

# Exact String Matching, Suffix Trees, and Applications

*CAP 5937 Bioinformatics  
University of Central Florida  
Fall 2004*

1

## Problem

- Given a string **P** called the **pattern** and longer string **T** called the **text**, the **exact matching** problem is to find all occurrences, if any, of pattern **P** in text.

2

## Notations

- T: Text  $m$ : length =  $|T|$
- P: Pattern  $n$ : length =  $|P|$
- S: String  $s_1 s_2 s_3 \dots s_n$ 
  - Example:  $S = \text{"AGCTTGA"} \quad |S| = 7$
- Substring:  $S_{i,j} = S_i S_{i+1} \dots S_j$ 
  - Example:  $S_{2,4} = \text{"GCT"}$
- Subsequence of  $S$ : deleting zero or more characters from  $S$ 
  - "ACT" and "GCTT" are subsequences.
- Prefix of  $S$ :  $S_{1,k}$ 
  - "AGCT" is a prefix of  $S$ .
- Suffix of  $S$ :  $S_{h,|S|}$ 
  - "CTTGA" is a suffix of  $S$ .

3

0	1	0	1
1234567890123		1234567890123	
<b>T: xabxyabxyabxz</b>		<b>T: xabxyabxyabxz</b>	
<b>P: abxyabxz</b>		<b>P: abxyabxz</b>	
<b>*abxyabxz</b>		<b>*abxyabxz</b>	
^ ^ ^ ^ ^ ^ ^ ^ *		^ ^ ^ ^ ^ ^ ^ ^ *	
abxyabxz		abxyabxz	
*abxyabxz		^ ^ ^ ^ ^ ^ ^ ^	
*abxyabxz		^ ^ ^ ^	
*abxyabxz			
^ ^ ^ ^ ^ ^ ^ ^			

4

# Z

- Given a string  $S$ , and position  $i > 1$ , let  $Z_i(S)$  be the *length* of the longest substring of  $S$  that starts at  $i$  and matches a prefix of  $S$ .

12345678901

**S = aabcaabxaaz**

$Z_5(S) = 3$  (**aabc...aabx**)

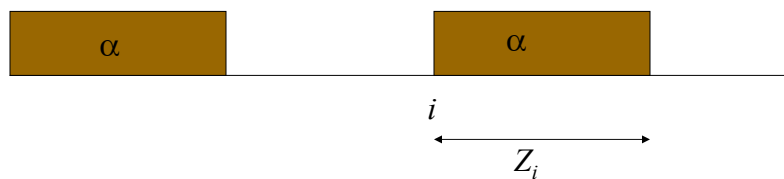
$Z_6(S) = 1$  (**aa...ab**)

$Z_7(S) = Z_8(S) = 0$

$Z_9(S) = 2$  (**aab...aaz**)

5

For any position  $i > 1$  where  $Z_i$  is greater than zero, the **Z-box** at  $i$  is defined as the substring starting at  $i$  and ending at position  $i + Z_i - 1$



6

- For every  $i > 1$ ,  $r_i$  is the right-most endpoint of the Z-boxes that begin at or before position  $i$  and  $l_i$  is the left end of the Z-box.

12345678901234567  
**S = aabaabcaxaabaabcy**

$$Z_{10} = 7, r_{15} = 16, l_{15} = 10$$

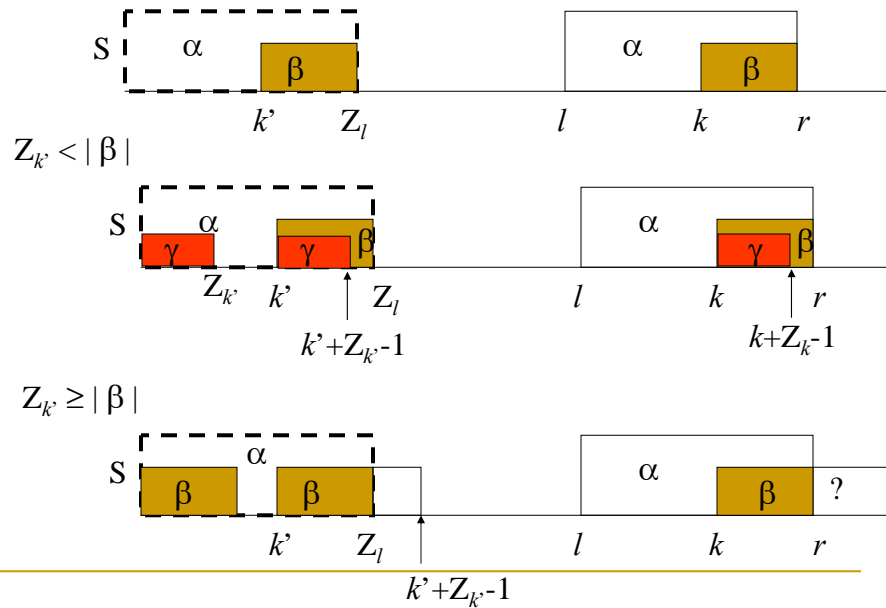
7

## Z calculation (Linear)

- Ref: Gusfield's book Section 1.4
- Given  $Z_i$  for all  $1 < i \leq k-1$  and the current values  $r$  and  $l$ ,  $Z_k$  and updated  $r$  and  $l$  are computed as follows:
  1.  $k > r$ . Find  $Z_k$  by comparing the characters starting at position 1 of S.

8

2.  $k < r$ .



## First string matching algorithm

- Consider string  $P\$T$ , where  $P$  is the pattern,  $T$  is the text and  $\$$  is special alphabet.
- The algorithm is to calculate the  $Z$  value of the string  $P\$T$ . For all  $i$  such that  $Z_i = |P|$ ,  $P$  matches the substring  $T[i..i+|P|-1]$ .
- The calculation time is linear.

## Classical Comparison-Based Methods

- Boyer-Moore Algorithm
- Knuth-Morris-Pratt Algorithm
- Apostolico-Giancarlo Algorithm
- Aho-Corasick Algorithm

11

## Boyer-Morris Algorithm

- Right-to-left scan

**12345678901234567890**

**T: xpbctbzabpqxctbpq**

**P: tpabxab**

12

### Bad character rule

- For each character  $x$  in the alphabet, let  $R(x)$  be the position of right-most occurrence of character  $x$  in  $P$ .  $R(x)$  is defined to be zero if  $x$  does not occur in  $P$ .

12345678901234567890

**T:** xpbctbxabp**q**xctbpq

**P:**     tp**a**bxab                   R(t)= 1

          tpab**x**ab                   R(q)= 0

                  tpab**x**ab

13

- Extended bad character rule

12345678901234567890

**T:** xpbca**a**bxabp**q**xctbpq

**P:**     ap**t**bxab                   R(a)= 6

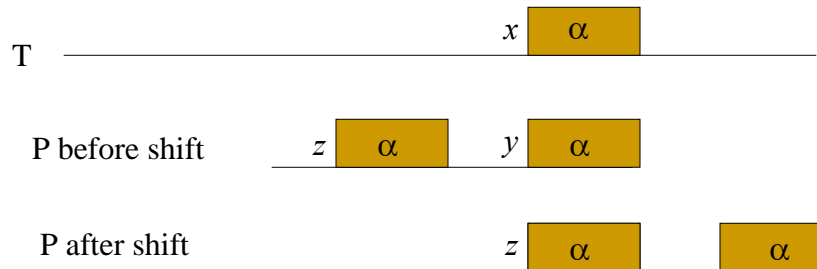
          ap**t**bxab                   R(q)= 0

                  ap**t**bxab

When a mismatch occurs at position  $i$  of  $P$  and the mismatched character in  $T$  is  $x$ , then shift  $P$  to the right so that the closest  $x$  to the left of position  $i$  in  $P$  is below the mismatched  $x$  in  $T$

14

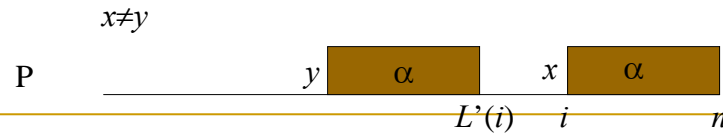
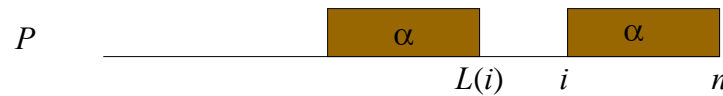
- Strong good suffix rule



0                    1  
 123456789012345678  
**T:** prstabst**ub**abvqxr**st**  
                                   \*  
**P:**    qcab**d**ab**d**ab  
           1234567890  
                   qcab**d**ab**d**ab    weak rule  
                   qcab**d**ab**d**ab    strong rule



- For each  $i$ ,  $L(i)$  is the largest position less than  $n$  such that string  $P[i..n]$  matches a suffix of  $P[1..L(i)]$ . If no,  $L(i) = 0$ .
- So is  $L'(i)$  with the characters to the left of the suffix are different.

