Design & Analysis of Algorithms
COMP 482 / ELEC 420

John Greiner

String Matching

Pattern: **CCATT**

Text: **ACTGCCATTCCTTAGGGCCATGTG**

- Brute force & variant
- Automata-like strategies
- Suffix trees

To do:
[CLRS] 32
Supplements #5

Some Applications

- Text & programming languages
  - Spell-checking
  - Linguistic analysis
  - Tokenization
  - Virus scanning
  - Spam filtering
  - Database querying
- DNA sequence analysis
- Music identification & analysis

Brute Force Exact Match

Pattern: **`CCATT`**

Text: **`ACTGCCATTCCTTAGGGCCATGTG`**

Algorithm?

Example of worst case?

Running time?

Rabin-Karp, 1981

Pattern: **`CCATT`**

Text: **`ACTGCCATTCCTTAGGGCCATGTG`**

Hash pattern & $|P|$-substrings

- Compare hashes.
- Compare strings only when hashes match.
- $h(xb)$ easily computed from $h(ax)$, b

Best and worst cases?

Intuition for Better Algorithms

Pattern: **`CCATT`**

Text: **`CCAGCCATTCCTTAGGGCCATGTG`**

After failing match at 1st position, what should we do?

Quick Overview of Two Algorithms

Knuth-Morris-Pratt, 1977

- Use previous intuition.
- Preprocessing builds shift table based upon pattern prefixes. $O(|P|)$
- Each text character compared once or twice. $O(|T|)$

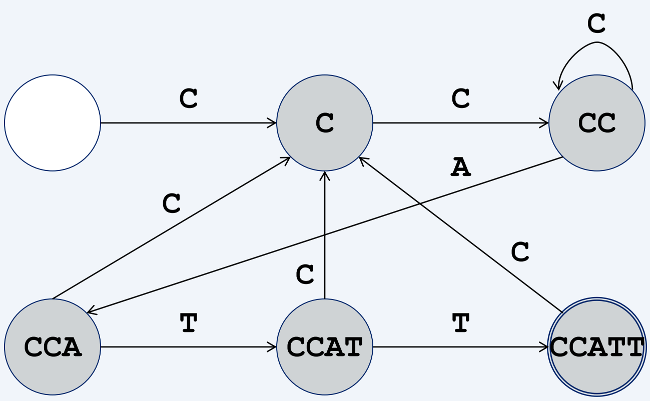
Boyer-Moore, 1977 & Turbo Boyer-Moore, 1992

- Use previous intuition, along with similar heuristics. Complicated.
- Match from right end of pattern.
- Can often skip some text characters. $O(|T|)$, with $O\left(\frac{|T|}{|P|}\right)$ best case.

