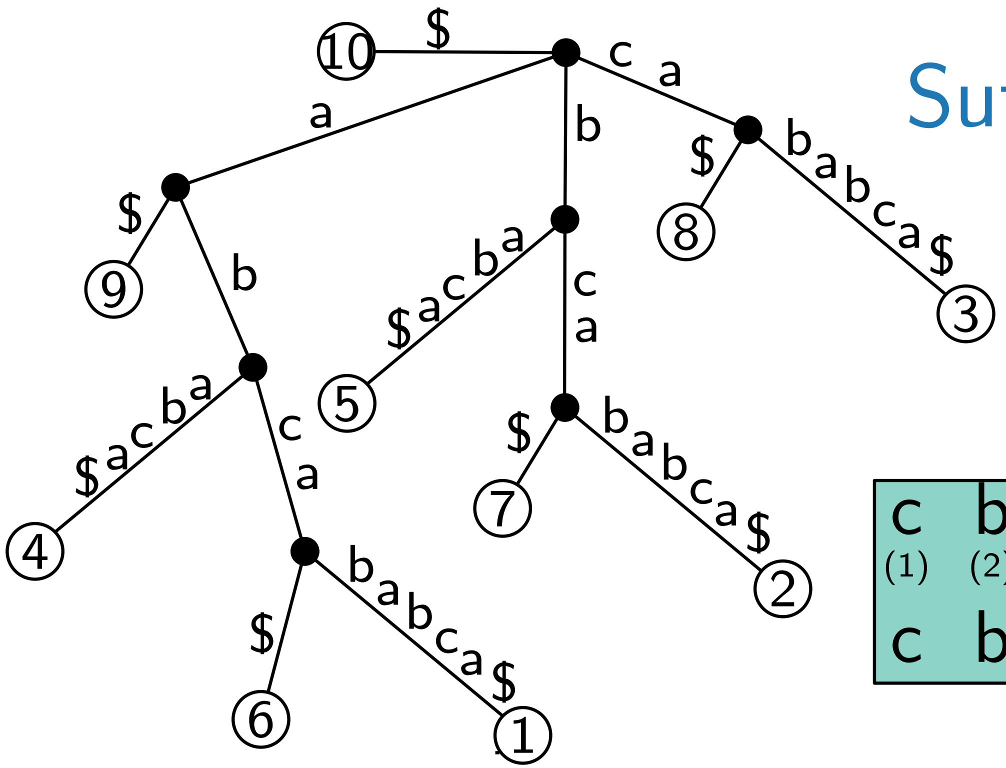
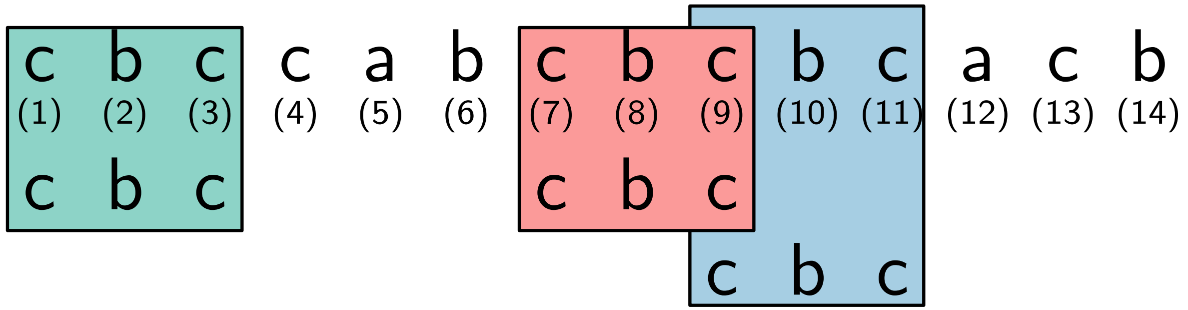
Advanced Algorithms

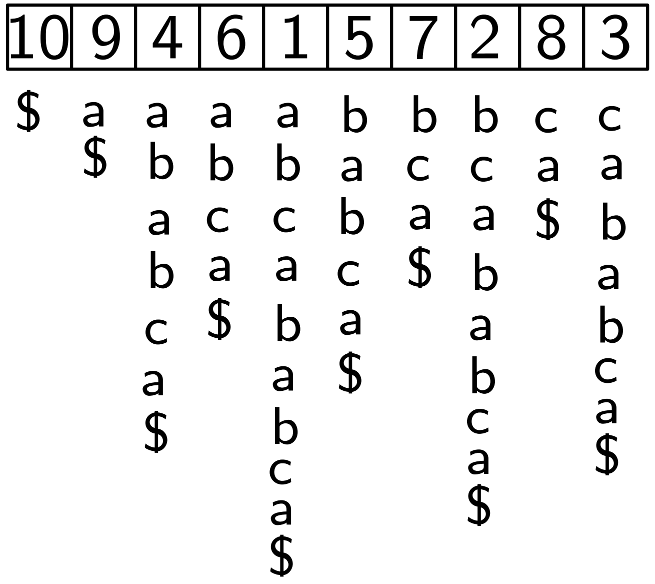
String Matching

Suffix Trees & Suffix Arrays

Johannes Zink · WS22







The "Ctrl+F" Problem

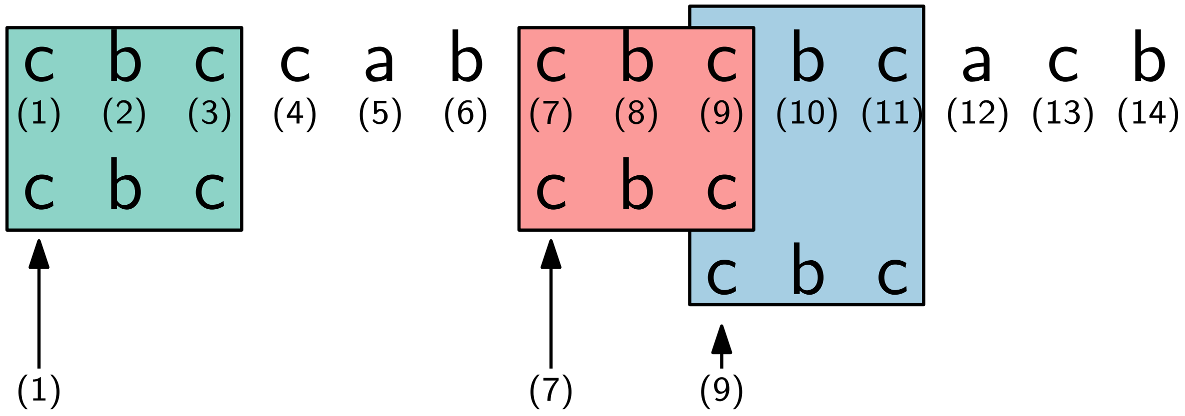
String Matching

Input: Strings T (text) and P (pattern) over an alphabet Σ s.t. $|P|, |\Sigma| \leq |T|$.

Task: Find all occurrences of P in T.

Example:

$\Sigma = \{a,b,c\}$ $P =$ cbc $T =$



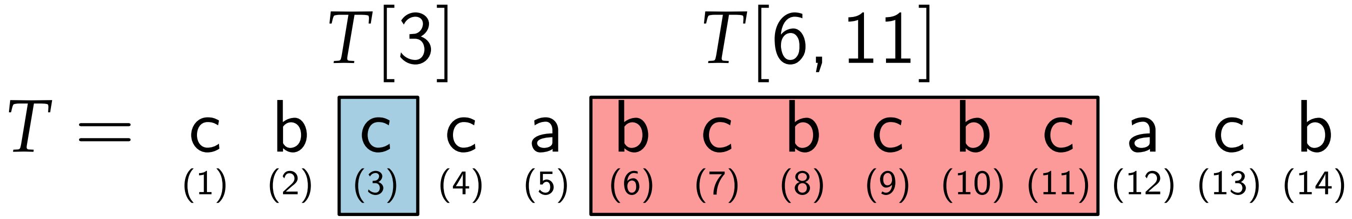
P occurs in T at positions 1, 7, and 9.

Applications:

- Searching a text document / e-book.
- Searching a particular pattern in a DNA sequence.
- Internet search engines: determine whether a page is relavent to the user query.

Notation

We assume T and P to be encoded as arrays with $n = |T|$ entries $T[1], T[2], \ldots, T[n]$ and $m = |P|$ entries $P[1], P[2], \ldots, P[m]$, respectively.

