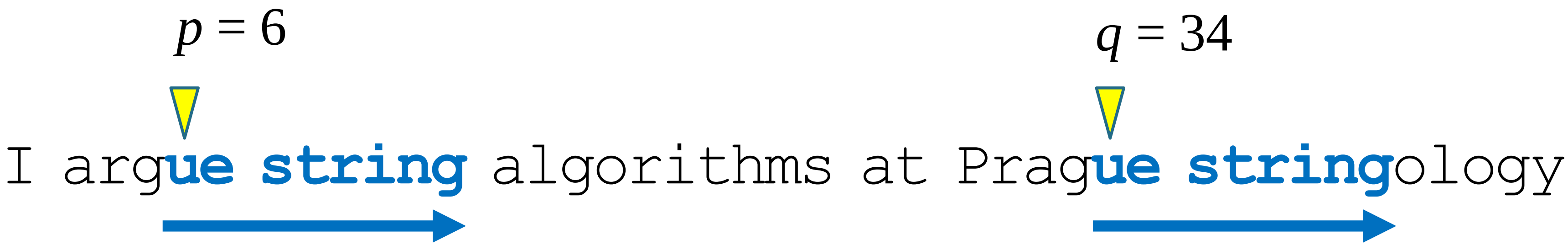
$p = 6$

$q = 34$

`I argue string algorithms at Prague stringology`

# Longest common extension (LCE)

**Longest common extension** (**LCE**) on string $T$ is a task such that, given two positions $p$ and $q$, compute the length of the longest common substring of $T$ starting at positions $p$ and $q$.



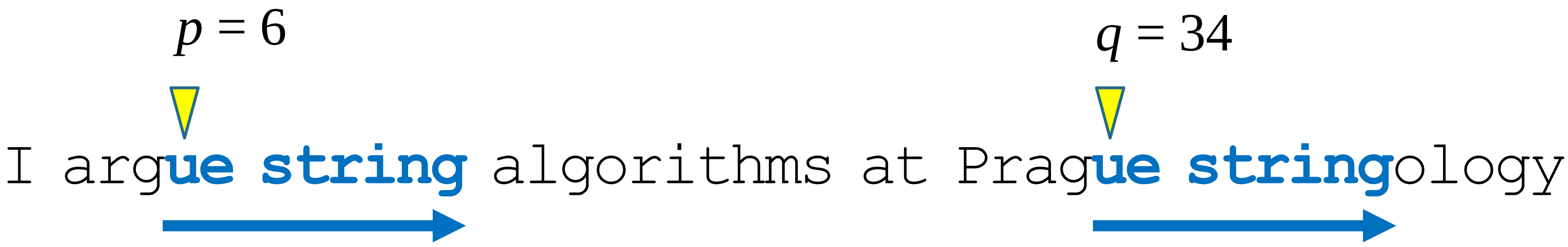
$p = 6$

$q = 34$

I arg**ue string** algorithms at Prag**ue string**ology

# Longest common extension (LCE)

**Longest common extension** (**LCE**) on string $T$ is a task such that, given two positions $p$ and $q$, compute the length of the longest common substring of $T$ starting at positions $p$ and $q$.



$p = 6$

$q = 34$

I arg**ue string** algorithms at Prag**ue string**ology

$$\mathbf{LCE}(6, 34) = 9$$

# Background & Motivation

- ✓ LCE has numerous applications, e.g., approximate pattern matching, computing palindromes, computing approximate repeats.

- ✓ A string $T$ of length $u$ can be preprocessed in $O(u)$ time and space so that each LCE query can be answered in $O(1)$ time [Demaine et al.].

- ✓ However, the $O(u)$ complexity can be prohibitive for large-scaled text.

- ✓ To save preprocessing time and space, we consider LCE on grammar-compressed text.

# Straight Line Program (SLP)

**Definition**

An SLP is a sequence of $n$ productions

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

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

✓ An SLP is a CFG in the Chomsky normal form which derives a single string.

✓ SLPs model outputs of grammar-based compression algorithms (e.g., Re-pair, LZ78, LZDF, OLCA, etc).

# Straight Line Program (SLP)

$n$ : size (# of productions) of a given SLP $S$

$h$ : height of the derivation tree of $S$

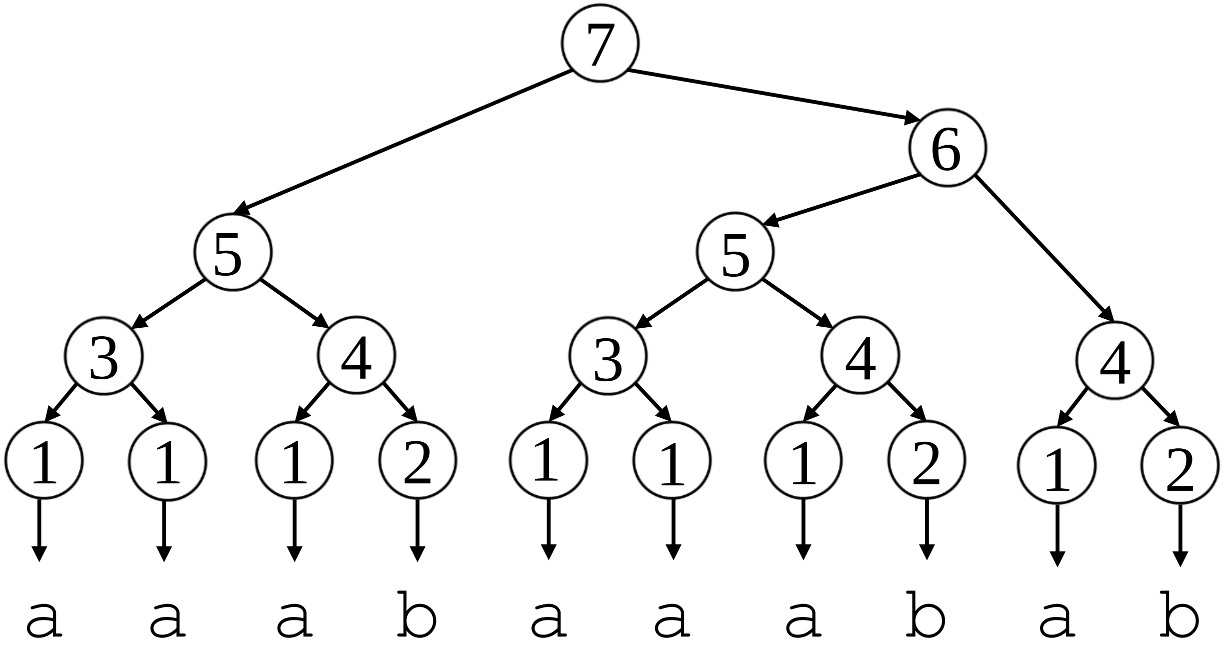
$u$ : length of the uncompressed string $T$ represented by SLP $S$

# Example of SLP

SLP $S$

$X_1 \rightarrow$ a
$X_2 \rightarrow$ b
$X_3 \rightarrow X_1\ X_1$
$X_4 \rightarrow X_1\ X_2$
$X_5 \rightarrow X_3\ X_4$
$X_6 \rightarrow X_5\ X_4$
$X_7 \rightarrow X_5\ X_6$

