6 Text Indexing – Searching entire genomes

7 November 2022

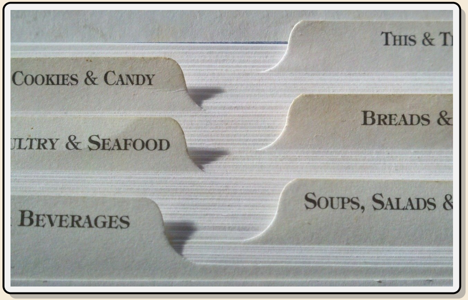
Sebastian Wild

COMP526 (Fall 2022)
University of Liverpool
version `2022-11-07 13:44`

Learning Outcomes

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.

Unit 6: ***Text Indexing***



Outline

6 Text Indexing

6.1 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 $\rightsquigarrow$ preprocess T (instead of P)
  - $\rightsquigarrow$ expect many queries P, answer them without looking at all of T
  - $\rightsquigarrow$ 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 $\mapsto$ page numbers where they occur
- assumption: searches are only for **whole** (key) **words**

$\rightsquigarrow$ often reasonable for natural language text

Inverted indices

- original indices in books: list of (key) words $\mapsto$ page numbers where they occur
  same as "indexes"
- assumption: searches are only for **whole** (key) **words**

$\rightsquigarrow$ 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` $\rightarrow$ `go`, `cats` $\rightarrow$ `cat`
- store mapping from words to a list of occurrences $\rightsquigarrow$ *how?*

dictionary BST $\rightsquigarrow$ $O(\log n)$

keys = words

values = lists of offsets / occurrences

Clicker Question

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.

→ sli.do/comp526

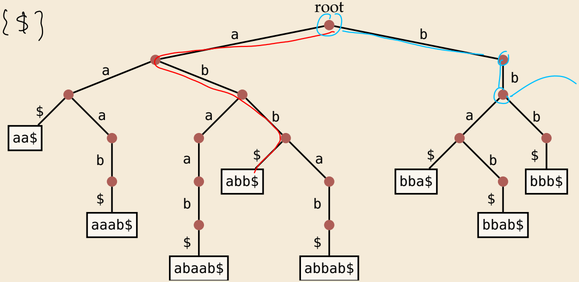
Tries

- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

- **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
  - strings of same length
  - strings have "end-of-string" marker $ (some character $\notin \Sigma$)

$\Sigma = \{a, b\} \cup \{\$\}$

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



Is bb$ in the set?

No

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. 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 $\Theta(\log n)$

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

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

D $\Theta(m + \log n)$

E $\Theta(m)$

F $\Theta(\log m)$

G $\Theta(q)$

H $\Theta(\log q)$

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

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

→ sli.do/comp526

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
We now search for a query string Q with $|Q| = q$ (with $q \leq m$).
How many **nodes** in the trie are **visited** during this **query**?

- **A** ~~$\Theta(\log n)$~~
- **B** ~~$\Theta(\log(nm))$~~
- **C** ~~$\Theta(m \cdot \log n)$~~
- **D** ~~$\Theta(m + \log n)$~~
- **E** ~~$\Theta(m)$~~
- **F** ~~$\Theta(\log m)$~~
- **G** $\Theta(q)$ ✓
- **H** ~~$\Theta(\log q)$~~
- **I** ~~$\Theta(q \cdot \log n)$~~
- **J** ~~$\Theta(q + \log n)$~~

