

13

## Text Indexing – Searching entire genomes

3 February 2025

Prof. Dr. Sebastian Wild

## **Learning Outcomes**

#### Unit 13: Text Indexing

- 1. Know and understand methods for text indexing: *inverted indices*, *suffix trees*, (*enhanced*) *suffix arrays*
- 2. Know and understand *generalized suffix trees*
- **3.** Know properties, in particular *performance characteristics*, and limitations of the above data structures.
- **4.** Design (simple) algorithms based on suffix trees.
- **5.** Understand *construction algorithms* for suffix arrays and LCP arrays.

#### **Outline**

## 13 Text Indexing

- 13.1 Motivation
- 13.2 Suffix Trees
- 13.3 Applications
- 13.4 Longest Common Extensions
- 13.5 Suffix Arrays
- 13.6 Linear-Time Suffix Sorting: Overview
- 13.7 Linear-Time Suffix Sorting: The DC3 Algorithm
- 13.8 The LCP Array
- 13.9 LCP Array Construction

# 13.1 Motivation

## **Text indexing**

- ► *Text indexing* (also: *offline text search*):
  - case of string matching: find P[0..m) in T[0..n)
  - ▶ but with *fixed* text  $\rightarrow$  preprocess T (instead of P)
  - $\rightarrow$  expect many queries P, answer them without looking at all of T
  - $\leadsto$  essentially a data structuring problem: "building an *index* of T"

Latin: "one who points out"

- application areas
  - web search engines
  - online dictionaries
  - online encyclopedia
  - ► DNA/RNA data bases
  - ... searching in any collection of text documents (that grows only moderately)

#### **Inverted indices**

same as "indexes"

- ▶ original indices in books: list of (key) words → page numbers where they occur
- ► assumption: searches are only for **whole** (key) **words**
- $\rightsquigarrow \ of ten \ reasonable \ for \ natural \ language \ text$

#### **Inverted indices**

- ▶ original indices in books: list of (key) words → page numbers where they occur
- ► assumption: searches are only for **whole** (key) **words**
- → often reasonable for natural language text

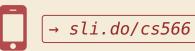
#### Inverted index:

- collect all words in T
  - ► can be as simple as splitting *T* at whitespace
  - actual implementations typically support stemming of words goes → go, cats → cat
- ► store mapping from words to a list of occurrences → how? Symbol tables
  (B5T)

Do you know what a *trie* is?



- (A) A what? No!
- (B) I have heard the term, but don't quite remember.
- C I remember hearing about it in a module.
- D) Sure.



#### **Tries**

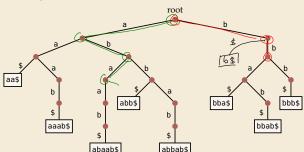
- efficient dictionary data structure for strings
- ▶ name from retrieval, but pronounced "try"
- tree based on symbol comparisons
- ► **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)

some character  $\notin \Sigma$ 

- strings of same length

strings have "end-of-string" marker \$

Example: {aa\$,aaab\$,abaab\$,abb\$, abbab\$, bba\$, bbab\$, bbb\$}



abab\$

Suppose we have a trie that stores n strings over  $\Sigma = \{A, ..., Z\}$ . Each stored string consists of m characters.

We now search for a query string Q with |Q| = q (with  $q \le m$ ). How many **nodes** in the trie are **visited** during this **query**?



F  $\Theta(\log m)$ 

B)  $\Theta(\log(nm))$ 

 $\Theta(q)$ 

C  $\Theta(m \cdot \log n)$ 

 $(H) \Theta(\log q)$ 

 $\bigcap$   $\Theta(q \cdot \log n)$ 

 $\bullet$   $\Theta(m)$ 

 $\int \Theta(q + \log n)$ 



Suppose we have a trie that stores n strings over  $\Sigma = \{A, ..., Z\}$ . Each stored string consists of m characters.

We now search for a query string Q with |Q| = q (with  $q \le m$ ). How many **nodes** in the trie are **visited** during this **query**?