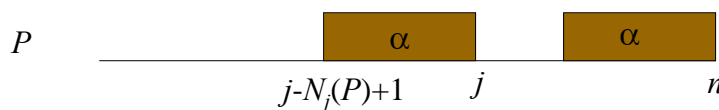


17

$N_j(P)$  is the length of the longest suffix of the substring  $P[1..j]$  that is also a suffix of the full string  $P$ .

$Z_i(P)$  is the length of the longest prefix of  $P[i..n]$  that is also a prefix of the full string  $P$ .

$$\text{So, } N_j(P) = Z_{n-j+1}(P)$$



18

$\alpha$  (from 1 to  $i+Z_i-1$ )  
 $\alpha$  (from  $i$  to  $n$ )

$P$  (from  $j-N_j(P)+1$  to  $j$ )  
 $\alpha$  (from  $j-N_j(P)+1$  to  $j$ )  
 $\alpha$  (from  $j$  to  $n$ )

**123456789**  
**P cabda**b**dab**       $L(8) = 6, L'(8) = 3$   
**abda**b****  
**abdab**       $N_3(P)=2, N_6(P)=5$

19

## Calculation of $L(i)$

- $L(i)$  is the largest index  $j$  less than  $n$  such that  $N_j(P) \geq |P[i..n]|$ .
- $L'(i)$  is the largest index  $j$  less than  $n$  such that  $N_j(P) = |P[i..n]|$ .
- Algorithm
  - For  $i := 1$  to  $n-1$  do  $L'(i) := 0$ ;
  - For  $j := 1$  to  $n-1$  do
    - Begin  $i := n-N_j(P) + 1$ ;  $L'(i) := j$  End

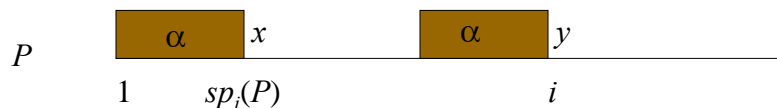
20

- Let  $l(i)$  denote the length of the largest suffix of  $|P[i..n]|$  that is also a prefix of  $P$ , if one exists. If none exists, then let  $l(i) = 0$ .
- $l(i)$  equals the largest  $j < |P[i..n]|$  such that  $N_j(P) = j$ .

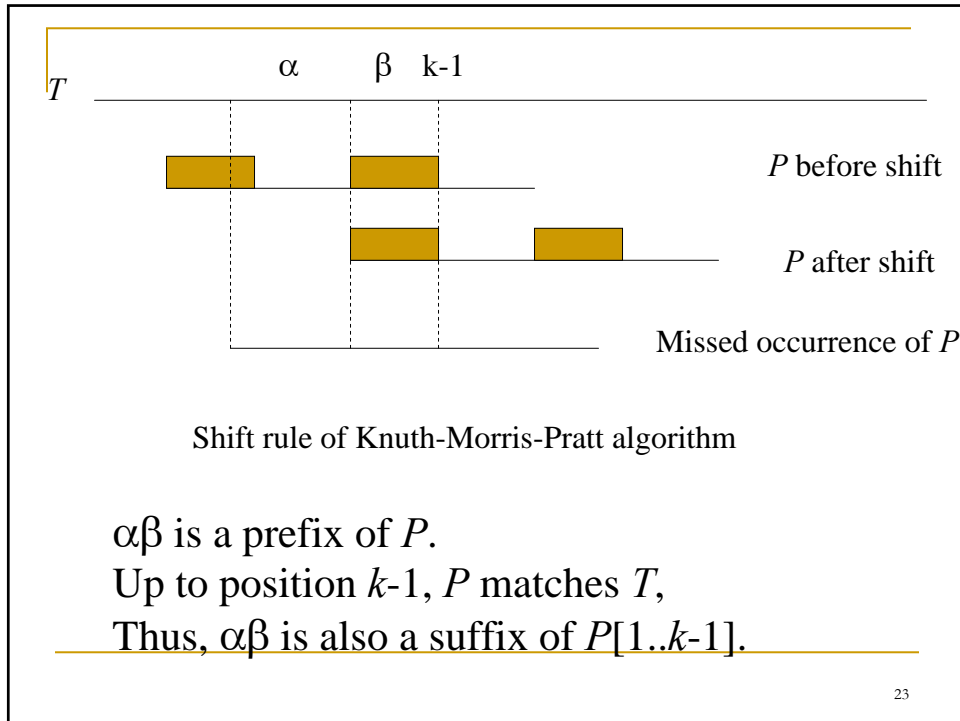
21

## Knuth-Morris-Pratt Algorithm

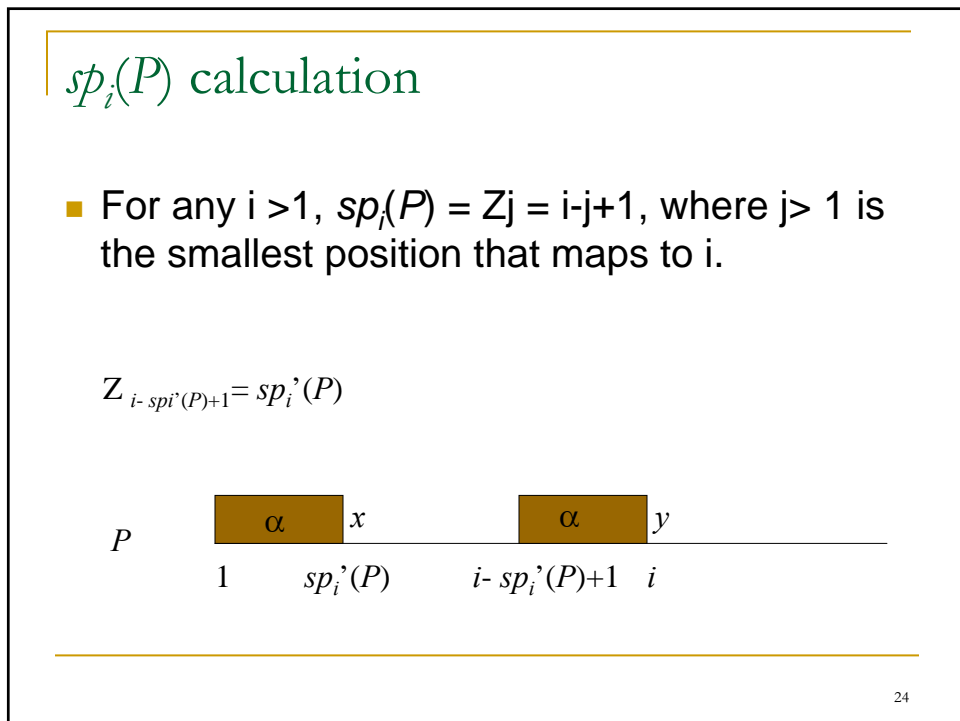
- For each position  $i$  in pattern  $P$ , defines  $sp_i(P)$  (resp.  $sp'_i(P)$ ) to be the length of the longest proper suffix of  $P[1..i]$  that matches a prefix of  $P$  and (resp.  $P(i+1) \neq P(sp'_i(P)+1)$ ).



22



23



24

- 123456789012345678
- xyabcxabcxadcdqfeg
- abcxabcde
- 123456789
- abcxabcde
- abcxabcde
- $sp_2=0, sp_3=0, sp_4=0, sp_5=1, sp_6=2,$   
 $sp_7=3, sp_8=0, sp_9=0.$

25

## Theorem

- After a mismatch at position  $i+1$  of  $P$  and a shift of  $i-sp_i'$  places to the right, the left-most  $i-sp_i'$  characters of  $P$  are guaranteed to match their counterparts in  $T$ .
- For any alignment of  $P$  with  $T$ , if character 1 through  $i$  of  $P$  match the opposing characters of  $T$  but character  $i+1$  mismatches  $T(k)$ , then  $P$  can be shifted by  $i-sp_i'$  places to the right without passing any occurrence of  $P$  in  $T$ .

26

## Classical Comparison-Based Methods

- Boyer-Moore Algorithm
- Knuth-Morris-Pratt Algorithm
- Apostolico-Giancarlo Algorithm
- Aho-Corasick Algorithm

### A Demonstration

27

## Exact matching with a set of patterns

**Exact set matching problem** is to find all the occurrences in a text  $T$  of a set of patterns  $\mathcal{P} = \{P_1, \dots, P_z\}$ .

**Dictionary problem:** Given a text  $T$ , ask if  $T$  is a pattern in  $\mathcal{P}$ .

28

## Keyword Tree

- Keyword tree  $\mathcal{K}$  for  $\mathcal{P}$ 
  - each edge is labeled with exactly one character
  - any two edges out of the same node have distinct labels
  - every pattern  $P_i$  in  $\mathcal{P}$  maps to some node  $v$  of  $\mathcal{K}$  such that the characters on the path from the root of  $\mathcal{K}$  to  $v$  exactly spell out  $P_i$ , and every leaf of  $\mathcal{K}$  is mapped to by some pattern in  $\mathcal{P}$ .

29

- Assumption: No pattern in  $\mathcal{P}$  is a proper substring of any other pattern in  $\mathcal{P}$ .

$\mathcal{L}(v)$  = the labels from root to the node  $v$ .

$lp(v)$  = the length of the longest proper suffix of string of  $\mathcal{L}(v)$  that is a prefix of some pattern in  $\mathcal{P}$ .

