

---

# Advanced Data Structures

String Pattern Matching/Text Search

# What is Pattern Matching?

---

- Definition:
  - given a text string T and a pattern string P, find the pattern inside the text
    - T: “the rain in spain stays mainly on the plain”
    - P: “n th”

# Text search

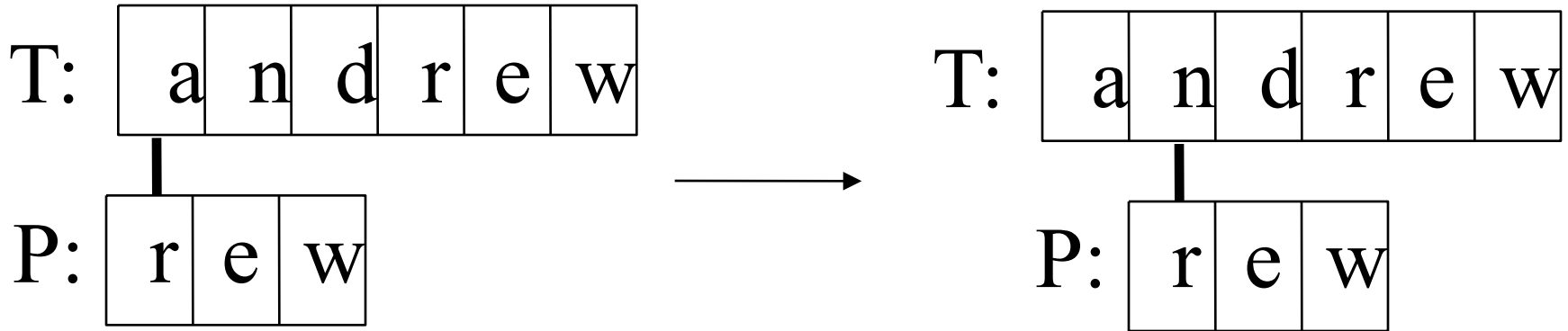
---

- Pattern matching directly
  - Brute force
  - Rabin-Karp
  - KMP
  - BM
- Regular expressions (Not in this course)
- Indices for pattern matching
  - Inverted files
  - **Signature files**
  - **Suffix trees** and **Suffix arrays**

# The Brute Force Algorithm

---

- Check each position in the text T to see if the pattern P starts in that position



P moves 1 char at a time through T



....

# Analysis

---

- Brute force pattern matching runs in time  $O(mn)$  in the worst case.
- But most searches of ordinary text take  $O(m+n)$ , which is very quick.

*continued*

- 
- The brute force algorithm is fast when the alphabet of the text is large
    - e.g. A..Z, a..z, 1..9, etc.
  - It is slower when the alphabet is small
    - e.g. 0, 1 (as in binary files, image files, etc.)

*continued*

- 
- Example of a worst case:
    - T: "aaaaaaaaaaaaaaaaaaaaaaaaah"
    - P: "aaah"
  - Example of a more average case:
    - T: "a string searching example is standard"
    - P: "store"

# Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a **Brute Force** comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- Perhaps an example will clarify some things...



# Rabin-Karp Example

- Hash value of “AAAAA” is 37
- Hash value of “AAAAH” is 100

1) AAAAAAAAAAAAAAAAAAAAAAAAAAH  
AAAAH  
 $37 \neq 100$       1 comparison made

2) AAAAAAAAAAAAAAAAAAAAAAAAAAAH  
AAAAH  
 $37 \neq 100$       1 comparison made

3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH  
AAAAH  
 $37 \neq 100$       1 comparison made

4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH  
AAAAH  
 $37 \neq 100$       1 comparison made

...

N) AAAAAAAAAAAAAAAAAAAAAAAABAAAH  
BAAAH  
5 comparisons made       $100 = 100$

# Rabin-Karp Algorithm

*pattern is M characters long*

**hash\_p**=hash value of pattern

**hash\_t**=hash value of first M letters in body of text

**do**

**if** (**hash\_p** == **hash\_t**)

    brute force comparison of pattern

    and selected section of text

**hash\_t**= hash value of next section of text, one character over

**while** (end of text      **or**

    brute force comparison == true)

# Rabin-Karp

- Common Rabin-Karp questions:
  - “What is the hash function used to calculate values for character sequences?”
  - “Isn’t it time consuming to hash every one of the M-character sequences in the text body?”

To answer some of these questions, we’ll have to get mathematical.

# Rabin-Karp Math

- Consider an M-character sequence as an M-digit number in base b, where b is the number of letters in the alphabet. The text subsequence  $t[i .. i+M-1]$  is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + \dots + t[i+M-1]$$

- Furthermore, given  $x(i)$  we can compute  $x(i+1)$  for the next subsequence  $t[i+1 .. i+M]$  in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + \dots + t[i+M]$$

$$x(i+1) = x(i) \cdot b$$

Shift left one digit

$$- t[i] \cdot b^M$$

Subtract leftmost digit

$$+ t[i+M]$$

Add new rightmost digit

- In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.

# Rabin-Karp Math Example

- Let's say that our alphabet consists of 10 letters.
- our alphabet = a, b, c, d, e, f, g, h, i, j
- Let's say that “a” corresponds to 1, “b” corresponds to 2 and so on.

The hash value for string “cah” would be ...

$$3*100 + 1*10 + 8*1 = 318$$

# Rabin-Karp Mods

- If  $M$  is large, then the resulting value ( $\sim b^M$ ) will be enormous. For this reason, we hash the value by taking it **mod** a **prime number  $q$** .
- The **mod** function ( $\%$  in Java) is particularly useful in this case due to several of its inherent properties:

$$[(x \bmod q) + (y \bmod q)] \bmod q = (x+y) \bmod q$$

$$(x \bmod q) \bmod q = x \bmod q$$

- For these reasons:

$$h(i) = ((t[i] \cdot b^{M-1} \bmod q) + (t[i+1] \cdot b^{M-2} \bmod q) + \dots + (t[i+M-1] \bmod q)) \bmod q$$

$$h(i+1) = (h(i) \cdot b \bmod q$$

Shift left one digit

$$- t[i] \cdot b^M \bmod q$$

Subtract leftmost digit

$$+ t[i+M] \bmod q )$$

Add new rightmost digit

$$\bmod q$$

# Rabin-Karp Complexity

- If a sufficiently large prime number is used for the *hash function*, the hashed values of two different patterns will usually be distinct.
- If this is the case, searching takes  $O(N)$  time, where  $N$  is the number of characters in the larger body of text.
- It is always possible to construct a scenario with a worst case complexity of  $O(MN)$ . This, however, is likely to happen only if the prime number used for hashing is small.

# The KMP Algorithm

---

- The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a *left-to-right* order (like the brute force algorithm).
- But it shifts the pattern more intelligently than the brute force algorithm.

*continued*



# Summary

---

- If a mismatch occurs between the text and pattern  $P$  at  $P[j]$ , what is the *most* we can shift the pattern to avoid wasteful comparisons?

# Summary

---

- If a mismatch occurs between the text and pattern  $P$  at  $P[j]$ , what is the *most* we can shift the pattern to avoid wasteful comparisons?
- *Answer*: the largest prefix of  $P[0 \dots j-1]$  that is a suffix of  $P[1 \dots j-1]$

# Example

T: 

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

P: 

1	2	3	4	5	6
<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

7  

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

8 9 10 11 12  

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

13  

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

14 15 16 17 18 19  

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

$k$	0	1	2	3	4
$F(k)$	0	0	1	0	1

# KMP Advantages

---

- KMP runs in optimal time:  $O(m+n)$ 
  - very fast
- The algorithm never needs to move backwards in the input text,  $T$ 
  - this makes the algorithm good for processing very large files that are read in from external devices or through a network stream

# KMP Disadvantages

---

- KMP doesn't work so well as the size of the alphabet increases
  - more chance of a mismatch (more possible mismatches)
  - mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later

# Boyer and Moore Algorithm

A fast string searching algorithm. *Communications of the ACM*.  
Vol. 20 p.p. 762-772, 1977.

**BOYER, R.S. and MOORE, J.S.**

# Boyer and Moore Algorithm

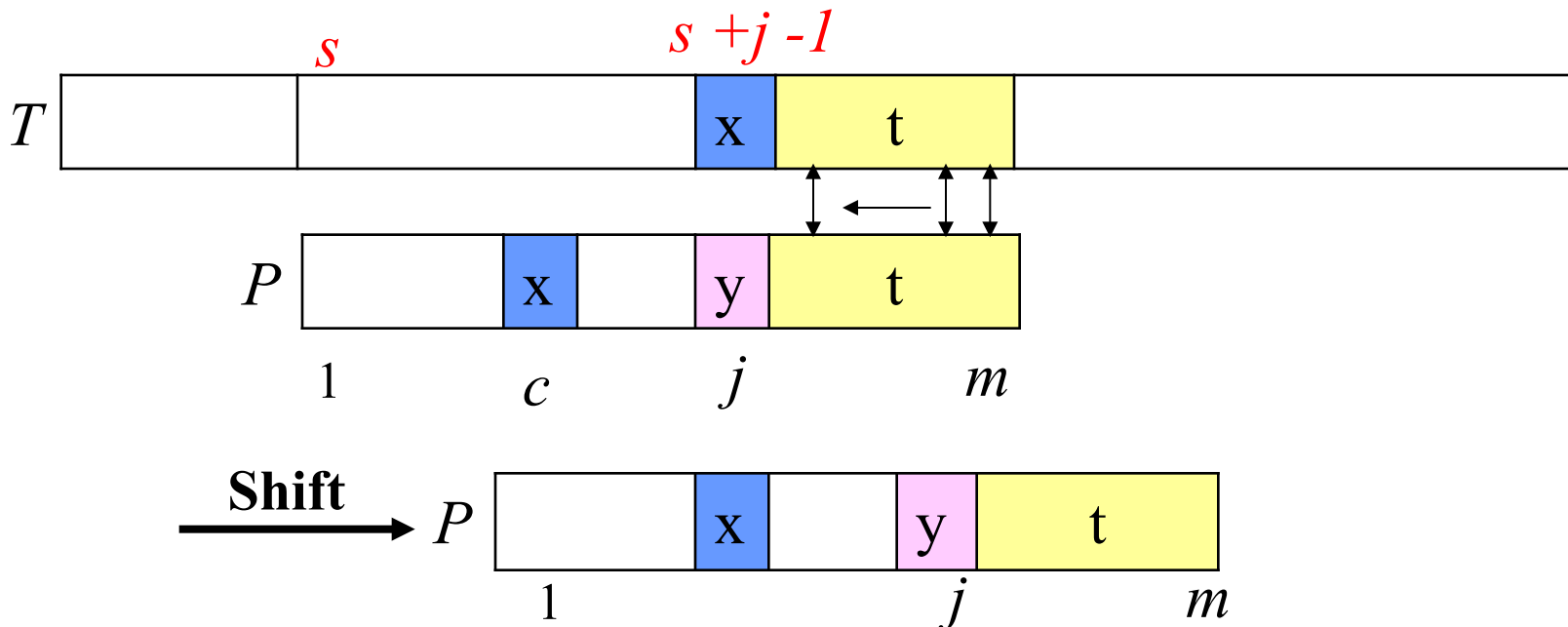
- The algorithm compares the pattern  $P$  with the substring of sequence  $T$  within a sliding window in the **right-to-left order**.
- The **bad character rule** and **good suffix rule** are used to determine the movement of sliding window.

# Bad Character Rule

Suppose that  $P_l$  is aligned to  $T_s$  now, and we perform a pair-wise comparing between text  $T$  and pattern  $P$  from right to left.

Assume that the first mismatch occurs when comparing  $T_{s+j-l}$  with  $P_j$ .

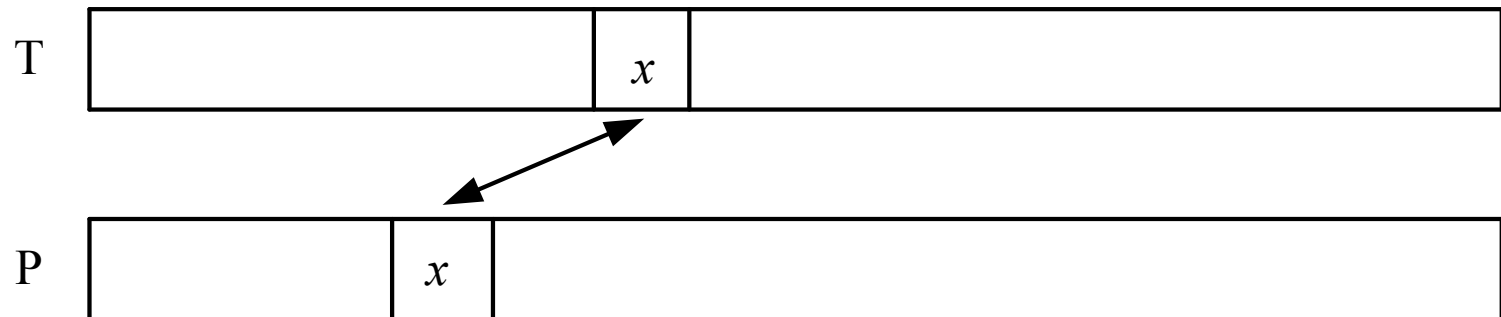
Since  $T_{s+j-l} \neq P_j$ , we move the pattern  $P$  to the right such that the largest position  $c$  in the left of  $P_j$  is equal to  $T_{s+j-l}$ . We can shift the pattern at least  $(j-c)$  positions right.





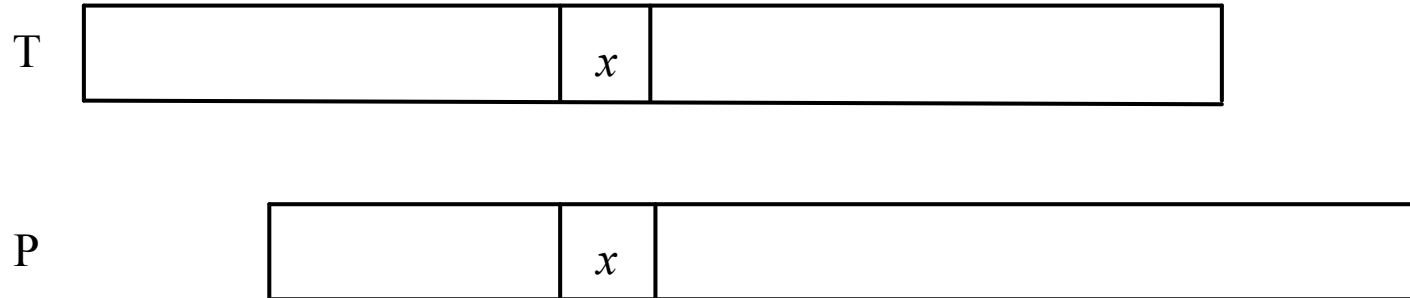
# Character Matching Rule

- Bad character rule uses Rule 2-1 (Character Matching Rule).
- For any character  $x$  in  $T$ , find the nearest  $x$  in  $P$  which is to the left of  $x$  in  $T$ .

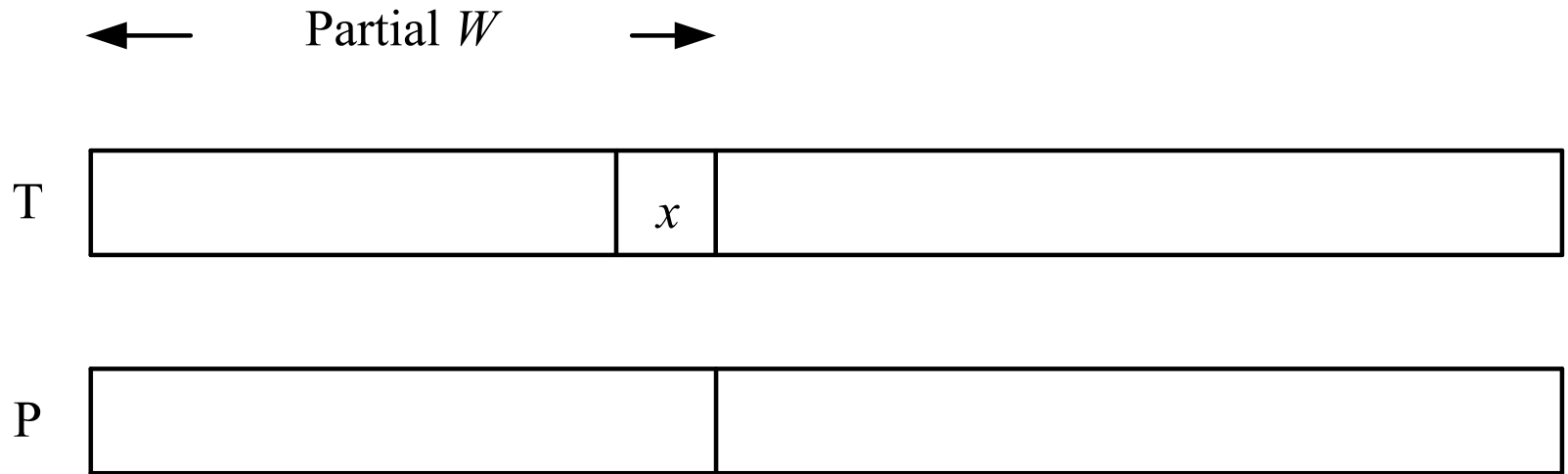


# Implication

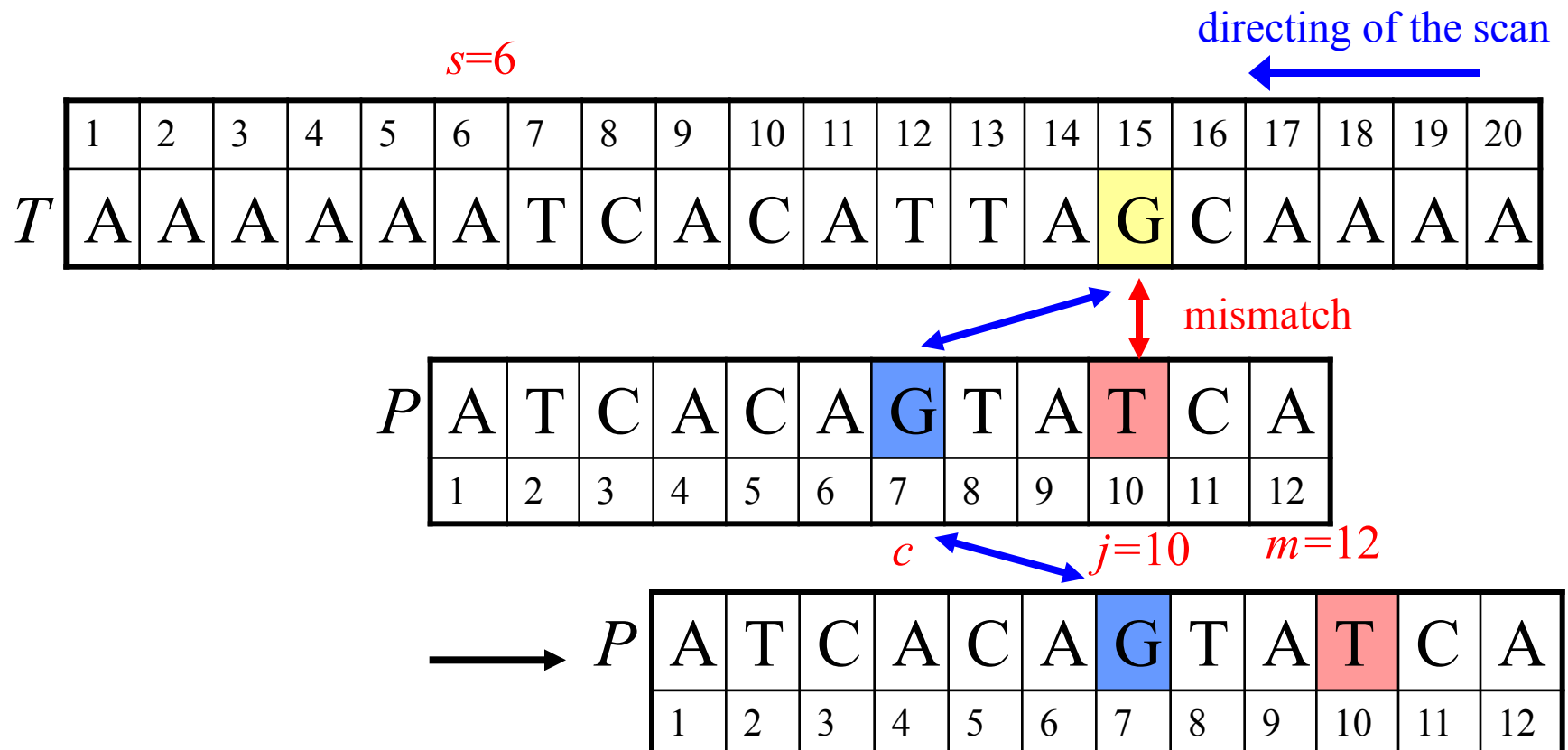
- Case 1. If there is a  $x$  in  $P$  to the left of  $T$ , move  $P$  so that the two  $x$ 's match.



- Case 2: If no such a  $x$  exists in  $P$ , consider the partial window defined by  $x$  in  $T$  and the string to the left of it.

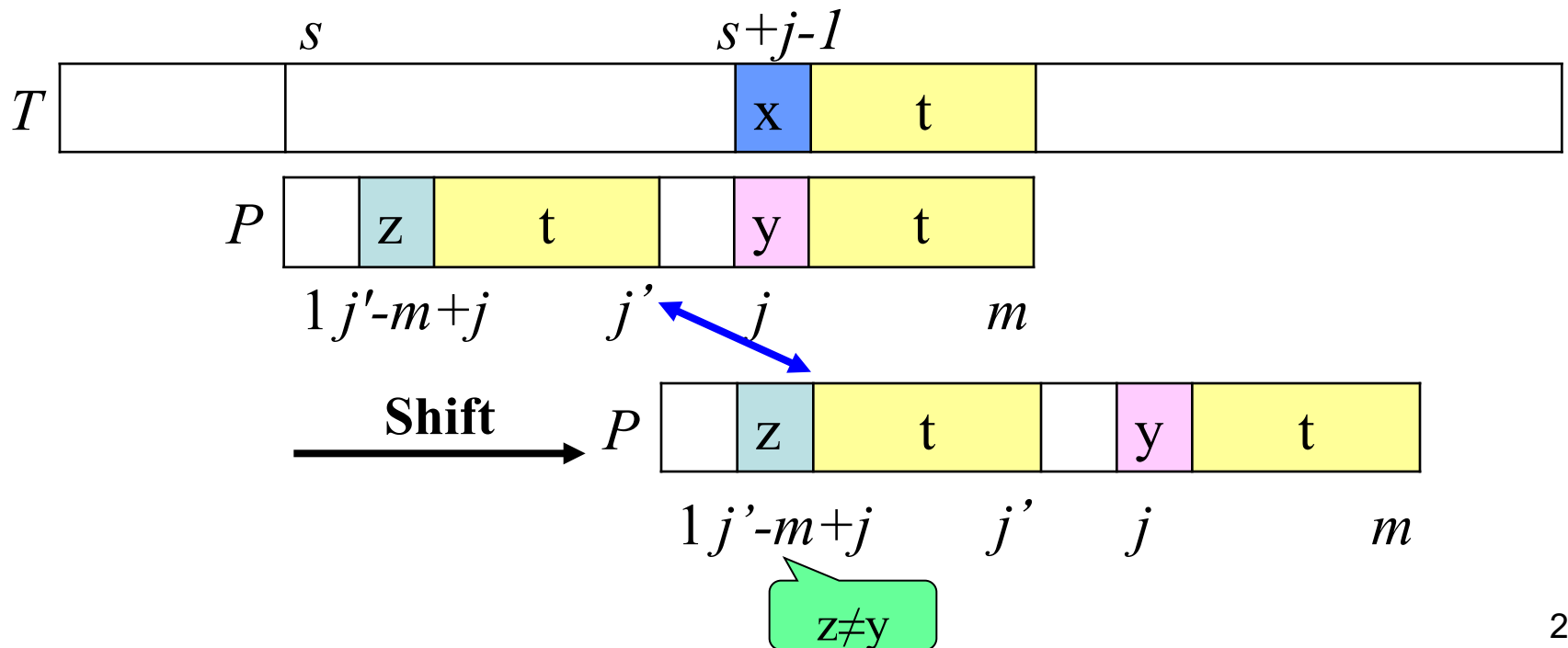


- Ex: Suppose that  $P_l$  is aligned to  $T_6$  now. We compare pairwise between  $T$  and  $P$  from right to left. Since  $T_{16,17} = P_{11,12} = \text{"CA"}$  and  $T_{15} = \text{"G"} \neq P_{10} = \text{"T"}$ . Therefore, we find the rightmost position  $c=7$  in the left of  $P_{10}$  in  $P$  such that  $P_c$  is equal to "G" and we can move the window at least  $(10-7=3)$  positions.



