Suffix Trees and their applications

Amit Metodi and Tali Weinberger

Seminar 2012/1

Outline

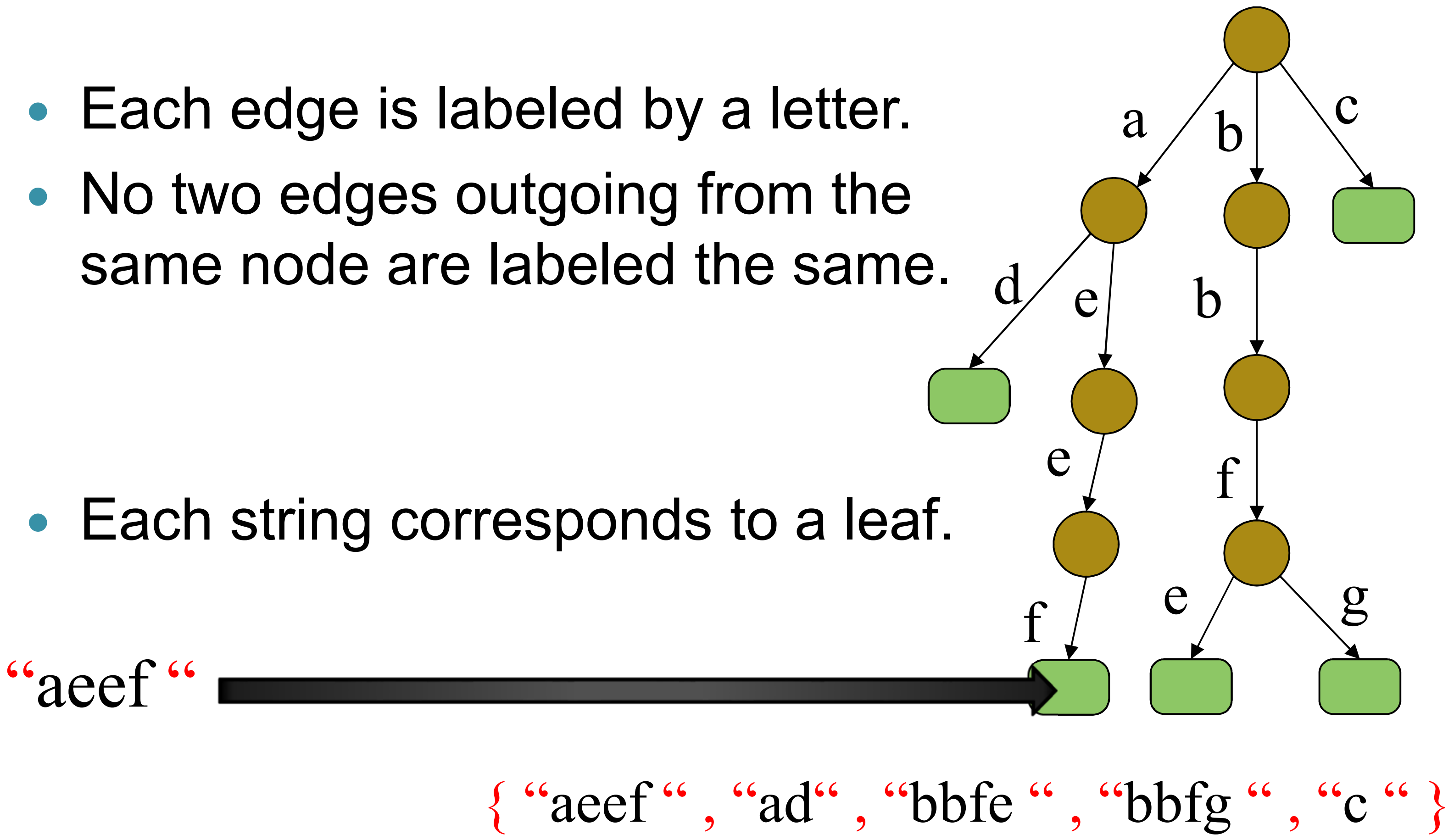
- Introduction to Suffix Trees
  - Trie and Compressed Trie
  - Suffix Tree
  - Trivial Construction Algorithm O(N^2)
  - Exact string matching
  - Generalized suffix tree
- Applications

Trie

- A tree representing a set of strings.
  (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.

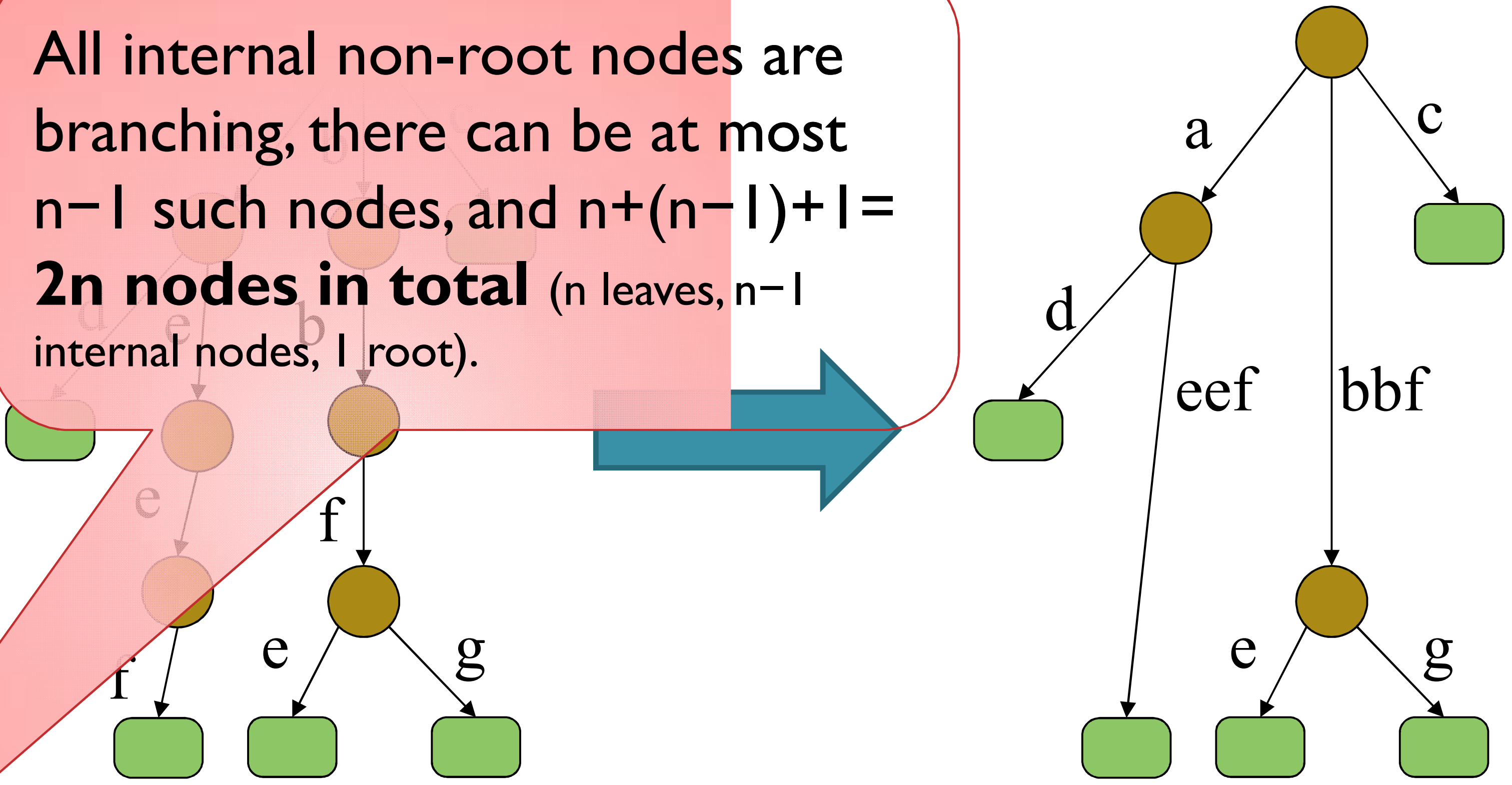
- Each string corresponds to a leaf.



Compressed Trie

- Compress unary nodes, label edges by strings.

All internal non-root nodes are branching, there can be at most n−1 such nodes, and n+(n−1)+1= **2n nodes in total** (n leaves, n−1 internal nodes, 1 root).



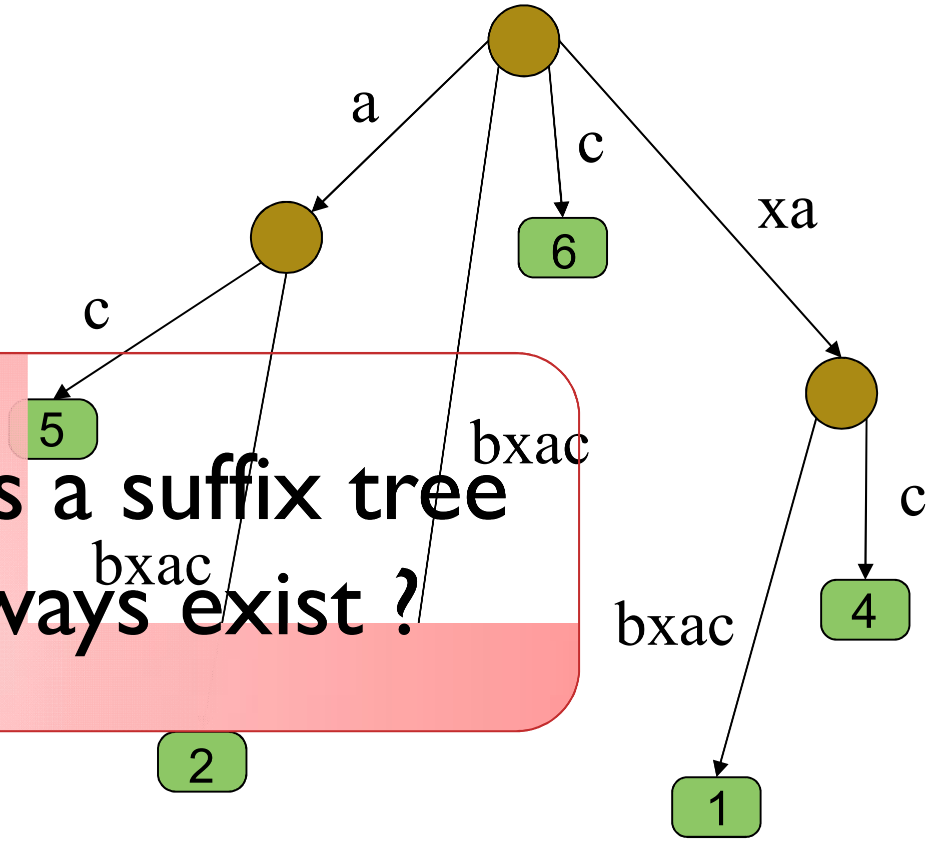
{ “aeef “ , “ad“ , “bbfe “ , “bbfg “ , “c “ }

Suffix Tree

- A suffix tree of string S[1..n] is a compressed trie of all suffixes of S.
- Denote S[i..n] by S_i

S = "x a b x a c"
1 2 3 4 5 6

{
S_6= c
S_5= ac
S_4= xac
S_3= bxac
S_2= abxac
S_1= xabxac
}



Does a suffix tree always exist ?