Finite Automata Matching Example

Pattern: **CCATT**

Text: **CCAGCCATTCCTTAGGGCCATGTG**



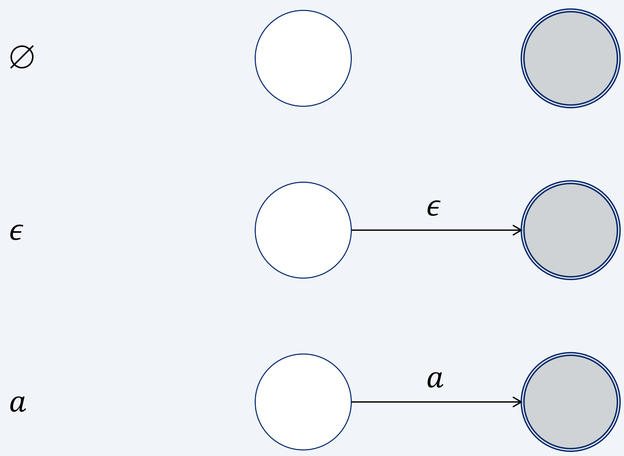
$O(|P| \cdot |\Sigma|)$ to build

Regular Expressions

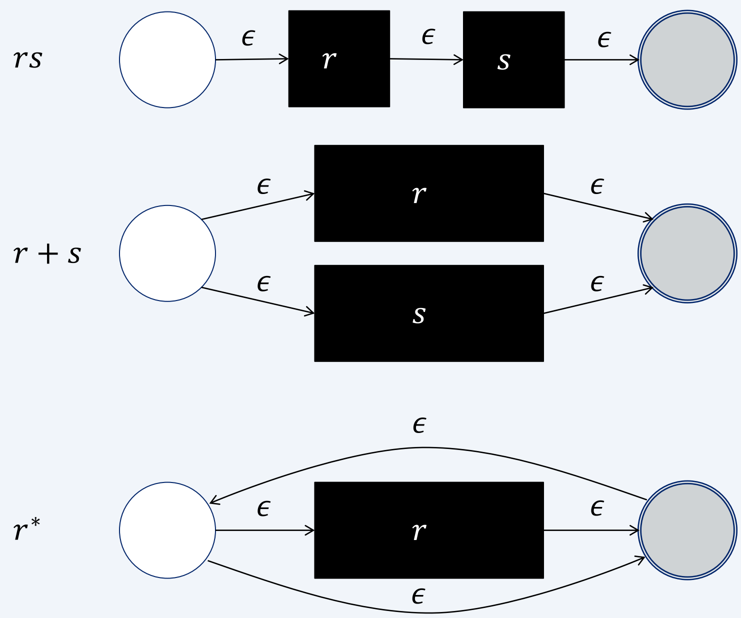
$$(0+1)(00+01+10+11)^*$$

Syntax: $\emptyset$, ϵ, a, rs, $r+s$, r^*

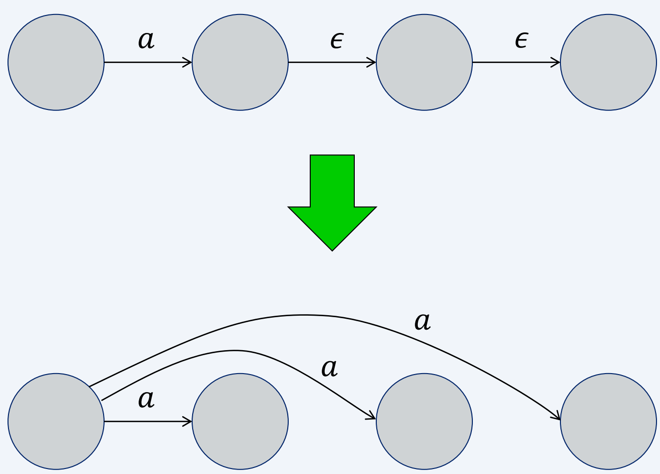
Regular Expression to Finite Automaton



Regular Expression to Finite Automaton

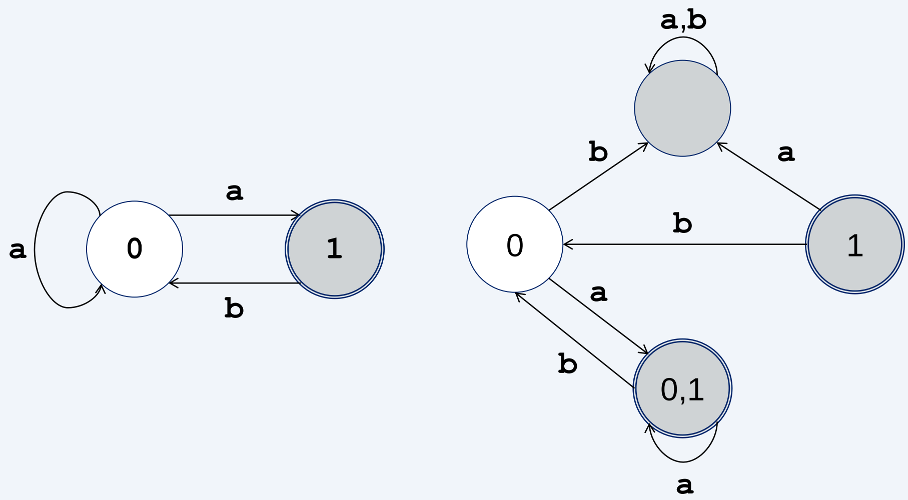


Eliminating ϵ



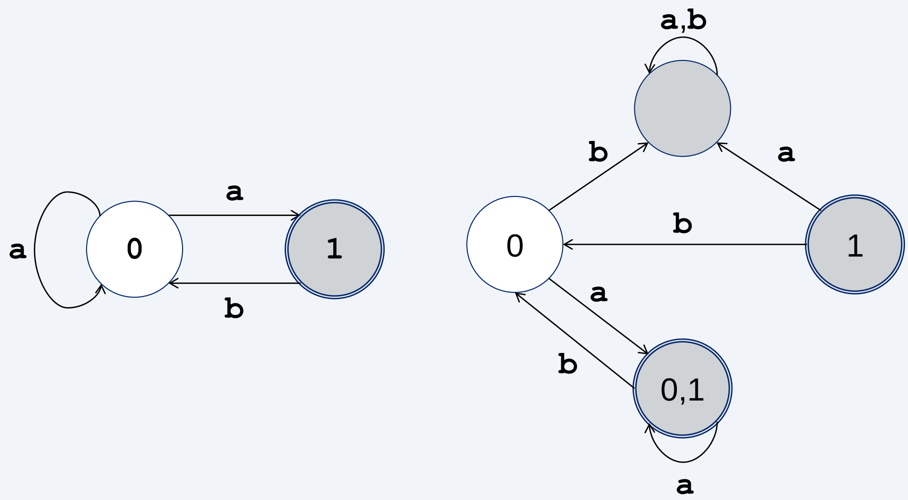
Eliminating Non-Determinism

Each state is a set of the old states.



