8 Text Indexing – Searching entire genomes

24 November 2023

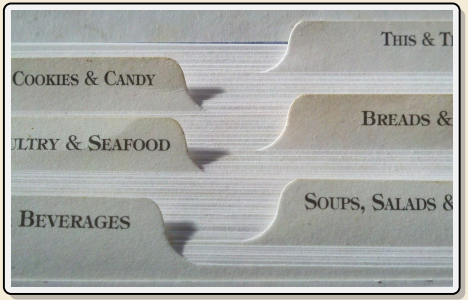
Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
version **2023-11-23 21:56**

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 8: *Text Indexing*



Outline

8 Text Indexing

8.1 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

- original indices (same as "indexes") 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

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 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?*

symbol table w/ keys are strings BSTs are an option.

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

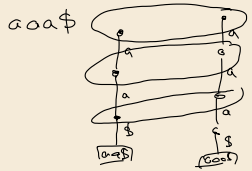
Tries

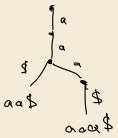
- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

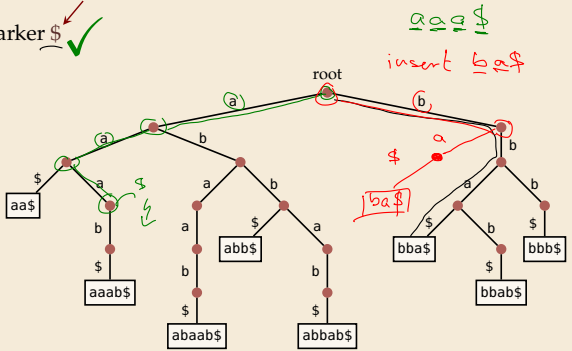
- **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
  - strings of same length ✓
  - strings have "end-of-string" marker $ (some character $\notin \Sigma$) ✓

- **Example**:

small ex. {aa$, aaab$, abaab$, abb$, abbab$, bba$, bbab$, bbb$}







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.
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 $\Theta(n \cdot m)$

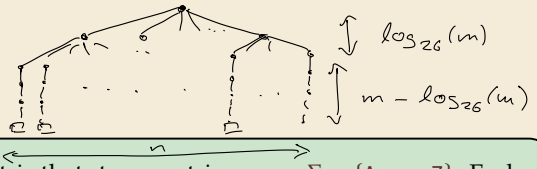
D $\Theta(n \log m)$

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

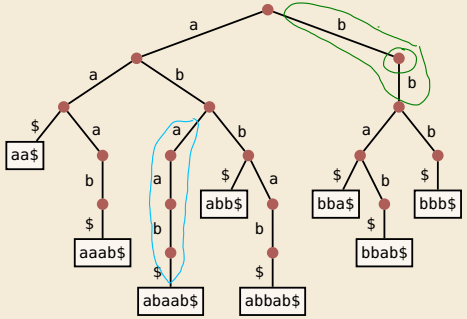
→ *sli.do/comp526*

Compact tries

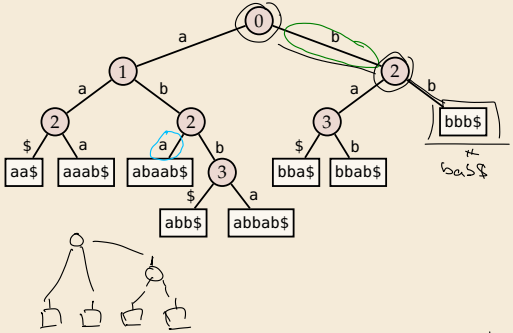
=1 child

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

standard trie



compact trie



storing child pointers $O(n\sigma)$

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

- all nodes ≥ 2 children ⇝ #nodes ≤ #leaves = #strings ⇝ linear space $O(n)$

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
CAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCCCAGCAATCCGCGCGCCGGGACAGAA
TGCCCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
```

  - binary streams