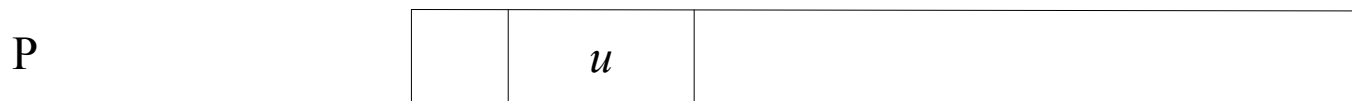
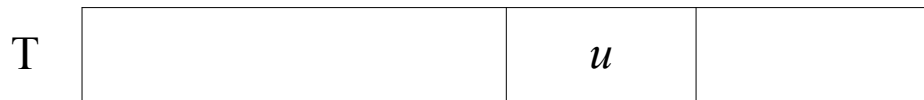
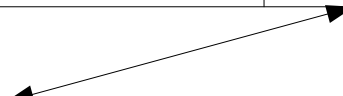
# Good Suffix Rule 1

- If a mismatch occurs in  $T_{s+j-l}$ , we match  $T_{s+j-l}$  with  $P_{j'-m+j}$ , where  $j'$  ( $m-j+1 \leq j' < m$ ) is the **largest position** such that
  - $P_{j+1,m}$  is a suffix of  $P_{1,j'}$
  - $P_{j'-(m-j)} \neq P_j$
- We can move the window at least  $(m-j')$  position(s).

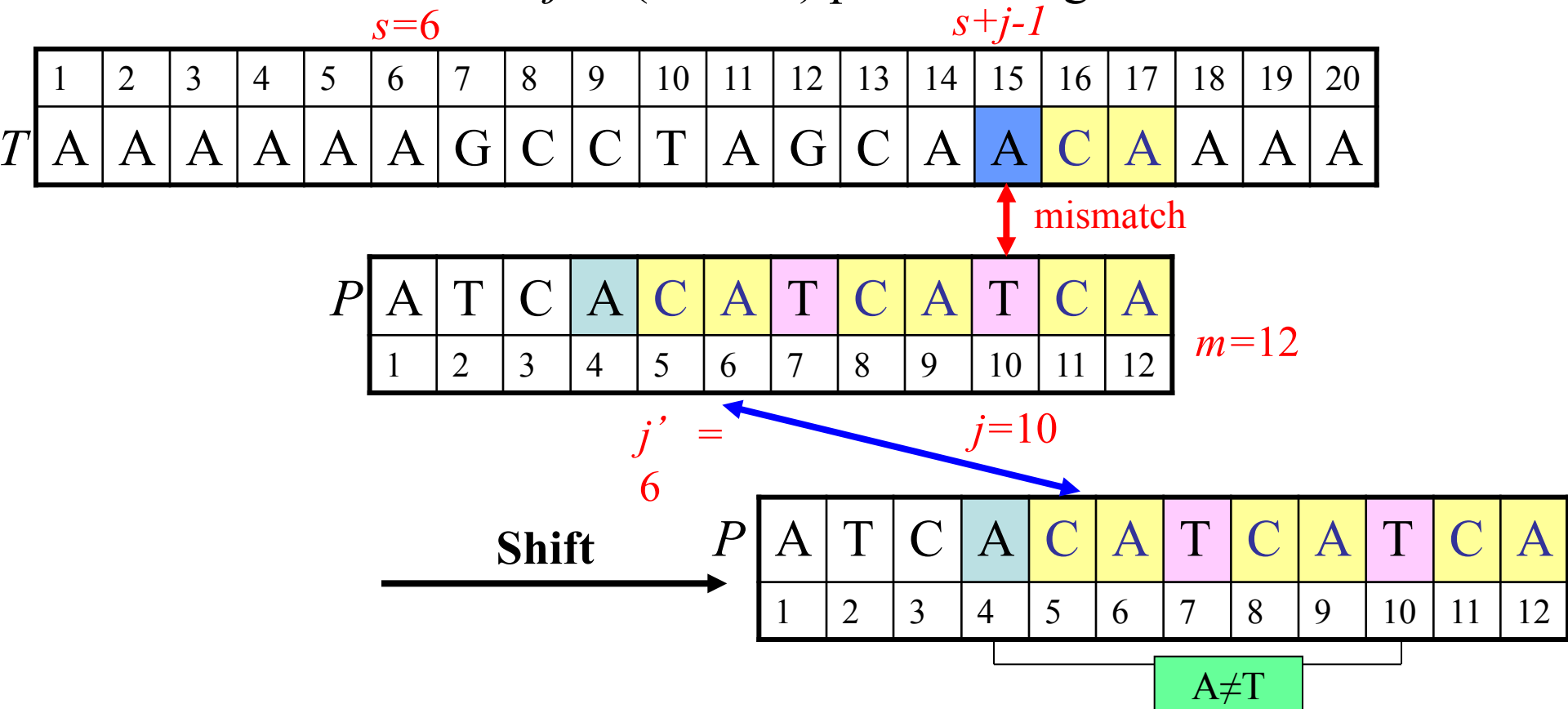


# Rule: The Substring Matching Rule

- For any substring  $u$  in  $T$ , find a nearest  $u$  in  $P$  which is to the left of it. If such a  $u$  in  $P$  exists, move  $P$ ; otherwise, we may define a new partial window.



- Ex: Suppose that  $P_l$  is aligned to  $T_6$  now. We compare pairwise between  $P$  and  $T$  from right to left. Since  $T_{16,17} = \text{"CA"} = P_{11,12}$  and  $T_{15} = \text{"A"} \neq P_{10} = \text{"T"} . We find the substring \text{"CA"} in the left of  $P_{10}$  in  $P$  such that \text{"CA"} is the suffix of  $P_{1,6}$  and the left character to this substring \text{"CA"} in  $P$  is not equal to  $P_{10} = \text{"T"} . Therefore, we can move the window at least  $m-j'$  (12-6=6) positions right.$$

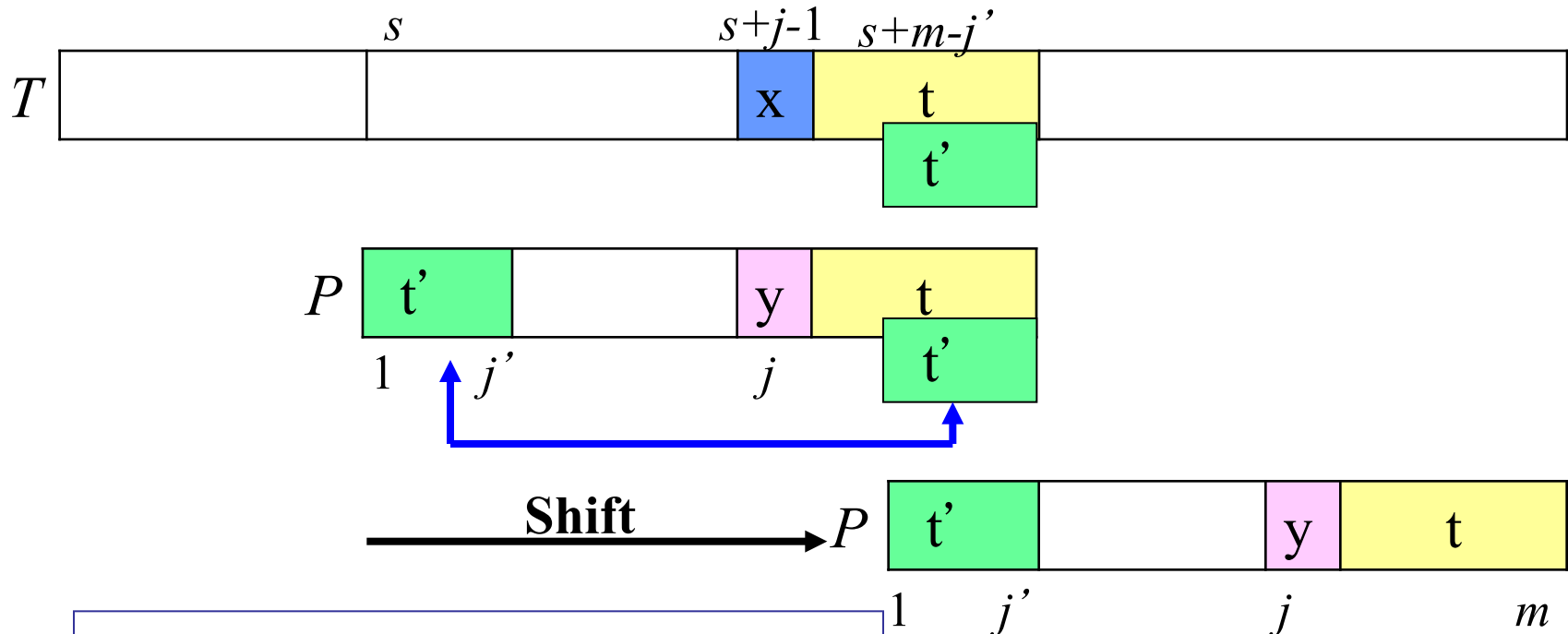


# Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 can not be used. That is,  $t$  does not appear in  $P(1, j)$ . Thus,  $t$  is **unique** in  $P$ .

- If a mismatch occurs in  $T_{s+j-1}$ , we match  $T_{s+m-j'}$  with  $P_1$ , where  $j'$  ( $1 \leq j' \leq m-j$ ) is **the largest position** such that

$P_{1,j'}$  is a suffix of  $P_{j+1,m}$ .

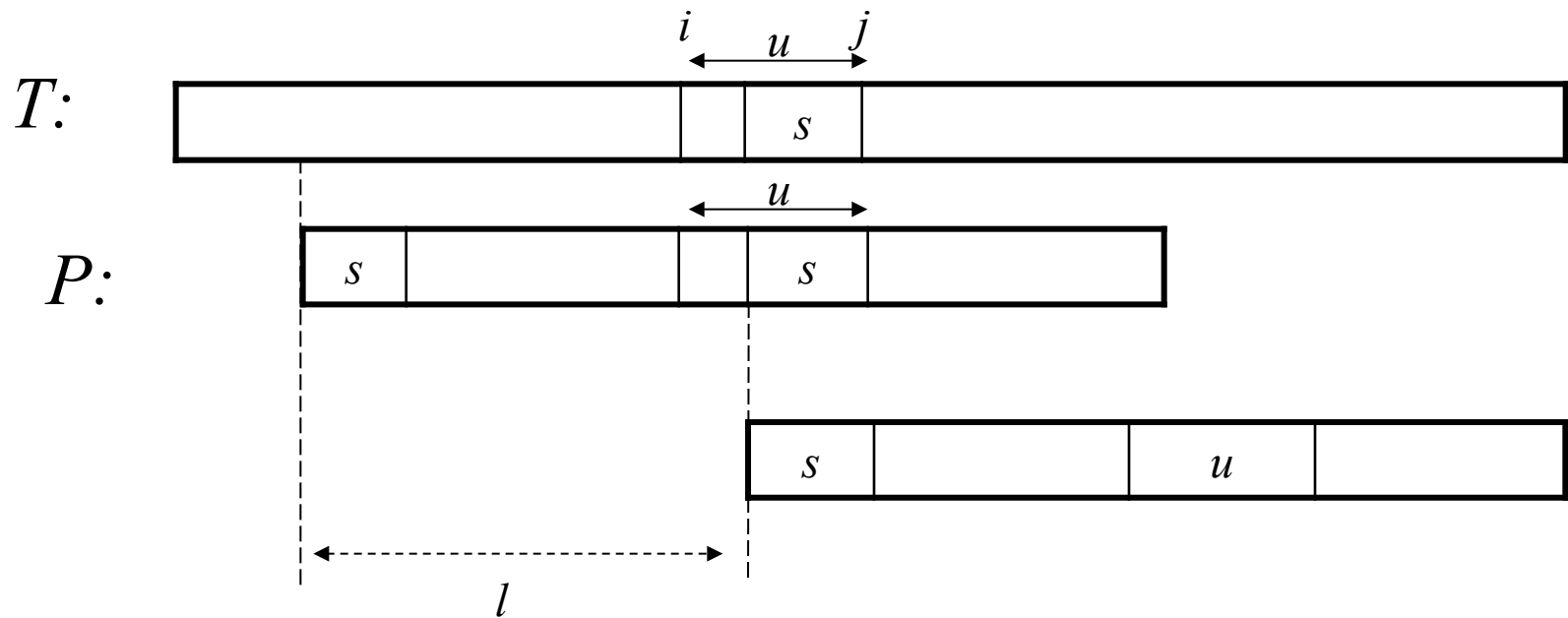


P.S. :  $t'$  is suffix of substring  $t$ .



# Rule: Unique Substring Rule

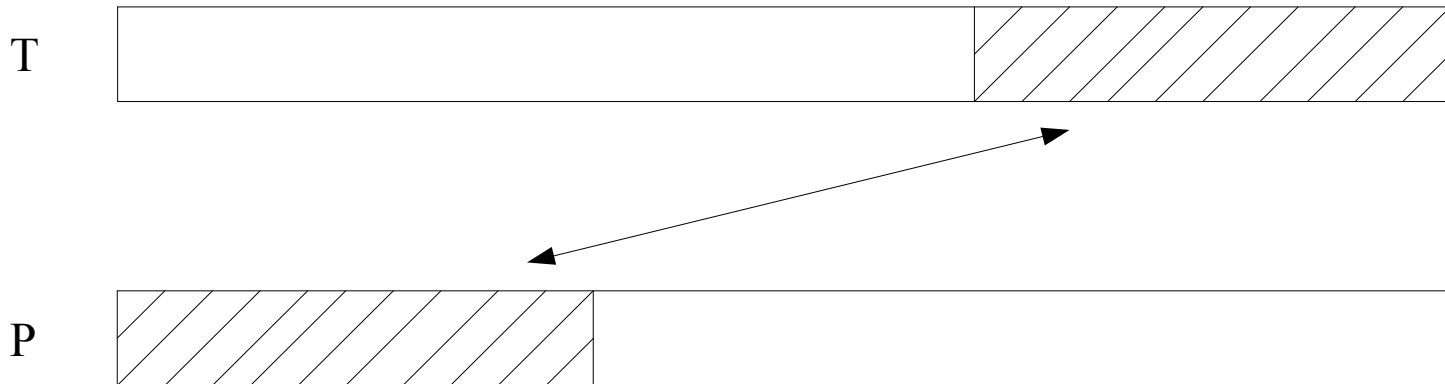
- The substring  $u$  appears in  $P$  exactly once.
- If the substring  $u$  matches with  $T_{i,j}$ , no matter whether a mismatch occurs in some position of  $P$  or not, we can slide the window by  $l$ .



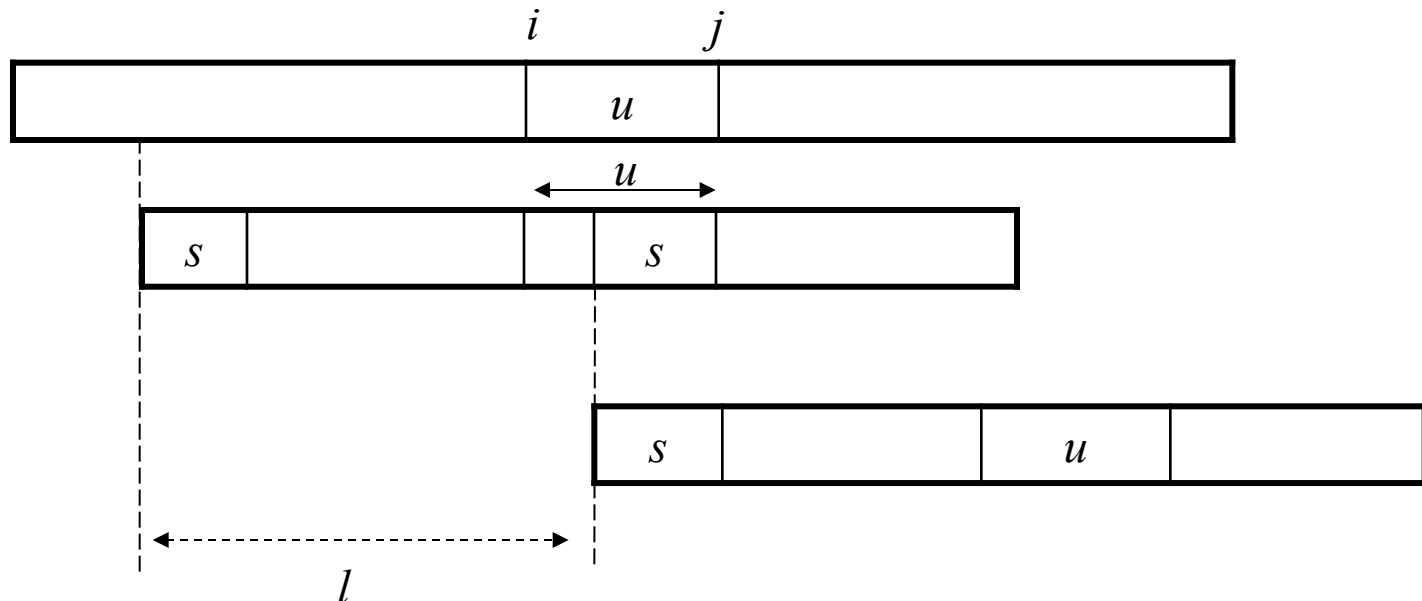
The string  $s$  is the longest prefix of  $P$  which equals to a suffix of  $u$ .

# The Suffix to Prefix Rule

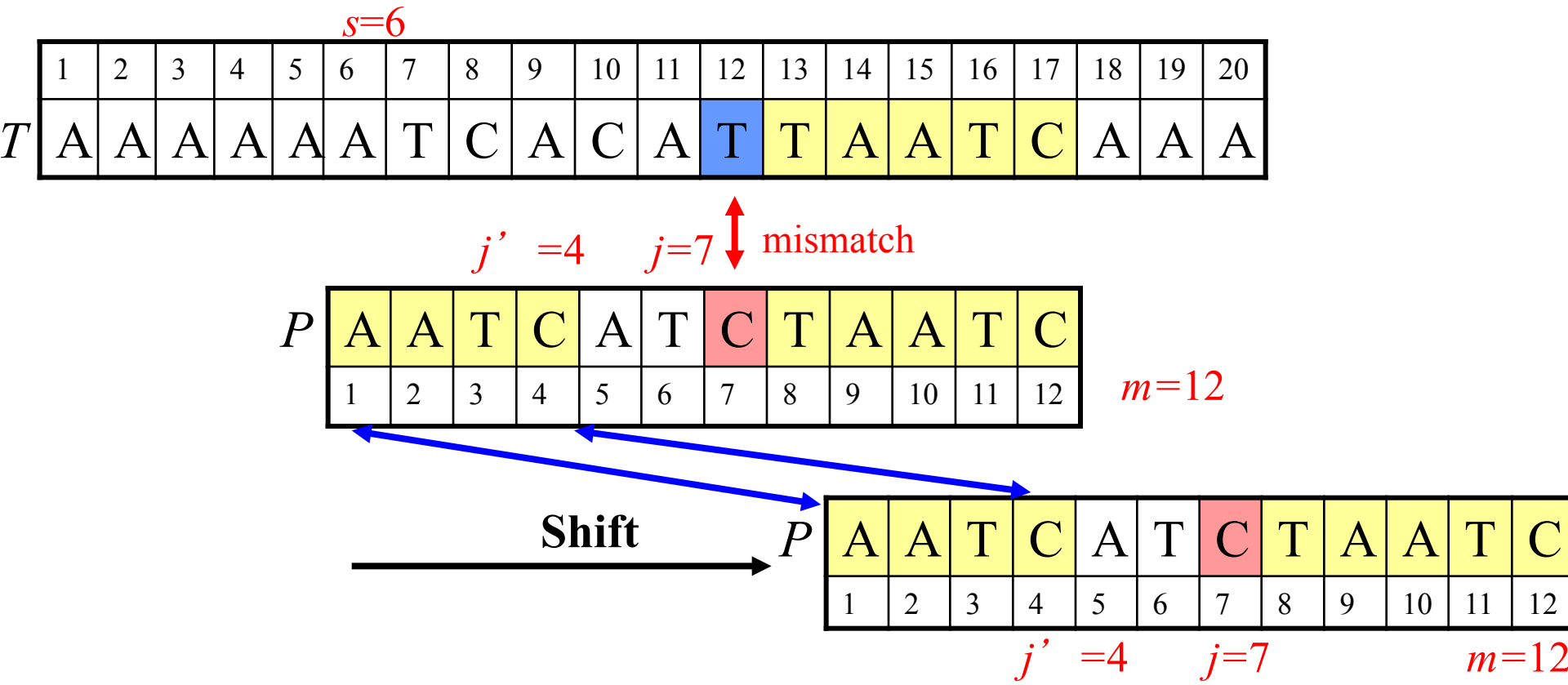
- For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.



- Note that the above rule also uses Rule 1.
- It should also be noted that the unique substring is the shorter and the more right-sided the better.
- A short  $u$  guarantees a short (or even empty)  $s$  which is desirable.



- Ex: Suppose that  $P_1$  is aligned to  $T_6$  now. We compare pair-wise between  $P$  and  $T$  from right to left. Since  $T_{12} \neq P_7$  and there is no substring  $P_{8,12}$  in left of  $P_8$  to exactly match  $T_{13,17}$ . We find a longest suffix “AATC” of substring  $T_{13,17}$ , the longest suffix is also prefix of  $P$ . We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least  $12-4=8$  positions.



- Let  $Bc(a)$  be the rightmost position of  $a$  in  $P$ . The function will be used for applying *bad character rule*.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A

$\Sigma$	A	C	G	T
$B$	12	11	0	10

- We can move our pattern right  $j - B(T_{s+j-1})$  position by above  $Bc$  function.

$j$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$T$	A	G	C	T	A	G	C	C	T	G	C	A	C	G	T	A	C	A

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A

Move  
 $10 - B(G) = 10$  positions

Let  $G_s(j)$  be the largest number of shifts by *good suffix rule* when a mismatch occurs for comparing  $P_j$  with some character in  $T$ .

- $gs_1(j)$  be the largest  $k$  such that  $P_{j+1,m}$  is a suffix of  $P_{1,k}$  and  $P_{k-m+j} \neq P_j$ , where  $m-j+1 \leq k < m$ ; 0 if there is no such  $k$ .

( $gs_1$  is for Good Suffix Rule 1)

- $gs_2(j)$  be the largest  $k$  such that  $P_{1,k}$  is a suffix of  $P_{j+1,m}$ , where  $1 \leq k \leq m-j$ ; 0 if there is no such  $k$ .

( $gs_2$  is for Good Suffix Rule 2.)

- $Gs(j) = m - \max\{gs_1, gs_2\}$ , if  $j = m$ ,  $Gs(j)=1$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$gs_1$	0	0	0	0	0	0	9	0	0	6	1	0
$gs_2$	4	4	4	4	4	4	4	4	1	1	1	0
$Gs$	8	8	8	8	8	8	3	8	11	6	11	1

$$gs_1(7)=9$$

$\because P_{8,12}$  is a suffix of  $P_{1,9}$   
and  $P_4 \neq P_7$

$$gs_2(7)=4$$

$\because P_{1,4}$  is a suffix of  $P_{8,12}$

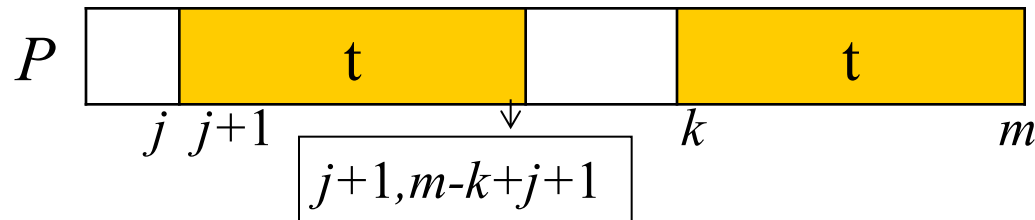
How do we obtain  $gs_1$  and  $gs_2$ ?