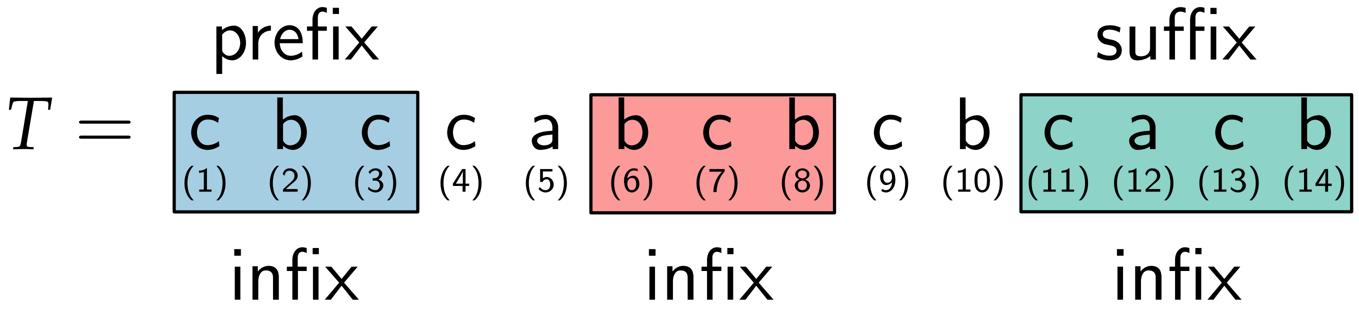


$T[3]$ $T[6,11]$

$T =$	c	b	c	c	a	b	c	b	c	b	c	a	c	b
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)

$T[i, j]$ with $1 \leq i \leq j \leq n$ denotes the substring of T formed by $T[i], T[i+1], \ldots, T[j]$.

Each substring $T[i, j]$ is called an **infix** of T.
If $i = 1$, then $T[i, j]$ is also called **prefix** of T.
If $j = n$, then $T[i, j]$ is also called **suffix** of T.



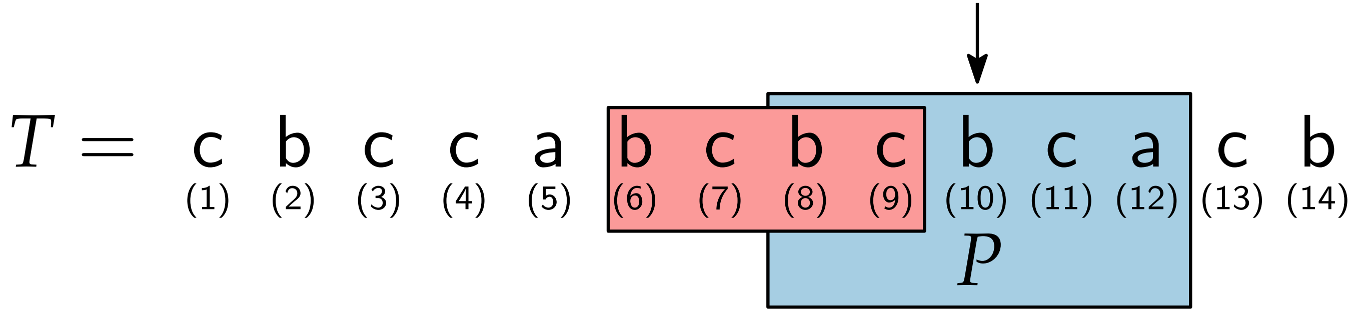
prefix suffix

$T =$	c	b	c	c	a	b	c	b	c	b	c	a	c	b
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)

infix infix infix

Algorithmic Complexity

Occurrences of (prefixes of) P may overlap.
$\Rightarrow$ A simple left-to-right traversal of T is not sufficient to find all occurrences of P!



$T =$	c	b	c	c	a	b	c	b	c	b	c	a	c	b
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)

Observation. String Matching can be solved in $\mathcal{O}(nm)$ time.

Theorem. String Matching can be solved in $\mathcal{O}(n+m)$ time, and this time bound is optimal. [Knuth, Morris, Pratt'77]

Often, many queries $P_1, P_2, P_3, \ldots$ are performed on the same text T.

Our goal: Design a data structure to store T such that each query P_i can be answered in time independent of n.

We will see two such data structures: **suffix trees** and **suffix arrays**.

Suffix Trees (I)

$T = \text{a b c a b a b c a}$

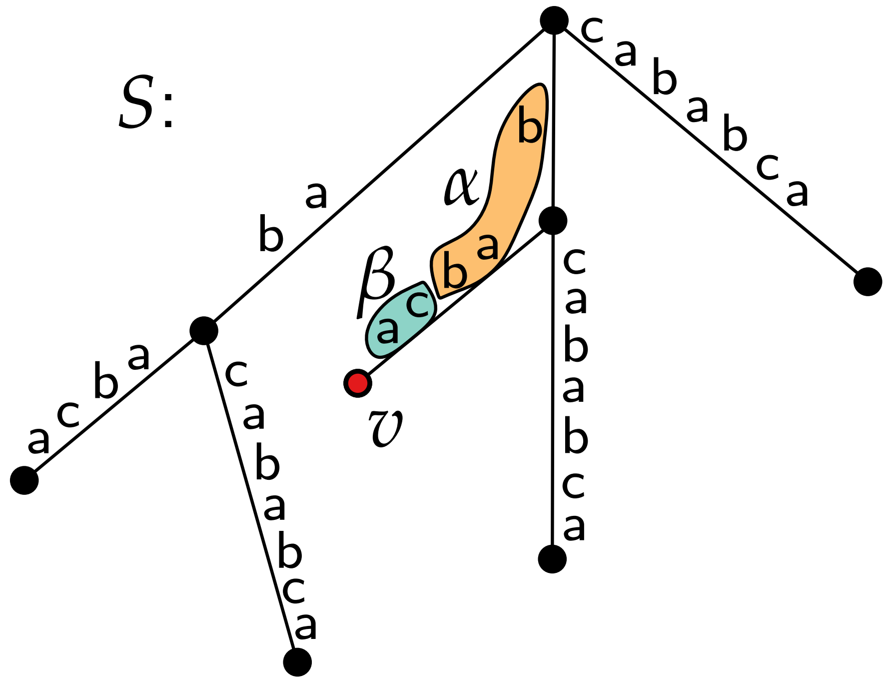
Idea: Represent T as a search tree.

A **Σ-tree** is a rooted tree $S = (V, E)$ whose edges are labeled with strings over Σ such that for each $v \in V$

- the labels of the edges that lead to the children of v start with pairwise distinct elements of Σ;
- if v is not the root, then v has $\neq 1$ children.

Notation:

- $\overline{v}$ = concatenation of the labels encountered on the path from the root to v;
- $d(v) = |\overline{v}|$ is the **string depth** of v;
- S **contains** a string α if there is a $v \in V$ and a (maybe empty) string β such that $\overline{v} = \alpha\beta$;
- words(S) = set of all strings contained in S.



$\overline{v} = babca$

$d(v) = |\overline{v}| = 5$

S contains $\alpha = \boxed{\text{b a b}}$ since there is a $v \in V$ with $\overline{v} = \alpha\beta$ where $\beta = \boxed{\text{c a}}$.

Suffix Trees (II)

A **suffix tree** S of T is a Σ-tree that contains exactly the infixes of T, that is, $\text{words}(S) = \{T[i,j] \mid 1 \leq i \leq j \leq n\}$.

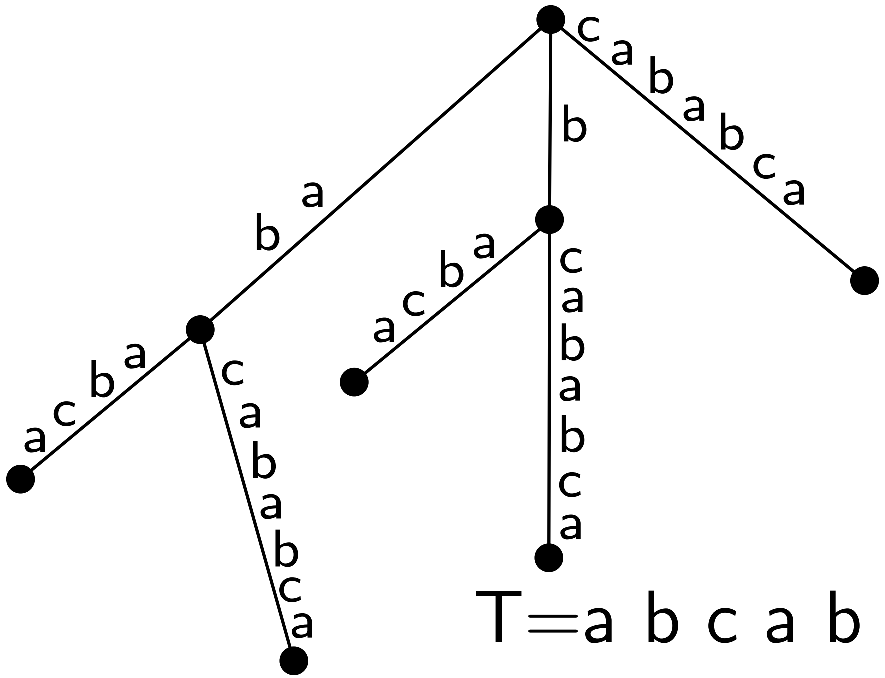
Lemma. For each leaf v of S, the infix $\overline{v}$ is a suffix of T.