Can Minimize

States unreachable or equivalent? $O(|Q|^2)$

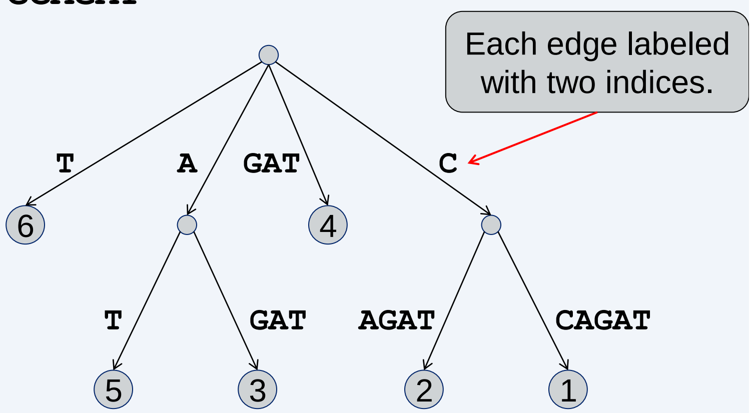


Finite Automaton Approach Summary

Preprocessing expensive for REs.
Matching is flexible and still linear.

Suffix Tree (Trie)

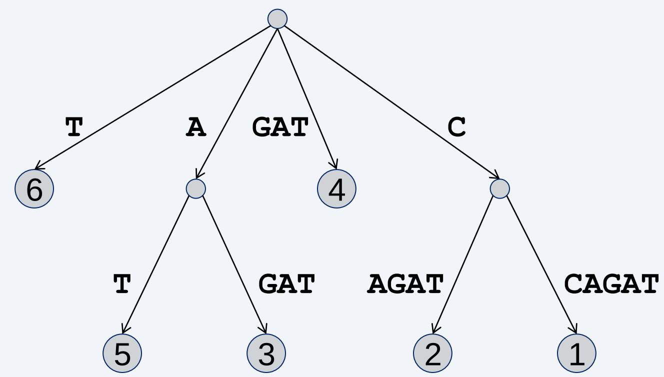
Text: **`CCAGAT`**



How many leaves, nodes, edges? Total space?
How to search for a pattern? Running time?

Require Suffixes to End at Leaves

Text: **CCAGAT**



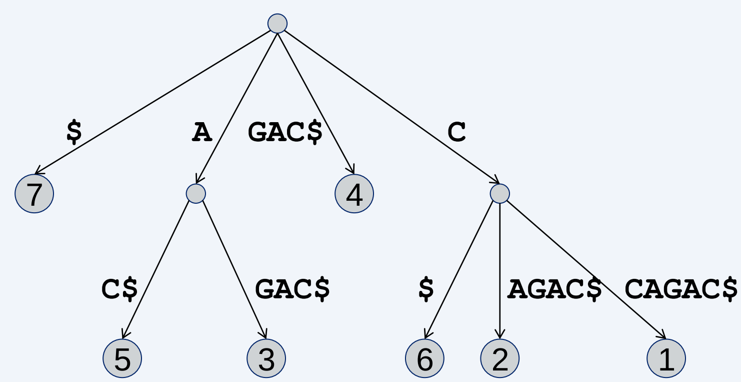
Reason: Simplicity. Distinguish nodes & leaves.

Example text that would break that?

17

Forcing Suffixes to End at Leaves

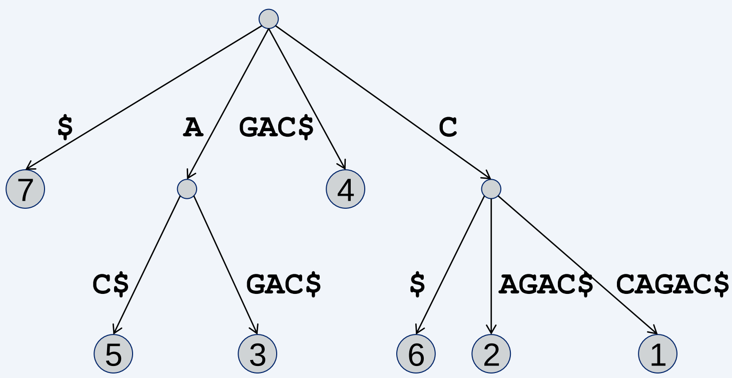
Text: **CCAGAC$**



18

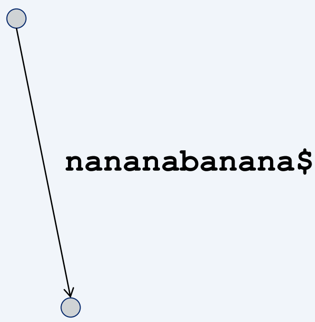
How Would You Create the Tree?

