Lec20-more-suffix-trees-arrays

Sunday, October 20, 2024 10:14 PM

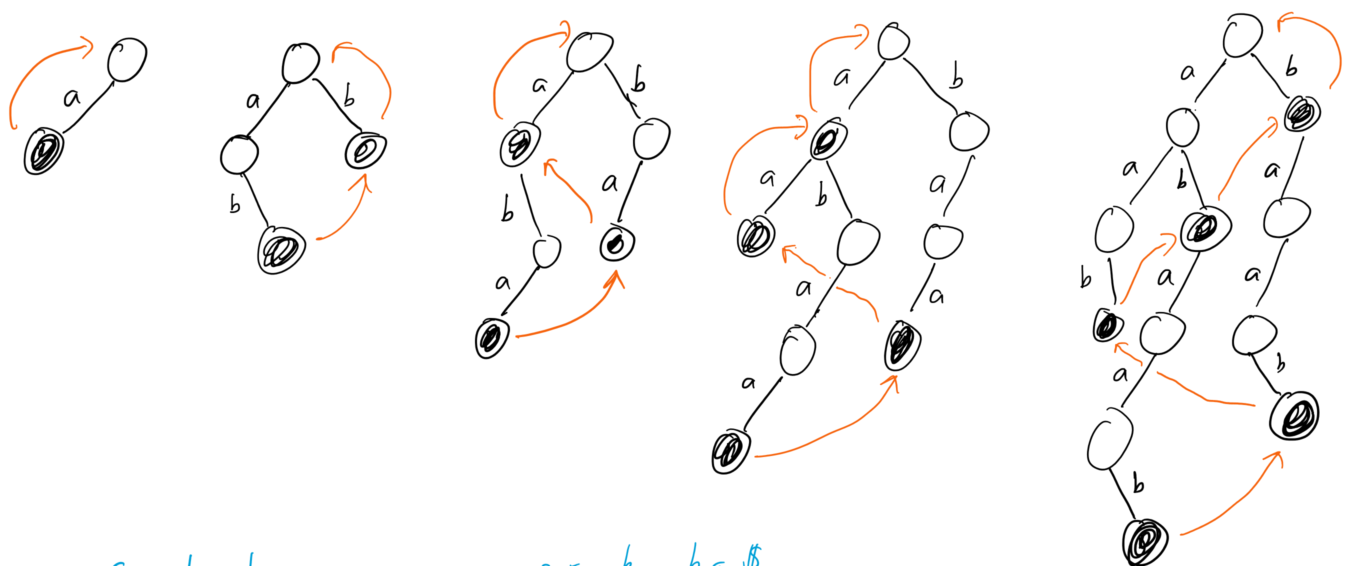
Last time we learned about suffix tries & trees.