Derivation tree of SLP $S$



# Example of SLP

## SLP $S$

$X_1 \rightarrow$ a

$X_2 \rightarrow$ b

$X_3 \rightarrow X_1\, X_1$

$X_4 \rightarrow X_1\, X_2$

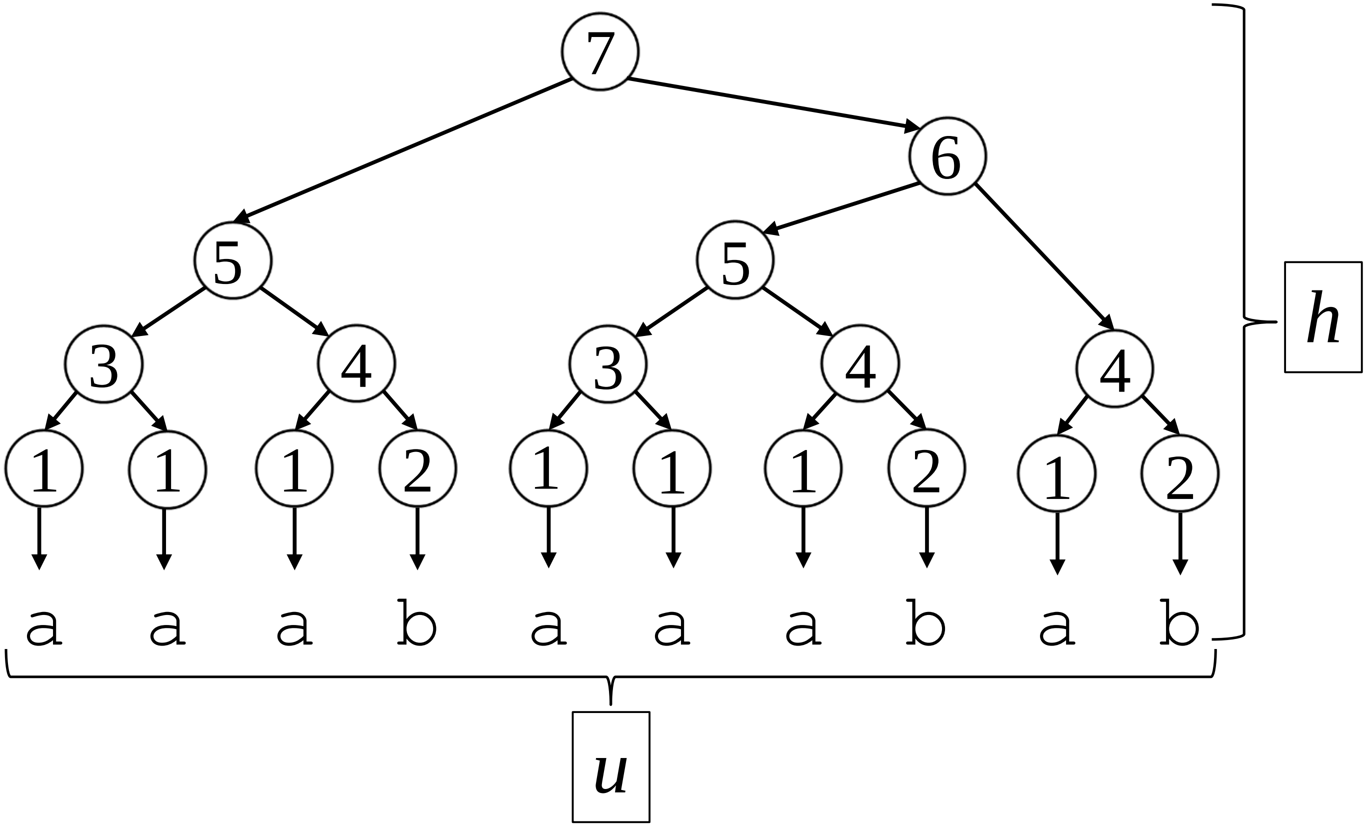
$X_5 \rightarrow X_3\, X_4$

$X_6 \rightarrow X_5\, X_4$

$X_7 \rightarrow X_5\, X_6$

$n$

## Derivation tree of SLP $S$



# Example of SLP

SLP $S$

$X_1 \rightarrow$ a

$X_2 \rightarrow$ b

$X_3 \rightarrow X_1 X_1$

$X_4 \rightarrow X_1 X_2$

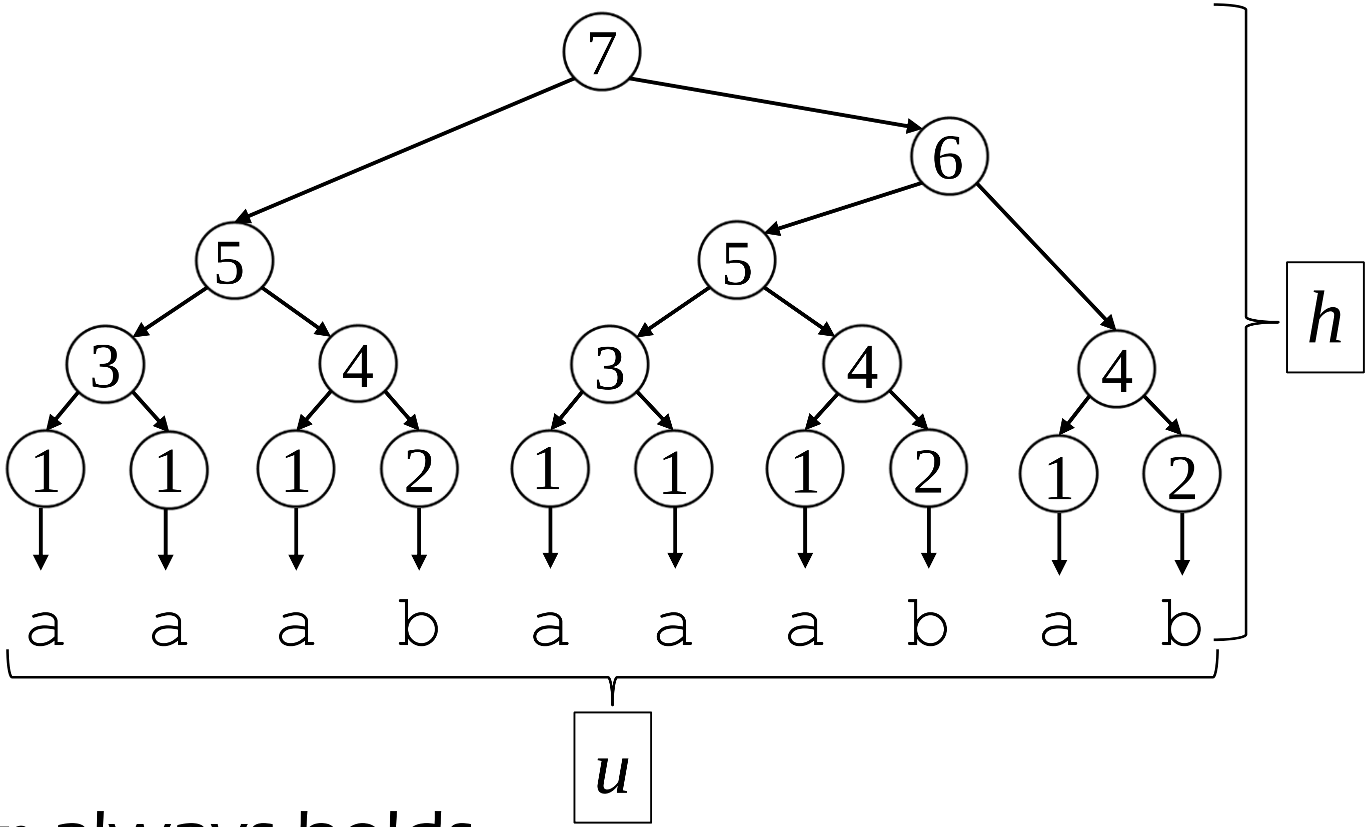
$X_5 \rightarrow X_3 X_4$

$X_6 \rightarrow X_5 X_4$

$X_7 \rightarrow X_5 X_6$

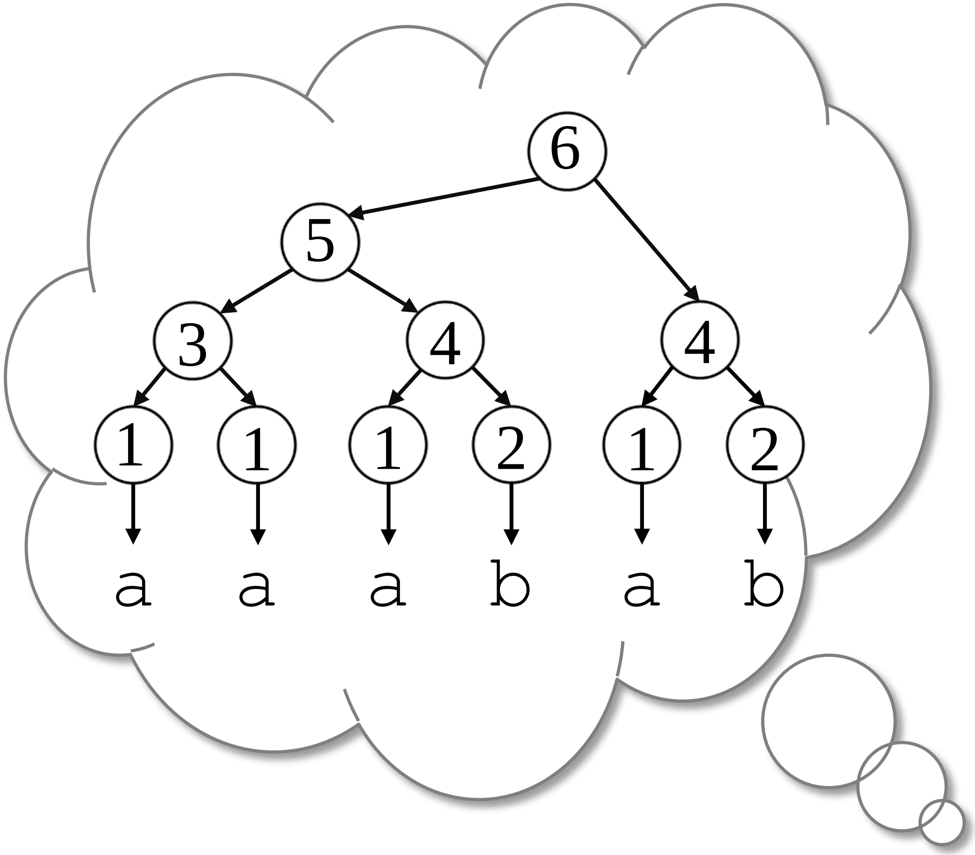
$n$

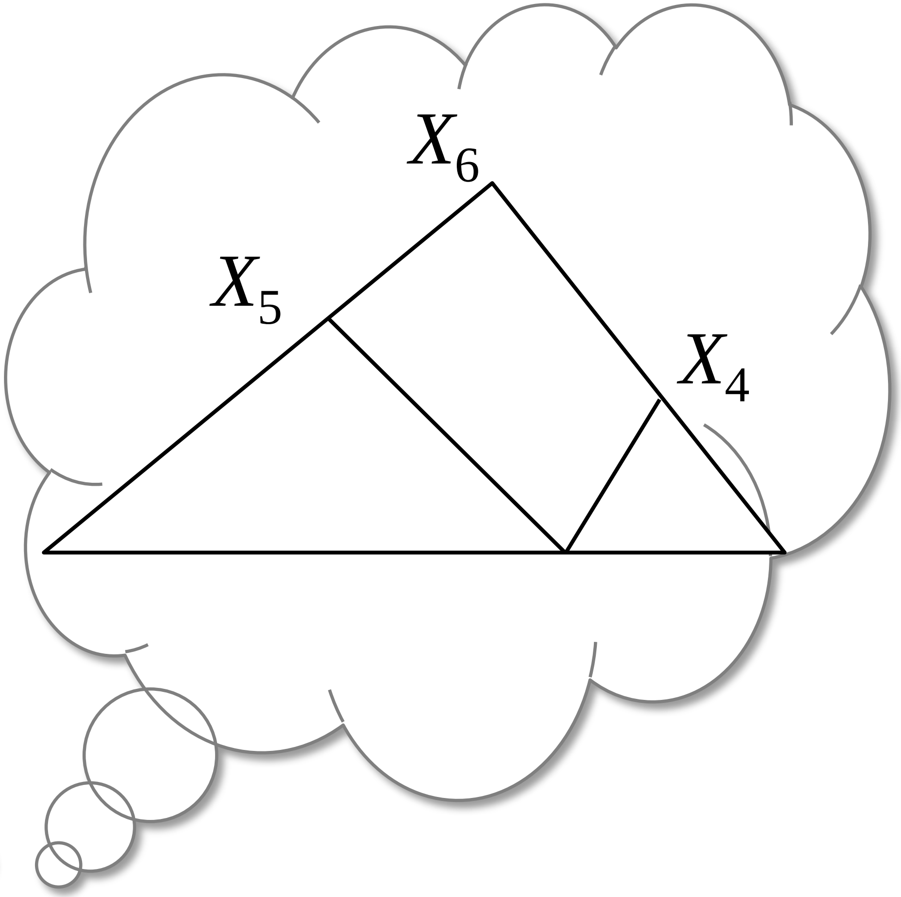
Derivation tree of SLP $S$



- ✓ $\log_2 u \leq h \leq n$ always holds.
- ✓ $u$ can be exponential in $n$ (e.g. consider string $a^u$).
  - ➢ Hence, $O(\mathrm{poly}(n))$ solutions are of significance.

# Important Remarks



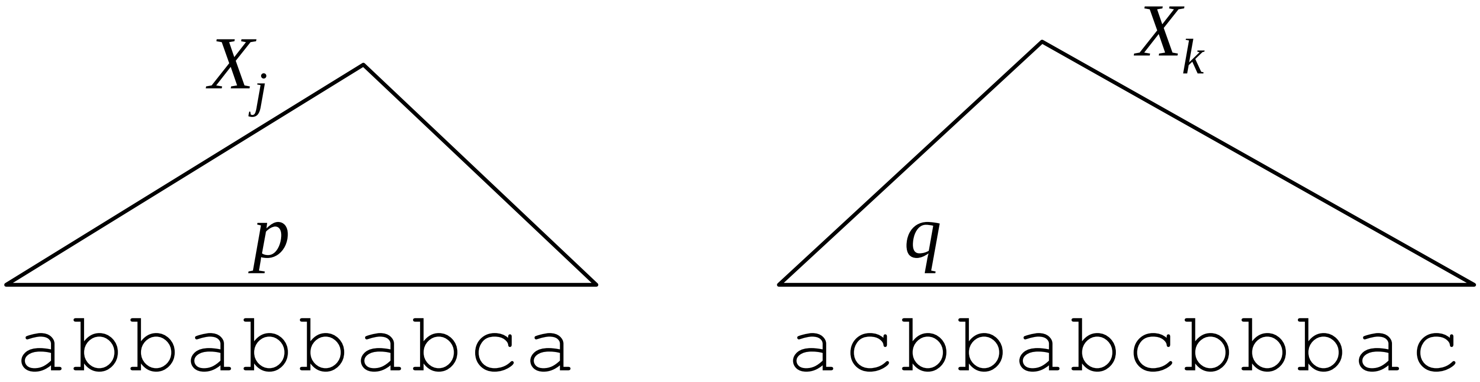


- ✓ Derivation trees are only **imaginary** (used only for explanations) and are never constructed explicitly.

# Longest Common Extension on SLP

Problem 1 (grammar compressed LCE)

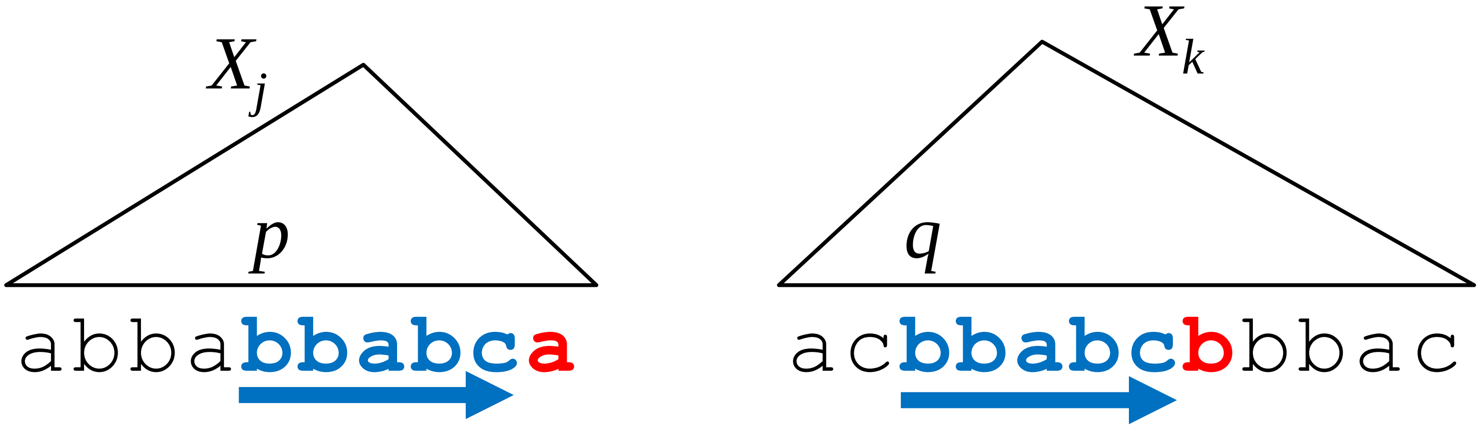
Preprocess an input SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ so that subsequent longest common extension queries $\mathbf{LCE}(X_j, X_k, p, q)$ can be answered quickly.



# Longest Common Extension on SLP

**Problem 1 (grammar compressed LCE)**

Preprocess an input SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ so that subsequent longest common extension queries $\mathbf{LCE}(X_j, X_k, p, q)$ can be answered quickly.



Query output is LCE length 5

# What is the difficulty?

- ✓ We are not allowed to expand the SLP (compressed text), since this takes $O(2^n)$ time in the worst case.
- ✓ But we want to know the length of the longest common extension!

# LCE algorithms on SLPs

| Algorithms | Query time | Preprocessing time | Space |
|---|---|---|---|
| Folklore | $O(hL)$ | $O(n)$ | $O(n)$ |
| (extended)<br>Miyazaki et al. '97 | $O(hn^2)$ | $O(n^4)$ | $O(n^2)$ |
| (extended)<br>Lifshits '07 | $O(hn^2)$ | $O(hn^2)$ | $O(n^2)$ |
| I et al. '15 | $O(h \log u)$ | $O(hn^2)$ | $O(n^2)$ |
| Bille et al. '15<br>(randomized) | $O(\log u + \log^2 L)$ | N/A | $O(n)$ |

$n$: size of SLP
$u$: length of uncompressed string $T$
$h$: height of SLP derivation tree
$L$: LCE length (output)
$z$: size of LZ77 factorization of $T$

- $\log u \leq h \leq n$
- $L = O(u)$
- $\log^* u = o(\log u)$
- $z \leq n$ (due to Rytter '03)

# LCE algorithms on SLPs

| Algorithms | Query time | Preprocessing time | Space |
|---|---|---|---|
| Folklore | $O(hL)$ | $O(n)$ | $O(n)$ |
| (extended)<br>Miyazaki et al. '97 | $O(hn^2)$ | $O(n^4)$ | $O(n^2)$ |
| (extended)<br>Lifshits '07 | $O(hn^2)$ | $O(hn^2)$ | $O(n^2)$ |
| I et al. '15 | $O(h \log u)$ | $O(hn^2)$ | $O(n^2)$ |
| Bille et al. '15<br>(randomized) | $O(\log u + \log^2 L)$ | N/A | $O(n)$ |
| This work | $O(\log u + \log^* u \log L)$ | $O(n \log\log n \log^* u \log u)$ | $O(n + z \log^* u \log u)$ |

$n$: size of SLP

$u$: length of uncompressed string $T$

$h$: height of SLP derivation tree

$L$: LCE length (output)

$z$: size of LZ77 factorization of $T$

- $\log u \leq h \leq n$
- $L = O(u)$
- $\log^* u = o(\log u)$
- $z \leq n$ (due to Rytter '03)

# Logstar (iterated logarithm)

Definition

The **logstar** of a positive integer $u$, denoted $\log^* u$, is the number of times the logarithm function needs to be iteratively applied to $u$ until the result becomes less than or equal to $1$.

✓ The logstar is a very slowly growing function, e.g., $\log^* 2^{65536} = 5$.

# LCE algorithms on SLPs

| Algorithms | Query time | Preprocessing time | Space |
|---|---|---|---|
| Folklore | $O(hL)$ | $O(n)$ | $O(n)$ |
| (extended)<br>Miyazaki et al. '97 | $O(hn^2)$ | $O(n^4)$ | $O(n^2)$ |
| (extended)<br>Lifshits '07 | $O(hn^2)$ | $O(hn^2)$ | $O(n^2)$ |
| I et al. '15 | $O(h \log u)$ | $O(hn^2)$ | $O(n^2)$ |
| Bille et al. '15<br>(randomized) | $O(\log u + \log^2 L)$ | N/A | $O(n)$ |
| This work | $O(\log u + \log^* u \log L)$ | $O(n \log\log n \log^* u \log u)$ | $O(n + z\log^* u \log u)$ |

Fastest deterministic queries

Fastest preprocessing

Smallest in many cases

$n$: size of SLP

$u$: essed strin

on

$z$: si ization of $T$

- $\log u \le h \le$

  $= O($

  $g^* u$

- $z \le n$ (due to

# Our strategy

- ✓ All previous algorithms work on the SLP derivation trees of two query non-terminals.

- ✓ Our new algorithm does NOT work on the SLP derivation trees.

- ✓ Instead, we construct a different tree of logarithmic height, based on
  - ➢ locally consistent parsing
  - ➢ signature encoding.

# Locally consistent parsing

Lemma 1 [Mehlhorn et al., Alstrup et al.]

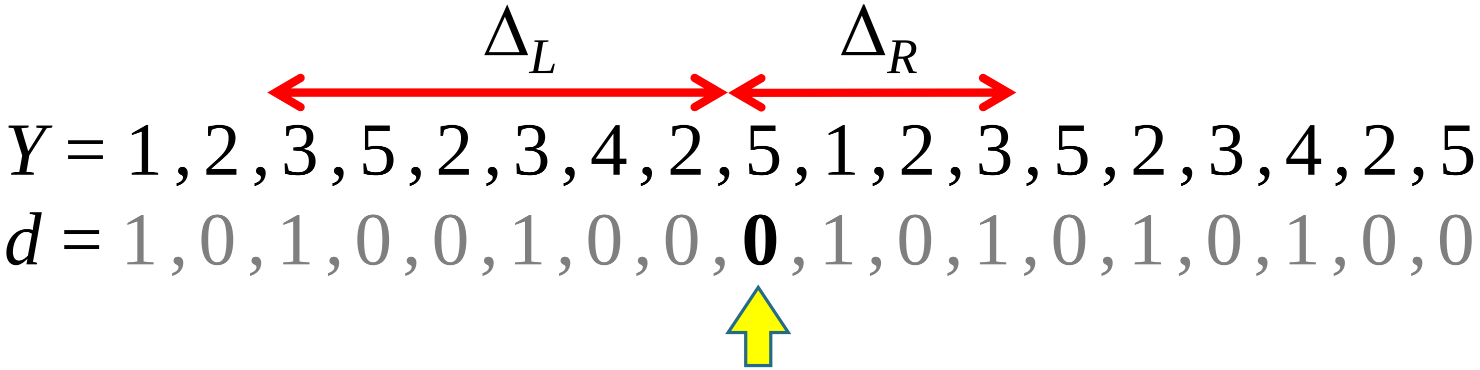
For any integer string $Y \in \{1..m\}^*$ in which no adjacent elements are equal (i.e. $Y[i] \neq Y[i+1]$ ), there is a bit string $d$ of length $|Y|$ such that

1. no $1$'s appear consecutively;
2. at most three $0$'s appear consecutively;
3. each $d[i]$ is determined locally, i.e., by $Y[i-\Delta_L \ldots i-1]$ and $Y[i \ldots i+\Delta_R]$, where $\Delta_L \leq \log^* m + 6$ and $\Delta_R \leq 4$;
4. $d$ can be computed in $O(|Y|)$ time.

# Locally consistent parsing

$$Y = 1,2,3,5,2,3,4,2,5,1,2,3,5,2,3,4,2,5$$
$$d = 1,0,1,0,0,1,0,0,0,1,0,1,0,1,0,1,0,0$$

# Locally consistent parsing



$\Delta_L$ $\Delta_R$

$$Y = 1,2,3,5,2,3,4,2,5,1,2,3,5,2,3,4,2,5$$

$$d = 1,0,1,0,0,1,0,0,\mathbf{0},1,0,1,0,1,0,1,0,0$$

$$\Delta_L \leq \log^* m + 6$$

$$\Delta_R \leq 4$$

# Locally consistent parsing

$$Y = \boxed{1,2}, \boxed{3,5,2}, \boxed{3,4,2,5}, \boxed{1,2}, \boxed{3,5}, \boxed{2,3}, \boxed{4,2,5}$$

$$d = 1,0,1,0,0,1,0,0,0,1,0,1,0,1,0,1,0,0$$

✓ Using the bit string $d$, any integer string $Y$ can be uniquely decomposed in linear time into blocks of length 2-4.

