ALGORITHMS OF BIOINFORMATICS

6 Suffix Trees

11 December 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version 2025-12-18 13:27

Outline

6 Suffix Trees

Context

We're still working towards practical solutions for the read mapping problem.

*So far, our preprocessing was mostly getting smart on the **reads/patterns**.*

- ⇝ Now preprocess the genome/text.

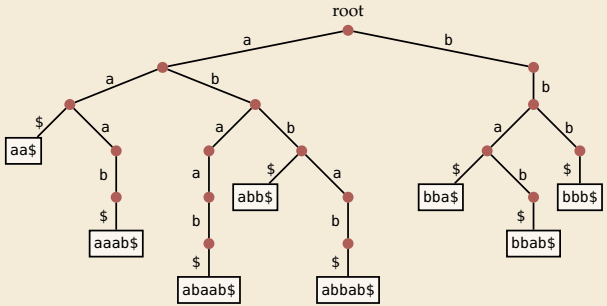
6.1 Suffix Trees

Recap: Tries

- efficient dictionary data structure for strings (or for Aho-Corasick automata 😉)
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

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

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

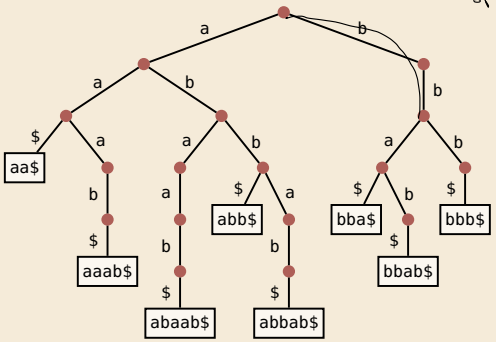


Compact tries

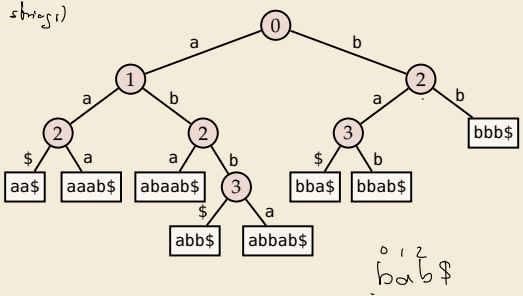
=1 child

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

standard trie size = O(total length of all strings)



compact trie size = O(#strings)



- search gives first character of edge only ⇝ must check for match against stored string
- all nodes ≥ 2 children ⇝ #nodes ≤ #leaves = #strings ⇝ linear space

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 $S_1, \ldots, S_k$ **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??

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

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

Suffix trees – Definition

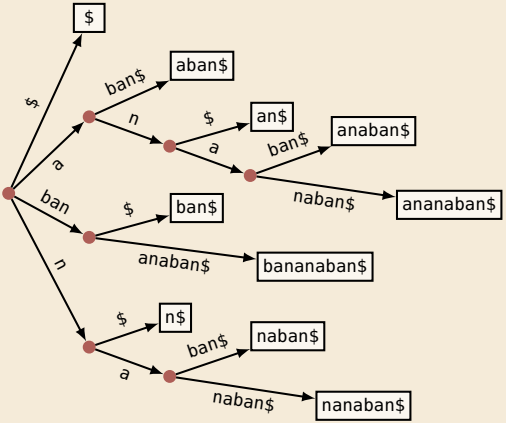
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

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

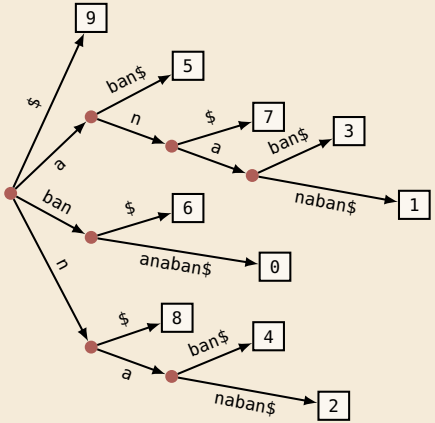
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

Example:

T = bananaban$

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

	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)

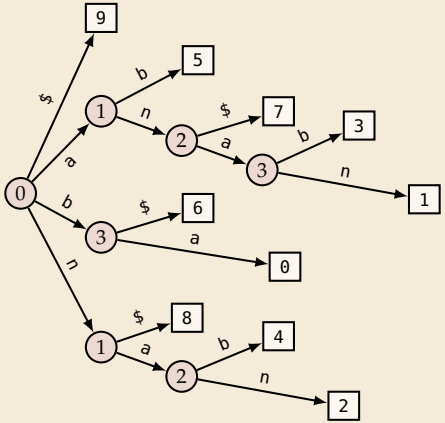
Example:

T = bananaban$

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

	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)



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 $\Theta(n^2)$. ⇝ not interesting!

same order of growth as reading the text!

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

- several fundamentally different methods known
- started as theoretical breakthrough
- now routinely used in bioinformatics practice

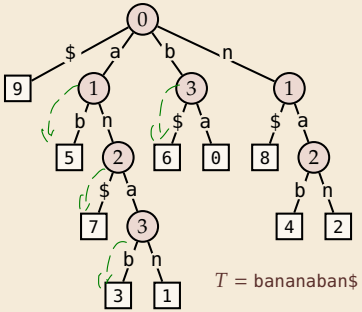
⇝ for now, take linear-time construction for granted. What can we do with them?

6.2 Direct Applications

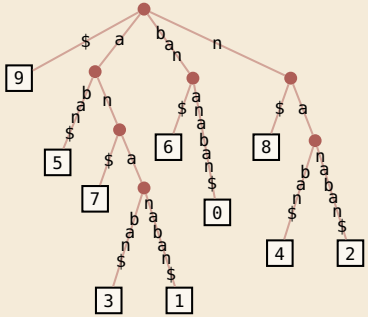
Applications of suffix trees

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

Recall: $\mathcal{T}$ stored like this:



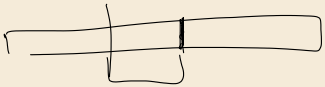
but think about this:



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

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 $\mathcal{T}$!



Application 1: Text Indexing / String Matching

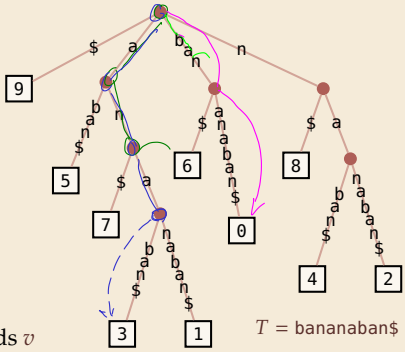
- P occurs in $T \iff P$ is a prefix of a suffix of T
- we have all suffixes in $\mathcal{T}$!

$\rightsquigarrow$ (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)
   $\rightsquigarrow$ 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
   $\rightsquigarrow$ 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!

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



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 $j \neq i$: $T[j..j+\ell) = T[i..i+\ell)$.

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 $j \neq i$: $T[j..j+\ell) = T[i..i+\ell)$.

