GICB Recitation Session 3/4

Wendy Yang / Hongyu Zheng

Overview

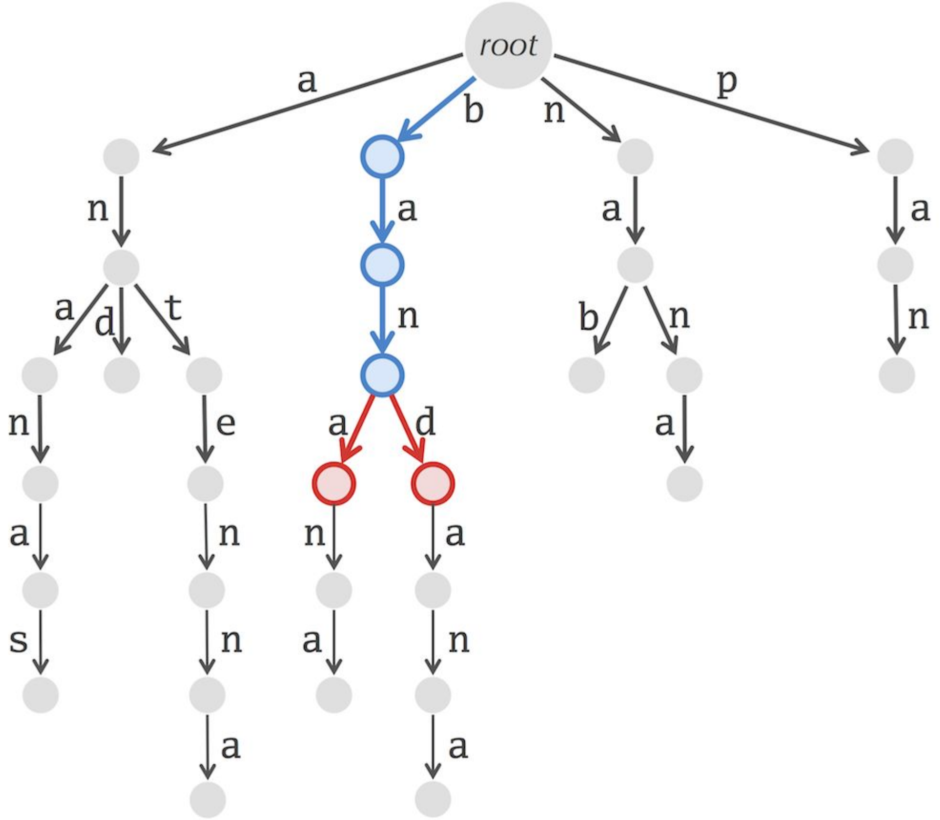
Stay ~~warm~~ cool?

- Lecture Review
- MOAR ALGORITHMS:
    - Implementing Suffix Tree
    - Suffix Array

Recall: Matching with Trie

Imagine sliding the trie shown below against *Text* = "bantenna". Like **TrieMatching**, the Aho-Corasick algorithm starts at the root and attempts to build a path spelling a prefix of "bantenna". This attempt fails after three nodes ("bantenna"). In **TrieMatching**, we would begin again at the root and attempt to find a match starting at the 2nd position of "bantenna".

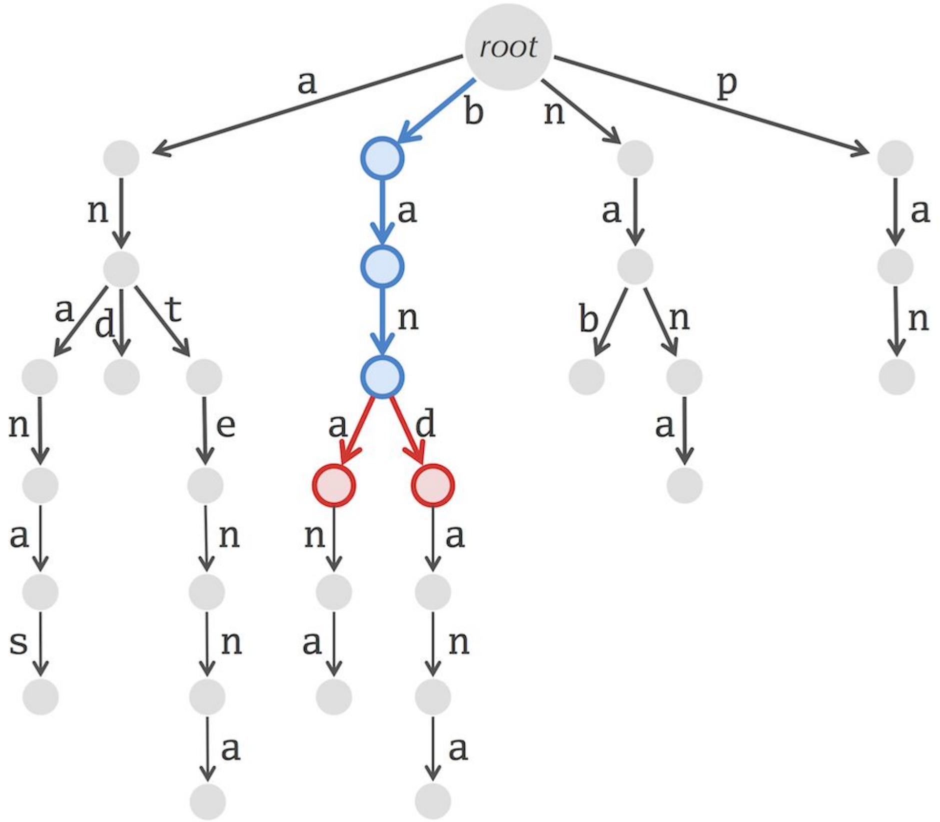
Think: Can I not move to root? I knew I matched ban already(and I missed at t)