Proof. Denote $\overline{v} = T[i,j]$ and assume $j < n$.

$\overline{v}$ is a prefix of $T[i,n]$. Let u be a vertex such that $T[i,n]$ is a prefix of $\overline{u}$.

$\Rightarrow$ the path from the root to v is a subpath of the path from the root to u.

$\Rightarrow$ v is not a leaf; a contradiction. $\square$



T=a b c a b a b c a

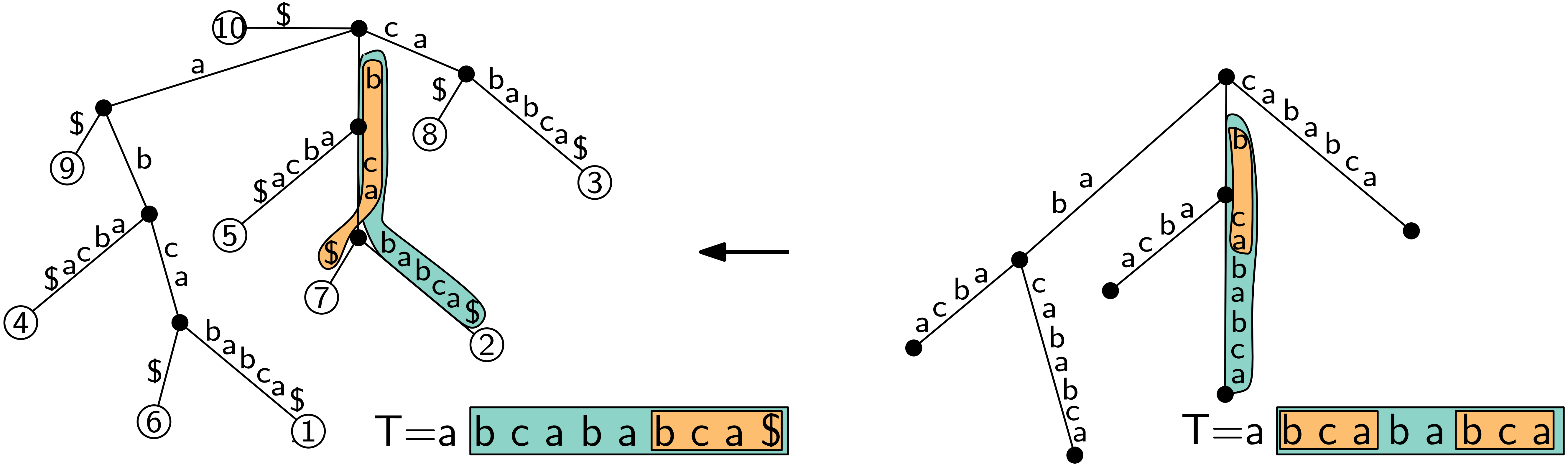
Suffix Trees (II)

A **suffix tree** S of T is a Σ-tree that contains exactly the infixes of T, that is, $\text{words}(S) = \{T[i,j] \mid 1 \leq i \leq j \leq n\}$.

Lemma. For each leaf v of S, the infix $\overline{v}$ is a suffix of T.

Remark. The converse is not true since a suffix can be a prefix of another suffix.

Fix: Append a symbol $\$ \notin \Sigma$ to T $\Rightarrow$ the leaves correspond bijectively to the suffixes.



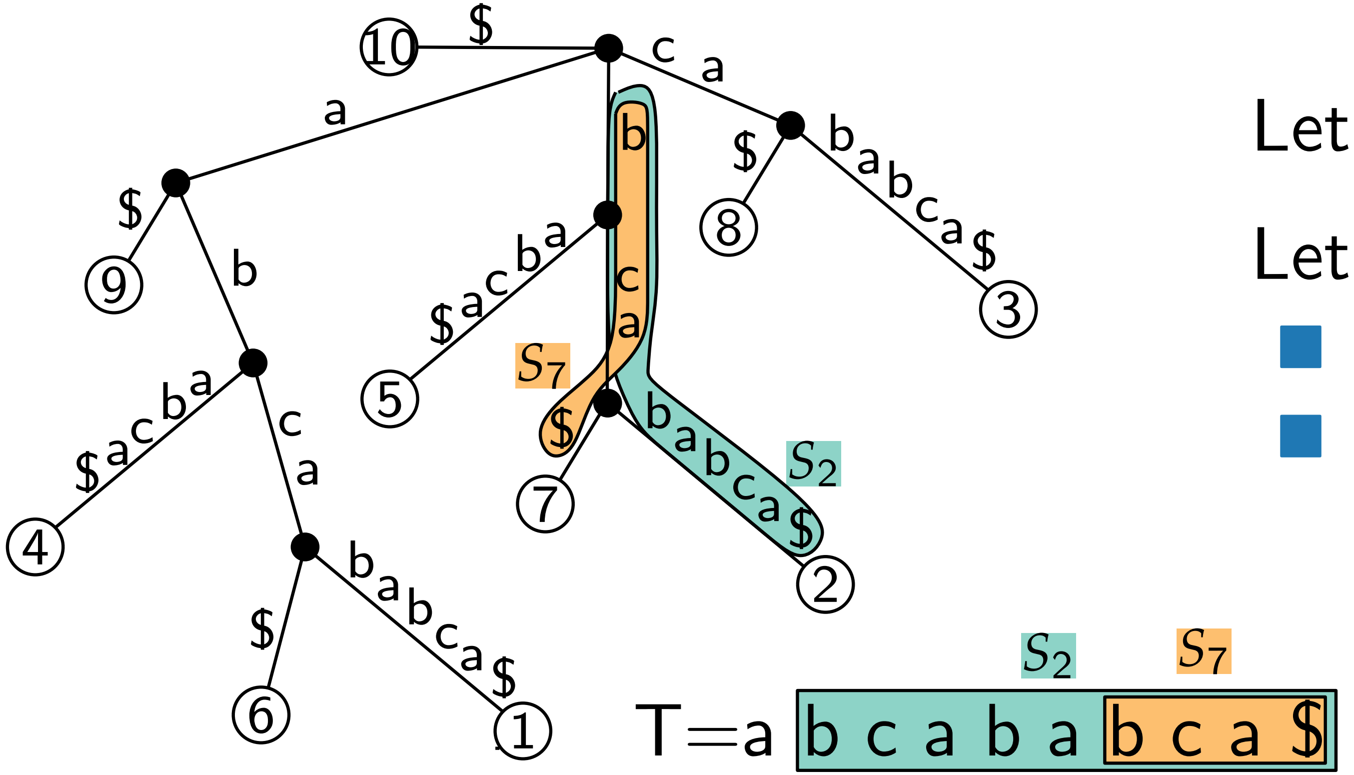
Suffix Trees (II)

A **suffix tree** S of T is a Σ-tree that contains exactly the infixes of T, that is, $\text{words}(S) = \{T[i,j] \mid 1 \leq i \leq j \leq n\}$.

Lemma. For each leaf v of S, the infix $\overline{v}$ is a suffix of T.

Remark. The converse is not true since a suffix can be a prefix of another suffix.

Fix: Append a symbol $\$ \notin \Sigma$ to T $\Rightarrow$ the leaves correspond bijectively to the suffixes.



Let i denote the leaf of S where $\bar{i} = T[i,n]$.