Suffix Tree

S = "x a b x a"
1 2 3 4 5

{

S_5= a

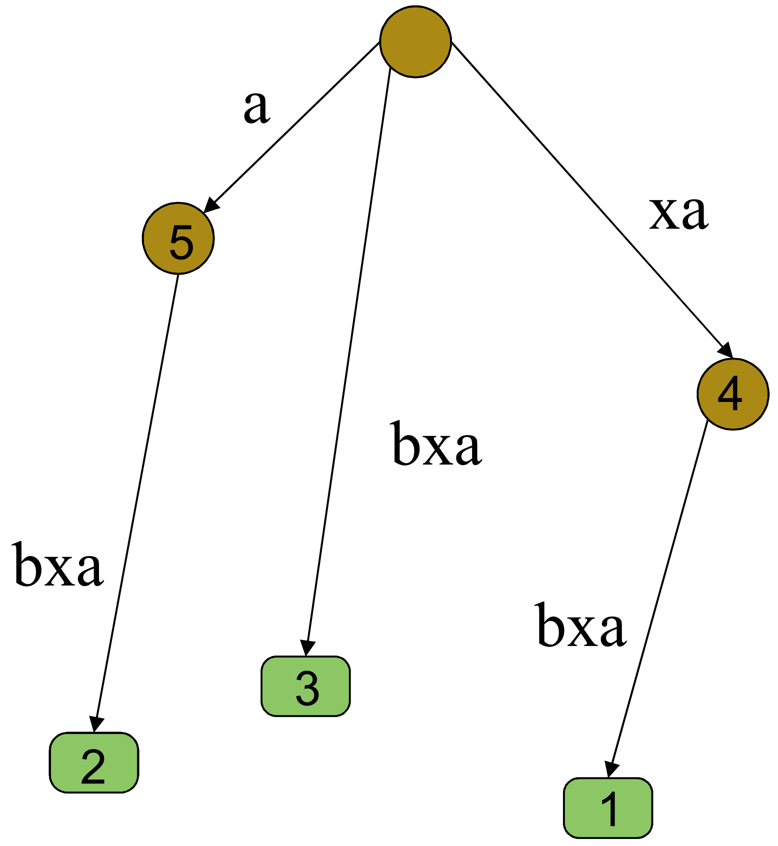
S_4= xa

S_3= bxa

S_2= abxa

S_1= xabxa

}



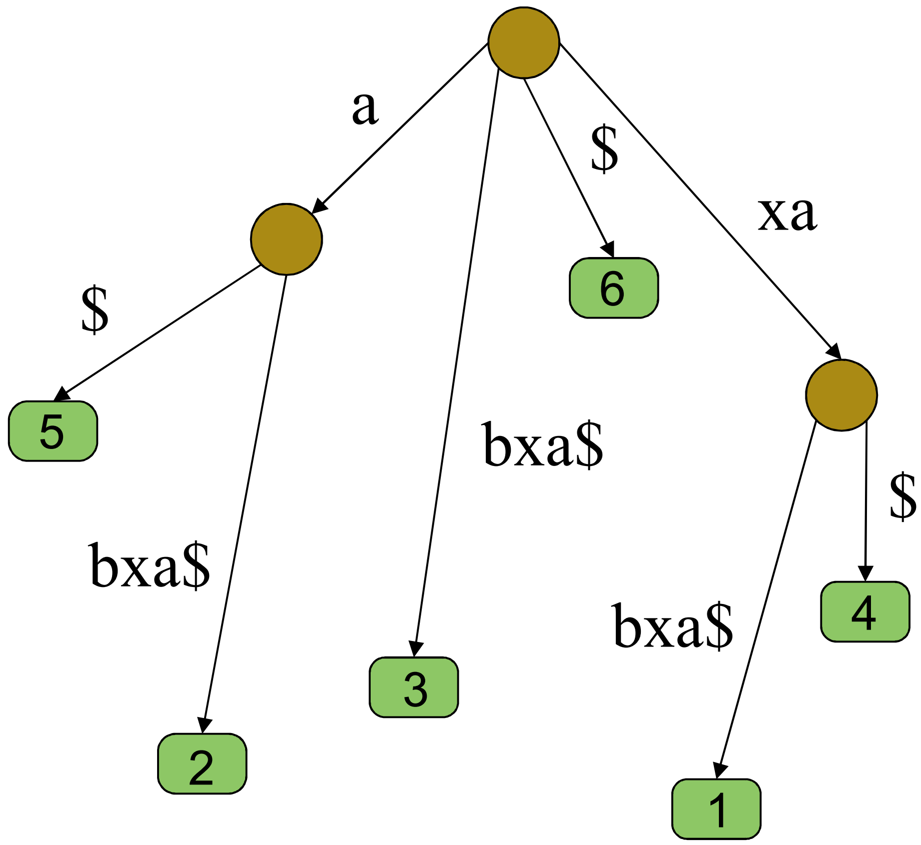
The fourth suffix *xa* or the fifth suffix *a* won't be represented by a leaf node.

Suffix Tree

- Solution: insert a special terminal character at the end such as $.
- Then, xa$ will not be a prefix of the suffix xabxa$.

S = "x a b x a $"
1 2 3 4 5 6

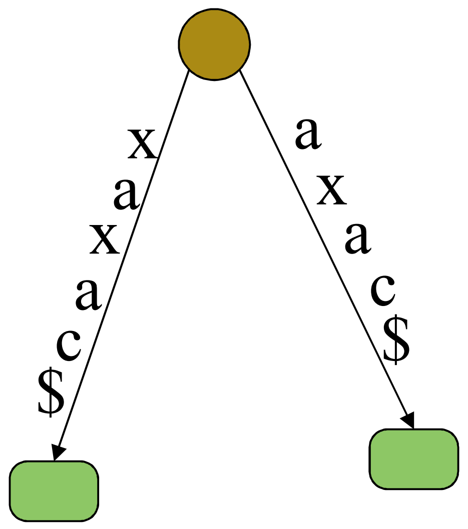
{
S_6= $
S_5= a$
S_4= xa$
S_3= bxa$
S_2= abxa$
S_1= xabxa$
}



Trivial algorithm to build Suffix tree

Build suffix tree S=xaxac (S'=xaxac$)

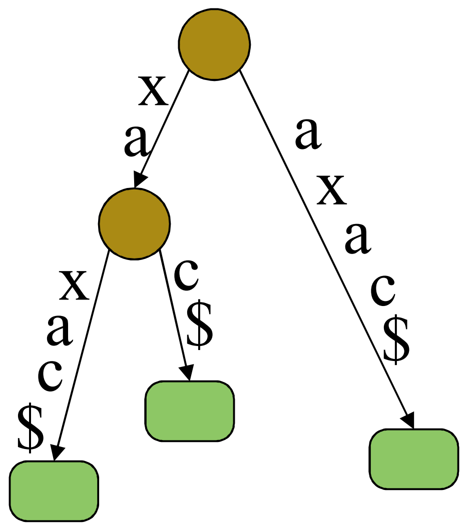
- Put the largest suffix xaxac$
- Put the suffix axac$
- Put the suffix xac$



Trivial algorithm to build Suffix tree

Build suffix tree S=xaxac (S'=xaxac$)

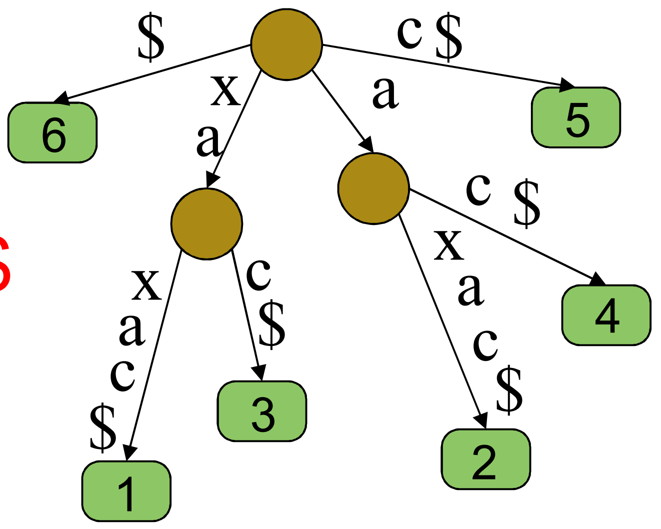
- Put the largest suffix xaxac$
- Put the suffix axac$
- Put the suffix xac$
- Put the suffix ac$



Trivial algorithm to build Suffix tree

Build suffix tree S=xaxac (S'=xaxac$)

- Put the largest suffix xaxac$
- Put the suffix axac$
- Put the suffix xac$
- Put the suffix ac$
- Put the suffix c$
- Put the smallest suffix $
- Label each leaf with the starting point.



Complexity – Run Time

- We need O(*n*-*i*+*1*) time for the ith suffix. Therefore the total running time is:

$$\sum_{1}^{n} O(i) = O(n^2)$$

- Ukkonen in 1995 provided the first online-construction of suffix trees with the running time that matched the then fastest algorithms. These algorithms are all linear-time for a constant-size alphabet, and have worst-case running time of *O*(*n*log*n*) in general.
- Martin Farach in 1997 gave the first suffix tree construction algorithm that is optimal for all alphabets O(*n*)

Complexity - Space

- Will also take $O(n^2)$ if we would store every suffix in the tree separately.