 $(A) \Theta(\log n)$ 

 $\mathbb{F} = \frac{\Theta(\log m)}{2}$ 

B)  $\Theta(\log(nm))$ 

 $G \Theta(q) \checkmark$ 

C)  $\Theta(m - \log n)$ 

 $\Theta(\log q)$ 

 $\bigcirc$   $\bigcirc$   $\bigcirc (m + \log n)$ 

I)  $\Theta(q - \log n)$ 

E) <del>@(m)</del>

 $J = \Theta(q + \log n)$ 



Suppose we have a trie that stores *n* strings over  $\Sigma = \{A, ..., Z\}$ . Each stored string consists of *m* characters. How many **nodes** does the trie have **in total** *in the worst case*?



 $\Theta(n)$ 

 $\Theta(n \log m)$ 

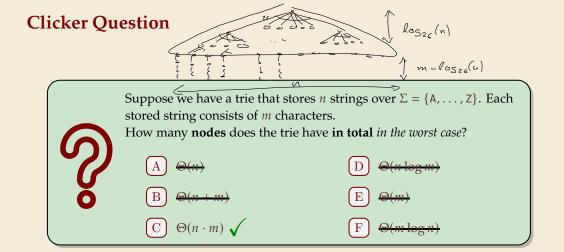
 $\Theta(n+m)$ 

 $\Theta(m)$ 

 $\Theta(n \cdot m)$ 

 $\Theta(m \log n)$ 







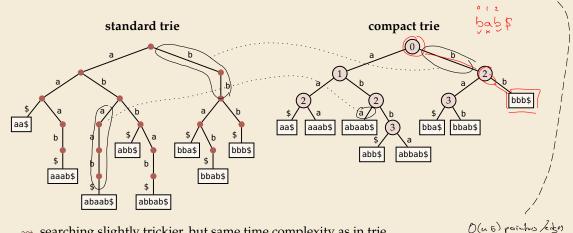
## Compact tries

=1 child

- compress paths of unary nodes into single edge
- ▶ nodes store *index* of next character to check

how to implement nodes? (BST) Ollos 5) time

(2) array chied (0.5) o(1) =



→ searching slightly trickier, but same time complexity as in trie

O(n) # nody ▶ all nodes ≥ 2 children → #nodes ≤ #leaves = #strings → linear space

#### Tries as inverted index

simple

lucene

- fast lookup
- cannot handle more general queries:
  - ▶ search part of a word
  - ► search phrase (sequence of words)

#### Tries as inverted index

- simple
- fast lookup
- cannot handle more general queries:
  - ▶ search part of a word
  - search phrase (sequence of words)
- what if the 'text' does not even have words to begin with?!
  - biological sequences

binary streams

→ need new ideas

## 13.2 Suffix Trees

## Suffix trees – A 'magic' data structure

**Appetizer:** Longest common substring problem

► Given: strings  $S_1, ..., S_k$  Example:  $S_1$  = superiorcalifornializes,  $S_2$  = sealizer

► Goal: find the longest substring that occurs in all *k* strings

## Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- ► Given: strings  $S_1, ..., S_k$  Example:  $S_1$  = superiorcalifornializes,  $S_2$  = sealizer
- ▶ Goal: find the longest substring that occurs in all k strings  $\longrightarrow$  alive



Can we do this in time  $O(|S_1| + \cdots + |S_k|)$ ? How??

## Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

► Given: strings  $S_1, ..., S_k$  Example:  $S_1$  = superiorcalifornializes,  $S_2$  = sealizer

▶ Goal: find the longest substring that occurs in all k strings  $\rightarrow$  alive



Can we do this in time  $O(|S_1| + \cdots + |S_k|)$ ? How??