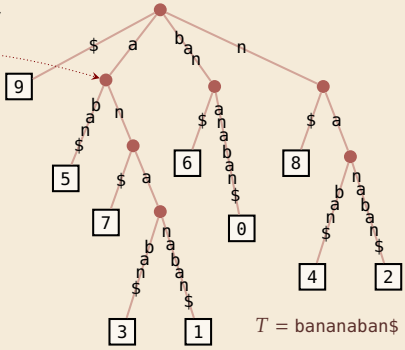
How can we efficiently check *all possible substrings?*

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



Application 2: Longest repeated substring

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

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

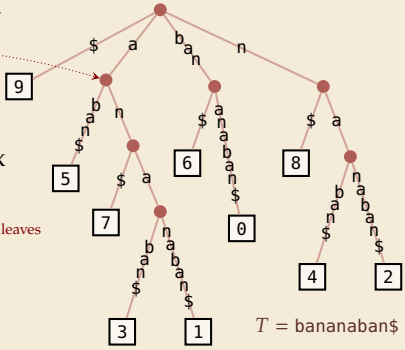
- T_5 = aban$ and T_7 = an$ have *longest common prefix* 'a'

⇝ ∃ 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



Application 2: Longest repeated substring

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

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

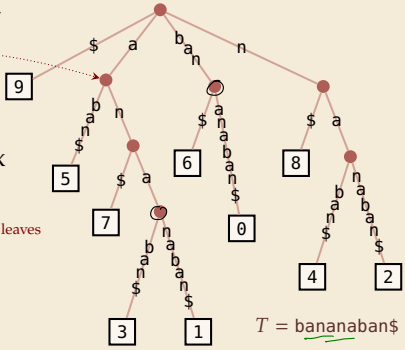
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

⇝ ∃ 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



- 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

6.3 Generalized Suffix Trees & Augmentation

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

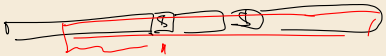
- longest *repeated* substring (of one string) feels very similar to
  longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
- can we solve that in the same way?
- could build the suffix tree for each $T^{(j)}$ ... but doesn't seem to help

$\rightsquigarrow$ 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

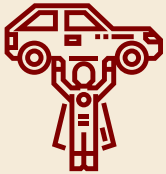
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts



Enter: ***generalized suffix tree***

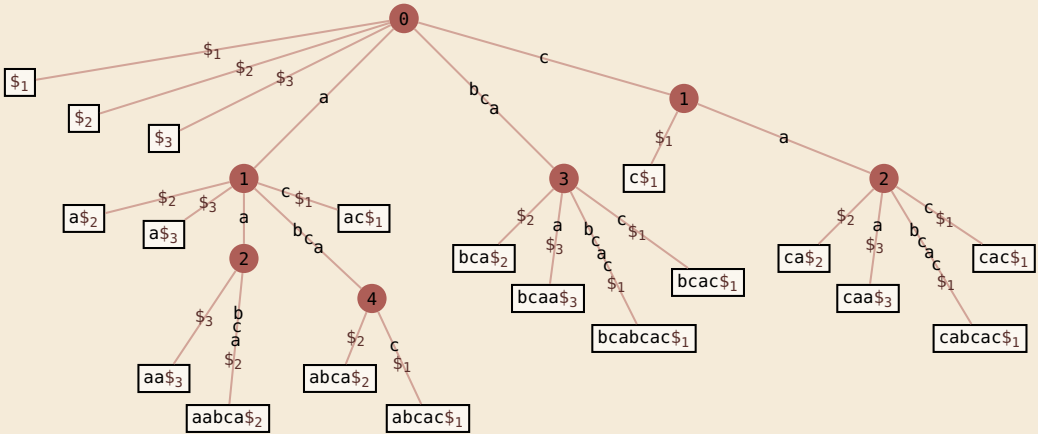
- Define $T := T^{(1)}\$_1 T^{(2)}\$_2 \cdots T^{(k)}\$_k$ for k new end-of-word symbols
- Construct suffix tree $\mathcal{T}$ for T

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



Generalized Suffix Tree – Example

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



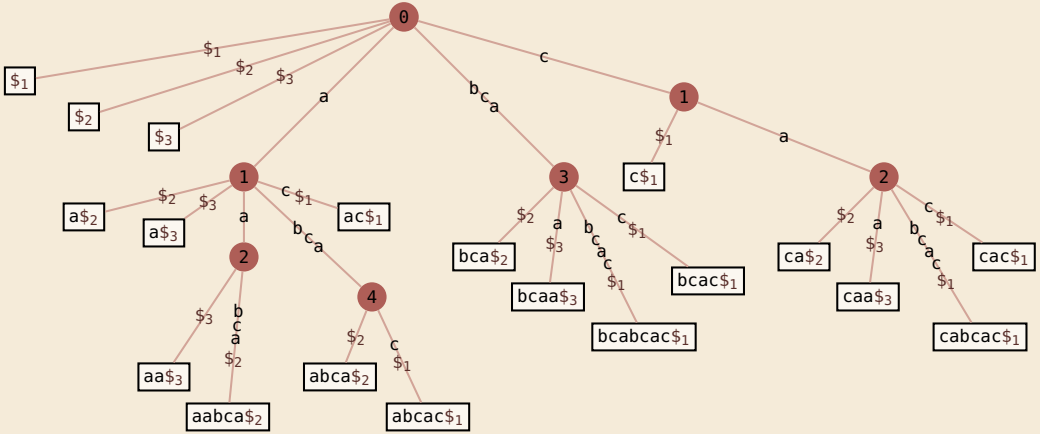
Application 3: Longest common substring

- 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 $\{1, \ldots, k\}$.

- Steps 1, 3, 4 take time $\Theta(n)$ for $n = n_1 + \cdots + n_k$ the total length of all texts.
- Step 2 takes $\Theta(kn)$ time $\rightsquigarrow$ linear if $k = O(1)$.
  - dependence on k can be removed with more clever labeling of vertices

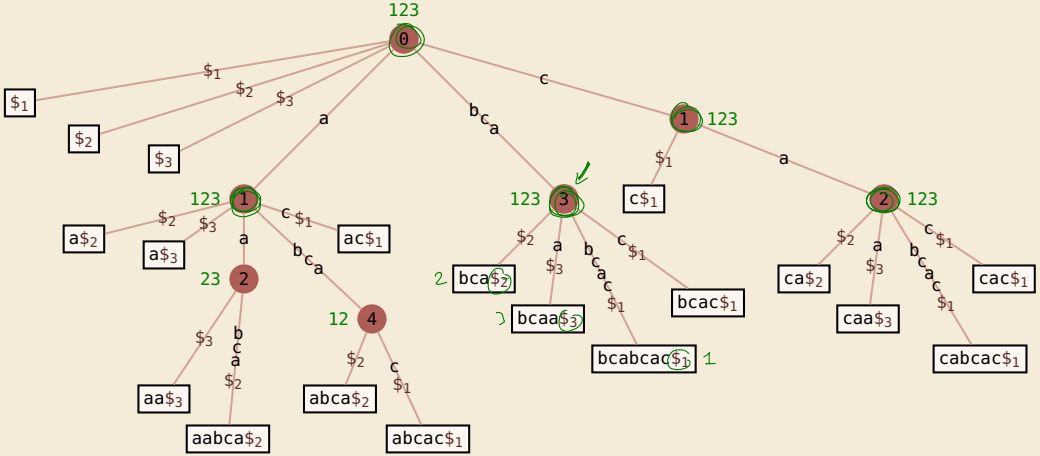
Longest common substring – Example

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



Longest common substring – Example

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

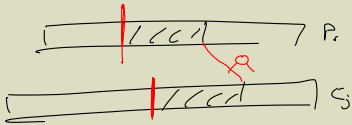


Application 4: DNA Sample Contamination

- **Given:** new reads $P[0..p)$, known contamination sources $C[0..c)$, length ℓ
- **Goal:** indices $I \subseteq [0..p)$ of reads that
  do **not** have common substring of length $\geq \ell$
  with **any** contamination source $C[j]$.