- Note that, we should not store the actual substrings $S[i \dots j]$ of S in the edges, but only their start and end indices (i, j).
- Nevertheless we keep thinking of the edge labels as substrings of S.

- This will reduce the space complexity to $O(n)$

Exact string matching

- Given S and P strings where |*S*|=*n* and |*P*|=*m*. Find all occurrences of *P* in *S*.

S=	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**
	1	**2**	3	4	**5**	6	7	8	9	10	11
P=	**i**	**i**	**i**	**i**	**i**	**s**	**s**	**i**	**s**	**s**	**i**

- Naïve algorithm = O(n*m)

Exact string matching

- Given S and P strings where |*S*|=*n* and |*P*|=*m*. Find all occurrences of *P* in *S*.

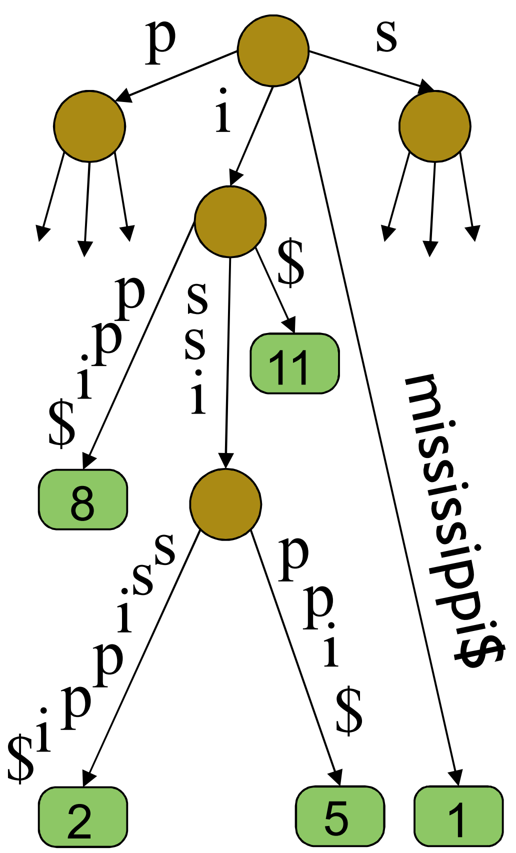
<u>Using suffix tree:</u>

1. Build suffix tree **O(n)**

2. Try to match P on a path. Three cases:
a. No match → P does not occur in T.
b. The match of P ends in a node u. Set x = u.
c. The match ends inside an edge (v,w). Set x=w.
O(m)

3. All leaves below x represent occurrences of P.
O(k) (where k = number of occurrences of P in S)

Total time: O(n+m+k) ~= **O(n)**



Generalized suffix tree

- Given a set of strings T a generalized suffix tree of T is a compressed trie of all suffixes of S $\in$ T.
- To make these suffixes prefix-free we add a special char at the end of S.
- To associate each suffix with a unique string in T, add a different special char to each S $\in$ T.

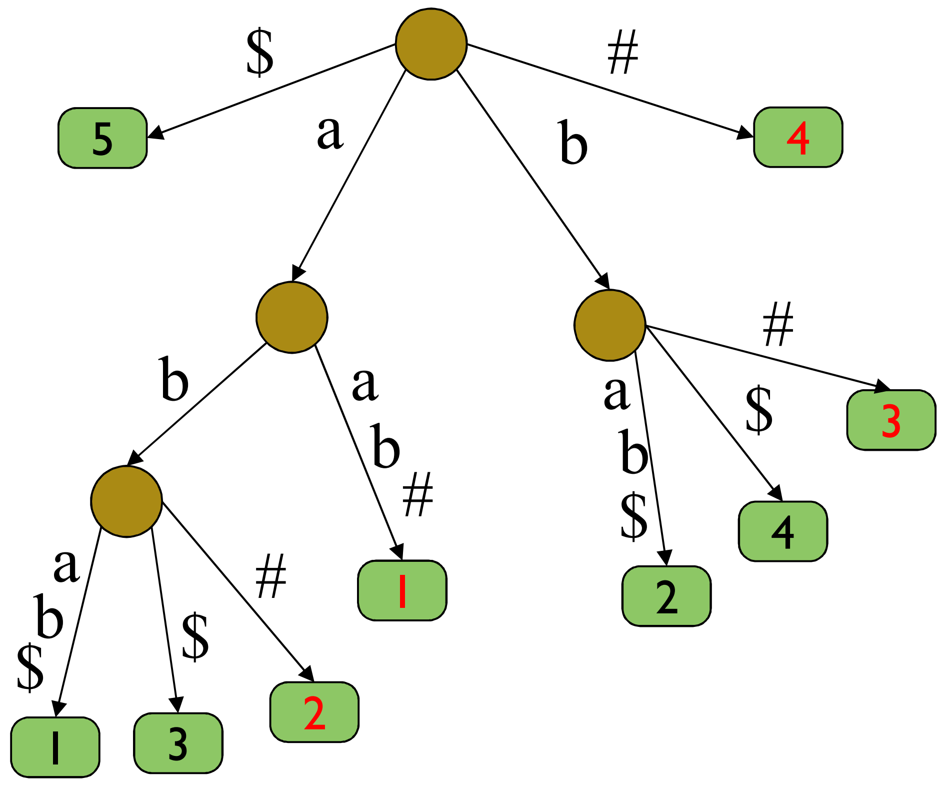
Generalized suffix tree

- 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$	

}



Applications of suffix trees

- Longest Common Substring
  - DNA Contamination Problem
- Maximal Repetitive Structures
- Longest common extension
- Finding maximal palindromes
- The k-mismatch problem

Longest Common Substring

- Given strings A and B find the longest substring common to both strings.

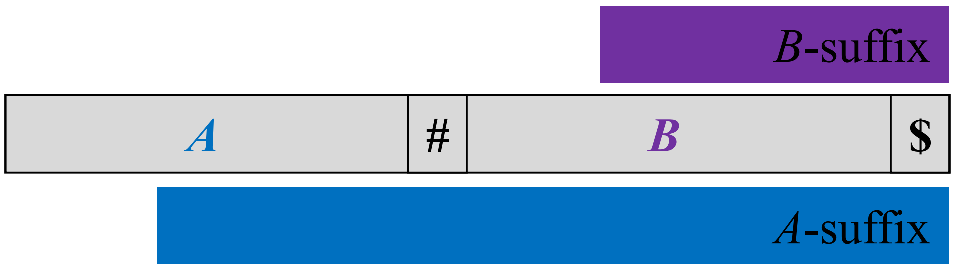
- String A= lam**bad**a
- String B=a**bad**y
- Longest Common Substring = bad

Donald E. Kn
conjectured in 1970 that

It is impossible
Longest Comm
problem in O(|

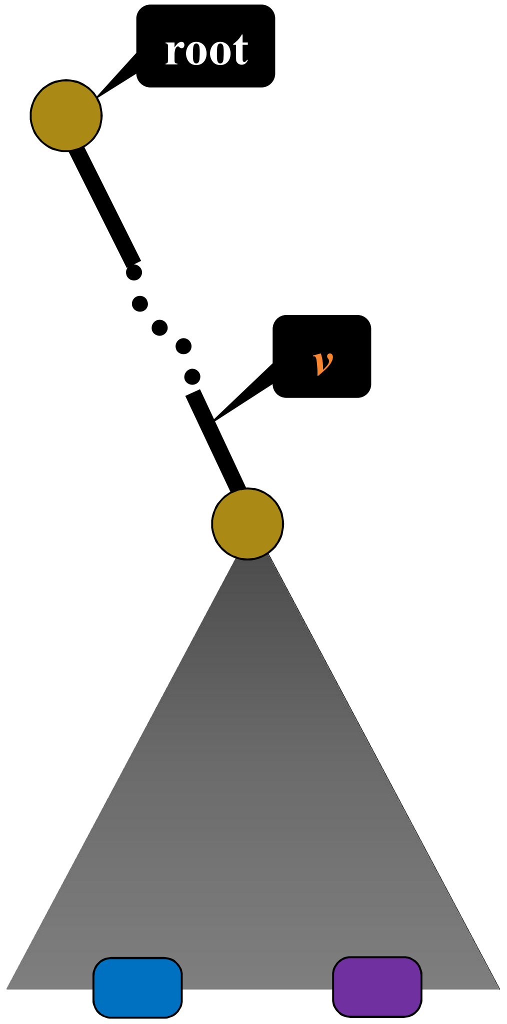
LCSubstring - Idea

- Construct a suffix tree T for $A\#B\$$, where # and \$ are two characters not in A and B.
- There are exactly $|A|+|B|+2$ leaves in T, each leaf corresponds to a suffix of $A\#B\$$.
  - A-leaf: with label in $\{1, 2, \ldots, |A|\}$
    - corresponds to an A-suffix.
  - B-leaf: with label in $\{|A|+2, \ldots, |A|+|B|+1\}$
    - corresponds to a B-suffix.