```
00000010101001111010111000001111100011111011111001101101000011100010011011110000010001101010
01101100001101011010000000100000000111010110000010000111101011101100100011001011011101111111
11000101000101100101000000111010101001100000000110110000011001111100000101 01010111011110000011
10101110010010101010100000011111010011000000111100110101000000010010010000010110001100011011 1
```

- need new ideas

8.2 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 $S_1, \ldots, S_k$ **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

	0	1	2	3	4	5	6
0	b	a	n	a	n	a	$

1 anana$

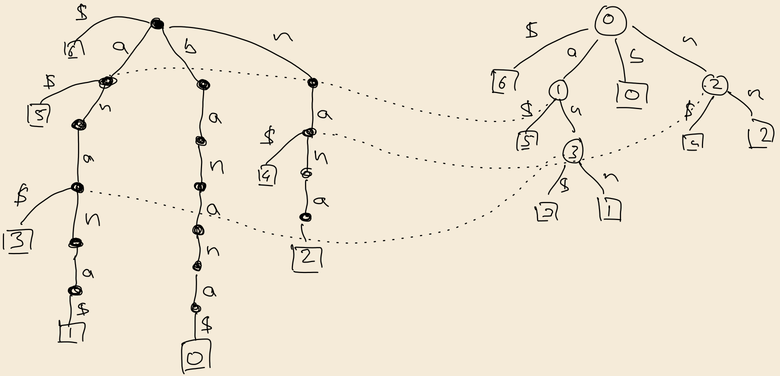
2 nana$

3 ana$

4 na$

5 a$

6 $



$\Theta(n^2)$ size

Suffix trees – Definition

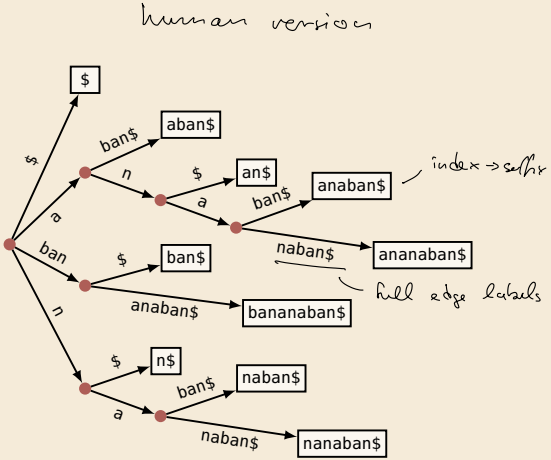
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

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

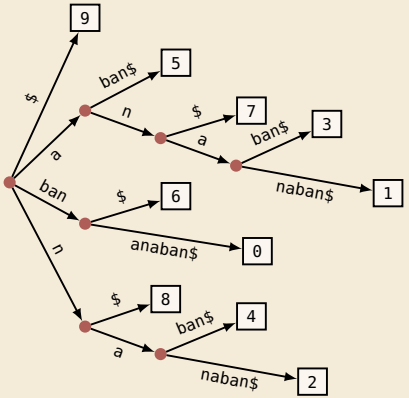
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

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)

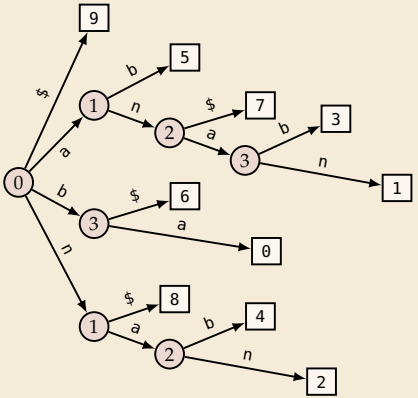
Example:

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 $\Theta(n^2)$. $\rightsquigarrow$ 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.~~
- **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.~~ $n+1$

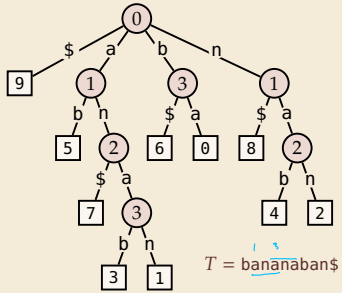
→ *sli.do/comp526*

8.3 Applications

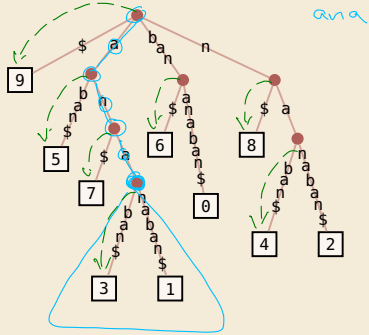
Applications of suffix trees

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

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

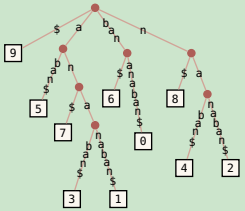


but think about this:



- 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 $P = \texttt{ana}$?

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.

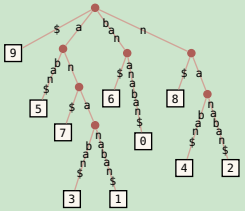
I P starts at index 3.

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 $P = \texttt{ana}$?

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

- 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

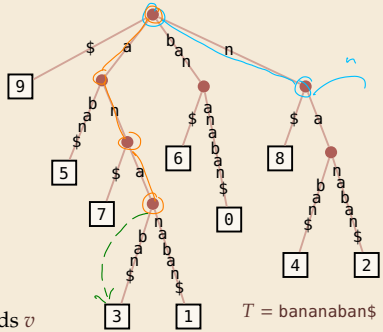
- P occurs in $T \iff P$ is a prefix of a suffix of T
- we have all suffixes in $\mathcal{T}$!

$\rightsquigarrow$ (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!



can find all matches by traversing subtree

