Lec19-suffix-trees-arrays

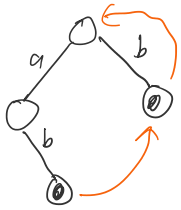
Tuesday, October 24, 2023 3:39 PM

Last time we learned about suffix tries & trees.
Let's remind ourselves of suffix trie construction.
We build it up while walking along the string from left to right.

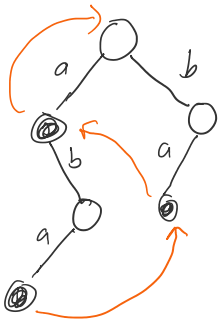
s = abaaba$

s = a

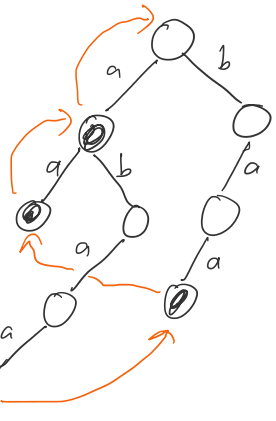
s = ab



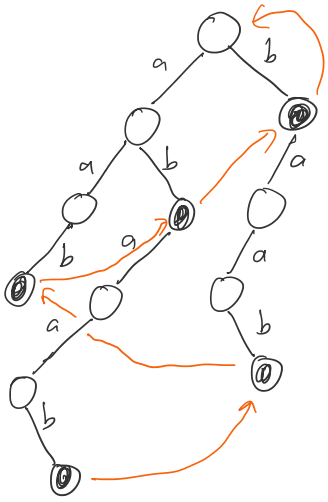
s = aba



s = abaa

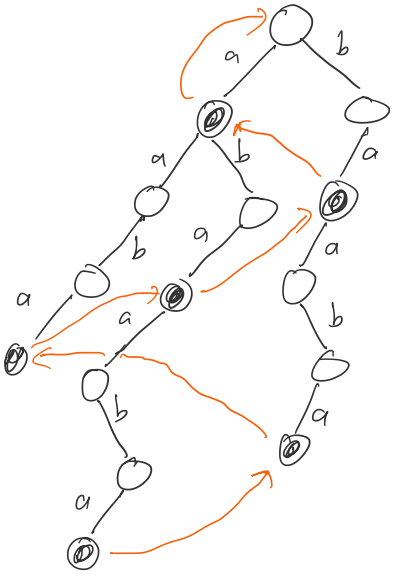


s = abaab

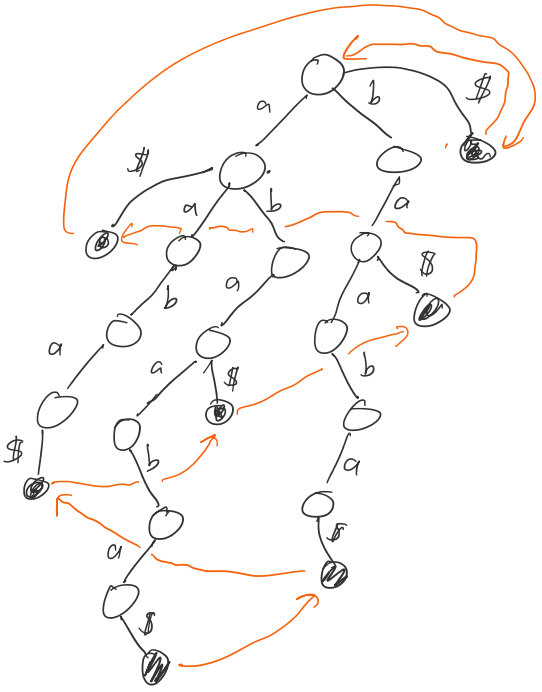


Notice that once you reach a node that already has b, all remaining ones are also already done, by the suffix link property