Failure Edges

A more sensible strategy is to jump directly to the node "an" and then continue downward in the trie. Indeed, this strategy will eventually match the pattern "antenna".

We can implement this "jumping ahead" strategy by augmenting the tree with a **failure edge** connecting node "ban" to node "an"..



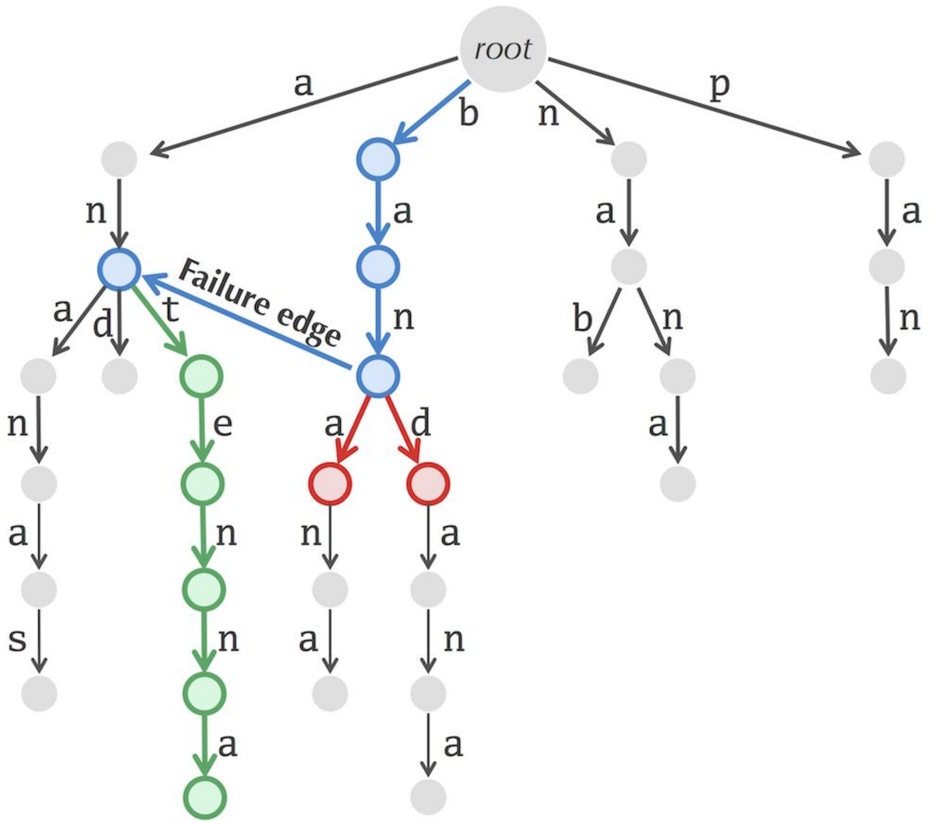
Failure Edges

Failure edges are formed by connecting node *v* to node *w* if *w* is the longest suffix of *v* that appears in the trie. After constructing all failure edges, the Aho-Corasick algorithm then follows failure edges during pattern matching whenever a mismatch is found in order to avoid going all the way back to the root, thus saving time.

bantenna

bantenna

Trivia: Failure edges on Suffix Trie is equivalent to next[] array in Knuth-Morris-Pratt(**KMP**)