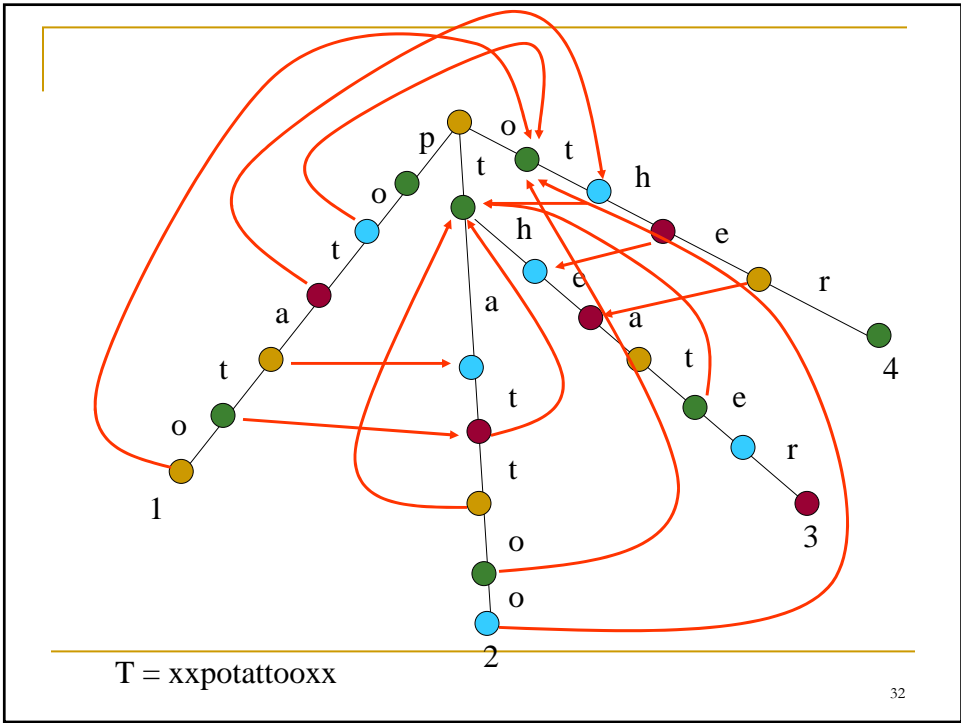
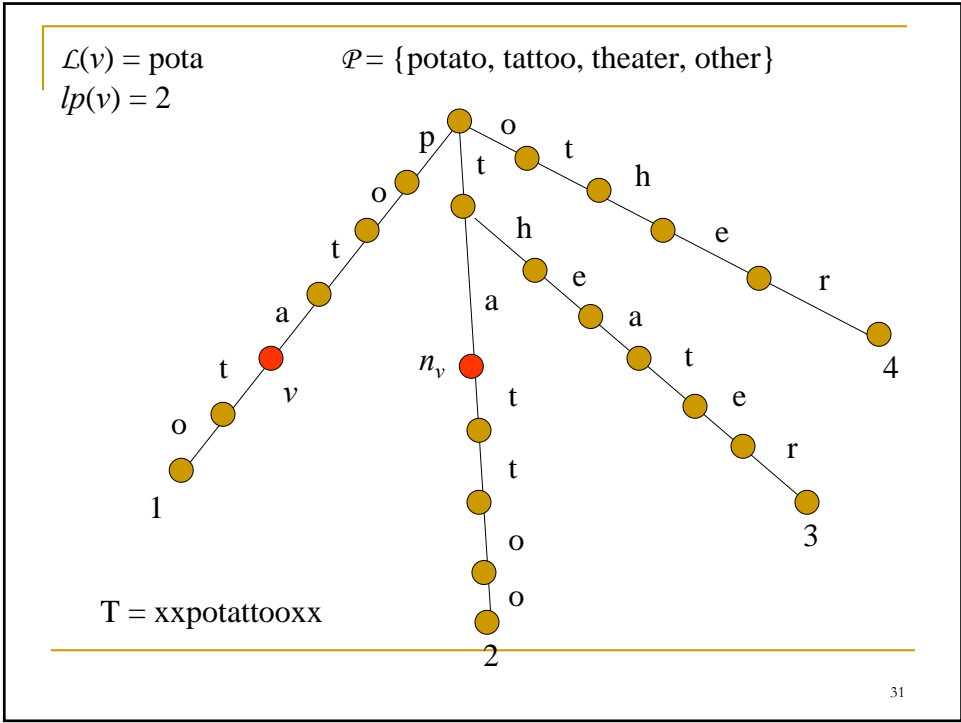
Lemma: Let  $\alpha$  be the  $lp(v)$ -length suffix of string  $\mathcal{L}(v)$ . Then there is a unique node in the keyword tree that is labeled by string  $\alpha$ .

The unique node is denoted by  $n_v$ .

When  $lp(v) = 0$ ,  $n_v$  is the root.

$n_v$  for all  $v$  can be constructed in linear time.

30



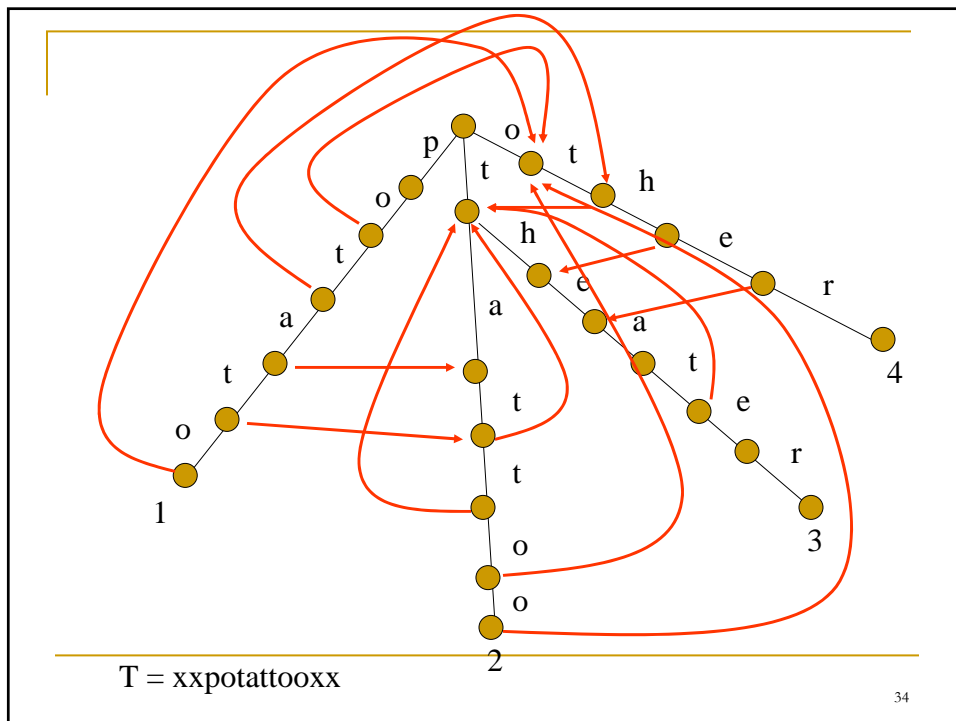


$n_v$  is computed in linear time.

Consider, for each pattern, two pointers, one points the current processing position and the other points to left end of the match suffix.

We will see that each operation causes the pointers move forward, but they only move  $2n$  times.

33



## Aho-Corasick Algorithm

- Without assumption.
- $\mathcal{P} = \{\text{acatt}, \text{ca}\}$ ,  $T = \text{acatx}$
- Suppose in a keyword tree  $\mathcal{K}$  there is a direct path of failure links from a node  $v$  to a node that numbered with pattern  $i$ . Then pattern  $P_i$  must occur in  $T$  ending at position  $c$  whenever node  $v$  is reached during the search of Aho-Corasick algorithm.

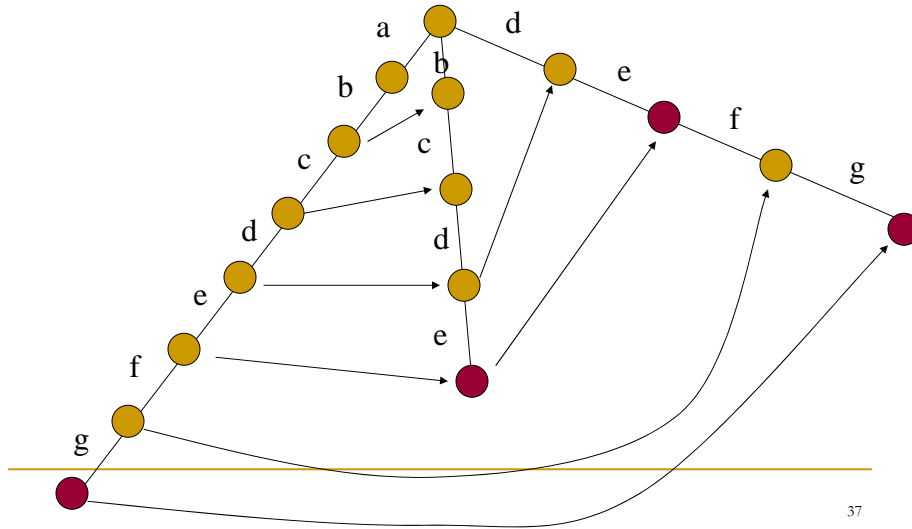
35

- Suppose a node  $v$  has been reached during the algorithm. Then the pattern  $P_i$  occurs in  $T$  ending at position  $c$  only if  $v$  is numbered  $i$  or there is a directed path of failure links from links from  $v$  to the node numbered  $i$ .
- The output link at  $v$  points to that numbered node other than  $v$  that is reachable from  $v$  by the fewest failure links.