s = abaaba



s = abaaba$



Longest Common substring q & s

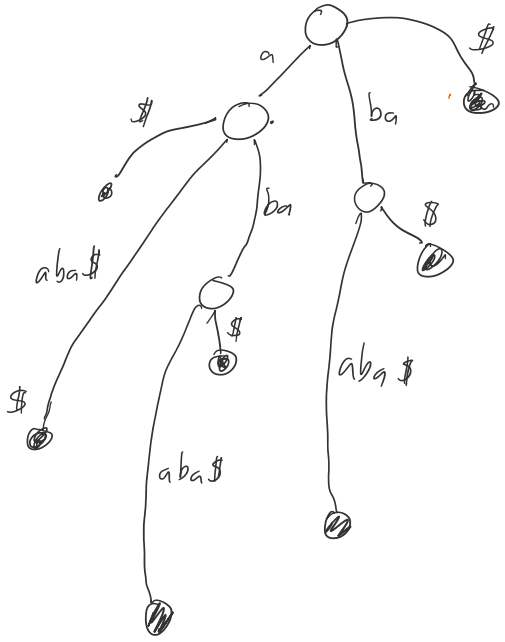
Walk down q as far as possible.
If dead end, save depth, follow suffix link & keep walking.
When q exhausted, return longest substring found.

abaaba$

Ex bbaa

Start at b.
Dead end.
Start at ∅ → b → a → a
Done.

We can also compress it to a suffix tree:

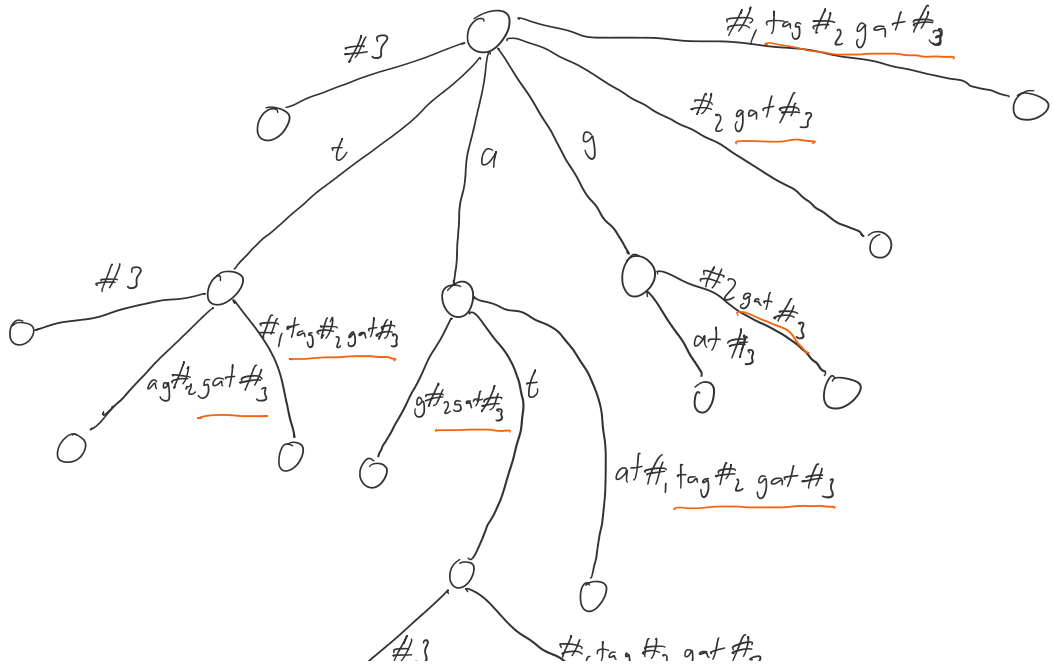


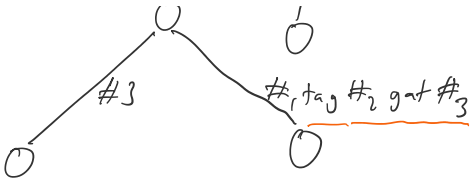
Generalizing suffix trees to multiple strings