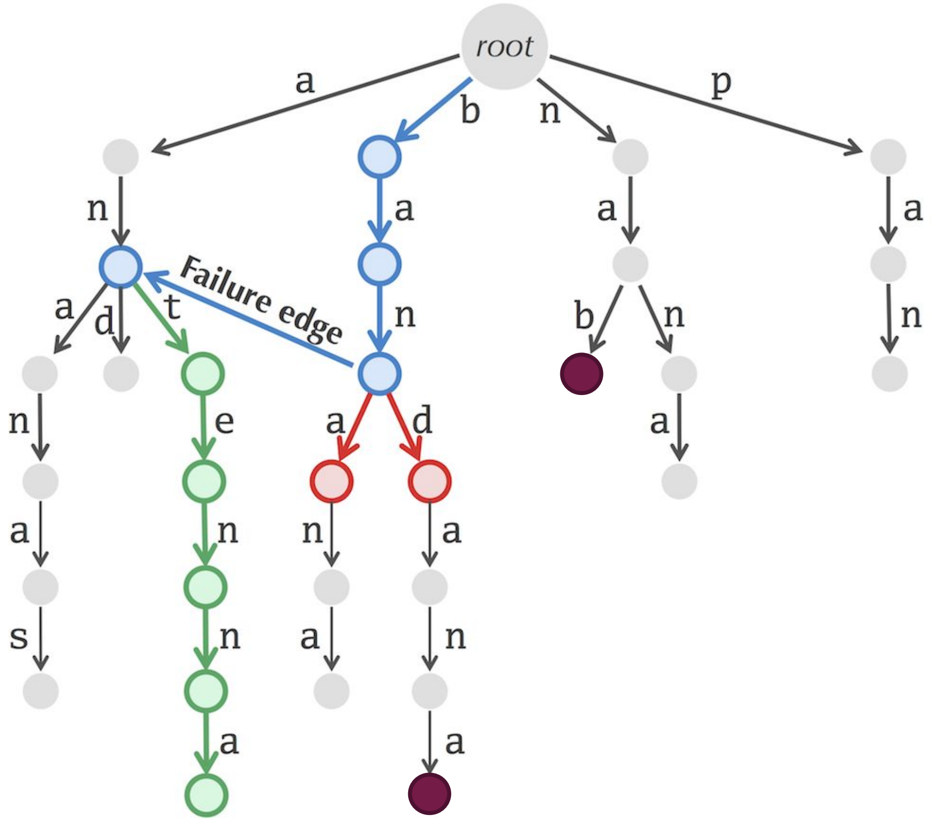
Aho-Corasick Automaton

After constructing all failure edges, the Aho-Corasick algorithm then follows failure edges during pattern matching whenever a mismatch is found in order to avoid going all the way back to the root, thus saving time.

(The resulting data structure can be seen as an automaton using Trie nodes as states.)

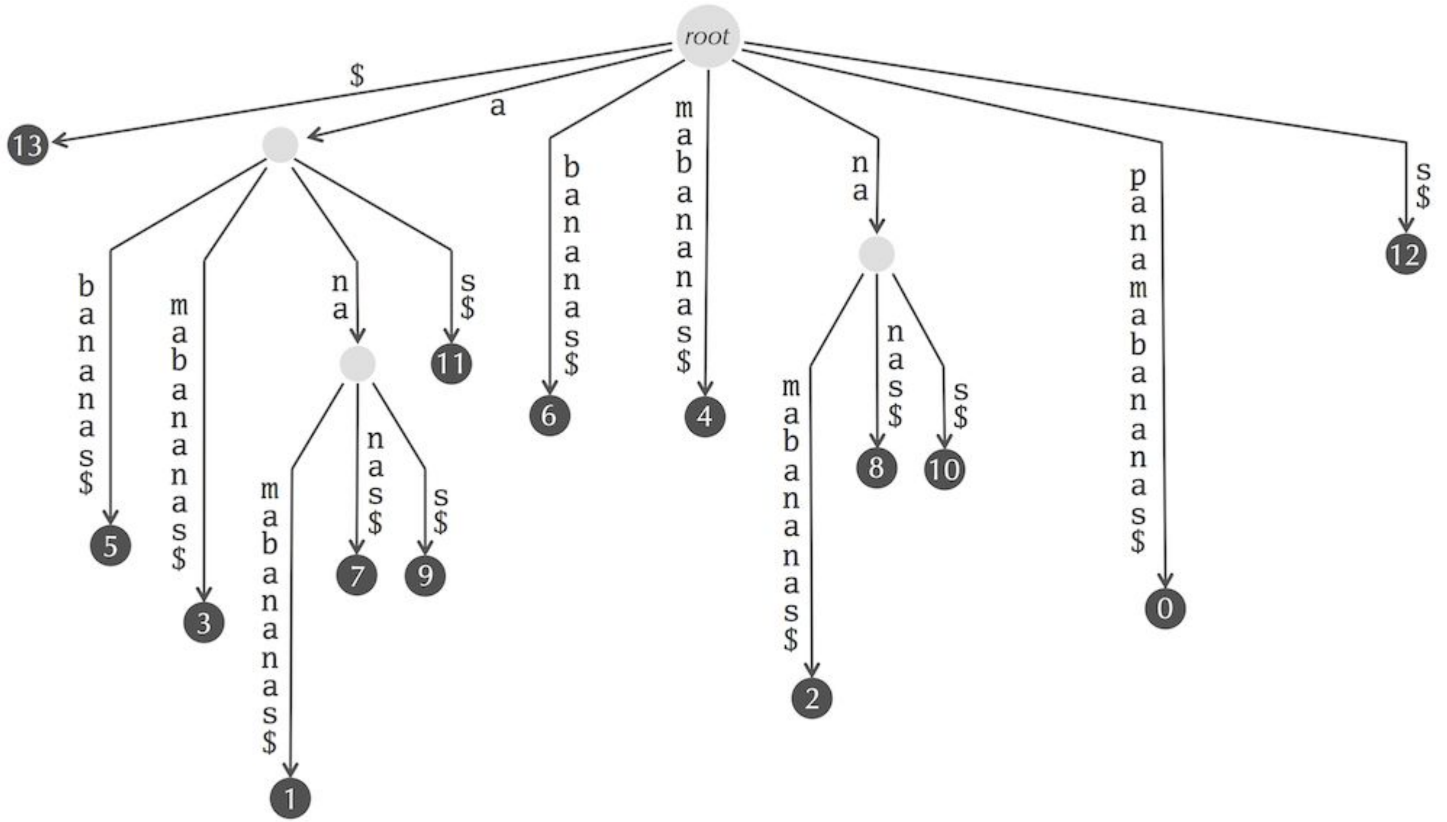
Question 1: What's the failure edge at "bandana" and "nab"?