→ sli.do/comp526

Clicker Question

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

A $\Theta(n)$

B $\Theta(n+m)$

C $\Theta(n \cdot m)$

D $\Theta(n \log m)$

E $\Theta(m)$

F $\Theta(m \log n)$

→ *sli.do/comp526*

Clicker Question

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

A ~~$\Theta(n)$~~

B ~~$\Theta(n + m)$~~

C $\Theta(n \cdot m)$ ✓

D ~~$\Theta(n \log m)$~~

E ~~$\Theta(m)$~~

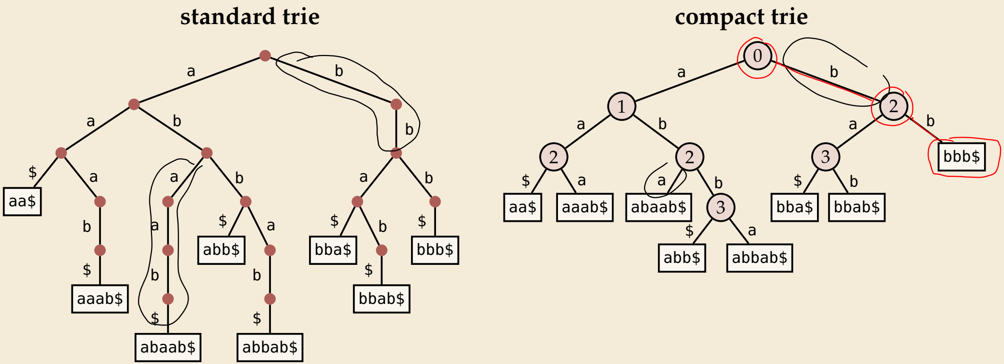
F ~~$\Theta(m \log n)$~~

→ sli.do/comp526

Compact tries

- compress paths of unary nodes into single edge =1 child
- nodes store *index* of next character to check

bab$



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

- all nodes ≥ 2 children ⇝ #nodes $\leq$ #leaves $=$ #strings ⇝ linear space $\Theta(n)$

Tries as inverted index

- simple
- 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

    ```
    ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGC
    CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGC
    CAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCCAGCAATCCGCGCGCCGGGACAGAA
    TGCCCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
    ```

  - binary streams

    ```
    00000010101001111010111000001111100011111011111001101101000011100010011011110000010001101010
    01101100001101011010000000100000000111010110000010000111101011101100100011001011011101111111
    11000101000101100101000000111010101001100000000110110000110011111000010 1 0101011101111000011
    1010111001001010101010000001111101001100000011110011010100000000100100100000101100011000110111
    ```

$\leadsto$ need new ideas

6.2 Suffix Trees

Suffix trees – A ‘magic’ data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$ **Example:** S_1 = `superiorcalifornialives`, S_2 = `sealiver`
- Goal: find the longest substring that occurs in all k strings

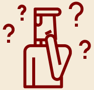
Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$
- Goal: find the longest substring that occurs in all k strings

Example: S_1 = `superiorcalifornialives`, S_2 = `sealiver`

$\rightsquigarrow$ `alive`

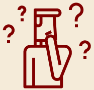


Can we do this in time $O(|S_1| + \cdots + |S_k|)$? How??

Suffix trees – A 'magic' data structure

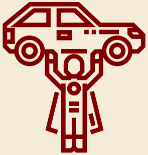
Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$ **Example:** $S_1 =$ `superiorcalifornialives`, $S_2 =$ `sealiver`
- Goal: find the longest substring that occurs in all k strings $\rightsquigarrow$ `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 trees – Definition

- $\boxed{\text{suffix tree } \mathcal{T} \text{ for text } T = T[0..n) \quad = \quad \text{compact trie of all suffixes of } T\$ \quad (\text{set } T[n] := \$)}$

Suffix trees – Definition

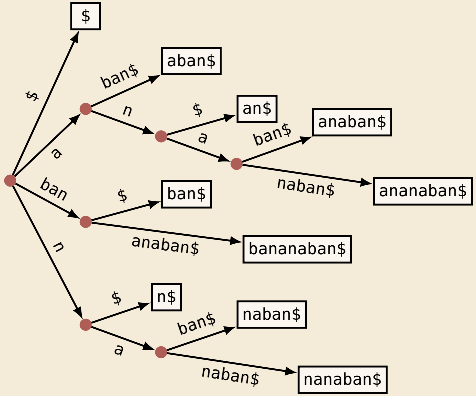
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ $=$ compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

Example:

T = bananaban$

suffixes: {bananaban$, ananaban$, nanaban$, anaban$, naban$, aban$, ban$, an$, n$, $}