# Signature encoding [Mehlhorn et al. '97]

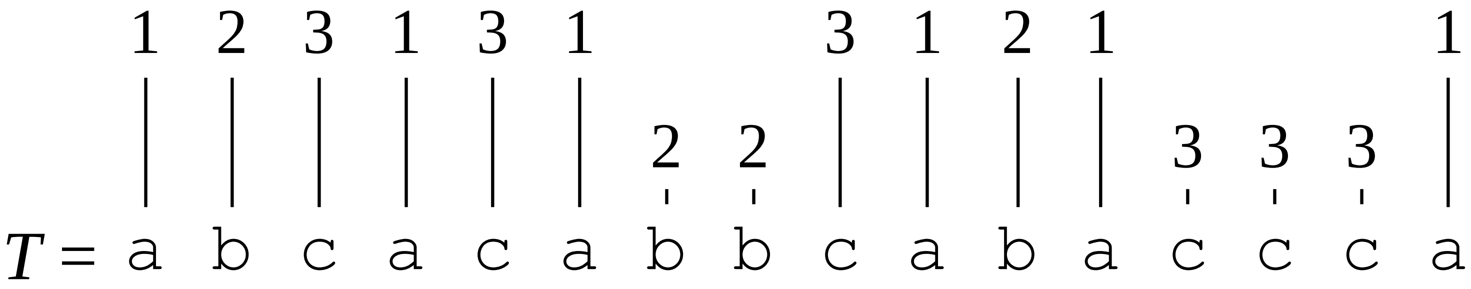
- ✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

$T =$ a b c a c a b b c a b a c c c a

# Signature encoding [Mehlhorn et al. '97]

✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

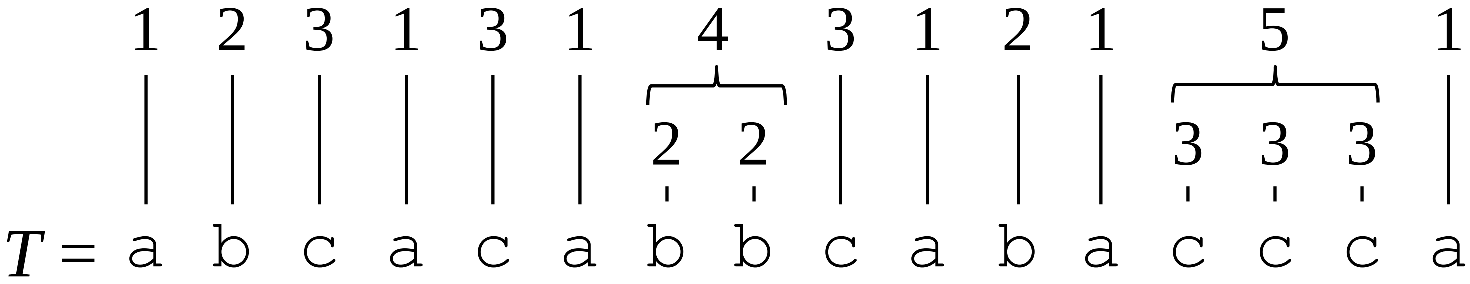
Each character is assigned to a unique integer called a signature.



# Signature encoding [Mehlhorn et al. '97]

✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

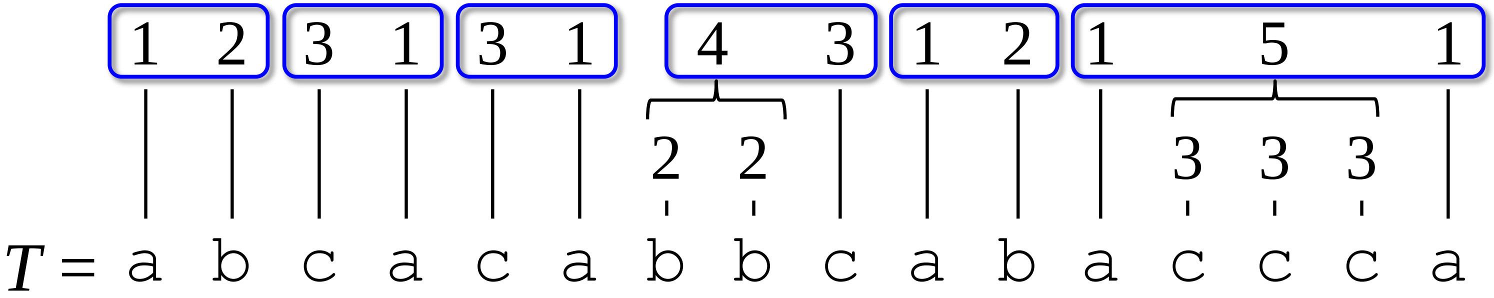
Maximal run of the same signatures is assigned to a new signature.



# Signature encoding [Mehlhorn et al. '97]

- ✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

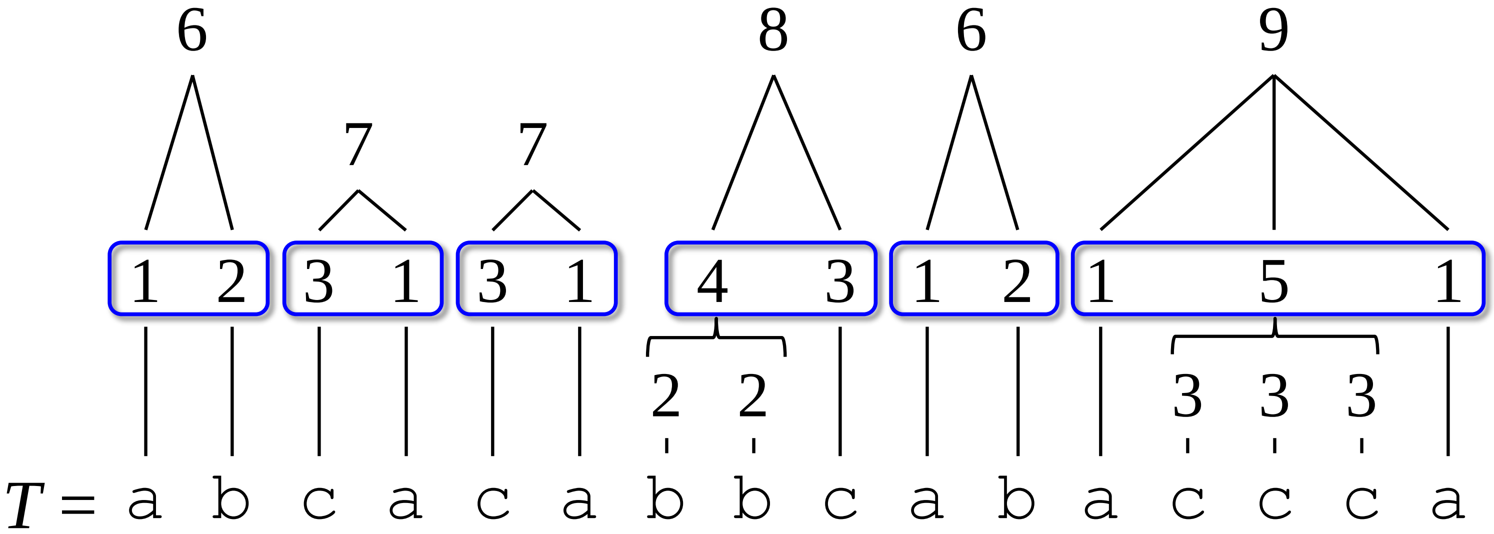
Apply locally consistent parsing to this string.



# Signature encoding [Mehlhorn et al. '97]

- ✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

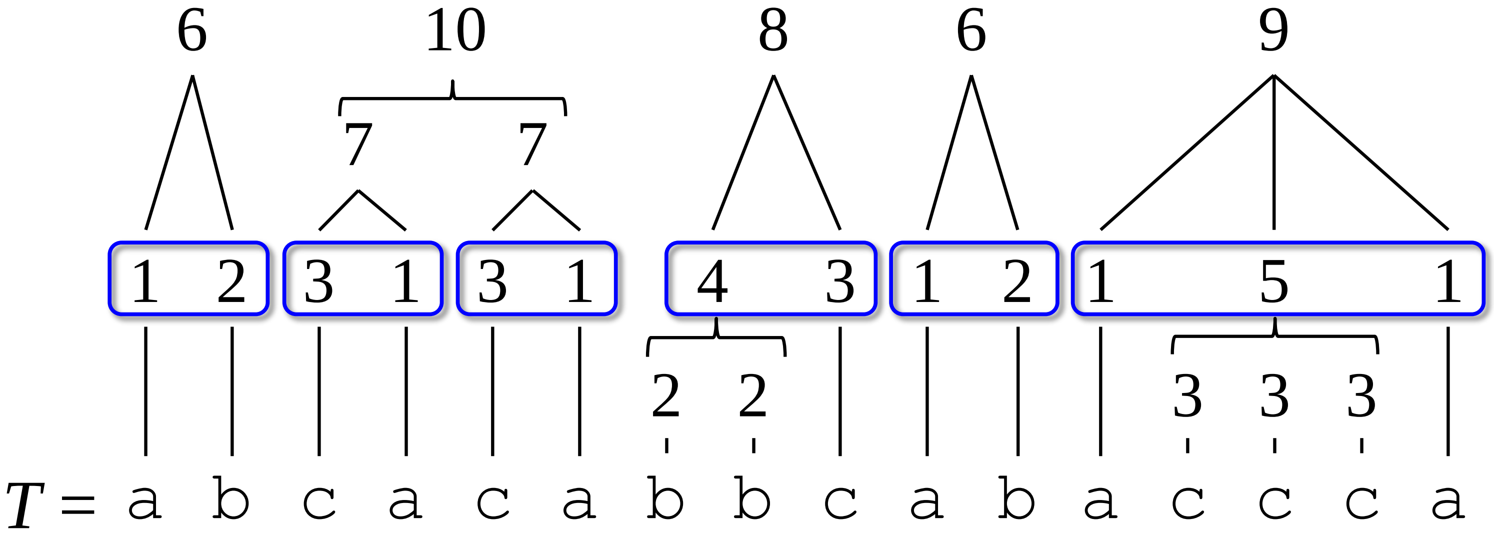
Each block is assigned to a new signature.



# Signature encoding [Mehlhorn et al. '97]

✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

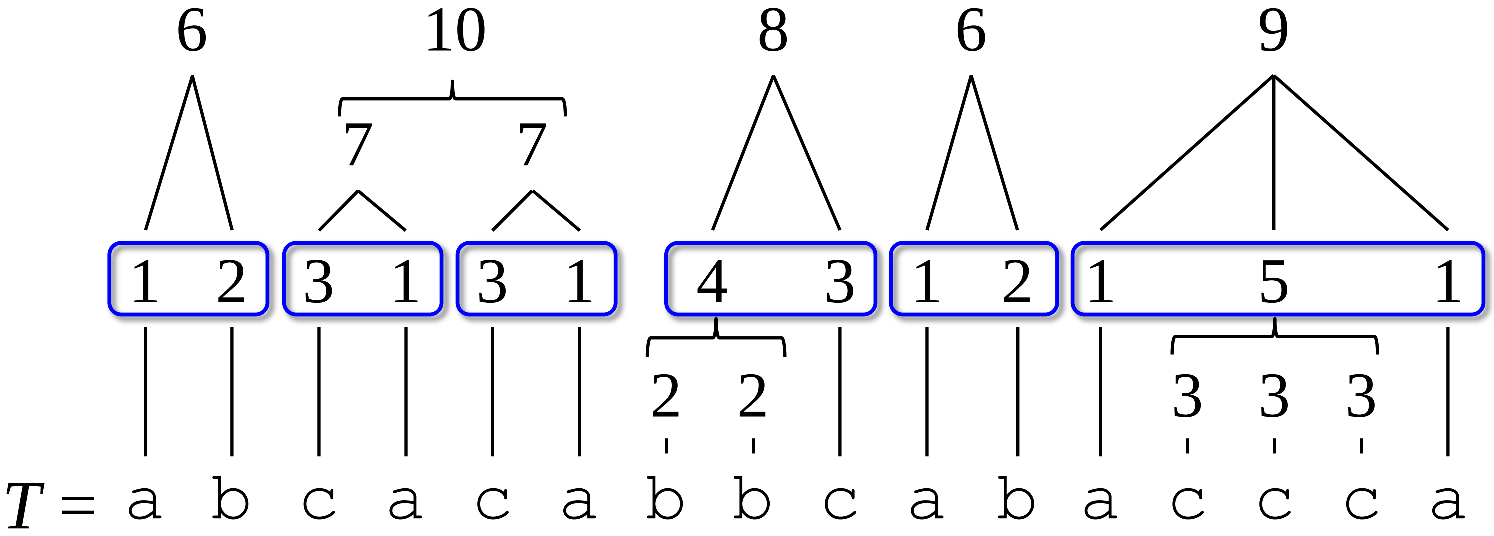
Maximal run of the same signatures is assigned to a new signature.



# Signature encoding [Mehlhorn et al. '97]

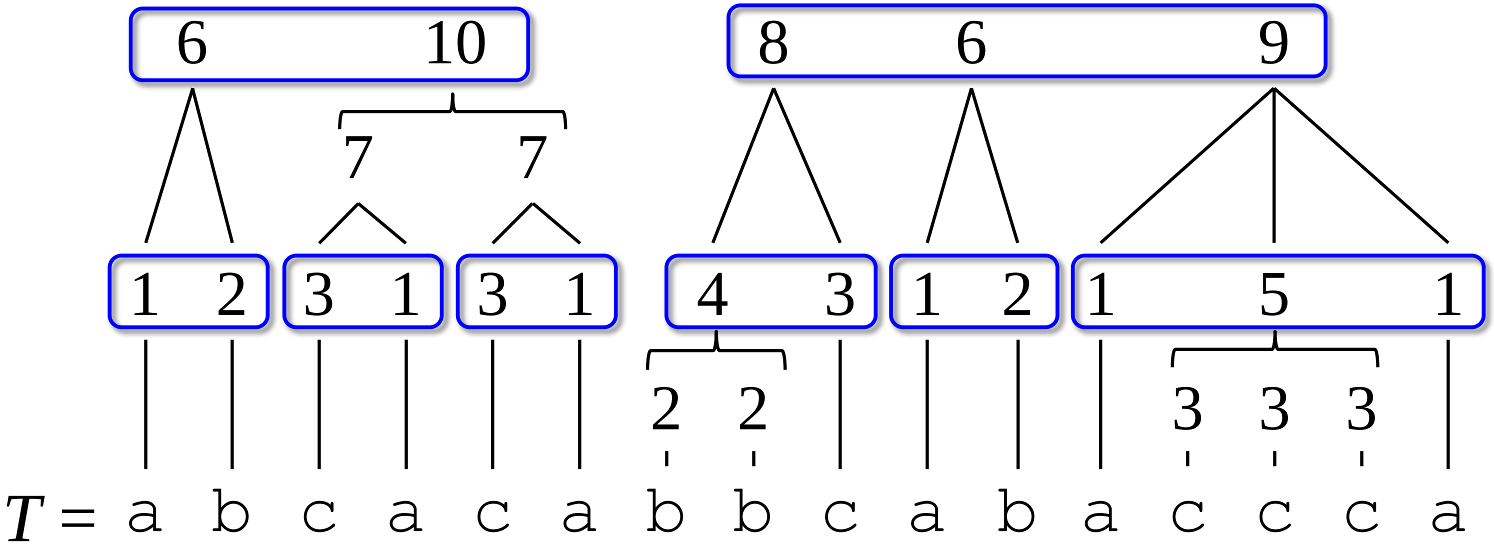
✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

Apply locally consistent parsing to this string.



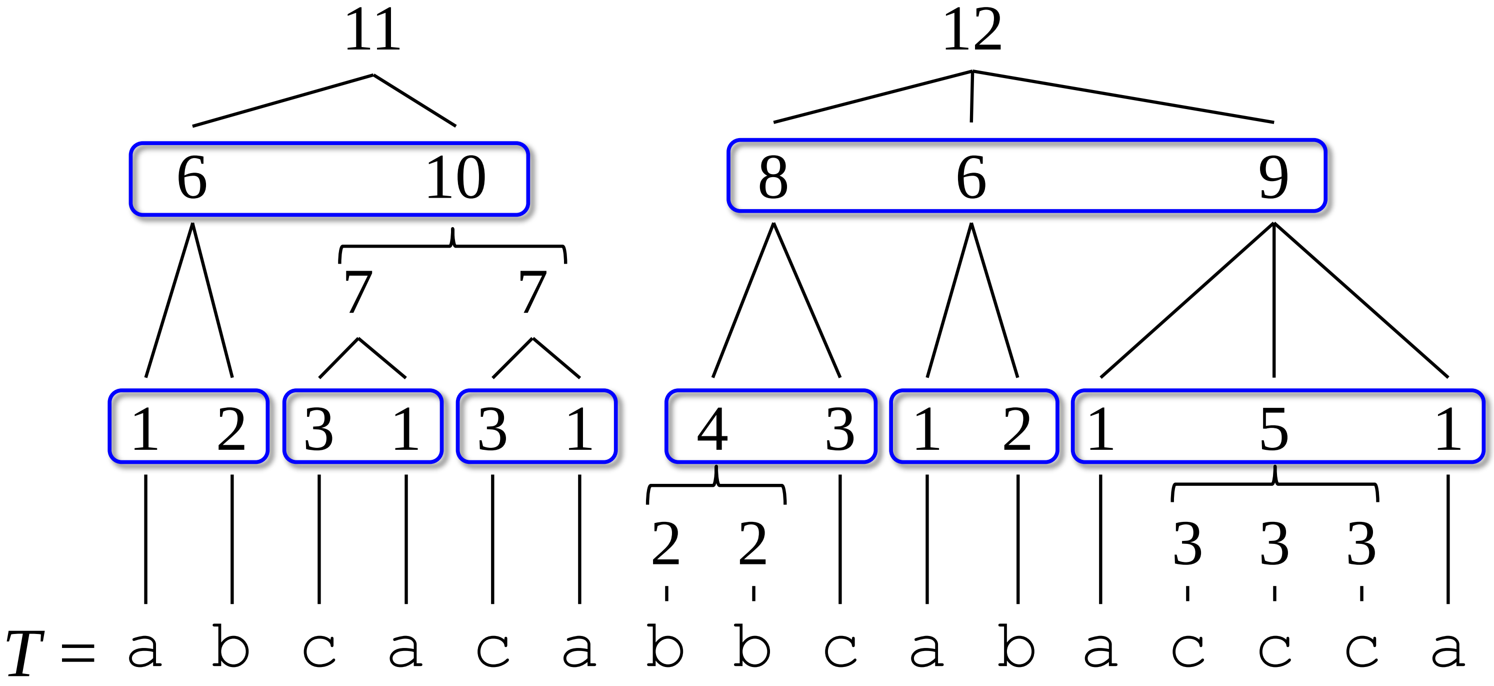
# Signature encoding [Mehlhorn et al. '97]

✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.