36

$\mathcal{P} = \{abcdefg, de, bcde, defg\}$

$T = xabcdefxcdefgx$



## Matching against DNA Library

- **Sequence-tagged-sites(STS)**
  - A DNA string of 200-300 bps whose right and left ends, of length 20 – 30 bps each, occur only once in the entire genome.
- **Expressed sequence tags (EST)**
  - A STS that comes from genes rather than parts of inter-gene DNA. (Obtained from mRNA or cDNA)

38

- The set of patterns: all known STSs or ESTs
- Text: a newly sequenced genome
- Goal: To identify STSs or ESTs occur in the newly sequences genome

39

## Seminumerical String Matching

- *Shift-And* Method
  - Let  $M$  be an  $n$  by  $m+1$  binary matrix.  $M(i,j) = 1$  if and only if the first  $i$  characters of  $P$  exact match the  $i$  characters of  $T$  ending at character  $j$ .
  - $M(n,j) = 1$  if and only if an occurrence of  $P$  ends at position  $j$  of  $T$ .
  - *Bit-Shift*( $j-1$ ) : shift column  $j-1$  down by one position and set the first to 1.

40

- **T= xabxab**a**axa**
- **P= abaac**
- $C(8)^T = (1\ 0\ 1\ 0\ 0)$
- Bit-Shift  $C(8)^T = (1\ 1\ 0\ 1\ 0)$
- $T(9) = a$ ,  $U_a^T = (1\ 0\ 1\ 1\ 0)$
- $C(9)^T = C(8)^T$  **AND**  $U_a^T = (1\ 0\ 0\ 1\ 0)$
- $M(i, j) = 1$  if and only if
  - $M(i-1, j-1) = 1$  and  $U_{T(i)}(j) = 1$

41

- Advantage of *Shift-And*
  - Very efficient if  $n$  is less than the size of single computer word.
  - Only two columns are needed in each computation time.
- *Agrep*: The Shift-And method with errors.
  - $M^k(i, j)$  is 1 if and only if at least  $i-k$  of the first  $i$  characters of  $P$  match the  $i$  characters up through character  $j$  of  $T$ .
- In *Agrep*, the user chooses a value of  $k$  and then the arrays  $M, M^1, \dots, M^k$  are computed.

42

$$M(j) = M^{l-1}(j)$$

**OR** [*Bit-Shift*( $M(j-1)$ ) **AND**  $U(T(j))$ ]

**OR**  $M^{l-1}(j-1)$

Computation time =  $O(kmn)$

43

## Karp-Rabin fingerprint method

- $T_r^n$  denote the  $n$ -length substring of  $T$  starting character  $r$ .

44

- There is an occurrence of  $P$  starting at position  $r$  if and only if  $H(P) = H(T_r)$ .
- $H_p(P) = H(P) \bmod p$  and  $H_p(T_r) = H(T_r) \bmod p$  are called fingerprint of  $P$  and  $T_r$ .
- $H_p(P) = H_p(T_r)$  may introduce false match.
- $\pi(u)$  = the number of primes that are less than or equal to  $u$ .
- 

45

- If  $u \geq 29$ , then the product of all the primes that are less than or equal to  $u$  is greater than  $2^u$ .
- If  $u \geq 29$  and  $x$  is any number less than or equal to  $2^u$ , then  $x$  has fewer than  $\pi(u)$  (distinct) prime divisors.

46

- Let  $P$  and  $T$  be any strings such that  $nm > 29$ . Let  $l$  be any positive integer. If  $p$  is a randomly chosen prime number less than or equal to  $l$ , then the probability of a false match between  $P$  and  $T$  is less than or equal to  $\pi(mn)/\pi(l)$ .
- R: the set of position in  $T$ ,  $P$  does not begin.
- Consider
- There are at most  $\pi(mn)$  prime divisors
- $p$  is randomly chosen from  $l$ .

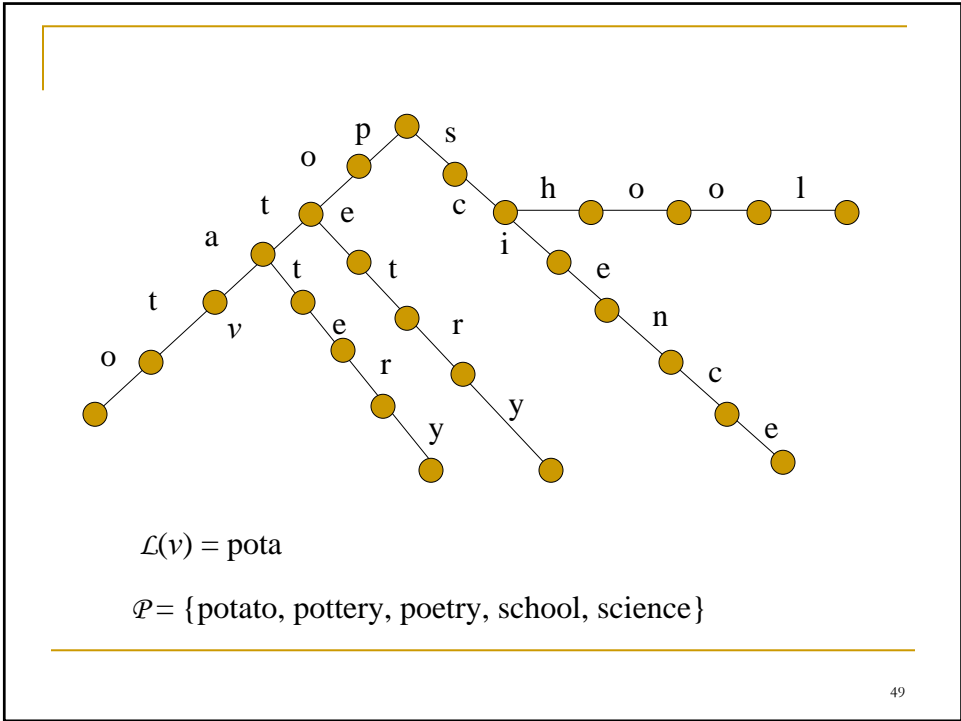
47

## Algorithm

- Choose a positive integer  $l$ .
- Randomly pick a prime number less than or equal to  $l$ , and compute  $H_p(P)$ .
- For each position  $r$  in  $T$ , compute  $H_p(T_r)$  and test if it equals  $H_p(P)$ .
- When  $l = nm^2$ , the probability of a false match is at most  $2.53/m$ .

48

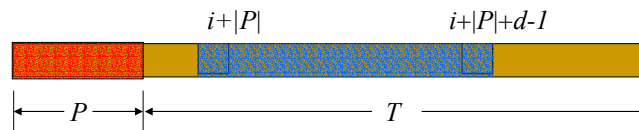




# Motivating Suffix Tree

## Exact String Matching

- Input:  $P$  and  $T$ .
- Output: All occurrences of  $P$  in  $T$ .
- Time:  $O(|P| + |T|)$
- Technique: Z values of  $PT$ .
  - $Z(i + |P|) \geq |P|$  iff  $P = T[i \dots i + |P| - 1]$ .



51

## Question 1

- Solving the *Exact String Matching* problem in  $O(|P|)$  time under the assumption that  $T$  is known and already pre-processed?
  - E.g.,  $T$  is a dictionary whose content does not change frequently.
- Answer:

52

## Question 2

- Solving the *Exact String Matching* problem in  $O(|T|)$  time under the assumption that  $P$  is known and already pre-processed?
  - E.g.,  $P$  is one of your private collection of DNA sequence.
- Answer:

53

## A Less Ambitious Version

### The *Substring Problem*

- Input:  $P$  and  $T$ .
- Output: an occurrence of  $P$  in  $T$ .