Application 4: DNA Sample Contamination

- **Given:** new reads $P[0..p)$, known contamination sources $C[0..c)$, length ℓ
- **Goal:** indices $I \subseteq [0..p)$ of reads that
  do **not** have common substring of length $\geq \ell$
  with **any** contamination source $C[j]$.

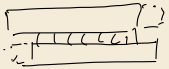


- Solution similar to longest common substring
  - Build generalized suffix tree $\mathcal{T}$ of $P[0..p)$ and $C[0..c)$
  - $P[r]$ has common substring with $C[j]$ of length $\geq \ell$
    iff $\exists$ node v in $\mathcal{T}$ of string depth $\geq \ell$ and leaves from $P[r]$ and $C[j]$ below.
  - $\rightsquigarrow$ Mark each node in $\mathcal{T}$ in a bottom-up traversal if its subtree contains any leaves from any $C[j]$.
  - For all marked nodes v of string depth $\geq \ell$, add r to I for all $P[r]$ with a leaf in subtree of v

$\rightsquigarrow$ Linear in the total length of P and C.

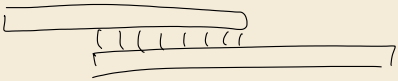
Application 5: Computing the Overlap Graph

- Recall the genome sequencing problem with inexact reads



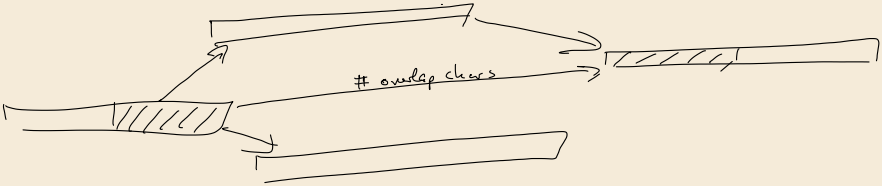
⇝ cannot assume perfect coverage / $k-1$ char overlaps for k-mers

- Here: Overlap graph with edge weights = **length of exact suffix-prefix match**



⇝ **All Pairs Suffix-Prefix Matching**

- **Given:** reads $P[0..p)$, $P[r] = P_r[0..m_r)$
- **Goal:** $O[0..p)[0..p)$ where $O[i][j]$ is length of longest suffix of P_i that is a prefix of P_j.



Application 5: Computing the Overlap Graph

- Recall the genome sequencing problem with inexact reads

$\rightsquigarrow$ cannot assume perfect coverage / $k-1$ char overlaps for k-mers

- Here: Overlap graph with edge weights = **length of exact suffix-prefix match**

$\rightsquigarrow$ **All Pairs Suffix-Prefix Matching**

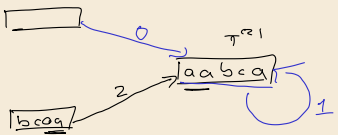
  - **Given:** reads $P[0..p)$, $P[r] = P_r[0..m_r)$
  - **Goal:** $O[0..p)[0..p)$ where $O[i][j]$ is length of longest suffix of P_i that is a prefix of P_j.

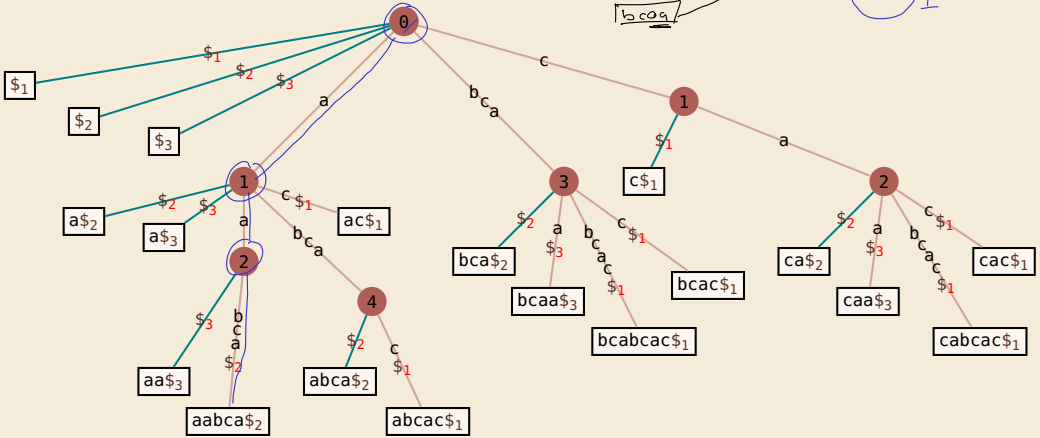
- To find suffix-prefix overlaps, use generalized suffix tree $\mathcal{T}$
- Mark edges labeled only with $\texttt{\$}_\texttt{i}$ as *terminal edges*
- Suffix of P_i equals prefix of P_j iff traversal of P_j in $\mathcal{T}$ reaches $\texttt{\$}_\texttt{j}$ terminal edge

$\rightsquigarrow$ in a single traversal, remember deepest terminal edge for each string
output all when reaching leaf for $P_j[0..m_j)$

All-Pairs Suffix-Prefix – Example

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

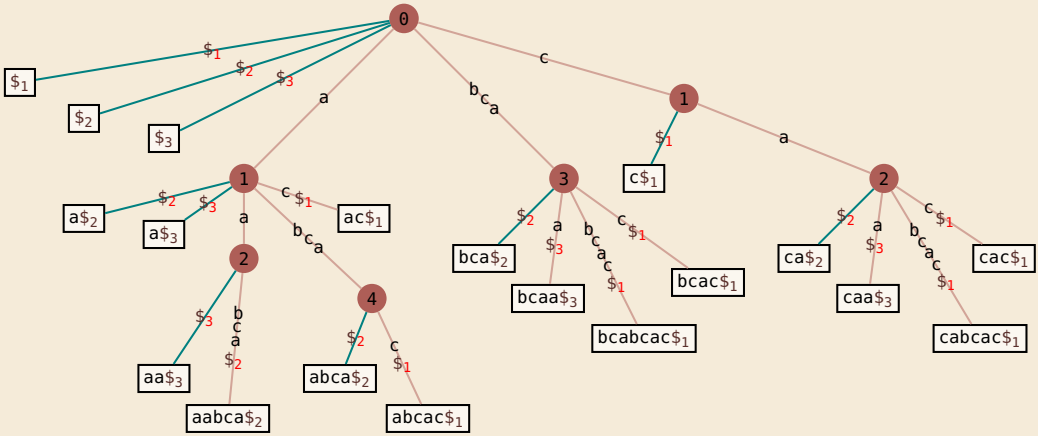




terminal edge

All-Pairs Suffix-Prefix – Example

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



terminal edge

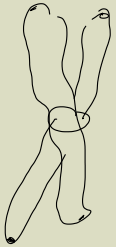
6.4 Tandem Repeats

Repetitions and “Junk DNA”

- only 1–2% of human genome are protein-coding genes . . . *what is the rest there for?*

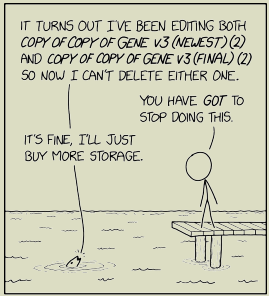
Repetitions and "Junk DNA"