# Signature encoding [Mehlhorn et al. '97]

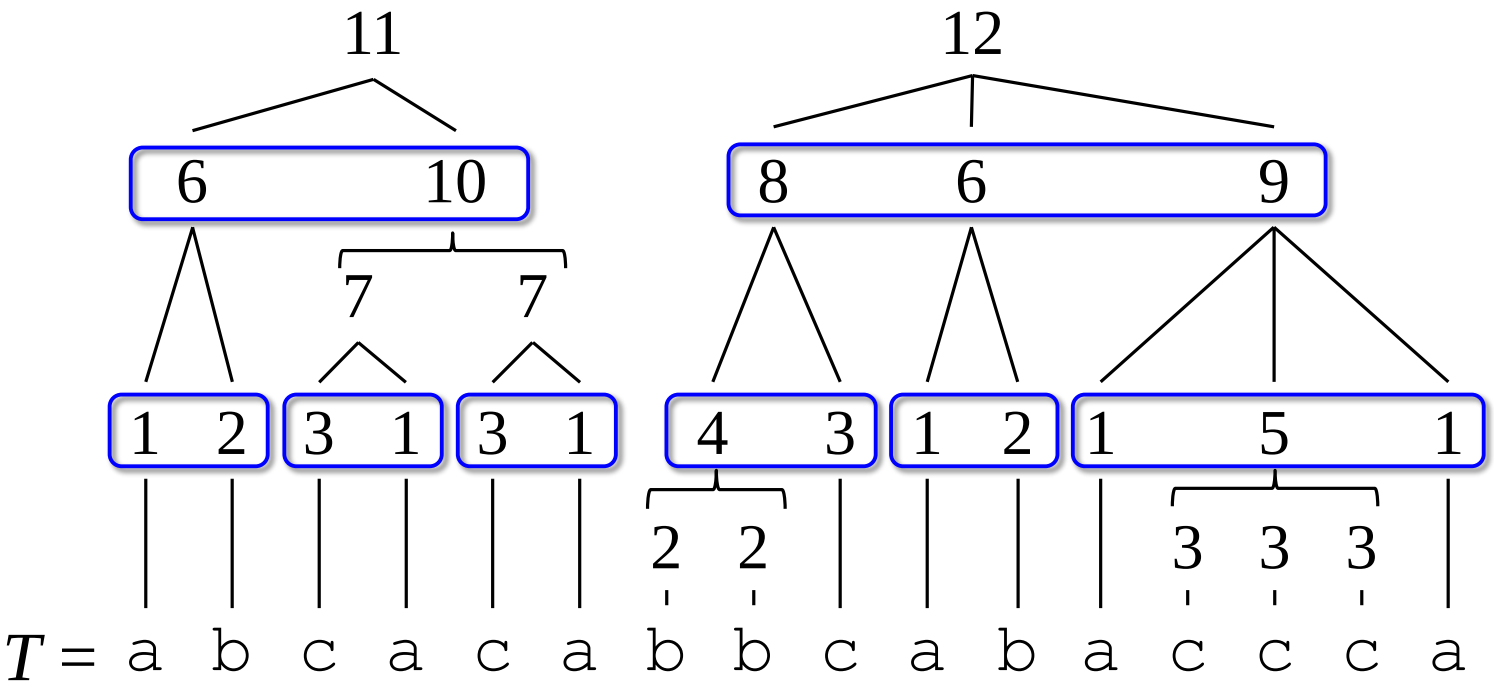
- ✓ Iteratively apply locally consistent parsing to input string *T* until a single integer is obtained.



# Signature encoding [Mehlhorn et al. '97]

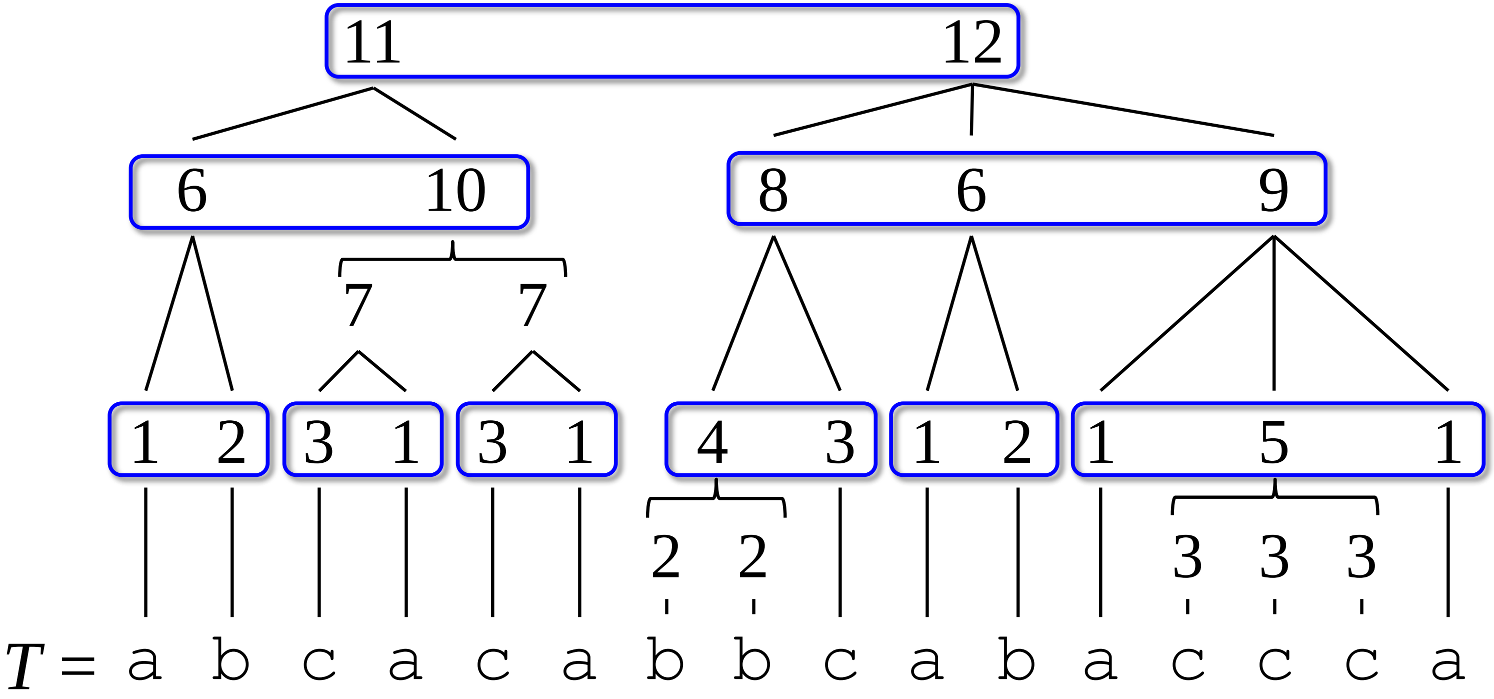
- ✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.

Apply locally consistent parsing to this string.



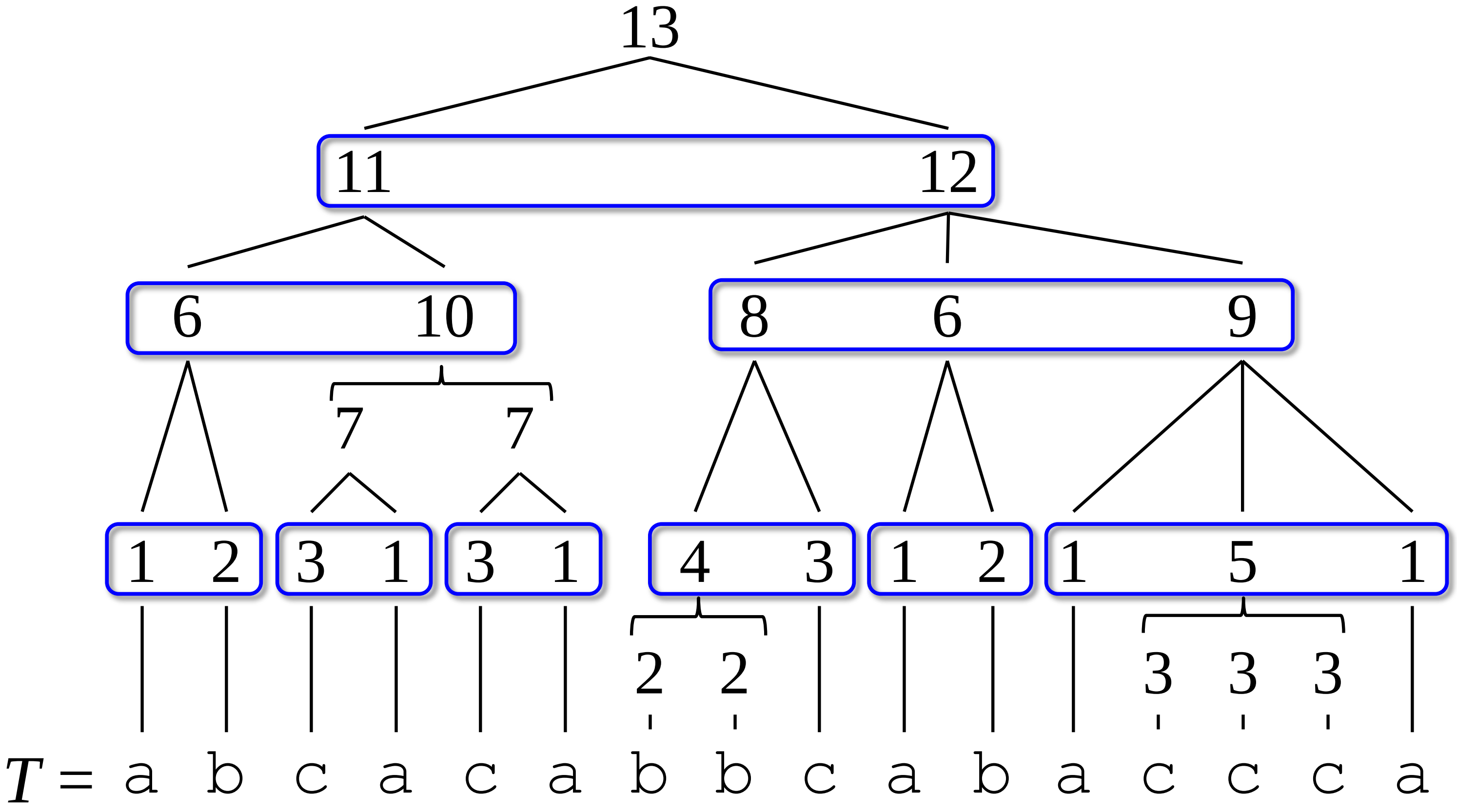
# Signature encoding [Mehlhorn et al. ’97]

✓ Iteratively apply locally consistent parsing to input string *T* until a single integer is obtained.



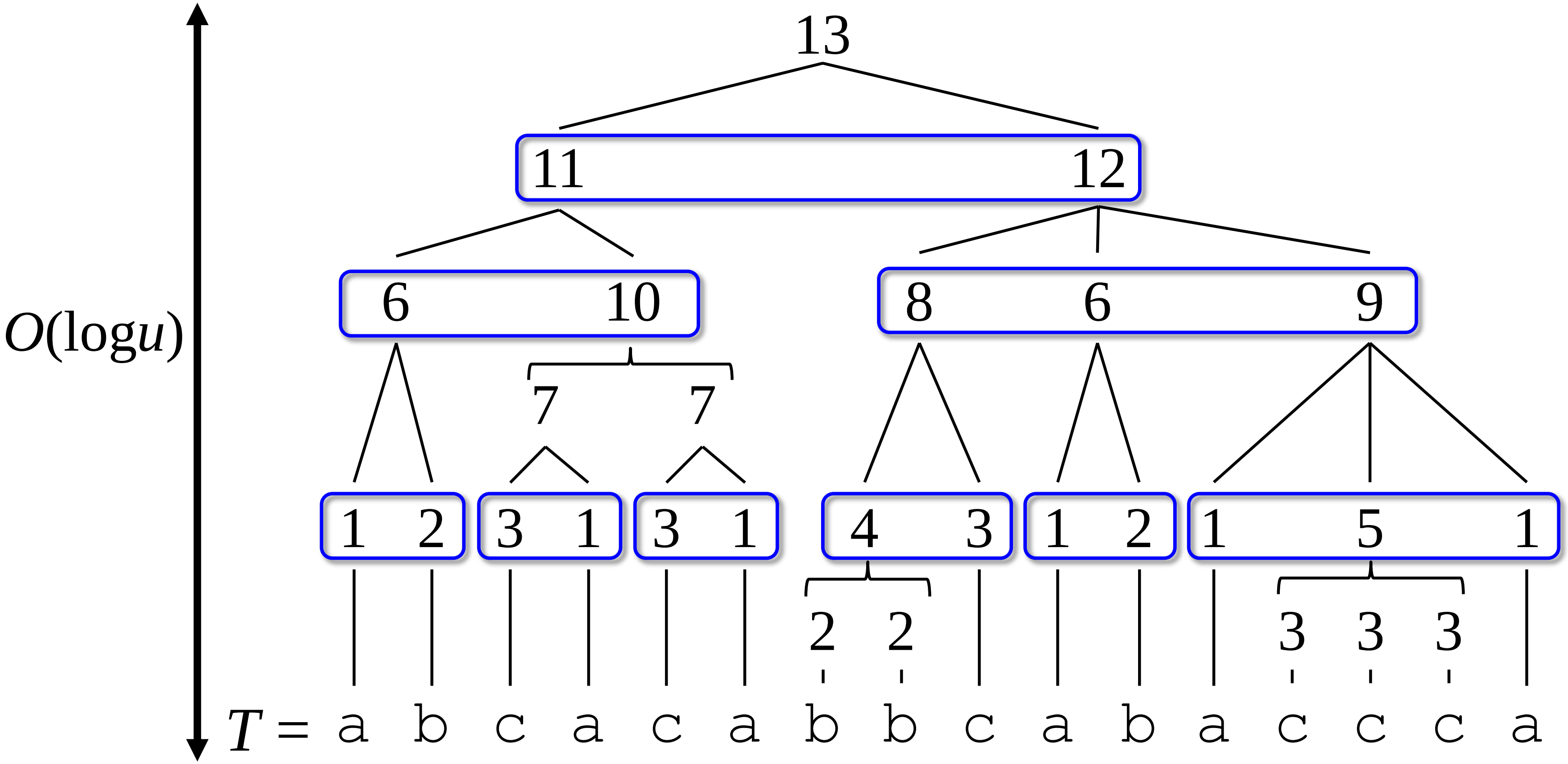
# Signature encoding [Mehlhorn et al. '97]

- ✓ Iteratively apply locally consistent parsing to input string $T$ until a single integer is obtained.



# Signature encoding [Mehlhorn et al. ’97]

✓ The height of this tree, called the signature tree, is $O(\log u)$, where $u = |T|$.



