3. Exact Pattern Matching

String Matching

Given: pattern P, text T (P, T are strings)

Aim: find occurrences of P in T

Exmpl:

```
      1 2 3 4 5 6 7 8 9 10 ...
T =   A A T G C A T G C A .....
P =   A T G
```

P occurse on position 2, 6, ...

This has applications in:

- Bioinformatics (eg. sequence assembly where one may align fragments of your DNA to reference genome to get read of your DNA; also applicciations in finding repeated regions & many more
- general word psocemy
- internet search
- "fgrep" in Unix
- search for plagiarism
- subtask for e.g inexact matching

Basics: string $S = x_1 \dots \dots x_n$, $|S| = n$

$S[i..j] = x_i x_{i+1} \dots x_j$

$S(i) = x_i$

$S[1..j] \hat{=}$ prefix of S ending at j

$S[j..n] \hat{=}$ suffix of S starting at j

```
    1 2 3 4 5 6 7 6
S:  H O N O L U L U
```

$S[1..4]$ prefix

$S[7..6]$ suffix

$S(5) = L$

if $P(i) = T(k)$ for some i,k then they match
else mismatch.

Naïve Method

```
NAIVE (P,T)                                   // compare $T(i \dots i+|P|-1)$
                                                 with P for all $i = 1 \dots |T|-|P|+1$
    Occurences = $\emptyset$
    FOR ( $i = 1 \dots |T|-|P|+1$ )           // loop over "pos." in T
        FOR( $j = 1 \dots |P|$ )              // loop over P
            match = TRUE
            IF ( $P(j) \neq T(i+j-1)$ )       // compare characters
                match = FALSE
                break
        IF (match)
            add i to occurrences              // save pos. on which
                                                 P occurs
    RETURN occurrences.
```

Q: How often can P occur in T?

A: $|T| - |P| + 1$ times

$$\begin{array}{ll} & 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6 \\ T = & A\ A\ A\ A\ A\ A \\ P = & A\ A \end{array}$$

P occurs at pos. 1, 2, 3, 4, 5

= 5 times = $6 - 2 + 1$

Q: What is greatest Nr of character comparisons?

A: $|P| \cdot (|T| - |P| + 1)$ (worst case)

See example above! $2 \cdot 5 = 10$ comparisons

Q: What is least Nr of character comparisons?

A: $|T| - |P| + 1$ (best case)

$$\begin{array}{ll} T = & A\ A\ A\ A\ A\ A \\ P = & B\ A \end{array}$$

⇒ RUNTIME NAIVE-alg

$|T| - |P| + 1$ 1. loop

$|P|$ 2 loop

$$\Rightarrow |T|\,|P| - |P|^2 + 1 \underset{\overset{\uparrow}{|T| \geq |P|}}{\leq} |T|\,|P|$$

$\Rightarrow O(|T|\,|P|)$ time

look now to one of many linear-time algorithms, i.e. instead of $O(|P||T|)$ we have runtime $O(|P|+|T|)$

Z - algorithm

[fundamental preprocessing used in many alg]

General idea: pre-process P in $O(|P|)$ time to gain insight of internal structure of P

DEF: let S be a string & $i > 1$. (usually $S = P$ pattern)

$Z_i := Z_i(S) =$ length of longest substring in S that starts at position i & matches prefix of S

Exmpl:

i	1	2	3	4	5	6	7	8	9	10	11
S	a	a	b	c	a	a	b	x	a	a	z
Z_i	—	1	0	0	3	1	0	0	2	1	0

DEF (Z-Box): $\forall i > 1$ s.t. $Z_i > 0$, the Z-Box at i is the interval $[i, i + Z_i - 1]$

Exmpl:

i	1	2	3	4	5	6	7	8	9	10	11
S	a	a	b	c	a	a	b	x	a	a	z
z_i	—	1	0	0	3	1	0	0	2	1	0

$z_i > 0$ at pos. 2, 5, 6, 9, 10

$$
\begin{aligned}
\text{Z-Box at } 2 &= [2,\ 2+1-1] = [2,2] \\
\text{at } 5 &= [5,\ 5+3-1] = [5,7] \\
\text{at } 6 &= [6,\ 6+1-1] = [6,6] \\
\text{at } 9 &= [9,\ 9+2-1] = [9,10] \\
\text{at } 10 &= [10,\ 10+1-1] = [10,10]
\end{aligned}
$$

$$
\begin{array}{c|ccccccccccc}
i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
S & a & \underbrace{a} & b & c & \multicolumn{3}{c}{\underbrace{a\quad \underbrace{a}\quad b}} & x & \multicolumn{2}{c}{\underbrace{a\quad \underbrace{a}}} & z
\end{array}
$$

this intervals $[i,j]$ correspond to $S[i..j]$ = longest substring starting at $i>1$ & matches prefix of S

DEF $\forall i > 1$:

(r = right) r_i denotes right-most endpoint of Z-Boxes Z_j with $Z_j > 0$ & $j \le i$

(equ. to r_i = largest value of $j + Z_j - 1$ over all $1 < j \le i$ st $Z_j > 0$)

"store index j" : $l_i = j$ for j satisfying this ↗

(l = left)

(equ. to l_i = position of left-end of Z-Box ending in r_i)

↓

if MORE than one Z-Box ends in r_i then l_i can be chosen arbitrarily among those values

Formal: $r_i = \max\limits_{2 \le j \le i} \{ j + Z_j - 1 : Z_j > 0 \}$

i	1	2	3	4	5	6	7	8	9	10	11
S	a	a	b	c	a	a	b	x	a	a	z
r_i	–	2	2	2	7	7	7	7	10	10	10
l_i	–	2	2	2	5	5	5	5	9	9	9

right : $r_6 = \max \{ \underbrace{2 + 1 - 1}_{\text{for } j=2}, \underbrace{5 + 3 - 1}_{j=5}, \underbrace{6 + 1 - 1}_{j=6} \} = 7$

left : $l_6 = 5$

this essentially gives pos. of "large" Z-Boxes.

How to compute z_i, r_i, l_i efficiently?

main idea: iterative approach

start with z_2 (explicite comparison from left-to-right)

Assume $\forall i < k$ values z_i, r_i, l_i have been computed.

for k use: z_i, l_i, r_i ($i < k$) WITHOUT EXPLICITE character comparisons as much as possible.

Overview of steps of Z-alg

1 compute z_2 (explizit comparison of $S[1...n]$ with $S[2..n]$ until mismatch

IF ($z_2 > 0$) put $l_2 = 2$, $r_2 = z_2 + 1$
ELSE put $l_2 = r_2 = 0$

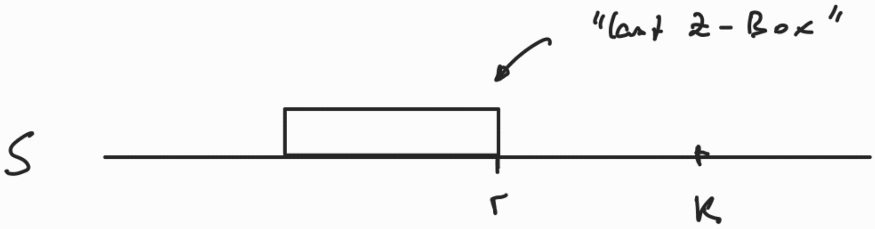
② k-th step ($k \geq 2$).

$\Rightarrow$ all z_i, l_i, r_i computed for all $i \leq k-1$

Compute z_k, l_k, r_k based on the folling cases.

$l := l_{k-1}$, $r := r_{k-1}$

Case 1: $r < k$



in this case compute z_k via explicit comparison of $S[k..n]$ & $S[1...n]$ until mismatch.