Question 2: By exactly what means is this better?



Suffix Tree

This is a suffix tree.

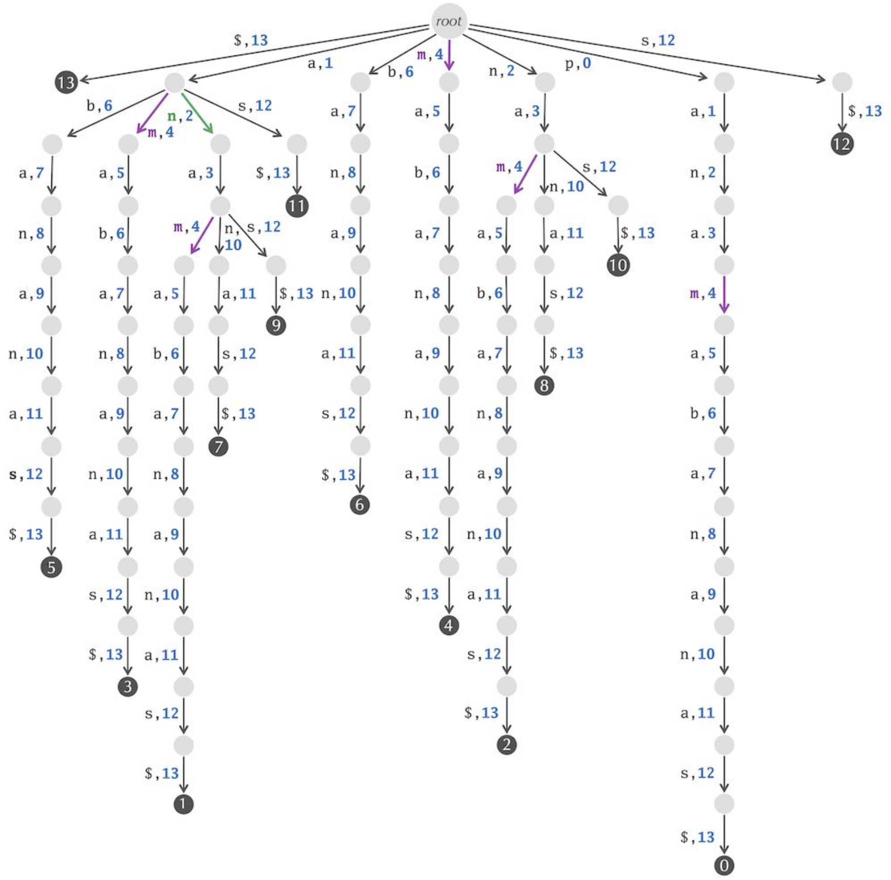


Locating Suffix Trie Nodes

For each edge, also store the earliest suffix it resides.

There are five edges labeled by "m" (colored purple), all of which are labeled by position 4(there is only one m)

The green edge corresponding to "n" corresponds to occurrences of "n" in the suffixes "anamabananas$", "ananas$", and "anas$", at positions 2, 8, and 10. As a result, we assign this edge the minimum of these positions.