54

## Question 2

- Solving the *Substring problem* in  $O(|T|)$  time under the assumption that  $P$  is known and already pre-processed?
- Answer:

55

## Question 1

- Solving the *Substring problem* in  $O(|P|)$  time under the assumption that  $T$  is known and already pre-processed?
- Answer:

56

## To $P$ or not to $P$ .....


- Preprocessing  $P$ 
  - Gusfield
  - Boyer-Moore
  - Knuth-Morris-Pratt
- Preprocessing  $T$ 
  - Suffix tree

57

## From Suffix Trie to Suffix Tree

58

## Notation Change

- Input:  $P$  and  $S$ .
- Output: an occurrence of  $P$  in  $S$ .
- For example,
  - $S = b b a b b a a b$
  - $P = b a a$  

59

## Suffixes of $S$

$S$	=	$b b a b b a a b$	
$S[1..8]$	=	$b b a b b a a b$	
$S[2..8]$	=	$b a b b a a b$	1st suffix
$S[3..8]$	=	$a b b a a b$	2nd suffix
$S[4..8]$	=	$b b a a b$	3rd suffix
$S[5..8]$	=	$b a a b$	4th suffix
$S[6..8]$	=	$a a b$	5th suffix
$S[7..8]$	=	$a b$	6th suffix
$S[8..8]$	=	$b$	7th suffix
			8th suffix

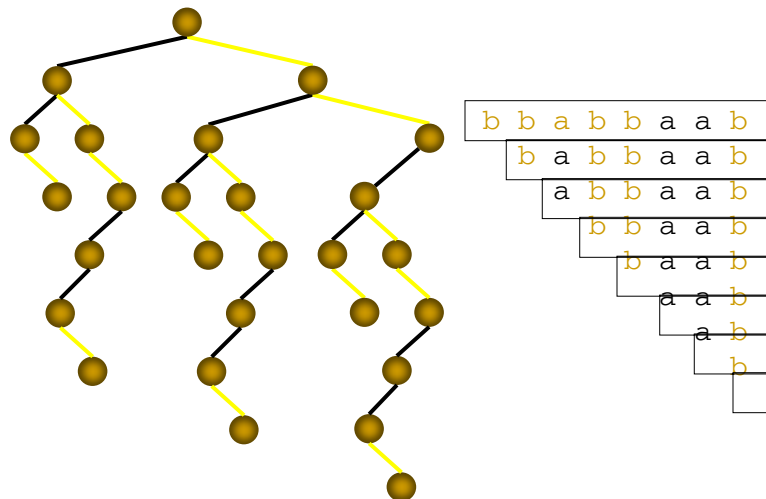
60

KEY: P occurs in S iff P is a prefix of a suffix of S.

S	=	b	b	a	b	b	a	a	b	
S[1...8]	=	b	b	a	b	b	a	a	b	1st suffix
S[2...8]	=		b	a	b	b	a	a	b	2nd suffix
S[3...8]	=			a	b	b	a	a	b	3rd suffix
S[4...8]	=				b	b	a	a	b	4th suffix
S[5...8]	=					b	a	a	b	5th suffix
S[6...8]	=						a	a	b	6th suffix
S[7...8]	=							a	b	7th suffix
S[8...8]	=								b	8th suffix

61

T = Suffix Trie of S



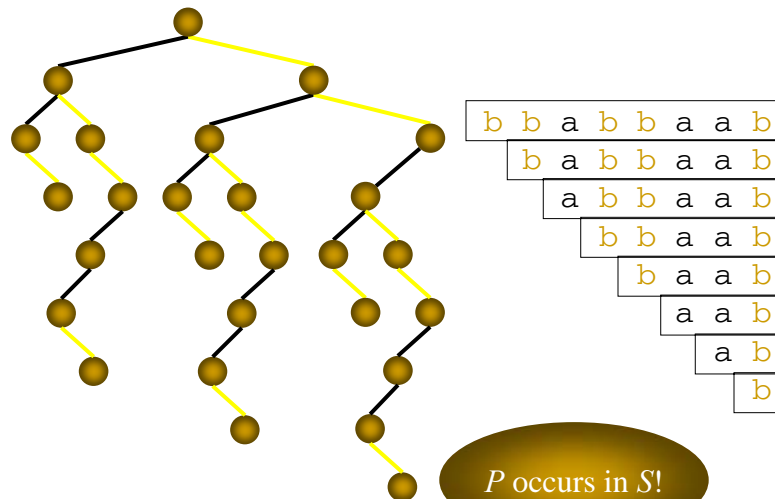
62

## Why suffix trie?

- The following statements are equivalent.
  - $P$  occurs in  $S$ .
  - $P$  is a prefix of a suffix of  $S$ .
  - $P$  corresponds to a path of  $T$  starting from the root of  $T$ .

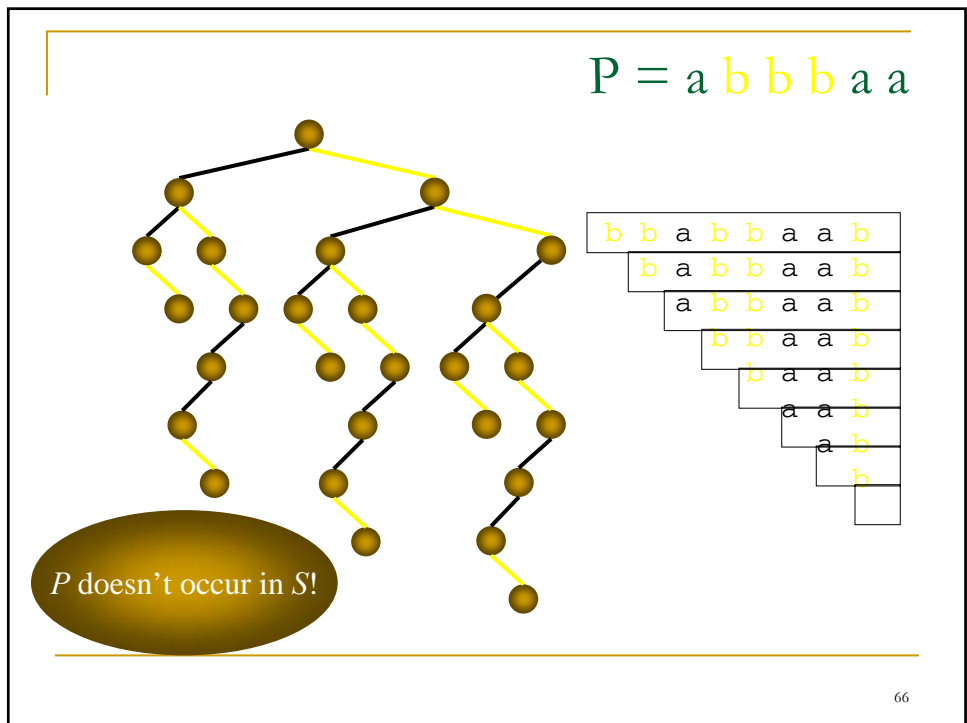
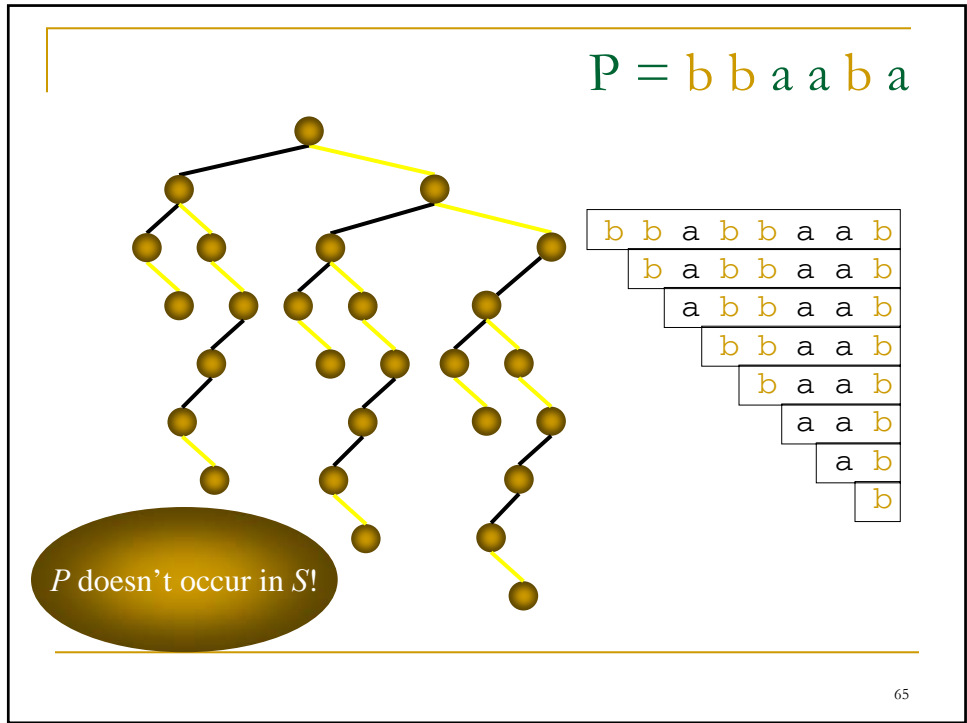
63

$P = b a b b a$



64

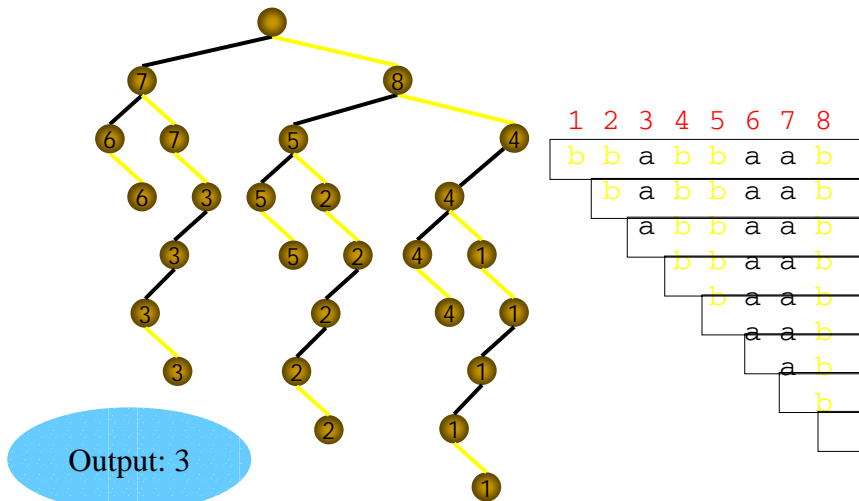




Q: Where does  $P$  occur in  $S$ ?