IF ($z_k > 0$) put $l_k = k$, $r_k = k + z_k - 1$

ELSE $l_k = l$, $r_k = r$

Exmpl:

i	1	2	3	4	5	6	7	8	9	10	11
S	a	a	b	c	a	a	b	x	a	a	z
r_i	–	2	2	2	7	7	7	7			
l_i	–	2	2	2	5	5	5	5			

$k = 9 \rightarrow r_8 = 8 < k = 9$ need to compare $S[9..11]$ with $S[1..n]$

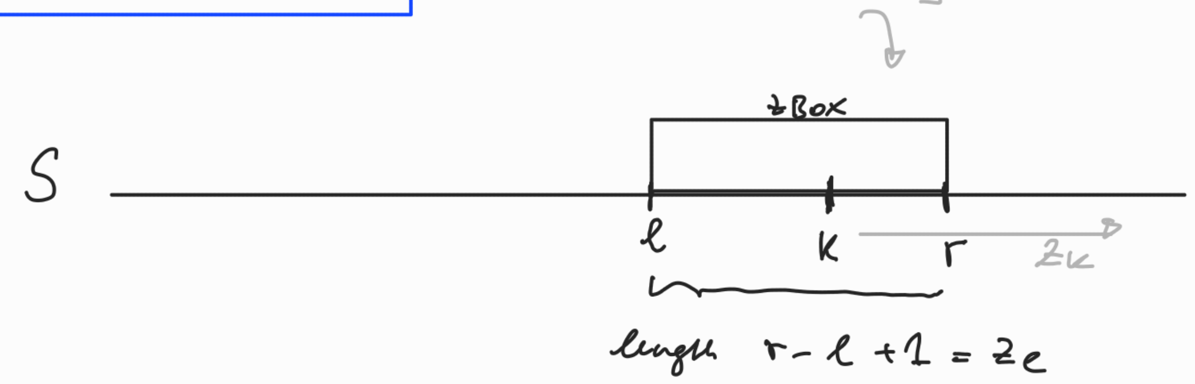
$\Rightarrow S[9] = S[1]$ ✓
$S[10] = S[2]$ ✓
$S[11] \neq S[3]$ ✗

$\Rightarrow z_9 = 2, \; l_9 = 9$

$r_9 = 9 + 2 - 1 = 10$

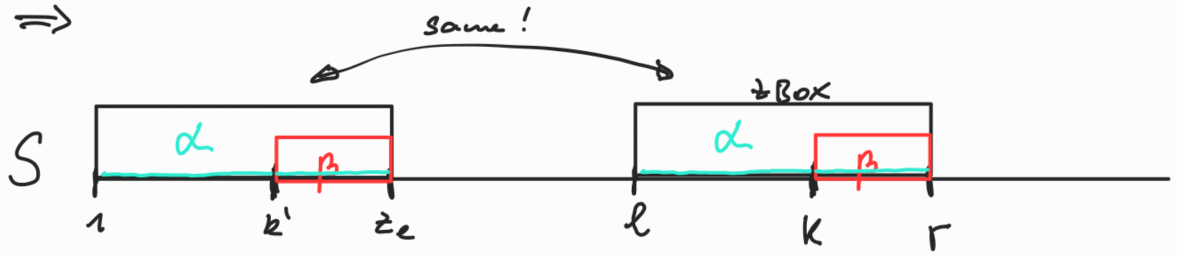
Case 2 : $r \geq k$

[Now we have some pre-knowledge about about z_k]



z-Box ≙ prefix of S

⇒



$$S[1..z_l] \;=\; S[l..r] = \alpha$$

$$S[k'..z_l] \;=\; S[k..r] = \beta$$

We want to know now z_k (longest string starting at k & is prefix of S)

Question : what we know about β ?

if we know β or "first part" of β is prefix of S, we don't need to compare the respective positions.

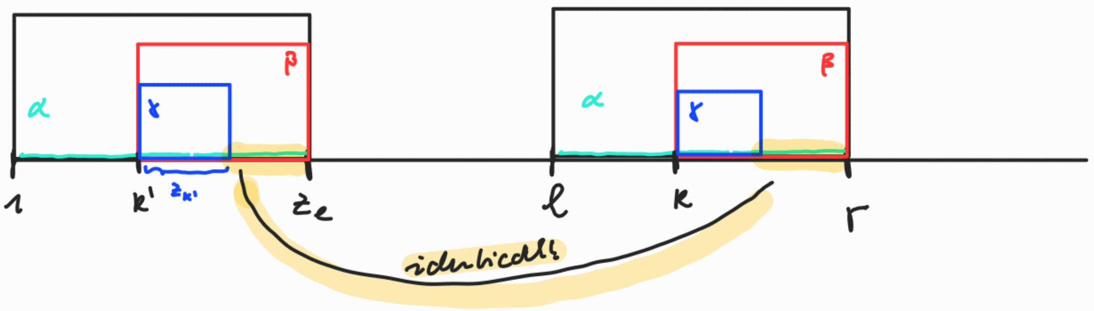
! This knowledge is already stored in $z_{k'}$

$$\boxed{k' = k - l + 1}$$

and leads to following sub cases.

2A, 2B

2A $z_{k'} < |\beta| = r - k + 1$



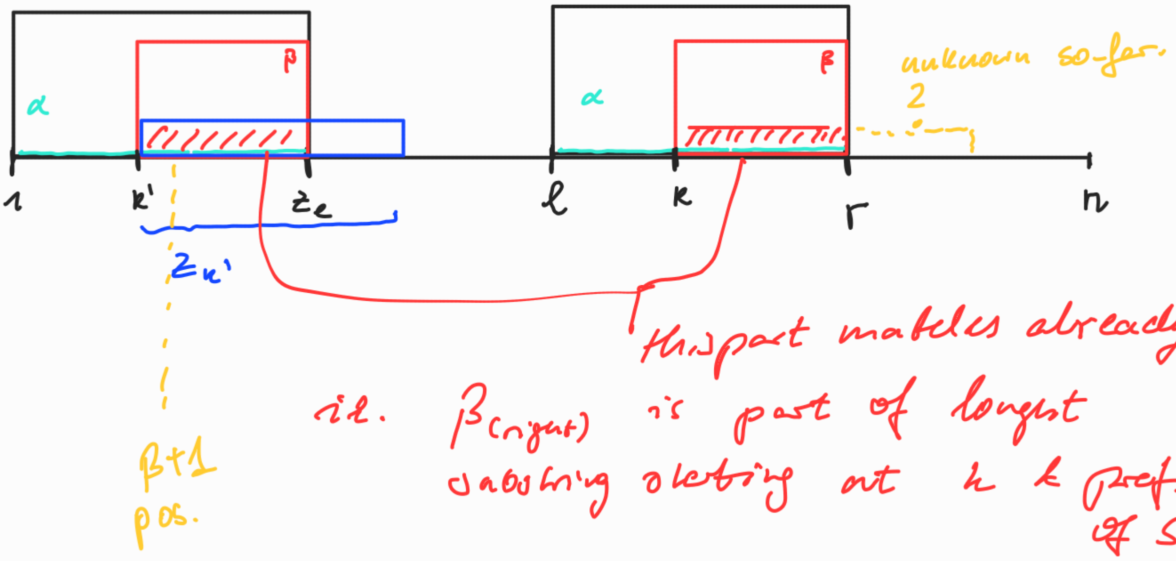
$z_{k'}$ = length of γ = length of longest substring starting at k' & is prefix of S

Since $\beta_{\text{left}} = \beta_{\text{right}}$ ⇒ $\gamma_{(\text{right})}$ is longest substring starting at k & is prefix of S.

⇒ $z_k = z_{k'}$

ℓ & r remain unchanged.

2B $z_{k'} \geq |\beta| = r - k + 1$



this part matches already, i.e. $\beta_{(\text{right})}$ is part of longest substring starting at k & prefix of S.

$\beta = S[k..r]$ is prefix of S.

& $z_k \geq |\beta| = r - k + 1$

However $Z_k > |\beta|$ may be possible.

[yellow part above]