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$



Suffix trees – Definition

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

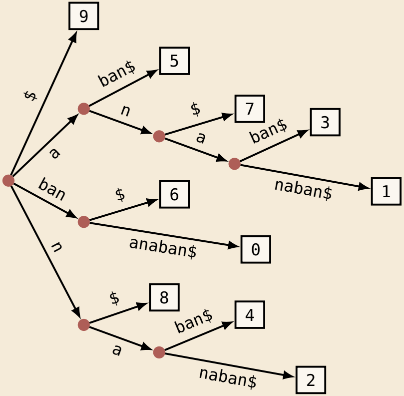
Example:

T = bananaban$

suffixes: {bananaban$, ananaban$, nanaban$, anaban$, naban$, aban$, ban$, an$, n$, $}

T =

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$



Suffix trees – Definition

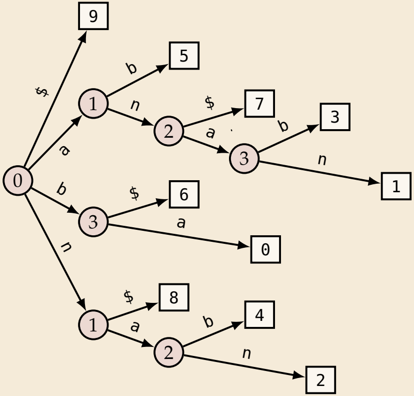
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

Example:

T = bananaban$

suffixes: {bananaban$, ananaban$, nanaban$,
anaban$, naban$, aban$, ban$, an$, n$, $}

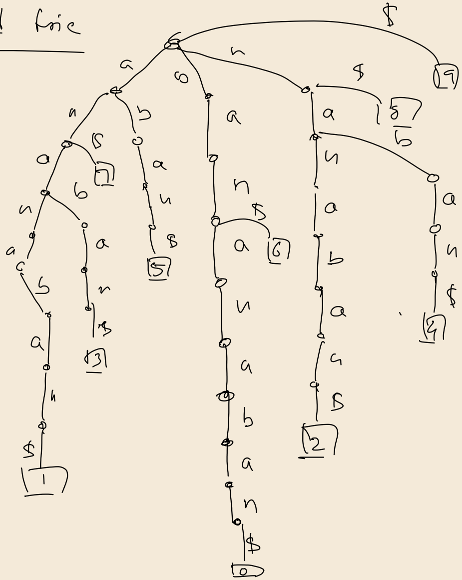
	0	1	2	3	4	5	6	7	8	9
T =	b	a	n	a	n	a	b	a	n	$



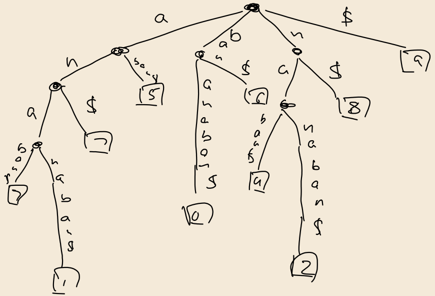
- also: edge labels like in compact trie
- (more readable form on slides to explain algorithms)

bananaban$

standard trie



compact trie



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 $\Theta(n^2)$. $\rightsquigarrow$ 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 $\Theta(n^2)$. $\rightsquigarrow$ 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)!

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

Clicker Question

Recap: Check all correct statements about suffix tree $\mathcal{T}$ 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.
- **C** $\mathcal{T}$ is a standard trie of all suffixes of $T\$$.
- **D** $\mathcal{T}$ is a compact trie of all suffixes of $T\$$.
- **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** $\mathcal{T}$ can be computed in $O(n)$ time (worst case).
- **H** $\mathcal{T}$ has n leaves.