In the following, we shall show that by constructing the **Suffix Function**, we can kill two birds with one arrow.



# Suffix function $f'$

- For  $1 \leq j \leq m-1$ , let the suffix function  $f'(j)$  for  $P_j$  be the **smallest**  $k$  such that  $P_{k,m} = P_{j+1,m-k+j+1}$ ; ( $j+2 \leq k \leq m$ )
  - If there is no such  $k$ , we set  $f' = m+1$ .
  - If  $j=m$ , we set  $f'(m)=m+2$ .



• Ex:

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

- $f'(4)=8$ , it means that  $P_{f'(4),m} = P_{8,12} = P_{5,9} = P_{4+1,4+1+m-f'(4)}$
- Since there is no  $k$  for  $13=j+2 \leq k \leq 12$ , we set  $f'(11)=13$ .

Suppose that the Suffix is obtained. How can we use it to obtain  $gs_1$  and  $gs_2$ ?

$gs_1$  can be obtained by scanning the Suffix function from right to left.

# Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$T$	G	A	T	C	G	A	T	C	A	A	T	C	A	T	C	A	C	A	T	G	A	T	C	A

$P$	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

# Example

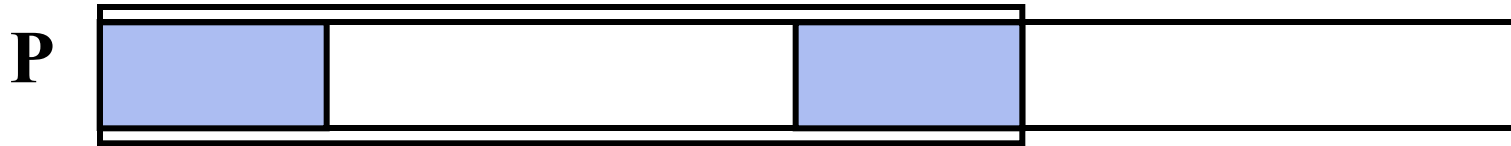
As for Good Suffix Rule 2, it is relatively easier.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

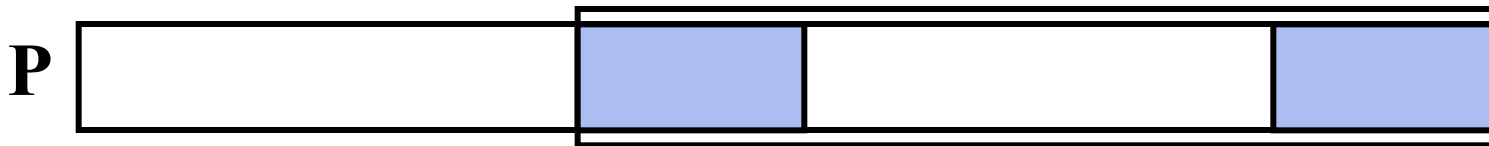
Question: How can we construct the Suffix function?

To explain this, let us go back to the prefix function used in the KMP Algorithm.

The following figure illustrates the prefix function in the KMP Algorithm.



The following figure illustrates the suffix function of the BM Algorithm.



We now can see that actually the suffix function is the same as the prefix. The only difference is now we consider a suffix. Thus, the recursive formula for the prefix function in KMP Algorithm can be slightly modified for the suffix function in BM Algorithm.

- The formula of suffix function  $f'$  as follows :

Let  $f'^x(y) = f'(f'^{x-1}(y))$  for  $x > 1$  and  $f'^1(y) = f'(y)$

$$f'(j) = \begin{cases} m+2, & \text{if } j = m \\ f'^k(j+1)-1, & \text{if } 1 \leq j \leq m-1 \text{ and there exists the smallest} \\ & k \geq 1 \text{ such that } P_{j+1} = P_{f'^k(j+1)-1}; \\ m+1, & \text{otherwise} \end{cases}$$



$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$												14

$j=m=12,$   
 $f' = m+2=14$   
 4

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$											13	14

No  $k$  satisfies  
 $P_{j+1} = P_{f' - k(j+1) - 1},$   
 $f' = m+1 = 12+1 = 13$

$k=1 \rightarrow$   
 $P_{12} \neq P_{13}$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$										13	13	14

No  $k$  satisfies  
 $P_{j+1} = P_{f' \ k(j+1)-1}$ ,  
 $f' = m+1 = 12+1 = 13$

$k=1 \rightarrow$   
 $P_{11} \neq P_{12}$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$									13	13	13	14

No  $k$  satisfies  
 $P_{j+1} = P_{f' \ k(j+1)-1}$ ,  
 $f' = m+1 = 12+1 = 13$

$k=1 \rightarrow$   
 $P_{10} \neq P_{12}$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$								12	13	13	13	14

$$\because P_{j+1} = P_{f' (j+1)-1} \Rightarrow P_9 = P_{12},$$

$$f' = f' (j+1) - 1 = 13 - 1 = 12$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$							11	12	13	13	13	14

$$\because P_{j+1} = P_{f' (j+1)-1} \Rightarrow P_8 = P_{11},$$

$$f' = f' (j+1) - 1 = 12 - 1 = 11$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$				8	9	10	11	12	13	13	13	14

$$\because P_{j+1} = P_{f' - 1(j+1)-1} \Rightarrow P_5 = P_8, \\ f' = f' (j+1) - 1 = 9 - 1 = 8$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$			12	8	9	10	11	12	13	13	13	14

$$\because P_{j+1} = P_{f' - 3(j+1)-1} \Rightarrow P_4 = P_{f' - 3(4)-1} = P_{12}, \\ f' = f' - 3(j+1) - 1 = 13 - 1 = 12$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$		11	12	8	9	10	11	12	13	13	13	14

$$\because P_{j+1} = P_{f' \ (j+1)-1} \Rightarrow P_3 = P_{f' \ (3)-1} = P_{11},$$

$$f' = f' \ (j+1) - 1 = 12 - 1 = 11$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

$$\because P_{j+1} = P_{f' \ (j+1)-1} \Rightarrow P_2 = P_{f' \ (2)-1} = P_{10},$$

$$f' = f' \ (j+1) - 1 = 11 - 1 = 10$$

- Let  $G'(j)$ ,  $1 \leq j \leq m$ , to be the largest number of shifts by good suffix rules.
- First, we set  $G'(j)$  to zeros as their initializations.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	0	0	0	0	0	0

- Step1:** We scan from right to left and  $gs_1(j)$  is determined during the scanning, then  $gs_1(j) \geq gs_2(j)$

**Observe:**  
 If  $P_j=P_4 \neq P_7=P_{f'(j)-1}$ , we know  $gs_1(f'(j)-1)=m+j-f'(j)+1=9$ .  
 If  $t=f'(j)-1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = m - gs_1(f'(j)-1) = f'(j) - 1 - j$ .  
 $f^{(k)}(x)=f^{(k-1)}(f'(x) - 1), k \geq 2$

- When  $j=12, t=13. \ t > m$ .
- When  $j=11, t=12$ . Since  $P_{11}= 'C' \neq 'A' = P_{12}$ ,  
 $G'(t) = m - \max \{gs_1(t), gs_2(t)\} = m - \underline{gs_1(t)}$   
 $\qquad \qquad \qquad = f'(j) - 1 - j$   
 $\Rightarrow G'(12)=13-1-11= 1.$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	0	0	0	0	0	1

If  $t = f'(j) - 1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = f'(j) - 1 - j$ .  
 $f^{(k)}(x) = f^{(k-1)}(f'(x) - 1)$ ,  $k \geq 2$

- When  $j=10$ ,  $t=12$ . Since  $P_{10} = 'T' \neq 'A' = P_{12}$ ,  $G'(12) \neq 0$ .
- When  $j=9$ ,  $t=12$ .  $P_9 = 'A' = P_{12}$ .
- When  $j=8$ ,  $t=11$ .  $P_8 = 'C' = P_{11}$ .
- When  $j=7$ ,  $t=10$ .  $P_7 = 'T' = P_{10}$ .
- When  $j=6$ ,  $t=9$ .  $P_6 = 'A' = P_9$ .
- When  $j=5$ ,  $t=8$ .  $P_5 = 'C' = P_8$ .
- When  $j=4$ ,  $t=7$ . Since  $P_4 = 'A' \neq P_7 = 'T'$ ,  $G'(7) = 8 - 1 - 4 = 3$

Besides,  $t = f'^{(2)}(4) - 1 = f'(f'(4) - 1) - 1 = 10$ . Since  $P_4 = 'A' \neq P_{10} = 'T'$ ,  $G'(10) = f'(7) - 1 - j = 11 - 1 - 4 = 6$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	0	6	0	1



If  $t = f'(j) - 1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = f'(j) - 1 - j$ .  
 $f^{(k)}(x) = f^{(k-1)}(f'(x) - 1)$ ,  $k \geq 2$

- When  $j=3$ ,  $t=11$ .  $P_3 = \text{'C'} = P_{11}$ .
- When  $j=2$ ,  $t=10$ .  $P_2 = \text{'T'} = P_{10}$ .
- When  $j=1$ ,  $t=9$ .  $P_1 = \text{'A'} = P_9$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	6	0	0	1

- By the above discussion, we can obtain the values using the Good Suffix Rule 1 by scanning the pattern from right to left.

- **Step2:** Continuously, we will try to obtain the values using *Good Suffix Rule 2* and those values are still zeros now and scan from left to right.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	0	6	0	1

- Let  $k'$  be the **smallest**  $k$  in  $\{1, \dots, m\}$  such that  $P_{f'(k)}(1) = P_1$  and  $f'(k)(1) - 1 \leq m$ .

**Observe:**

$\because P_{1,4} = P_{9,12}, \therefore gs_2(j) = m - (f'(1) - 1) + 1 = 4$ , where  $1 \leq j \leq f'(k')(1) - 2$ .

- If  $G'(j)$  is not determined in the first scan and  $1 \leq j \leq f'(k')(1) - 2$ , thus, in the second scan, we set  $G'(j) = m - \max\{gs_1(j), gs_2(j)\} = m - gs_2(j) = f'(k')(1) - 2$ . If no such  $k$  exists, set each undetermined value of  $G$  to  $m$  in the second scan.
- $k=1=k'$ , since  $P_{f'(1)}(1) = P_9 = \text{"A"} = P_1$ , we set  $G'(j) = f'(1) - 2$  for  $j=1, 2, 3, 4, 5, 6, 8$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	0	6	0	1

- Let  $z$  be  $f^{(k')}(1)-2$ . Let  $k''$  be the **largest value  $k$**  such that  $f^{(k)}(z)-1 \leq m$ .
- Then we set  $G'(j) = m - gs_2(j) = m - (m - f^{(i)}(z) - 1) = f^{(i)}(z) - 1$ , where  $1 \leq i \leq k''$  and  $f^{(i-1)}(z) < j \leq f^{(i)}(z) - 1$  and  $f^{(0)}(z) = z$ .
- For example,  $z=8$  :
  - $k=1, f^{(1)}(8)-1=11 \leq m=12$
  - $k=2, f^{(2)}(8)-1=12 \leq m=12 \Rightarrow k''=2$
  - $i=1, f^{(0)}(8)-1 = 7 < j \leq f^{(1)}(8)-1=11$ .
  - $i=2, f^{(1)}(8)-1 = 11 < j \leq f^{(2)}(8)-1=12$ .
  - We set  $G(9)$  and  $G(11)=f^{(1)}(8) - 1 = 12-1 = 11$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

We essentially have to decide the maximum number of steps.  
We can move the window right when a mismatch occurs. This is decided by the following function:

$$\max \{G'(j), j - B(T_{s+j-1})\}$$

# Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$T$	G	A	T	C	G	A	T	C	A	C	A	T	A	T	C	A	C	A	T	C	A	T	C	A

mismatch



$P$	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

$P$	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

$\Sigma$	A	C	G	T
$B$	12	11	0	10

We compare  $T$  and  $P$  from right to left. Since  $T_{12} = \text{"T"} \neq P_{12} = \text{"A"}$ , the largest movement =  $\max\{G'(j), j - B(T_{s+j-l})\} = \max\{G'(12), 12 - B(T_{12})\} = \max\{1, 12 - 10\} = 2$ .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$T$	G	A	T	C	G	A	T	C	A	C	A	T	A	T	C	A	C	A	T	C	A	T	C	A

mismatch



$P$	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

$P$	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

$\Sigma$	A	C	G	T
$B$	12	11	0	10

After moving, we compare  $T$  and  $P$  from right to left. Since  $T_{14} = \text{"T"} \neq P_{12} = \text{"A"}$ , the largest movement =  $\max\{G'(j), j - B(T_{14})\} = \max\{G'(12), 12 - B(T_{14})\} = \max\{1, 12 - 10\} = 2$ .

# Time Complexity

- The preprocessing phase in  $O(m+\Sigma)$  time and space complexity and searching phase in  $O(mn)$  time complexity.
- The worst case time complexity for the *Boyer-Moore* method would be  $O((n-m+1)m)$ .
- It was proved that this algorithm has  $O(m)$  comparisons when  $P$  is not in  $T$ . However, this algorithm has  $O(mn)$  comparisons when  $P$  is in  $T$ .



# Reference

- **Algorithms for finding patterns in strings** , AHO, A.V., **Handbook of Theoretical Computer Science** , Volume A , Chapter 5 Elsevier , Amsterdam , 1990, pp. 255-300.
- **Computer algorithms: string pattern matching strategies** , Jun-ichi, A. , IEEE Computer Society Press , 1994.
- **Computer Algorithms: Introduction to Design and Analysis** , BAASE, S. and VAN GELDER, A. , **Addison-Wesley Publishing Company** , Chapter 11 , 1999.
- **Indexing and Searching** , BAEZA-YATES, R. , NAVARRO, G. and RIBEIRO-NETO, B. , **Modern Information Retrieval** , Chapter 8 , 1999 , pp. 191-228.
- **Éléments d'algorithmique** , BEAUQUIER, D., BERSTEL, J. and CHRÉTIENNE, P., Masson Paris , Chapter 10 , 1992 , pp. 337-377.
- **A fast string searching algorithm** , BOYER R.S. and MOORE J.S. , **Communications of the ACM** , Vol 20 , 1977 , pp. 762-772 .
- **Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm** , COLE, R., **SIAM Journal on Computing** , Vol 23 , 1994 , pp. 1075-1091.
- **Introduction to Algorithms** , CORMEN, T.H. , LEISERSON, C.E. and RIVEST, R.L. , **The MIT Press** , Chapter 34 , 1990 , pp. 853-885.
- **Off-line serial exact string searching** , CROCHEMORE, M. , **Pattern Matching Algorithms** , Chapter 1 , 1997 , pp 1-53
- **Pattern Matching in Strings** , CROCHEMORE, M. and HANCART, C. , **Algorithms and Theory of Computation Handbook** , Chapter 11, 1999 , pp. 11-1-11-28.

- **Pattern matching and text compression algorithms** , CROCHEMORE, M. and LECROQ, T. , **CRC Computer Science and Engineering Handbook** , Chapter 8 , 1996 , pp. 162-202.
- **Text Algorithms**, CROCHEMORE, M. and RYTTER, W., **Oxford University Press** , 1994.
- **Handbook of Algorithms and Data Structures in Pascal and C**, GONNET, G.H. and BAEZA-YATES, R.A. , **Addison-Wesley Publishing Company** , Chapter 7 , 1991 , pp. 251-288,.
- **Data Structures and Algorithms in JAVA** , GOODRICH, M.T. and TAMASSIA, R. , **John Wiley & Sons** , Chapter 11 , 1998 , pp. 441-467.
- **Algorithms on strings, trees** , GUSFIELD, D. , **Cambridge University Press** , 1997.
- **Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte** , HANCART, C., University Paris 7, France , 1993.
- **Fast pattern matching in strings** , KNUTH, D.E. , MORRIS, J.H. and PRATT, V.R. , **SIAM Journal on Computing** , 1977 , pp.323-350.
- LECROQ, T., 1992, *Recherches de mot.*, Thesis, University of Orléans, France.
- **Experimental results on string matching algorithms** , LECROQ, T. , **Software - Practice & Experience** , Vol 25 , 1995 , pp. 727-765.
- **Algorithms** , SEDGEWICK, R., **Addison-Wesley Publishing Company** , Chapter 19 , 1988 , pp. 277-292.
- **Algorithms in C** , SEDGEWICK, R. , **Addison-Wesley Publishing Company** , Chapter 19 , 1988.
- **String Searching Algorithms** , STEPHEN, G.A. , **World Scientific** , 1994.
- **Taxonomies and Toolkits of Regular Language Algorithms** , WATSON, B.W. , **Eindhoven University of Technology** , 1995.
- **Algorithms & Data Structures** , WIRTH, N. , **Prentice-Hall** , Chapter 1 , 1986 , pp. 17-72.

# Text search

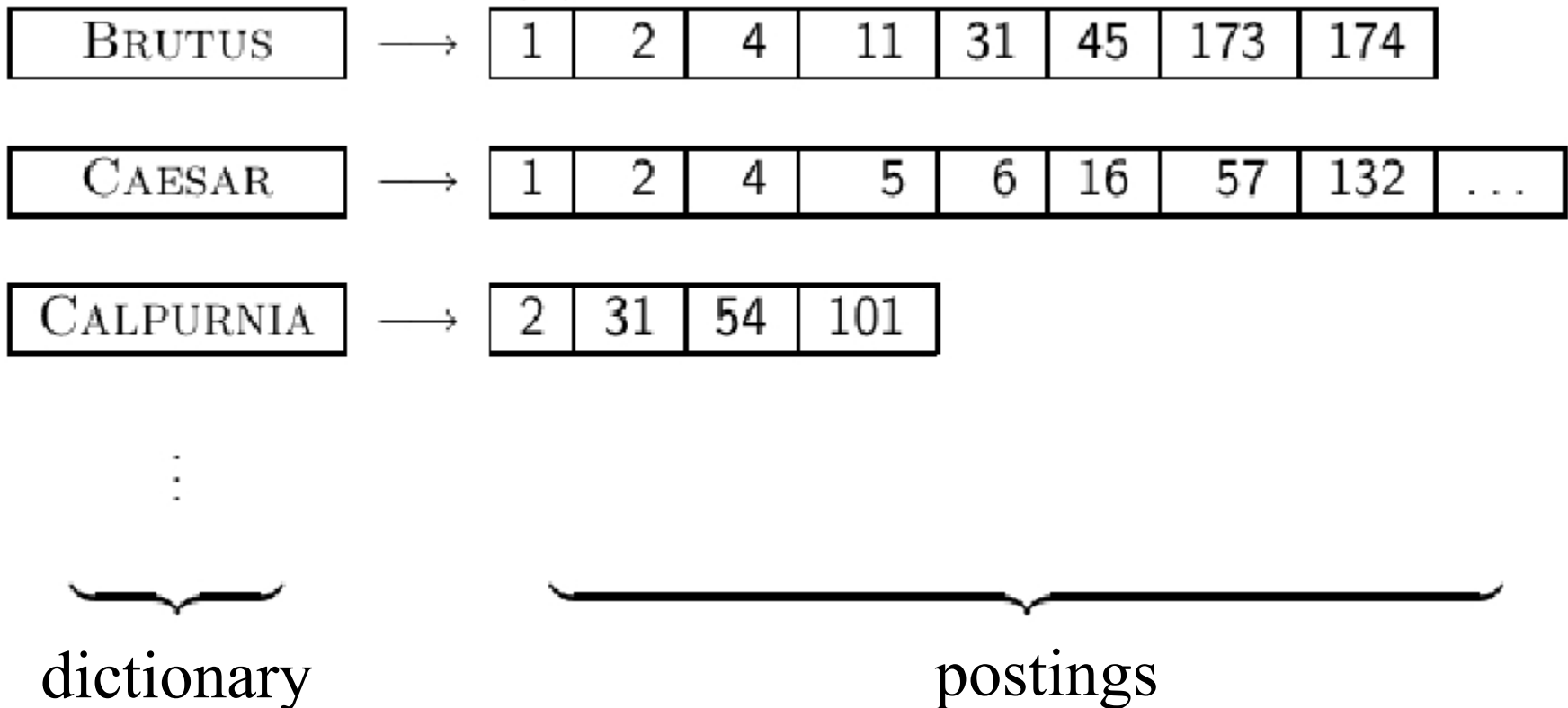
---

- Pattern matching directly
  - Brute force
  - BM
  - Rabin-Karp
  - KMP
- Regular expressions
- Indices for pattern matching
  - Inverted files
  - **Signature files**
  - **Suffix trees** and **Suffix arrays**

# Inverted Index

---

For each term  $t$ , we store a list of all documents that contain  $t$ .



# Create postings lists, determine document frequency

term	docID		term	doc. freq.	→	postings lists
ambitious	2		ambitious	1	→	2
be	2		be	1	→	2
brutus	1		brutus	2	→	1 → 2
brutus	2		capitol	1	→	1
capitol	1		caesar	2	→	1 → 2
caesar	1		did	1	→	1
caesar	2		enact	1	→	1
caesar	2		hath	1	→	2
did	1		i	1	→	1
enact	1		i'	1	→	1
hath	1		it	1	→	2
i	1		julius	1	→	1
i	1		killed	1	→	1
i'	1		let	1	→	2
it	2		me	1	→	1
julius	1		noble	1	→	2
killed	1		so	1	→	2
killed	1		the	2	→	1 → 2
let	2		told	1	→	2
me	1		you	1	→	2
noble	2		was	2	→	1 → 2
so	2		with	1	→	2
the	1					
the	2					
told	2					
you	2					
was	1					
was	2					
with	2					

# Positional indexes

---

- Postings lists in a **nonpositional** index: each posting is just a docID
- Postings lists in a **positional** index: each posting is a docID and **a list of positions**

# Positional indexes: Example

---

Query: “to<sub>1</sub> be<sub>2</sub> or<sub>3</sub> not<sub>4</sub> to<sub>5</sub> be<sub>6</sub>”

TO, 993427:

< 1: <7, 18, 33, 72, 86, 231>;

2: <1, 17, 74, 222, 255>;

4: <8, 16, 190, 429, 433>;

5: <363, 367>;

7: <13, 23, 191>; . . . >

BE, 178239:

< 1: <17, 25>;

4: <17, 191, 291, 430, 434>;

5: <14, 19, 101>; . . . >

Document 4 is a match!