Let's remind ourselves of suffix trie construction,

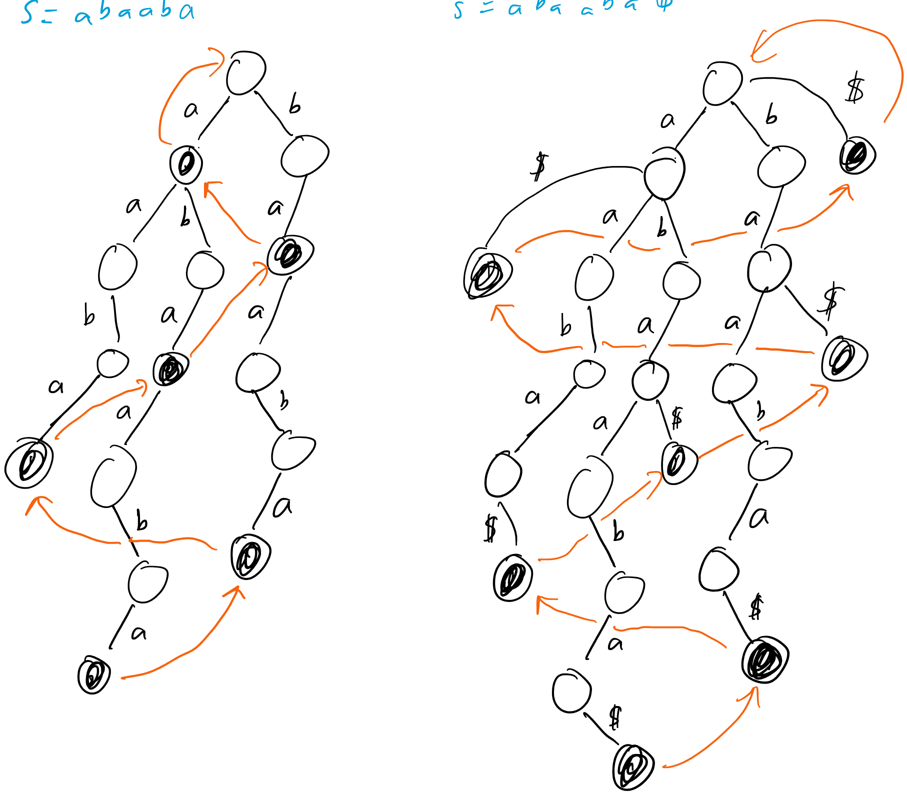
We build it up while walking the string from left to right.

s = abaaba$

s = a　　s = ab　　s = aba　　s = abaa　　s = abaab



s = abaaba　　s = abaaba$



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

Ex　abaaba$
　　bbaa

Longest Common Substring ... of ... S

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.

Ex: bbaa

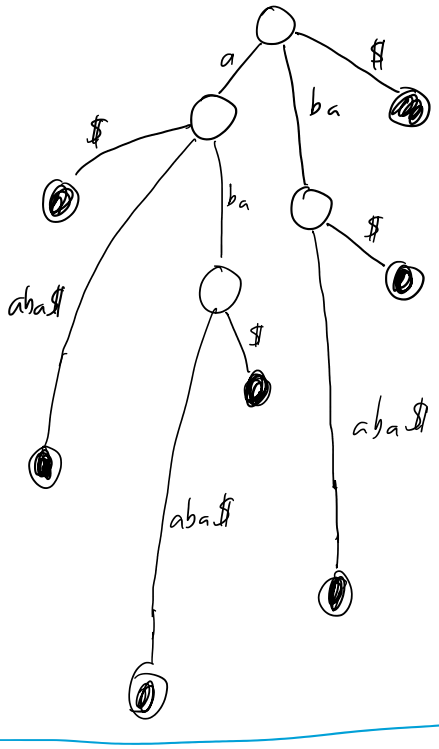
Start at b

Dead end.

Start at ∅ → b → a → a.

Done.

We can also compress it to a suffix tree.



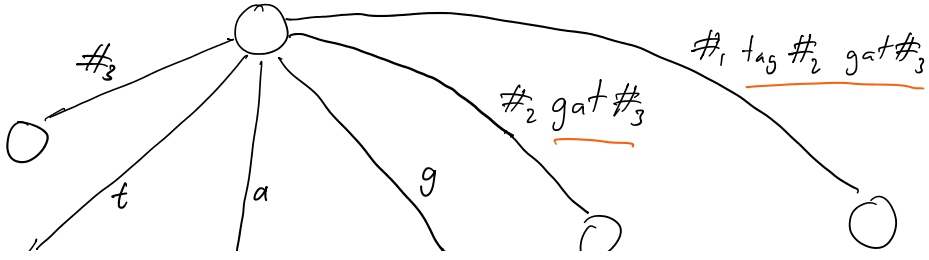
Generalizing suffix trees to multiple strings (GST)