→ sli.do/comp526

Clicker Question

Recap: Check all correct statements about suffix tree $\mathcal{T}$ 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.~~ $\Theta(n)$ space
- **C** ~~$\mathcal{T}$ is a standard trie of all suffixes of $T\$$.~~
- **D** $\mathcal{T}$ is a compact trie of all suffixes of $T\$$. ✓
- **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** $\mathcal{T}$ can be computed in $O(n)$ time (worst case). ✓
- **H** ~~$\mathcal{T}$ has n leaves.~~

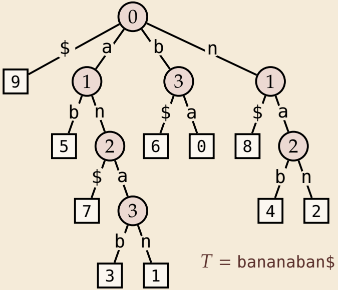
→ sli.do/comp526

6.3 Applications

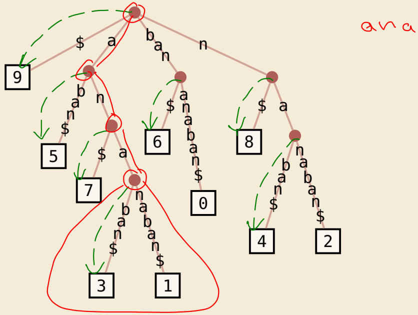
Applications of suffix trees

- In this section, always assume suffix tree $\mathcal{T}$ for T given.

Recall: $\mathcal{T}$ stored like this: but think about this:

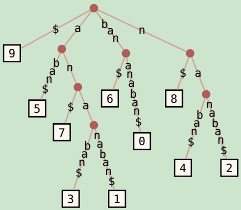


T = bananaban$



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

Clicker Question



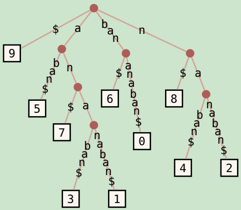
What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = `ana`?

Check all that apply to this example.

- **A** Nothing.
- **B** P occurs in T.
- **C** P does not occur in T.
- **D** P occurs once in T.
- **E** P occurs twice in T.
- **F** P starts at index 0.
- **G** P starts at index 1.
- **H** P starts at index 2.
- **I** P starts at index 3.
- **J** P starts at index 4.
- **K** P starts at index 7.

→ `sli.do/comp526`

Clicker Question



What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = `ana`?

Check all that apply to this example.

A ~~Nothing.~~

B P occurs in T. ✓

C ~~P does not occur in T.~~

D ~~P occurs once in T.~~

E P occurs twice in T. ✓

F ~~P starts at index 0.~~

G P starts at index 1. ✓

H ~~P starts at index 2.~~

I P starts at index 3. ✓

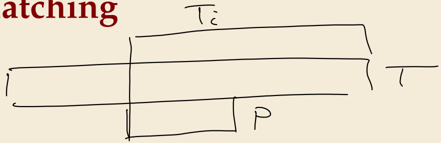
J ~~P starts at index 4.~~

K ~~P starts at index 7.~~

→ sli.do/comp526

Application 1: Text Indexing / String Matching

- $\boxed{P \text{ occurs in } T \iff P \text{ is a prefix of a suffix of } T}$
- we have all suffixes in $\mathcal{T}$!



Application 1: Text Indexing / String Matching

- $\boxed{P \text{ occurs in } T \iff P \text{ is a prefix of a suffix of } T}$
- we have all suffixes in $\mathcal{T}$!

$\rightsquigarrow$ (try to) follow path with label P, until

1. **we get stuck**

   nb at *internal node* (no node with next character of P)

   baa or *inside edge* (mismatch of next characters)

   $\rightsquigarrow$ P does not occur in T

2. **we run out of pattern** ba

   ana reach end of P at internal node v or inside edge towards v

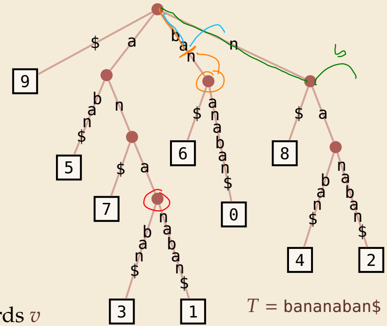
   $\rightsquigarrow$ P occurs at all leaves in subtree of v

3. **we run out of tree**

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

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

   not possible for text indexing if \$ not in P



- Finding first match (or NO_MATCH) takes $O(|P|)$ time!