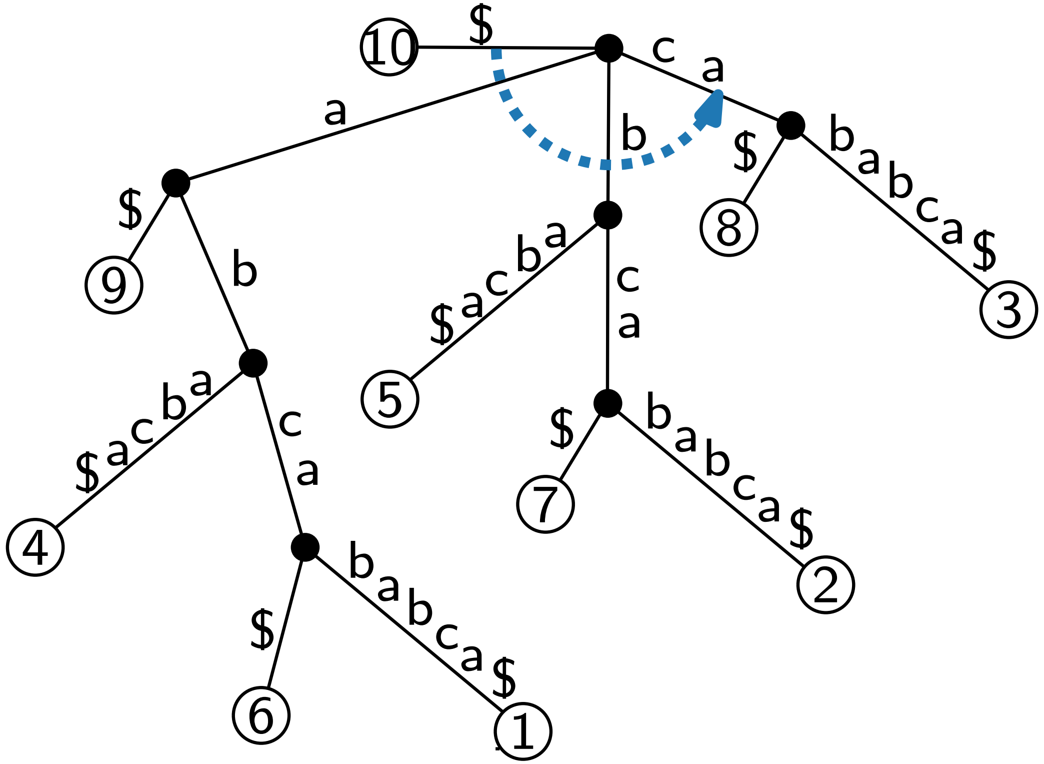
Let S_i denote

- the i-th suffix $T[i,n]$ of T;
- the path from the root of S to i.

Suffix Trees (III)

Implementation details:

- Each edge is labeled with an infix $T[i,j]$. It suffices to store the indices i and j.
  $\Rightarrow S$ requires $\mathcal{O}(n)$ space since #leaves $=$ #suffixes $=$ n.
- At each vertex v with k children, the edges leading to these children are stored in an array of length k sorted by the first letter of their labels.

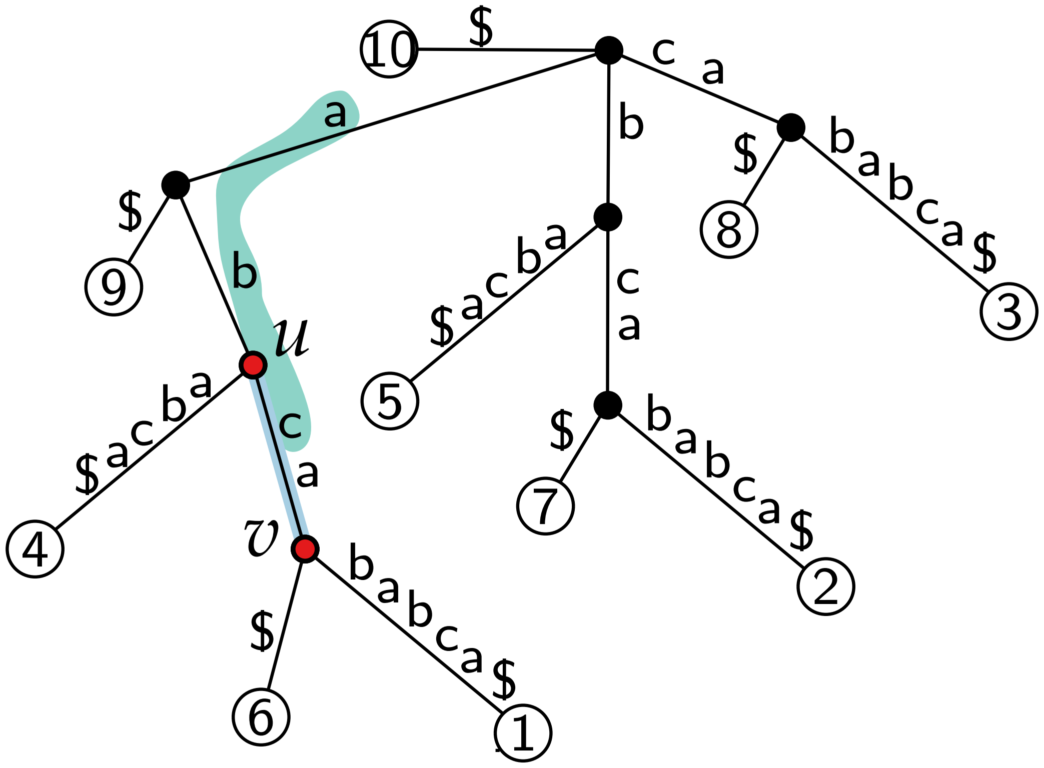


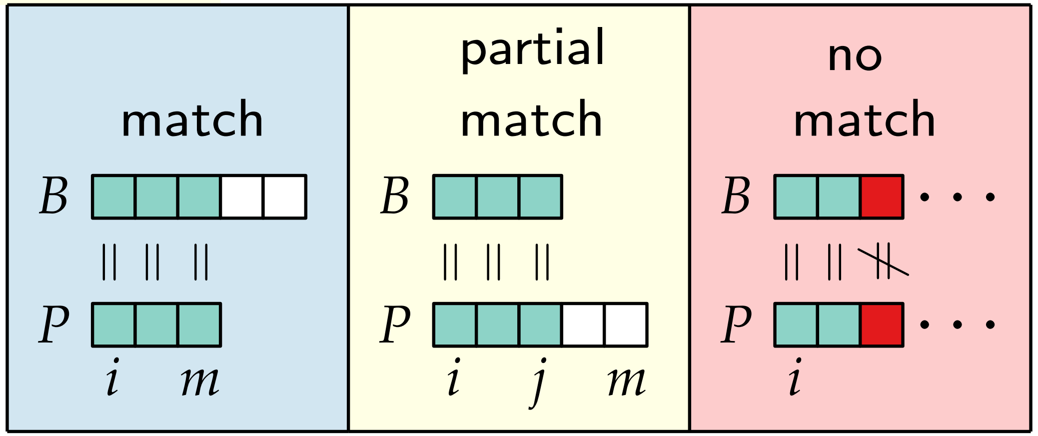
→ allows for binary search!

Searching in Suffix Trees

```
SEARCH($S$,$P$)
  $u \leftarrow$ root of $S$
  $i \leftarrow 1$
  while $u$ is not a leaf do
      Search edge $e = (u, v)$ whose label $B$ starts with $P[i]$.
      if $e$ does not exist then
          return “no match“
      Compare $B$ with $P[i, m]$
      if $P[i, m]$ is prefix of $B$ then
          return the indices of all leaves in the subtree rooted at $v$
      else if $P[i, j] = B$ for some $j < m$ then
          $i \leftarrow j + 1$
          $u \leftarrow v$
      else
          return “no match“
  return “no match“
```