$\Rightarrow$ compare $S[r+1..n]$

with $S[|\beta|+1,\dots n]$ until mismatch

$\parallel$

$r-k+1+1$

$= r-k+2$

to possibly extend Z-Box starting at k

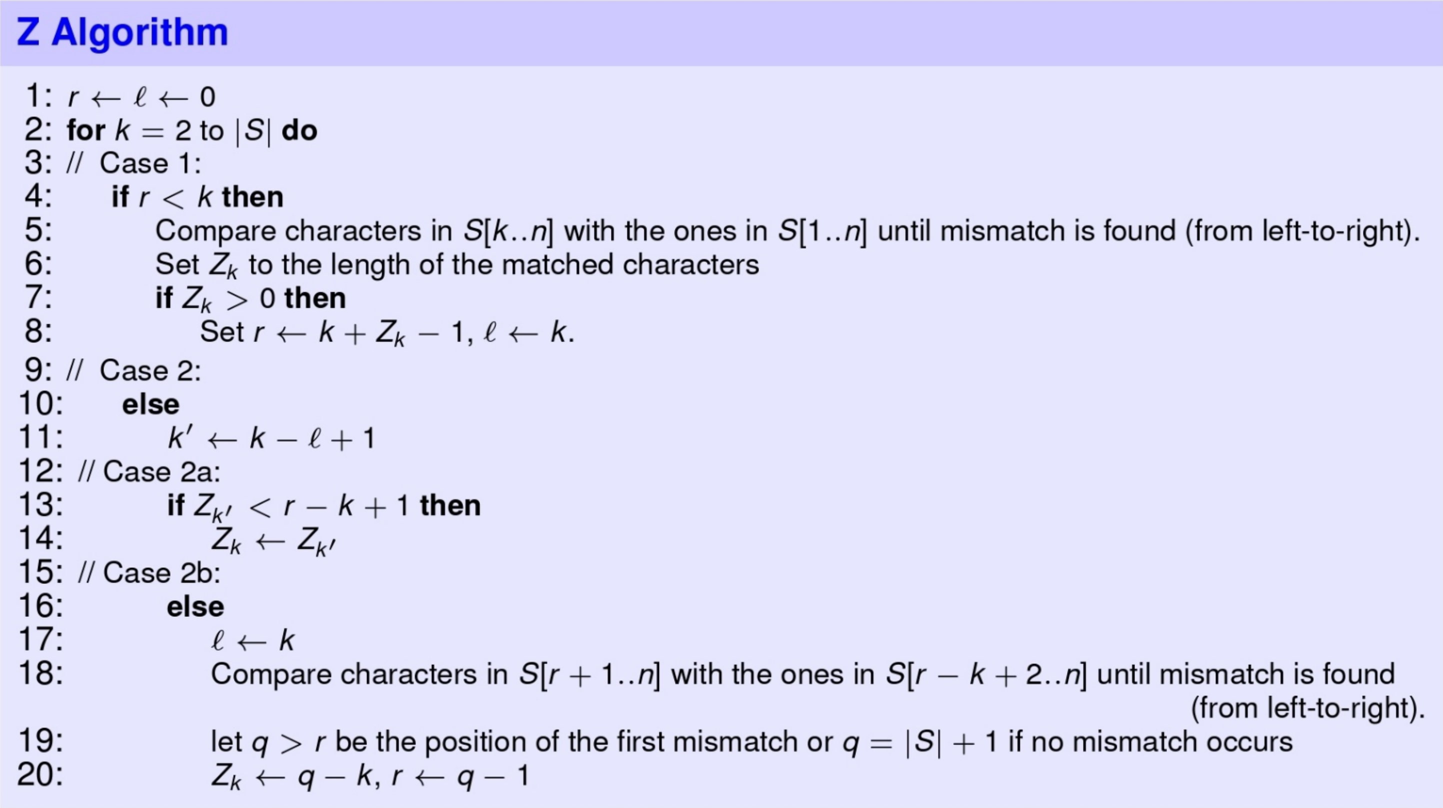
let q be <u>pos.</u> in S of 1st mismatch

& if no mismatch put $q = |S|+1$

put $Z_k = q - k$ $\quad (= k + Z_k - 1)$

$r = q - 1$

$\ell = k$



Z Algorithm

```
r ← ℓ ← 0
for k = 2 to |S| do
// Case 1:
  if r < k then
    Compare characters in S[k..n] with the ones in S[1..n] until mismatch is found (from left-to-right).
    Set Z_k to the length of the matched characters
    if Z_k > 0 then
      Set r ← k + Z_k − 1, ℓ ← k.
// Case 2:
  else
    k' ← k − ℓ + 1
// Case 2a:
    if Z_k' < r − k + 1 then
      Z_k ← Z_k'
// Case 2b:
    else
      ℓ ← k
      Compare characters in S[r + 1..n] with the ones in S[r − k + 2..n] until mismatch is found
                                                                        (from left-to-right).
      let q > r be the position of the first mismatch or q = |S| + 1 if no mismatch occurs
      Z_k ← q − k, r ← q − 1
```

latter discussion implies:

Thm: Z-alg. correctly computes all Z-Box Z_i, $i > 2$

however more important:

Thm: All $Z_i(S)$ values are computed in $O(|S|)$ time.

proof:

FOR loop (line 2-20) runs $O(|S|)$ times.

Q: What happens within FOR-loop?

to answer this question let us count character-comparisons.

Each character comparison results either in match or mismatch.

⇒ let us count match / mismatches.

each comparison ends when 1st mismatch occurs

⇒ since $O(|S|)$ different comparisons

⇒ total $O(|S|)$ mismatches

Notation: C_k = NR of comparisons in k-th iteration

m_k = NR of matches in k-th iteration

Claim: $r_k - r_{k-1} \geq m_k$

proof: in Z-Alg we either use Case 1/2A/2B.

Assume CASE 1 applies:

& thus, $r_{k-1} < k \xrightarrow{Alg.}$ explicit comparison of S[k..n] with S[1..m] until mismatch.

$\Rightarrow$ at most $n-k+1$ comparisons

in Alg we put $Z_k = m_k$

in Alg case $Z_k > 0$ & [$\underset{m_k}{\overset{\parallel}{Z_k}} = 0$ implicit by leaving "$r = r_k = r_{k-1}$" unchanged].

$k-1 \geq r_{k-1}$

$Z_k > 0$: $r_k = k + Z_k - 1 = k + m_k - 1 \geq r_{k-1} + m_k$

$\Leftrightarrow r_k \geq r_{k-1} + m_k$

$\Leftrightarrow r_k - r_{k-1} \geq m_k$ ✓

$Z_k = 0$: $r_k = r_{k-1} = r_{k-1} + \underset{0}{\overset{\parallel}{m_k}}$

(implizit im algo)

$\Leftrightarrow r_k - r_{k-1} = m_k = 0$. ✓

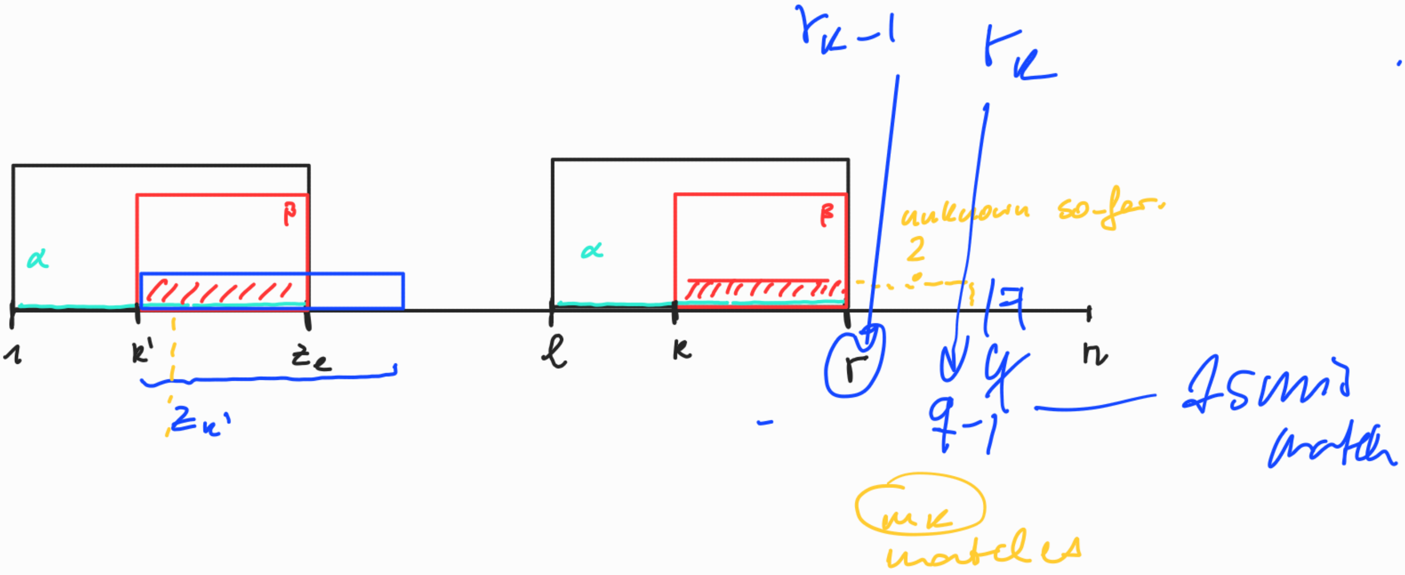
Now Case 2A/2B.

CASE 2A: in Alg "r" $= r_{k-1}$ remains unchanged, i.e. $r_k = r_{k-1}$

moreover no comparisons are made, i.e., $m_k = 0$

$\Longleftrightarrow \; r_k - r_{k-1} = m_k$ ✓

CASE 2B: in Alg: $q > r_{k-1}$ & $r_k = q-1$:



$\Rightarrow \; r_k - r_{k-1} \geq m_k$ ✓ [see picture].