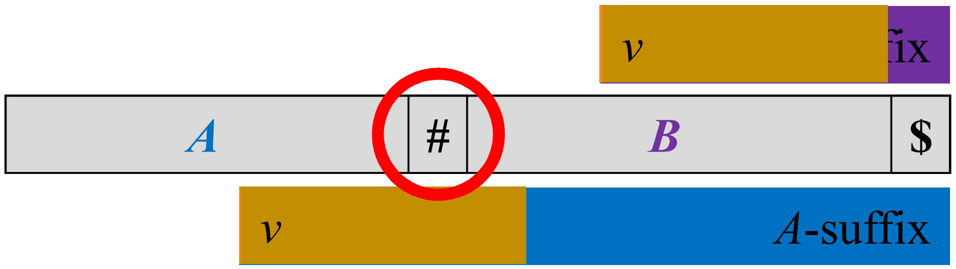
LCSubstring - Observation

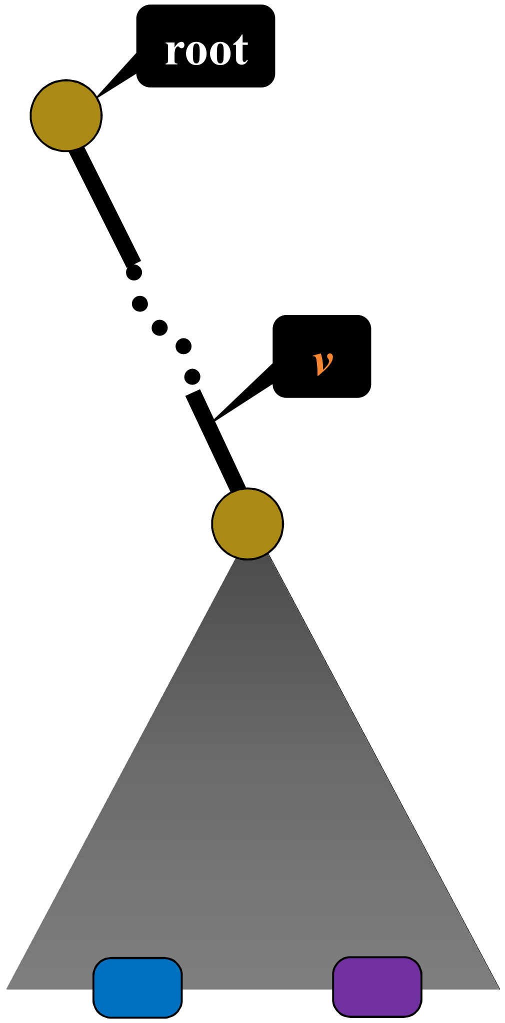
- Let *v* be an arbitrary position of *T* (i.e., *v* is not necessarily a node of *T*.)
  - *v* has a descendant *A*-leaf if and only if *v* corresponds to a prefix of an *A*-suffix of *A*#*B*$.
  - *v* has a descendant *B*-leaf if and only if *v* corresponds to a prefix of a *B*-suffix of *A*#*B*$.



LCSubstring - Lemma

- Let *v* be a position of *T*. *v* has descendant *A*-leaf and *B*-leaf if and only if *v* corresponds to a common substring of *A* and *B*.





LCSubstring – Algorithm

- Construct a suffix tree *T* for *A*#*B*$. **O(|A|+|B|)**
- Marking the colors of each node, including each leaf and each internal nodes. **O(|A|+|B|)**
- Computing the depths of all nodes. **O(|A|+|B|)**
- Find a deepest internal node with both colors. **O(|A|+|B|)**

Single DFS

- Output the string corresponding to the deepest internal node *v* such that the subtree of *T* rooted at *v* contains both *A*-leaf and *B*-leaf.
- **Time: O(|A|+|B|)**
- **Space: O(|A|+|B|)**

Space can be reduced to **O(|A|)**

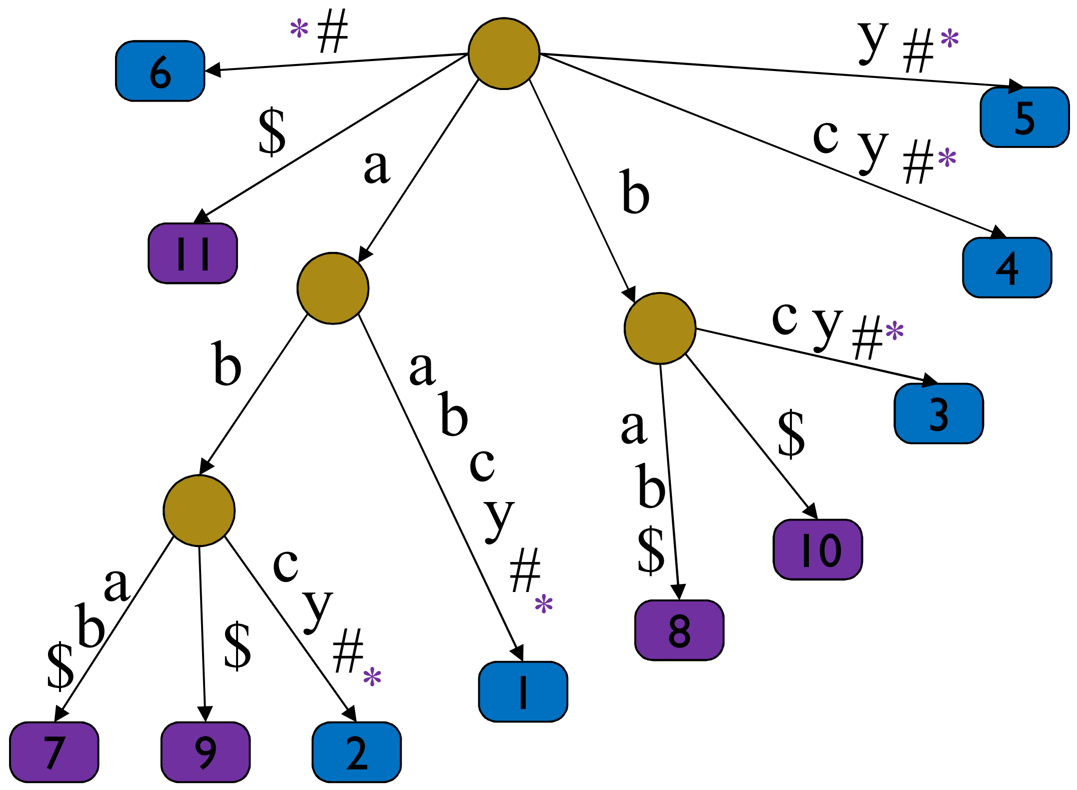
LCSubstring - Example

- Let A=aabcy and B=abab, here is a generalized suffix tree for A and B.

{

$	#*
b$	y#*
ab$	cy#*
bab$	bcy#*
abab$	abcy#*
	aabcy#*

}



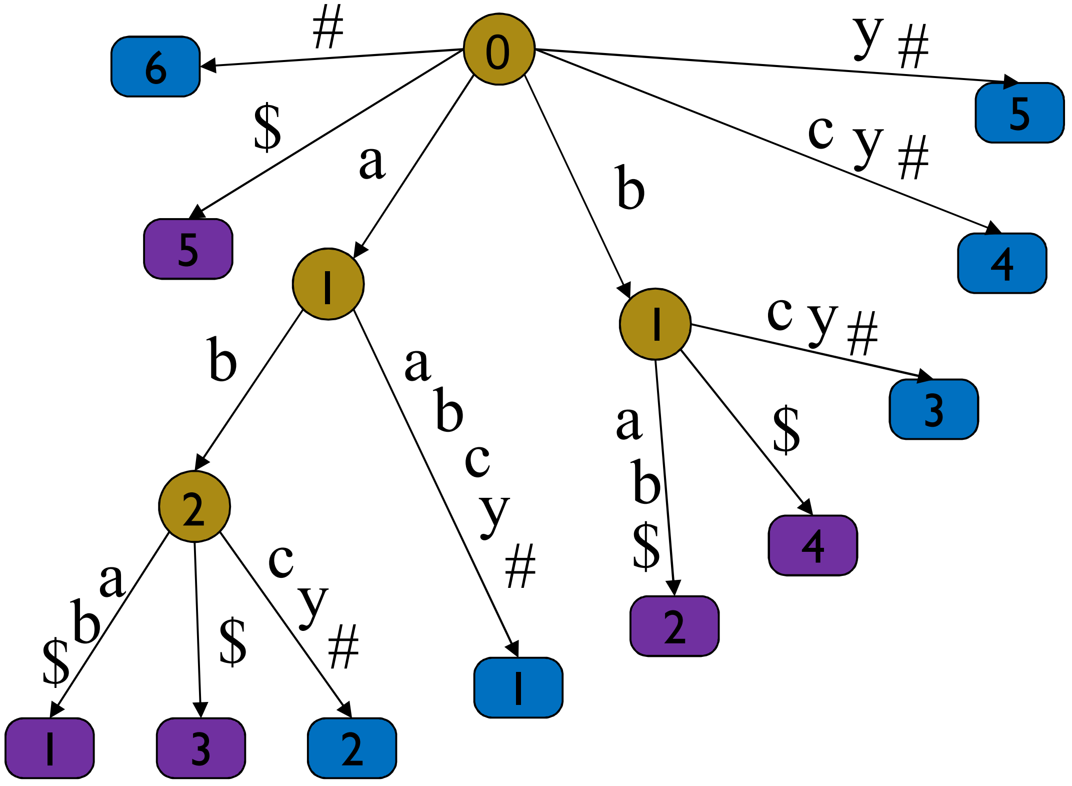
LCS$_{\text{ubstring}}$ - Example

- Let A=aabcy and B=abab, here is a generalized suffix tree for A and B.

{

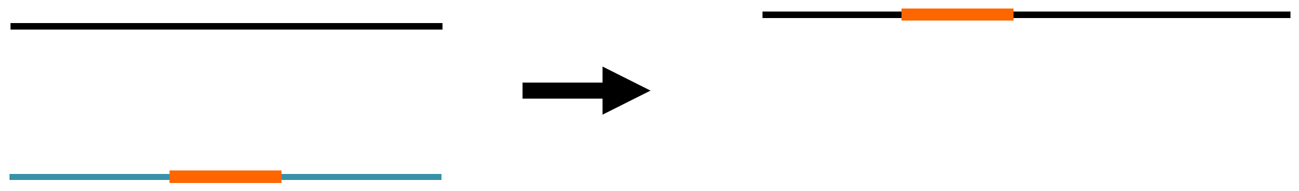
$	#
b$	y#
ab$	cy#
bab$	bcy#
abab$	abcy#
	aabcy#

}



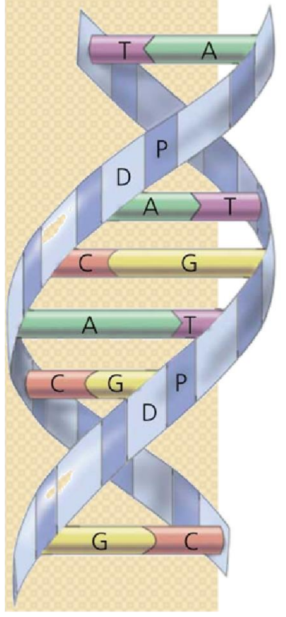
DNA Contamination Problem

DNA contamination: During laboratory processes, unwanted DNA inserted into the DNA of interest.



Contamination sources: Human, bacteria,…

DNA from Dinosaur bone: More similar to human DNA than to bird and crockodilian DNA



DNA Contamination Problem

S: DNA of interest

P: DNA of possible contamination source

If S and P share a common substring longer than *l*, then S has been contaminated by P.

To find all common substrings of S and P that are longer than *l* .

In general, P is set of DNA that are potential contamination sources.