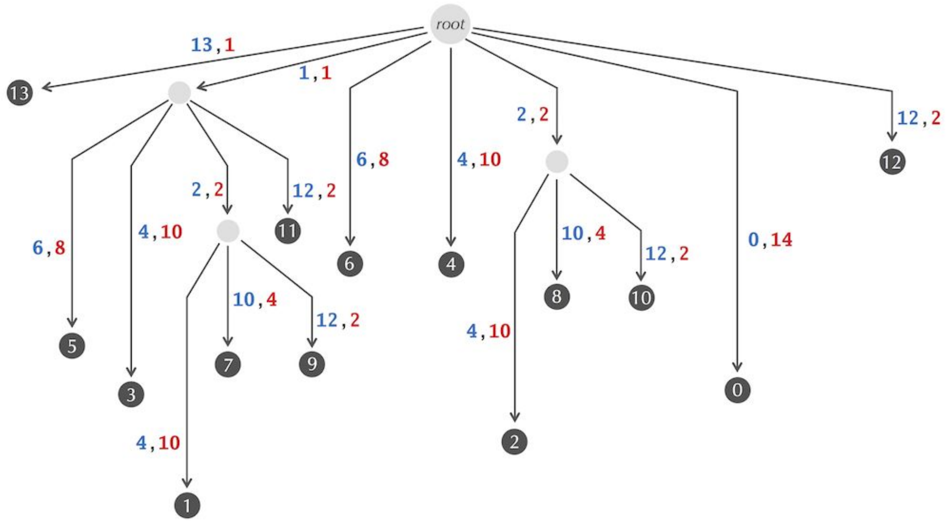


Compact Representation

Recall some time ago we have a thing called "max nonbranching path"...

There is usually no need to store the whole substring on the suffix tree since we know it's a substring of original string.

What's the time and space complexity for building a suffix tree in this way?



Suffix Tree for Multiple Strings

What if I want a tree for all suffixes of string A and B?

(For example, when I want the suffix tree for all 23 chromosomes to query on them together)

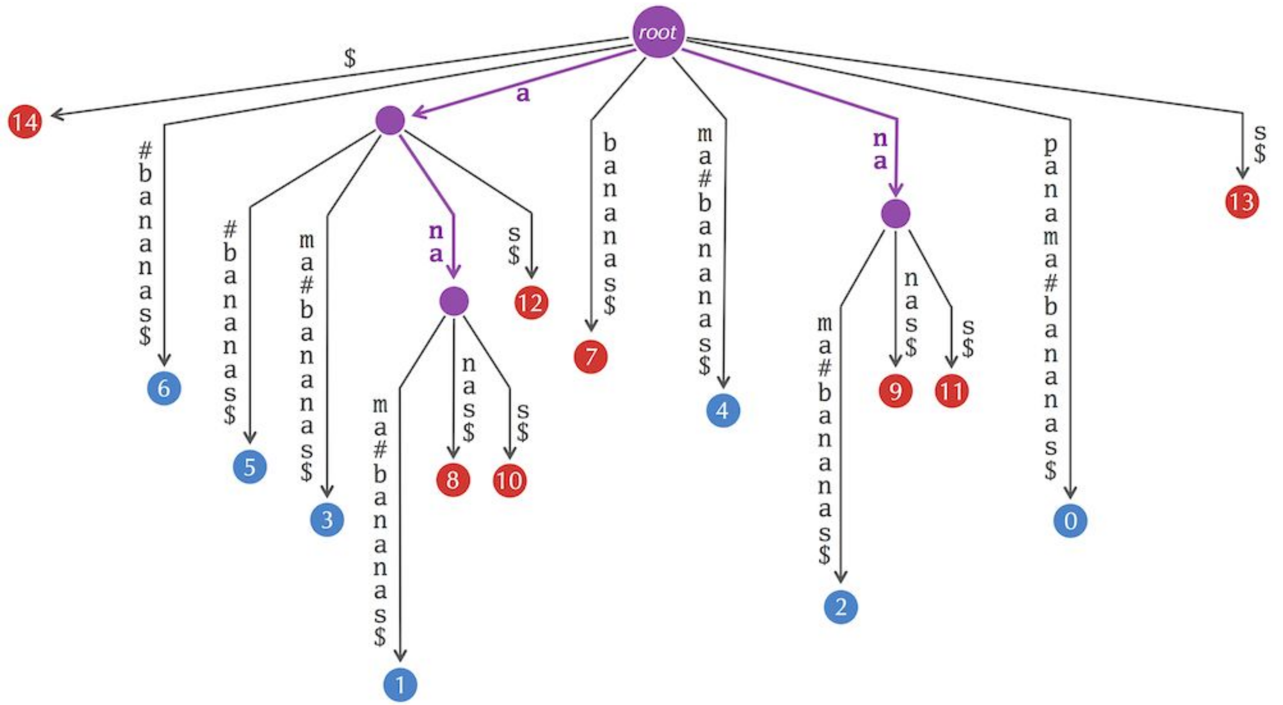
Tips: DO NOT invent new data structures

Suffix Tree for Multiple Strings

We build the suffix tree for string of form “str1#str2$”