Text: **CCAGAC$**



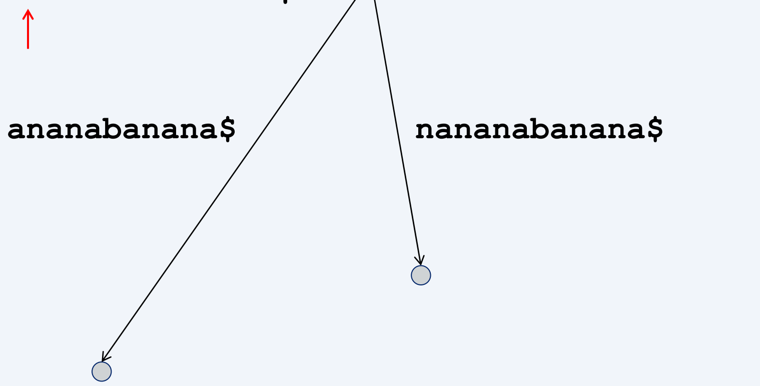
Suffix Tree Construction Example

Text: **nananabanana$**



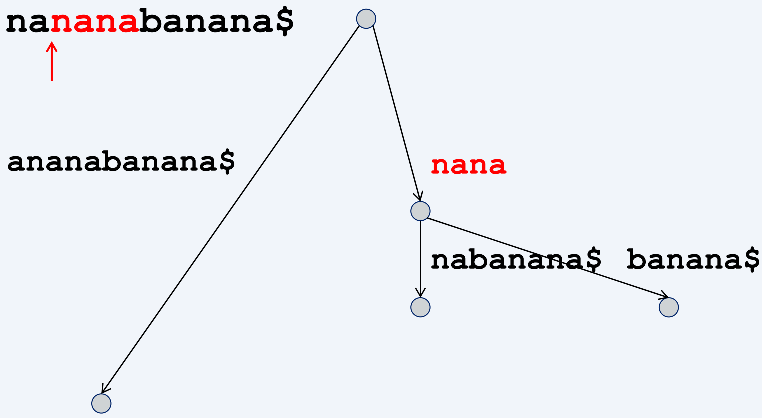
Suffix Tree Construction Example

Text: **nananabanana$**



Suffix Tree Construction Example

Text: **nananabanana$**

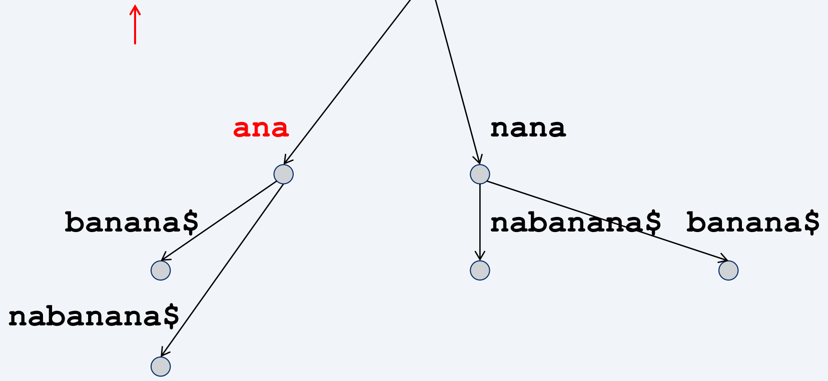


Match **nanabanana$**, **nananabanana$**: 4 comparisons

Suffix Tree Construction Example

Text: **nananabanana$**

↑



Match **anabanana$**, **ananabanana$**: 3 redundant comps

Next …

Match **nabanana$**, **nanabanana$**: 2 redundant comps

Match **abanana$**, **anabanana$**: 1 redundant comp

23

First Algorithm Improvement: No Redundant Matching

When inserting aXY, **nanabanana$**

If we discover aX is a prefix in tree, **nana**

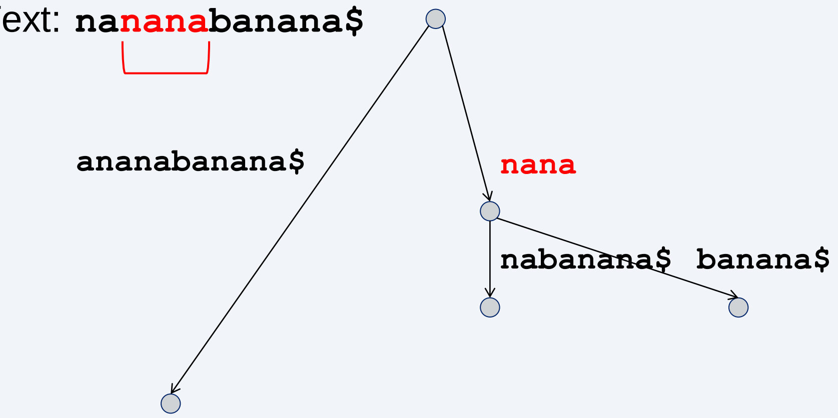
Then X will also be a prefix in tree. **ana**

So, don't bother matching to verify.