# Signature files

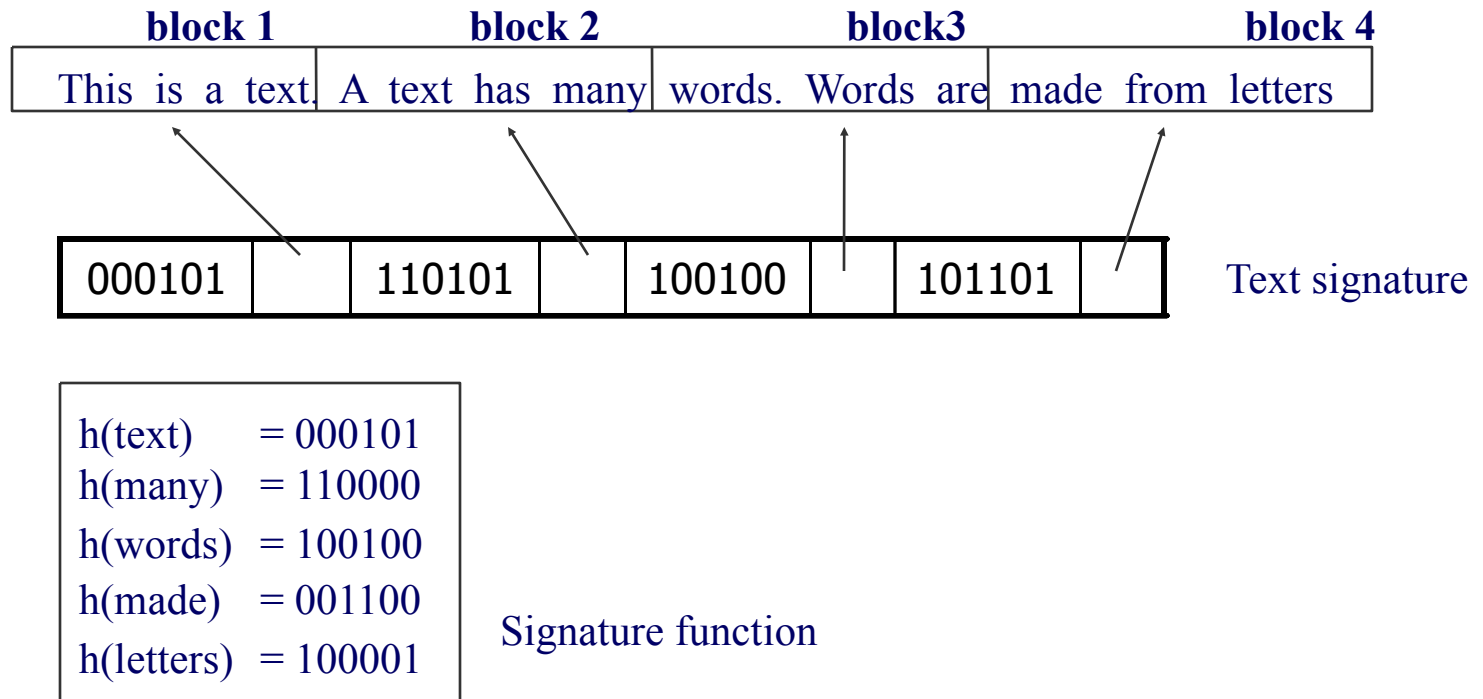
---

- Definition
  - Word-oriented index structure based on hashing.
  - Use liner search.
  - Suitable for not very large texts.
- Structure
  - Based on a Hash function that maps words to bit masks.
  - The text is divided in blocks.
    - **Bit mask of block is obtained by bitwise ORing the signatures of all the words in the text block.**
    - **Word not found, if no match between all 1 bits in the query mask and the block mask.**



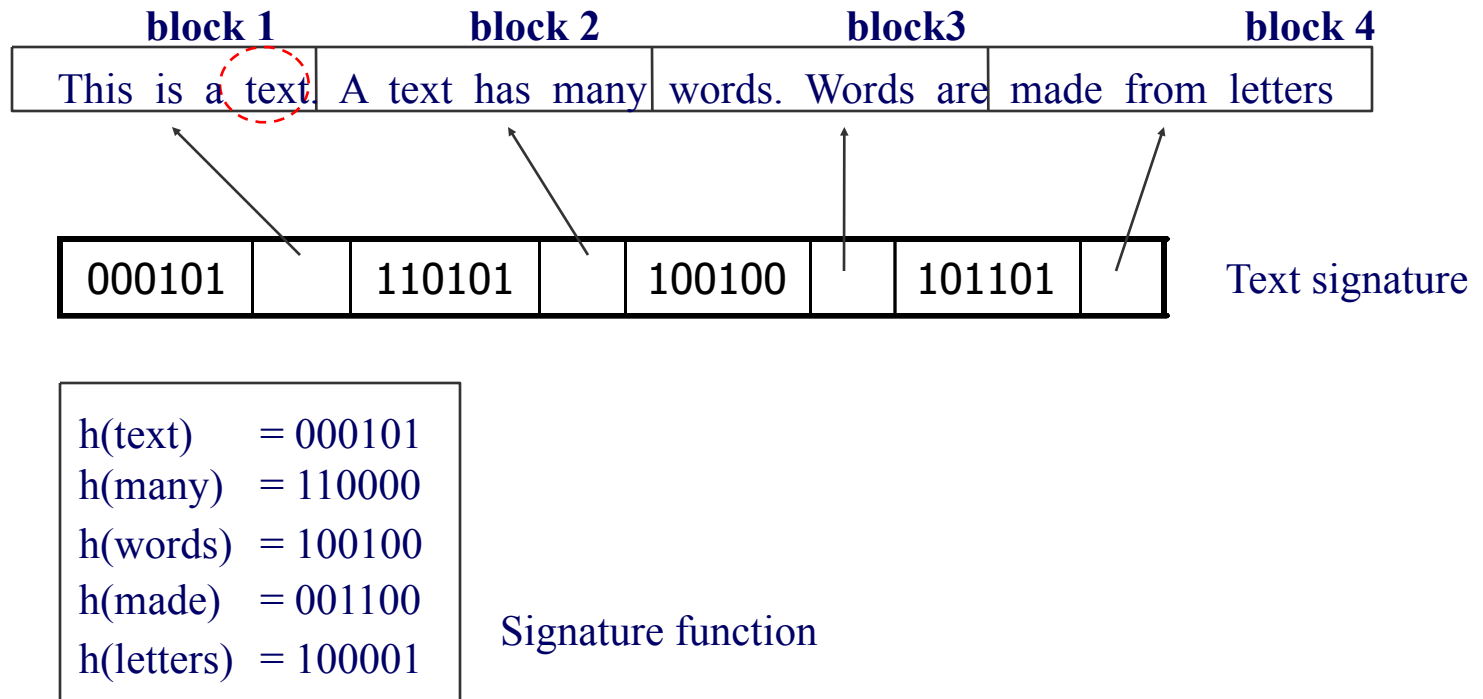
# Signature files

- Example:



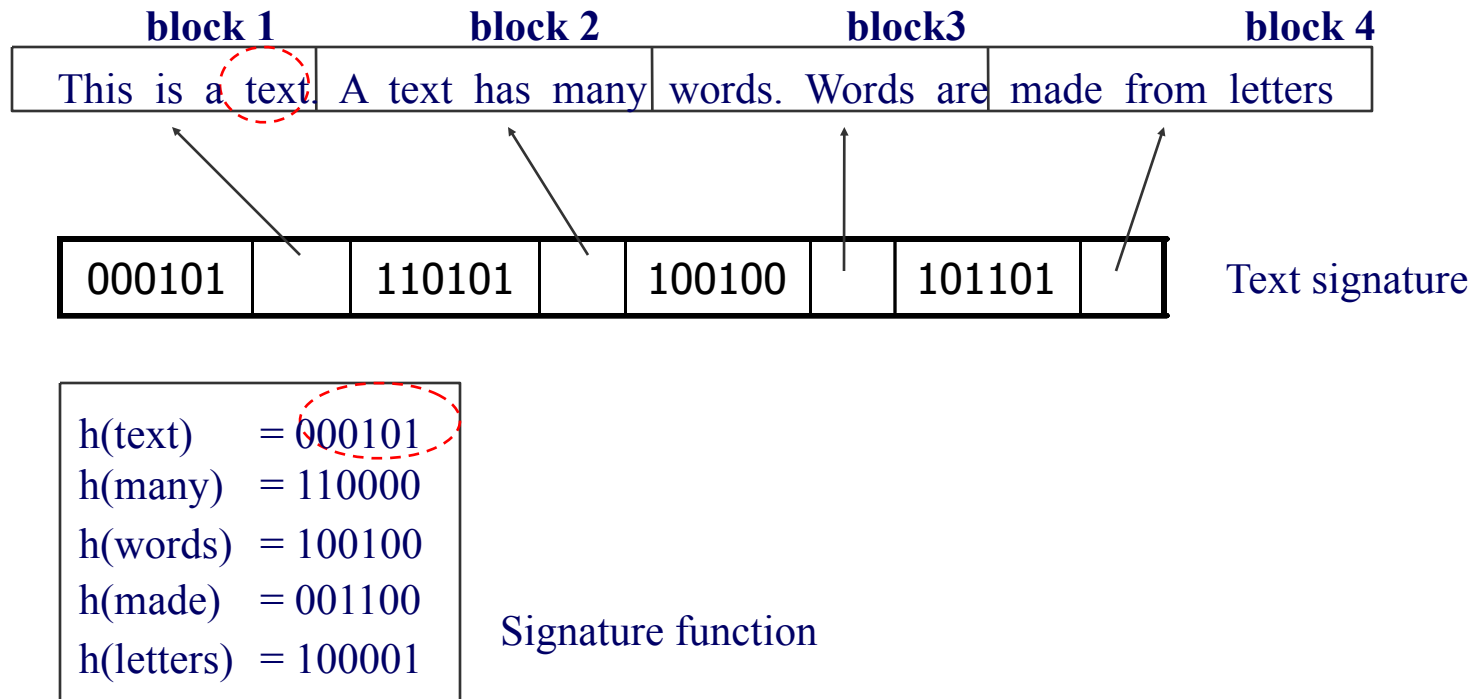
# Signature files

- Example:



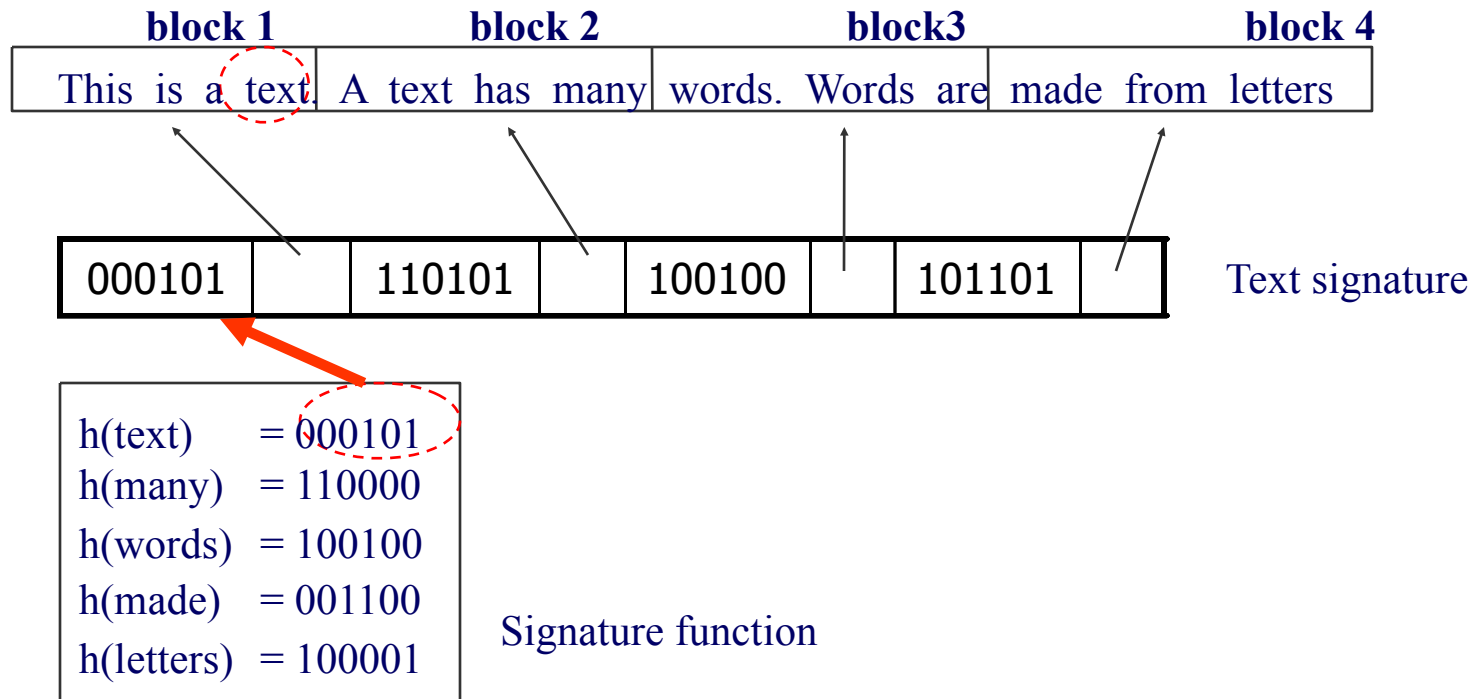
# Signature files

- Example:



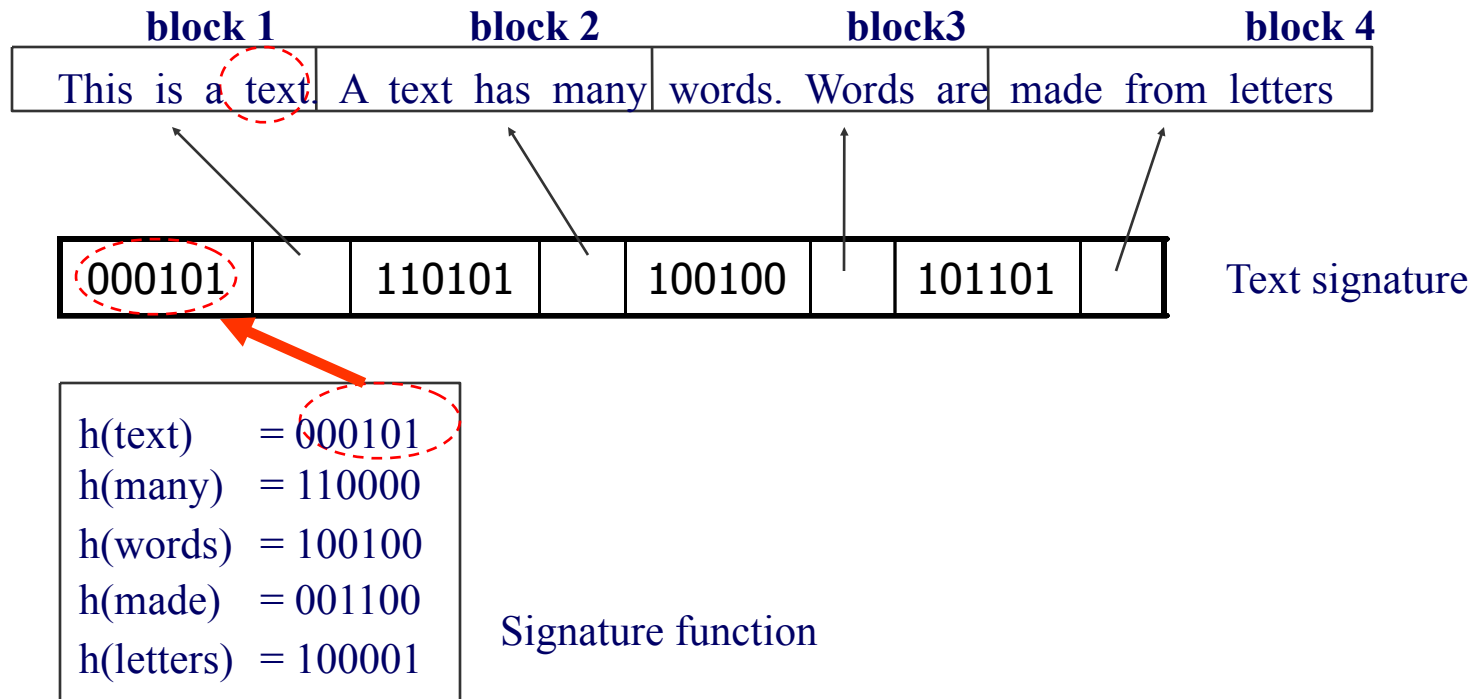
# Signature files

- Example:



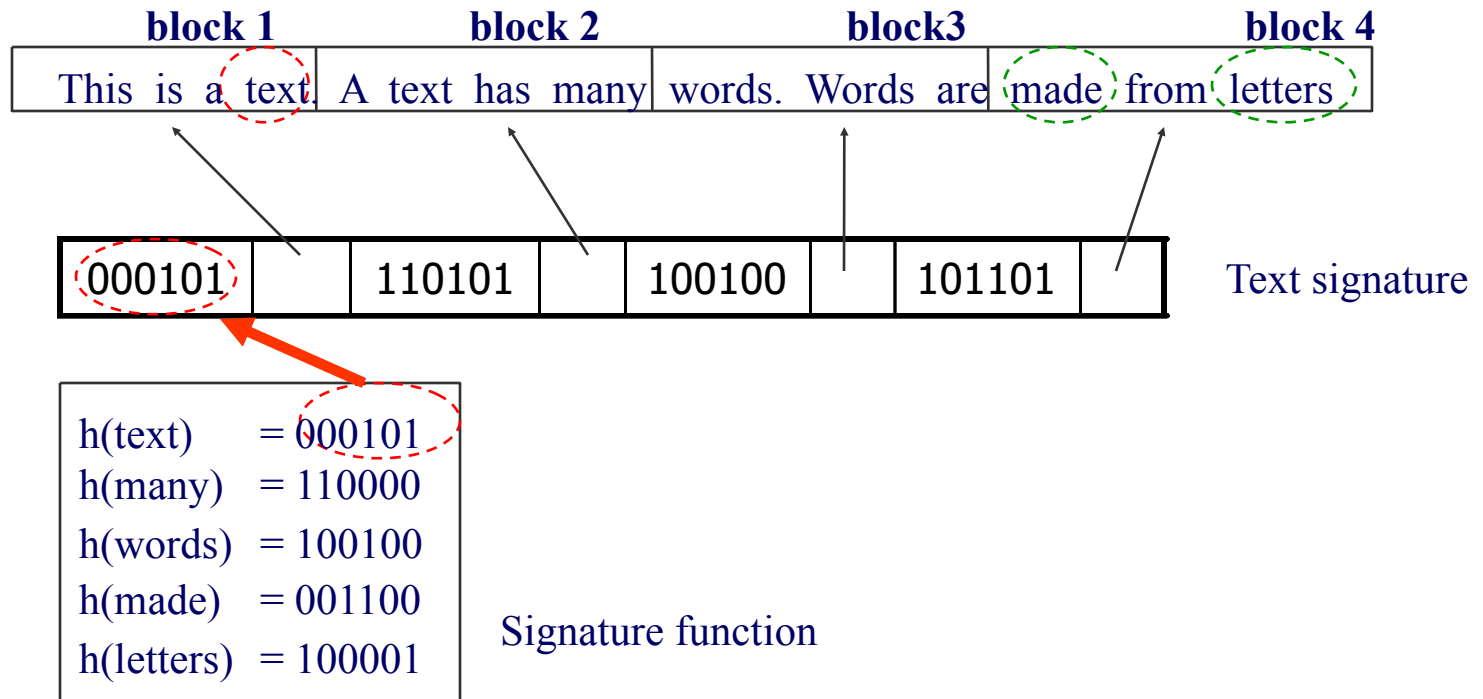
# Signature files

- Example:



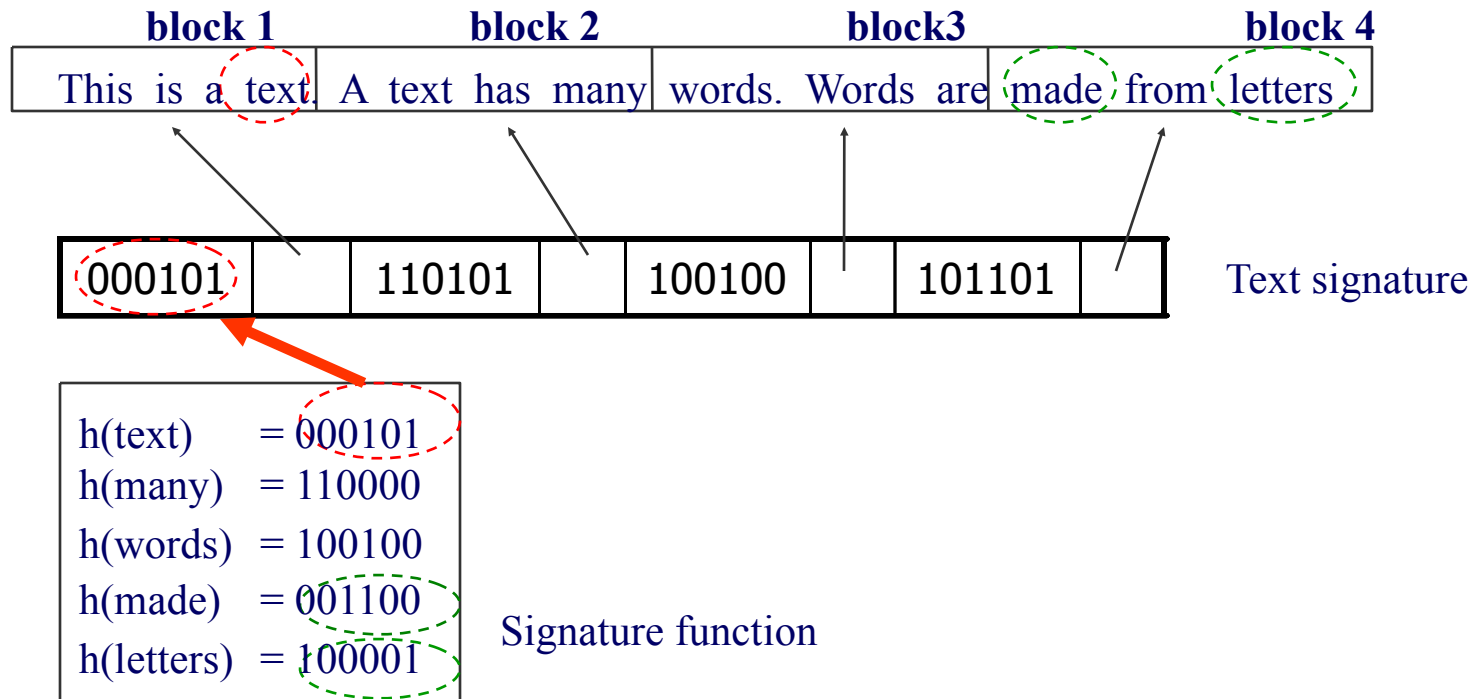
# Signature files

- Example:



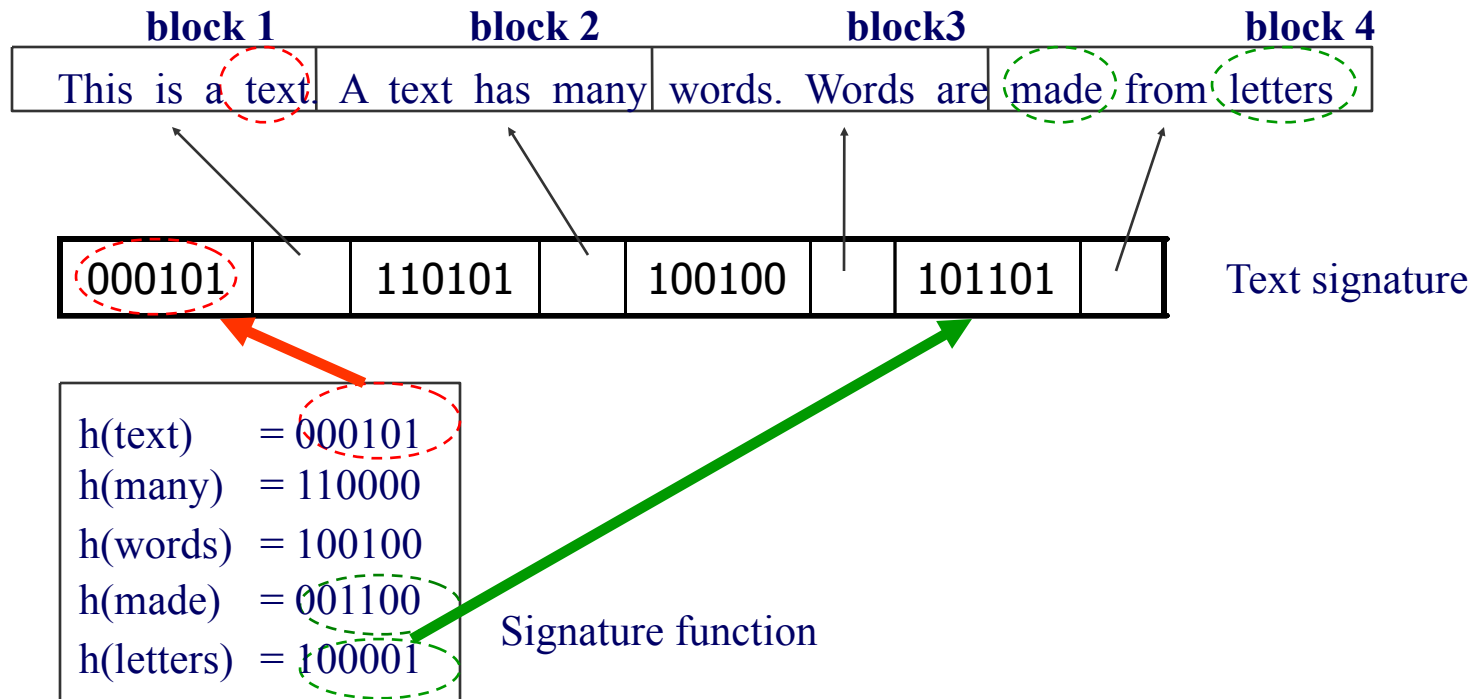
# Signature files

- Example:



# Signature files

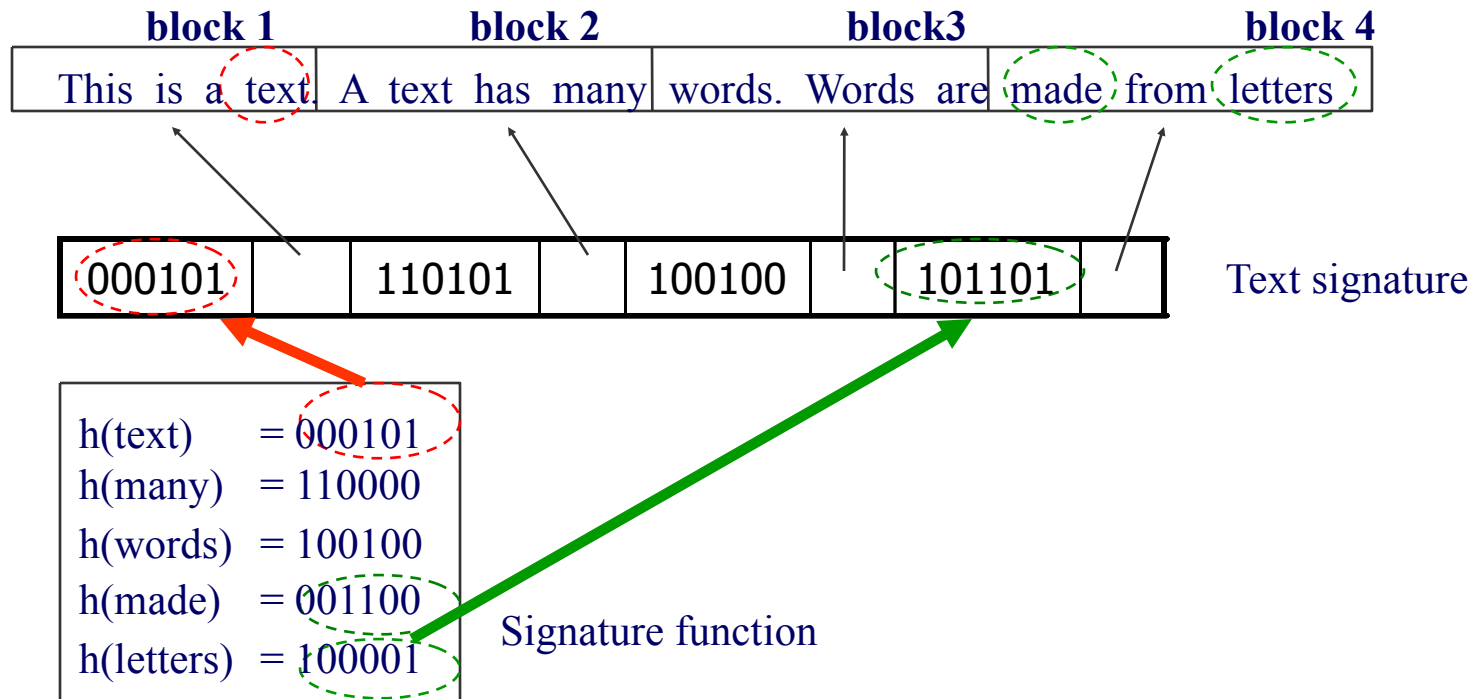
- Example:





# Signature files

- Example:



# Signature files

---

- False drop Problem
  - The corresponding **bits** are **set** even though the word is **not there!**
  - The **design** should **insure** that the **probability** of false drop is **low**.
    - Also the Signature file should be as short as possible.
  - **Enhance** the **hashing function** to minimize the error probability.

# Signature files

---

- Searching
  1. For a single word, Hash word to a bit mask  $W$ .
  2. For phrases,
    - 1) Hash words in query to a bit mask.
    - 2) Bitwise OR of all the query masks to a bit mask  $W$ .
  3. Compare  $W$  to the bit masks  $B_i$  of all the text blocks.
    - If all the bits set in  $W$  are also in  $B_i$ , then text block may contain the word.
  4. For all candidate text blocks, an online traversal must be performed to verify if the actual matches are there.
- Construction
  1. Cut the text in blocks.
  2. Generate an entry of the signature file for each block.
    - This entry is the bitwise OR of the signatures of all the words in the block.

---

# Suffix trees and suffix arrays

# String/Pattern Matching

- You are given a source string  $S$ .
- Answer queries of the form: is the string  $p_i$  a substring of  $S$ ?
- Knuth-Morris-Pratt (KMP) string matching.
  - $O(|S| + |p_i|)$  time per query.
  - $O(n|S| + \sum_i |p_i|)$  time for  $n$  queries.
- Suffix tree solution.
  - $O(|S| + \sum_i |p_i|)$  time for  $n$  queries.

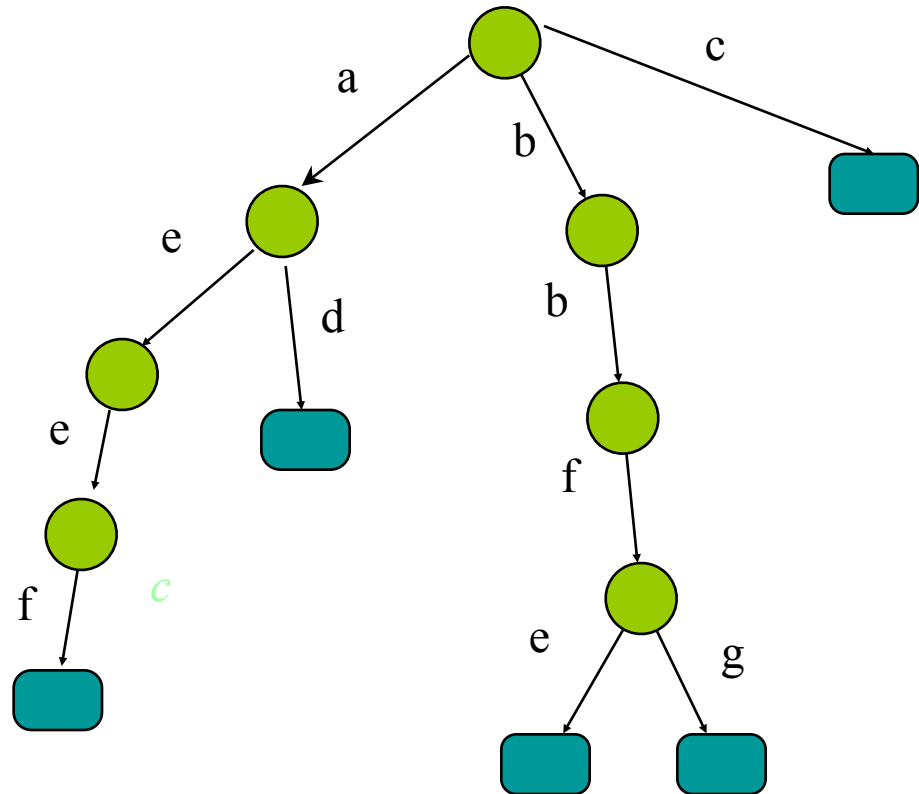
# String/Pattern Matching

- KMP/BM preprocesses the query string  $p_i$ , whereas the suffix tree method preprocesses the source string  $S$ .

# Trie

- A tree representing a set of strings.

{  
aeef  
ad  
bbfe  
bbfg  
c  
}

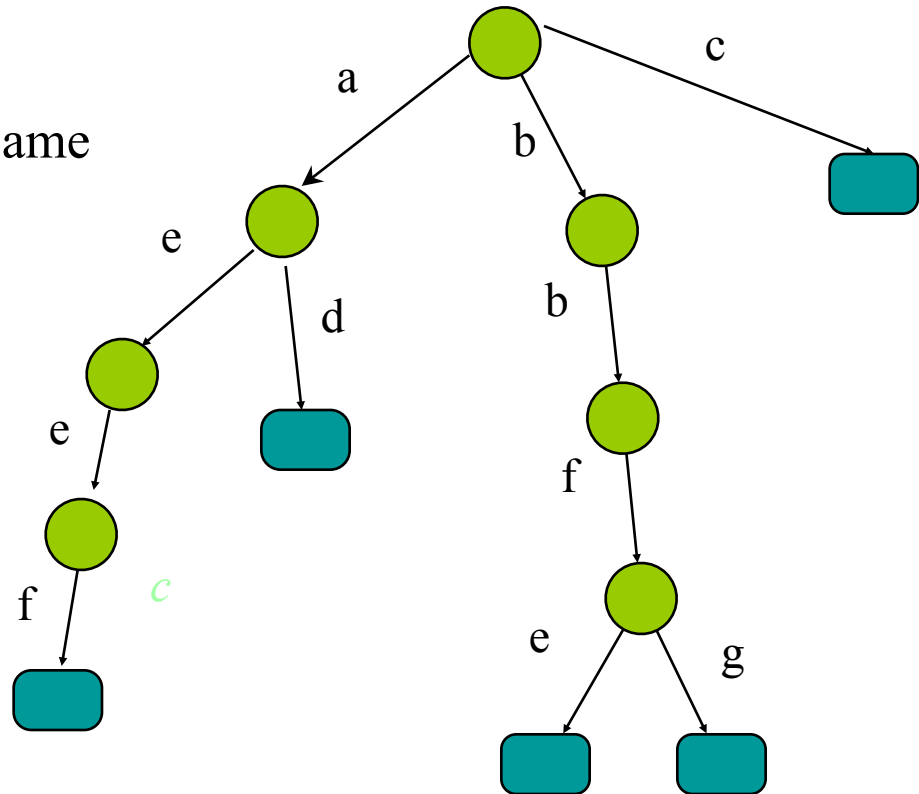


# Trie (Cont)

- Assume no string is a prefix of another

Each edge is labeled by a letter,  
no two edges outgoing from the same  
node are labeled the same.

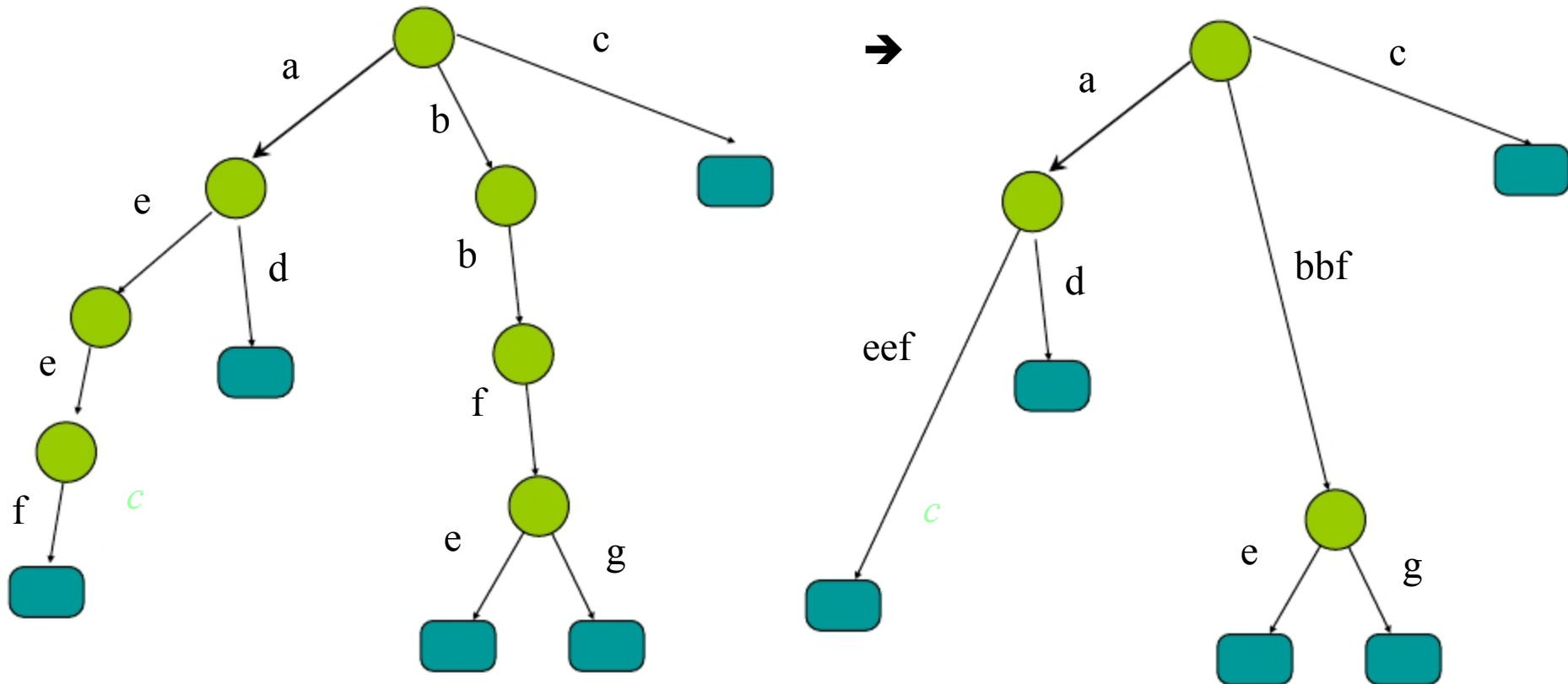
Each string corresponds to a leaf.





# Compressed Trie

- Compress unary nodes, label edges by strings



# Suffix tree

---

Given a string **s** a suffix tree of **s** is a compressed trie of all suffixes of s

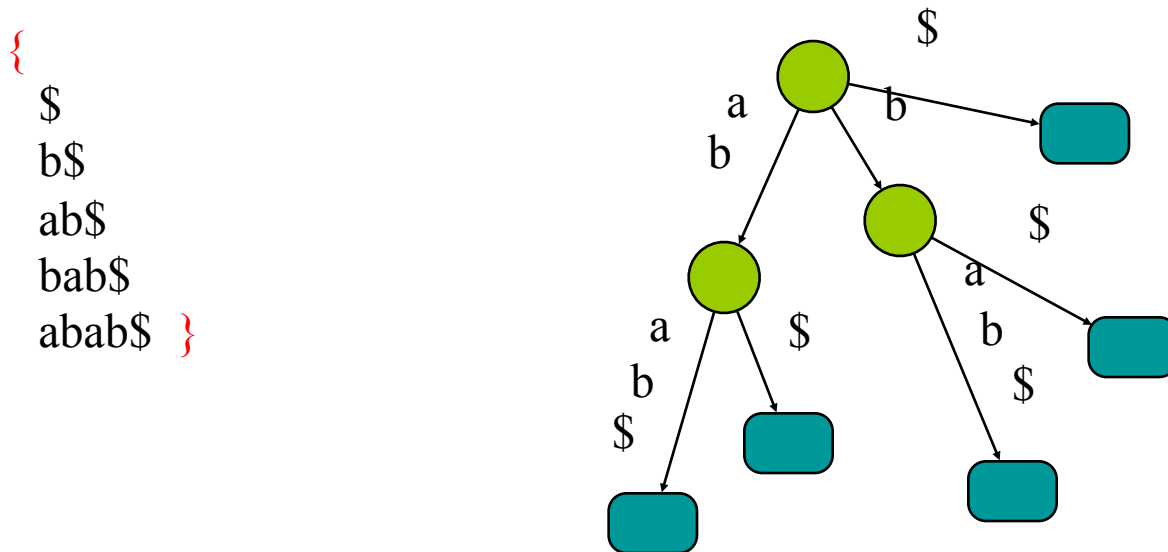
To make these suffixes prefix-free we add a special character, say **\$**, at the end of **s**

# The suffix tree $\text{Tree}(T)$ of $T$

- data structure **suffix tree**,  $\text{Tree}(T)$ , is compacted trie that represents all the suffixes of string  $T$
- linear size:  $|\text{Tree}(T)| = O(|T|)$
- can be constructed in linear time  $O(|T|)$
- has *myriad virtues* (A. Apostolico)
- is well-known: Google hits

# Suffix tree (Example)

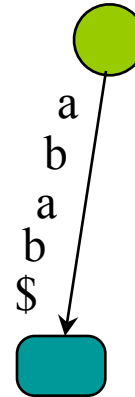
Let  $s=abab$ , a suffix tree of  $s$  is a compressed trie of all suffixes of  $s=abab\$$



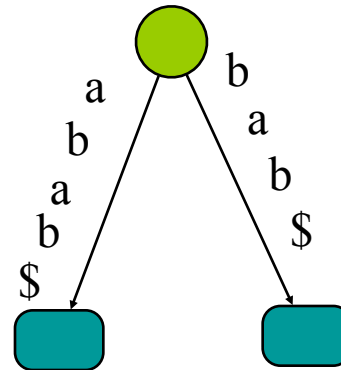
# Trivial algorithm to build a Suffix tree

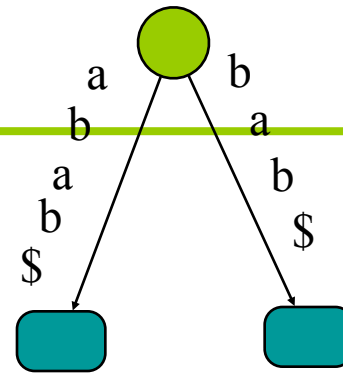
---

Put the largest suffix in

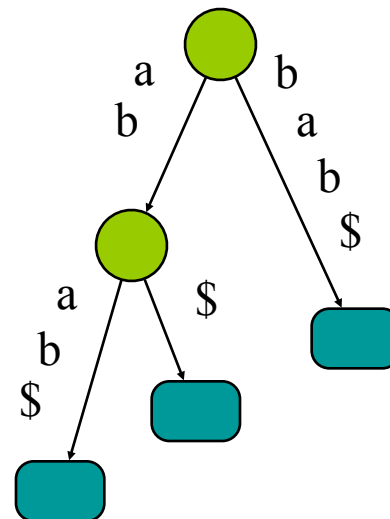


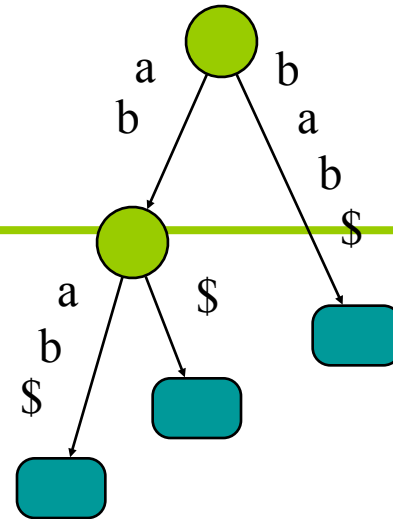
Put the suffix **bab**\$ in



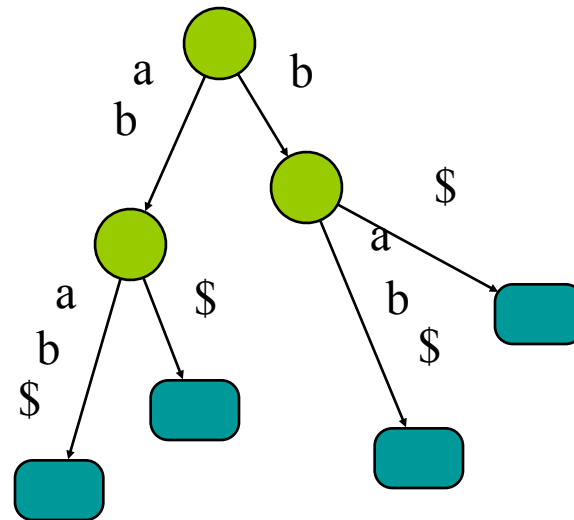


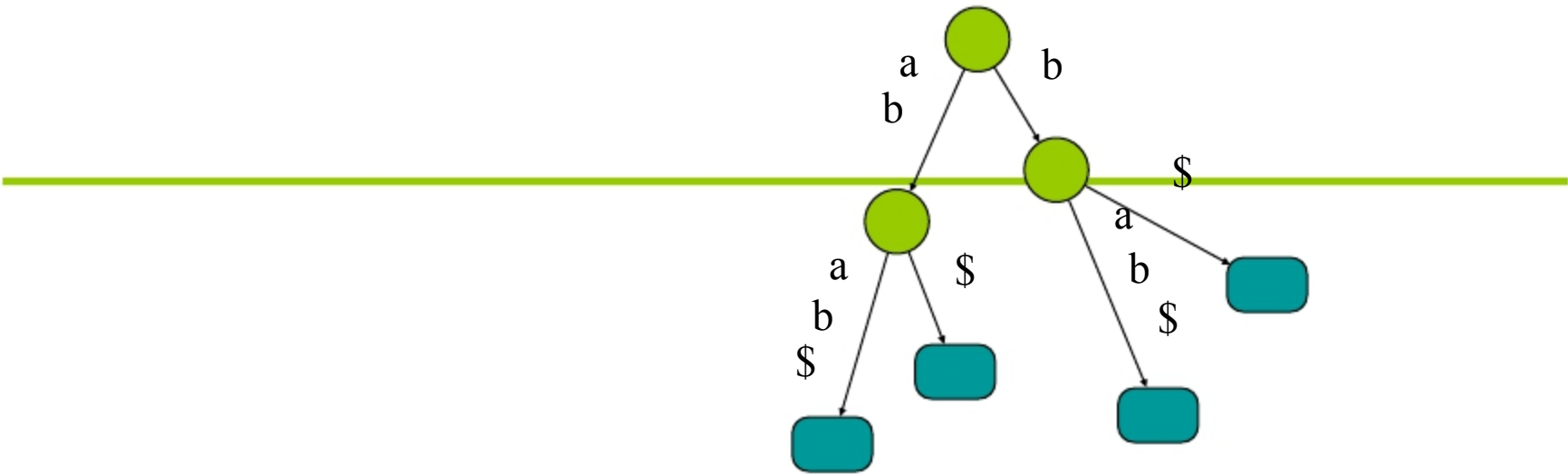
Put the suffix **ab**\$ in



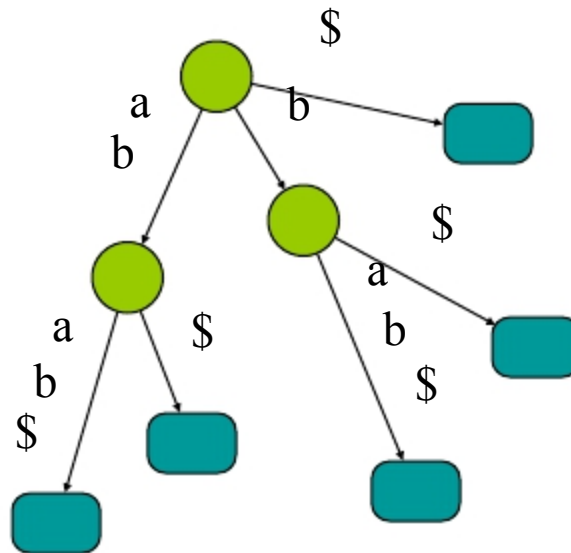


Put the suffix **b**\$ in

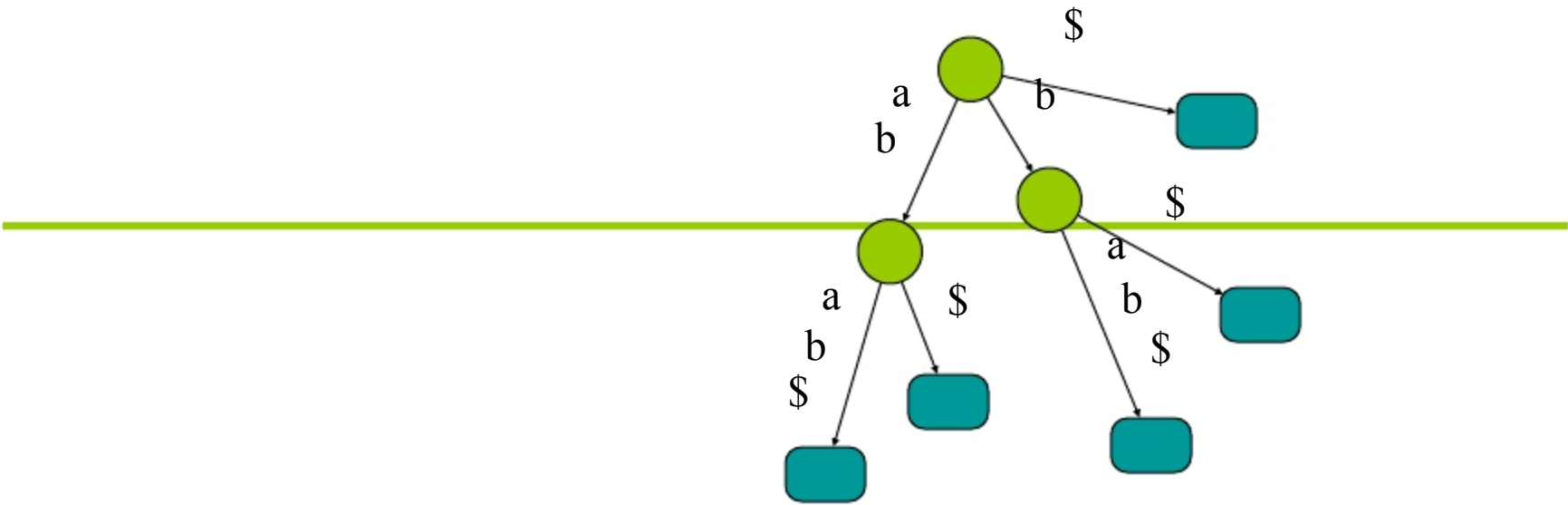




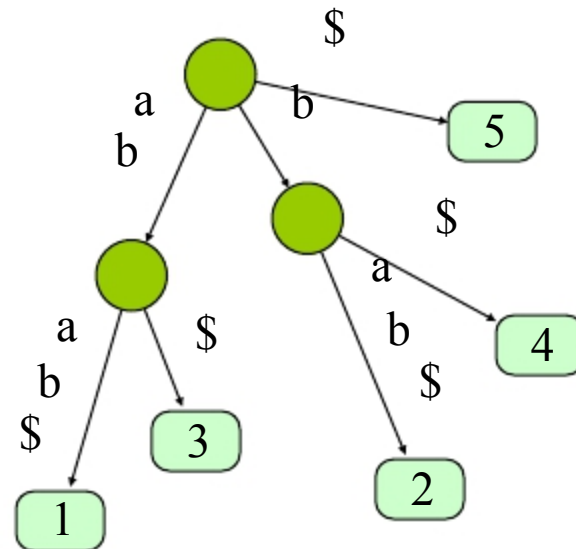
Put the suffix \$ in







We will also label each leaf with the starting point of the corres. suffix.