Applications of suffix trees

- Longest Common Substring
  - DNA Contamination Problem
- Maximal Repetitive Structures
- Longest common extension
- Finding maximal palindromes
- The k-mismatch problem

Maximal Repetitive Structures

Maximal Pair

- A maximal pair in string S:

  A pair of identical substrings α and β in S s.t. the characters to the immediate left and right of α is different from the characters to the immediate left and right of β, respectively.

- That is, Extending α and β in either direction would destroy the equality of the two strings.

- Example: S = xabcyiiizabcqabcyrxar

Maximal Pair (continued)

- Overlap is allowed:

  S = cxxaxxaxxb

  cxxaxxa

  axxaxxb

- To allow a prefix or suffix of S to be part of a maximal pair:

  S → #S$ (#,$ don't appear in S).

  Example: #abcxabc$

Maximal Repeat

- A maximal repeat in string S:

  A substring of S that occurs in a maximal pair in S.

- Example: S = xabcyiiizabcqabcyrxar

  maximal repeats: abc, abcy, ...

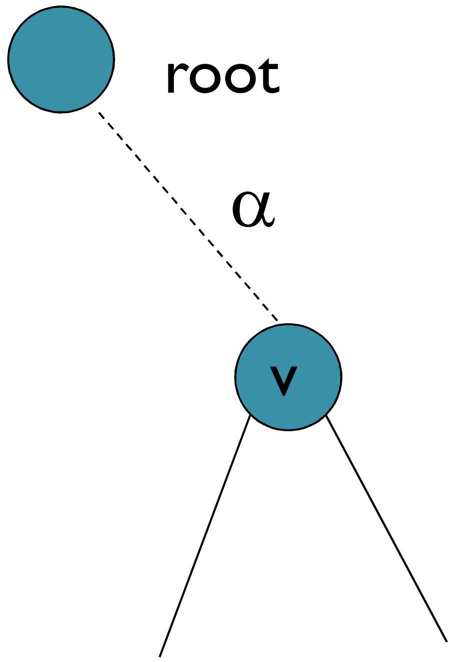
Finding All Maximal Repeats In Linear Time

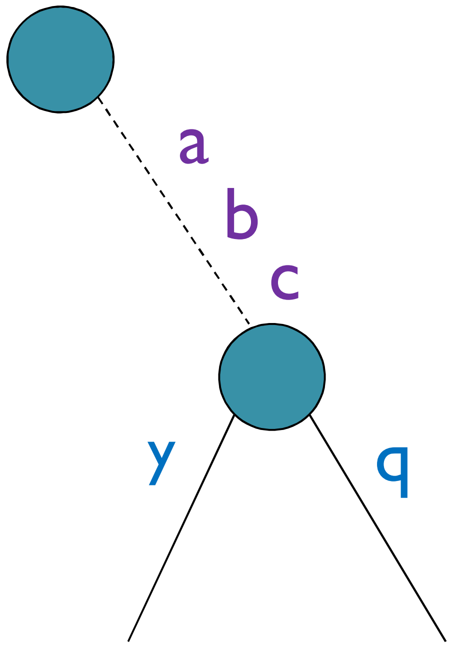
- Given: String S of length n.
- Goal: Find all maximal repeats in O(n) time.
- Lemma:

  Let T be a suffix tree for S.
  If string α is a maximal repeat in S,
  then α is the path-label of an internal node v in T.

Proof – by def. of maximal repeat

S = xabcyiiizabcqabcyrxar





A <u>maximal repeat</u> in string S:

A substring of S that occurs in a maximal pair in S.

Observation

- T has at most n internal nodes.

- <u>Why?</u>

  Since T has n leaves (one for each index), and each internal node other than the root must have at least two children, T can have at most n internal nodes.

Conclusion

- There can be at most n maximal repeats in any string of length n.

- <u>Proof</u>:

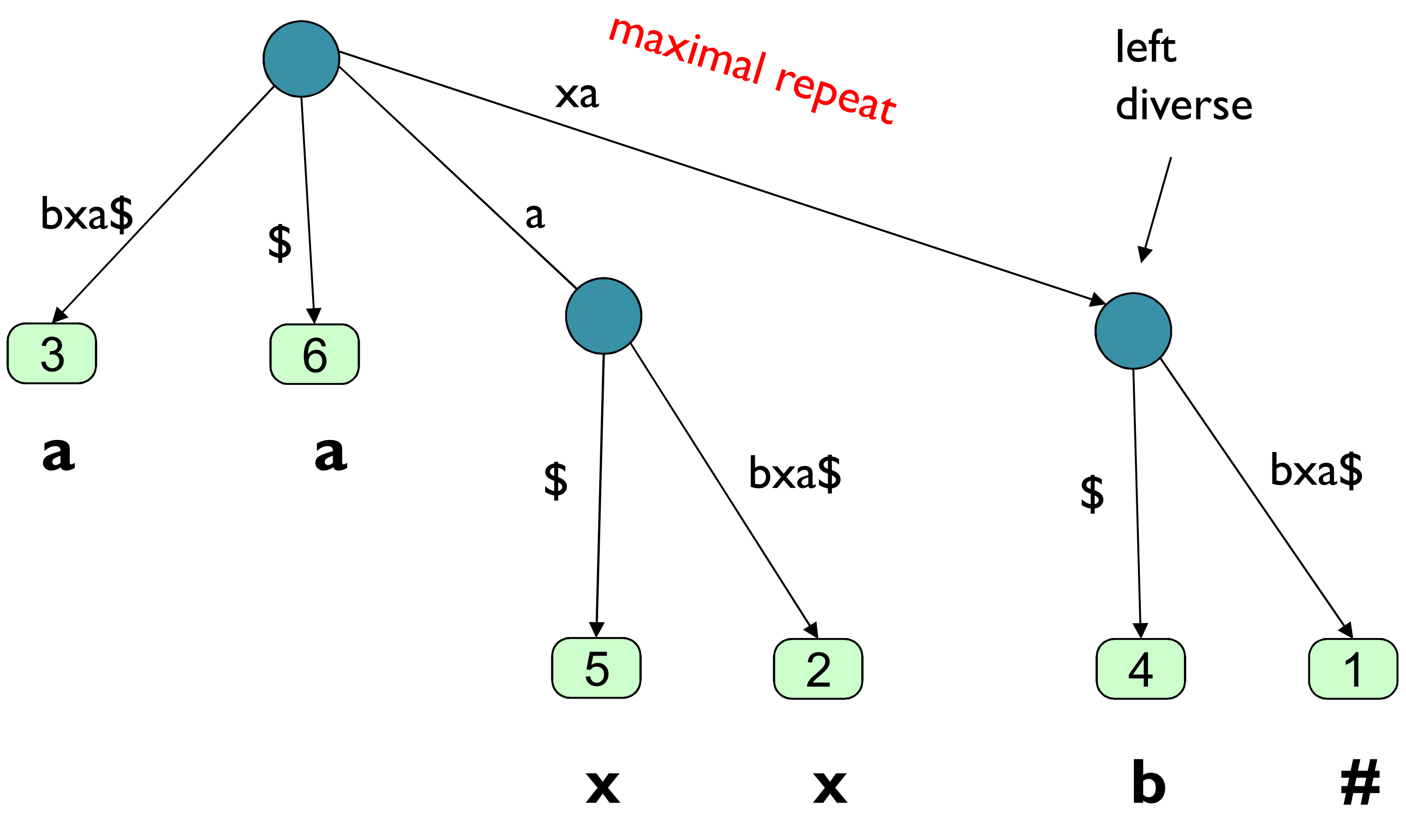
  by the lemma, since T has at most n internal nodes.

Which internal nodes correspond to maximal repeats?

- The <u>left character</u> of leaf i in T is S(i-1).
- Node v of T is called <u>left diverse</u> if at least 2 leaves in v’s subtree have different left characters.
- A leaf can’t be left diverse.
- Left diversity propagates upward.

Example: S = #xabxa$

1 2 3 4 5 6



Theorem

The string α labeling the path to an internal node v of T is a maximal repeat

$\Leftrightarrow$

v is left diverse.

Proof of $\Rightarrow$

- Suppose α is a maximal repeat $\rightarrow$
- It participates in a maximal pair $\rightarrow$
- It has at least two occurrences with distinct left characters: $x\alpha$, $y\alpha$, $x \neq y$ $\rightarrow$
- Let i and j be the two starting positions of α.
  Then leaves i and j are in v's subtree and have different left characters x,y. $\rightarrow$
- v is left diverse.

Proof of $\Leftarrow$

- Suppose v is left diverse $\rightarrow$ there are substrings $x\alpha p$ and $y\alpha q$ in S, $x \neq y$.

- If $p \neq q$ $\rightarrow$ α's occurrences in $x\alpha p$ and $y\alpha q$ form a maximal pair $\rightarrow$ α is a maximal repeat.

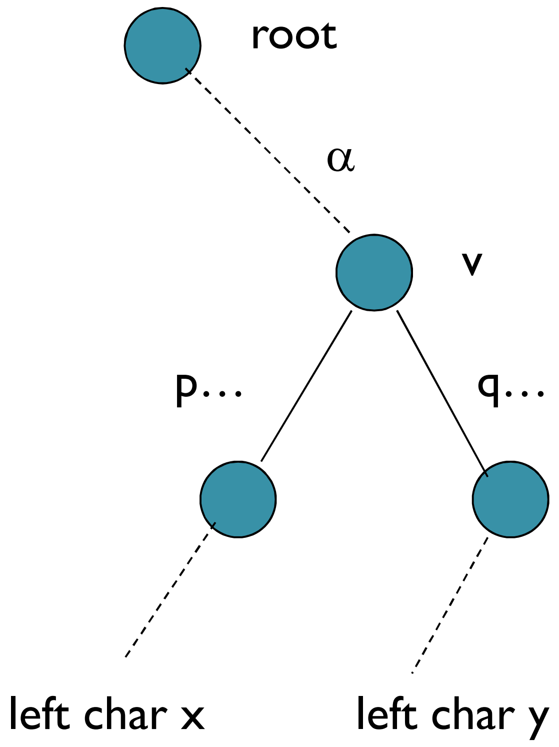
- If $p=q$ $\rightarrow$ since v is a branching node, there is a substring $z\alpha r$ in S, $r \neq p$.