Goal: represent set of strings $P = \{s_1, s_2, \ldots, s_m\}$

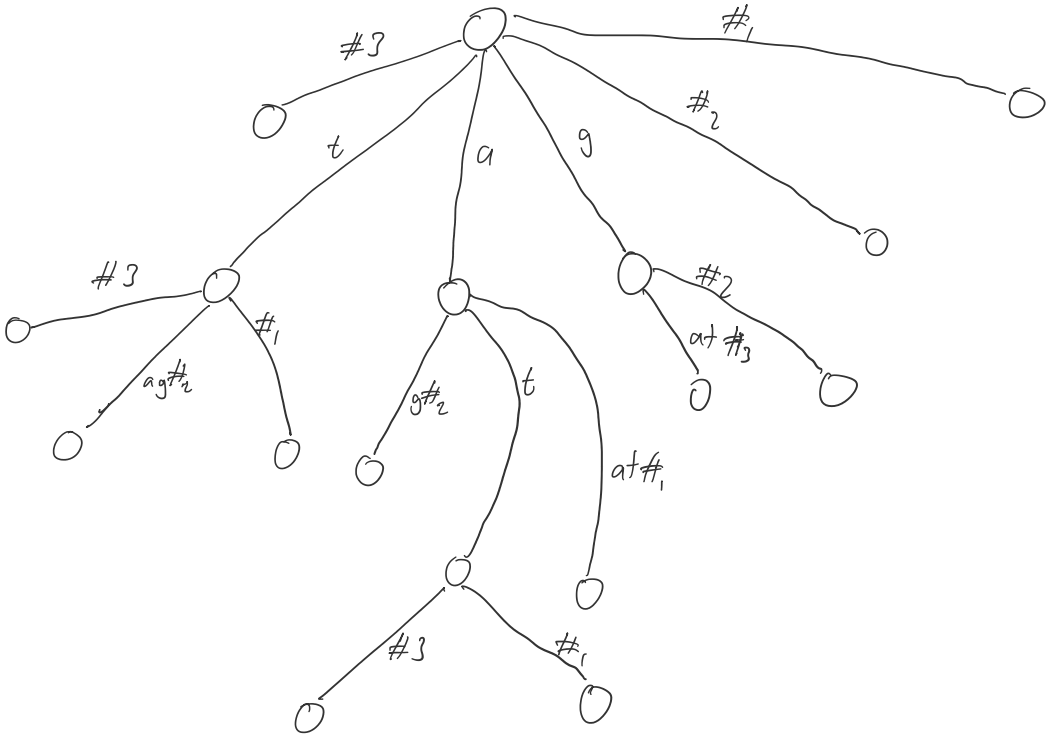
Ex. aat, tag, gat

Sol: build suffix tree for $aat\#_1 tag\#_2 gat\#_3$





Notice the #'s are only on leaf nodes because they are unique.
Remove any text after the first # on a leaf



When building suffix tree, can just stop on any end-of-string char #.

Applications:

Longest common substring of S, T.

Build generalized suffix tree for $\{S, T\}$.
Find deepest node that has descendents from both strings (containing both $\#_1$ & $\#_2$)

Determine strings in database $\{S_1, \ldots, S_m\}$ that contain query q.

Build generalized suffix tree.
Follow path for q.
If we end at a node u, traverse the tree below u & emit i if $\#_i$ is found.

Longest Common Extension at i, j

Given strings S & T, we want to quickly find the longest substring of S starting at i that matches a substring of T starting at j.

Ex. AAAATTTTCCCCGGGG

$i = 9$

TTTTCCCCGGGGAAAA

$j = 5$

Can solve by walking on both strings until mismatch, but we want to do this faster by preprocessing.

Sol. Build generalized suffix tree for S, T. $O(|S|+|T|)$ time

Preprocess tree so that lowest common ancestors can be found in constant time.

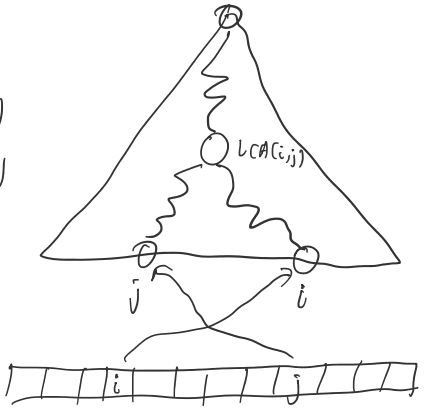
} We have not covered how to do this, but preprocessing takes $O(|S|+|T|)$ time. (very complicated data structure)

Create array mapping suffix numbers to leaf nodes $O(|S|+|T|)$ time.