$n := |S|$

$$\Rightarrow \sum_{k=2}^{n} c_k \leq \underbrace{\sum_{k=2}^{n} 1}_{\text{(mism.)}} + \underbrace{\sum_{k=2}^{n} m_k}_{\text{(match)}} = n-1 + \sum_{k=2}^{n} m_k$$

$$\underset{\text{(claim)}}{\leq} n-1 + (r_2-r_1) + (r_3-r_2) + (r_4-r_3) \ldots + (r_n - r_{n-1}) = n-1 + \underbrace{r_n}_{\leq n} - \underbrace{r_1}_{=0} \leq 2n-1 \in O(n)$$

□

in Algo $r_1 = \ell_1 = 0$

Exmpl

i	1	2	3	4	5	6	7	8	9	10
S	a	c	a	c	a	b	a	c	a	c
z_i	-									
l_i	-									
r_i	-									

Init $r = l = 0$

$k = 2$: $k = 2 > r = 0$ (Case 1)

compare $S[2 \ldots n]$ with $S[1 \ldots n]$

$\Rightarrow S(2) \neq S(1)$ mismatch

$\Rightarrow z_k = 0$, r, l unchanged.

i	1	2	3	4	5	6	7	8	9	10
S	a	c	a	c	a	b	a	c	a	c
z_i	-	0								
l_i	-	0								
r_i	-	0								

$k = 3$. , $k = 3 > r = 0$ (case 1)

Compare $S[3 \ldots n]$ with $S[1 \ldots n]$

$S(3) = S(1)$
$S(4) = S(2)$
$S(5) = S(3)$
$S(6) \neq S(4)$
$\Rightarrow$

$$\boxed{\begin{array}{l} z_k = 3 > 0 \\ r = 3 + 3 - 1 = 5 \\ l = 3 \end{array}}$$

i	1	2	3	4	5	6	7	8	9	10
S	a	c	a	c	a	b	a	c	a	c
z_i	-	0	3							
l_i	-	0	3							
r_i	-	0	5							

$k=4$, $r=5 \geq k=4$ (case 2)

($\alpha = aca$, $\beta = c\alpha$)

$k' = k - l + 1 = 4 - 3 + 1 = 2$

$z_{k'} = z_2 = 0 < r - k + 1 = 5 - 4 + 1 = 0$

$\Rightarrow$ case 2A: $z_4 = z_{k'} = 0$, r, l unchanged.

i	1	2	3	4	5	6	7	8	9	10
S	a	c	a	c	a	b	a	c	a	c
z_i	-	0	3	0						
l_i	-	0	3	3						
r_i	-	0	5	5						

$k=5$, $r=5 \geq k$ (case 2 with $\alpha = aca$, $\beta = a$)

$k' = k - l + 1 = 5 - 3 + 1 = 3$

$z_{k'} = z_3 = 3 > r - k + 1 = 5 - 5 + 1 = 1$

$\Rightarrow$ case 2A again and so on...

i	1	2	3	4	5	6	7	8	9	10
S	a	c	a	c	a	b	a	c	a	c
z_i	–	0	3	0	1	0	4	0	2	0
l_i	–	0	3	3	5	5	7	7	9	9
r_i	–	0	5	5	5	5	10	10	10	10

Based on preprocessing with Z-alg. we can design a:

Simple linear-time exact matching Alg

```
Simple-exact-matching ( P, T, $ )        // $ character not in T & P
    occurrences = ∅
    n = |P| ,   m = |T|
    S = P $ T
    preprocess S with Z-Alg to compute $Z_2 \dots Z_{|S|}$
    FOR ( i = n+2, ..., |S| = m+n+1 )
        IF ( $Z_i = n$ )
            add i-n-1 to occurrences
```

Exmpl   P = b b a c , T = a b b a b b a c a

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S	b	b	a	c	$	a	b	b	a	b	b	a	c	a
Z_i	(doesn matter ..)					0	3	1	0	$\boxed{4}$	1	0	0	0

$Z_{10} = |P| = 4$ => P occurs on pos $10 - 4 - 1 = 5$ of S runtime: Line 1 constant

2 $n + m \in O(m+n)$

3 $n+m+1 \in O(m+n)$

4 $O(|S|) = O(m+n)$

5-7 $m+n-1-(n+2) = O(m)$

TOTAL: $O(m+n)$ time.

correctness:

Since \$ not in P and T

$\Rightarrow z_i \leq |P| \quad \forall i$

if $|z_i| = |P| \Rightarrow \underbrace{S[1 \ldots |P|]}_{\overset{\text{Def}}{=} P} = \underbrace{S[i \ldots i+n-1]}_{= \text{substring of } T}$

$\Rightarrow P$ in S at pos i

$= P$ in T at pos $n-i-1$

if P occurs in T at pos j, then

P occurs in S at pos $i = j+n+1$

where $i \in [n+2, m+2]$

$\Rightarrow |z_i| \geq |P|$

Since $|z_i| \leq |P| \rightarrow |z_i| = |P|$ & occurence at pos j is reported

□

Thm: Simple-exact matching correctly reports all occurrences of pattern P in text T in $O(|P| + |T|)$ time.

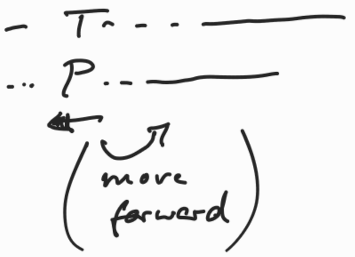
[Back to slides]

There is another important algorithm for pattern matching:

Boyer-Moore Algorithm is standart alg. that is used in many applications to textsearch (e.g. google..)

BM-Alg. is based on 3 essential ideas:

(1) Right-to-left scan
(2) Bad-Character Rule
(3) Good-Suffix Rule.



[due to limited time & since we want to cover further topics, we skip this alg.
more information about this alg in any Bioinf-textbook.]

So far: Z-alg used to preprocess P & then find P.

Now: preprocess T instead!
(text is static & remains unchanged (eg DNA))

8. Suffix Trees

classical "real world" problem:

For given Text & Pattern P
where and how often does P in Text occur?

Application:
- word processing
- internet search
- Bioinformatics (T = genom, P = gene sequ.)
- fgrep in Unix
- search for plagiarism

⋮

Def: • Σ = finite alphabet,

• string S (over Σ) = sequence of characters in Σ

Let $S = x_1 x_2 \dots x_l$, $x_i \in \Sigma$, $1 \le i \le l$

$|S| = l$ = length of S

$S[i..j]$ = substring $x_i x_{i+1} \dots x_j$ of S, starting at pos i
ending at pos j

[if $i>j$, then $S[i..j] = \varepsilon$ empty string]

$S[1..j]$ - prefix of S

$S[i..l]$ = suffix of S

prefix/suffix proper if it is not S or ε.

$S(i)$ = i-th character of S

$x,y \in \Sigma$ match if $x=y$, else mismatch

Σ^* = set of all strings over Σ, Σ^l = set of all strings over Σ of length l

S' substring of S, if $\exists\, i,j$ st $S' = S[i..j]$

Aim: given string P check where/how often
P occurs in string T

tree data structures can be seen as a collection of entities (= vertices) that are linked to simulate a hierarchy

= **rooted tree**

= trees T where one vertex $\rho \in V(T)$ is called root

Given $\rho \in V(T)$ we obtain a **partial order** $\leq_T$ on $V(T)$ st

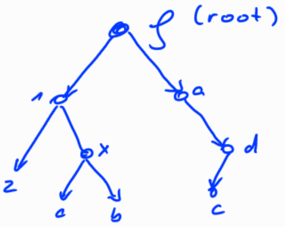
$x \leq_T y$ if y lies on simple path from ρ to x.

in this case: x is **descendant** of y & y is **ancestor** of x

$v \in V(T)$ is **common ancestor** of vertices in $W \subseteq V(T)$, if $w \leq_T v \; \forall w \in W$

a $\leq_T$-minimal common ancestor v of vertices in W is called **least common ancestor (lca(W))**

Exmpl:



the arrows indicate $\leq_T$

Note: $x \leq_T x \quad \forall x \in V(T)$

Notation: $x <_T y$ if $x \leq_T y$ & $x \neq y$

root

children of x: $y \in V(T)$ st $y <_T x$ & $(xy) \in E(T)$

x is parent of its children

Siblings are vertices with the same parent

leaf: vertices with no children

Exmpl

ρ

1 has children 2 and x which are siblings

leaves are 2, a, b, c

Exmpl: S = honolulu

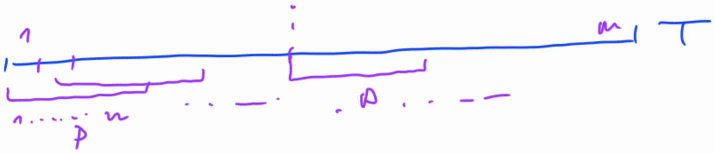
$S[1..3]$ = hon , prefix & substring of S , no suffix

$S[5..8]$ = lulu , suffix & substring of S , no prefix

$S[5..7]$ = lul , substring of S , no prefix, no suffix

NOTE: empty string ε is substring, prefix, suffix of all strings.