T=a b c a b a b c a





Beispiel: $P =$ a b c

1 2 3

i

Searching in Suffix Trees

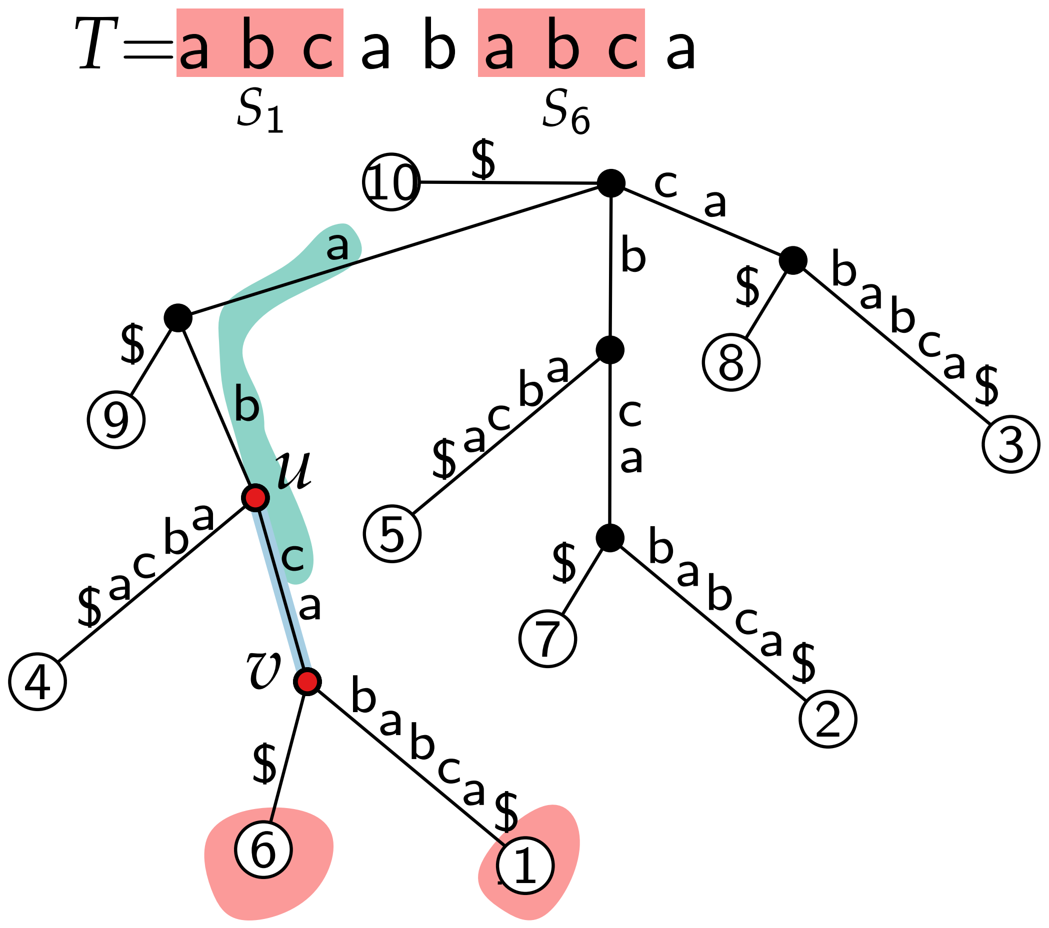
```
SEARCH($S$,$P$)
  $u \leftarrow$ root of $S$
  $i \leftarrow 1$
  while $u$ is not a leaf do
      Search edge $e = (u, v)$ whose label $B$ starts with $P[i]$.
      if $e$ does not exist then
          return "no match"
      Compare $B$ with $P[i, m]$
      if $P[i, m]$ is prefix of $B$ then
          return the indices of all leaves in the subtree rooted at $v$
      else if $P[i, j] = B$ for some $j < m$ then
          $i \leftarrow j + 1$
          $u \leftarrow v$
      else
          return "no match"
  return "no match"
```

$\leq m$ iterations

$\mathcal{O}(\log|\Sigma|)$

m comparisons in total

$\mathcal{O}(k)$ in total



Beispiel: $P =$ a b c
1 2 3
i

This is a parameterized, **output-sensitive** algorithm!

Correctness. Each occurrence of P is a prefix of exactly one suffix of T. We report all suffixes with P as a prefix.

Running time. $\mathcal{O}(m \log|\Sigma| + k)$ where k is the number of leaves in the subtree rooted at v.

Constructing Suffix Trees

Task. Given a string T with $n = |T|$ over alphabet Σ, construct a suffix tree S for T.

Idea. Construct Σ-trees $N_1, N_2, \dots, N_n$ s.t. N_i contains the suffixes $S_1, S_2, \dots, S_i$.

Initialization. N_1 consists of a single edge labeled S_1.

Constructing N_{i+1} from N_i. Search the longest prefix P of S_{i+1} contained in N_i.

Case 1. P ends in the middle of an edge e. Subdivide e and attach a new edge.

Case 2. P ends at a vertex v. Attach a new edge, then re-sort the neighbors of v.

Running time.

$$\mathcal{O}\Big(\big((n-1)+(n-2)+\dots+1\big)\log|\Sigma| + n|\Sigma|\Big) \subseteq \mathcal{O}(n^2 \log|\Sigma|)$$

It is also possible to construct suffix trees in $\mathcal{O}(n)$ time

- directly, e.g., with an algorithm by Farach (1997); or
- indirectly, by first constructing a **suffix array**, e.g., with an algorithm by Kärkkäinen and Sanders (2003).

Suffix Arrays

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

$A =$	10	9	4	6	1	5	7	2	8	3
	\$	a\$	ababca\$	abca\$	abcababca\$	babca\$	bca\$	bcababca\$	ca\$	cababca\$

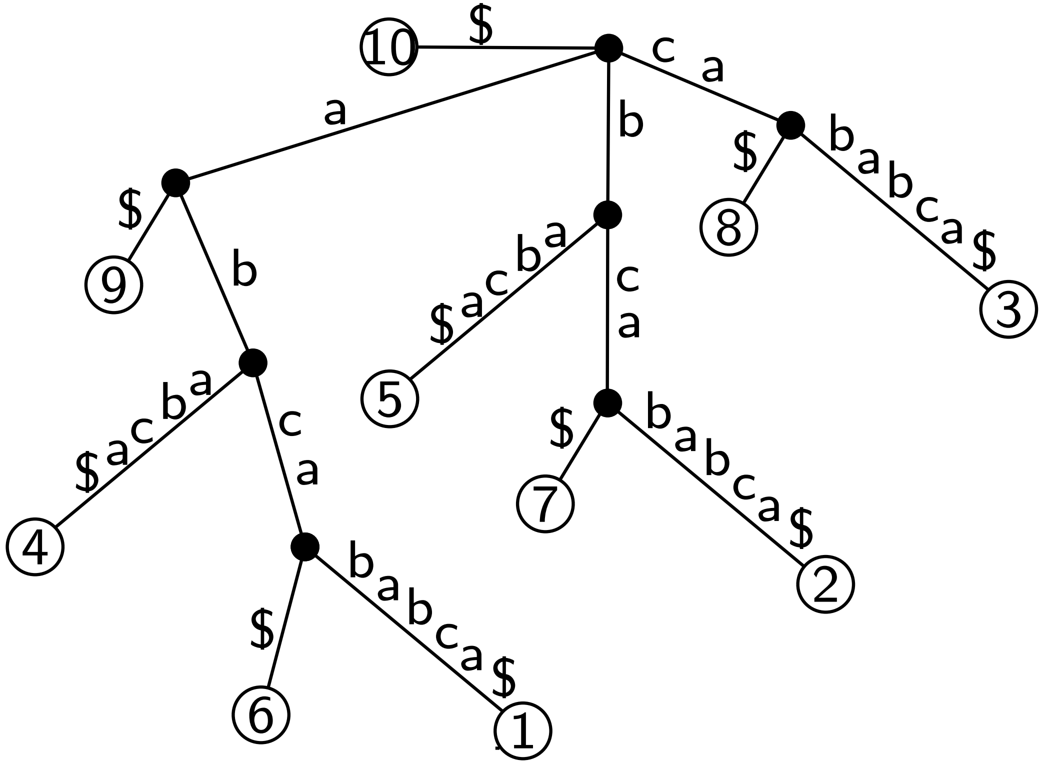
A **suffix array** A of a text T with $n = |T|$ stores a permutation of the indices $\{1, 2, \ldots, n\}$ s.t. $S_{A[i]}$ **is the i-th smallest suffix** of T in lexicographical order.

$S_{A[i-1]} < S_{A[i]}$ for each $1 < i \leq n$

Convention. \$ is the smallest letter.

Properties.

- The entries of A correspond to a lexicographical sorting of the suffixes of T.
- The entries of A corresponds to the order in which the leaves of a suffix tree S of T are encoutered by a DFS that chooses the next edge according to the lexicographical order.



Searching in Suffix Arrays

Observation. The occurrences of a pattern P in T form an interval in A.

Idea. Find the left and the right boundary of the interval via two binary searches.

Report all entries in the interval!

```
FindLeftBoundary($P$, $A$)
  $\ell \leftarrow 1$ // left index of candidates
  $r \leftarrow A$.length // right index of candidates
  while $\ell < r$ do
      $i \leftarrow \ell + \lfloor (r - \ell)/2 \rfloor$
      if $P > S_{A[i]}[1, m]$ then
          $\ell \leftarrow i + 1$ // continue w/ right half
      else
          $r \leftarrow i$ // continue w/ left half
  if $P$ is no prefix of $A[\ell]$ then
      return "no match"
  return $\ell$
```

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

$A =$

10	9	4	6	1	5	7	2	8	3
$	a$	ababca$	abca$	abcababca$	babca$	bca$	bcababca$	ca$	cababca$

$P =$ a b

Searching in Suffix Arrays

Observation. The occurrences of a pattern P in T form an interval in A.

Idea. Find the left and the right boundary of the interval via two binary searches.

Report all entries in the interval!