Application 1: Text Indexing / String Matching

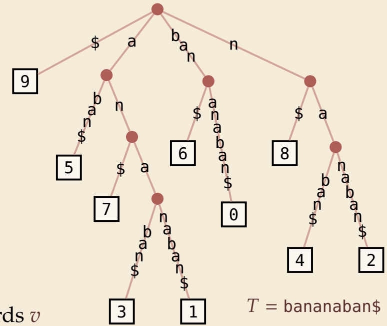
- $\boxed{P \text{ occurs in } T \iff P \text{ is a prefix of a suffix of } T}$
- we have all suffixes in $\mathcal{T}$!

⇝ (try to) follow path with label P, until

1. **we get stuck**
   at internal node (no node with next character of P)
   or *inside edge* (mismatch of next characters)
   ⇝ P does not occur in T
2. **we run out of pattern**
   reach end of P at internal node v or inside edge towards v
   ⇝ P occurs at all leaves in subtree of v
3. **we run out of tree**
   reach a leaf ℓ with part of P left ⇝ compare P to ℓ.

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

- Finding first match (or `NO_MATCH`) takes $O(|P|)$ time!



T = `bananaban$`

Examples:

- P = `ann`
- P = `baa`
- P = `ana`
- P = `ba`
- P = `briar`

Application 2: Longest repeated substring

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

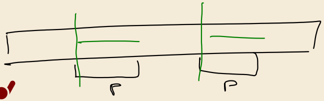
e. g. for compression $\rightsquigarrow$ Unit 7

How can we efficiently check *all possible substrings?*

Application 2: Longest repeated substring

- **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 $\rightsquigarrow$ Unit 7



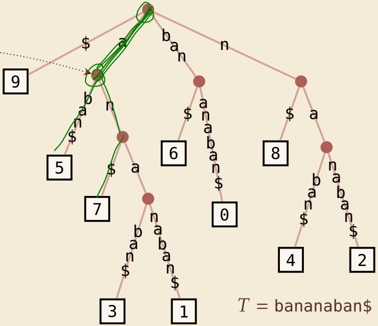
How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* ‘`a`’

$\rightsquigarrow$ $\exists$ internal node with path label ‘`a`’

here single edge, can be longer path



T = `bananaban$`

Application 2: Longest repeated substring

- **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 $\rightsquigarrow$ Unit 7

How can we efficiently check *all possible substrings?*

Repeated substrings $=$ shared paths in *suffix tree*

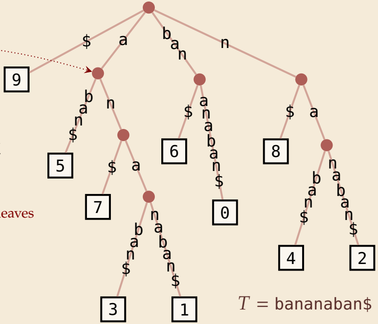
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* ‘`a`’

$\rightsquigarrow$ $\exists$ internal node with path label ‘`a`’

here single edge, can be longer path

$\rightsquigarrow$ longest repeated substring $=$ longest common prefix (LCP) of two suffixes

actually: adjacent leaves



T = `bananaban$`

Application 2: Longest repeated substring

- **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

bananaban

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

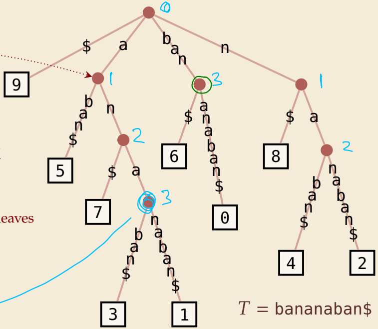
- T_5 = aban\$ and T_7 = an\$ have *longest common prefix* 'a'