67

$P = a b b a a$



68

# Question

$$\Theta(|S|)$$

$$\Theta(|S| \log |S|)$$

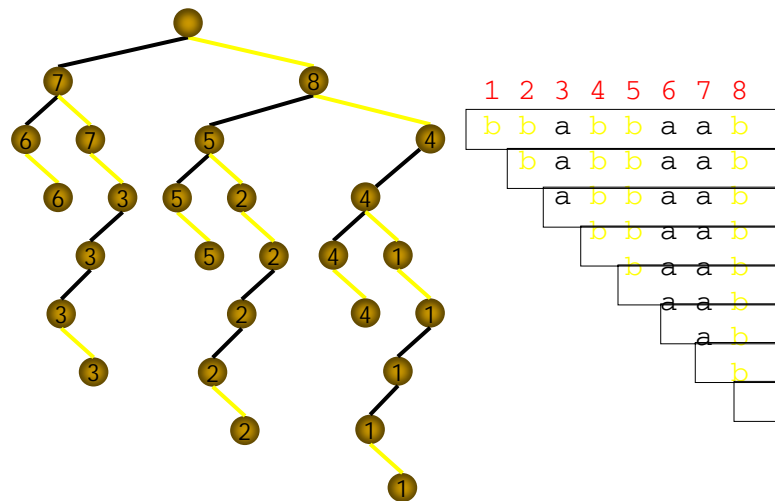
$$\Theta(|S|^2)$$

$$\Theta(|S|^3)$$

Time complexity for constructing the suffix trie  $T$  of  $S$ ?

69

$$\text{Time} = O(|S|^2)$$



70

---

$$\text{time} = \Omega(|S|^2)$$

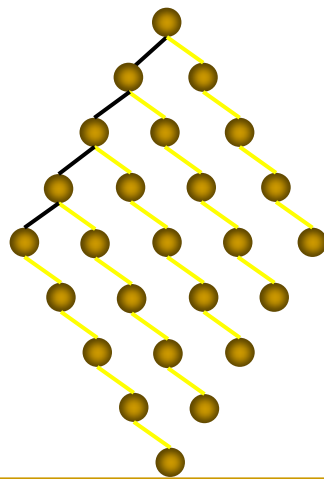
- How to establish a lower bound?
- Answer:

---

71

---

$$S = a a a a b b b b$$



---

72

---

## Summary

- Suffix trie is good in solving *Substring Problem*, but may require  $\Omega(|S|^2)$  time and space.
- Question: is there a **compact representation** of suffix trie that needs only  $O(|S|)$  time and space?

---

73

---

## Suffix Tree

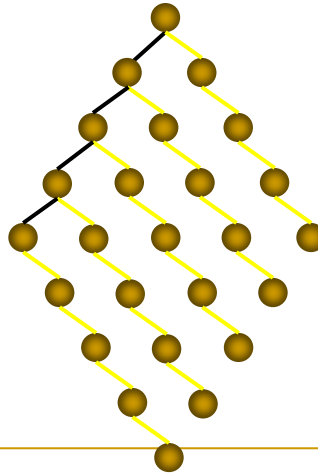
---

A compact representation of suffix trie

74

## Observations on Trie T of S

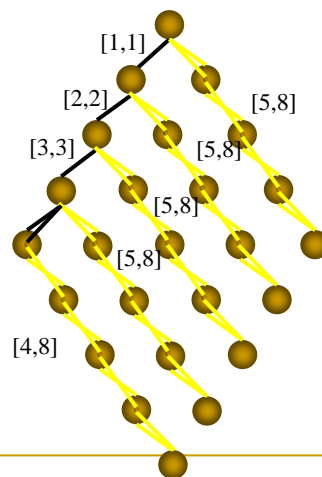
- T has at most  $|S|$  leaves.
  - Why?
- T has at most  $|S|$  branching nodes.
  - Why?



75

$S = a a a a b b b b$

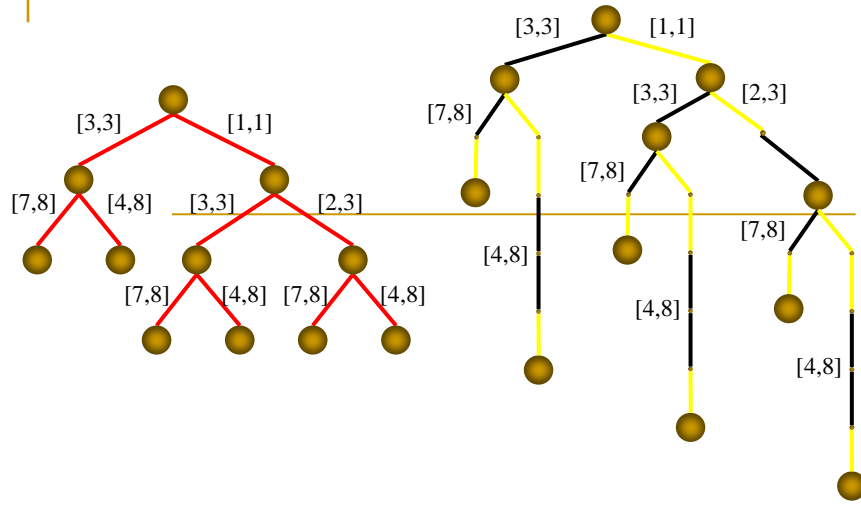
- Keeping leaves and branching nodes only.
- compact representation of edge labels



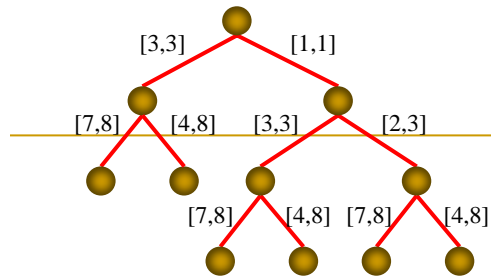
76



S = b b a b b a a b



S = b b a b b a a b





## Question

- The space complexity of suffix tree
  - $O(|S|)$
  - $O(|S| \log |S|)$
  - $O(|S|^2)$
  - $O(|S|^3)$
- Why?
  - Number of nodes =
  - Number of edges =
  - Space required by each edge =

81

## The challenge

Constructing Suffix Tree in Linear Time

82

## History of Suffix Tree Algorithms

- [Weiner, *IEEE FOCS* 1973]
  - Linear time but expensive in space.
  - D. E. Knuth: “the algorithm of 1973”.
- [McCreight, *J. ACM* 1976]
  - Linear time and quadratic space.
- [Ukkonen, *Algorithmica* 1995]
  - Linear time and linear space.
  - Much better readability.

83



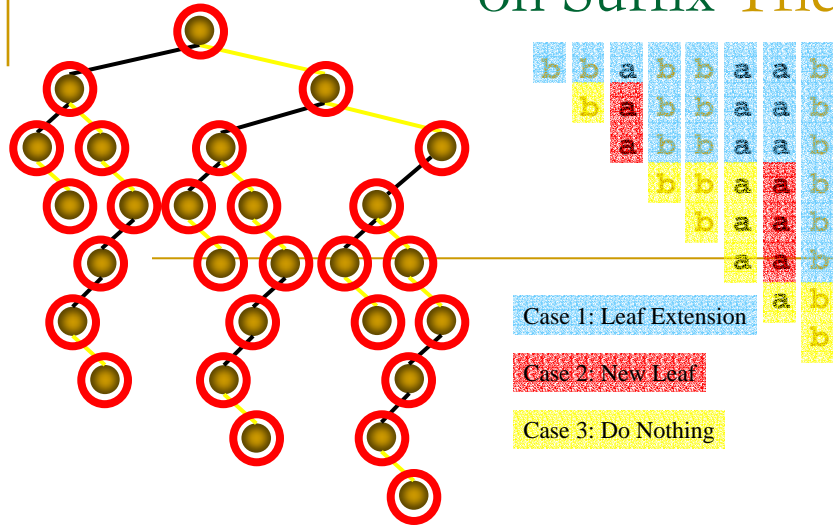
Academy Professor ,  
Department of Computer  
Science , University of  
Helsinki, Finland

<http://www.cs.helsinki.fi/u/ukkonen/>

Esko Ukkonen: On-line construction of  
suffix-trees. *Algorithmica* 14 (1995), 249-  
260

84

## Ukkonen's approach on Suffix Trie



85

## Growing Suffix Trie

- Three cases while growing trie
  - Case 1: growing an edge at a leaf.
  - Case 2: growing a new branch of leaf.
  - Case 3: does not change the tree structure.