- only 1–2% of human genome are protein-coding genes . . . *what is the rest there for?*
- some 15-25% are actually functional in that they are involved in some direct actions
  - protein-coding genes
  - regions representing non-coding RNA (*transfer RNA*, *ribosomal RNA*, *interfering RNA*, . . .)
  - structural support for chromosomes (*telomeres*, *centromeres*, *satellite DNA*)



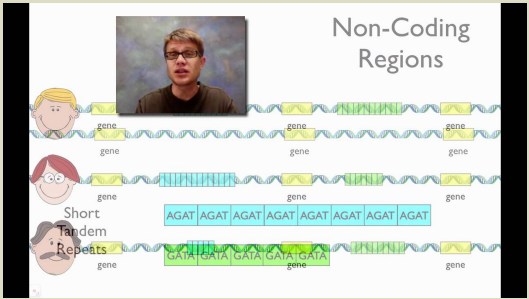
Repetitions and “Junk DNA”

- only 1–2% of human genome are protein-coding genes . . . *what is the rest there for?*
- some 15-25% are actually functional in that they are involved in some direct actions
  - protein-coding genes
  - regions representing non-coding RNA (*transfer RNA*, *ribosomal RNA*, *interfering RNA*, . . .)
  - structural support for chromosomes (*telomeres*, *centromeres*, *satellite DNA*)
- *how about the rest?*
  - a lot is *transposons* (“jumping genes”, “mobile DNA”)
    function is unclear
    each copy leaves a few newly inserted bases in your genome . . .
  - *introns* (cut out parts in eukaryotic split genes)
  - *pseudogenes*
    former copies of genes that lost function due to mutation



DNA Fingerprinting

- for identifying individuals, need highly variable DNA region
- but also one that we can easily find in reads



DNA Fingerprinting
https://youtu.be/DbR9xMXuK7c

How can we find tandem repeats?

Tandem Repeats

Formally, finding tandem repeats amounts to the following problem.

- **Given:** Text $T[0..n)$
- **Goal:** Report all pairs (i, ℓ), so that $T[i..i+\ell) = T[i+\ell, i+2\ell)$

Tandem Repeats

Formally, finding tandem repeats amounts to the following problem.

- **Given:** Text $T[0..n)$
- **Goal:** Report all pairs (i, ℓ), so that $T[i..i+\ell) = T[i+\ell, i+2\ell)$

How can suffix trees help us here?

Tandem Repeats

Formally, finding tandem repeats amounts to the following problem.

- **Given:** Text $T[0..n)$
- **Goal:** Report all pairs (i, ℓ), so that $T[i..i+\ell) = T[i+\ell, i+2\ell)$

How can suffix trees help us here? They can't (directly) 😬

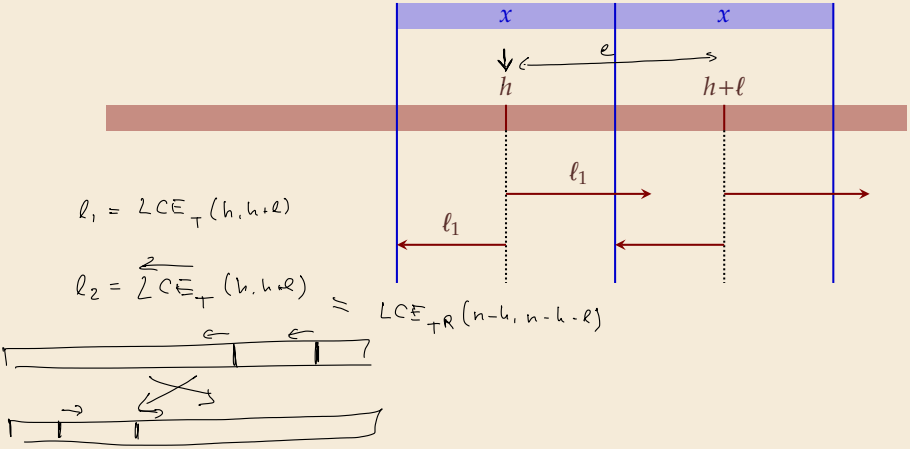
But remember the ***longest common extension (LCE)*** data structure:

- **Given:** String $T[0..n)$
- **Goal:** Answer queries $\mathrm{LCE}_T(i, j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

After $\Theta(n)$ preprocessing (time and space), query time $O(1)$

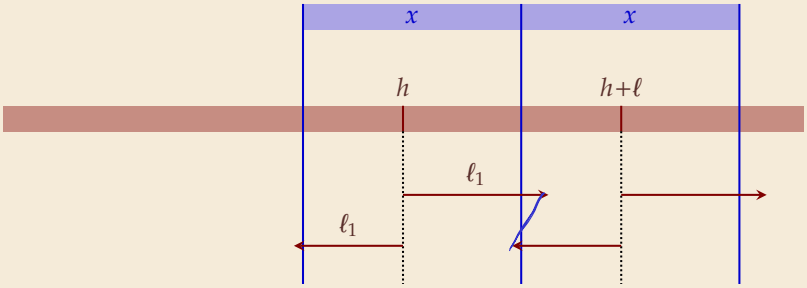
Tandem Repeats – Idea

- a tandem repeat is a substring xx
- if we fix the length $\ell = |x|$ and a "seed position" h,
  a tandem repeat must allow extending outwards from h by at least ℓ positions



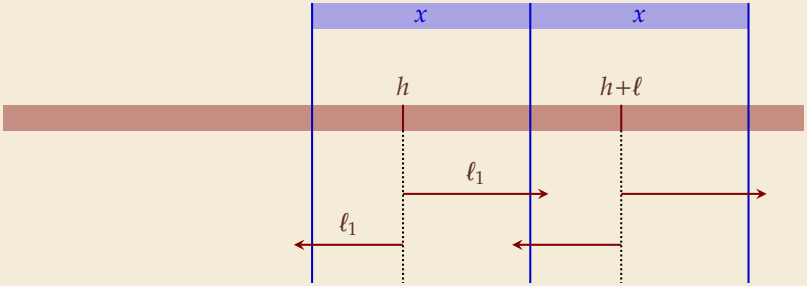
Tandem Repeats – Idea

- a tandem repeat is a substring xx
- if we fix the length $\ell = |x|$ and a "seed position" h,
  a tandem repeat must allow extending outwards from h by at least ℓ positions



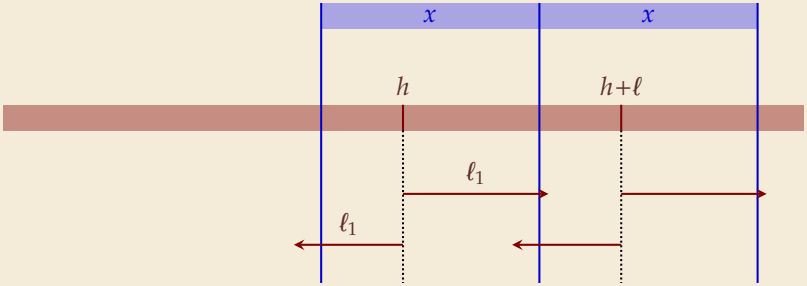
Tandem Repeats – Idea

- a tandem repeat is a substring xx
- if we fix the length $\ell = |x|$ and a "seed position" h,
  a tandem repeat must allow extending outwards from h by at least ℓ positions