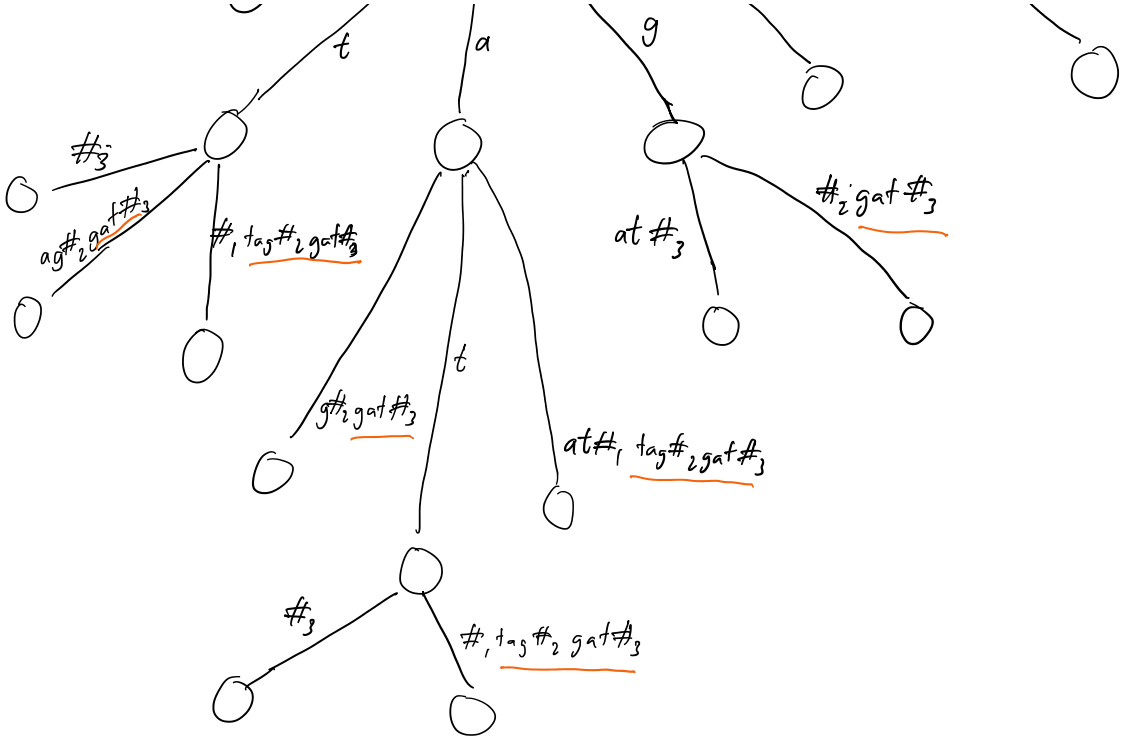
Goal: represent set of strings $P = \{S_1, S_2, \ldots, S_m\}$