Naive Way :



check occurence of P in T

by comparing 1 letter of P with i letter of T

$\vdots$

j letter of P with $i+j-1$ letter of T

$\vdots$

n letter of P with $i+n-1$ letter of T

$\forall\ i = 1 .. n-m+1$

$\Rightarrow$ run time $O(m\,n)$

Often, the text (= long string) is fixed & does not change

eg. • collected work of Shakespeare

• Genom

To find a pattern P (= string), we need a datastructure that represents the text so that we can find efficiently P

To this end: suffix trees

Def [suffix tree] let S be string of length $|S| = m$

A suffix tree for S is a rooted tree T (with root g_r) that satisfies the following properties:

(X1) T has precisely m leaves that are uniquely labeled with $1, 2 \dots m$

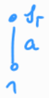
(X2) all inner vertices, (= vertices that are not leaves) except possibly the root, have at least 2 children

(X3) every edge of T is labeled by a non-empty substring of S

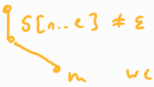
(X4) For all distinct children v_1, v_2 of v we have: label of edge (v, v_1) & (v, v_2) start with different characters

(X5) if we concatenate the labels on the edges in order from g_r to leaf i, we obtain the suffix $S[i..m]$

Exmpl: $S = a$, T



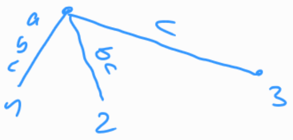
[this is the only case where g_r has only one child, since if $|S| > 1$ & T,



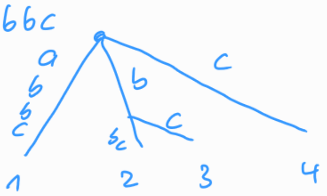
we dont get $S[m..m]$, since $|S[m...m]| = 1$ & (X3).]

$S = abc$

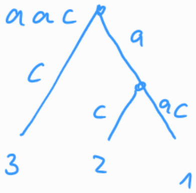
T



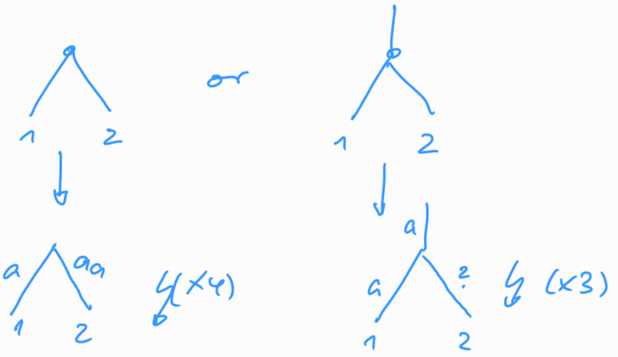
$S = abbc$



$S = aac$



What is with $S = aa$? $T \hat{=}$



Observation: not for all strings a suffix tree exists!

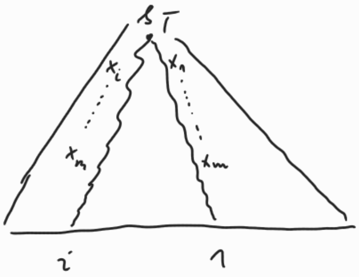
Lemma 8.1 If $S = \underbrace{x_1 \cdots x_k}_{\text{prefix}} \cdots \underbrace{x_i \cdots x_m}_{\text{suffix}}$ st $x_1 \cdots x_k = x_i \cdots x_m$

that is S contains prefix that is also a suffix of S.

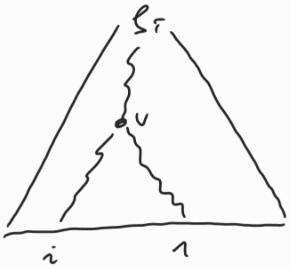
Then, no suffix tree for S exists

Proof: Wlog, we assume that such suffix/prefix are chosen in S to be maximal (there is no longer prefix that is also suffix)

T:



or

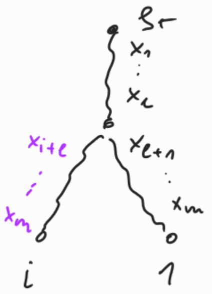


lowest common ancestor of 1 & i is ρ_T

Since $x_i = x_1$, this does not work due to (X4)

lowest common ancestor of 1 & i is $v <_T \rho_T$

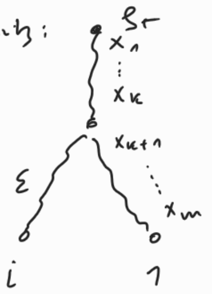
$$\begin{array}{ccccccc} x_1 & x_2 & x_3 & \cdots & x_k & \cdots & x_m \\ \| & \| & \| & & \| & & \\ x_i & x_{i+1} & x_{i+2} & \cdots & x_m & & \end{array}$$



, $\ell < m$

↯ (X4) since $x_{i+\ell} = x_{\ell+1}$

⇒ only possibility:



↯ (X3)

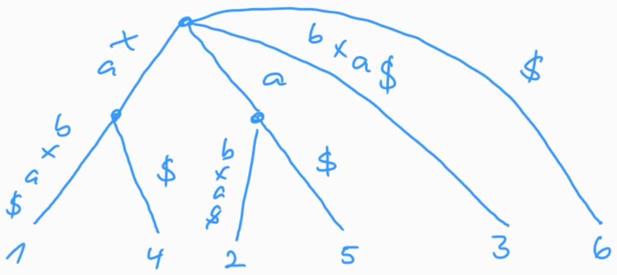
□

$\Rightarrow$ if suffix of S is also prefix of S no suffix tree exists!

IDEA: simply add special symbol \$ at end of S, where \$ does not occur in S.

From here on: $S = x_1 \dots x_m$ with $x_m = \$$

Example: $S = \underset{1}{x}\underset{2}{a}\underset{3}{b}\,\underset{4}{x}\underset{5}{a}\underset{6}{\$}$ (for "xabxa" no suffix tree)

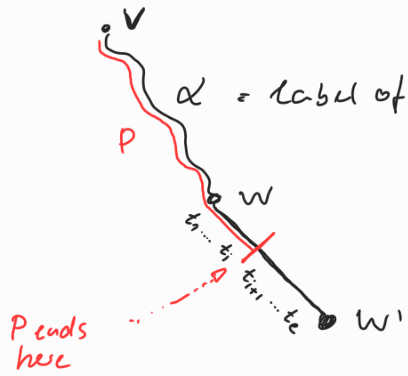


Def.

Let $v \le_T w$ & P be the unique simple path from v to w

label of P = concatenation of labels of edges in order from v to w.

! By slight abuse of notation, we also consider simple paths in suffix tree that may end on edge (and not on vertex)



α = label of path from v to w

$(w\,w') \in E(T)$ with label $t_1 \dots t_\ell$

label of P = $\alpha\, t_1 \dots t_i$

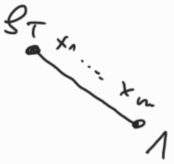
How to construct suffixtree & how does this help?

Alg. SUFFIXTREE (Idea)

// iteratively build suffixtrees $T_1 \ldots T_m$ by adding $S[1\ldots m], \ldots S[m\ldots m]$ in previously constructed T_i

Input: $S = x_1 \ldots x_m$

1) construct T_1 =



2) Assume T_i is constructed, then construct T_{i+1} $(i<m)$ as follows:

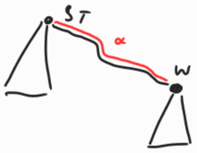
(a) Find path P in T_i that starts in ϱ_{T_i} with longest label that is a prefix of $S[i+1..m]$ [path could end in ϱ]

(i) By similar arguments as in proof of L. 8.1 this path is uniquely determined since no two edges $(v v_1), (v v_2)$, $v_1 v_2$ children of v have labels starting with same symbol (X4).

(ii) this path will never end in leaf of T_i since P starts at root & concatenation of edge labels to leaf i ields $S[i..m]$ where $|S[i\ldots m]| > |S[i+1..m]|$ [by induction - EXNC]

(b) This path P either ends in edge or non-leaf vertex of T_i

P ends in vertex:

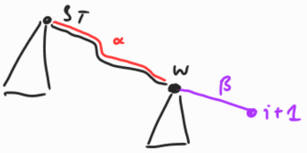


w no leaf => w has children.