What are the purple nodes?

panama#bananas$



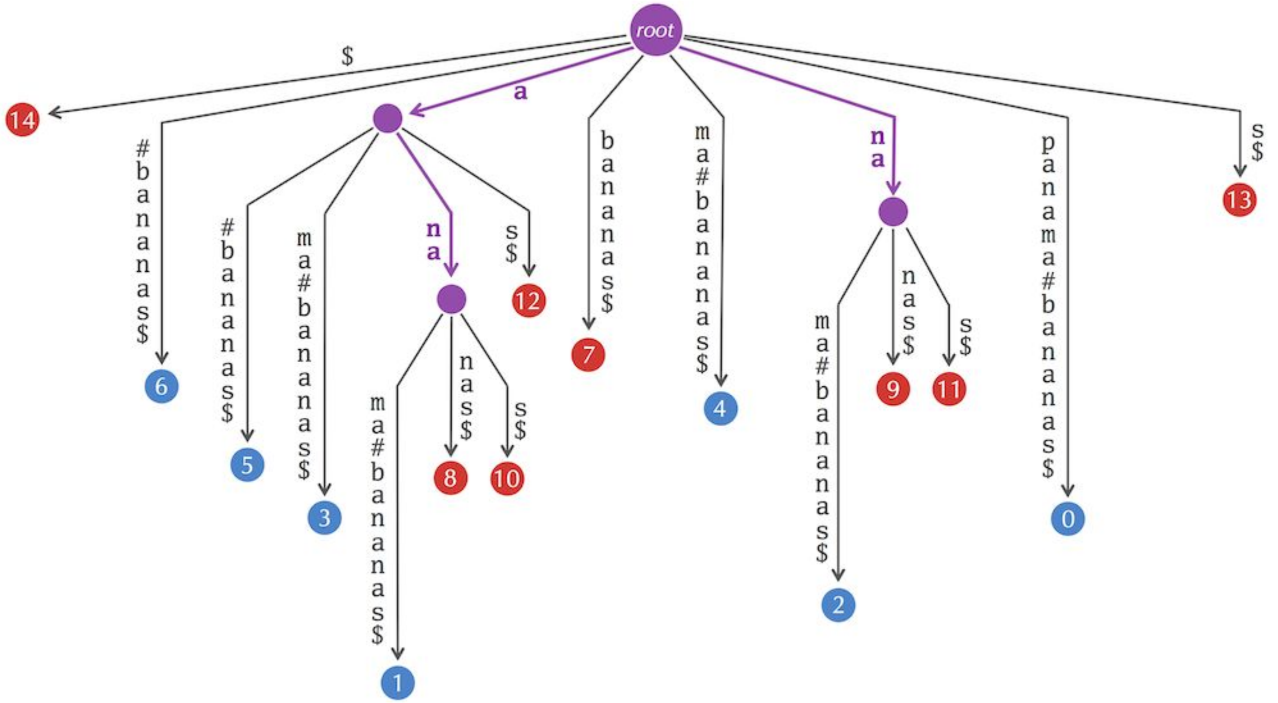
Suffix Tree for Multiple Strings

panama#bananas$

Cool fact □: purple nodes represents all common substrings

panama

bananas



Some practice problems (Suffix Tree)

Exercise 1. Extend the previous algorithm to find longest common substring for K strings.

Exercise 2. How would you build the suffix tree for a circular string(say a circular bacterial chromosome)?

Exercise 3. Use suffix trie to find the longest recurring substring in S. Overlap is allowed.

Example: S = **abc**def**abc**efg, S = **ab<u>ab</u>ab**cd

Solution to Practice Problems

These problems are only for "fun" and are not necessarily covered in exams.

Solution 1. You can extend the process naturally by attaching a unique terminal identifier for each string, then build the suffix trie of it.

Solution 2. Usually we will build the suffix tree of SS(S written twice). For circular S you can start reading it from any location, which will correspond to a substring of SS.

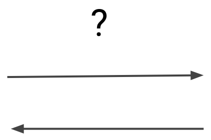
Solution 3. It's the deepest non-leaf node in the suffix trie.