24

Suffix Tree Construction Example

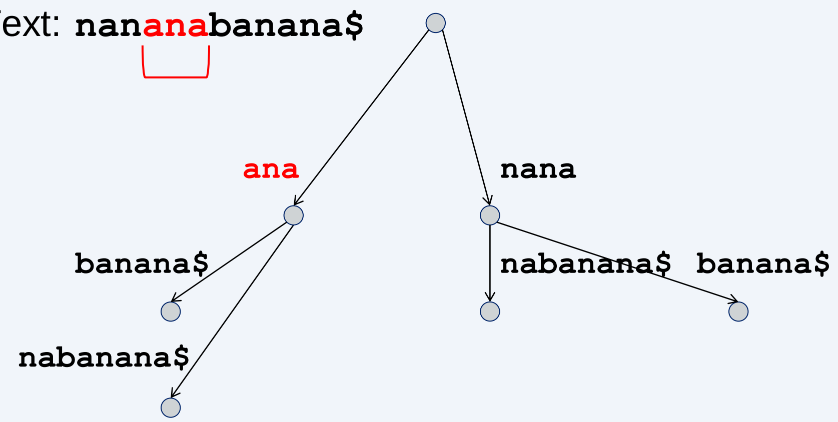
Text: **nananabanana$**



Match **nanabanana$**, **nananabanana$**: 4 comparisons

Suffix Tree Construction Example

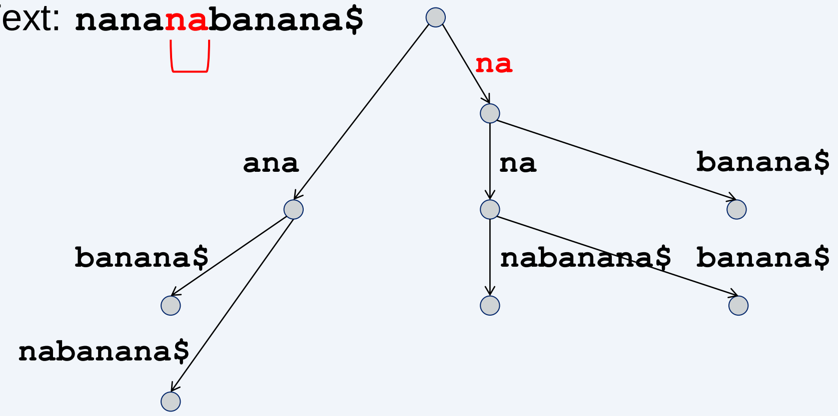
Text: **nananabanana$**



No matching. Just form new node and adjust edge string indices.

Suffix Tree Construction Example

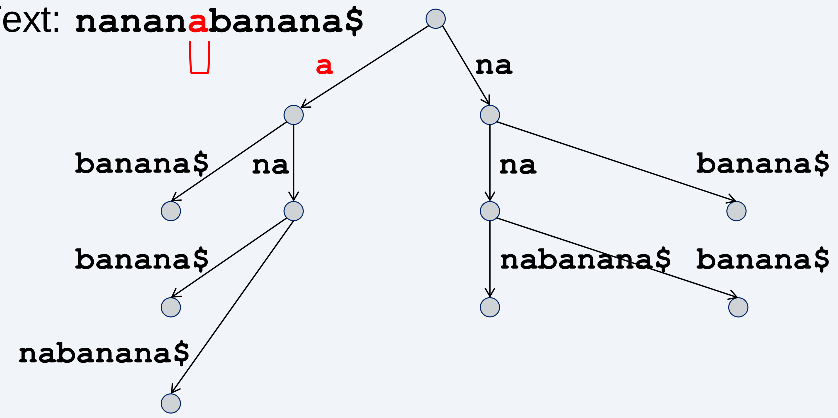
Text: **nananabanana$**



No matching. Just form new node and adjust edge string indices.

Suffix Tree Construction Example

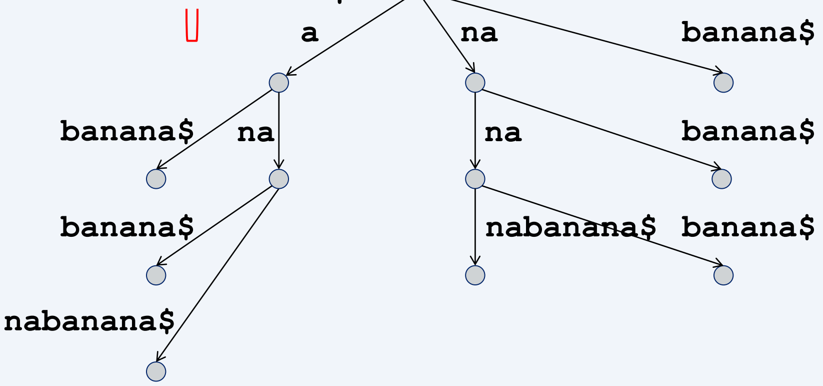
Text: **nananabanana$**



No matching. Just form new node and adjust edge string indices.