Given query (i, j):

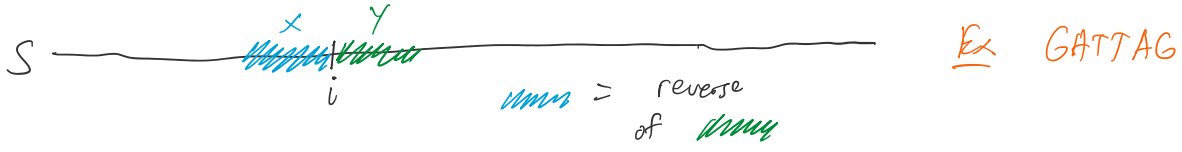
Find leaf nodes for i & j. $O(1)$

Return string of LCA for i, j. $O(1)$



Using LCE to find palindromes

Maximal even palindrome at position i: the longest string & the left & right such that the left half = reverse of right half.



Ex GATTAG

Goal: find all maximal even palindromes

Construct S^r, the reverse of S. $O(|S|)$

Preprocess S & S^r so LCE queries can be solved in $O(1)$ time. $O(|S|)$.

$LCE(i, n-i)$ is the length of the longest palindrome centered at i.

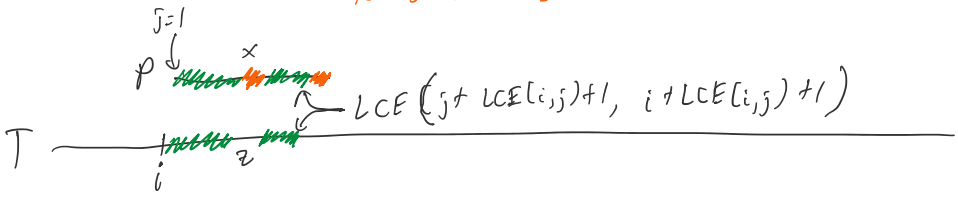
$\forall$ pos i: $O(|S|)$

Compute $LCE(i, n-i)$ $O(1)$ } $O(|S|)$

k-mismatch using LCE

Given a long reference string T, and a short string P, check if there is a k-mismatch of P starting at pos i of T.

└ k substitutions of characters in P to be an exact match.



j=1 pos in P

c=0 number of mismatches so far

O(k) { Repeat until c>k:

  j = j + LCE(i,j) + 1 } $O(1)$ to get longest extension of (i,j)

  i = i + LCE(i,j) + 1

  If j ≥ |P|+1, return True matched all of P

  c = c+1

return False

Finding all k-mismatches takes $O(k|T|)$ time.

Suffix arrays

Suffix trees are $O(n)$ space, but the constant is large (≈ 20 bytes per char in good implementation)

10 Gb genome ⇒ 200 GB to store

Suffix arrays are more efficient even than suffix trees & do most of the same ops.

Idea: Sort all suffixes of string lexicographically & keep the indices

S = catttcat $

1	catttcat $
2	attcat $
3	ttcat $
4	tcat $
5	cat $
6	at $
7	t $
8	$

sorted →

8	$
6	at $
2	attcat $ ← 2 ✓
5	cat $
1	catttcat $ ← 1
7	t $
4	tcat $
3	ttcat $

↳ only keep indices & original string S.

Search substring by binary search on suffixes e.g. at

Counting is easy because all suffixes starting w/ "at" are next to each other.

Application: k-mer counting for all k-mers

Construction: $O(n^2 \log n)$ by sorting n suffixes, where each comparison takes $O(n)$ time.

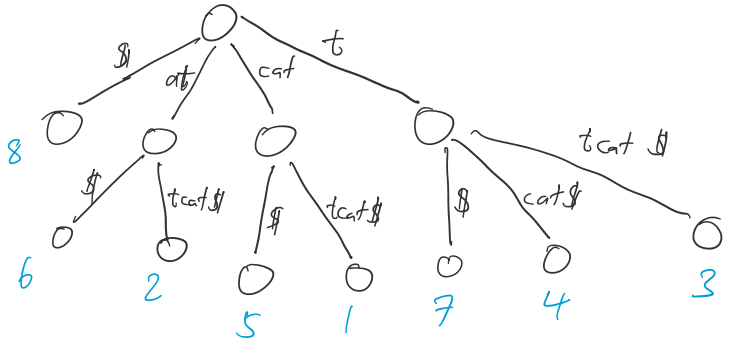
But better $O(n)$ algorithms, such as Skew via Divide & Conquer.

Alternately, can use suffix tree.

$\Sigma = \{\$, a, c, t\}$

S = catttcat $

1 2 3 4 5 6 7 8



If edges sorted lexicographically, the leaf labels are exactly suffix array.