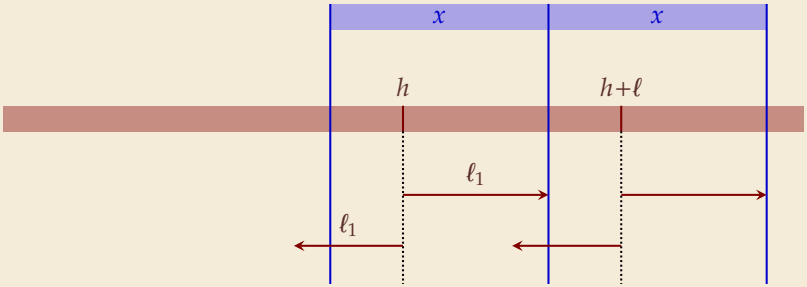
Tandem Repeats – Idea

- a tandem repeat is a substring xx
- if we fix the length $\ell = |x|$ and a "seed position" h,
  a tandem repeat must allow extending outwards from h by at least ℓ positions



Tandem Repeats – Idea

- a tandem repeat is a substring xx
- if we fix the length $\ell = |x|$ and a "seed position" h,
  a tandem repeat must allow extending outwards from h by at least ℓ positions



Tandom Repeats – Code

```
1  procedure tandemRepeats(T[0..n))
2      Build LCE data structure on T and T^R
3      // T^R = reversed text
4      // We abbreviate ē(i, j) = LCE_T(i, j)
5      // resp. ẽ(i, j) = LCE_{T^R}(n − j, n − i)
6      tr := ∅
7      appendRepeats(T[0..n), tr)
8      return tr
```

```
1  procedure appendRepeats(T[0..n), tr)
2      if n ≤ 1 return
3      h := ⌊n/2⌋
4      appendRepeats(T[0..h))
5      appendRepeats(T[h..n))
6      // Add all repeats xx with h inside first copy of x
7      for ℓ ∈ [1 .. n − h) // length of x
8          q := h + ℓ
9          ℓ1 := ē(h, q)
10         ℓ2 := ẽ(h − 1, q − 1)
11         if ℓ1 + ℓ2 ≥ ℓ
12             tr := tr ∪ {(j, ℓ) : j ∈ [h − ℓ2 .. h + ℓ1 − ℓ]}
13     // Symmetrically for h inside second copy of x
14     for ℓ ∈ [1..h]
15         q := h − ℓ
16         ℓ1 := ē(h, q)
17         ℓ2 := ẽ(h − 1, q − 1)
18         if ℓ1 + ℓ2 ≥ ℓ
19             tr := tr ∪ {(j, ℓ) : j ∈ [q − ℓ2 .. q + ℓ1 − ℓ]}
```

Tandom Repeats – Analysis

- **Running time**
  - nonrecursive part $O(n)$ (using LCE preprocessing)
  - recording output pairs (j, ℓ) over all recursive calls $O(output)$ time
  - recursive function uses $O(n)$ LCE queries $\rightsquigarrow$ $O(n)$ time
  - recursive function satisfies $T(n) = \Theta(n) + 2T(n/2)$ $\rightsquigarrow$ $T(n) = O(n \log n)$

  $\rightsquigarrow$ $O(n \log n + output)$

6.5 Longest Common Extensions

Longest Common Extensions & Suffix Trees

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

  Recall ***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: $\mathrm{LCE}(i, j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

Longest Common Extensions & Suffix Trees

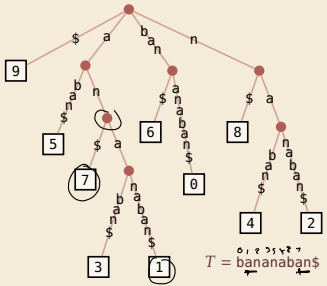
- We implicitly used a special case of a more general, versatile idea:

  Recall ***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: $\mathrm{LCE}(i,j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

$\rightsquigarrow$ 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: $\mathrm{LCE}(i,j) = \mathrm{LCP}(T_i, T_j) = \mathrm{stringDepth}\big(\mathrm{LCA}(\boxed{i}, \boxed{j})\big)$

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

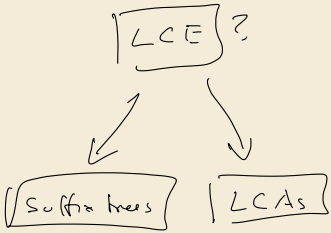
How to find lowest common ancestors?

- 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 in **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.

Suffix trees – Discussion

- Suffix trees were a threshold invention


linear time and space


suddenly many questions efficiently solvable in theory

Suffix trees – Discussion

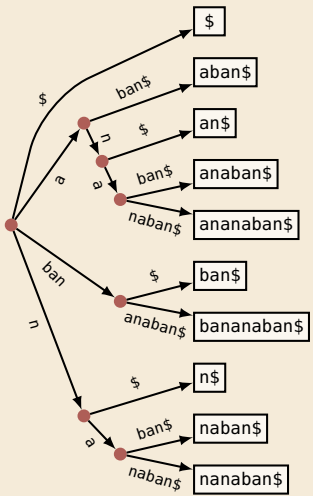
- Suffix trees were a threshold invention

- 👍 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

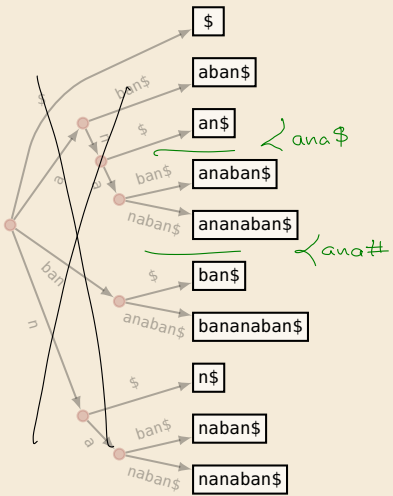
6.6 Suffix Arrays

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*

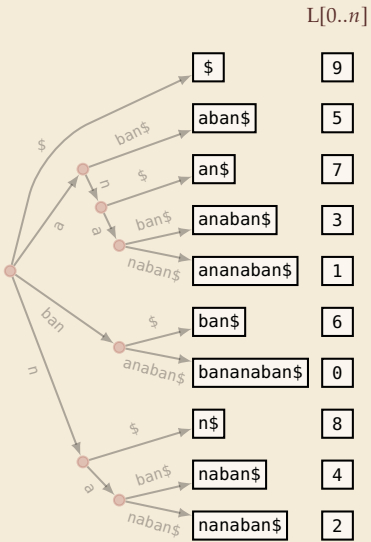
  suffix array

- Idea: only store list of leaves $L[0..n]$
- Sufficient 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

  ana$

  ana#

Putting suffix trees on a diet



- **Observation:** order of leaves in suffix tree = suffixes lexicographically *sorted*
- Idea: only store list of leaves $L[0..n]$
- Sufficient 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

$\rightsquigarrow$ $L[0..n]$ 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

Suffix arrays – Construction

How to compute $L[0..n]$?

- from suffix tree
  - possible with traversal ...
  - 👎 but we are trying to avoid constructing suffix trees!

- sorting the suffixes of T using general purpose sorting method
  - 👍 trivial to code!
  - but: comparing two suffixes can take $\Theta(n)$ character comparisons
  - 👎 $\Theta(n^2 \log n)$ time in worst case

Suffix arrays – Construction

How to compute $L[0..n]$?

- from suffix tree
  - possible with traversal ...
  - 👎 but we are trying to avoid constructing suffix trees!