Application 1: Text Indexing / String Matching

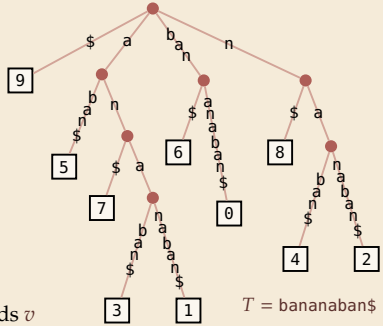
- P occurs in $T \iff$ P is a prefix of a suffix of T
- we have all suffixes in $\mathcal{T}$!

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

e. g. for compression ⇝ Unit 7

How can we efficiently check *all possible substrings?*

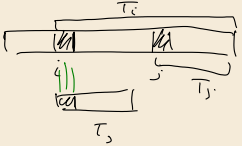
Application 2: Longest repeated substring

- **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?*

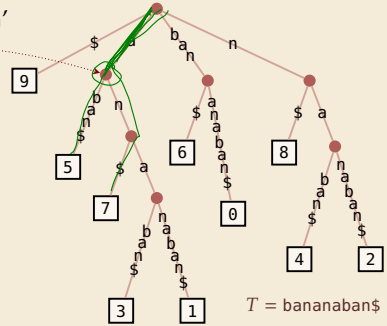
Repeated substrings = shared paths in *suffix tree*



- 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



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

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

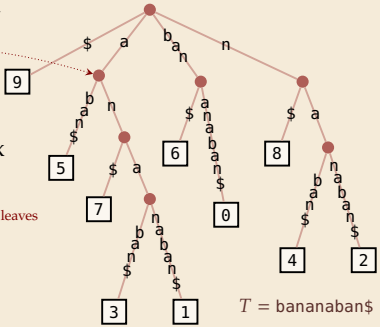
- T_5 = aban$ and T_7 = an$ have *longest common prefix* 'a'

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

e. g. for compression ⇝ Unit 7

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

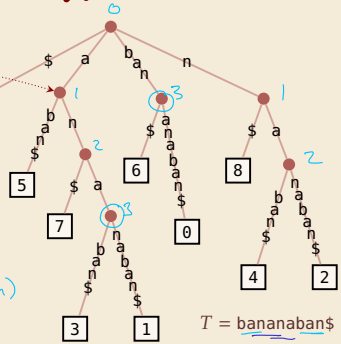
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

⇝ ∃ internal node with path label '`a`'

here single edge, can be longer path

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

actually: adjacent leaves



- Algorithm:
  1. Compute *string depth* (=length of path label) of nodes $\Theta(n)$
  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

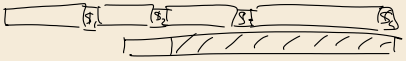
- longest *repeated* substring (of one string) feels very similar to
  longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
- can we solve that in the same way?
- could build the suffix tree for each $T^{(j)}$ ... but doesn't seem to help

$\rightsquigarrow$ 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

$\rightsquigarrow$ 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 $T_i^{(j)} = T^{(j)}[i..n_j]$

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 $\mathcal{T}$. $O(n)$
  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.

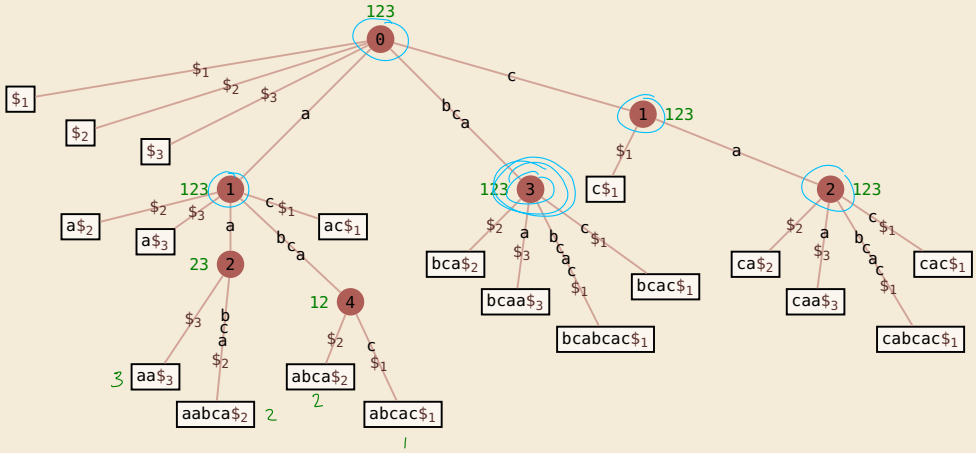
     $O(n)$
  3. In top-down traversal, compute *string depths* of nodes. (as above) $O(n)$
  4. Report deepest node (by string depth) whose subset is $\{1, \ldots, k\}$.

- 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