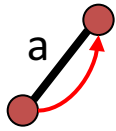


# On-line construction of Trie(T)

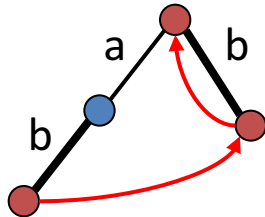
- $T = t_1 t_2 \dots t_n \$$
- $P_i = t_1 t_2 \dots t_i$     *i:th prefix* of  $T$
- on-line idea: update  $Trie(P_i)$  to  $Trie(P_{i+1})$
- $\Rightarrow$  very simple construction

# *Trie(abaab)*

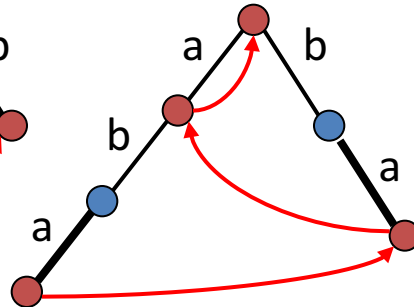
*Trie(a)*



*Trie(ab)*



*Trie(aba)*



aba**a**

ba**a**

a**a**

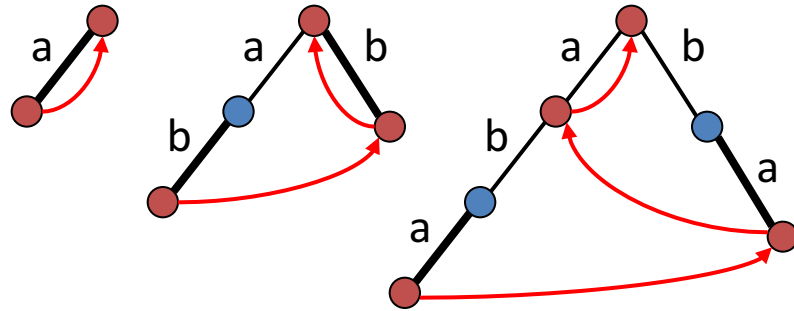
$\epsilon$ **a**

$\epsilon$

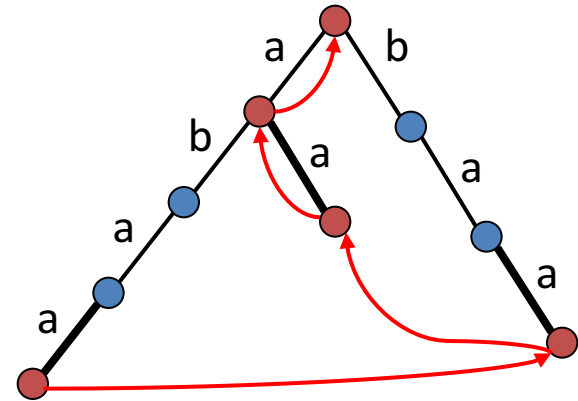
chain of links  
suffixes

 connects the end points of current

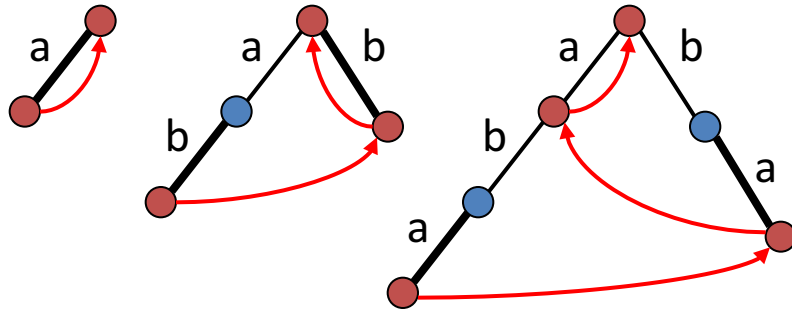
*Trie(abaab)*



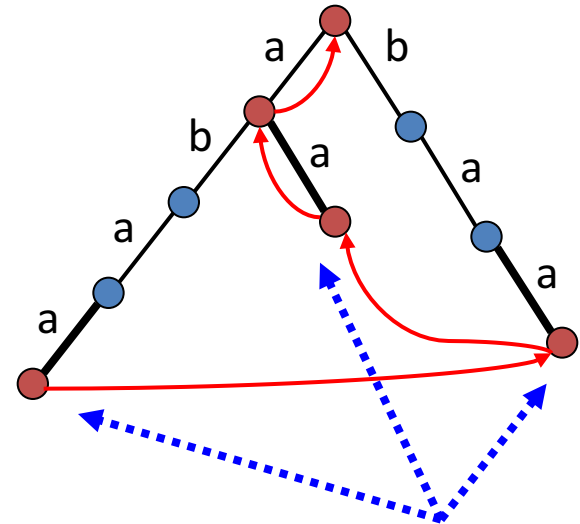
*Trie(abaa)*



# *Trie(abaab)*

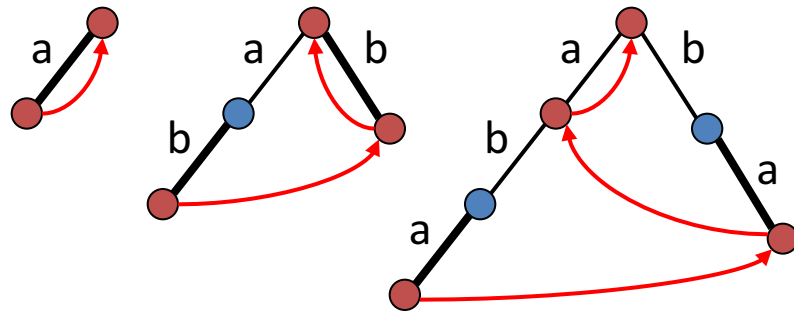


*Trie(abaa)*

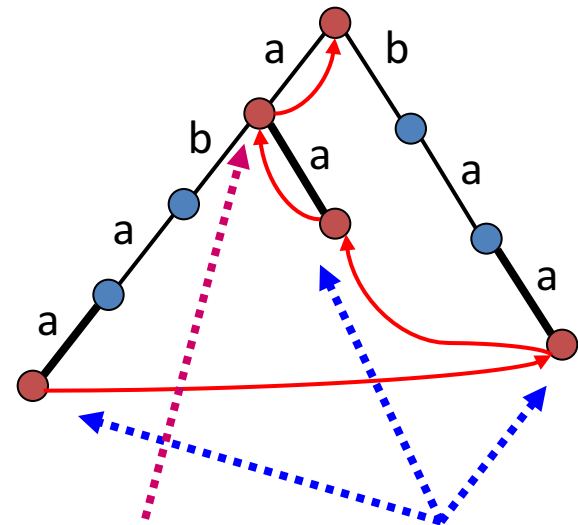


Add next symbol = b

# *Trie(abaab)*



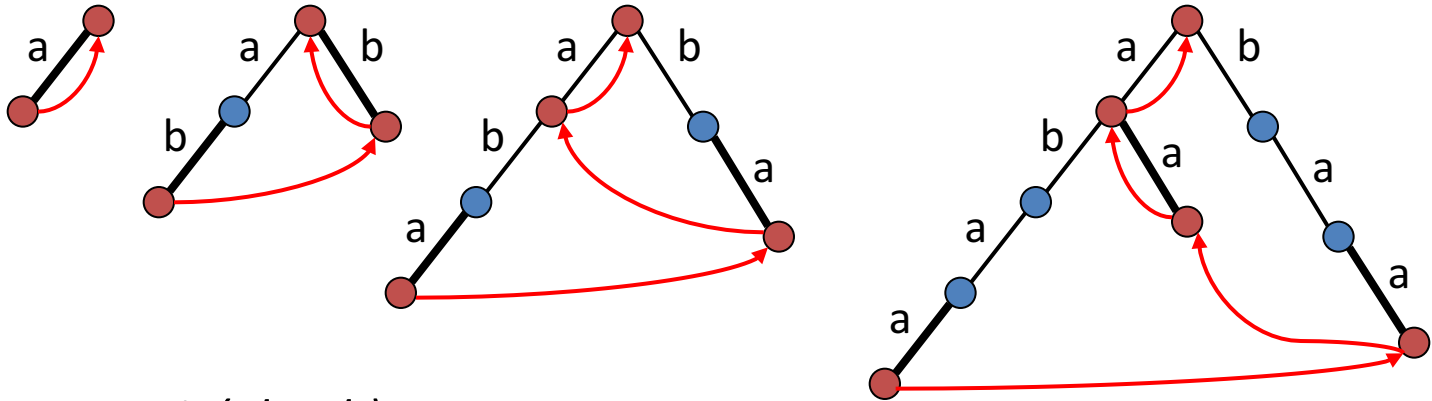
*Trie(abaa)*



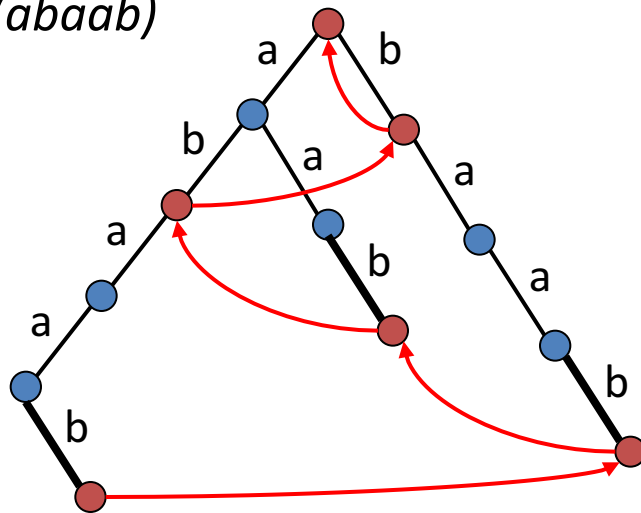
Add next symbol = b

From here on b-arc already exists

# *Trie(abaab)*

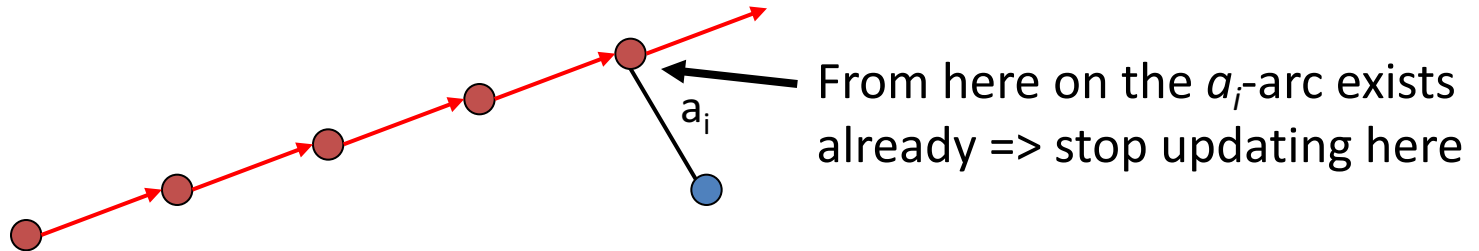


*Trie(abaab)*

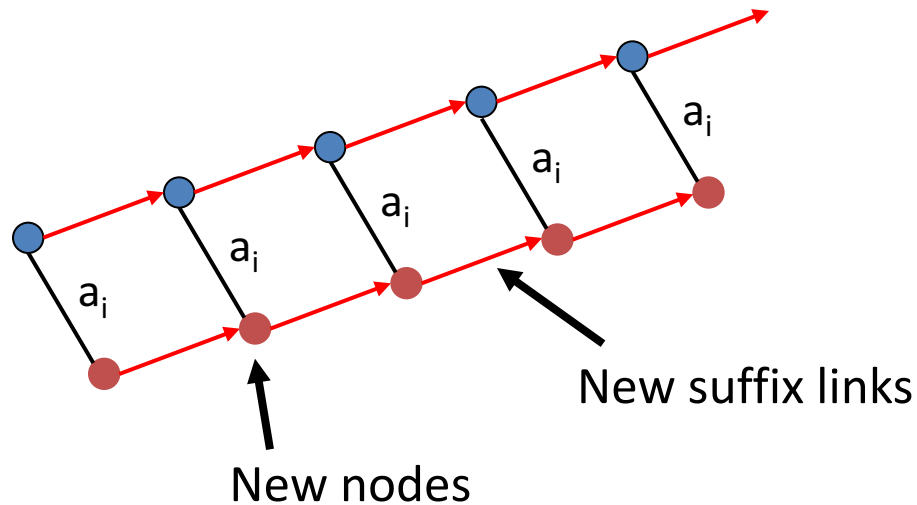


# What happens in $\text{Trie}(P_i) \Rightarrow \text{Trie}(P_{i+1})$ ?

Before



After





What happens in  $Trie(P_i) \Rightarrow Trie(P_{i+1})$  ?

- time:  $O(\text{size of Trie}(T))$
- suffix links:  
     $\text{slink}(\text{node}(a\alpha)) = \text{node}(\alpha)$

# What can we do with it ?

---

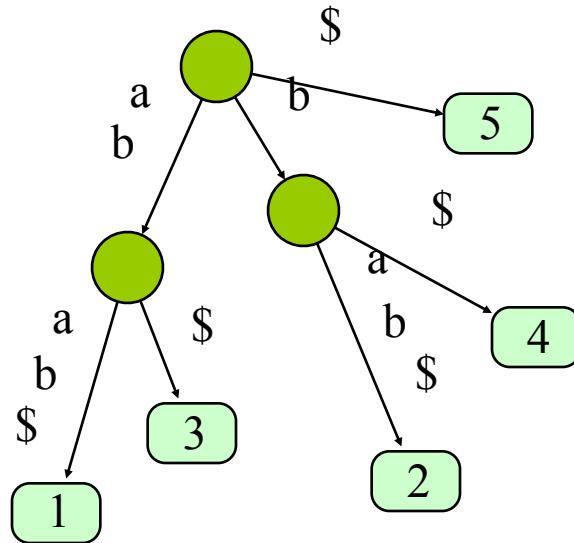
## Exact string matching:

Given a Text  $T$ ,  $|T| = n$ , preprocess it such that when a pattern  $P$ ,  $|P| = m$ , arrives you can quickly decide when it occurs in  $T$ .

We may also want to find all occurrences of  $P$  in  $T$

# Exact string matching

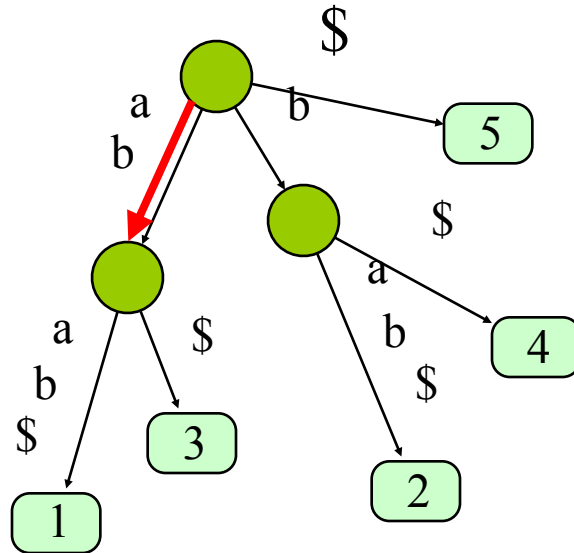
In preprocessing we just build a suffix tree in  $O(n)$  time



Given a pattern  $P = \text{ab}$  we traverse the tree according to the pattern.

# Exact string matching

In preprocessing we just build a suffix tree in  $O(n)$  time



Given a pattern  $P = \text{ab}$  we traverse the tree according to the pattern.



If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all  $k$  occurrences in  $O(n+k)$  time

# Generalized suffix tree

---

Given a set of strings  $S$  a generalized suffix tree of  $S$  is a compressed trie of all suffixes of  $s \in S$

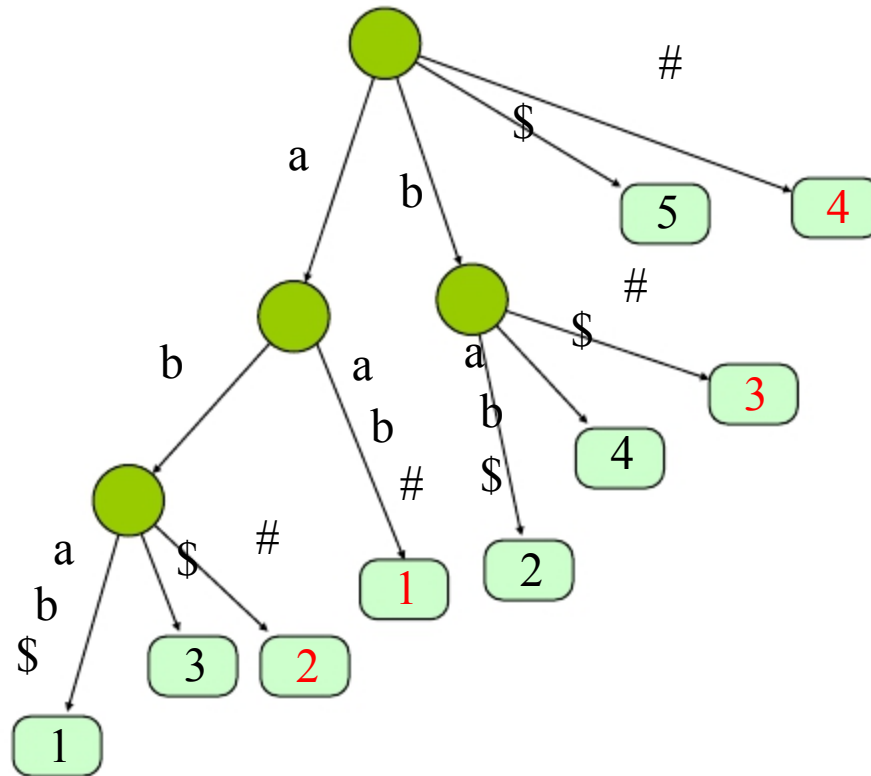
To make these suffixes prefix-free we add a special char, say  $\$,$  at the end of  $s$

To associate each suffix with a unique string in  $S$  add a different special char to each  $s$

# Generalized suffix tree (Example)

Let  $s_1 = abab$  and  $s_2 = aab$  here is a generalized suffix tree for  $s_1$  and  $s_2$

{  
\$ #  
b\$ b#  
ab\$ ab#  
bab\$ aab#  
abab\$  
}



# So what can we do with it ?

---

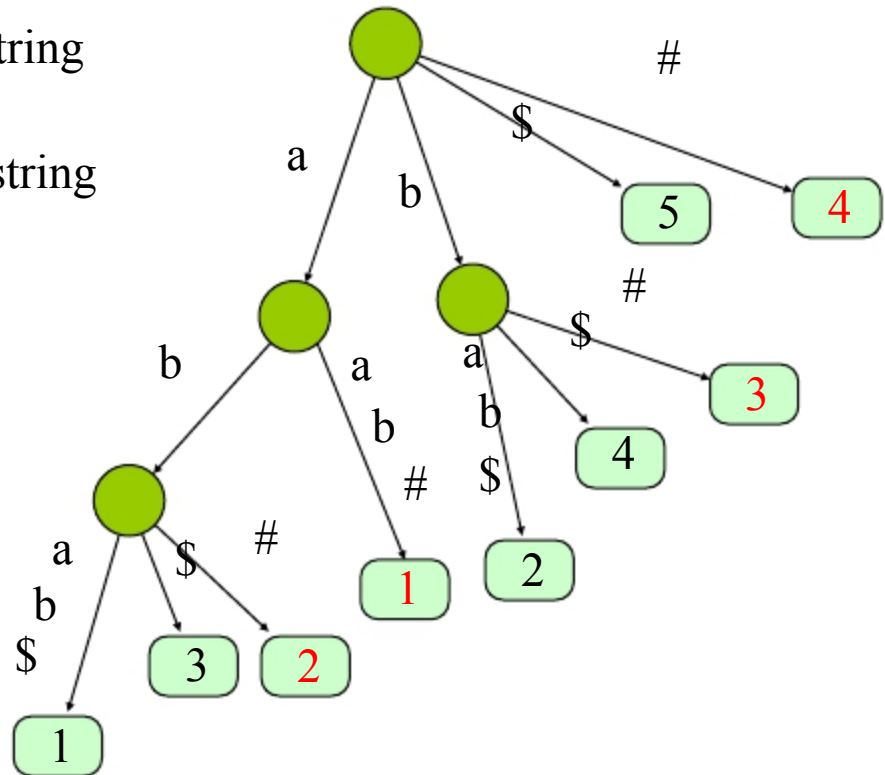
Matching a pattern against a database of strings



# Longest common substring (of two strings)

Every node with a leaf descendant from string **S1** and a leaf descendant from string **S2** represents a maximal common substring and vice versa.