- sorting the suffixes of T using general purpose sorting method
  - 👍 trivial to code!
  - but: comparing two suffixes can take $\Theta(n)$ character comparisons
  - 👎 $\Theta(n^2 \log n)$ time in worst case

- We can do better!

Excursion: String sorting

- when sorting strings, "blind" string comparisons can cost $\Theta(n)$ character comparisons
- happens iff strings share long prefix!

$\rightsquigarrow$ dedicated string sorting methods need to remember common prefixes between strings then we can avoid redoing these comparisons

Excursion: String sorting

- when sorting strings, "blind" comparisons can cost $\Theta(n)$ character comparisons
- happens iff strings share long prefix!

⇝ dedicated string sorting methods need to remember common prefixes between strings then we can avoid redoing these comparisons

(length of) longest common prefix

- Option 1: Mergesort with LCP values for adjacent elements in runs
- Option 2: Fat-pivot radix quicksort
  - **partition** based on d**th character** only (initially $d = 0$)
  - ⇝ 3 segments: smaller, equal, or larger than dth symbol of pivot
  - recurse on smaller and large with same d, on equal with $d + 1$
  - ⇝ never compare equal prefixes twice

§5.1 of Sedgewick, Wayne *Algorithms 4th ed.* (2011), Pearson

Fat-pivot radix quicksort – Example

she

sells

seashells

by

the

sea

shore

the

shells

she

sells

are

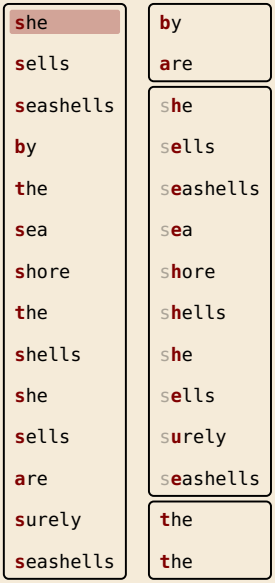
surely

seashells

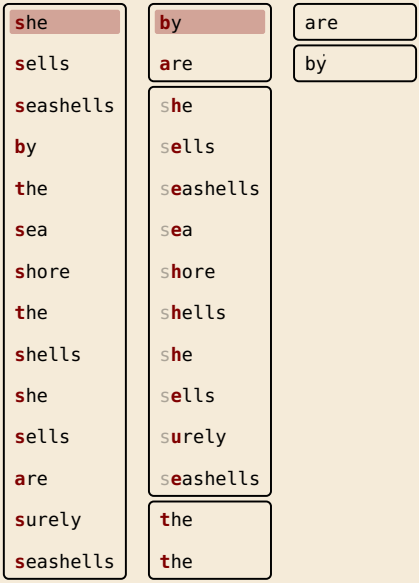
Fat-pivot radix quicksort – Example

she
sells
seashells
by
the
sea
shore
the
shells
she
sells
are
surely
seashells

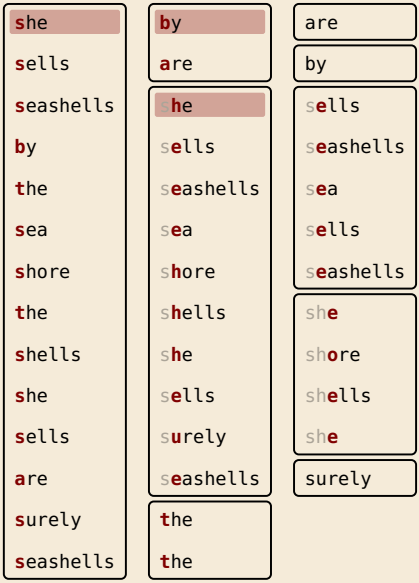
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example