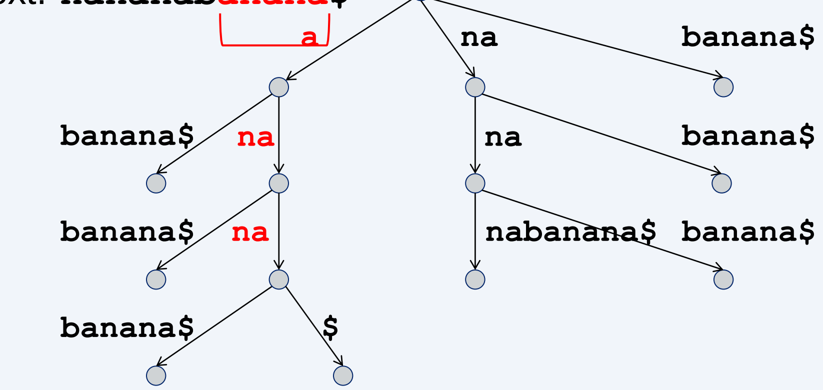
Suffix Tree Construction Example

Text: **nananabanana$**



Suffix Tree Construction Example

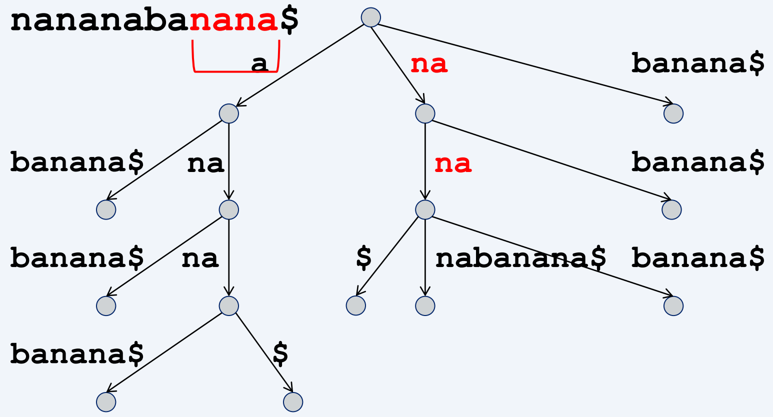
Text: **nananabanana$**



anana is there, but it takes work to match & follow links.

Suffix Tree Construction Example

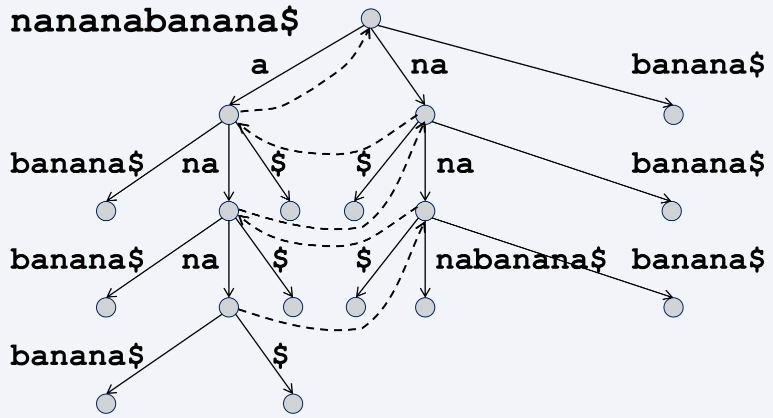
Text: **nananabanana$**



nana is there, but it takes work to match & follow links.

Second Algorithm Improvement: Suffix Links

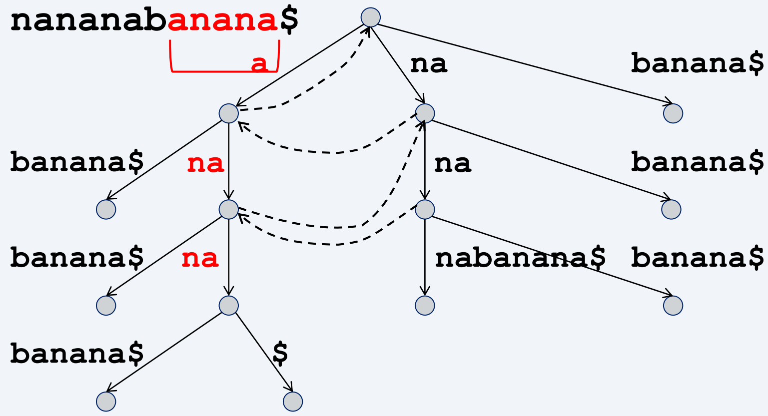
Text: **nananabanana$**



Each internal node for xA has pointer to node for A.

Suffix Tree Construction Example

Text: **nananabanana$**

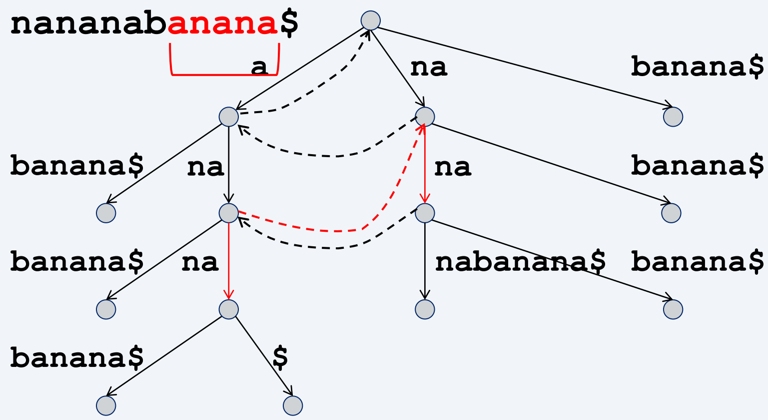


Have to follow links for match on first time.