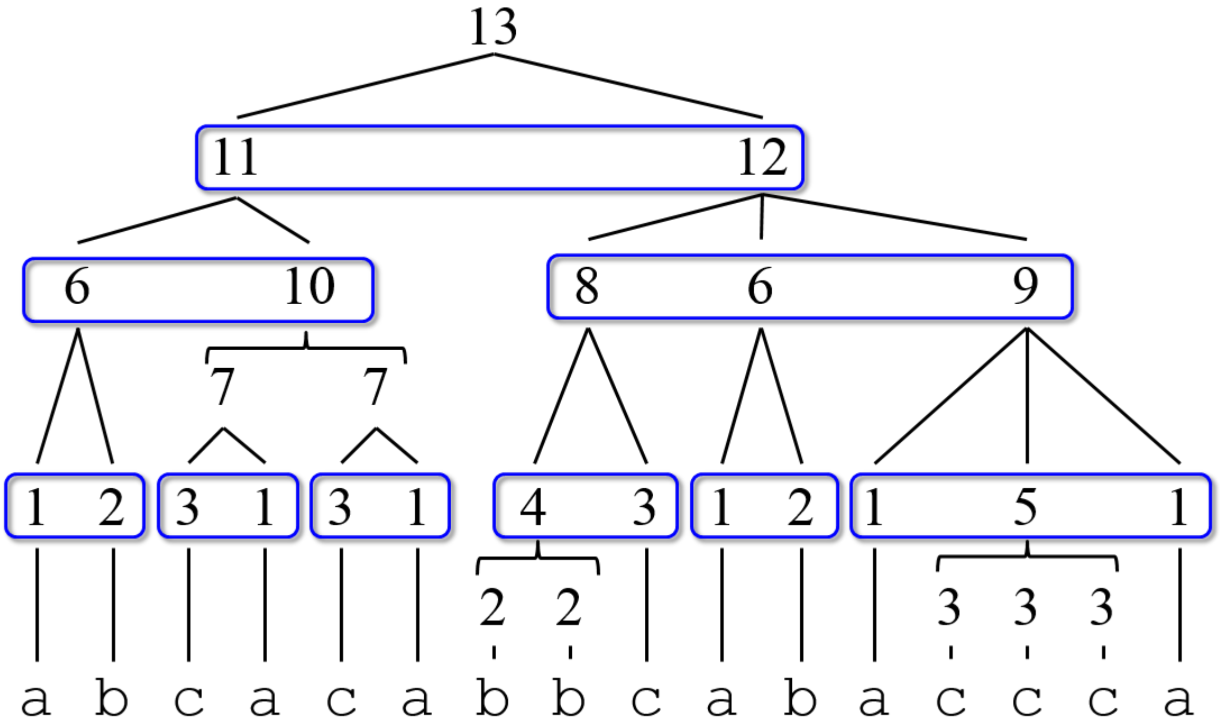
# Signature encoding [Mehlhorn et al. '97]

✓ The dictionary $D_T$ of signatures is
the **signature encoding** of input string $T$.

$D_T$

$13 \rightarrow 11, 12$
$12 \rightarrow 8, 6, 9$
$11 \rightarrow 6, 10$
$10 \rightarrow 7^2$
$9 \rightarrow 1, 5, 1$
$8 \rightarrow 4, 3$
$7 \rightarrow 3, 1$
$6 \rightarrow 1, 2$
$5 \rightarrow 3^3$
$4 \rightarrow 2^2$
$3 \rightarrow \mathtt{c}$
$2 \rightarrow \mathtt{b}$
$1 \rightarrow \mathtt{a}$

signature tree of $T$



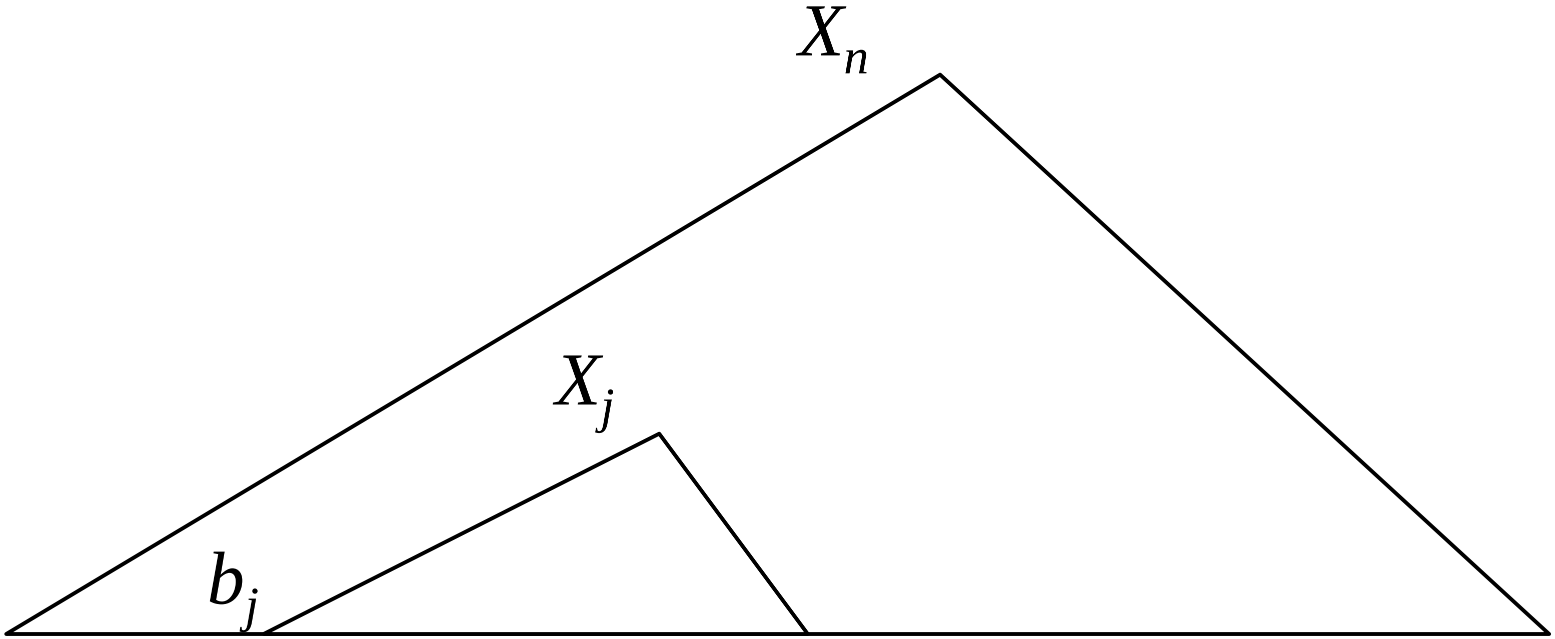
# Faster LCE algorithm on SLP

Lemma 2 (Faster LCE on SLP)

Given the signature encoding $D_T$ of string $T$ of length $u$, we can compute $\mathbf{LCE}(X_j, X_k, p, q)$ for any variables $X_j$, $X_k$ and positions $p$, $q$ in $O(\log u + \log^* u \log L)$ time, where $L$ is the answer to the query (LCE length).

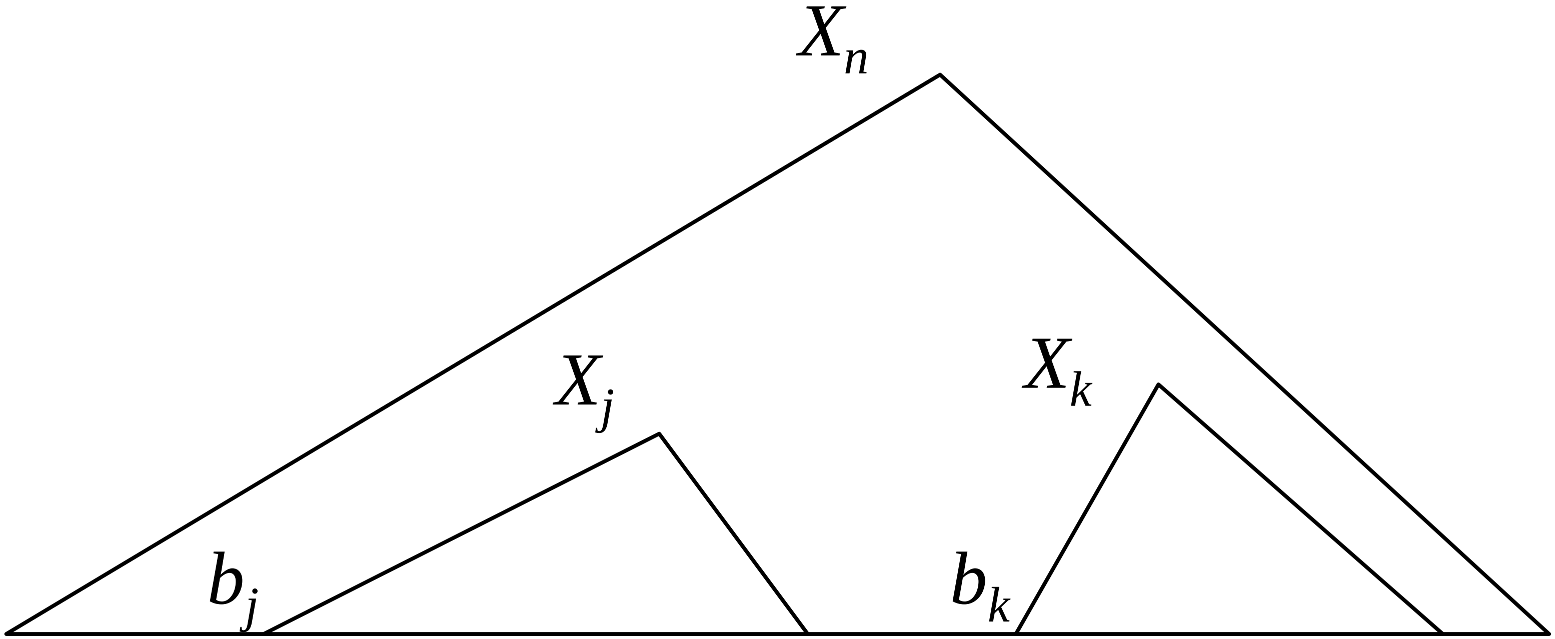
# Faster LCE algorithm on SLP

1. For every non-terminal $X_j$, we precompute and store its occurrence $b_j$ in the derivation tree of $X_n$.



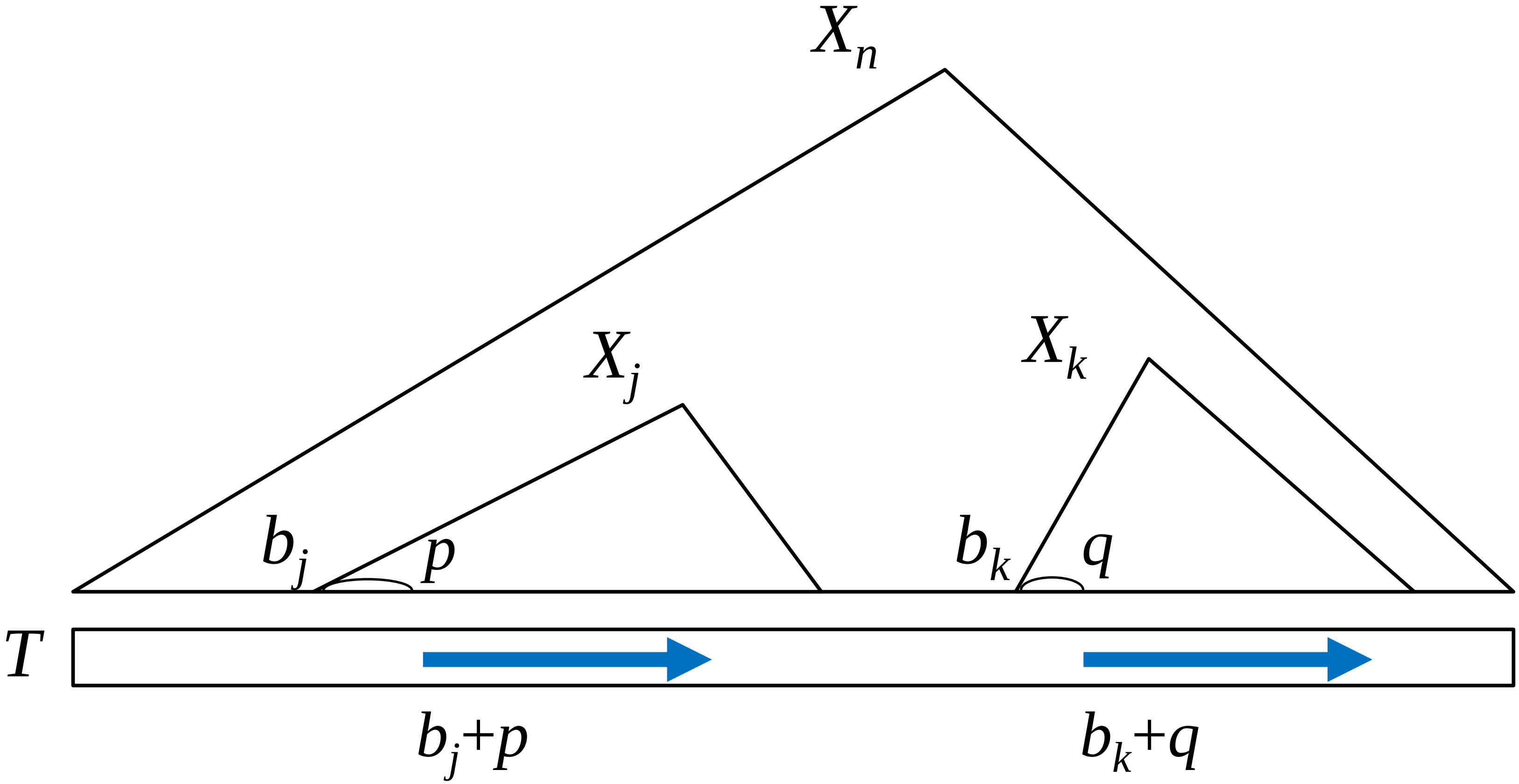
# Faster LCE algorithm on SLP

2. Given query variables $X_j$ and $X_k$ for LCE, we retrieve $b_j$ and $b_k$.



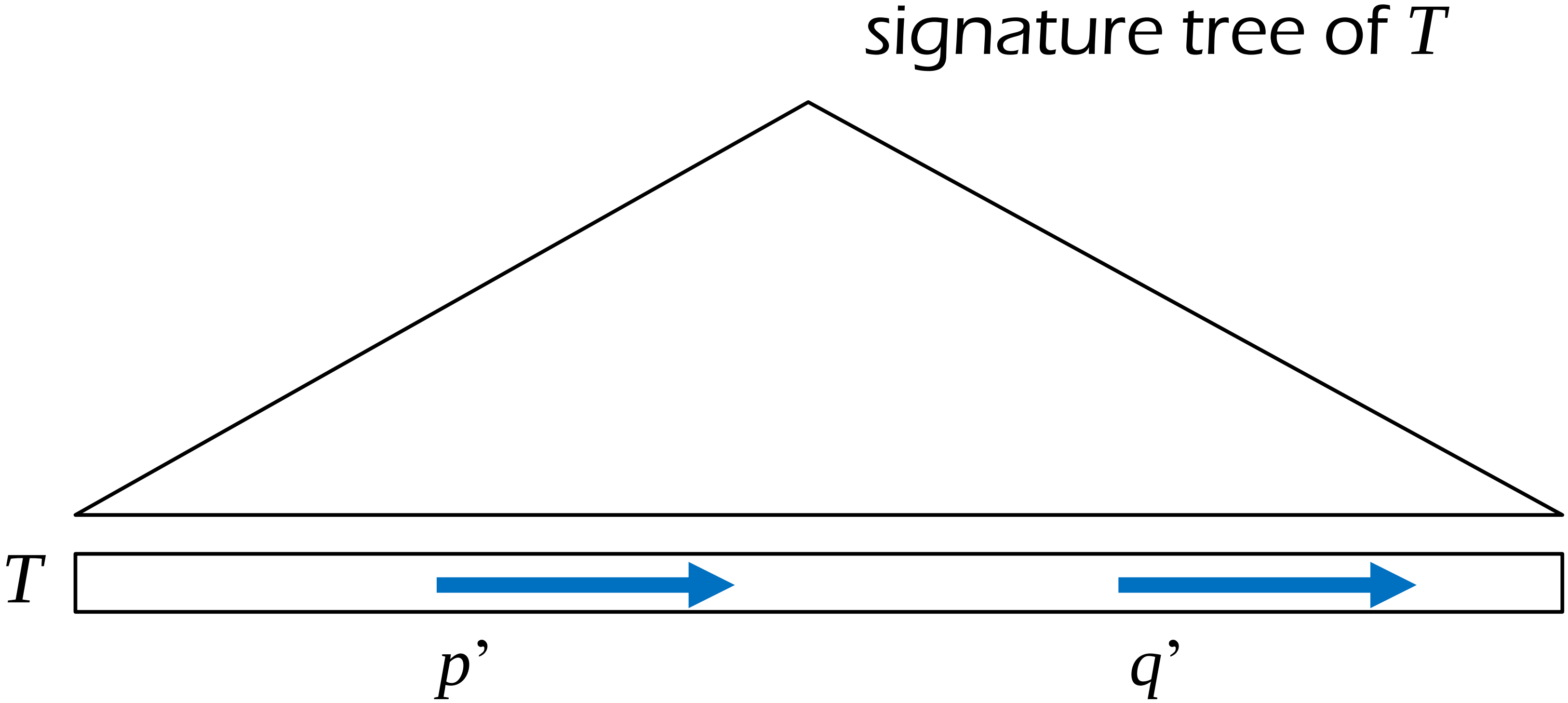
# Faster LCE algorithm on SLP

3. Since the last variable $X_n$ derives string $T$, $\mathbf{LCE}(X_j, X_k, p, q)$ reduces to $\mathbf{LCE}(b_j+p, b_k+q)$ on string $T$.