  If $z \neq x$ $\rightarrow$ It forms a maximal pair with $x\alpha p$.
  If $z \neq y$ $\rightarrow$ It forms a maximal pair with $y\alpha p$.
  In either case, α is a maximal repeat.

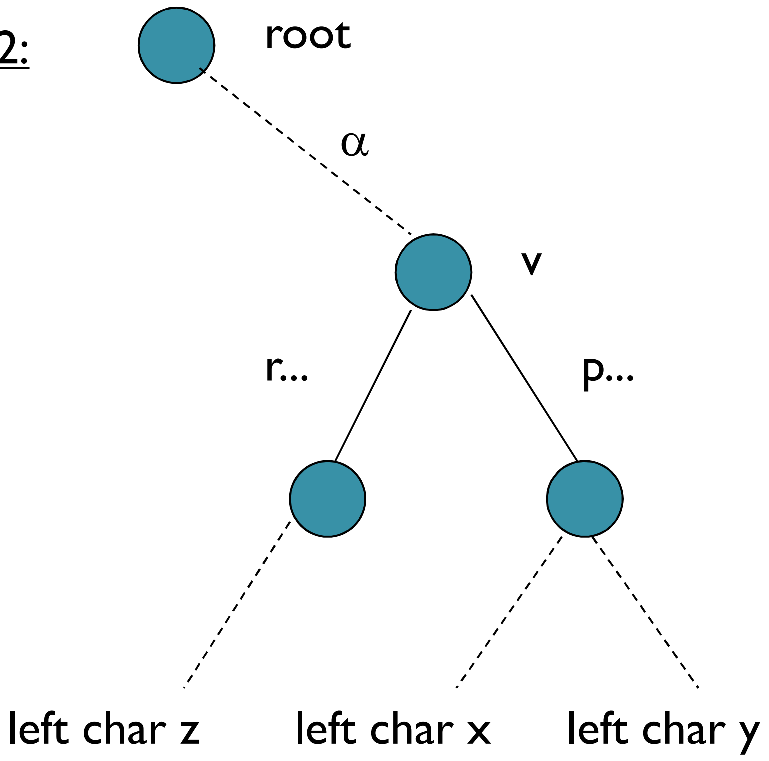
These cases cover all the cases, since $x \neq y$.

Proof of ⇐ (continued)

<u>Case 1:</u>



<u>Case 2:</u>



Compact Representation

- Node v in T is a <u>frontier node</u> if:
  - v is left diverse.
  - none of v's children are left diverse.

- Each node at or above the frontier is left diverse.

- The subtree of T from the root down to the frontier nodes is the <u>compact representation</u> of the set of all maximal repeats of S.

- Representation in O(n) though total length of all maximal repeats may be larger.

Linear time algorithm

- Build suffix tree T.
- Find all left diverse nodes in linear time.
- Delete all nodes that aren’t left diverse, to achieve the compact representation.

finding all left diverse nodes in linear time

- Traverse T bottom-up, recording for each node:
  - either that it is left diverse
  - or the left character common to all leaves in its subtree.
- For each leaf: record its left character.
- For each internal node v:
  - If any child is left diverse $\rightarrow$ v is left diverse.
  - Else If all children have a common character x $\rightarrow$ record x for v.
    - Else record that v is left diverse.

Time Analysis

- Suffix tree construction $\rightarrow$ O(n).
- Bottom-up traversal $\rightarrow$ O(n).
- Total O(n).

Applications of suffix trees

- Longest Common Substring
  - DNA Contamination Problem
- Maximal Repetitive Structures
- Longest common extension
- Finding maximal palindromes
- The k-mismatch problem

Longest common extension

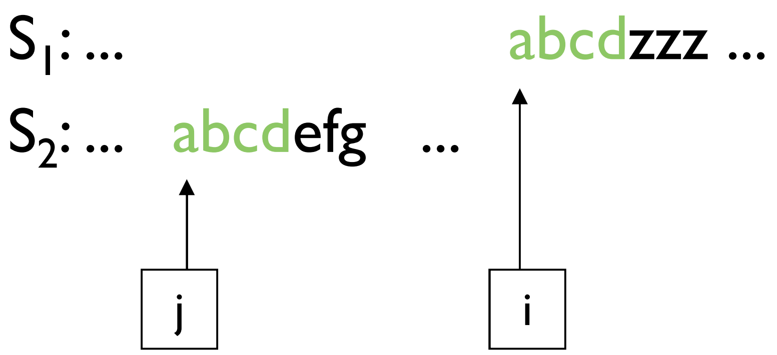
Longest common extension:

a bridge to inexact matching

Longest common extension problem

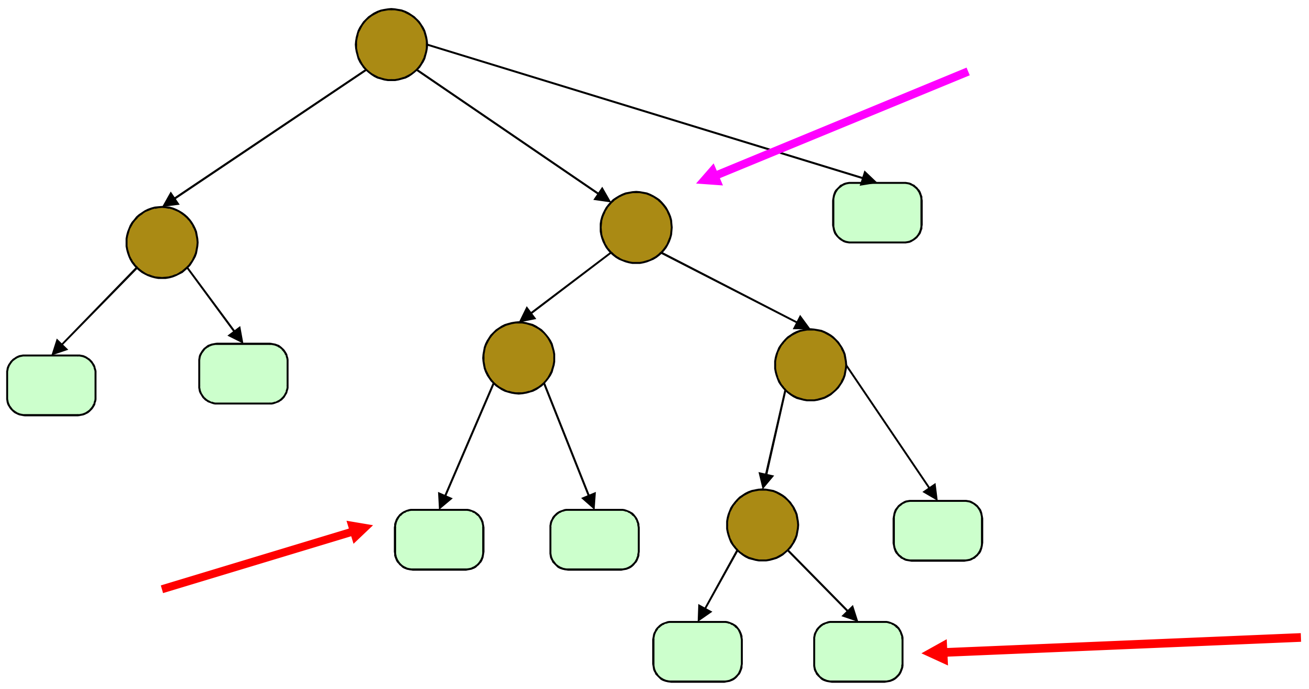
Preprocess strings S_1 and S_2 s.t. the following queries can be computed in O(1) time each:

- Given index pair (i,j), find the length of the longest substring of S_1 starting at position i that matches a substring of S_2 starting at position j.



Lowest common ancestors

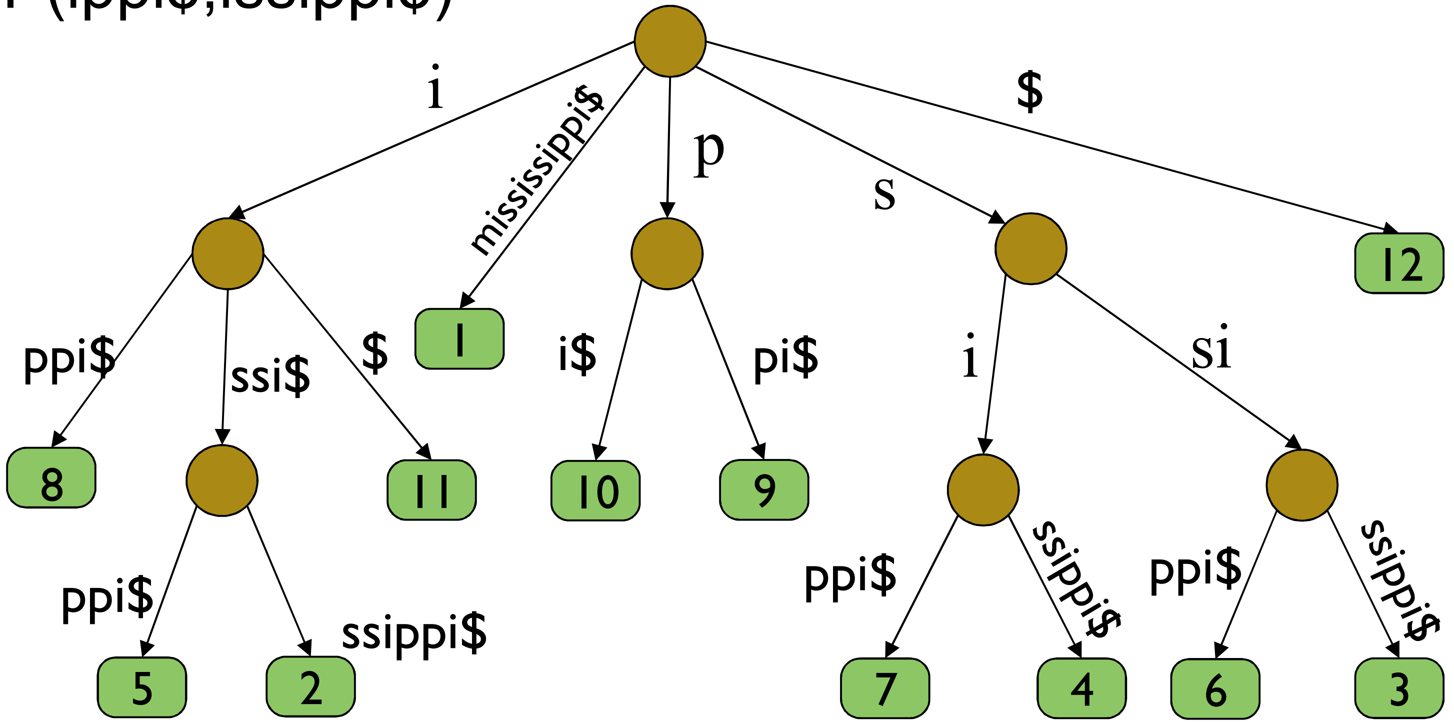
A lot more can be gained from a suffix tree if we preprocess it so that we can answer LCA queries on it



Why to find LCA?