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

$A =$

10	9	4	6	1	5	7	2	8	3
$	a$	ababca$	abca$	abcababca$	babca$	bca$	bcababca$	ca$	cababca$

```
FindRightBoundary($A, P$)
  $\ell \leftarrow 1$ // left index of candidates
  $r \leftarrow A$.length // right index of candidates
  while $r > \ell$ do
      $i \leftarrow \ell + \lceil (r - \ell)/2 \rceil$
      if $P < S_{A[i]}[1, m]$ then
          $r \leftarrow i - 1$ // continue w/ left half
      else
          $\ell \leftarrow i$ // continue w/ right half
  if $P$ is no prefix of $A[r]$ then
      return “no match“
  return $r$
```

$P =$ a b

Each lexicographic comparison can be done in time $\mathcal{O}(m)$.

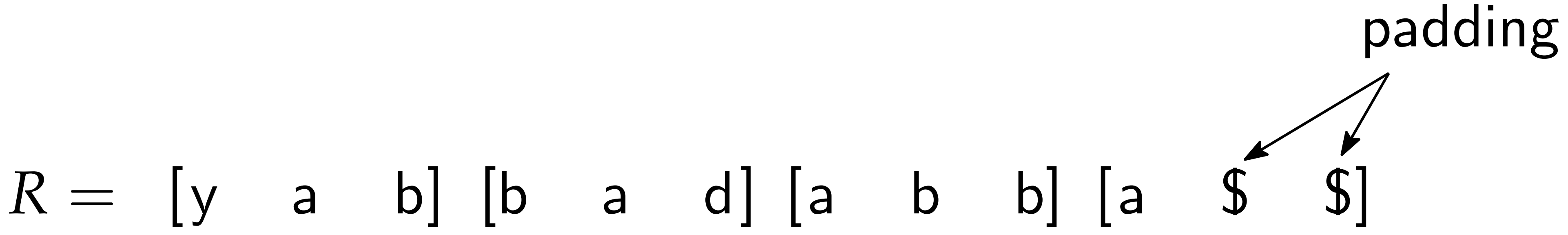
$\Rightarrow$ The k occurrences of P can be found in $\mathcal{O}(m \log n + k)$ time.

Constructing Suffix Arrays – First Attempt

Task. Given a string T with $n = |T|$ over alphabet Σ, construct a suffix array A for T.

Idea.

- If $n \in \mathcal{O}(1)$ use brute-force.
- Otherwise, dissect T into triples.
- Interpret the triples as letters over an alphabet $\Sigma' \subseteq \Sigma^3$.
- Interpret T as a string R over Σ' with $|R| = \lceil n/3 \rceil$.
- Recurse!



$$R = [\mathrm{y}\ \ \mathrm{a}\ \ \mathrm{b}]\ [\mathrm{b}\ \ \mathrm{a}\ \ \mathrm{d}]\ [\mathrm{a}\ \ \mathrm{b}\ \ \mathrm{b}]\ [\mathrm{a}\ \ \$\ \ \$]$$

Problem. But how can a suffix array for R be used to create a suffix array for T?

Constructing Suffix Arrays – Overview

Shortened notation: $T = t_0 t_1 \dots t_{n-1}$ and $x \equiv z(y)$ is a shorthand for $x \mod y = z$.

	0	1	2	3	4	5	6	7	8	9	10	11
$T=$	y	a	b	b	a	d	a	b	b	a	d	o

$\mathcal{S}_0$ = suffixes with index $i \equiv 0(3)$

$\mathcal{S}_1$ = suffixes with index $i \equiv 1(3)$

$\mathcal{S}_2$ = suffixes with index $i \equiv 2(3)$

```
ConstructSuffixArray($T$)
    if $n \in \mathcal{O}(1)$ then
        construct $A$ in $\mathcal{O}(1)$ time.
    else
        sort $\mathcal{S}_1 \cup \mathcal{S}_2$ into an array $A_{12}$
        use $A_{12}$ to sort $\mathcal{S}_0$ into an array $A_0$
        merge $A_{12}$ with $A_0$
```

using the idea from the previous slide!

For simplicity, we assume $n \equiv 0(3)$.

$\mathcal{S}(T)$ = suffixes of T =

S_0	y a b b a d a b b a d o
S_1	a b b a d a b b a d o
S_2	b b a d a b b a d o
S_3	b a d a b b a d o
S_4	a d a b b a d o
S_5	d a b b a d o
S_6	a b b a d o
S_7	b b a d o
S_8	b a d o
S_9	a d o
S_{10}	d o
S_{11}	o

Step 1: Sorting $\mathcal{S}_1 \cup \mathcal{S}_2$

Shortened notation: $T = t_0 t_1 \dots t_{n-1}$ and $x \equiv z(y)$ is a shorthand for $x \mod y = z$.

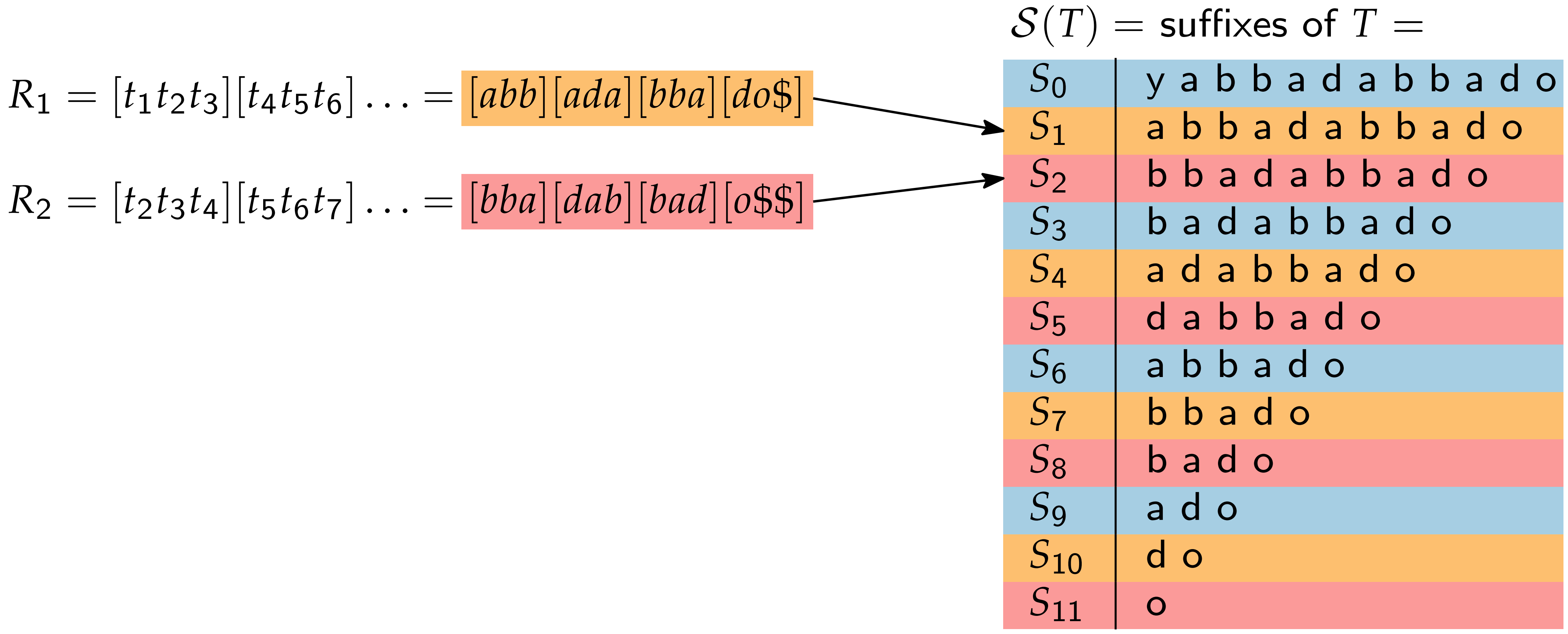
Dissect $\mathcal{S}_1$ and $\mathcal{S}_2$ into triples and concatenate them:

$\mathcal{S}_0$ = suffixes with index $i \equiv 0(3)$

$\mathcal{S}_1$ = suffixes with index $i \equiv 1(3)$

$\mathcal{S}_2$ = suffixes with index $i \equiv 2(3)$

$R=$ [abb][ada][bba][do\$][bba][dab][bad][o\$\$]



$R_1 = [t_1 t_2 t_3][t_4 t_5 t_6] \dots = [abb][ada][bba][do\$]$

$R_2 = [t_2 t_3 t_4][t_5 t_6 t_7] \dots = [bba][dab][bad][o\$\$]$

$\mathcal{S}(T)$ = suffixes of T =

S_0	y a b b a d a b b a d o
S_1	a b b a d a b b a d o
S_2	b b a d a b b a d o
S_3	b a d a b b a d o
S_4	a d a b b a d o
S_5	d a b b a d o
S_6	a b b a d o
S_7	b b a d o
S_8	b a d o
S_9	a d o
S_{10}	d o
S_{11}	o

Step 1: Sorting $\mathcal{S}_1 \cup \mathcal{S}_2$

$S_i < S_j \quad \Leftrightarrow \quad S_i\$ < S_j\$ \quad \Leftrightarrow \quad S_i\$ \ldots < S_j\$ \ldots$
since the positions of the first \$ symbols in the strings $S_k(R)$ are pairwise distinct.