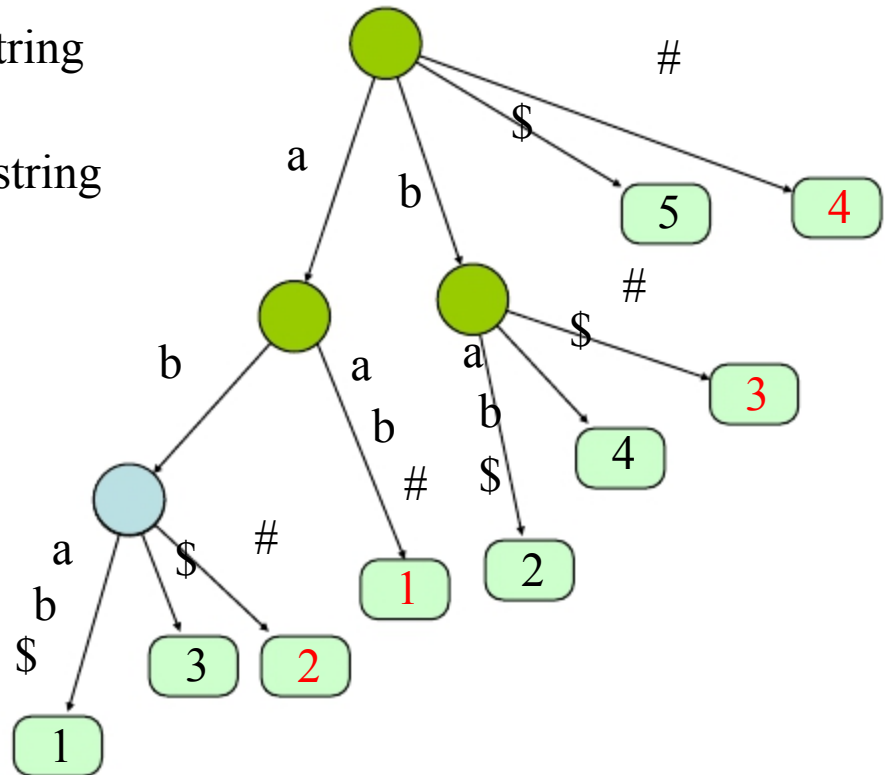
Find such node with largest “string depth”



# Longest common substring (of two strings)

Every node with a leaf descendant from string **S1** and a leaf descendant from string **S2** represents a maximal common substring and vice versa.

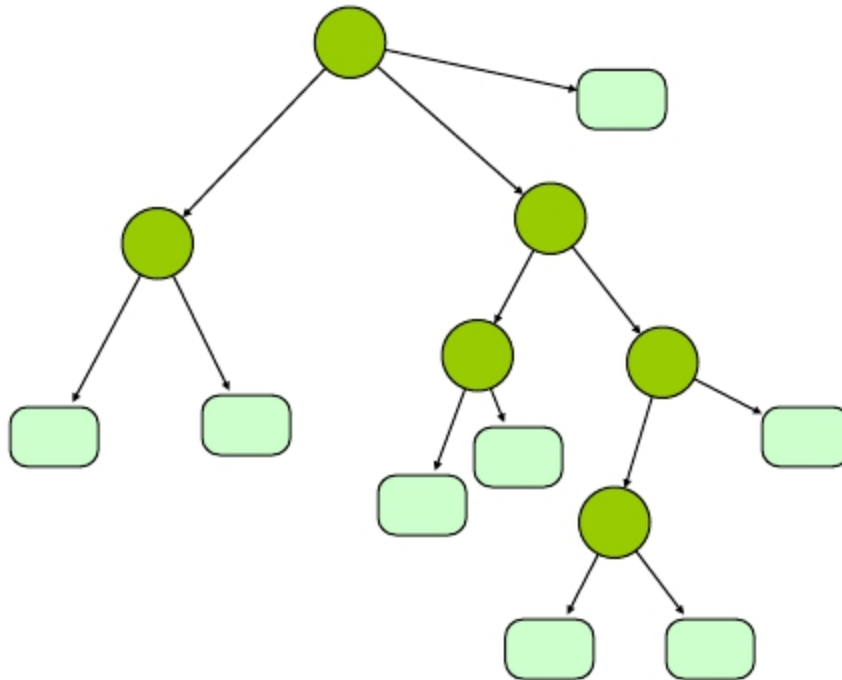
Find such node with largest “string depth”



# Lowest common ancestor

---

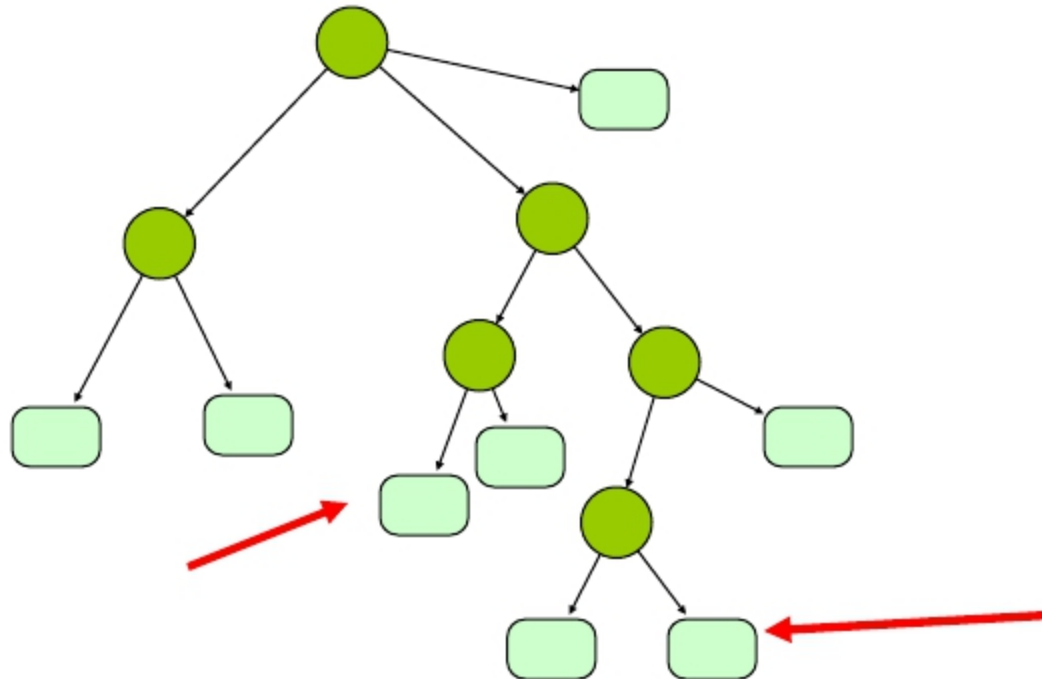
A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it



# Lowest common ancestor

---

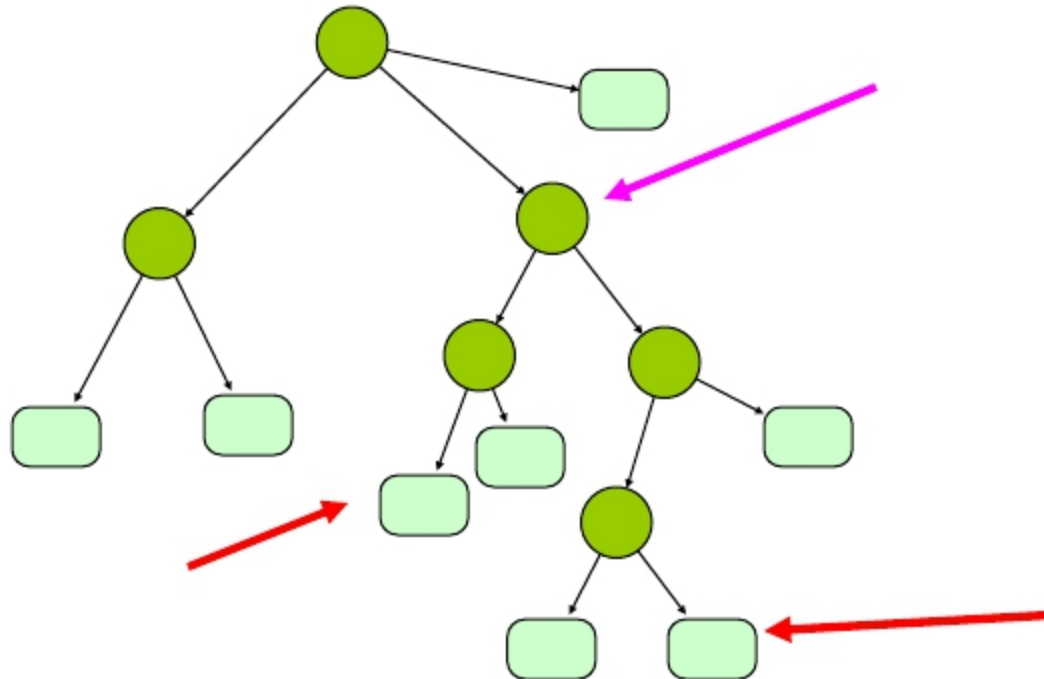
A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it



# Lowest common ancestor

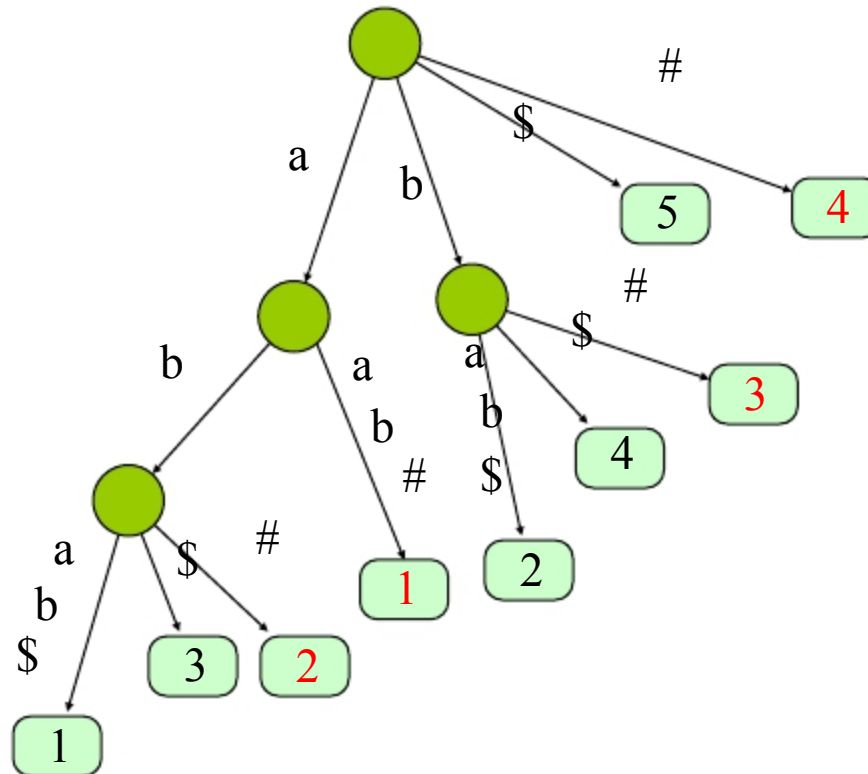
---

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it



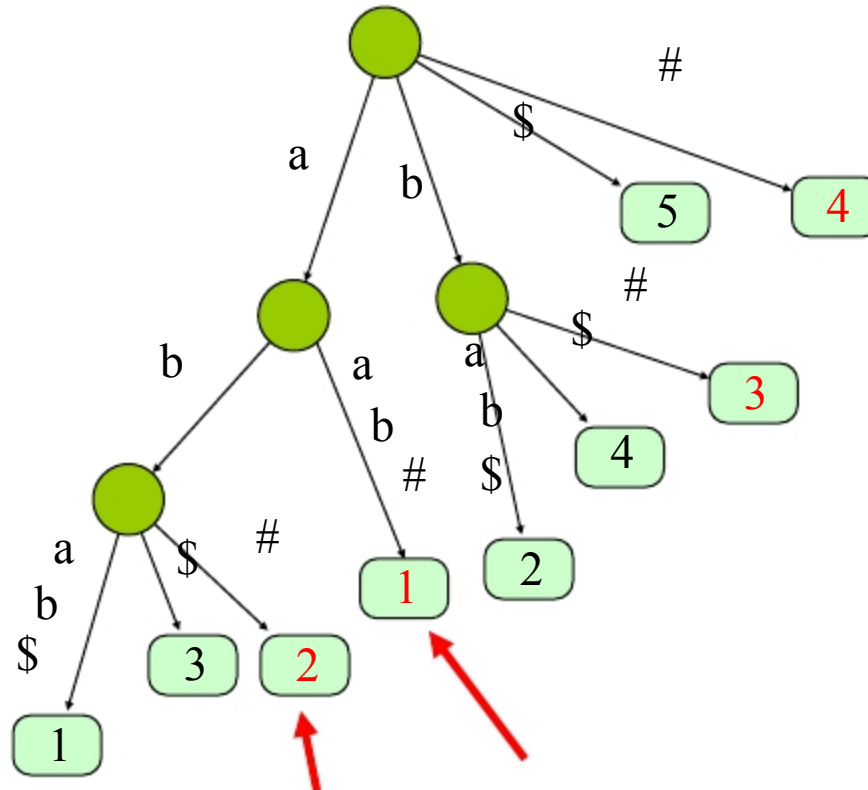
# Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes



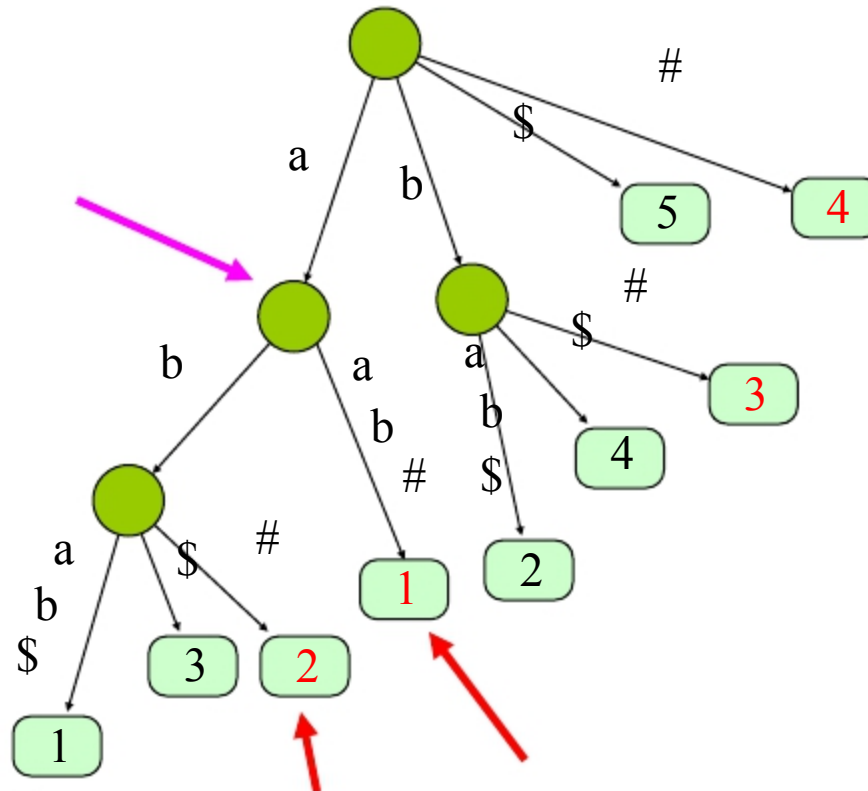
# Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes



# Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes





# Finding maximal palindromes

---

- A palindrome: caabaac, cbaabc
- Want to find all maximal palindromes in a string **s**

Let **s** = cbaaba

The maximal palindrome with center between  $i-1$  and  $i$  is the LCP of the suffix at position  $i$  of **S** and the suffix at position  $m-i+1$  of **Sr**

# Maximal palindromes algorithm

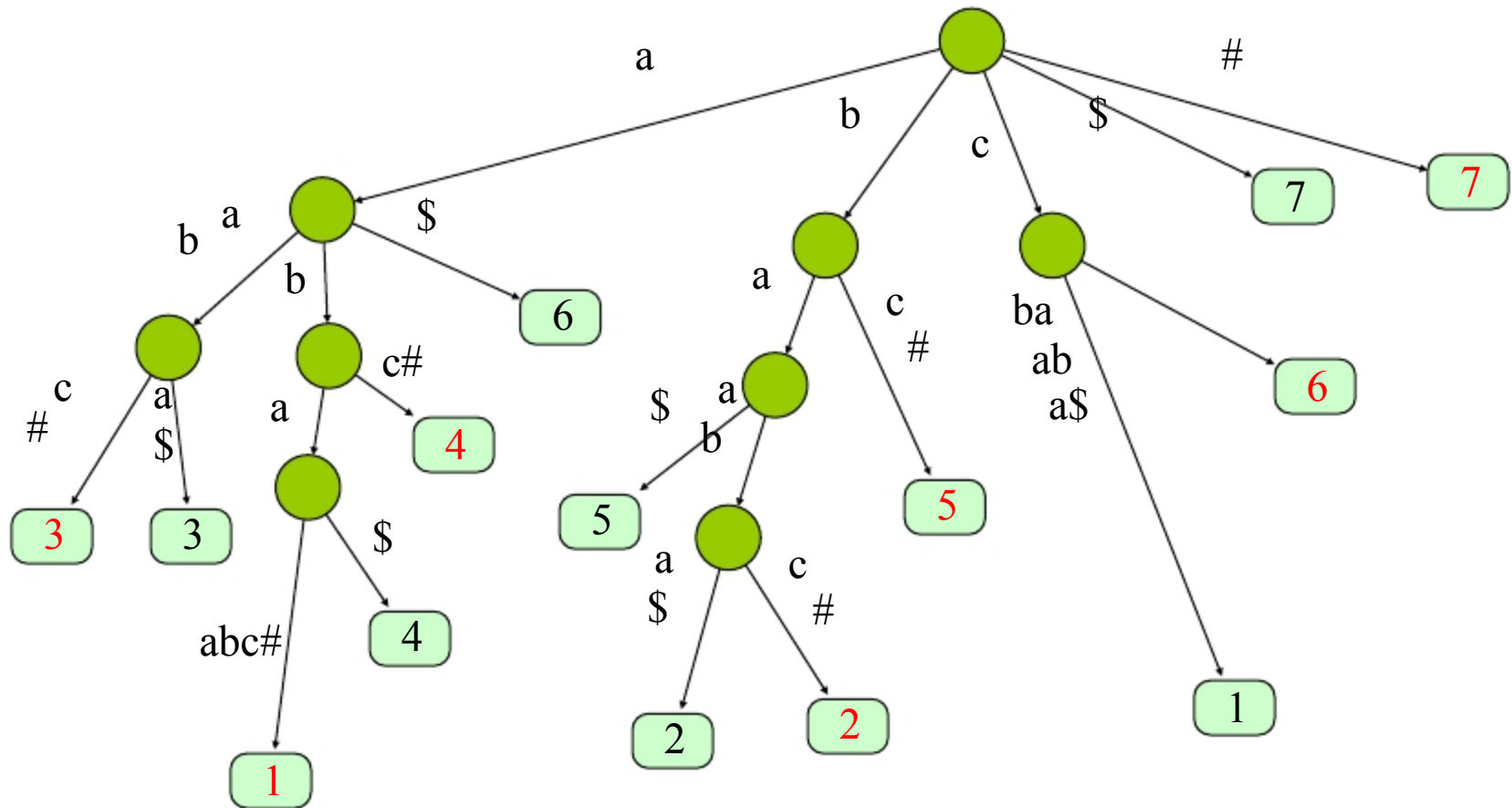
---

Prepare a generalized suffix tree for

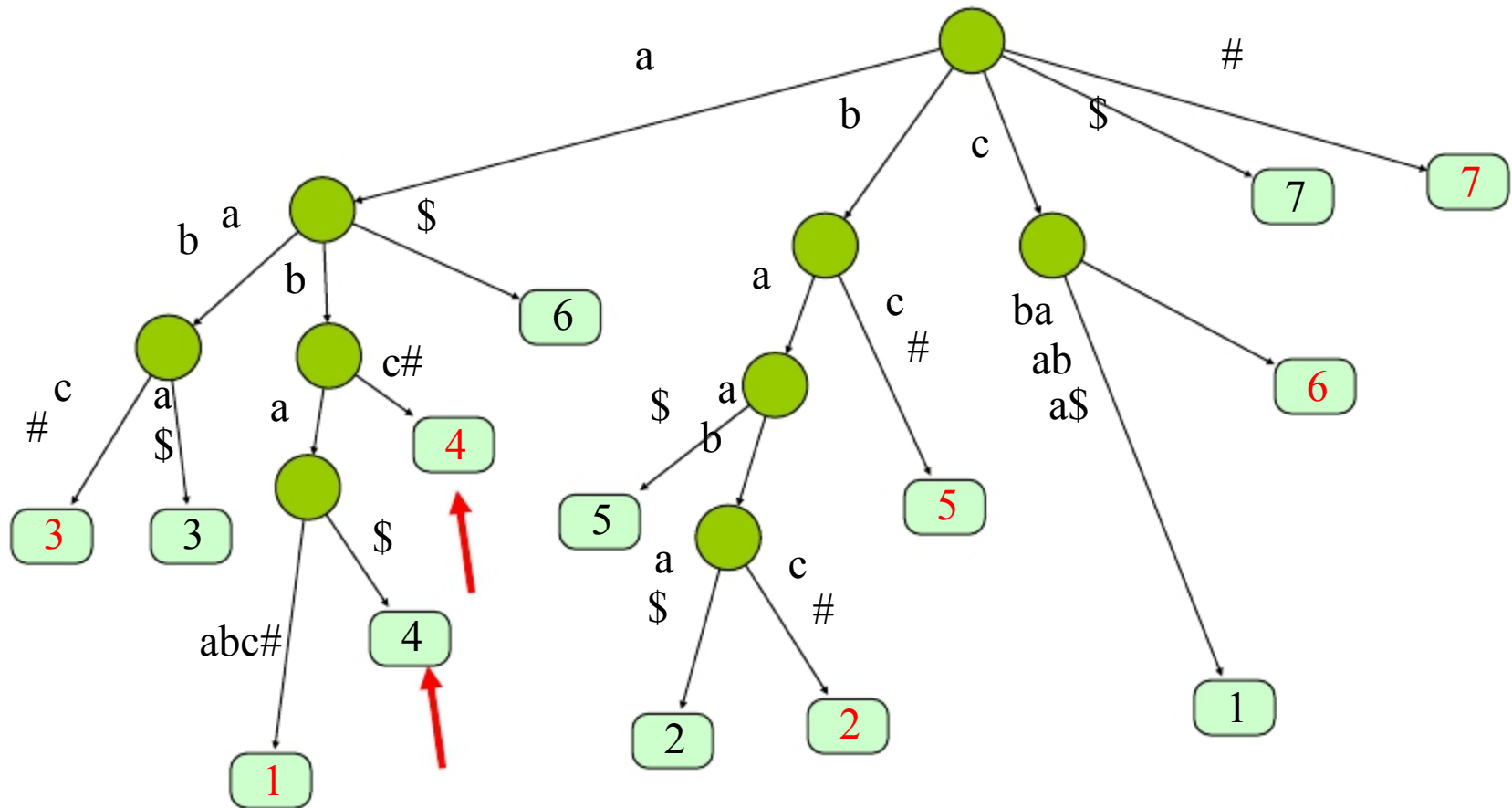
$s = cbaaba\$$  and  $s_r = abaabc\#$

For every  $i$  find the LCA of suffix  $i$  of  $s$  and  
suffix  $m-i+1$  of  $s_r$

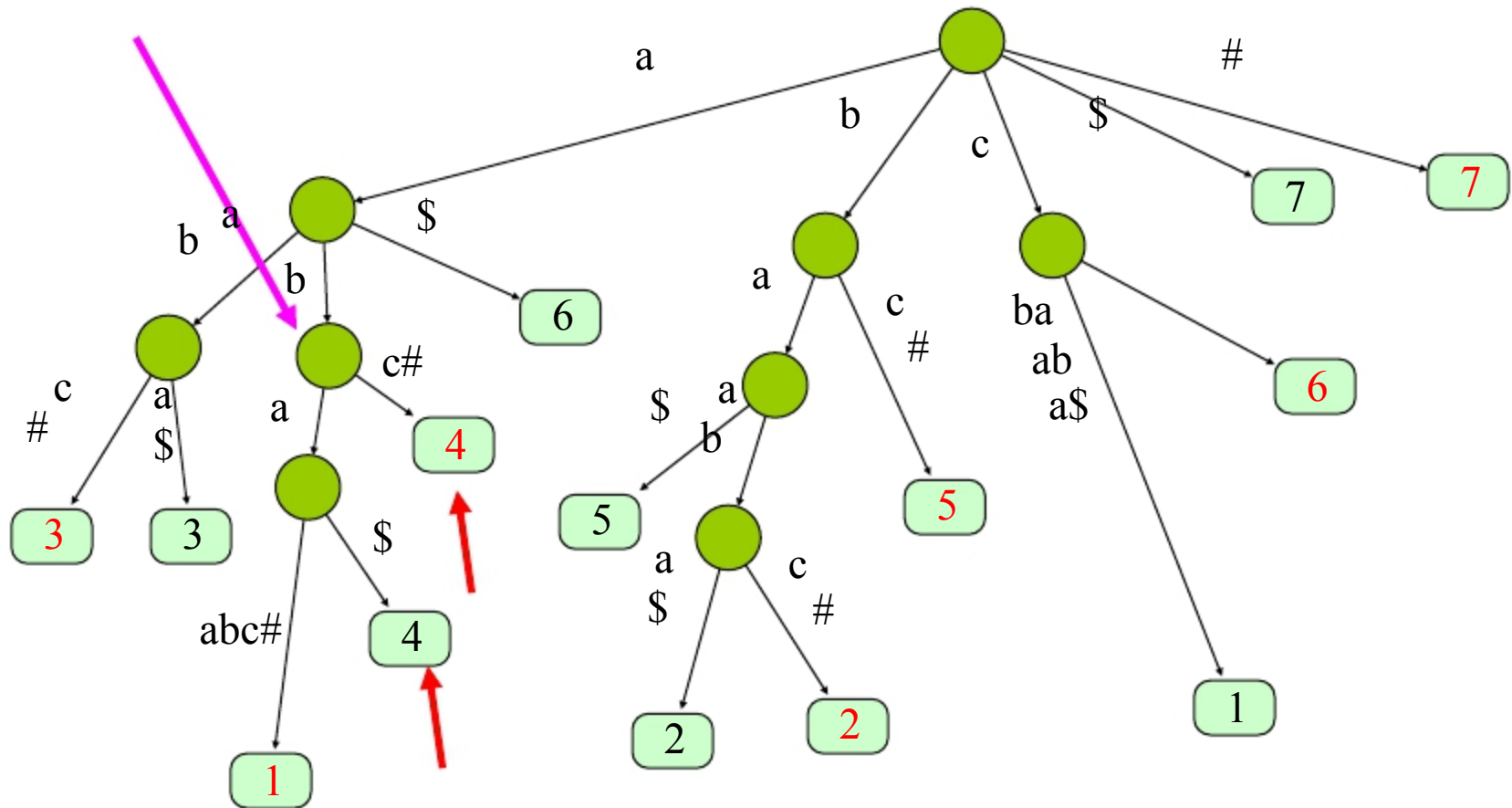
Let  $s = cbaaba\$$  then  $s_r = abaabc\#$



Let  $s = cbaaba\$$  then  $s_r = abaabc\#$



Let  $s = cbaaba\$$  then  $s_r = abaabc\#$



# Analysis

---

$O(n)$  time to identify all palindromes

# Drawbacks

---

- Suffix trees consume a lot of space
- It is  $O(n)$  but the constant is quite big
- Notice that if we indeed want to traverse an edge in  $O(1)$  time then we need an array of ptrs. of size  $|\Sigma|$  in each node

# Suffix array

---

- We loose some of the functionality but we save space.