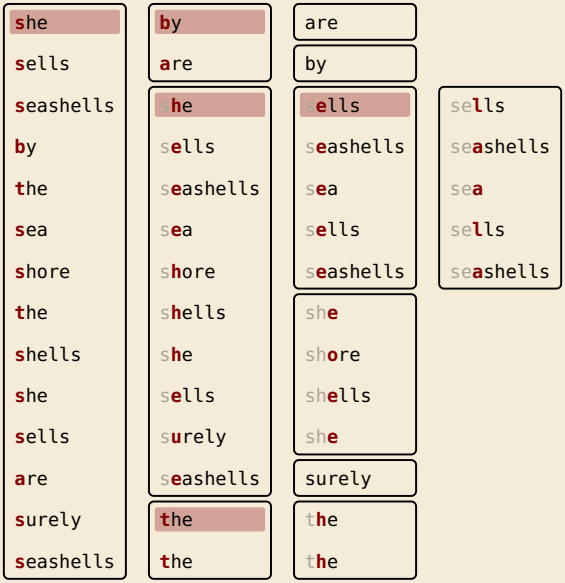


Fat-pivot radix quicksort – Example

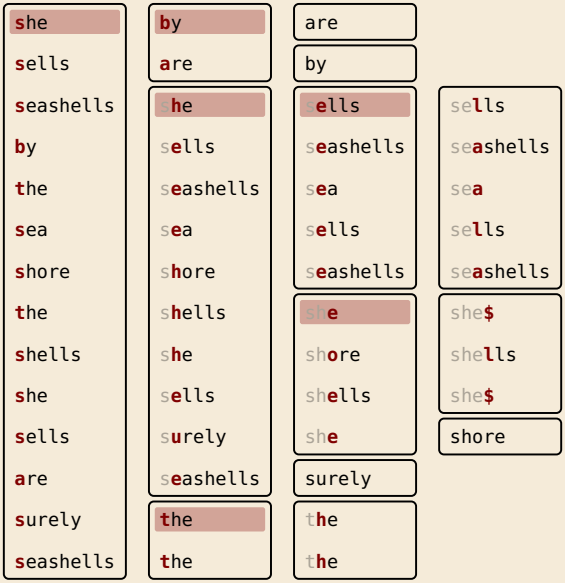


Fat-pivot radix quicksort – Example

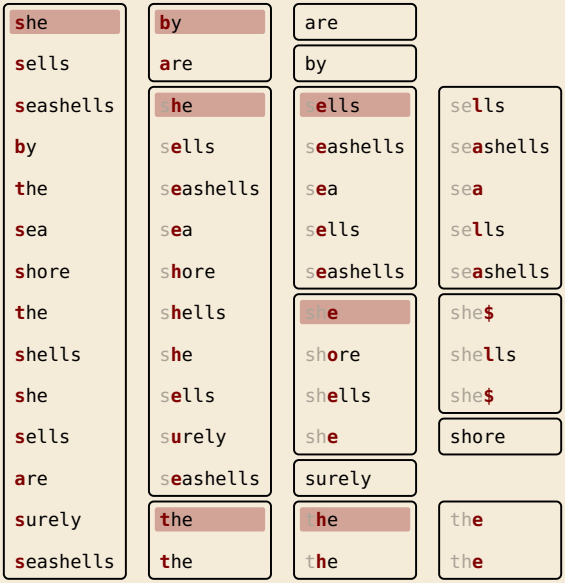
Fat-pivot radix quicksort – Example



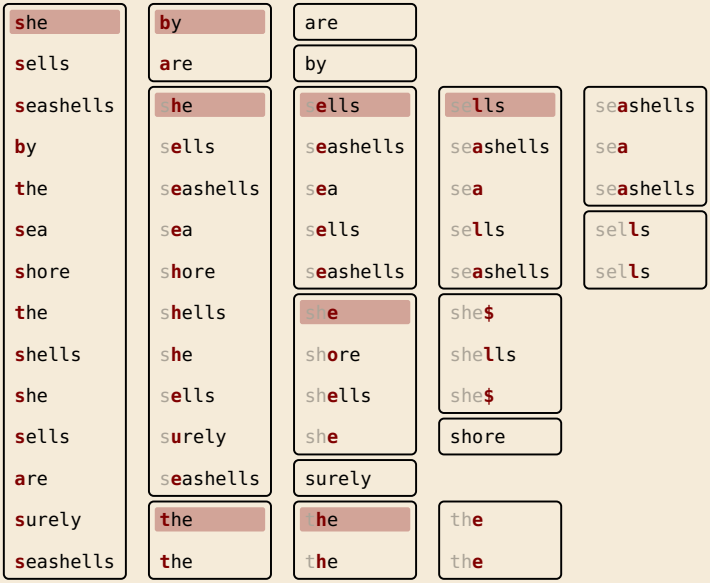
Fat-pivot radix quicksort – Example



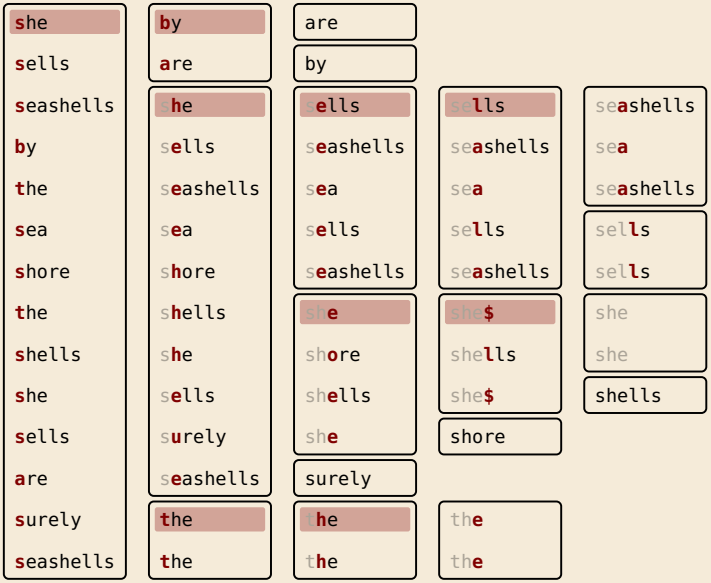
Fat-pivot radix quicksort – Example



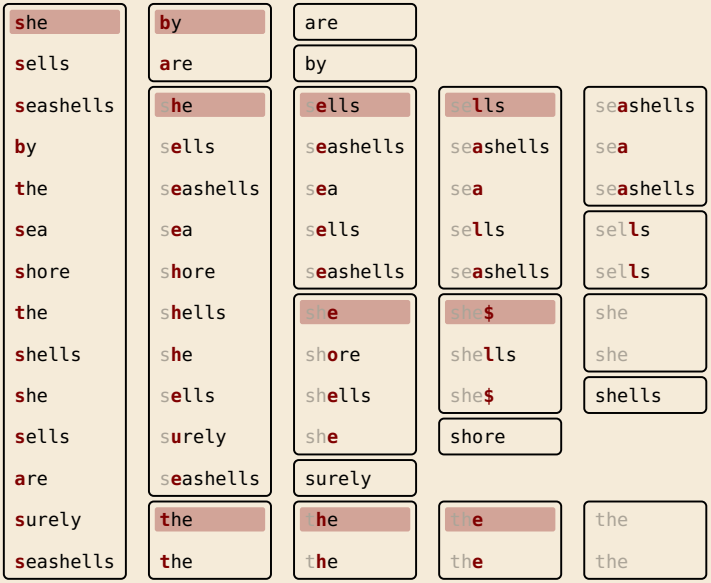
Fat-pivot radix quicksort – Example



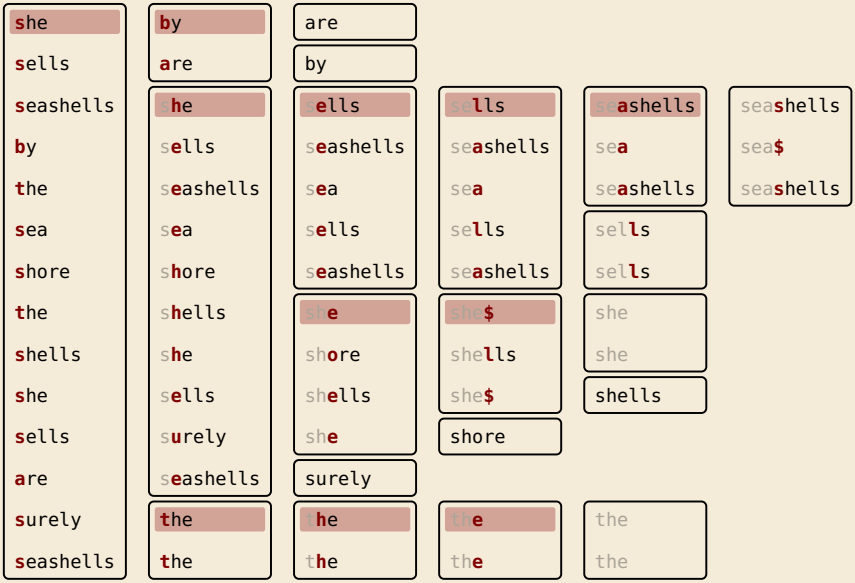
Fat-pivot radix quicksort – Example



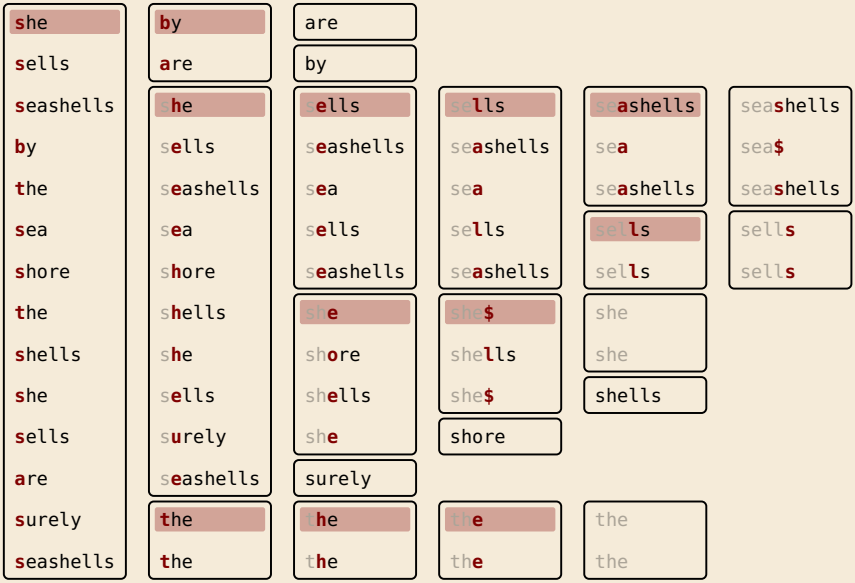
Fat-pivot radix quicksort – Example



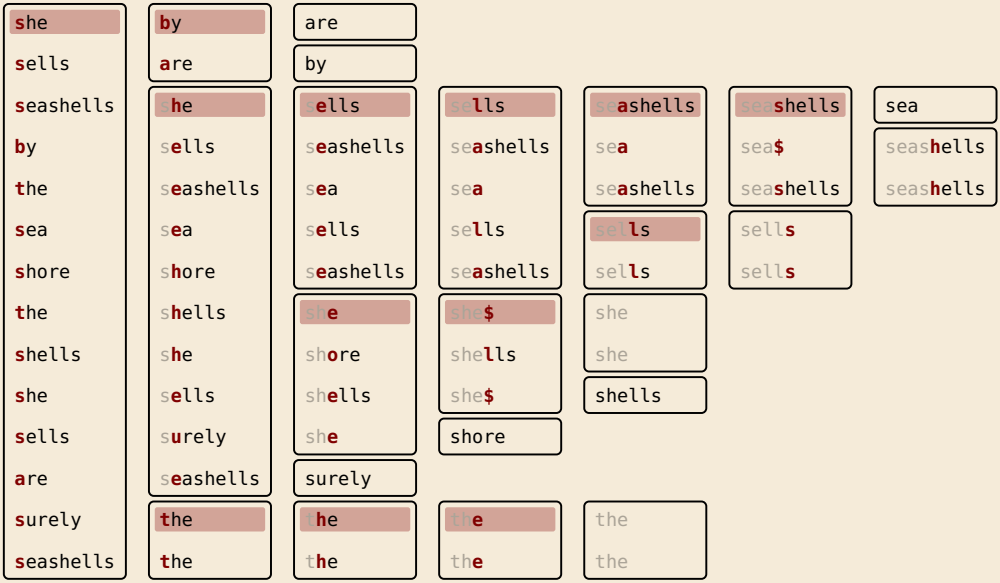
Fat-pivot radix quicksort – Example



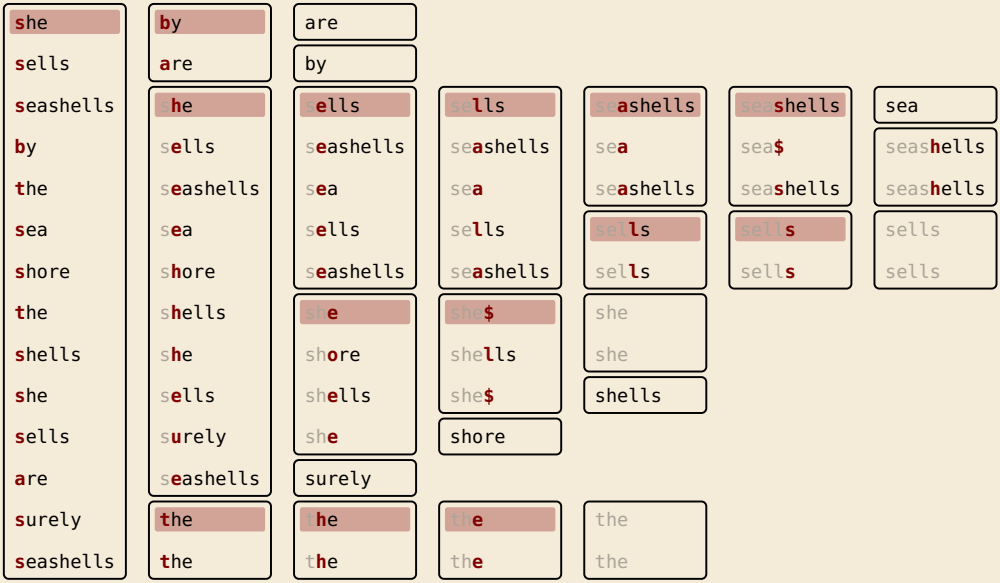
Fat-pivot radix quicksort – Example



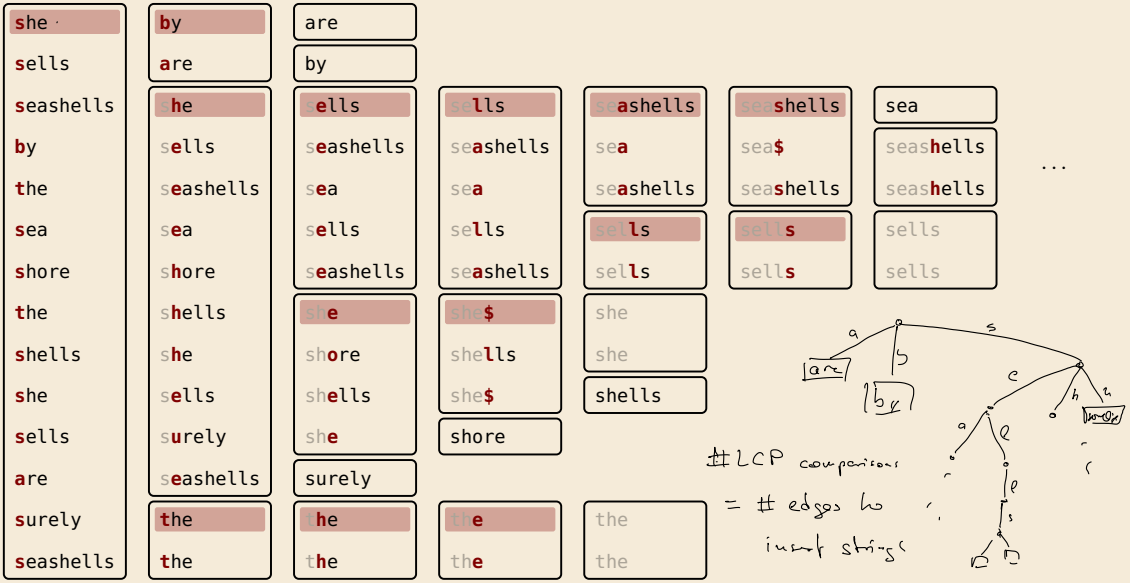
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Analysis

Separately analyze character comparisons **by outcome**

1. ***"Decisive Comparisons:"*** character comparisons with outcome "$<$" or "$>$"
   - can have at most one in any **string comparison** (afterwards done!)
   - $\rightsquigarrow$ Same number of decisive comparisons as in standard quicksort (just delayed)
   - $\rightsquigarrow$ expected $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ decisive comparisons

Fat-pivot radix quicksort – Analysis

Separately analyze character comparisons **by outcome**

1. ***"Decisive Comparisons:"*** character comparisons with outcome "<" or ">"