For two suffixes of S, we can compute their Longest Common Prefix by finding the LCA of the corresponding leaves in the suffix tree.

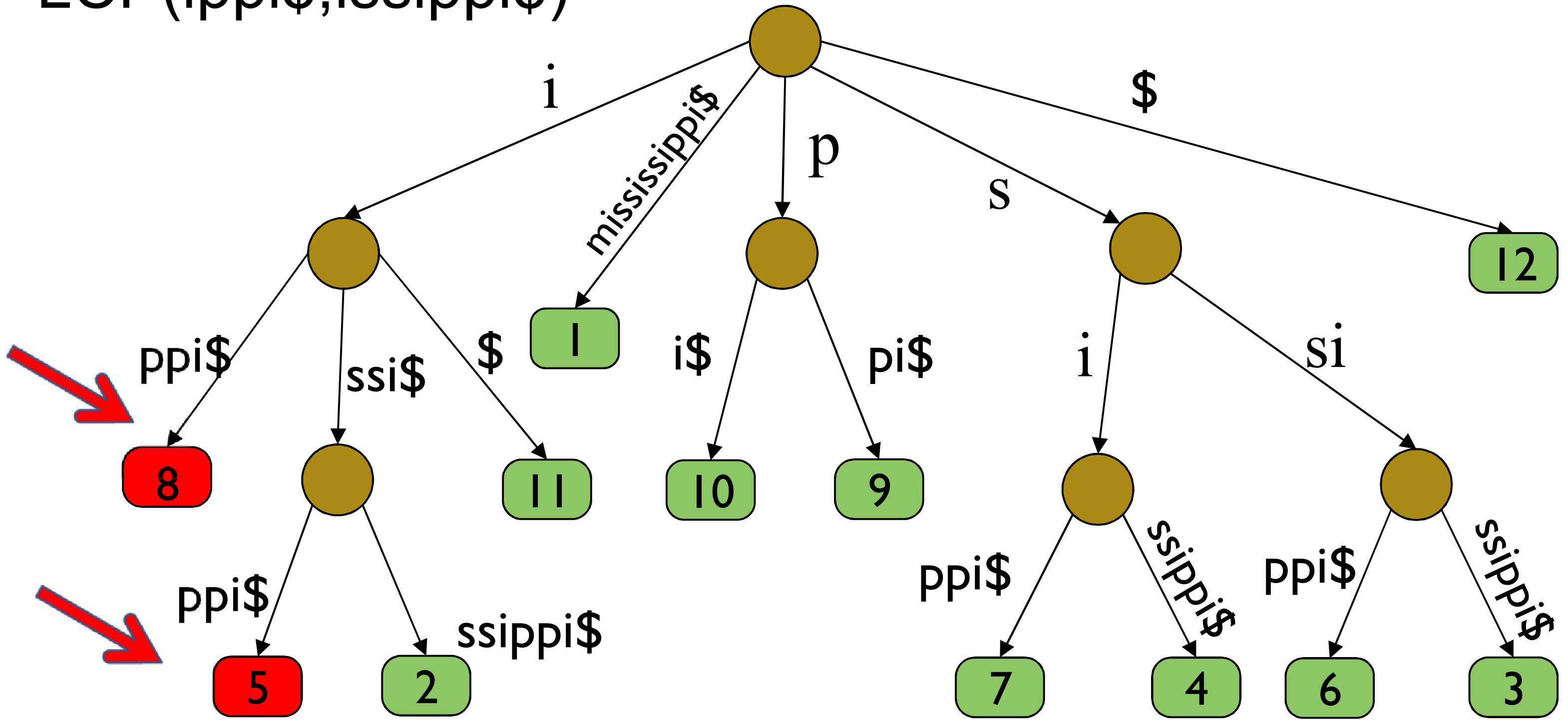
LCP(ippi$,issippi$)=



Why to find LCA?

For two suffixes of S, we can compute their Longest Common Prefix by finding the LCA of the corresponding leaves in the suffix tree.

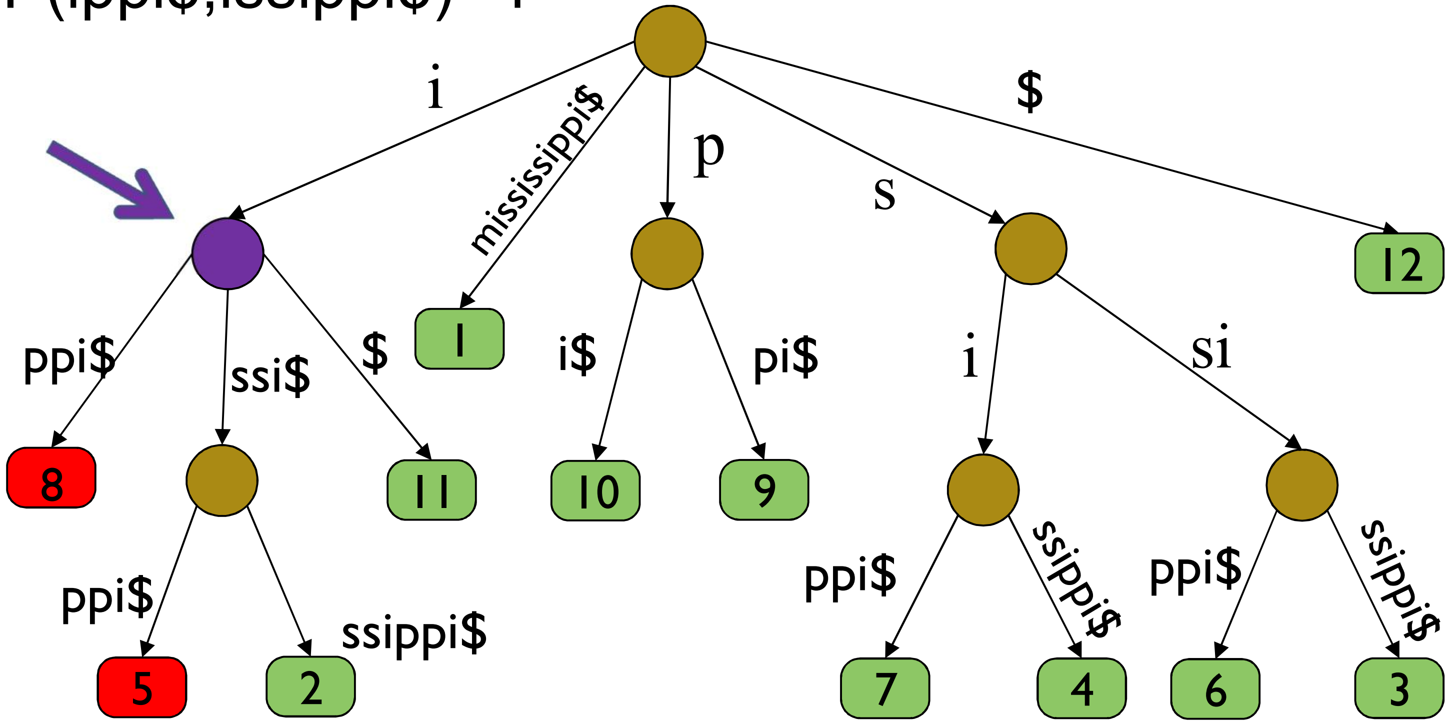
LCP(ippi$,issippi$)=



Why to find LCA?

For two suffixes of S, we can compute their Longest Common Prefix by finding the LCA of the corresponding leaves in the suffix tree.

LCP(ippi$,issippi$)= i



Lowest common ancestors

after a linear amount of preprocessing of a rooted tree, for any two specified nodes, their lowest common ancestor can be found in a constant time, independent of n.

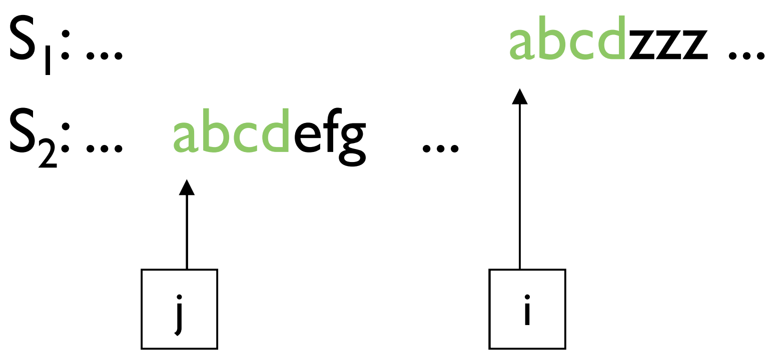
The lca result was first obtained by Harel and Tarjan:
Harel, Dov; Tarjan, Robert E. (1984), "Fast algorithms for finding nearest common ancestors", *SIAM Journal on Computing* **13**.

and later simplified by Schieber and Vishkin:
Schieber, Baruch; Vishkin, Uzi (1988), "On finding lowest common ancestors: simplification and parallelization", *SIAM Journal on Computing* **17**.

Longest common extension problem

Preprocess strings S_1 and S_2 s.t. the following queries can be computed in O(1) time each:

- Given index pair (i,j), find the length of the longest substring of S_1 starting at position i that matches a substring of S_2 starting at position j.



Longest common extension - Solution

Preprocess: $O(|S_1|+|S_2|)$

- Build generalized suffix tree T for S_1 and S_2.
- Preprocess T for constant-time LCA queries.
- Compute string-depth of every node.

To answer query (i,j): $O(1)$

- Find LCA node v of leaves corresponding to suffix i of S_1 and suffix j of S_2.
- Return string-depth(v).