Shortened notation: $T = t_0 t_1 \ldots t_{n-1}$ and $x \equiv z(y)$ is a shorthand for $x \mod y = z$.

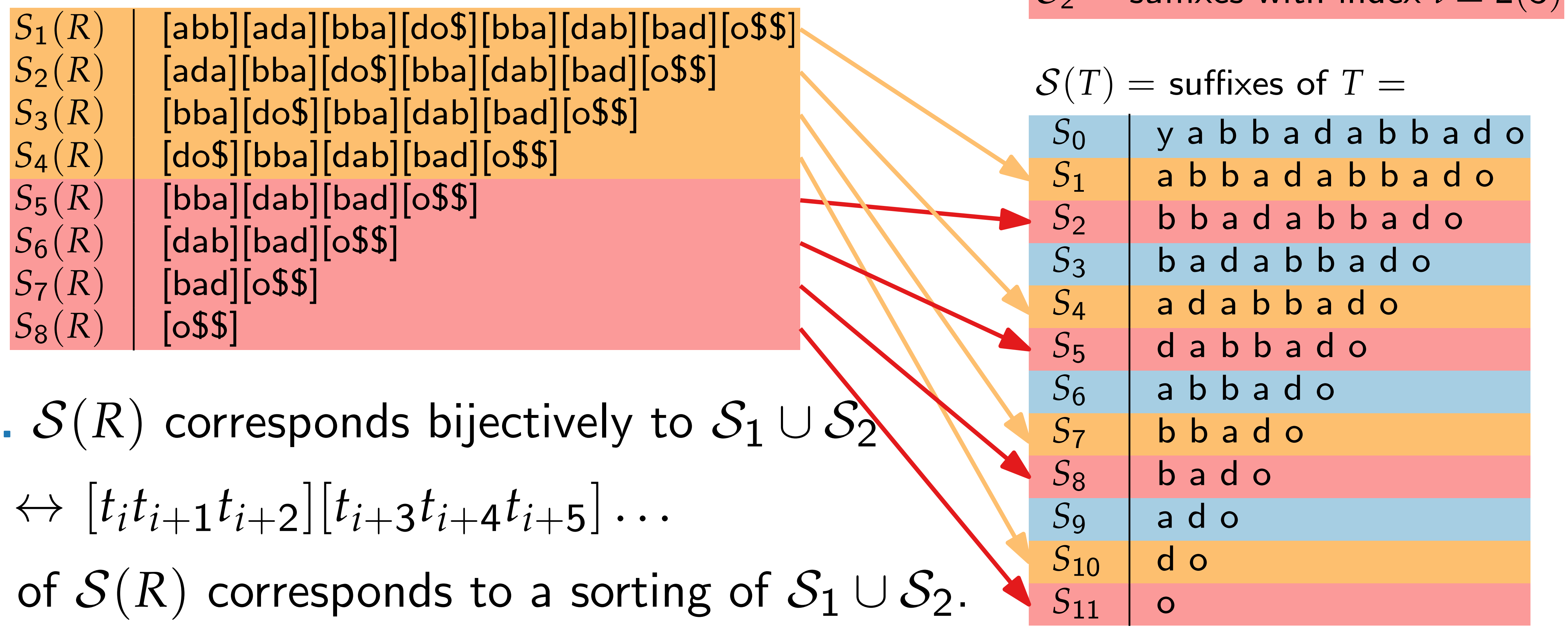
Dissect $\mathcal{S}_1$ and $\mathcal{S}_2$ into triples and concatenate them:

$\mathcal{S}_0$ = suffixes with index $i \equiv 0(3)$

$\mathcal{S}_1$ = suffixes with index $i \equiv 1(3)$

$\mathcal{S}_2$ = suffixes with index $i \equiv 2(3)$

$R =$ [abb][ada][bba][do\$][bba][dab][bad][o\$\$]



$\mathcal{S}(R) =$

$S_1(R)$	[abb][ada][bba][do\$][bba][dab][bad][o\$\$]
$S_2(R)$	[ada][bba][do\$][bba][dab][bad][o\$\$]
$S_3(R)$	[bba][do\$][bba][dab][bad][o\$\$]
$S_4(R)$	[do\$][bba][dab][bad][o\$\$]
$S_5(R)$	[bba][dab][bad][o\$\$]
$S_6(R)$	[dab][bad][o\$\$]
$S_7(R)$	[bad][o\$\$]
$S_8(R)$	[o\$\$]

$\mathcal{S}(T)$ = suffixes of T =

S_0	y a b b a d a b b a d o
S_1	a b b a d a b b a d o
S_2	b b a d a b b a d o
S_3	b a d a b b a d o
S_4	a d a b b a d o
S_5	d a b b a d o
S_6	a b b a d o
S_7	b b a d o
S_8	b a d o
S_9	a d o
S_{10}	d o
S_{11}	o

Observation. $\mathcal{S}(R)$ corresponds bijectively to $\mathcal{S}_1 \cup \mathcal{S}_2$

$$S_i \leftrightarrow [t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \ldots$$

and a sorting of $\mathcal{S}(R)$ corresponds to a sorting of $\mathcal{S}_1 \cup \mathcal{S}_2$.

Sorting $\mathcal{S}(R)$

Sort the "letters" (= triples) of R via RadixSort. This can be done in time

$$\mathcal{O}(3(\tfrac{2}{3}n + |\Sigma|)) \subseteq \mathcal{O}(n)$$

#digits #objects alphabet size

ConstructSuffixArray(R')

Replace each triple of R with its rank → string R' with alphabet size $\leq \frac{2}{3}n \leq n$.

A sorting of $\mathcal{S}(R')$ corresponds to a sorting of $\mathcal{S}(R)$ and can be obtained recursively.

$R=$ [abb][ada][bba][do\$][bba][dab][bad][o\$\$]

$R'=$ 1 2 4 6 4 5 3 7

Rank	**triple**
1	[abb]
2	[ada]
3	[bad]
4	[bba]
5	[dab]
6	[do\$]
7	[o\$\$]

$\mathcal{S}(R)=$

$S_1(R)$	[abb][ada][bba][do\$][bba][dab][bad][o\$\$]
$S_2(R)$	[ada][bba][do\$][bba][dab][bad][o\$\$]
$S_3(R)$	[bba][do\$][bba][dab][bad][o\$\$]
$S_4(R)$	[do\$][bba][dab][bad][o\$\$]
$S_5(R)$	[bba][dab][bad][o\$\$]
$S_6(R)$	[dab][bad][o\$\$]
$S_7(R)$	[bad][o\$\$]
$S_8(R)$	[o\$\$]

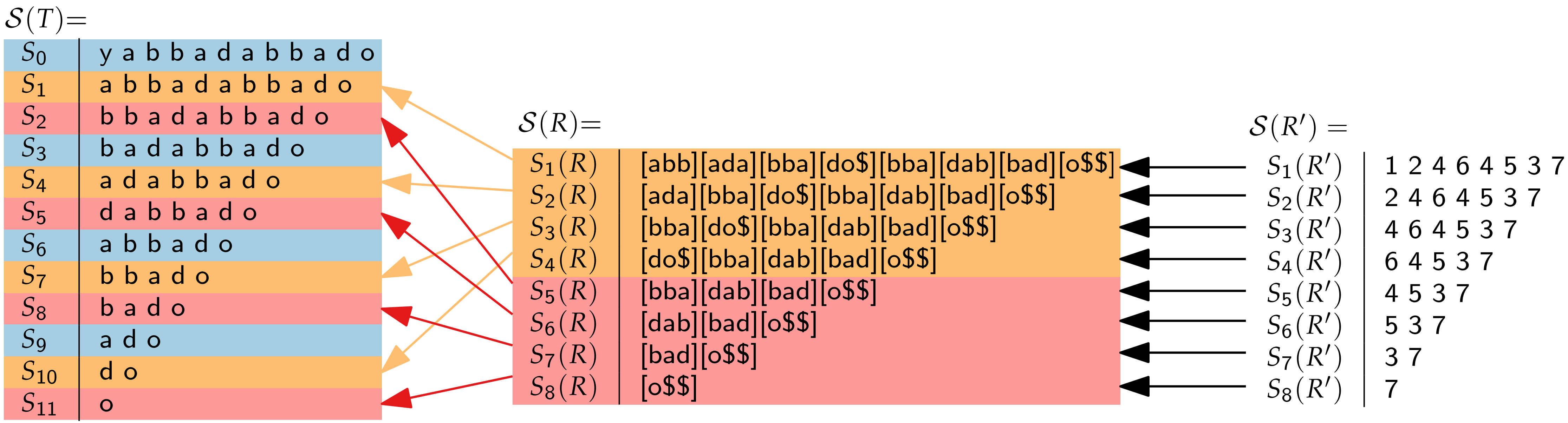
$\mathcal{S}(R') =$

$S_1(R')$	1 2 4 6 4 5 3 7
$S_2(R')$	2 4 6 4 5 3 7
$S_3(R')$	4 6 4 5 3 7
$S_4(R')$	6 4 5 3 7
$S_5(R')$	4 5 3 7
$S_6(R')$	5 3 7
$S_7(R')$	3 7
$S_8(R')$	7

Summary of Step 1

Rank	triple
1	[abb]
2	[ada]
3	[bad]
4	[bba]
5	[dab]
6	[do$]
7	[o$$]

Full example.