⇝ ∃ internal node with path label 'a'

here single edge, can be longer path

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

actually: adjacent leaves



T = bananaban\$

- 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 ⇝ $\Theta(n)$ time

Generalized suffix trees

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

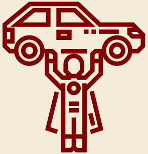
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts

Generalized suffix trees

- longest *repeated* substring (of one string) feels very similar to
  longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
- can we solve that in the same way?
- 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 $\boxed{T := T^{(1)}\$_1 T^{(2)}\$_2 \cdots T^{(k)}\$_k}$ for k new end-of-word symbols
- Construct suffix tree $\mathcal{T}$ for T

⇝ $\$_j$-edges always leads to leaves ⇝ $\exists$ leaf (j, i) for each suffix $T_i^{(j)} = T^{(j)}[i..n_j]$

Clicker Question

What is the longest common substring of the strings `bcabcac`, `aabca` and `bcaa`?

→ *sli.do/comp526*

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 $\mathcal{T}$ 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)

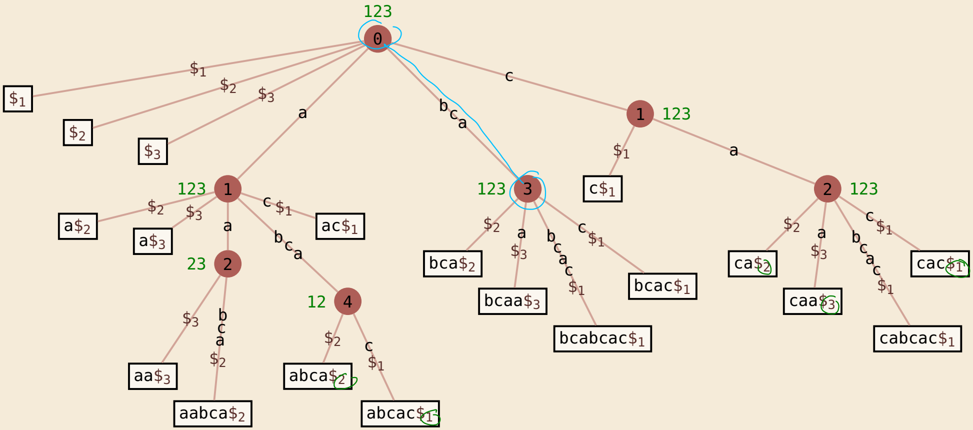
  4. Report deepest node (by string depth) whose subset is $\{1, \ldots, k\}$.

- Each step takes time $\Theta(n)$ for $n = n_1 + \cdots + n_k$ the total length of all texts.

"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

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



6.4 Longest Common Extensions

Application 4: Longest Common Extensions

- We implicitly used a special case of a more general, versatile idea:

  The ***longest common extension (LCE)*** data structure:

  - **Given:** String $T[0..n)$
  - **Goal:** Answer LCE queries, i. e.,
    given positions i, j in T,
    how far can we read the same text from there?
    formally: $\mathrm{LCE}(i, j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

trivial solution: "store-nothing approach"

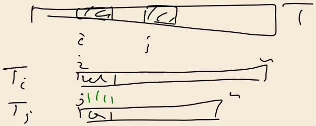
LCE: compare chars in a loop

w.c. $T = a^n \$$ time $\Theta(n)$

Application 4: Longest Common Extensions

- We implicitly used a special case of a more general, versatile idea:

  The ***longest common extension (LCE)*** data structure:
  - **Given:** String $T[0..n)$
  - **Goal:** Answer LCE queries, i. e.,
    given positions i, j in T,
    how far can we read the same text from there?
    formally: $\mathrm{LCE}(i,j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$



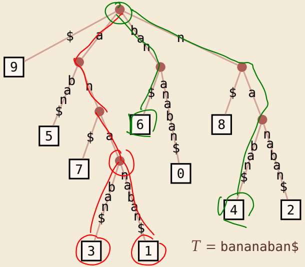
$LCE(1,3) = 3$

$LCE(6,4) = 0$

$\rightsquigarrow$ use suffix tree of T!