Need to create suffix link for new node.

Suffix Tree Construction Example

Text: **nananabanana$**

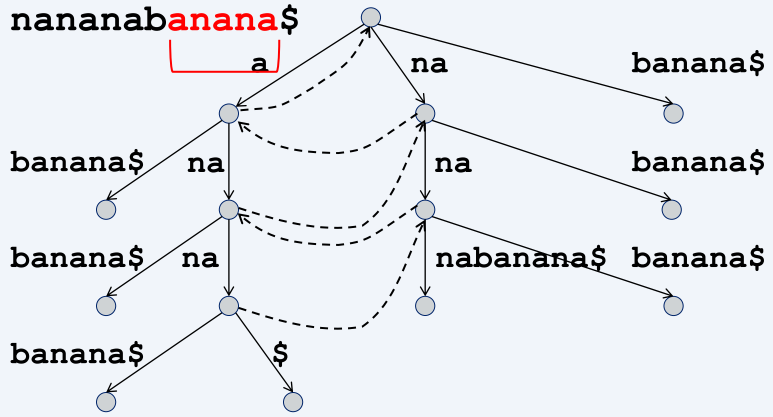


Need to create suffix link for new node.

Use parent's suffix link to get close.

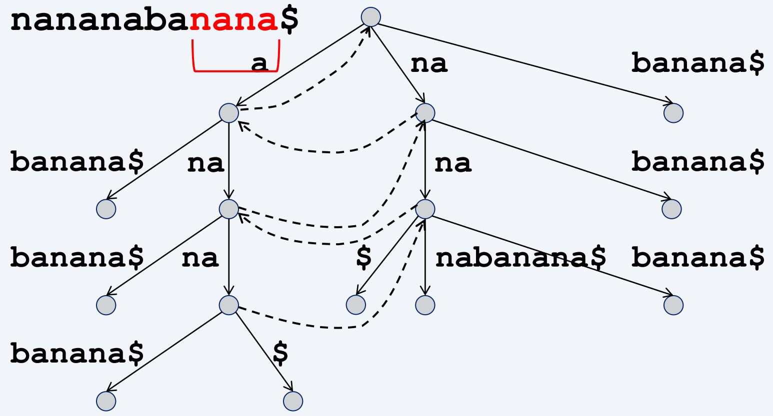
Suffix Tree Construction Example

Text: **nananabanana$**



Suffix Tree Construction Example

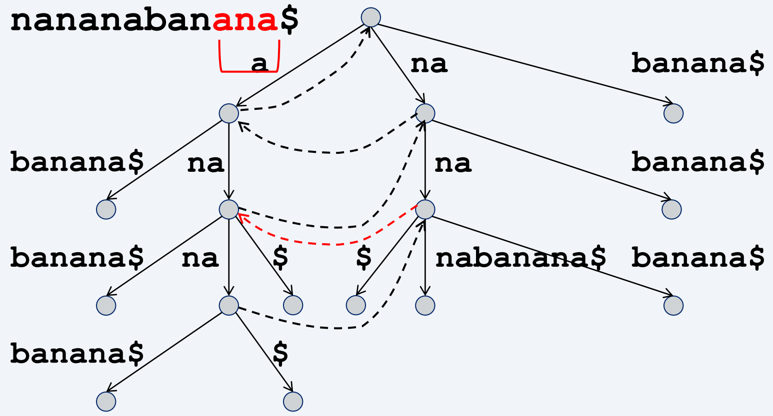
Text: **nananabanana$**



Already found node. No searching or matching.

Suffix Tree Construction Example

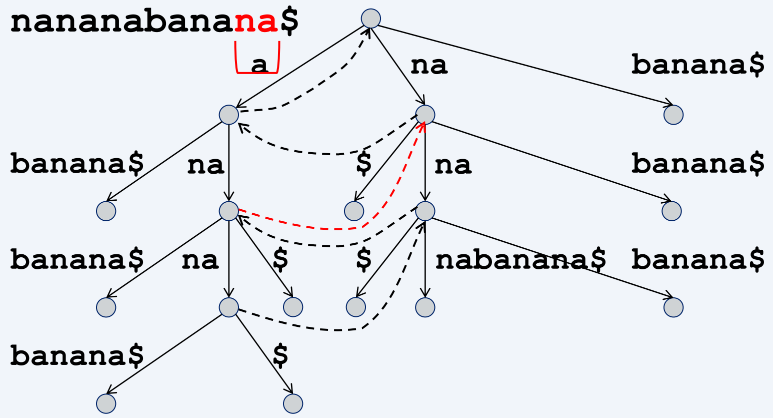
Text: **nananabanana$**



Follow suffix link.