Let  $s = \text{abab}$

Sort the suffixes lexicographically:

$\text{ab}, \text{abab}, \text{b}, \text{bab}$

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2
---	---	---	---



# How do we build it ?

---

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- $O(n)$  time

# How do we search for a pattern ?

---

- If  $P$  occurs in  $T$  then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes  $O(m \log n)$  time

# Example

Let **S** = mississippi

---

Let **P** = issa

<b>L</b> →	11	i
	8	ippi
	5	issippi
	2	ississippi
	1	mississippi
<b>M</b> →	10	pi
	9	ppi
	7	sippi
	4	sisippi
	6	ssippi
<b>R</b> →	3	ssissippi

# Supra index

---

- Structure
  - **Suffix arrays** are **space efficient** implementation of **suffix trees**.
  - Simply an array **containing all the pointers** to the text suffixes listed in lexicographical order.
  - **Supra-indices**:
    - If the suffix array is **large**, this binary search can perform **poorly** because of the number of random disk accesses.
    - Suffix arrays are designed to allow **binary searches** done by comparing the contents of each pointer.
    - To remedy this situation, the use of **supra-indices** over the suffix array has been proposed.

# Supra index

- Example

1    6 9 11    17 19    24 28    33    40    46 50    55    60  
This is a text. A text has many words. Words are made from letters

<b>60</b>	<b>50</b>	<b>28</b>	<b>19</b>	<b>11</b>	<b>40</b>	<b>33</b>
-----------	-----------	-----------	-----------	-----------	-----------	-----------

SuffixArray

lett		text		word		
60	50	28	19	11	40	33

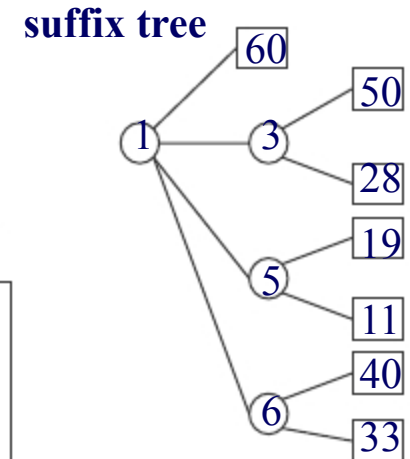
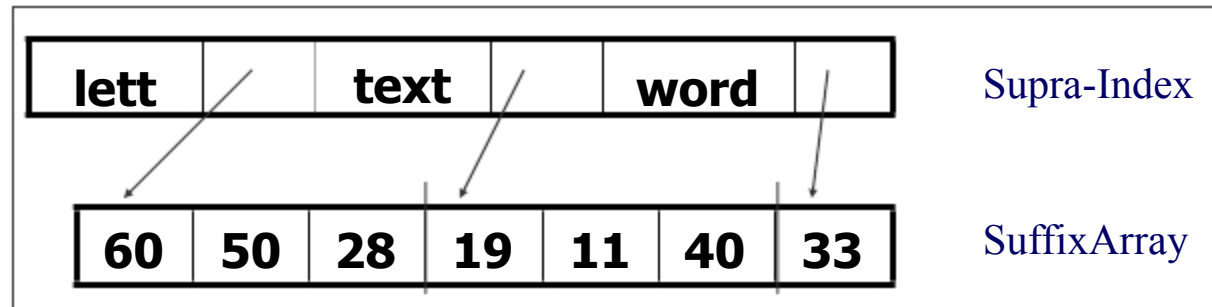
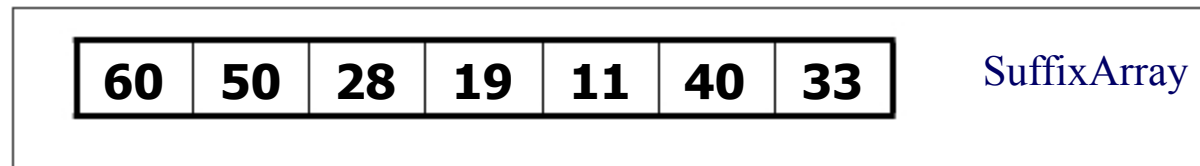
Supra-Index

SuffixArray

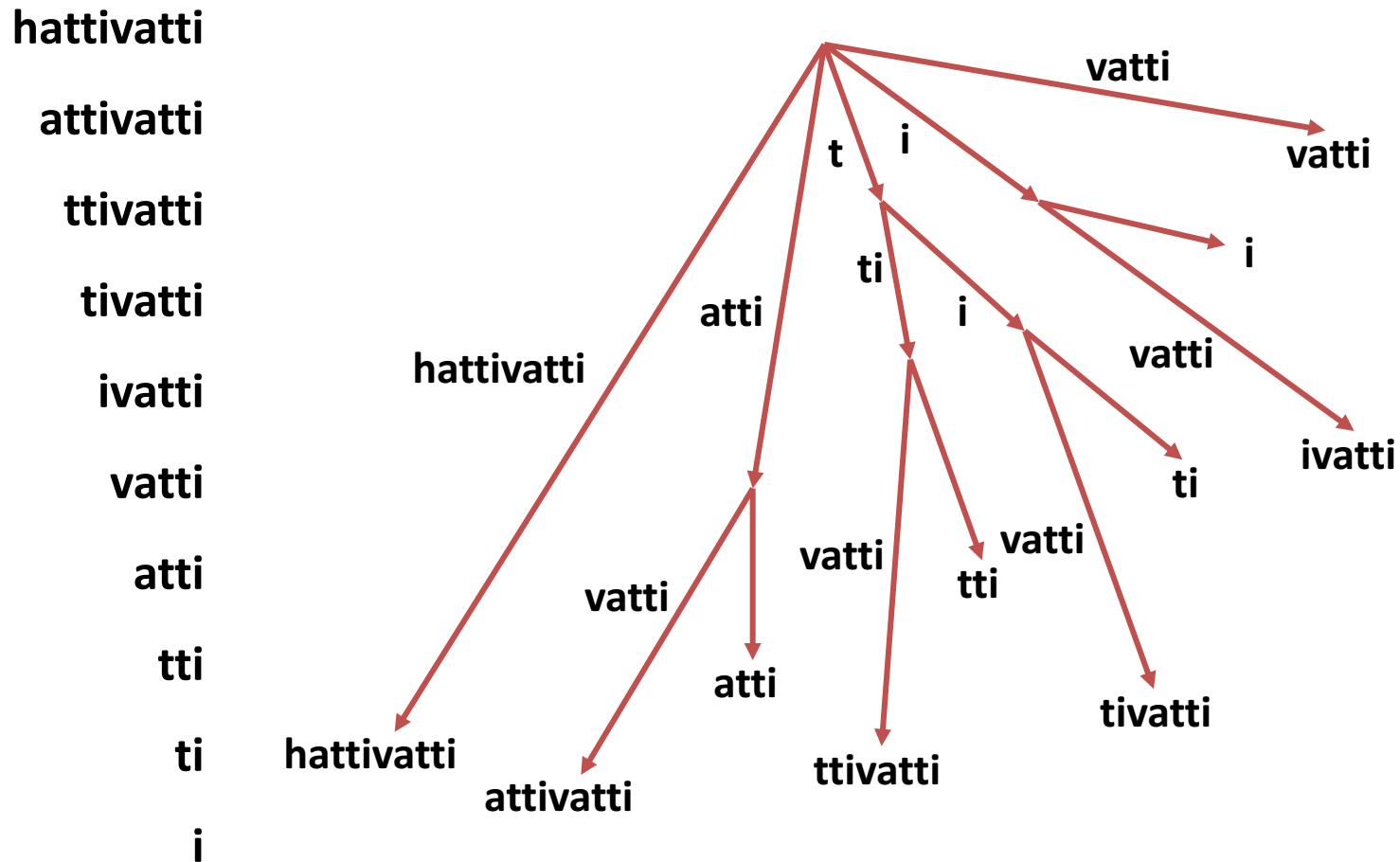
# Supra index

- Example

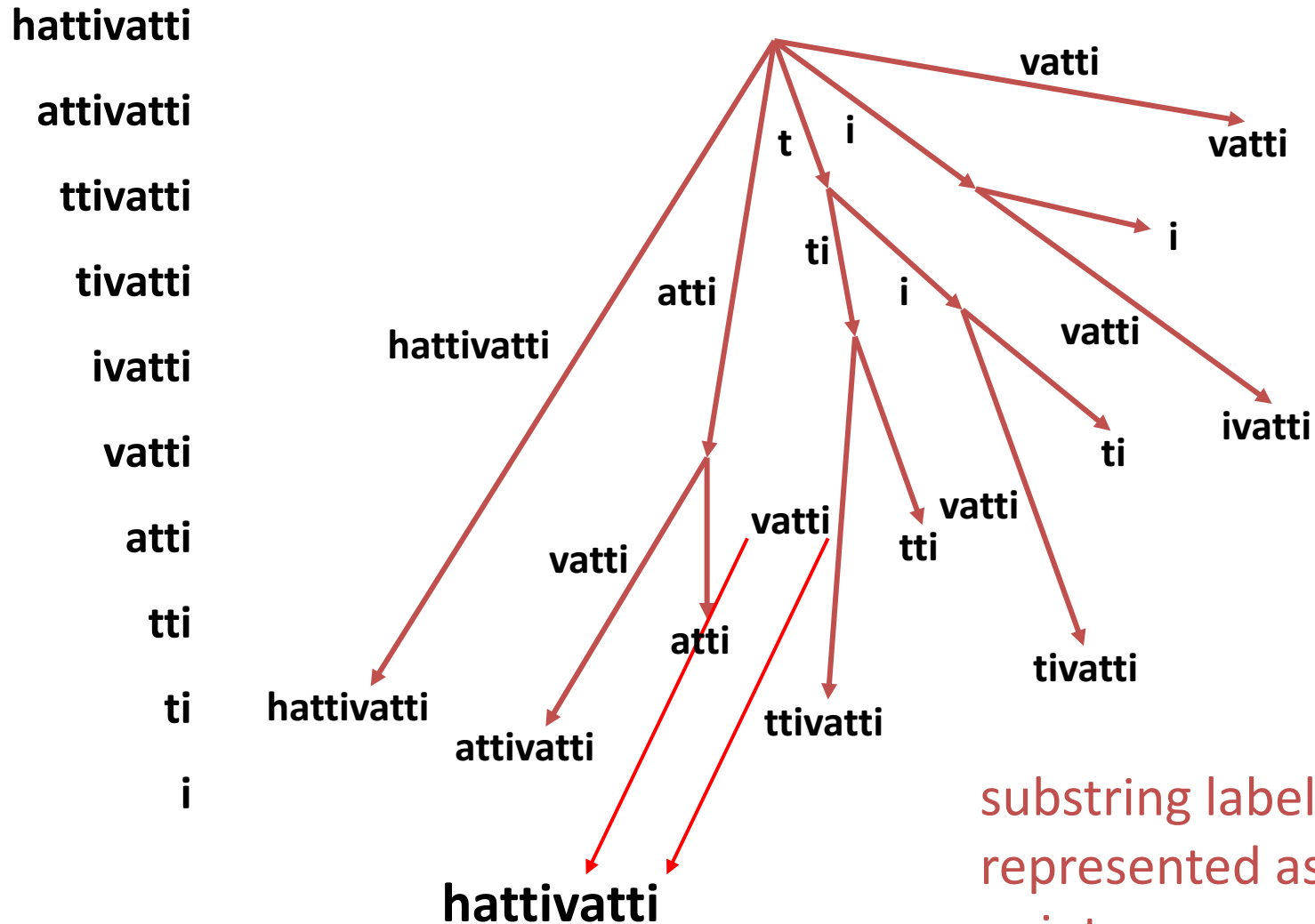
1    6 9 11   17 19   24 28    33    40    46 50   55   60  
This is a text. A text has many words. Words are made from letters



# Tree(hattivatti)



# Tree(hattivatti)



substring labels of edges  
represented as pairs of  
pointers



# Tree(hattivatti)

hattivatti

attivatti

ttivatti

tivatti

ivatti

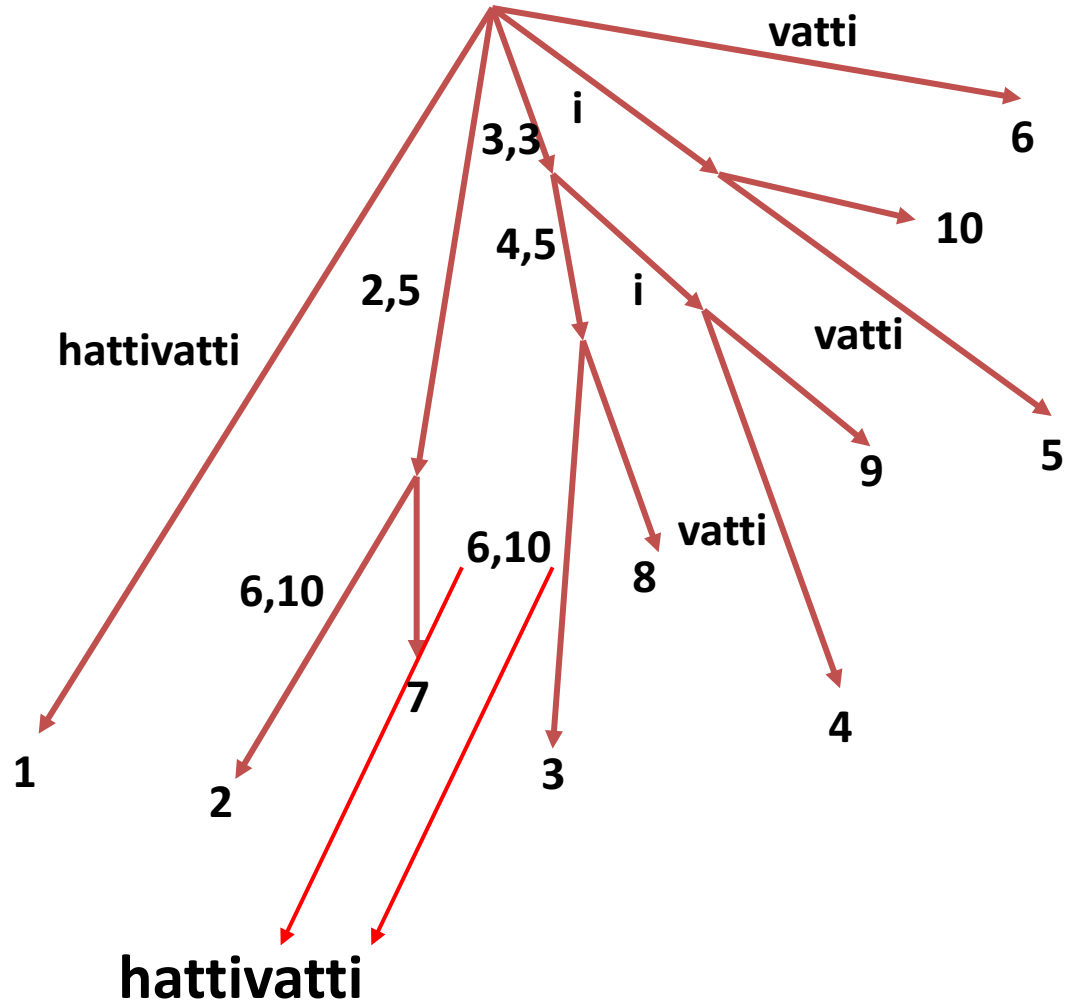
vatti

atti

tti

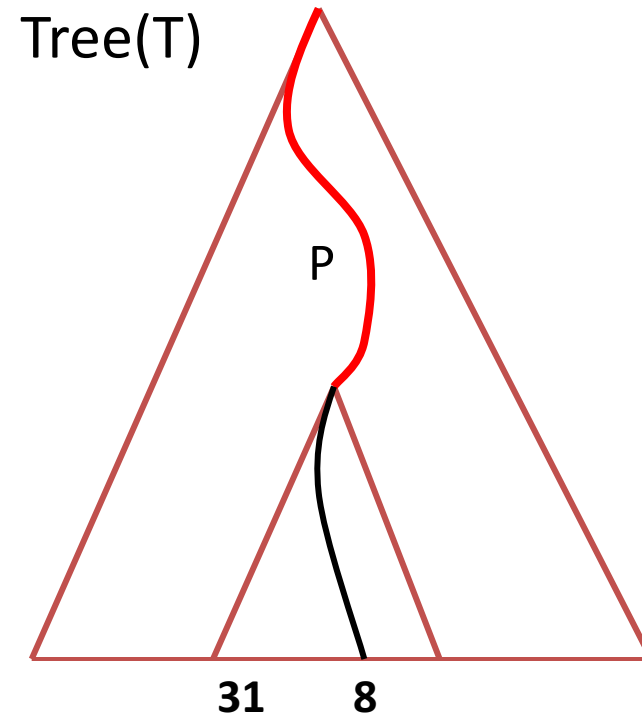
ti

i



# Tree(T) is *full* text index

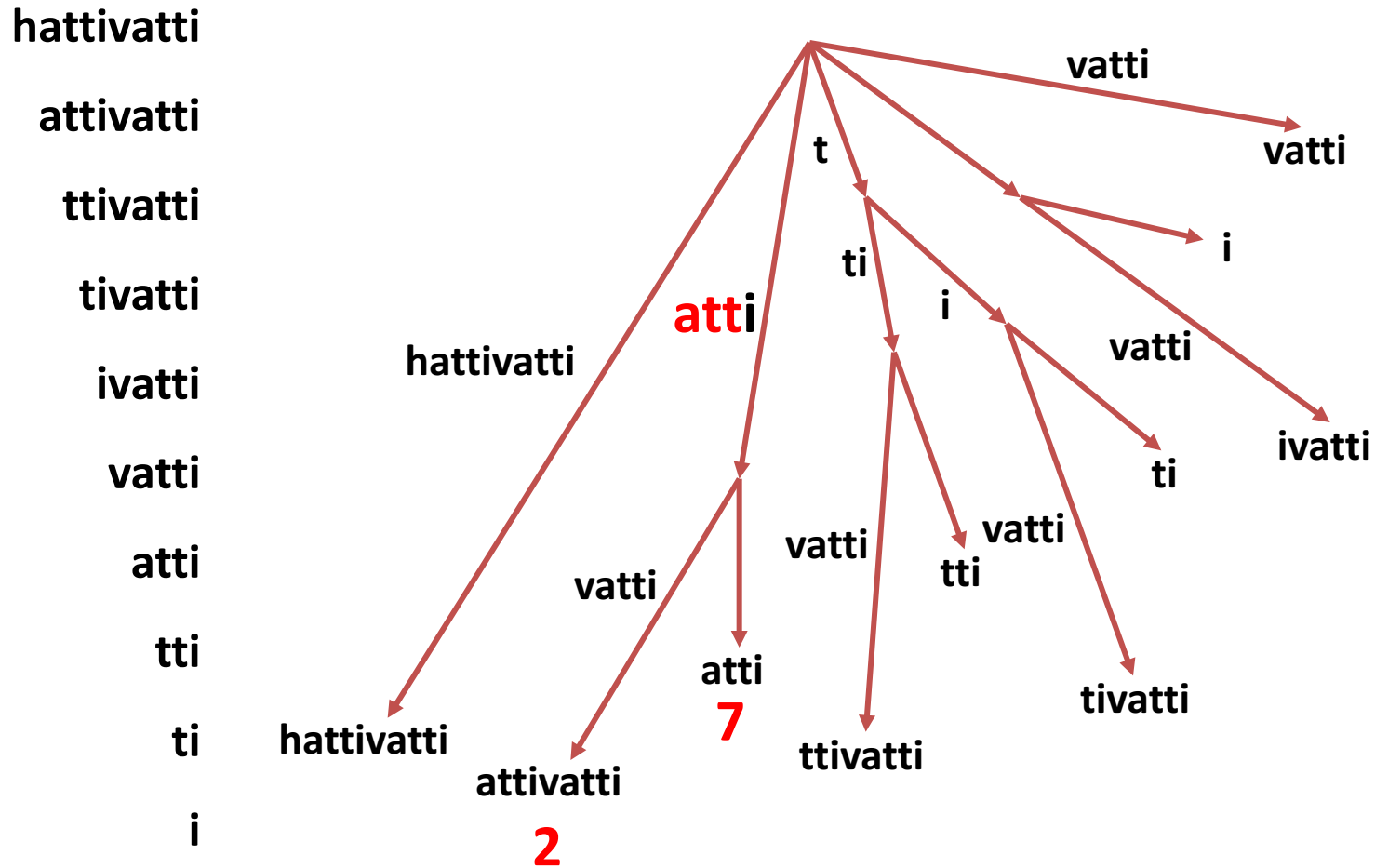
P occurs in T at  
locations 8, 31, ...



P occurs in T  $\Leftrightarrow$  P is a prefix of some suffix of T  $\Leftrightarrow$   
Path for P exists in Tree(T)

All occurrences of P in time  $O(|P| + \#occ)$

# Find att from Tree(hattivatti)



# Linear time construction of Tree(T)