Ex: aat, tag, gat

Sol. build suffix tree for aat#1 tag#2 gat#3

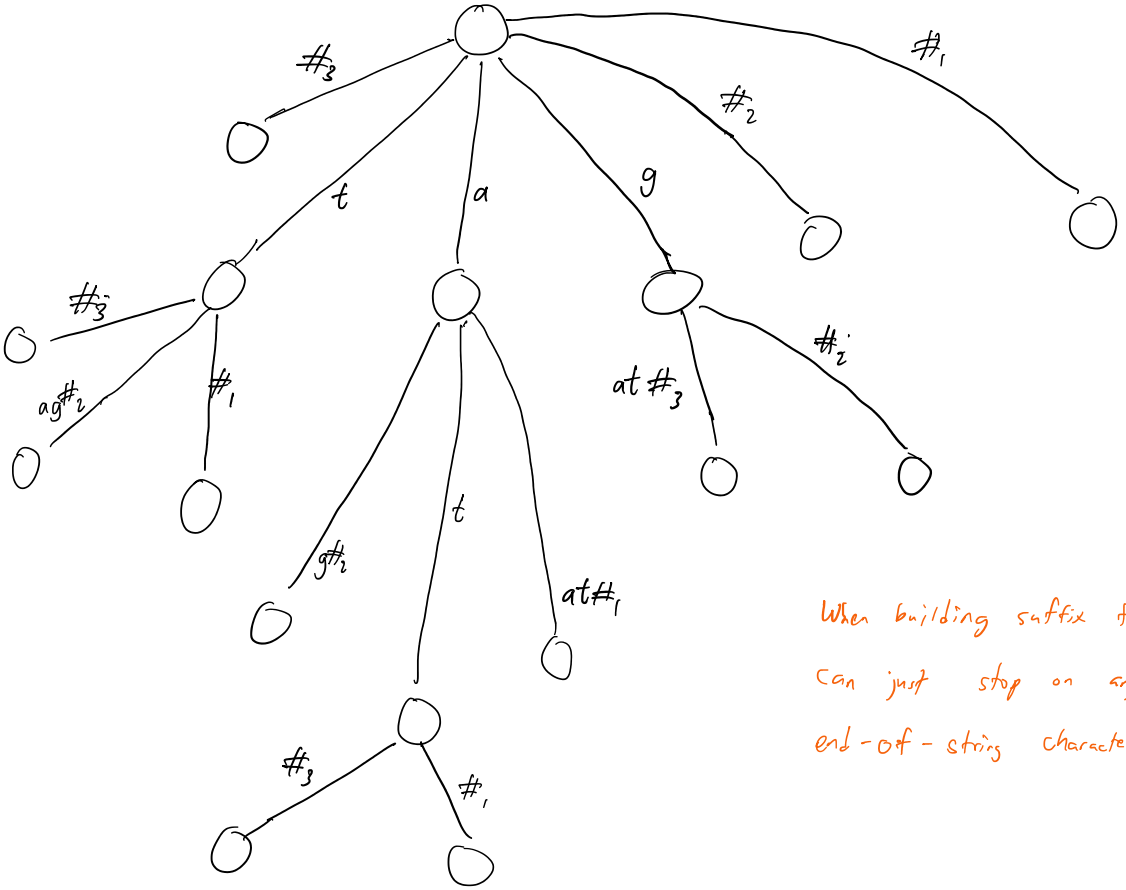
unique string terminators





Notice the #'s are only on leaf nodes because they are unique

Remove any text after first # on leaf.



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

Applications:

Longest common substring of S, T

Build generalized suffix tree for $\{S, T\}$.

Find deepest node that has descendants from both strings (containing both $\#_1$ & $\#_2$)

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

Build GST.

Follow path for q.

If we end at node u, traverse the tree below u & emit i if $\#_i$ is found.

Longest Common Extension at i, j

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

Ex. AAAATTTTCCCCGGGG

$i = 9$

TTTTCCCCGGGGAAAA

$i = 5$

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

Sol. Build GST for S, T. $O(|S| + |T|)$ time.

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

We are not covering LCA in 15351, but exists complicated data structure that takes $O(|S|+|T|)$ time to build initially.

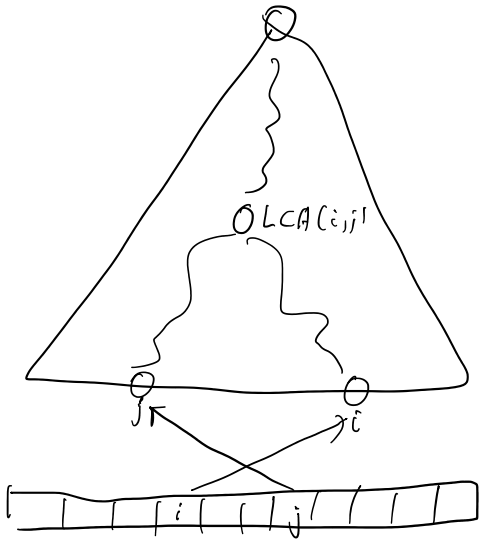
In 15351/02613, we take for granted this LCA data structure.