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



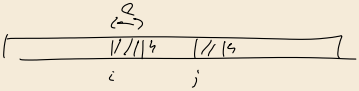
8.4 Longest Common Extensions

Application 4: Longest Common Extensions

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

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

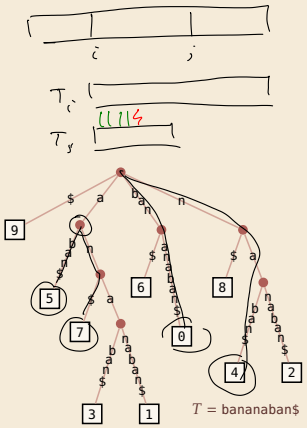
  - **Given:** String $T[0..n)$
  - **Goal:** Answer LCE queries, i. e.,
    given positions i, j in T,
    how far can we read the same text from there?
    formally: $\mathrm{LCE}(i, j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$



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



$\rightsquigarrow$ use suffix tree of T!

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

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

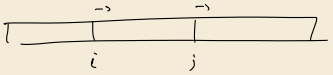
- in short: $\mathrm{LCE}(i,j) = \mathrm{LCP}(T_i, T_j) = \mathrm{stringDepth}(\mathrm{LCA}(\boxed{i}, \boxed{j}))$

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

LCE



compute LCE naively at query time

$O(\ell)$ w.c. $O(n)$

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

$\rightsquigarrow$ Unit 9

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

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

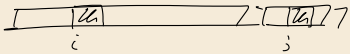
- Adapted brute-force algorithm $\rightsquigarrow$ $O(n \cdot m)$

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

Kangaroo Algorithm for approximate matching

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

$O(n \cdot m)$

- **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

$\notin$ exam

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

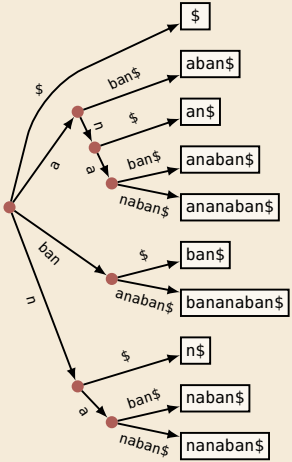
- construction of suffix trees:
  linear time, but significant overhead
- construction methods fairly complicated
- many pointers in tree incur large space overhead

↳ node as array child[0..σ) ⇝ σ space overhead

• BST over child labels ⇝ lg σ time overhead

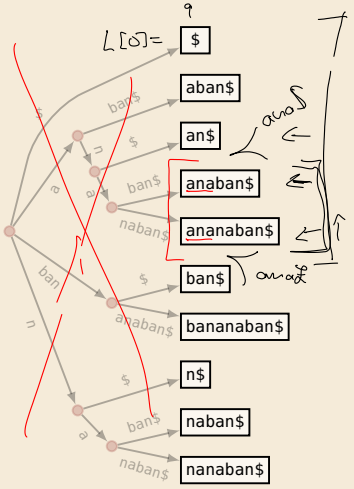
8.5 Suffix Arrays

Putting suffix trees on a diet



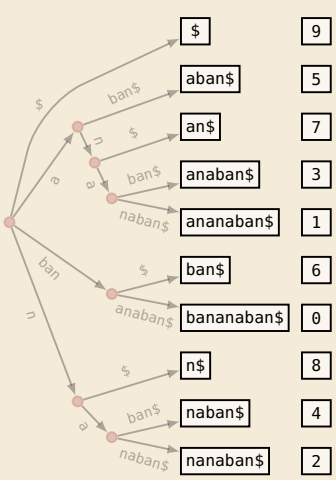
- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*

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 vs. $\Theta(n)$

Clicker Question