   - can have at most one in any **string comparison** (afterwards done!)
   - ⇝ Same number of decisive comparisons as in standard quicksort (just delayed)
   - ⇝ expected $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ decisive comparisons

2. ***LCP comparisons:*** character comparisons that return "="
   - must be all remaining character comparisons
   - every such comparison contributes to common prefix between strings, never compare same characters again
   - every string sort must discover longest common prefixes in sorted order
   - ⇝ #LCP comparisons = #comparisons when inserting all strings into a **trie**

(
w/ unary path
to leaves collapsed

Fat-pivot radix quicksort – Discussion

- simple to code
- efficient for sorting many lists of strings
- fat-pivot radix quicksort finds suffix array in $O(n \log n)$ expected time (random string)
- worst case remains $\Omega(n^2)$, i. e., $T = \mathsf{a}^n$

  Note: Not quicksort's fault! Any generic string sorting method must take $\Omega(n^2)$ time here

Fat-pivot radix quicksort – Discussion


simple to code


efficient for sorting many lists of strings

- fat-pivot radix quicksort finds suffix array in $O(n \log n)$ expected time

random string


worst case remains $\Omega(n^2)$, i. e., $T = \mathsf{a}^n$

Note: Not quicksort's fault! Any generic string sorting method must take $\Omega(n^2)$ time here

*but we can do $O(n)$ time **worst case**!*