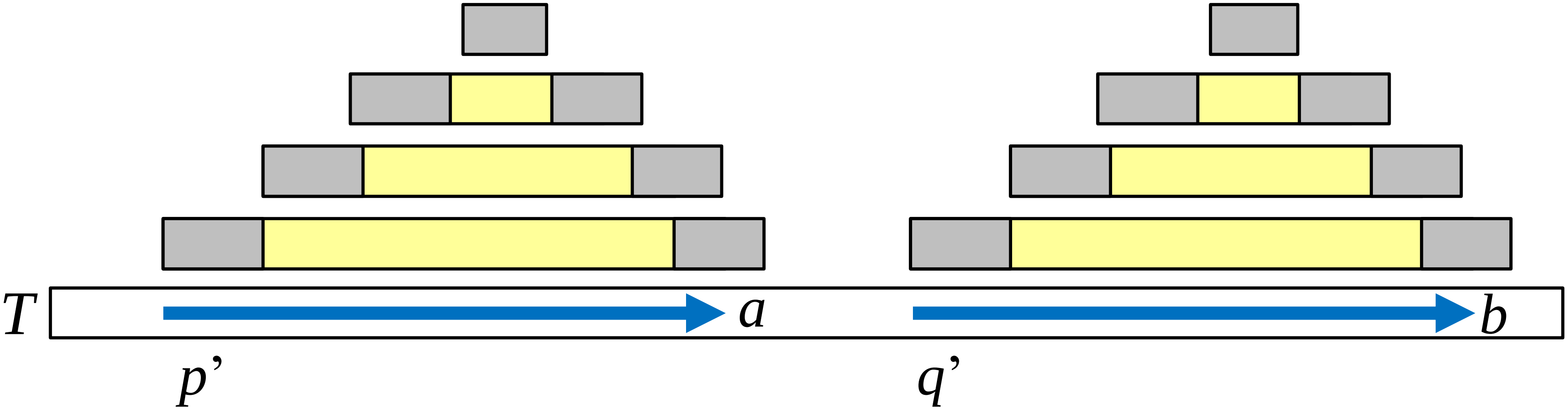
# Faster LCE algorithm on SLP

4. We turn attention to the signature tree of $T$, and compute $\mathbf{LCE}(p', q')$ there, where $p' = b_j+p$ and $q' = b_k+q$.



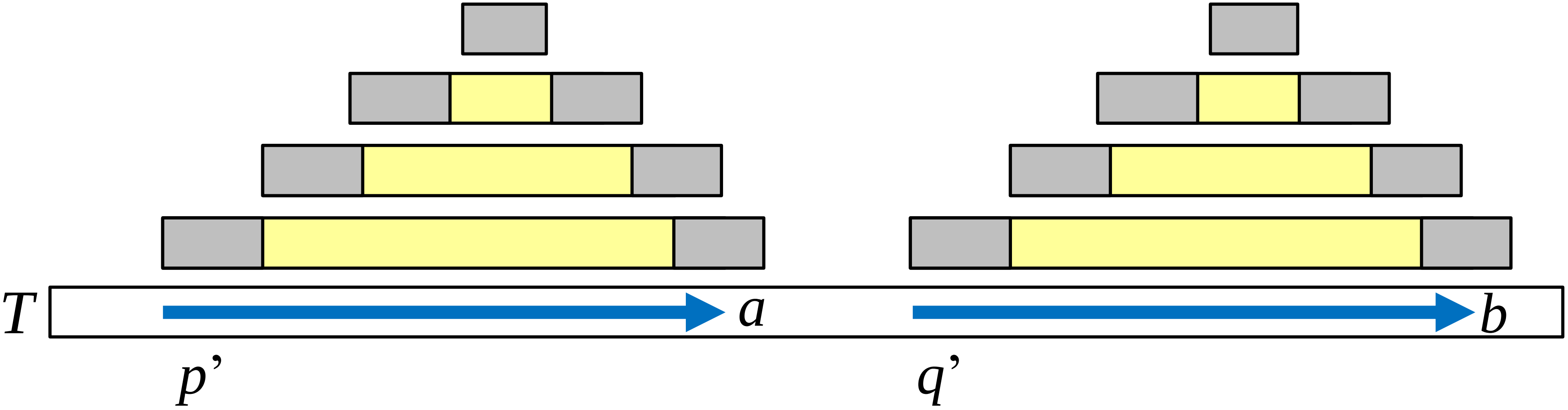
# Faster LCE algorithm on SLP

5. By the property of signature encoding, at each level of the signature tree, there must be a common sequence of signatures for **LCE**($p$', $q$') (yellow parts).



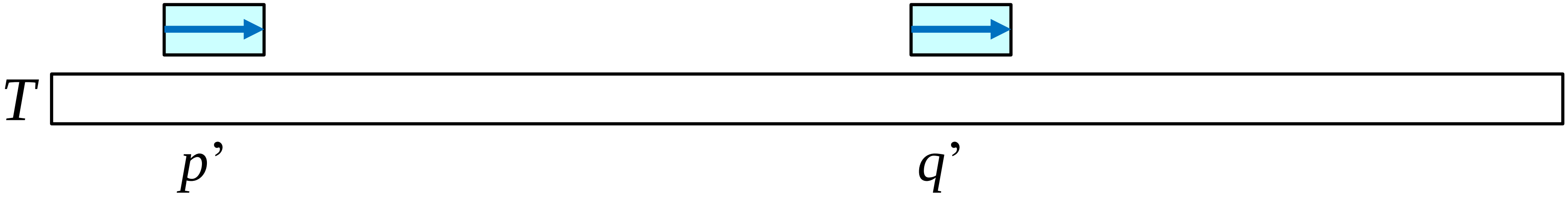
# Faster LCE algorithm on SLP

5. [Cont.] The left boundaries of length $\Delta_L$+$O(1)$ may or may not be equal depending on the left contexts at each level, while the right boundaries of length $\Delta_R$+$O(1)$ always have a mismatch.



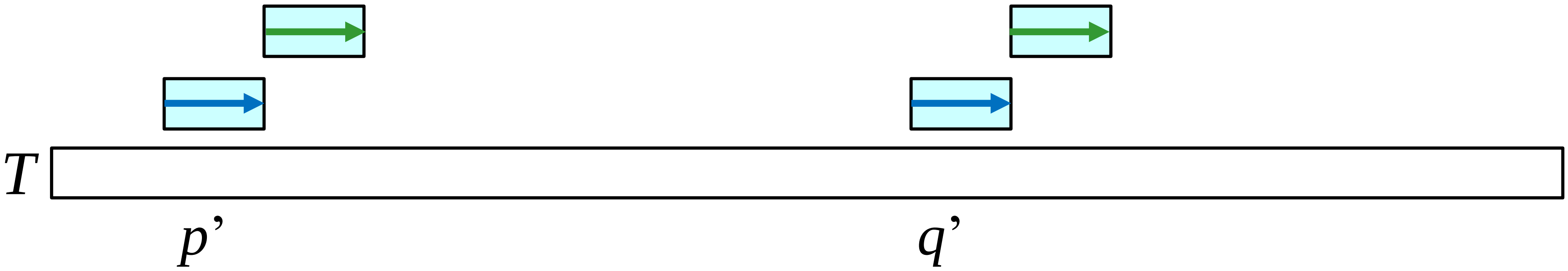
# Faster LCE algorithm on SLP

6. In a bottom-up manner, we re-compute the left boundary signatures of length $\Delta_L + O(1)$ ignoring their left contexts, and compare them until we find a mismatch.



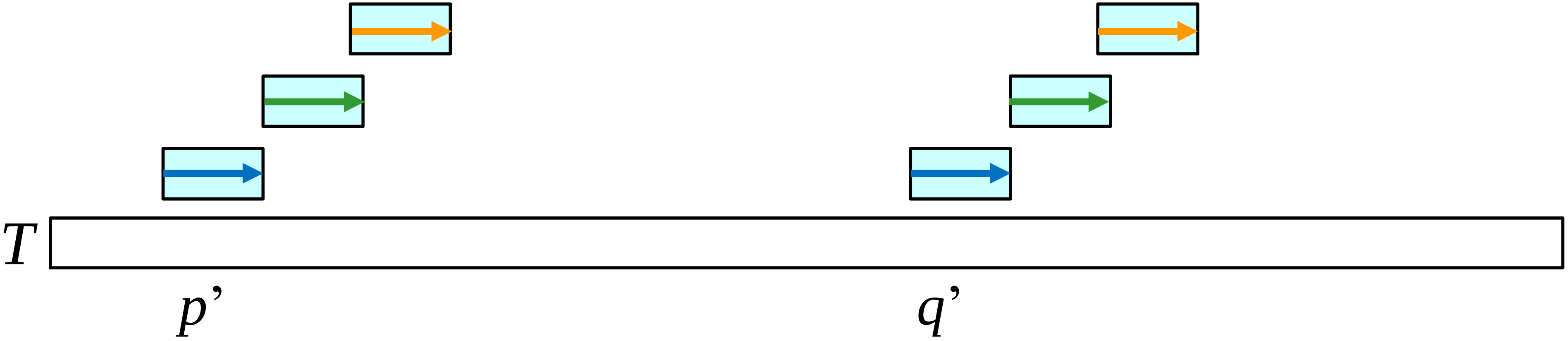
# Faster LCE algorithm on SLP

6. In a bottom-up manner, we re-compute the left boundary signatures of length $\Delta_L+O(1)$ ignoring their left contexts, and compare them until we find a mismatch.



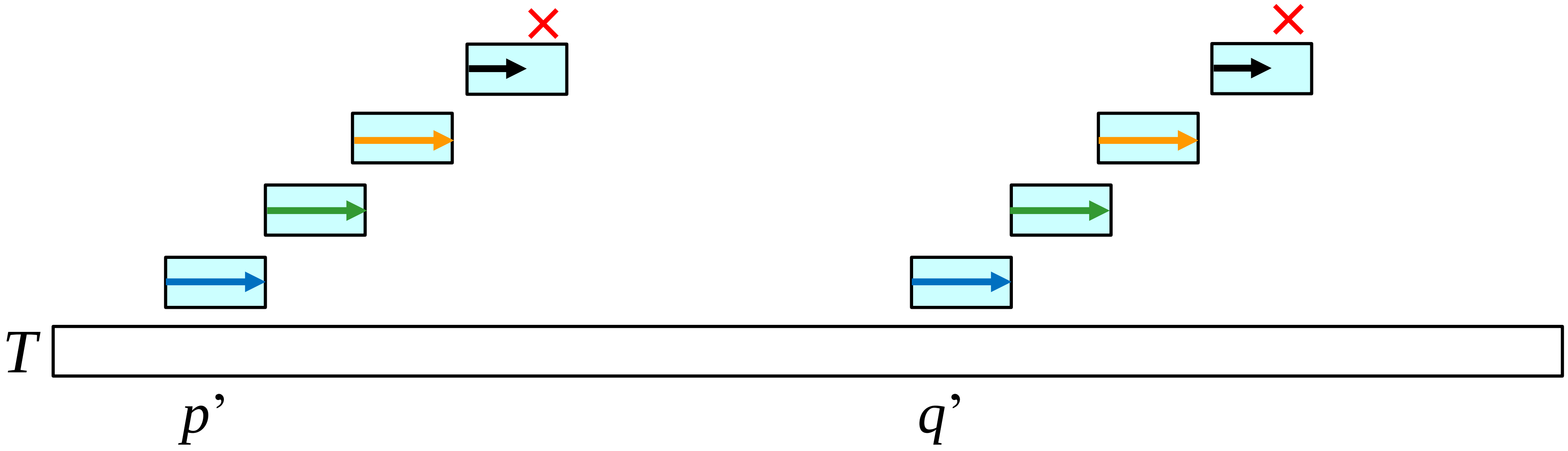
# Faster LCE algorithm on SLP

6. In a bottom-up manner, we re-compute the left boundary signatures of length $\Delta_L + O(1)$ ignoring their left contexts, and compare them until we find a mismatch.



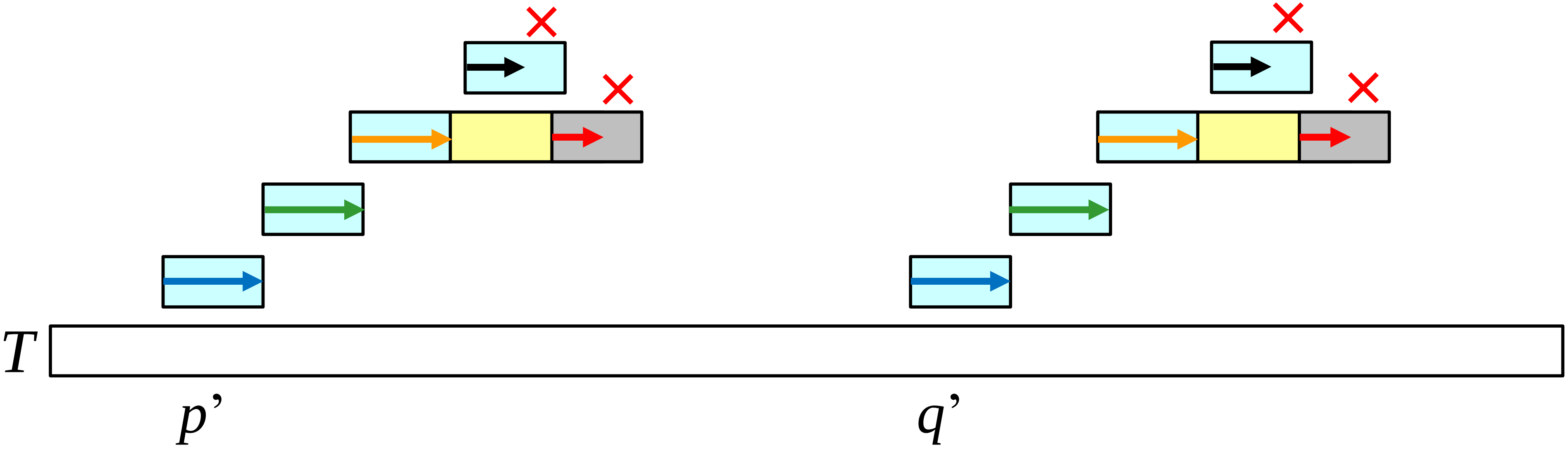
# Faster LCE algorithm on SLP

6. In a bottom-up manner, we re-compute the left boundary signatures of length $\Delta_L$+$O(1)$ ignoring their left contexts, and compare them until we find a mismatch.



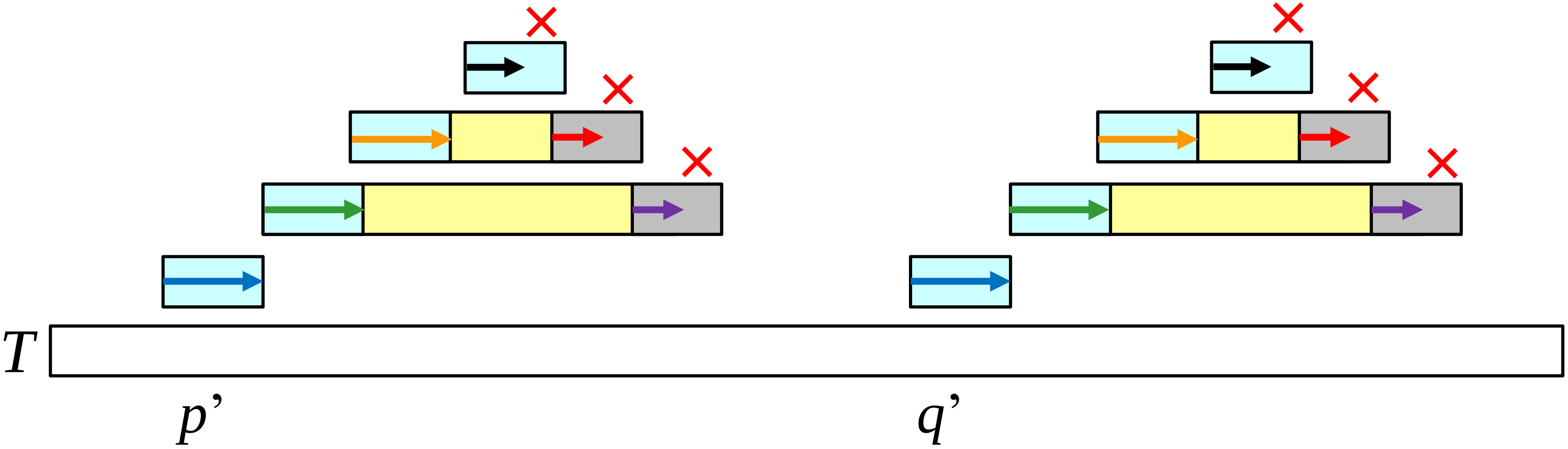
# Faster LCE algorithm on SLP

7. In a top-down manner, we compare the right boundary signatures of length $\Delta_R + O(1)$ until we find the first mismatch.