Suffix Tree Construction Example

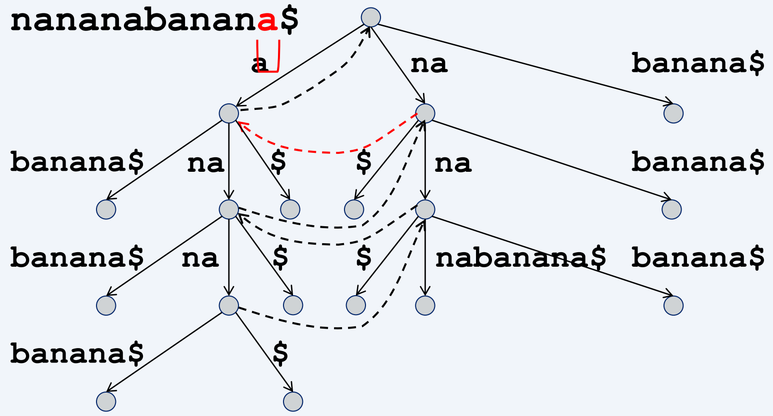
Text: **nananabanana$**



Follow suffix link.

Suffix Tree Construction Example

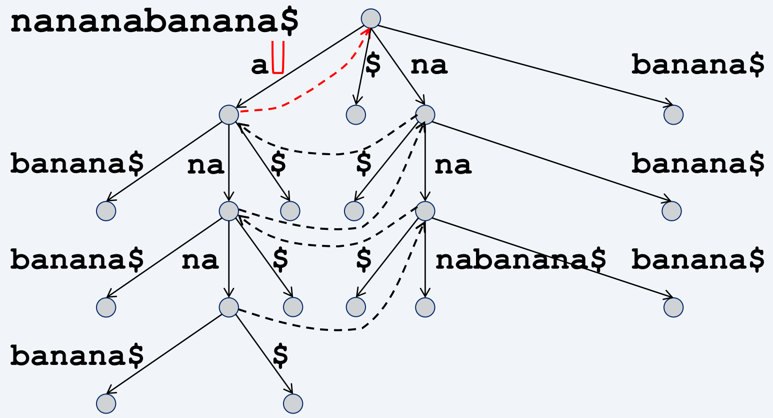
Text: **nananabanana$**



Follow suffix link.

Suffix Tree Construction Example

Text: **nananabanana$**



Done.

$O(c|T|)$, but roughly how big is c?

Almost Correct Analysis of Construction

Two indices: i j

j: Each increment takes $O(1)$ time.

- Just search for one more character.

i: Each increment takes $O(1)$ time.

- Follow suffix link, or from root, get to suffix match.
- Possibly split node & create suffix link.
- Add one edge & leaf.

i, j each incremented n times → $O(n)$ total.

41

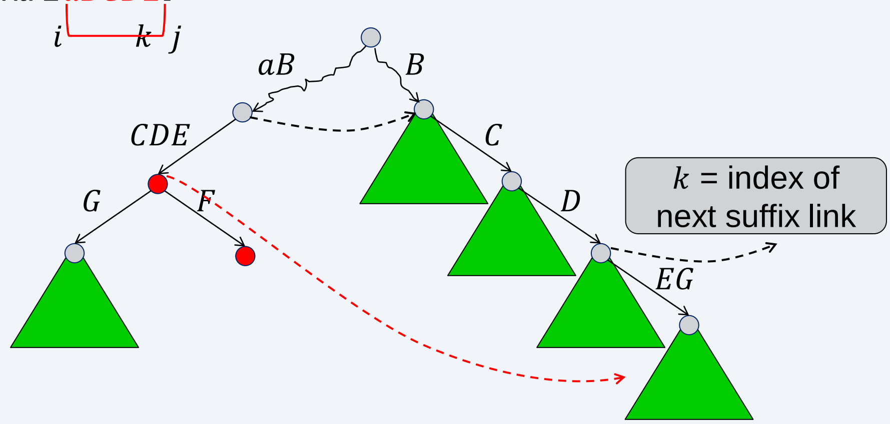
Creating & Following Suffix Links

When creating new node, need new suffix link.

Follow parent's suffix link to get close, then search down.

Text: $Z\color{red}{aBCDE}F$

i k j



42

Correct Analysis of Construction

Three indices: i k j (k not part of alg.)

j: Each increment takes $O(1)$ time.

k: Follow some number l of links in $O(l)$ time. Increments k by at least l.

i: Each increment takes $O(1)$ time in addition to that considered for k.

i, j, k each incremented at most n times → $O(n)$ total.

Some Applications of Suffix Trees

Search for fixed patterns
Search for regular expressions
Find longest common substrings, longest repeated substrings
Find most commonly repeated substrings
Find maximal palindromes
Find Lempel-Ziv decomposition (for text compression)

As used in

Bioinformatics
Data compression
Data clustering

Supplementary Resources

Exact String Matching Algorithms

>30 algorithms, with animations

Course on string matching (Biosequencing)

Wikipedia: Suffix trees

Tutorial on Suffix trees

Tutorial on Suffix trees with applet & code

Notes on string matching and building suffix trees

Suffix tree slides adapted from those by Guy Blelloch, CMU 15-853.