Suffix Arrays and Suffix Trees

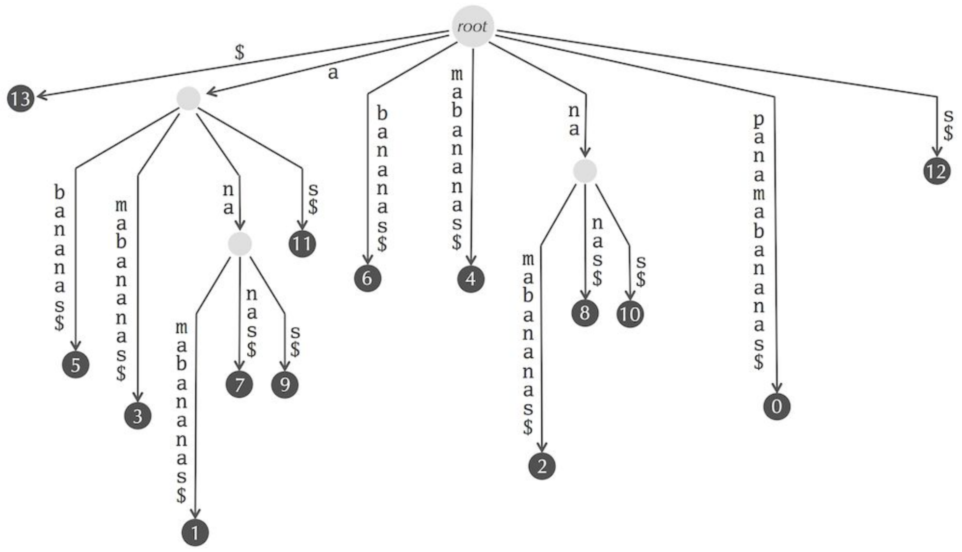
Recall:
Suffix Array

Starting Positions	Sorted Suffixes
13	\$
5	abananas\$
3	amabananas\$
1	anamabananas\$
7	ananas\$
9	anas\$
11	as\$
6	bananas\$
4	mabananas\$
2	namabananas\$
8	nanas\$
10	nas\$
0	panamabananas\$
12	s\$

SuffixArray("panamabananas\$") = (13, 5, 3, 1, 7, 9, 11, 6, 4, 2, 8, 10, 0, 12).



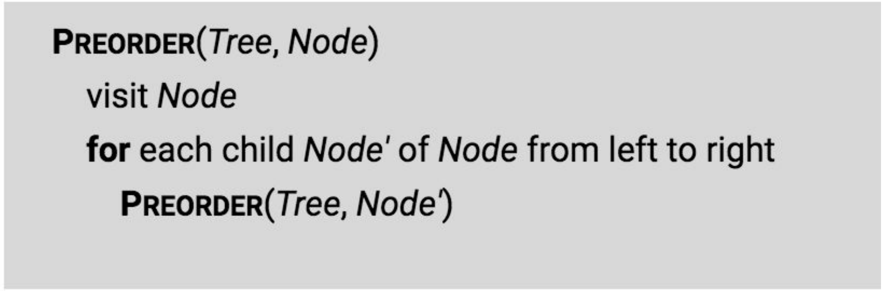
Suffix Tree



Suffix Tree -> Suffix Array

Assume we have a lexicographically arranged suffix tree

Then just do a preorder traversal to find the suffix array!



```
PREORDER(Tree, Node)
    visit Node
    for each child Node' of Node from left to right
        PREORDER(Tree, Node')
```

A linear-time algorithm to build suffix array?

Suffix Array -> Suffix Tree

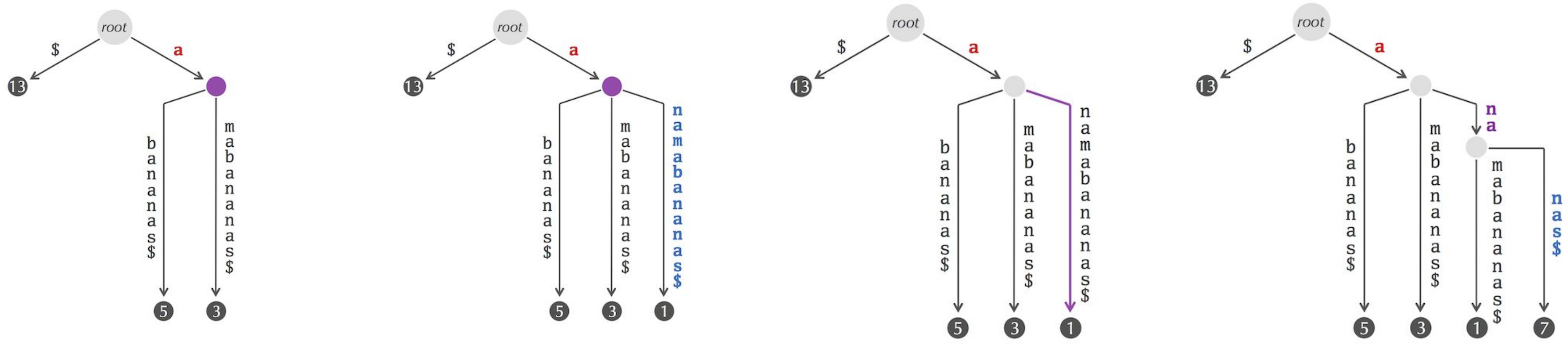
Challenging! Let’s get some help from Longest Common Prefix(LCP) array!