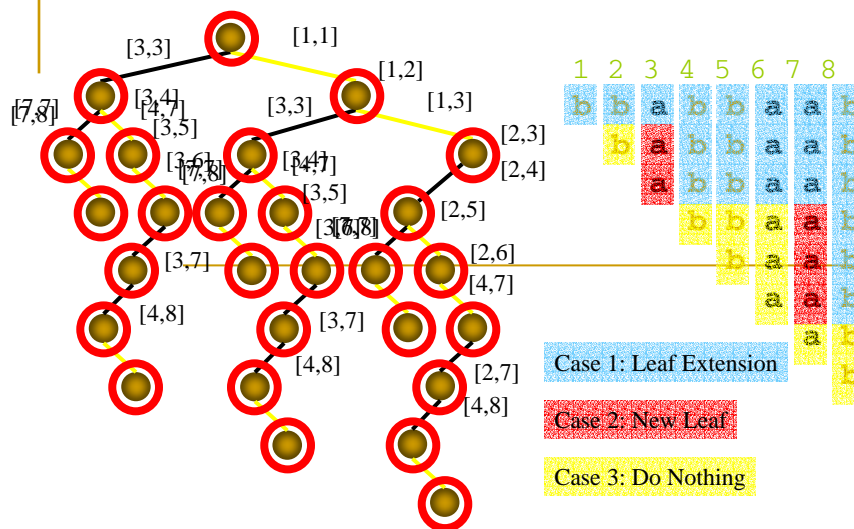
86

## Three Phase Theorem

- Those  $k$  steps in the  $k$ -th iteration have the following pattern:
  - some (at least one) Case-1 steps,
  - followed by some (could be zero) Case-2 steps,
  - followed by some (could be zero) Case-3 steps.

87

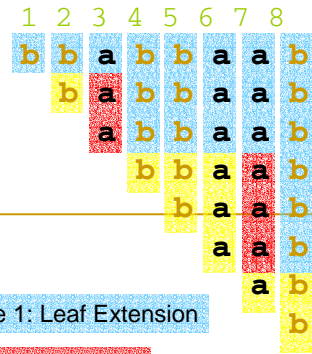
## Thinking in Suffix Tree



88

## Saving a lot of efforts

- We can simply ignore all Case-1 steps.
- Recall that the number of Case-2 steps is at most  $|S|$ .
- Q: Is this good enough?

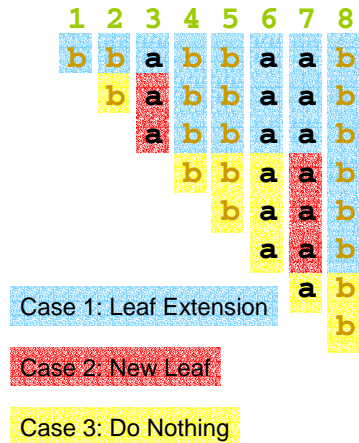


Case 1: Leaf Extension  
 Case 2: New Leaf  
 Case 3: Do Nothing

## How does Ukkonen overcome the problem of too many Case-3 steps?

Completely ignore them.....  
 Do nothing when nothing happen.....

## Saving even more efforts



91

## Rough idea

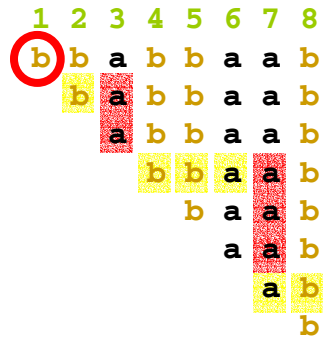
- Just keep one **current growing point** throughout the execution.
- **Deriving the** new position of the current growing point from its previous position (with the help of **suffix links** )

92

## Only one growing point

The challenges: How do we **derive** the position of the current growing point?

- Vertically (case 2)
- Horizontally (case 3)
- Q: Which one is easier?



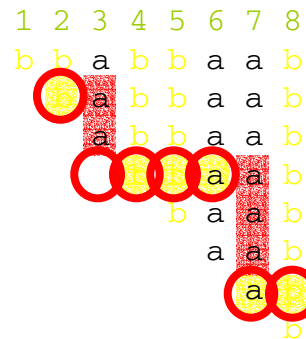
Case 2: New Leaf

Case 3: Do Nothing

93

## Horizontally, ...

- Moving from iteration  $k-1$  to iteration  $k$ .
- The growing point does not move!
- This is the easier case.



Case 2: New Leaf

Case 3: Do Nothing

94

## Vertically, ...

- Moving from Step  $i$  to Step  $i+1$  in the same iteration.
- The growing point moves **dramatically**.
- This is the tougher case.

1	2	3	4	5	6	7	8
b	b	a	b	b	a	a	b
		a	b	b	a	a	b
		a	b	b	a	a	b
			b	b	a	a	b
				b	a	a	b
					a	a	b
						a	b
							b

Case 2: New Leaf

Case 3: Do Nothing

95

## Suffix link

Keep records of what have been done --  
- (Dynamic Programming)

96



## Recording What's Done

- Whenever a **vertical movement** reaching the destination, keep a record of the movement by using a link.
- Later on, we might want to follow these recorded linkages.
- These links are thus called the **suffix links**.

97

## Why called “Suffix Links”?

- Note that the destination of the link is the **(-1)-suffix** of the starting.
- That is, a suffix link links a length  $n+1$  suffix to a length  $n$  suffix.

	1	2	3	4	5	6	7	8
	b	b	a	b	b	a	a	b
			a	b	b	a	a	b
			a	b	b	a	a	b
				b	b	a	a	b
					b	a	a	b
						a	a	b
							a	b
								b

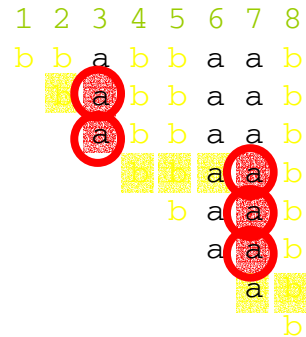
Case 2: New Leaf

Case 3: Do Nothing

98

## Property of Suffix Links (1)

- The starting point of a suffix is an internal node,
  - Not a leaf
  - No the middle part of some suffix tree edge.
- Why?



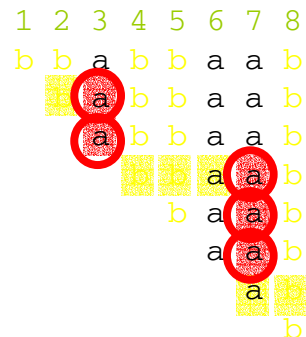
Case 2: New Leaf

Case 3: Do Nothing

99

## Property of Suffix Links (2)

- Every internal node must be a starting point of a suffix link.
- Why?



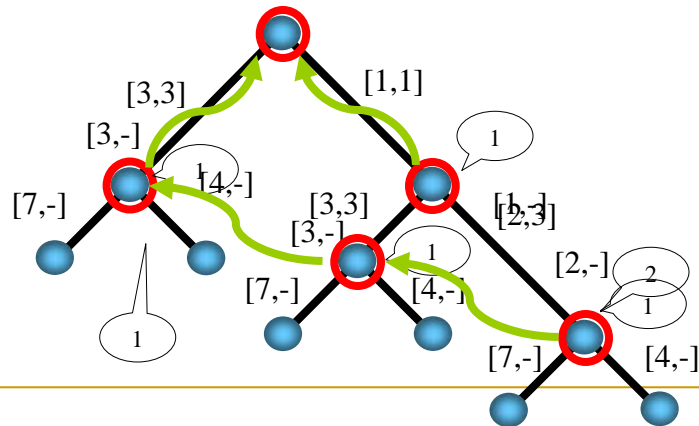
Case 2: New Leaf

Case 3: Do Nothing

100

## Using suffix links

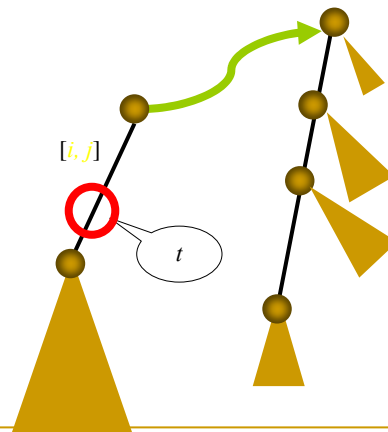
$S = b b a b b a a b$



101

## Traversal with the help of suffix links: phase (1)

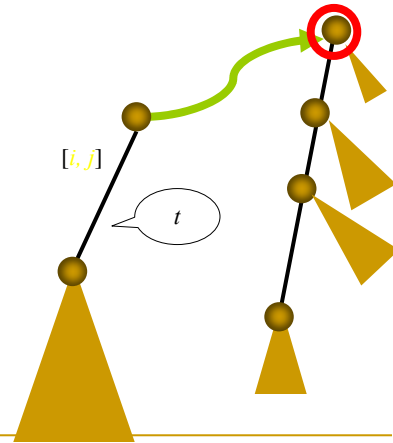
- Going up to a closest internal node (whose suffix link must be available). Suppose this upward traversal passes through  $t$  characters.
- Following the suffix link that starts from this internal node.