$\mathcal{S}(T)=$

S_0	yabbadabbado
S_1	abbadabbado
S_2	bbadabbado
S_3	badabbado
S_4	adabbado
S_5	dabbado
S_6	abbado
S_7	bbado
S_8	bado
S_9	ado
S_{10}	do
S_{11}	o

$\mathcal{S}(R)=$

$S_1(R)$	[abb][ada][bba][do$][bba][dab][bad][o$$]
$S_2(R)$	[ada][bba][do$][bba][dab][bad][o$$]
$S_3(R)$	[bba][do$][bba][dab][bad][o$$]
$S_4(R)$	[do$][bba][dab][bad][o$$]
$S_5(R)$	[bba][dab][bad][o$$]
$S_6(R)$	[dab][bad][o$$]
$S_7(R)$	[bad][o$$]
$S_8(R)$	[o$$]

$\mathcal{S}(R') =$

$S_1(R')$	1 2 4 6 4 5 3 7
$S_2(R')$	2 4 6 4 5 3 7
$S_3(R')$	4 6 4 5 3 7
$S_4(R')$	6 4 5 3 7
$S_5(R')$	4 5 3 7
$S_6(R')$	5 3 7
$S_7(R')$	3 7
$S_8(R')$	7

A_{12}

1	S_1	abbadabbado	$S_1(R')$	1 2 4 6 4 5 3 7
2	S_4	adabbado	$S_2(R')$	2 4 6 4 5 3 7
3	S_8	bado	$S_7(R')$	3 7
4	S_2	bbadabbado	$S_5(R')$	4 5 3 7
5	S_7	bbado	$S_3(R')$	4 6 4 5 3 7
6	S_5	dabbado	$S_6(R')$	5 3 7
7	S_{10}	do	$S_4(R')$	6 4 5 3 7
8	S_{11}	o	$S_8(R')$	7

Running time.

$$T_1(n) = \mathcal{O}(n) + T(\tfrac{2}{3}n)$$

where $T(n)$ is the time to execute ConstructSuffixArray on a string of length n.

Step 2: Sorting $\mathcal{S}_0$

Shortened notation: $T = t_0 t_1 \ldots t_{n-1}$ and $x \equiv z(y)$ is a shorthand for $x \mod y = z$.

	0	1	2	3	4	5	6	7	8	9	10	11
$T=$	y	a	b	b	a	d	a	b	b	a	d	o

$\mathcal{S}_0$ = suffixes with index $i \equiv 0(3)$

$\mathcal{S}_1$ = suffixes with index $i \equiv 1(3)$

$\mathcal{S}_2$ = suffixes with index $i \equiv 2(3)$

Each $S_i \in \mathcal{S}_0$ can be written as (t_i, S_{i+1}) s.t. $S_{i+1} \in \mathcal{S}_1$.

Observation. Let $S_i, S_j \in \mathcal{S}_0$. Then $S_i < S_j$ if and only if

- $t_i < t_j$; or
- $t_i = t_j$ and $S_{i+1} < S_{j+1}$.

$\Rightarrow$ $\mathcal{S}_o$ can be sorted by sorting all tuples (t_i, S_{i+1}) with $i \equiv 0(3)$. This can be done via RADIXSORT in $\mathcal{O}(n)$ time since the ordering of the entries in $\mathcal{S}_1$ is already implicit in A_{12}.

$\mathcal{S}(T)$ = suffixes of T =

S_0	y a b b a d a b b a d o
S_1	a b b a d a b b a d o
S_2	b b a d a b b a d o
S_3	b a d a b b a d o
S_4	a d a b b a d o
S_5	d a b b a d o
S_6	a b b a d o
S_7	b b a d o
S_8	b a d o
S_9	a d o
S_{10}	d o
S_{11}	o

Step 3: Merging A_{12} and A_0

Shortened notation: $T = t_0 t_1 \dots t_{n-1}$ and $x \equiv z(y)$ is a shorthand for $x \mod y = z$.

	0	1	2	3	4	5	6	7	8	9	10	11
T=	y	a	b	b	a	d	a	b	b	a	d	o

$\mathcal{S}_0$ = suffixes with index $i \equiv 0(3)$

$\mathcal{S}_1$ = suffixes with index $i \equiv 1(3)$

$\mathcal{S}_2$ = suffixes with index $i \equiv 2(3)$

Each $S_i \in \mathcal{S}_0$ can be written as (t_i, S_{i+1}) s.t. $S_{i+1} \in \mathcal{S}_1$ and as (t_i, t_{i+1}, S_{i+2}) s.t. $S_{i+2} \in \mathcal{S}_2$.

Observation. Let $S_i \in \mathcal{S}_0$.

- Let $S_j \in \mathcal{S}_1$. Then $S_i < S_j$ if and only if
  - $t_i < t_j$; or
  - $t_i = t_j$ and $S_{i+1} < S_{j+1}$ where $S_{j+1} \in \mathcal{S}_2$.
- Let $S_j \in \mathcal{S}_2$. Then $S_i < S_j$ if and only if
  - $t_i < t_j$; or
  - $t_i = t_j$ and $t_{i+1} < t_{j+1}$; or
  - $t_i t_{i+1} = t_j t_{j+1}$ and $S_{i+2} < S_{j+2}$ where $S_{j+2} \in \mathcal{S}_1$.

Since the ordering of $\mathcal{S}_1 \cup \mathcal{S}_2$ is already implicit in A_{12}, we can perform these comparisons in $\mathcal{O}(1)$ time.

$\Rightarrow$ A_{12} and A_0 can be merged as in MERGESORT to obtain A.

Construction of Suffix Arrays – Summary

```
$\textsc{ConstructSuffixArray}(T)$
    if $n \in \mathcal{O}(1)$ then
        construct $A$ in $\mathcal{O}(1)$ time.
    else
        sort $\mathcal{S}_1 \cup \mathcal{S}_2$ into an array $A_{12}$          $\mathcal{O}(n) + T(\frac{2}{3}n)$
        use $A_{12}$ to sort $\mathcal{S}_0$ into an array $A_0$            $\mathcal{O}(n)$
        merge $A_{12}$ with $A_0$                              $\mathcal{O}(n)$
```

Total running time:

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = \mathcal{O}(1) \\ \mathcal{O}(n) + T(\frac{2}{3}n), & \text{otherwise} \end{cases}$$

$$\overset{\text{Master Theorem}}{\Rightarrow} \quad T(n) \in \mathcal{O}(n)$$

Summary and Discussion

Let T be a string over an alphabet Σ where $n = |T|$.

Lemma. A suffix array for T can be used to compute an LCP (“longest common prefix”) array and a suffix tree of T in $\mathcal{O}(n)$ time. [without proof]

Theorem. A suffix tree for T can computed in $\mathcal{O}(n)$ time and space. It can be used to answer String Matching queries of length m in $\mathcal{O}(m \log |\Sigma| + k)$ time.

Theorem. A suffix array for T can computed in $\mathcal{O}(n)$ time and space. It can be used to answer String Matching queries of length m in $\mathcal{O}(m \log n + k)$ time.

Remark. The suffix array is a simpler and more compact alternative to the suffix tree.

The suffix tree (and the suffix array + LCP array) have several additional applications:

- Finding the longest repeated substring
- Finding the longest common substring of two strings.
- ...

Literature and References

The content of this presentation is based on Dorothea Wagner’s slides for a lecture on “String-Matching: Suffixbäume“ as part of the course “Algorithmen II“ held at KIT WS 13/14. Most figures and examples were taken from these slides.

Literature:

- Simple Linear Work Suffix Array Construction. Kärkkäinen and Sanders, ICALP’03
- Optimal suffix tree construction with large alphabets. Farach, FOCS’97
- Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Gusfield, 1999, Cambridge University Press