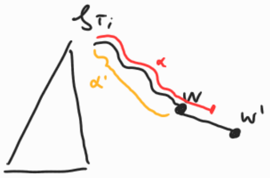
α prefix of $S[i+1\ldots m]$, $m = \$$
But $\alpha \neq S[i+1\ldots m]$
otherwise $S[i+1\ldots m] = S[\ell..m]$ for some ℓ but $\ell < i+1$ }

$\Rightarrow \alpha = S[i+1..m-j]$, $j>1$

$\Rightarrow$ add edge $(w, i+1)$ with label $\beta = S[m-j+1..m]$



P ends in edge:



$\alpha = S[i+1 \dots m-j]$

label edge $(w,w') = t_1 .. t_r t_{r+1} .. t_s$

$\Rightarrow$ $\alpha = \alpha' t_1 .. t_r$ for some $1 \le r < s$

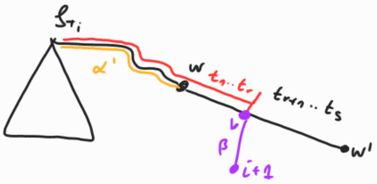
add new vertex v on edge (w,w')

& put label$(wv) = t_1 .. t_r$

label$(vw') = t_{r+1} .. t_s$

+ add new edge $(v, i+1)$

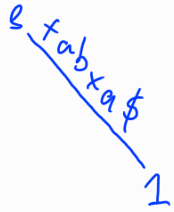
with label$(v,i+1) = S[m-j+1 .. m] = \beta$



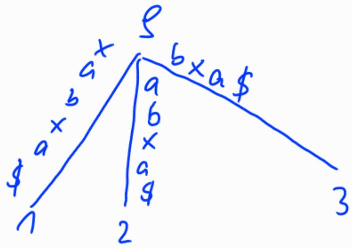
For both cases, T_{i+1} contains now path from $\varrho_{T_{i+1}}$ to $i+1$ with label $S[i+1 \dots m]$

Exmpl: $S = xabxa\$$

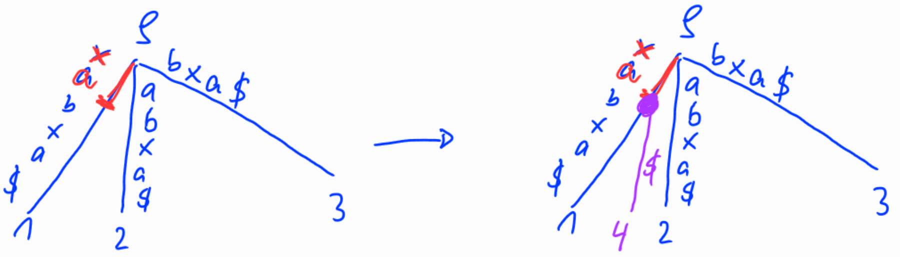
T_1:



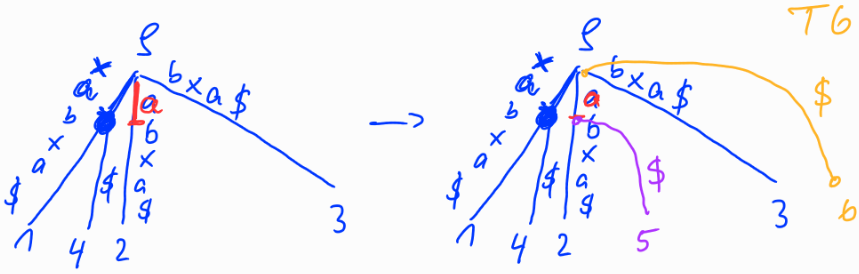
T_2 / T_3 (longest path ends in ϱ)



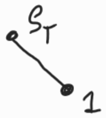
T_4: $S[4..m] = xa\$$ longest path P



T_5: $S[5..m] = a\$$



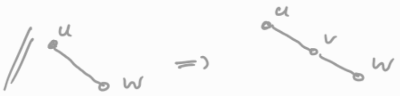
```
SUFFIX-TREE(S)
    init T as
```



```
    and label($\rho_T$,1) = S[1..m]

    FOR( i=2..m ) DO
        (end, i', l') = FIND-LONGEST-PATH( T, S[i..m] )  // returns end of path, that is,
                                                          either end = v or end = e
                                                          & index i' & l'
        IF (end = edge e)  // e = (uw) with label $\gamma$
            remove e = (uw)
            add new vertex v
            add new edges $e_1$ = (uv), $e_2$ = (v,w)
                & label $e_1$ = $\gamma$[1..l']
                  label $e_2$ = $\gamma$[l'+1..|$\gamma$|]
```

//



```
        add edge (v,i) to T with label S[i+i'..m]
```

FIND-LONGEST-PATH (T, S') [sketch → more details in Appendix]

"Follow path from ρ_T to v/e as long as possible, i.e, as long as letter on this path match with letters $S' = x_i \ldots x_m$" & return" corresponding positions i', l' + if end is edge or vertex".

Theorem [Ukkonen]: for S of length m the suffix tree can be constructed in O(m) time

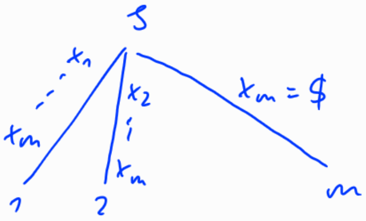
[without proof]

[quite sophisticated pointer-adjustments].

Space complexity   Suffix tree without labels $O(m)$

but we need to store labels!

worst case:



each edge label of (S,i) is of size i

$\Rightarrow \sum_{i=1}^{m} i = O(m^2)$ space

$O(m^2)$ space is bad (e.g. genome of small bacteria has 10^6 characters ~ 1TB storage

IDEA:

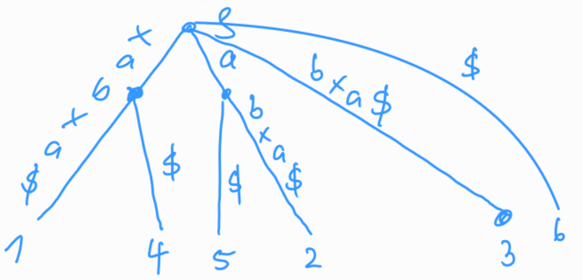
instead of saving label $S[i..j]$ for e

we only save pair (i,j)

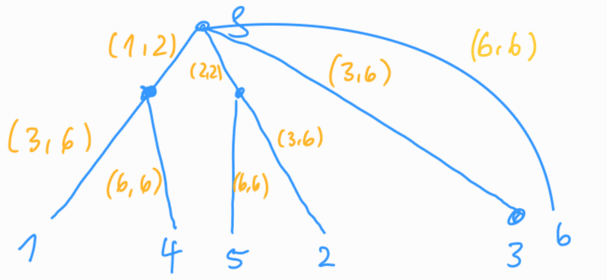
= compressed suffix tree

$\Rightarrow O(m)$ space (same space as text S)

Exampl: $S = xabxa\$$
1 2 3 4 5 6



$\Rightarrow$



Why suffix trees?

A: Examples!

! In what follows, we assume to have Ukkonen's version: $O(m)$ time

Exmpl 1 "exact text-search"

Given (long) text S & patter P (= string)

Does P in S occur?

Let T be suffix tree of S.

Call FLP(T, P) once $O(|P|)$ time

return value is (end, index i', ..)

st $P[1..i']$ corresponds to label of path in T that starts in ϱ_T

Observe, any path in T von ϱ to leaf i corresponds to $S[i..m]$

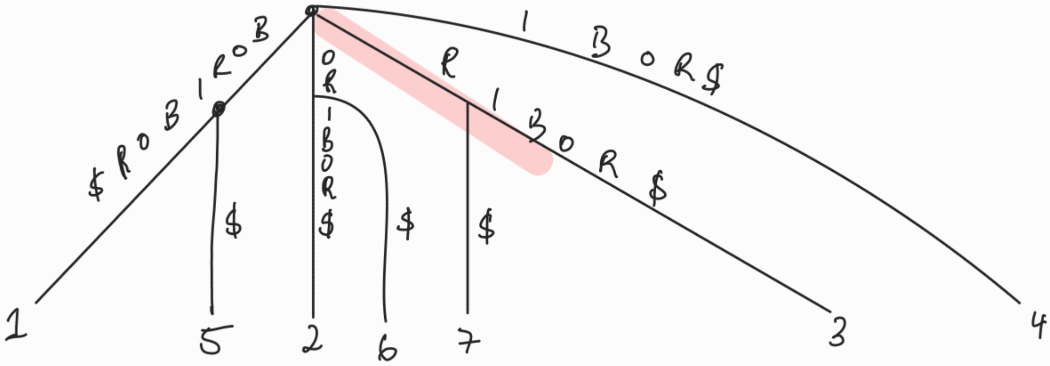
& $P[1..i']$ corresponds therefore to a prefix of some suffix of S, that is, a substring of S.