#### Enter: *suffix trees*

- versatile data structure for index with full-text search
- ▶ linear time (for construction) and linear space
- ▶ allows efficient solutions for many advanced string problems



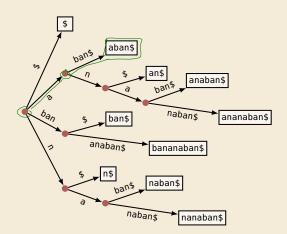
"Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible." [Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]

suffix tree T for text T = T[0..n) = compact trie of all suffixes of <math>T\$ (set T[n] := \$)

▶ suffix tree  $\mathcal{T}$  for text T = T[0..n) = compact trie of all suffixes of T\$ (set <math>T[n] := \$)

#### **Example:**

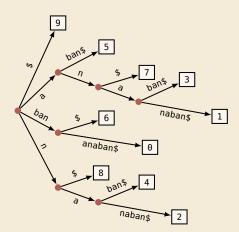
T = bananaban\$



- suffix tree T for text T = T[0..n) = compact trie of all suffixes of T\$ (set <math>T[n] := \$)
- except: in leaves, store *start index* (instead of copy of actual string)

#### **Example:**

T = bananaban\$

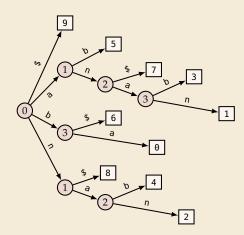


- suffix tree T for text T = T[0..n) = compact trie of all suffixes of T\$ (set <math>T[n] := \$)
- except: in leaves, store start index (instead of copy of actual string)

#### **Example:**

T = bananaban\$

- ▶ also: edge labels like in compact trie
- (more readable form on slides to explain algorithms)



#### **Suffix trees – Construction**

- ► T[0..n] has n + 1 suffixes (starting at character  $i \in [0..n]$ )
- ▶ We can build the suffix tree by inserting each suffix of T into a compressed trie. But that takes time  $\Theta(n^2)$ .  $\leadsto$  not interesting!

#### **Suffix trees – Construction**

- ► T[0..n] has n + 1 suffixes (starting at character  $i \in [0..n]$ )
- ▶ We can build the suffix tree by inserting each suffix of T into a compressed trie. But that takes time  $\Theta(n^2)$ .  $\leadsto$  not interesting!



same order of growth as reading the text!

**Amazing result:** Can construct the suffix tree of T in  $\Theta(n)$  time!

- algorithms are a bit tricky to understand
- but were a theoretical breakthrough
- ▶ and they are efficient in practice (and heavily used)!

→ for now, take linear-time construction for granted. What can we do with them?

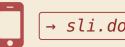
**Recap:** Check all correct statements about suffix tree  $\mathbb{T}$  of T[0..n].

- (A) We require T to end with \$.
- B The size of  $\mathcal{T}$  can be  $\Omega(n^2)$  in the worst case.
- $\bigcirc$  T is a standard trie of all suffixes of T\$.
- D T is a compact trie of all suffixes of T\$.
- $oxed{E}$  The leaves of  $\mathcal{T}$  store (a copy of) a suffix of T\$.
- F Naive construction of  $\mathcal{T}$  takes  $\Omega(n^2)$  (worst case).
- G T can be computed in O(n) time (worst case).
- (H) Thas n leaves.



**Recap:** Check all correct statements about suffix tree  $\Im$  of T[0..n].

- We require T to end with  $\$^{2}$
- The size of T can be  $\Omega(n^2)$  in the worst case.
- $\mathcal{T}$  is a compact trie of all suffixes of T\$.
- The leaves of T store (a copy of) a suffix of T\$.
  - Naive construction of T takes  $\Omega(n^2)$  (worst case).
- $\mathcal{T}$  can be computed in O(n) time (worst case).



## 13.3 Applications

## **Applications of suffix trees**

ana O(q)

▶ In this section, always assume suffix tree  $\Im$  for T given.