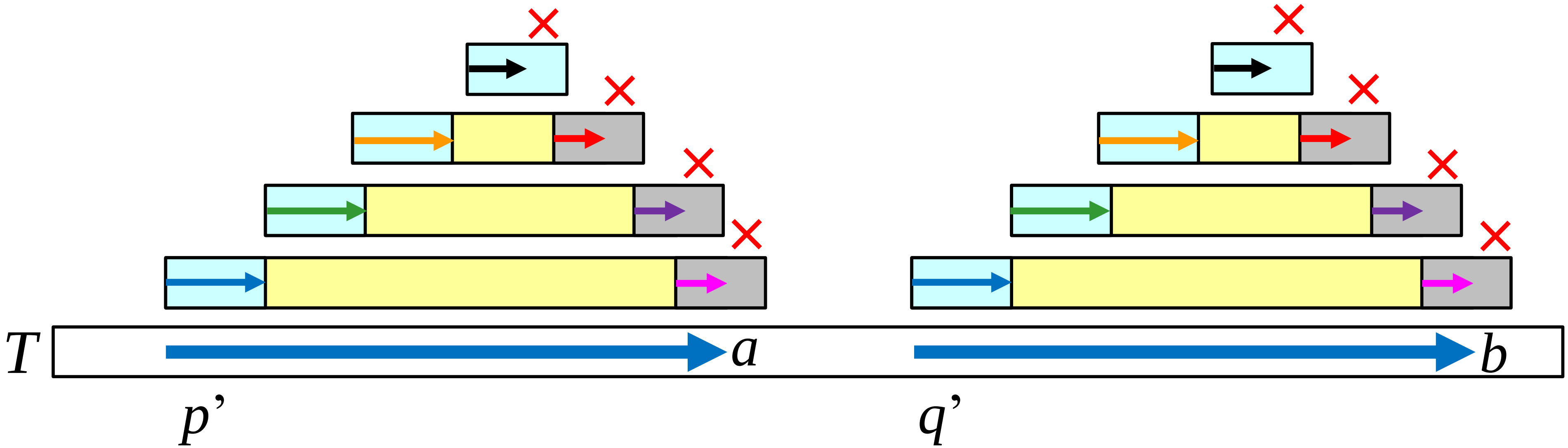
# Faster LCE algorithm on SLP

7. In a top-down manner, we compare the right boundary signatures of length $\Delta_R + O(1)$ until we find the first mismatch.



# Faster LCE algorithm on SLP

7. In a top-down manner, we compare the right boundary signatures of length $\Delta_R + O(1)$ until we find the first mismatch.



# Analysis of LCE query time

✓ The paths from the root to the $p$'th and $q$'th leaves of the signature tree can be found in $O(\log u)$ time, since its height is $O(\log u)$.

✓ The total number of signatures to re-compute and to compare is $O(\log^* u \log L)$, since:

- ➢ $\Delta_L \leq \log^* u + 6$ and $\Delta_R \leq 4$, and
- ➢ the first mismatch is found at the $(\log L)$th level from the bottom.

✓ Therefore, LCE query can be answered in $O(\log u + \log^* u \log L)$ time.

# From SLP to signature encoding

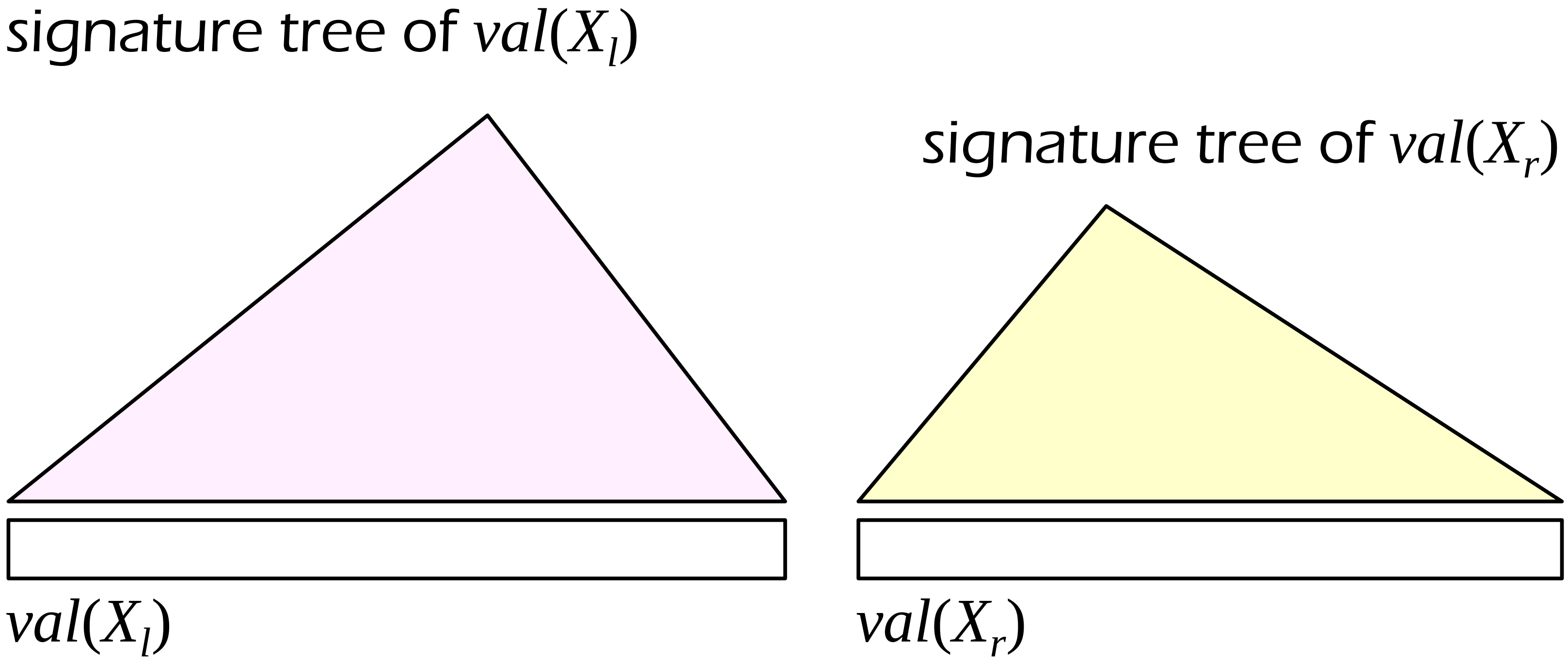
Lemma 3 (SLP to signature encoding)

Given an SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ of size $n$ which derives a string $T$ of length $u$, we can compute the signature encoding of $T$ in $O(n \text{ loglog } n \log^* u \log u)$ time.

✓ In this talk I show a simpler $O(n \log n \log^* u \log u)$-time construction.

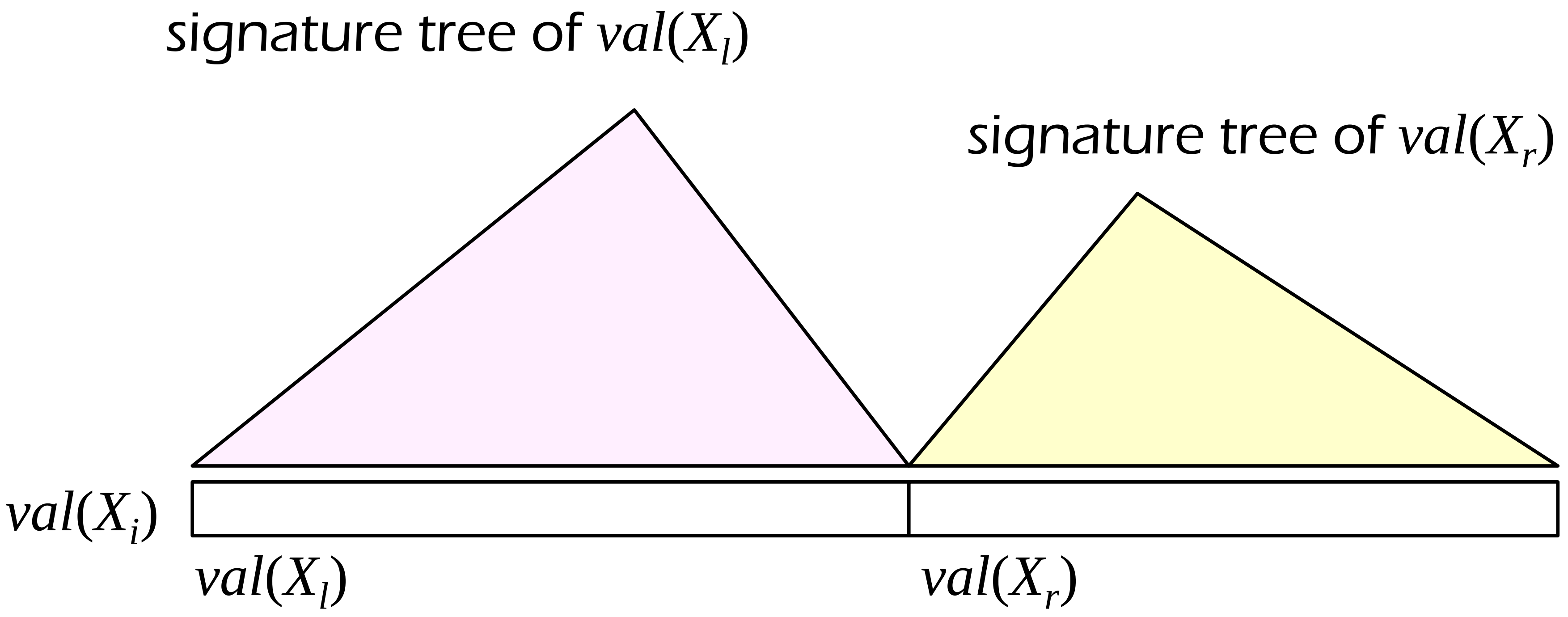
# From SLP to signature encoding

✓ Assume that, for a production $X_i \rightarrow X_l X_r$, we have computed the signature encodings of the decompressed strings $val(X_l)$ and $val(X_r)$.



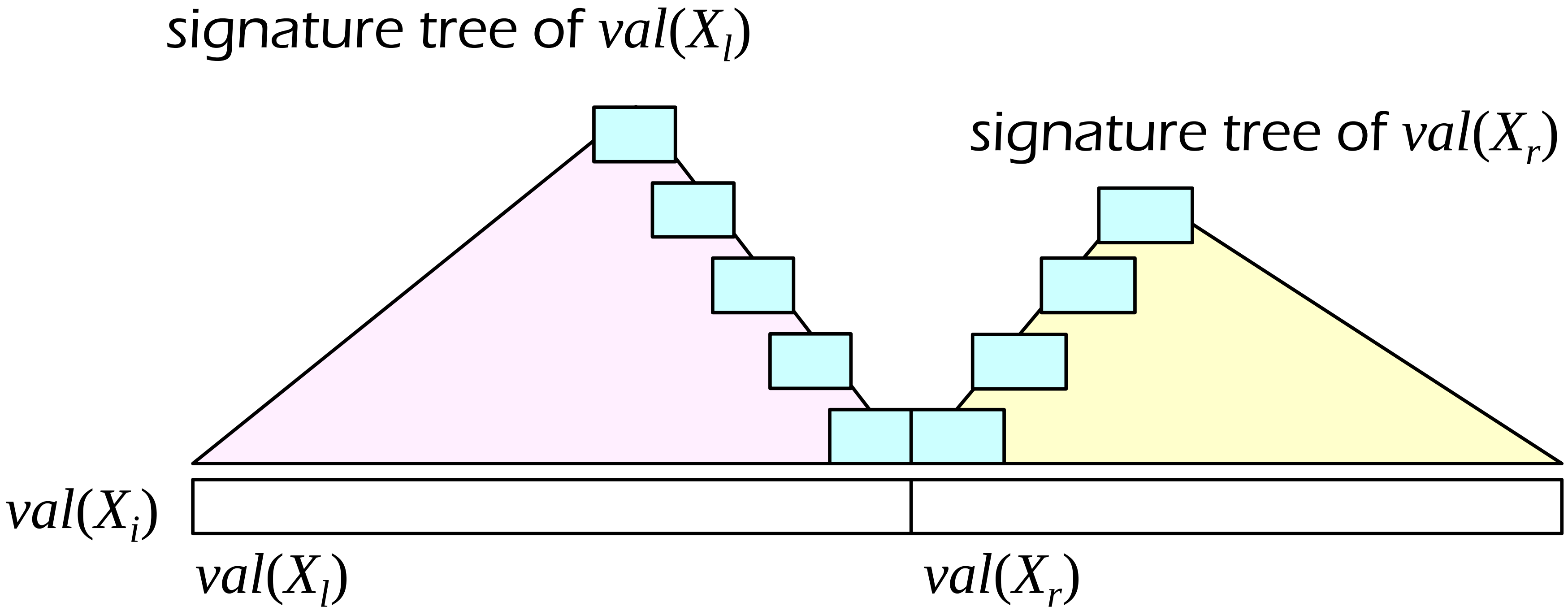
# From SLP to signature encoding

- ✓ By "concatenating" the signature trees of $val(X_l)$ and $val(X_r)$, we obtain the signature tree of $val(X_i)$.



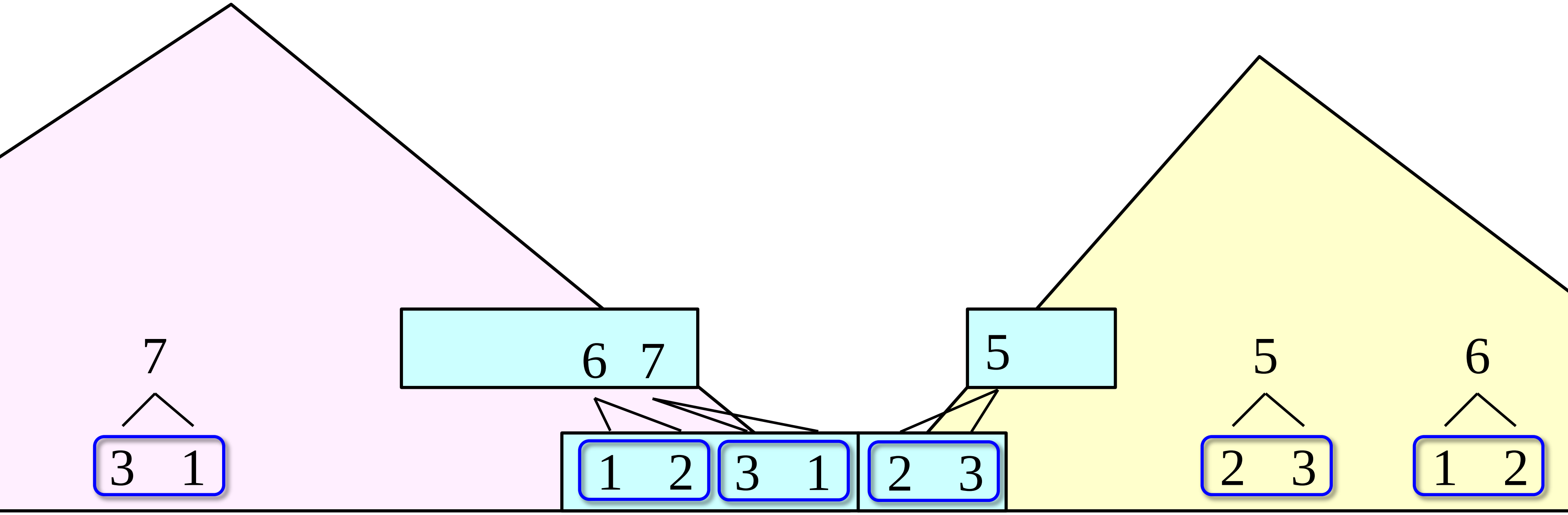
# From SLP to signature encoding

- ✓ In a bottom-up manner, we re-compute the boundary signatures of length $\Delta_R+O(1)$ and $\Delta_L+O(1)$ each, and concatenate the new signatures level-wise.