LCP stores the length of the longest common prefix shared by consecutive lexicographically ordered suffixes of *Text*.

LCP Array	Sorted Suffixes
0	$
0	abananas$
1	amabananas$
1	anamabananas$
3	ananas$
3	anas$
1	as$
0	bananas$
0	mabananas$
0	namabananas$
2	nanas$
2	nas$
0	panamabananas$
0	s$

Suffix Array -> Suffix Tree

Problem: Constructing suffix tree given suffix array and LCP array.



The power of text compression tools

Example of Bzip2 on:

- E.coli genome (⅓ of original size!)
- War and peace

More discussion!

1. A frequent question is why not use some other column of the Burrows-Wheeler matrix as the Burrows-Wheeler transform; the natural candidate is the second-column since it is adjacent to the first column. Why is using the second column of the matrix a bad idea? Produce an example to justify your answer.

2. A similar question that is sometimes asked is whether it’s possible for two different strings to have the same first two columns of the Burrows-Wheeler matrix. Is it possible? What about the first *k* columns? What is this connected to that we have done before?

Some practice problems (Suffix Array)

Assume you only have access to the suffix array and LCP array of S, and nothing else, except for 3.

Problem 1(Easy). Find the length of longest recurring substring in S. Overlap is allowed.

Problem 2(Medium). Answer the LCP of S[i:] and S[j:] (two suffixes of S). **Optional:** O(log n) per query.

Problem 3(Medium). A Palindrome is a string that reads the same in both directions. Find the longest palindromic substring of S.

Problem 4(Hard). Given a string S, how many different substrings does it have?

Problem 5(Hard). Find the length of longest recurring substring in S. Overlap is **NOT** allowed.

Solutions to Practice Problems

These problems are only for "fun" and are not necessarily covered in exams.

You are not required to understand all the solutions(esp. 4&5), but it's good to understand Solution 2 for it is an fundamental part in suffix array applications.

Solution 1(Easy). It's the maximum element in the LCP array.

Solution 2(Medium). As shown in recitation, find the "interval" on LCP array that starts with S[i:] and ends with S[j:] by using suffix array as a lookup table, then the answer is the minimum element on this interval.

To achieve log(n) per query, use a data structure for RMQ(Range Minimum Query) (Segment tree, linearithmic lookup table, etc). This is the real takeaway: LCP arrays can be used to answer these questions much more efficiently.

Solutions to Practice Problems

Solution 3(Medium). Build the suffix array and LCP array of SS’(S concatenated with reverse of S).

We iterate over possible centers of palindromes and assume it’s odd length. Then, use the procedure in Problem 2, query the LCP of S[i:] and S’[n-i:], which will be the number of “extensions” you can go anchoring your palindrome at location i. Answer will be 2*LCP+1.

Palindromes of even length can be proceed in a similar way.

Solutions to Practice Problems

Solution 4(Hard). The answer is n(n+1)/2 - sum(LCP).

Every substring of S is a prefix of a suffix of S(think about it), so we can first count number of prefixes for each suffix of S, then remove duplicates. The number is n(n+1)/2 by a simple counting argument.

We now use the convention that a substring is valid the first time it appears as a prefix of a suffix, in the suffix array order(lexicographically sorted suffixes). Now, we want to know for a suffix, how many of its prefixes are invalid. Intuitively, the number is the LCP of this suffix and the suffix before it in the suffix array order.

Solutions to Practice Problems

Solution 5(Hard). First do a binary search on the answer and we will answer the question of the form “Is there a length k recurring substring in S”. Now, this is equivalent to asking if there exists indexes i and j, such that j-i>=k and LCP(S[j:], S[i:])>=k.

From Solution 2, we know LCP(S[j:], S[i:]) is minimum over a interval in LCP. So LCP(S[j:], S[i:]) >= k is equivalent to say there are no <k elements between S[i:] and S[j:] in LCP array. We now “break” the LCP array into contigs by removing all elements less than k, and check for each remaining contigs if its maximum and minimum index(as suffix of S) is apart by at least k.

This can be done in linear time, making total time complexity O(nlogn) where n is length of S.

End of Recitation. Feedbacks welcome!