102

## Traversal with the help of suffix links: phase (2)

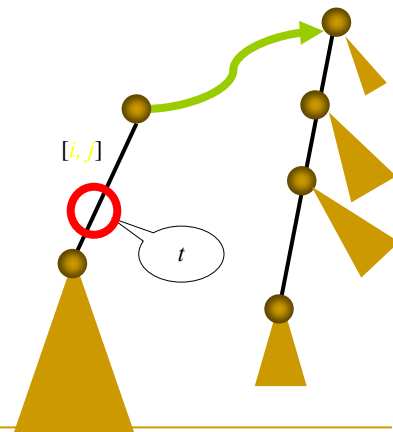
- Going down by matching the  $t$ -character substring  $S[i, i+t-1]$  of  $S$ .



103

## Running Time?

- Naïvely:  $O(t)$ .
- Cleverly:  $O(1+d)$ , where  $d$  is the number of internal nodes being went through during phase (2).



104

## Overall Time = $O(|S|)$

- Suppose  $d_i$  is the  $d$  in the  $i$ -th Case-2-step traversal.
- It suffices to show  $d_1 + d_2 + \dots + d_{|S|} = O(|S|)$ .

1	2	3	4	5	6	7	8
b	b	a	b	b	a	a	b
		a	b	b	a	a	b
		a	b	b	a	a	b
			b	b	a	a	b
				b	a	a	b
				a	a	b	
				a	b		
					b		

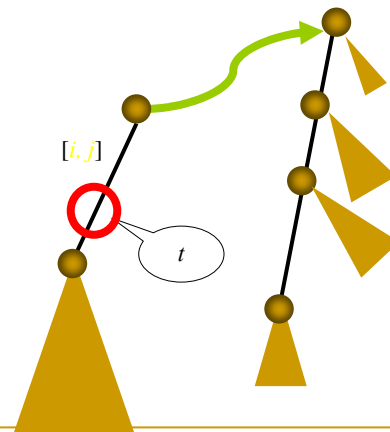
Case 2: New Leaf

Case 3: Do Nothing

105

## $\Phi$ = the “slack” of the growing point

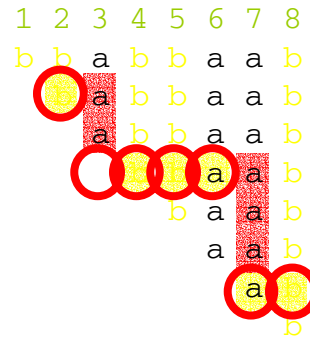
- The slack means the distance between a position  $P$  and the closest internal node above  $P$ .



106

## case-3 traversal

- Each case-3 traversal (i.e., horizontal movement) can only increase the value of  $\Phi$  by **at most one**.
- (It can even decrease the value of  $\Phi$ .)



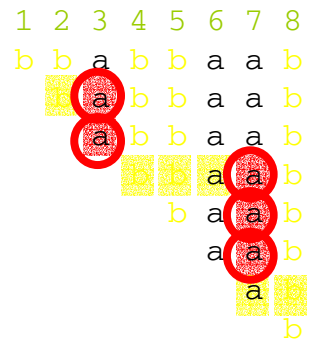
Case 2: New Leaf

Case 3: Do Nothing

107

## case-2 traversal

- The  $i$ -th case-2 traversal (i.e., vertical movement) decreases the value of  $\Phi$  by at least  $d_i$ .



Case 2: New Leaf

Case 3: Do Nothing

108

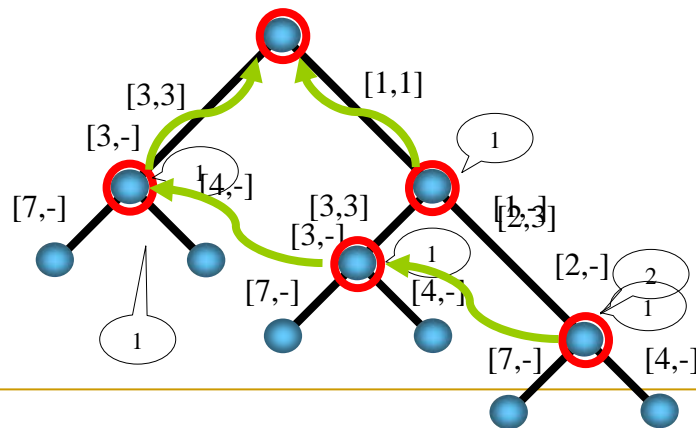
$$d_1 + d_2 + \dots + d_{|S|} = O(|S|)$$

- Initial  $\Phi = O(1)$ .
- $\Phi$  can be increased by one for at most  $|S|$  times (because there are at most  $|S|$  horizontal movements (i.e., case-3 traversals)).
- Since  $\Phi$  is always non-negative, the above bound is proved.

109

## Using suffix links

$S = b b a b b a a b$



110

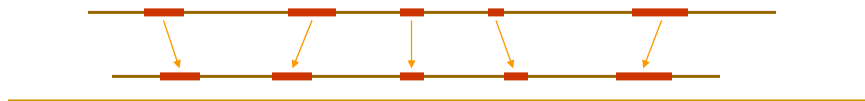
# Applications of Suffix Tree in Bioinformatics

111

## Rapid global alignment

Genomic regions of interest contain ordered islands of similarity

- E.g. genes
- 1. Find local alignments
- 2. Chain an optimal subset of them



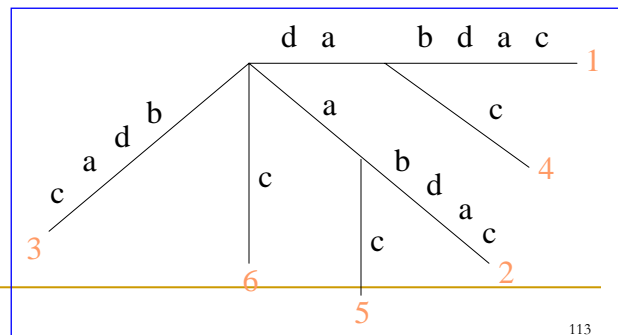
112



## Suffix Trees

- Suffix trees are a method to find all maximal matches between two strings (and much more)

Example:  
x = dabdac



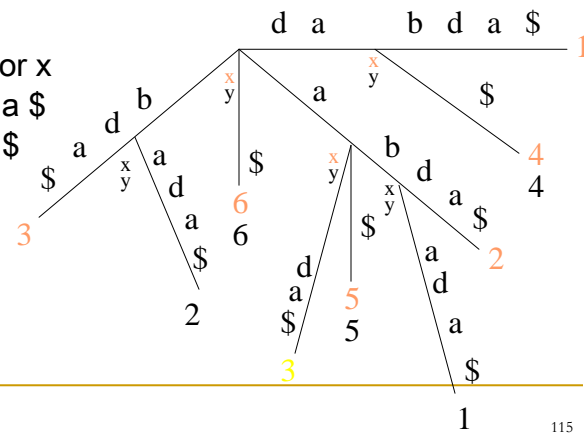
## Application: Find all Matches Between x and y

1. Build suffix tree for x, mark nodes with x
2. Insert y in suffix tree, mark all nodes y "passes from" with y
  - The path label of every node marked both 0 and 1, is a common substring

## Example of Suffix Tree Construction for x, y

x = d a b d a \$  
y = a b a d a \$

1. Construct tree for x
2. Insert a b a d a \$
3. Insert b a d a \$
4. Insert a d a \$
5. Insert d a \$
6. Insert a \$
7. Insert \$



## Application: Online Search of Strings on a Database

Say a database  $D = \{s_1, s_2, \dots, s_n\}$   
(eg. proteins)

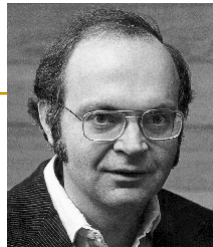
Question: given new string  $x$ , find all matches of  $x$  to database

1. Build suffix tree for  $\{s_1, \dots, s_n\}$
2. All new queries  $x$  take  $O(|x|)$  time (somewhat like BLAST)

## Longest Common Substring

- Given two strings **S** and **T**.
- Find the longest common substring.
- **S = carport, T = airports**
  - Longest common substring = **rport**
  - Longest common subsequence = **arport**
- Longest common substring may be found in  $O(|S|*|T|)$  time using dynamic programming.
- Longest common subsequence ? How much time is needed ?

117



Donald E. Knuth

conjectured in 1970 that ...

it is impossible to solve this longest common substring problem in  $O(|A|+|B|)$  time.

118

## Application: Longest Common Substrings

- Say we want to find the longest common substring of  $s_1, s_2, \dots, s_n$ 
  1. Build suffix tree for  $s_1, \dots, s_n$
  2. All nodes labeled  $\{s_{i_1}, \dots, s_{i_k}\}$  represent a match between  $s_{i_1}, \dots, s_{i_k}$
  3. Keep the substring length informations on these  $\{s_{i_1}, \dots, s_{i_k}\}$  match; find the largest values.

119

Acknowledgement:  
Adopted from Dr. Yaw-Ling Lin's slides

## The End

120