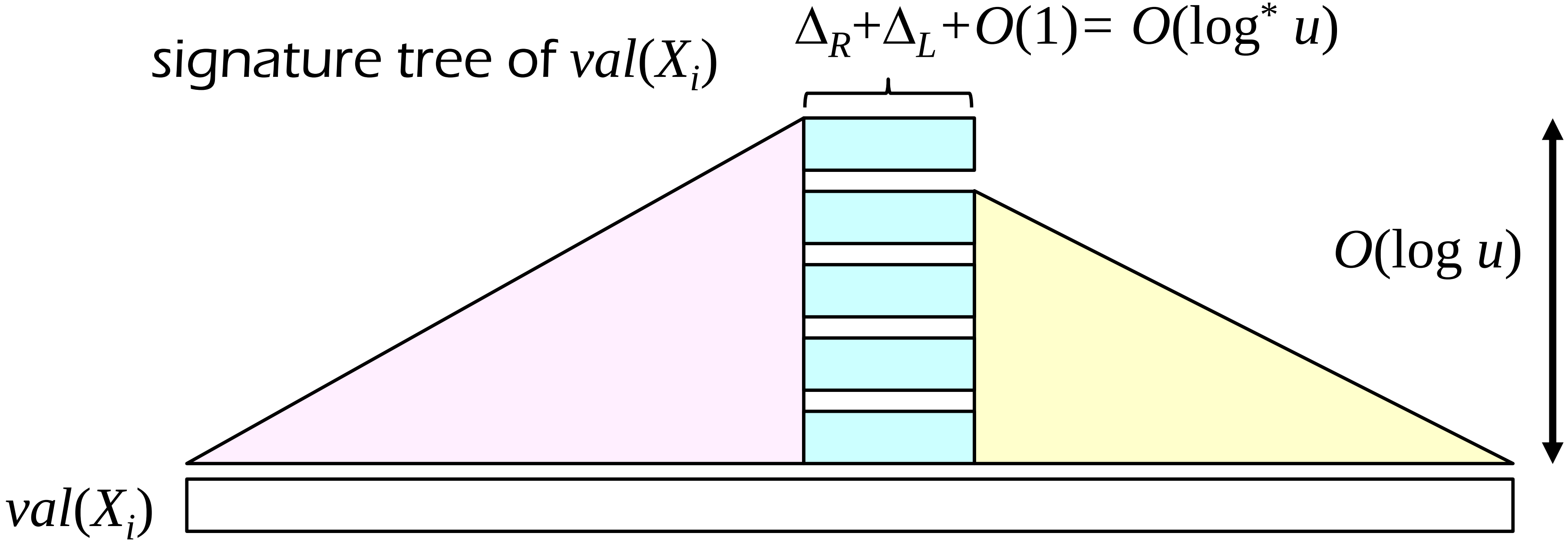
# From SLP to signature encoding

- ✓ If a block of re-computed signatures already exists somewhere else, then we assign the same signature to the block at the next level. This is done in $O(\log n)$ time each, using a BST.



# From SLP to signature encoding

- ✓ Since the height of each signature tree is $O(\log u)$, we can compute the signature encoding of $val(X_i)$ for each $X_i$ in $O(\log n \log^* u \log u)$ time.


signature tree of $val(X_i)$
$\Delta_R+\Delta_L+O(1)= O(\log^* u)$
$O(\log u)$
$val(X_i)$

# How much space?

Lemma 4 [Sahinalp & Vishkin, ’95]

The number of signatures involved in the signature encoding of string $T$ of length $u$ is $O(z \log^{*} u \log u)$, where $z$ is the number of factors in the Lempel-Ziv 77 factorization of $T$.

- ✓ In our data structure, we need an additive $n$ term to store beginning positions of occurrences of all non-terminals in the derivation tree of $X_n$.

# Main result

Theorem 1

For any SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ of size $n$ which represents a string $T$ of length $u$, there exists a data structure which

- supports LCE in $O(\log u + \log^* u \log L)$ time;
- requires $O(n + z \log^* u \log u)$ space;
- can be built in $O(n \log\log n \log^* u \log u)$ time,

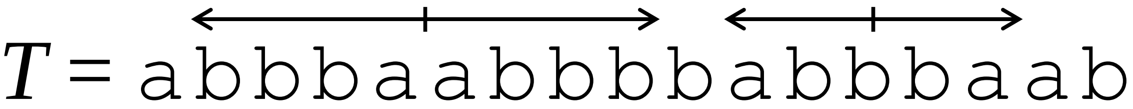
where $L$ is the LCE length and $z$ is the size of the LZ77 factorization of $T$.

# App 1: Finding palindromes

**Problem 2 (finding palindromes on SLP)**

Given an SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ representing a string $T$, compute a compact representation of all maximal palindromes in $T$.

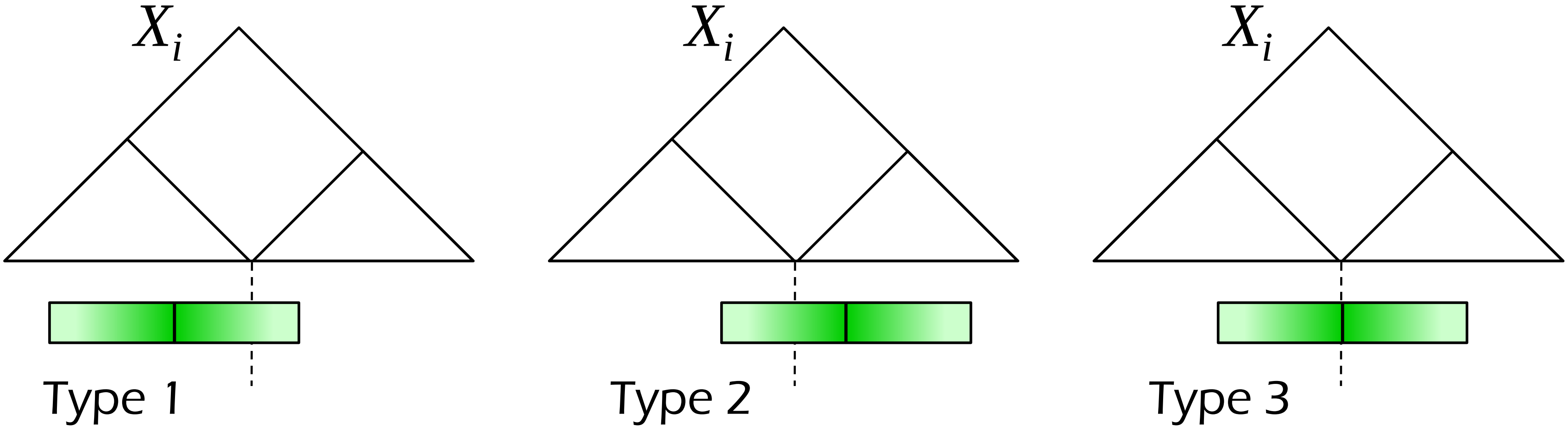
maximal palindromes



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

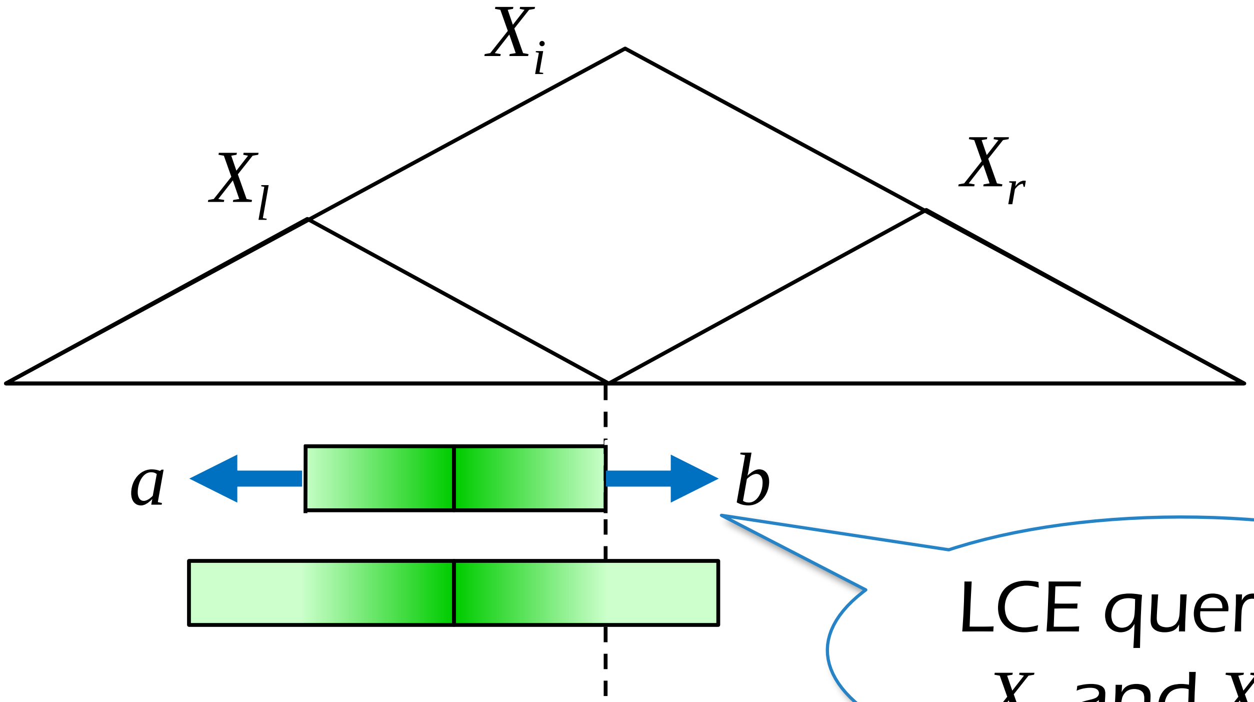
# Stabbed Palindromes

✓ For each non-terminal $X_i$, there are 3 different types of “stabbed” maximal palindromes.



# Computing Type 1 Palindromes

- ✓ Each Type 1 maximal palindrome of $X_i$ can be computed by extending the arms of a suffix palindrome of $X_l$.



LCE query for $X_r$ and $X_l^{\mathrm{rev}}$ .

# Suffix Palindromes

Lemma 5 [Apostolico et al., '95]

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

- ✓ We can extend the arms of the suffix palindromes belonging to the same arithmetic progression in a batch, using periodicity.

# App 1: Finding Palindromes

Theorem 2

Given an SLP of size $n$, an $O(n \log u)$-size representation of all maximal palindromes of string $T$ can be computed in $O(n \log^* u \log^2 u)$ time.

With this representation, given an interval $[i, j]$, we can decide whether the substring $T[i..j]$ is a maximal palindrome or not in $O(\log u)$ time.

# App 2: Comparing Suffixes on SLP

Problem 3 (lexicographical comparison of suffixes)

Preprocess an input SLP representing string $T$ so that later, any suffixes of the string $T$ can be lexicographically compared efficiently.

# App 2: Comparing Suffixes on SLP

Theorem 3

We can preprocess an input SLP of size $n$ representing string $T$ of length $u$ in $O(n \log\log n \log^* u \log u)$ time such that later, any suffixes of $T$ can be lexicographically compared in $O(\log u + \log^* u \log L)$ time, where $L$ is the length of the LCP of the suffixes.

✓ Since the height of the signature tree is $O(\log u)$, this theorem is immediate from our LCE data structure.

# App 3: Lyndon factorization on SLP

Problem 4 (Lyndon factorization on SLP)

Given an SLP $S = \{X_i \rightarrow expr_i\}_{i=1}^{n}$ representing a string $T$, compute the factor boundaries of the Lyndon factorization of $T$.

# Lyndon word

Definition

A string is said to be a Lyndon word if it is lexicographically smaller than any of its proper cyclic shifts.

For example, "`aaaab`", "`abc`", "`bcbcc`" are Lyndon words.

# Lyndon factorization

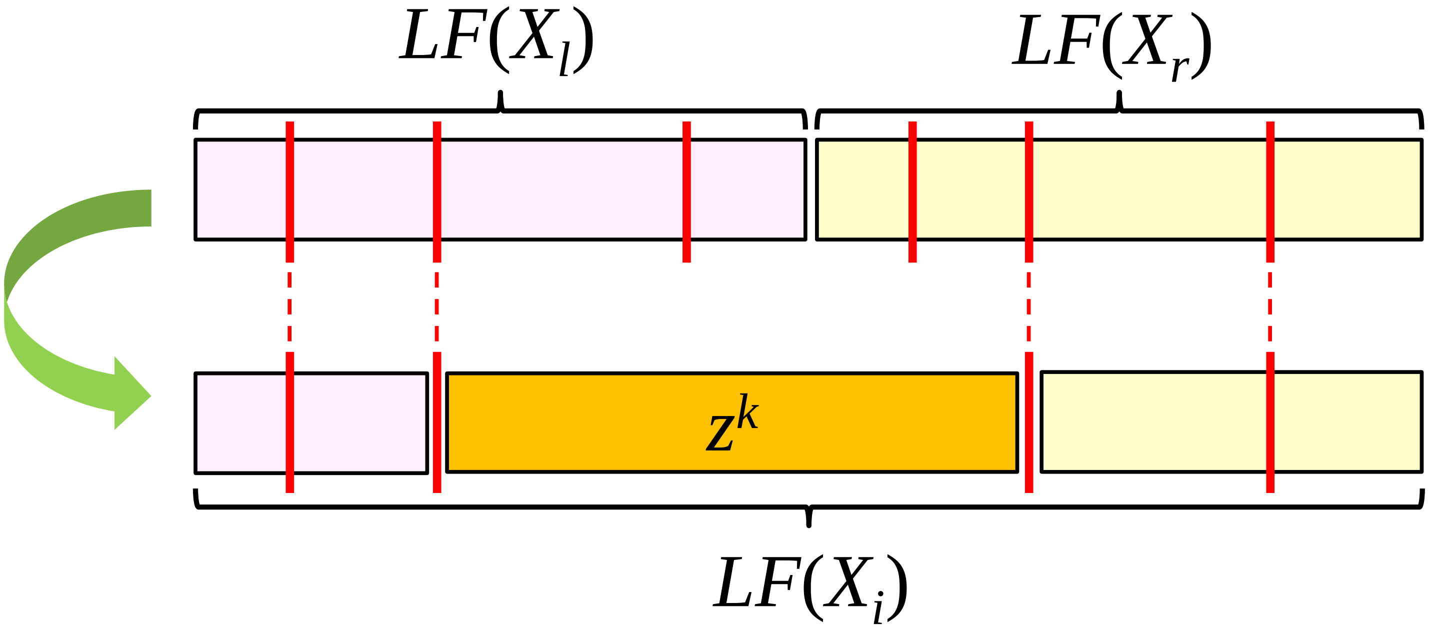
**Definition**

The Lyndon factorization $LF(T)$ of a string $T$ is the factorization $u_1^{p_1}, \ldots, u_m^{p_m}$ of $T$ such that $u_1, \ldots, u_m$ is a sequence of Lyndon words in lexicographical descending order, and $p_i \geq 1$.

$$T = \underbrace{\mathrm{a\,b\,c}}_{u_1} \mid \underbrace{\mathrm{a\,b\,b}}_{u_2} \vdots \underbrace{\mathrm{a\,b\,b}}_{u_2} \mid \underbrace{\mathrm{a\,a\,b\,c}}_{u_3} \mid \underbrace{\mathrm{a}}_{u_4} \vdots \underbrace{\mathrm{a}}_{u_4} \vdots \underbrace{\mathrm{a}}_{u_4}$$

$$LF(T) = \underbrace{(\mathrm{abc})^1}_{u_1^1} \mid \underbrace{(\mathrm{abb})^2}_{u_2^2} \mid \underbrace{(\mathrm{aabc})^1}_{u_3^1} \mid \underbrace{(\mathrm{a})^3}_{u_4^3}$$

# Lyndon factorization on SLP



- ✓ I et al. showed an algorithm which computes $LF(X_i)$ with $X_i \to X_l X_r$ in the above manner.
- ✓ The beginning and ending positions of the median Lyndon factor $z^k$ can be found by a binary search based on lex-comparison of suffixes.

# App 3: Lyndon factorization on SLP

Theorem 4

Given an SLP of size $n$ representing string $T$ of length $u$, we can compute the factor boundaries of the Lyndon factorization of $T$ in $O(n \log\log n \log^* u \log u)$ time and $O(n^2 + z \log^* u \log u)$ space.

# Conclusions & further work

- ■ We proposed a new LCE algorithm on SLPs with $O(\log u + \log^* u \log L)$ query time.
  - ✓ This is the fastest deterministic solution to date.
  - ✓ More details can be found in our arxiv paper: "Dynamic index, LZ factorization, and LCE queries in compressed space".

- ◆ Lower bound?
- ◆ Other applications?