⇒ if $i' = |P|$ ⇒ P occurs in T
else, not.

⇒ runtime $O(|P|)$

Exmpl: T = BORIBOR$. P = RIB

[every exist letter appears once at first letter of edge (ϱ, v) for some v.]



Exmpl 2 "substring - database - search"

Given strings $S_1 \cdots S_\ell$ (Database of Texts)
Does P occur in (at least) one of the S_i?

$\Rightarrow$ let $\$_1 \dots \$_\ell$ pairw. distinct symbols that do not occure in any S_i & P

put $S = S_1 \$_1 S_2 \$_2 \cdots S_\ell \$_\ell$

Construct suffixtree T for S ($O(|S_1|+|S_2|+ .. |S_\ell|)$ time)

use IDEA of Exmpl 1

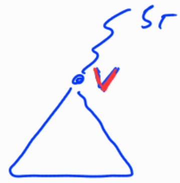
Exmpl 3 Find all occurences of P

given S with suffixtree T & pattern P with $|P| = n$

Find all occurences of P in T, that is,
all indices i st $P = S[i \dots i+n-1]$

Call FLP(T, P)

$\longrightarrow$ this gives end $= v$ or end $=$ edge (uv)

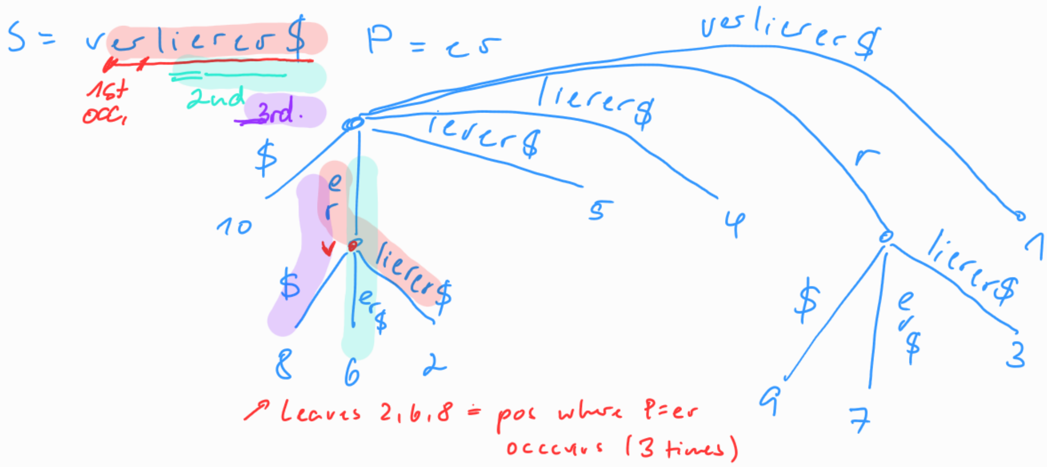


leaves $i \leq_T v$ = # occurrences of P in S

& P occurs in pos i $\forall$ all leaves $i \leq_T v$.

[Exercise : works in $O(|P| + k)$ time)

k = nr of occurences of P



↗ Leaves 2,6,8 = pos where P=er occurs (3 times)

Exampl 4

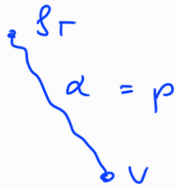
Find longest substring in S that occurs at least 2 times

[Application: Find repeated regions in genom]

MISSISSIPPI

issi

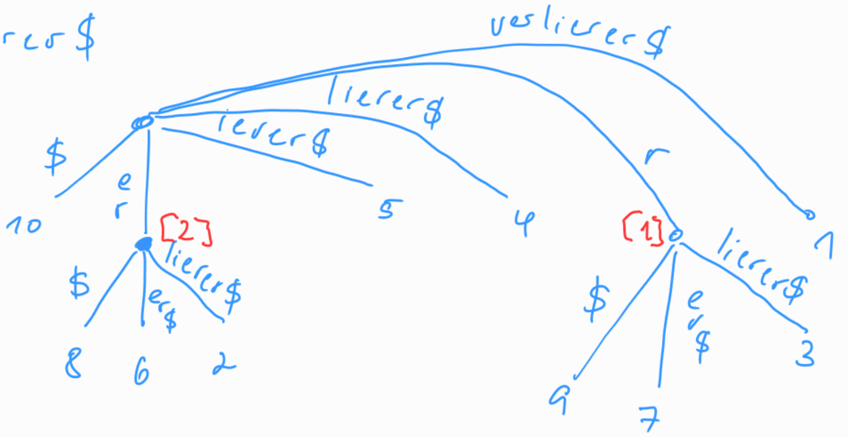
string depth of v in T is $|\alpha|$



α = path label of path from r_T to v

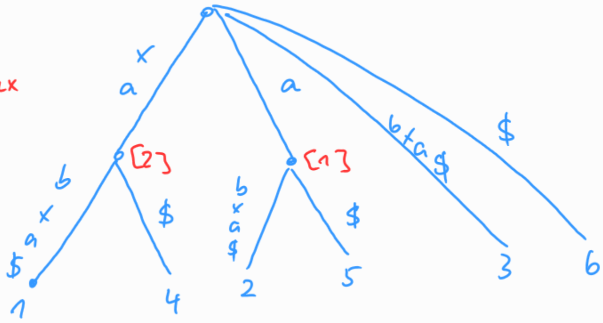
S = verlierer$

string depth of inner vertex



S = xabxa$

string depth of inner vertex



xa occurs 2 times.

- every inner vertex has 2 children (at least) & first symbol on edges with same parent have distinct symbols

⇒ length of longest substring that occurs at least 2 times = maximum string-depth of inner vertices.

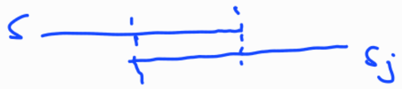
[Exus. : works in $O(|S|)$ time.]

Exampl 5 efficient detection of pairwise overlaps

given $\mathcal{J} = \{S_1 \dots S_N\}$ set of strings (e.g. sequenced DNA fragments)

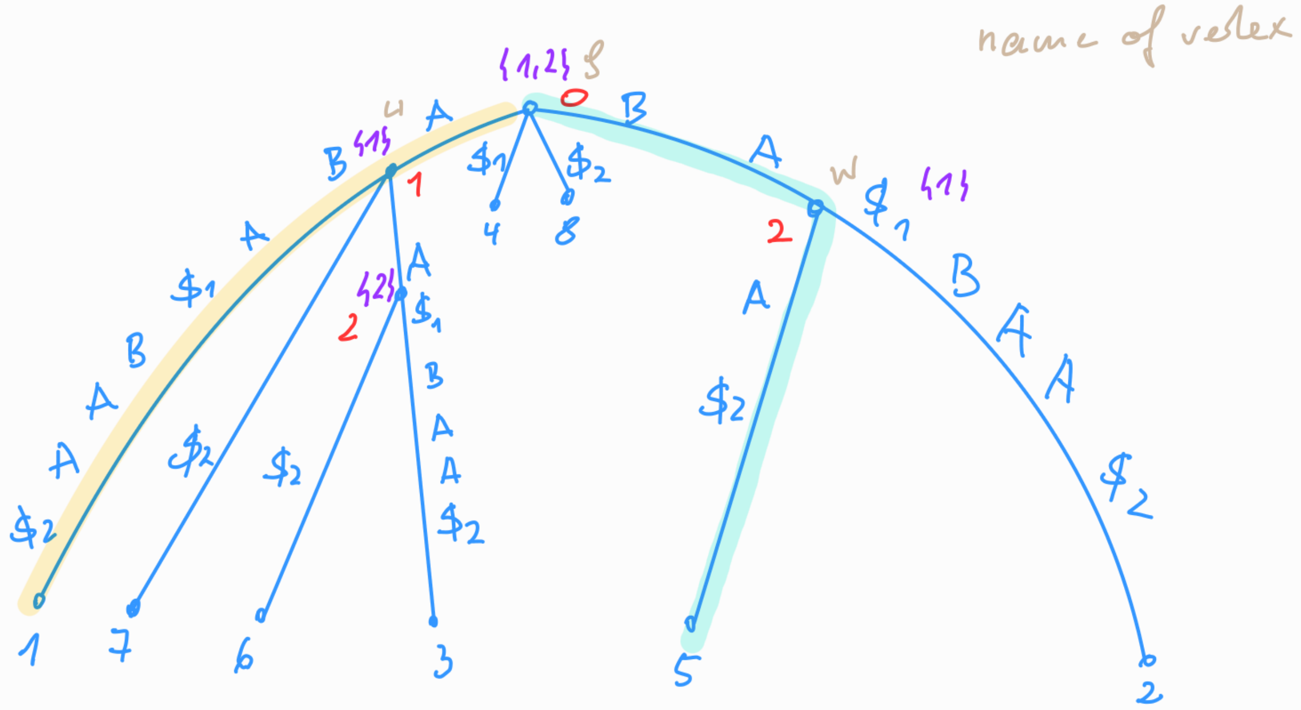
Aim: define $ov(S_i, S_j)\ \forall\, i,j$
$\quad =$ size of largest overlap