(length of) longest common prefix of ith and jth suffix

- In $\mathcal{T}$: $\mathrm{LCE}(i,j) = \mathrm{LCP}(T_i, T_j)$ $\rightsquigarrow$ same thing, different name!
  $=$ string depth of *lowest common ancester (LCA)* of leaves $\boxed{i}$ and $\boxed{j}$



- in short: $\boxed{\mathrm{LCE}(i,j) = \mathrm{LCP}(T_i, T_j) = \mathrm{stringDepth}(\mathrm{LCA}(\boxed{i}, \boxed{j}))}$

Efficient LCA

How to find lowest common ancestors?

- Could walk up the tree to find LCA $\rightsquigarrow$ $\Theta(n)$ worst case 👎
- Could store all LCAs in big table $\rightsquigarrow$ $\Theta(n^2)$ space and preprocessing 👎

Efficient LCA

How to find lowest common ancestors?

- Could walk up the tree to find LCA $\rightsquigarrow$ $\Theta(n)$ worst case
- Could store all LCAs in big table $\rightsquigarrow$ $\Theta(n^2)$ space and preprocessing

Amazing result: Can compute data structure in $\Theta(n)$ time and space that finds any LCA is **constant(!) time**.

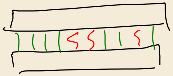
- a bit tricky to understand
- but a theoretical breakthrough
- and useful in practice

and suffix tree construction inside . . .

$\rightsquigarrow$ for now, use $O(1)$ LCA as black box.

$\rightsquigarrow$ After linear preprocessing (time & space), we can find LCEs in $O(1)$ time.

Application 5: Approximate matching



HD 3

k-mismatch matching:

- **Input:** text $T[0..n)$, pattern $P[0..m)$, $k \in [0..m)$
- **Output:**

  "Hamming distance $\leq k$"

  - smallest i so that $T[i..i+m)$ are P differ in at most k characters
  - or `NO_MATCH` if there is no such i

$\rightsquigarrow$ searching with typos

- Adapted brute-force algorithm $\rightsquigarrow$ $O(n \cdot m)$

Application 5: Approximate matching

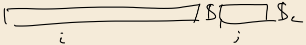
k-mismatch matching:

- **Input:** text $T[0..n)$, pattern $P[0..m)$, $k \in [0..m)$
- **Output:**

  "Hamming distance $\leq k$"

  - smallest i so that $T[i..i+m)$ are P differ in at most k characters
  - or `NO_MATCH` if there is no such i

$\rightsquigarrow$ searching with typos



- Adapted brute-force algorithm $\rightsquigarrow$ $O(n \cdot m)$

- Assume longest common extensions in $T\$_1P\$_2$ can be found in $O(1)$
  - $\rightsquigarrow$ generalized suffix tree $\mathcal{T}$ has been built
  - $\rightsquigarrow$ string depths of all internal nodes have been computed
  - $\rightsquigarrow$ constant-time LCA data structure for $\mathcal{T}$ has been built

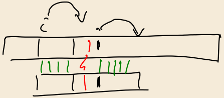
Clicker Question

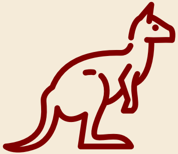
What is the Hamming distance between `heart` and `beard`?

beard

→ *sli.do/comp526*

Kangaroo Algorithm for approximate matching





```
procedure kMismatch(T[0..n − 1], P[0..m − 1])
    // build LCE data structure
    for i := 0, . . . , n − m − 1 do
        mismatches := 0;  t := i;  p := 0
        while mismatches ≤ k ∧ p < m do
            ℓ := LCE(t, p) // jump over matching part
            t := t + ℓ + 1;  p := p + ℓ + 1
            mismatches := mismatches + 1
        if p == m then
            return i
```

- **Analysis:** $\Theta(n + m)$ preprocessing $+$ $O(n \cdot k)$ matching

$\rightsquigarrow$ very efficient for small k

- State of the art
  - $O\left(n \frac{k^2 \log k}{m}\right)$ possible with complicated algorithms
  - extensions for edit distance $\leq k$ possible