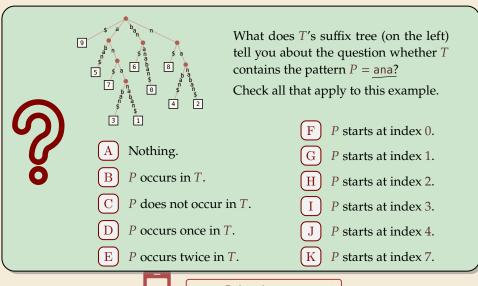
**Recall:** T stored like this:

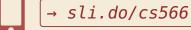
\$ a b n 1 1 2 5 2 6 0 8 2 5 4 2 5 7 3 4 2 2 5 5 7 3 1 T = bananaban\$

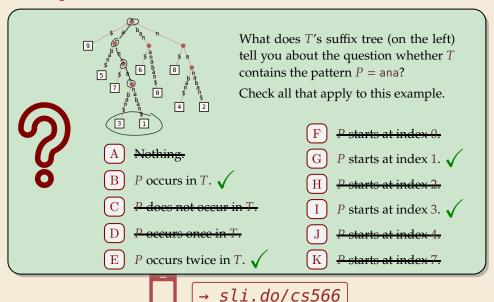
but think about this:



- ▶ Moreover: assume internal nodes store pointer to *leftmost leaf in subtree*.
- ► Notation:  $T_i = T[i..n]$  (including \$)

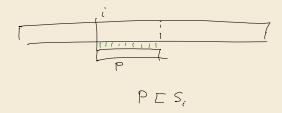






## **Application 1: Text Indexing / String Matching**

- ▶ we have all suffixes in T!



## Application 1: Text Indexing / String Matching

- ▶ P occurs in T  $\iff$  P is a prefix of a suffix of T
- ▶ we have all suffixes in T!
- $\rightsquigarrow$  (try to) follow path with label P, until

 $\rightarrow$  P does not occur in T

- we get stuck
   at internal node (no node with next character of P)
   or inside edge (mismatch of next characters)
  - 2. we run out of pattern

reach end of P at internal node v or inside edge towards  $v \rightarrow P$  occurs at all leaves in subtree of v

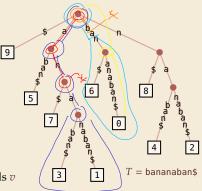
3. we run out of tree

reach a leaf  $\ell$  with part of P left  $\rightsquigarrow$  compare P to  $\ell$ .



This cannot happen when testing edge labels since  $\xi \notin \Sigma$ , but needs check(s) in compact trie implementation!

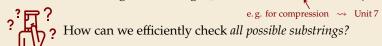
Finding first match (or NO\_MATCH) takes O(|P|) time!



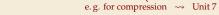
#### **Examples:**

- ► P = ann wot in T
- ightharpoonup P = baa
- ightharpoonup P = ana
- ightharpoonup P = ba
- ightharpoonup P = briar

▶ **Goal:** Find longest substring  $T[i..i + \ell)$  that occurs also at  $j \neq i$ :  $T[j..j + \ell) = T[i..i + \ell)$ .

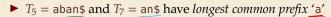


▶ **Goal:** Find longest substring  $T[i..i + \ell)$  that occurs also at  $j \neq i$ :  $T[j..j + \ell) = T[i..i + \ell)$ .



? How can we efficiently check all possible substrings?

- Repeated substrings = shared paths in *suffix tree* 

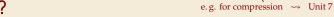


→ ∃ internal node with path label 'a

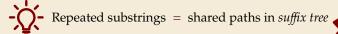
here single edge, can be longer path

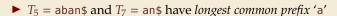


▶ **Goal:** Find longest substring  $T[i..i + \ell)$  that occurs also at  $j \neq i$ :  $T[j..j + \ell) = T[i..i + \ell)$ .



How can we efficiently check all possible substrings?



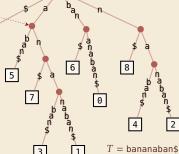


→ ∃ internal node with path label 'a'

here single edge, can be longer path

→ longest repeated substring = longest common prefix (LCP) of two suffixes

actually: adjacent leaves

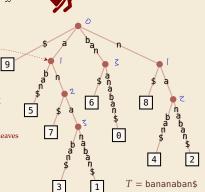


▶ **Goal:** Find longest substring  $T[i..i + \ell)$  that occurs also at  $j \neq i$ :  $T[j..j + \ell) = T[i..i + \ell)$ .