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

Given query (i, j):

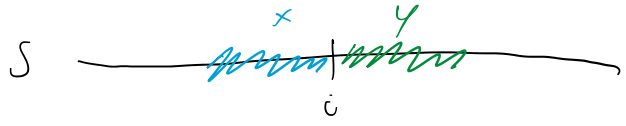
Find leaf nodes for i, j. $O(1)$

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



Using LCE to find palindromes

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

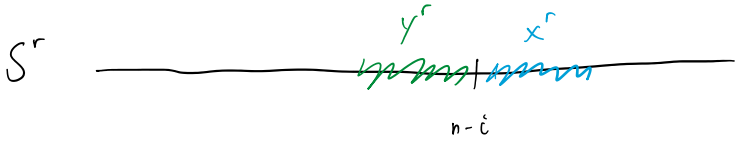


E.g. GATTAG

$x = \text{reverse}(y)$

Goal: find all maximal even palindromes.

Construct S^r = reverse(S). $O(|S|)$



Preprocess S & S^r so LCE queries take $O(1)$ time. $O(|S|)$

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

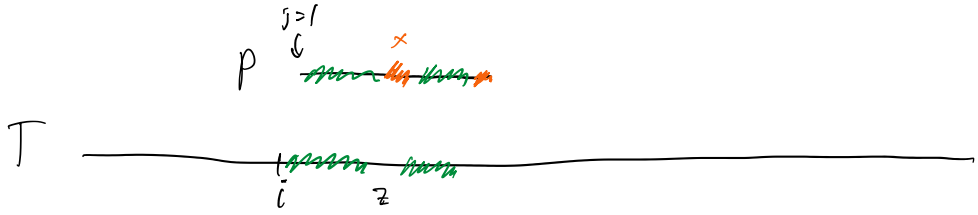
$\forall$ pos i:

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

k-mismatch using LCE

Given long reference string T, & a short string P, check if $\exists$ a k-mismatch of P starting at pos i of T.

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



$j = 1$ pos in P

$c = 0$ # mismatches so far

$O(k)$ {

Repeat until $c > k$:

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

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

If $j \geq |P| + 1$, return True. matched all of P

$c = c + 1$

return False

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

Suffix arrays

Suffix trees are $O(n)$ space, but the constant is large.

($\approx$ 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.

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

$S = catttcat\$$

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

sorted →

8	$
6	at$
2	attttcat$
5	cat$
1	catttcat$
7	t$
4	ttcat$
3	tttcat$

Search substring by binary search of suffixes.

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

App: k-mer counting for all k-mers.

Only keep indices of original str S.

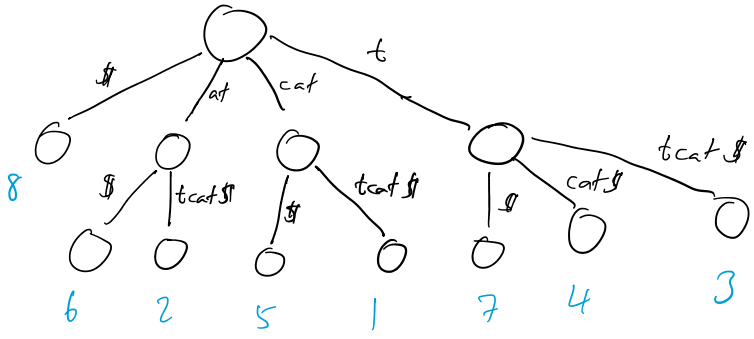
Construction: $O(n^2 \log n)$ by sorting suffixes, where each comparison takes $O(n)$ time.

But better $O(n)$ algorithms, such as building suffix tree.

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

$S = catttcat\$$

12345678



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