Application 6: Matching with wildcards

- Allow a wildcard character in pattern

  stands for arbitrary (single) character

```
      unit*           P
in␣unit5␣we␣will      T
```

- similar algorithm as for k-mismatch $\rightsquigarrow$ $O(n \cdot k + m)$ when P has k wildcards

Application 6: Matching with wildcards

- Allow a wildcard character in pattern

  stands for arbitrary (single) character

```
  unit*            P
in␣unit5␣we␣will   T
```

- similar algorithm as for k-mismatch $\rightsquigarrow$ $O(n \cdot k + m)$ when P has k wildcards

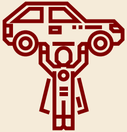
* * *

Many more applications, in particular for problems on biological sequences

20+ described in Gusfield, *Algorithms on strings, trees, and sequences* (1999)

Suffix trees – Discussion

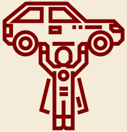
- Suffix trees were a threshold invention



- 👍 linear time and space
- 👍 suddenly many questions efficiently solvable in theory

Suffix trees – Discussion

- Suffix trees were a threshold invention

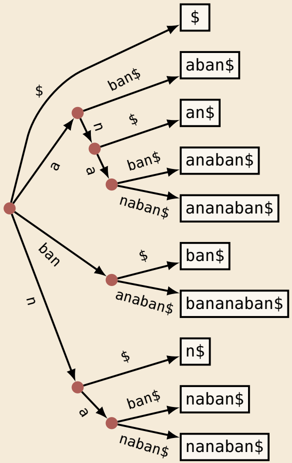


- linear time and space
- suddenly many questions efficiently solvable in theory

- construction of suffix trees:
  linear time, but significant overhead
- construction methods fairly complicated
- many pointers in tree incur large space overhead

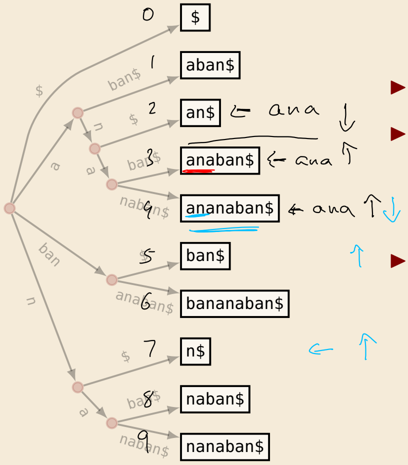
6.5 Suffix Arrays

Putting suffix trees on a diet



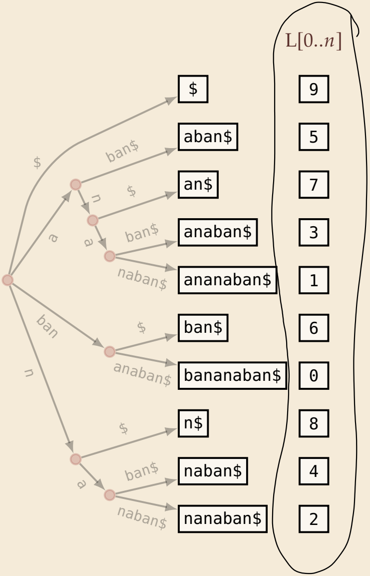
- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*

Putting suffix trees on a diet



- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*
- Idea: only store list of leaves $L[0..n]$
- Enough to do efficient string matching!
  1. Use binary search for pattern P
  2. check if P is prefix of suffix after position found
- **Example:** P = ana

Putting suffix trees on a diet



- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*

- Idea: only store list of leaves $L[0..n]$
- Enough to do efficient string matching!
  1. Use binary search for pattern P
  2. check if P is prefix of suffix after position found
- **Example:** P = `ana`

⇝ $L[0..n]$ is called ***suffix array***:

$L[r]$ = (start index of) rth suffix in sorted order

- using L, can do string matching with
  $\leq (\lg n + 2) \cdot m$ character comparisons