- e.g. for compression → Unit 7
- ? How can we efficiently check all possible substrings?
- Repeated substrings = shared paths in *suffix tree*
- ►  $T_5$  = aban\$ and  $T_7$  = an\$ have longest common prefix 'a'
- → ∃ internal node with path label 'a'

here single edge, can be longer path

- → longest repeated substring = longest common prefix
  (LCP) of two suffixes
  - (LCP) of two suffixes
- ► Algorithm:
  - 1. Compute *string depth* (=length of path label) of nodes
  - 2. Find internal nodes with maximal string depth
- ▶ Both can be done in depth-first traversal  $\rightsquigarrow$   $\Theta(n)$  time



#### Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings  $T^{(1)}, ..., T^{(k)}$  with  $T^{(j)} \in \Sigma^{n_j}$
- ► can we solve that in the same way?
- ightharpoonup could build the suffix tree for each  $T^{(j)}$  ... but doesn't seem to help

#### Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings  $T^{(1)}, ..., T^{(k)}$  with  $T^{(j)} \in \Sigma^{n_j}$
- ► can we solve that in the same way?
- ightharpoonup could build the suffix tree for each  $T^{(j)}$  ... but doesn't seem to help
- → need a *single/joint* suffix tree for *several* texts

#### Generalized suffix trees

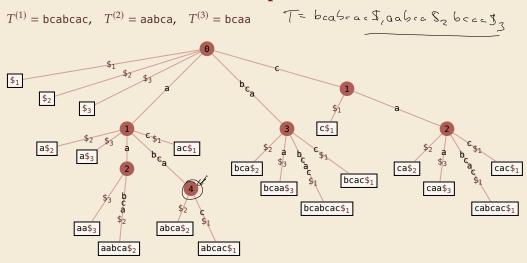
- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings  $T^{(1)}, ..., T^{(k)}$  with  $T^{(j)} \in \Sigma^{n_j}$
- can we solve that in the same way?
- ightharpoonup could build the suffix tree for each  $T^{(j)}$  ... but doesn't seem to help
- → need a *single/joint* suffix tree for *several* texts

#### Enter: generalized suffix tree

- ▶ Define  $T := T^{(1)} \$_1 T^{(2)} \$_2 \cdots T^{(k)} \$_k$  for k new end-of-word symbols
- ightharpoonup Construct suffix tree T for T
- $\Rightarrow$  \$j-edges always leads to leaves  $\Rightarrow$   $\exists$  leaf (j,i) for each suffix  $T_i^{(j)} = T^{(j)}[i..n_j]$



## **Generalized Suffix Tree – Example**



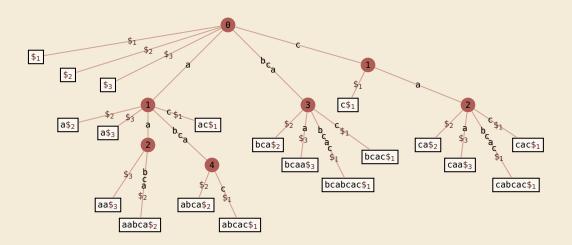
## **Application 3: Longest common substring**

- ▶ With that new idea, we can find longest common substrings:
  - **1.** Compute generalized suffix tree  $\mathcal{T}$ .
  - 2. Store with each node the *subset of strings* that contain its path label:
    - **2.1.** Traverse T bottom-up.
    - **2.2**. For a leaf (j, i), the subset is  $\{j\}$ .
    - **2.3**. For an internal node, the subset is the union of its children.
  - 3. In top-down traversal, compute *string depths* of nodes. (as above)
  - **4.** Report deepest node (by string depth) whose subset is  $\{1, \ldots, k\}$ .
- ▶ Each step takes time  $\Theta(n)$  for  $n = n_1 + \cdots + n_k$  the total length of all texts.

<sup>&</sup>quot;Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible." [Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]

## **Longest common substring – Example**

$$T^{(1)}=$$
 bcabcac,  $T^{(2)}=$  aabca,  $T^{(3)}=$  bcaa



## **Longest common substring – Example**

$$T^{(1)} = bca\underline{bcac}$$
,  $T^{(2)} = aa\underline{bca}$ ,  $T^{(3)} = \underline{bcaa}$ 