Applications of suffix trees

- Longest Common Substring
  - DNA Contamination Problem
- Maximal Repetitive Structures
- Longest common extension
- Finding maximal palindromes
- The k-mismatch problem

Finding maximal palindromes

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

Let S = cbaaba

S^r - the reverse of string S

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

Example: S = cbaaba$ and S^r = abaabc#

$\boxed{i + 1}$ $\boxed{m - i + 1}$

Maximal palindromes algorithm

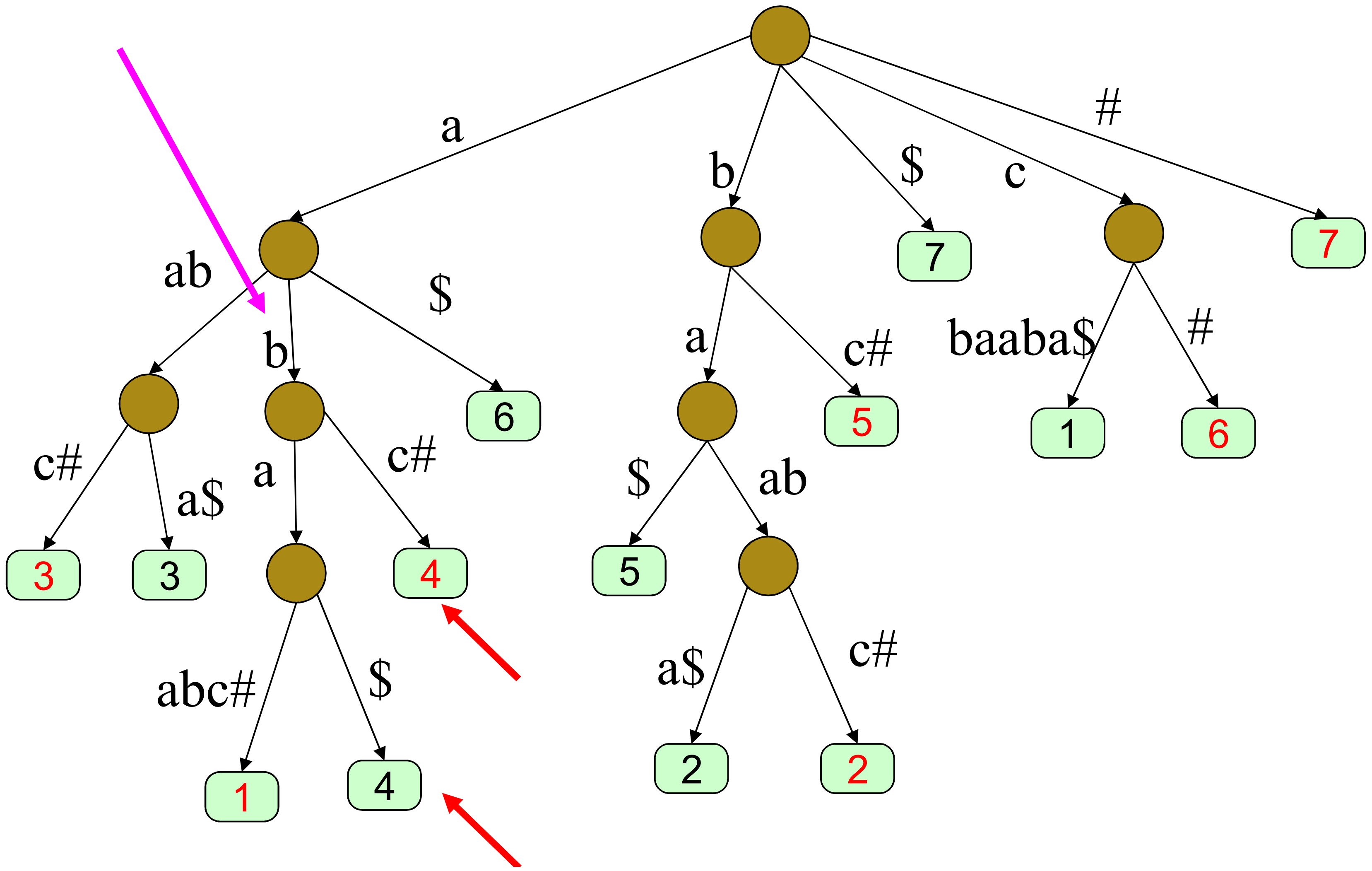
Prepare a generalized suffix tree for

S = cbaaba$ and S^r = abaabc#

<u>Preprocess</u>: O(n)
Build generalized suffix tree T for S and S^r.
Preprocess T for constant-time longest common extension.

For every i find the LCA of suffix i of S and suffix m-i+1 of S^r -> solve the longest common extension for (i+1, m-i+1)
If the extension has nonzero length k, then there is a maximal palindrom of radius k center at i

Let s = cbaaba$ then s^r = abaabc#



Applications of suffix trees

- Longest Common Substring
  - DNA Contamination Problem
- Maximal Repetitive Structures
- Longest common extension
- Finding maximal palindromes
- The k-mismatch problem

The k-mismatch problem

- Given: pattern P, text T, fixed number k.
- k-mismatch of P: a |P|-length substring of T that matches at least |P|-k characters of P
(i.e. it matches P with at most k mismatches).
- The k-mismatch problem:
Find all k-mismatches of P in T.

Example

P = bend

T = abentbananaend

k = 2

$\Rightarrow$T contains three 2-mismatches of P:

a b e n t b a n a n a e n d

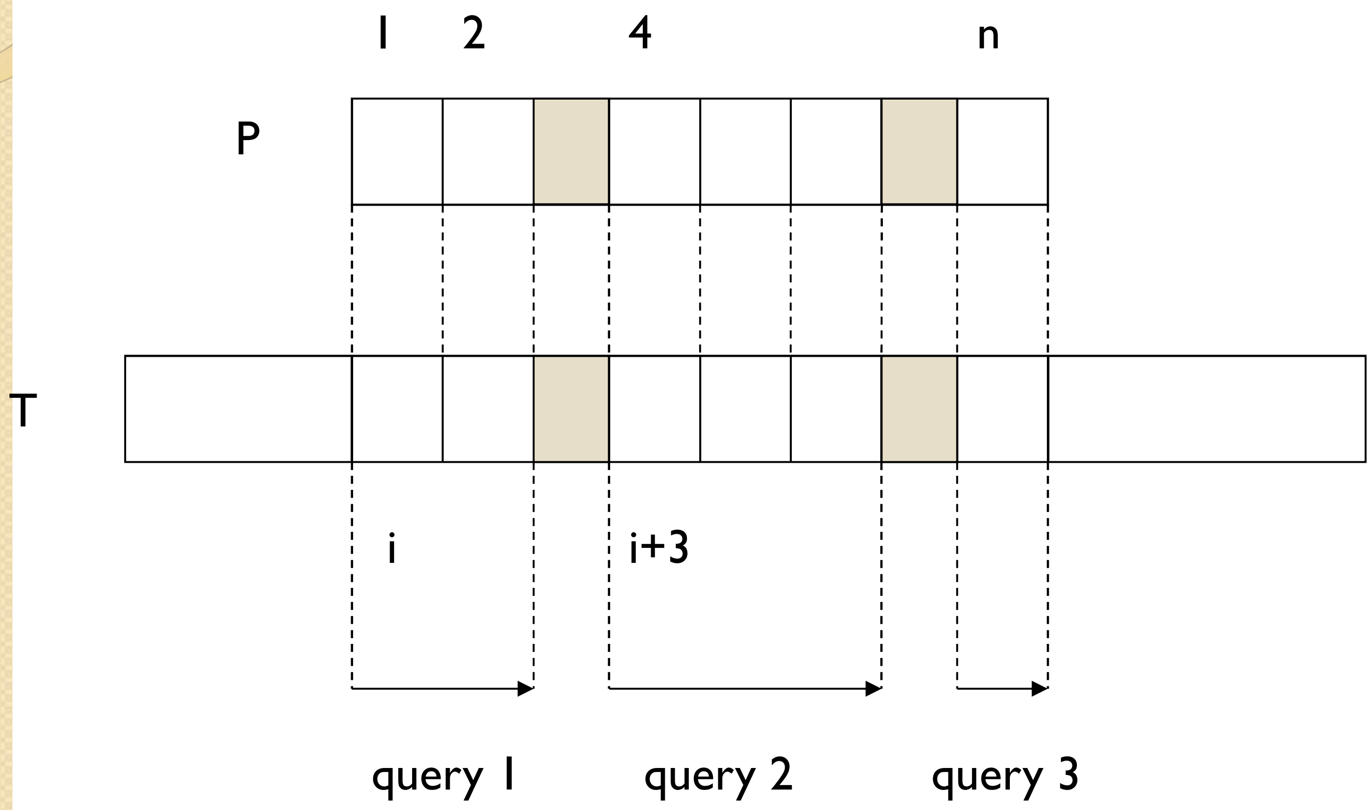
b e n d b e n d b e n d

1-mismatch 2-mismatch 1-mismatch

Solution

- Notation: |P|=m, |T|=n, k independent of n and m (k<<m).
- General idea:
  - For each position i in T, determine whether a k-mismatch of P begins at position i.
  - To do this efficiently: successively execute up to k+1 longest common extension queries.
  - A k-mismatch of P begins at position i if these extensions reach the end of P.

solution (continued)



Algorithm for index i

```
1.  j ← 1
    i' ← i
    count ← 0
2.  Compute the length l of the longest common extension starting
    at positions j of P and i' of T.
3.  if j+l=m+1
    then a k-mismatch of P occurs in T starting at i; stop.
4.  if count<k
    then count ← count+1
         j ← j+l+1
         i' ← i'+l+1
         go to step 2.
    else, a k-mismatch of P does not occur in T starting at i; stop.
```

Example

$P = \quad abcaabaccc$

$T = cabcdabbcccd$

Example

$P = \quad abcaabaccc$

$T = cabcdabbcccd$

$j_1 = 3$

Example

$P = \quad abcaabaccc$

$T = cabcdabbcccd$

$j_1 = 3,\ j_2 = 2$

Example

$P = \quad abcaabaccc$

$T = cabcdabbcccd$

$j_1 = 3,\ j_2 = 2,\ j_3 = 3$

Time Analysis

- Preprocessing of T and P for longest common extension queries $\rightarrow$ O(n).

- For each index i=1,...,n-m+1 of T, up to k+1 longest common extension queries $\rightarrow$
  O(k) per index $\rightarrow$ O(kn) total.

- Total O(kn) time.