$S = S_1 \$_1 S_2 \$_2 \dots S_N \$_N$



$S_1 = ABA \longrightarrow ov(S_1 S_2) = 2$
$S_2 = BAA \qquad ov(S_2 S_1) = 1$

$S = \overset{1}{A}\overset{2}{B}\overset{3}{A}\overset{4}{\$_1}\ \overset{5}{B}\overset{6}{A}\overset{7}{A}\overset{8}{\$_2}$



P_1 = path starting with label $S_1 \$_1$
P_2 = path starting with label $S_2 \$_2$

for non-leaf x: $L(x) = \{ i \mid \exists$ leaf v s.t. $(x v)$ edge in suffix tree
& label $(x v)$ starts with $\$_i \}$.

string depth of x = # characters on path ξ to x.

```
For ($j=1..N$)
    $x = r$ (root)
    WHILE ($x$ not leaf)
        FOR (all $i \in L(x)$, $i \neq j$)
            IF ($depth(x) < \min(|S_i|,|S_j|)$
                ov$(S_i,S_j) = depth(x)$
        $x \leftarrow$ child of $x$ on $P_j$
```

$j=1$, $x=r$

FOR ($i \in L(r) = \{1,2\}$, $i \neq 1$)

$\Rightarrow$ $i=2$

$depth(r) = 0 < \min(|S_1|\,|S_2|)$

$\Rightarrow$ ov$(S_2 S_1) = 0$

$x \leftarrow u$

FOR ($i \in L(u) = \{1\}$, $i \neq 1$)

nothing

$x \leftarrow$ leaf stop.

$j=2$, $x=r$

For ($i \in L(r) = \{1,2\}$, $i \neq 2$)

$\Rightarrow$ $i=1$

$dept(r) = 0 < \min(|S_1|\,|S_2|)$

ov$(S_1,S_2) = 0$

$x \leftarrow w$

For ($i \in L(w) = \{1\}$, $i \neq 2$)

$\Rightarrow$ $i=1$

$depth(w) = 2 < \min(S_1,S_2)$

$\Rightarrow$ ov$(S_1,S_2) = 2$

$x \leftarrow$ leaf stop.

[without proof of correctness or runtime analysis]

$O(N\,\|S\|)$

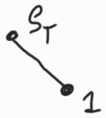
Exercise ↗ Book "Alg. Aspects of Bioinf.")

Appendix

Details of $O(n^2)$ Algorithm

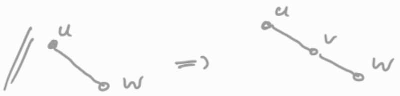
SUFFIX-TREE(S)

```
init $\mathcal{T}$ as
```



```
and label$(\rho_T, 1) = S[1..m]$

FOR( $i = 2..m$ ) DO
    $(end, i', \ell')$ = FIND_LONGEST_PATH ( $\mathcal{T}$, $S[i..m]$ )  // returns end of path, that is,
                                                          // either end = v or end = e
                                                          // & index i' & l'
    IF( end = edge e )  // $e = (uw)$ with label $\gamma$
        remove $e = (uw)$
        add new vertex $v$
        add new edges $e_1 = (uv)$, $e_2 = (v,w)$
              & label $e_1 = \gamma[1..\ell']$
                label $e_2 = \gamma[\ell'+1..|\gamma|]$
```



```
    add edge $(v,i)$ to $\mathcal{T}$ with label $S[i+i'..m]$
```

FIND_LONGEST_PATH ($\mathcal{T}$, S')

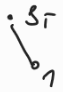
```
$j = 1$,  $v = \rho_T$
WHILE ( $j \le |S'|$ ) DO
    Find edge $e = (vw)$ in $\mathcal{T}$, $w \preceq_T v$, whose label starts with $S'(j)$
    IF ( such edge does not exist )
        RETURN ( $v$, $j-1$, $\emptyset$ )  // path ends in v
    Let $\gamma$ be label of $e$
    $\ell = 1$
    WHILE ( $j \le |S'|$ & $\ell \le |\gamma|$ & $S'(j) = \gamma(\ell)$ ) DO
        $j = j+1$
        $\ell = \ell+1$
    IF ( $\ell \le |\gamma|$ )  // while loop ended at some point before "$S'(j) = \gamma(\ell)$
                                                                  $\forall \ell = 1..|\gamma|$"
                                                         ie, $S(j) \neq S(\ell)$
        RETURN( $e$, $j-1$, $\ell-1$ )
    $v = w$  // go to consider edges starting at w
RETURN( $v$, $|S'|$, $\emptyset$ )
```

Prop. 8.2 Alg. SUFFIXTREE correctly computes suffix tree for $S = x_1 \dots x_m$, $x_m = \$$.

proof: (sketch)

(X1): T_1 has 1 leaf, T_2 has 2 leaves $\overset{\text{Ind}}{\dots}$ T_m has m leaves ✓

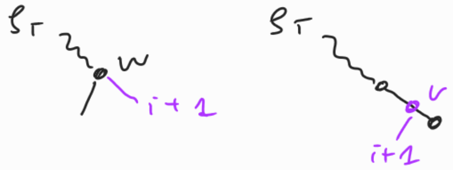
(X2): T_1:



✓

Assume, for T_i, each inner vertex (except possibly ρ) has at least 2 children

per constr. T_{i+1}:

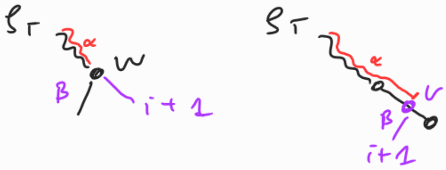


✓

(X3) by construction & induction each edge has non-empty string as label

(X4) T_1 ✓ Assume for T_i $\forall v$: label $(vv_1), (vv_2)$ starts with different symbol, where v_1, v_2 are children of v.

T_{i+1}:



Since α is longest substring that is prefix of

$S[i+1..m] = \underbrace{x_{i+1} \dots x_r}_{\alpha} \underbrace{\dots x_m}_{\beta}$ $\beta(1) \neq x_{r+1}$ as otherwise α not longest prefix! ✓

(X5) by constr. & induct. concatenating edge-labels from ρ to i yields $S[i \dots m]$.

□

Runtime:

[suffix-tree has $O(m)$ edges & vertices]

SUFFIXTREE(S), $S = x_1 \dots x_m$

add/remove etc in $O(1)$ time

→ m times FIND_LONGEST_PATH (FLP) is called.

FPL: • in each of the $|S'|$-steps in 1st while-loop

Find-edge → for v all neighbors w are considered

$\Rightarrow$ $\deg_T(v)$ many vertices

over all calls of 1st-while loop, "FIND_edge" is called,

$$\leq \sum_{v \in V} \deg(v) = 2|E| = O(|E|) \text{ times}$$
$$= O(m)$$

• 1st + 2nd while loop:

```
WHILE ($j \leq |S'|$) DO
    ⋮
    WHILE ($j \leq |S'|$ & $\ell \leq |\gamma|$ & $S'(j) = \gamma(\ell)$) DO
        $j = j + 1$
        ⋮
```

$\Rightarrow$ $j \leq m$ $\Rightarrow$ $O(m)$ time

• Remaining operation in 1st-while loop: $O(1)$

$\Rightarrow$ FLP runs in $O(m)$ time

$\Rightarrow$ SUFFIX-TREE(S) runs in $O(m^2)$ time.

Theorem [Ukkonen]: for S of length m the suffix tree can be constructed in $O(m)$ time

[without proof]