Check all correct statements about *suffix array* $L[0..n]$ and *suffix tree* $\mathcal{T}$ of text $T[0..n)$ (for $\sigma = O(1)$)

A $L[0..n]$ lists the start indices of leaves of $\mathcal{T}$ in left-to-right order.

B $T\big[L[r]..n\big]$ is the path label in $\mathcal{T}$ to the leaf storing r.

C $T\big[L[r]..n\big]$ is the path label to the rth leaf in $\mathcal{T}$.

D $T_{L[r]}$ is the rth smallest suffix of T (lexicographic order).

E In terms of Θ-classes, $\mathcal{T}$ needs more space than L.

F L (and T) suffice to solve the text indexing problem.

→ *sli.do/comp526*

Clicker Question

Check all correct statements about *suffix array* $L[0..n]$ and *suffix tree* $\mathcal{T}$ of text $T[0..n)$ (for $\sigma = O(1)$)

A $L[0..n]$ lists the start indices of leaves of $\mathcal{T}$ in left-to-right order. ✓

B ~~$T\big[L[r]..n\big]$ is the path label in $\mathcal{T}$ to the leaf storing r.~~

C $T\big[L[r]..n\big]$ is the path label to the rth leaf in $\mathcal{T}$. ✓

D $T_{L[r]}$ is the rth smallest suffix of T (lexicographic order). ✓

E ~~In terms of Θ-classes, $\mathcal{T}$ needs more space than L.~~

F L (and T) suffice to solve the text indexing problem. ✓

→ *sli.do/comp526*

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

$\overline{1} = a\,a\,a\,a\,a\,a\,a$

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!

Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

A L can be very easily computed from B and T

B B can be very easily computed from L and T

C Both A and B

D Neither A nor B

→ *sli.do/comp526*

Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

- **A** ~~L can be very easily computed from B and T~~
- **B** B can be very easily computed from L and T ✓
- **C** ~~Both A and B~~
- **D** ~~Neither A nor B~~

→ *sli.do/comp526*

Digression: Recall BWT

Burrows-Wheeler Transform

1. Take all cyclic shifts of S
2. Sort cyclic shifts
3. Extract last column

S = `alf␣eats␣alfalfa$`
B = `asff$f␣e␣lllaaata`

Digression: Computing the BWT

How can we compute the BWT of a text efficiently?

Digression: Computing the BWT

How can we compute the BWT of a text efficiently?

- cyclic shifts $S \;\widehat{=}\;$ suffixes of S
  - comparing cyclic shifts stops at first $
  - for comparisons, anything after $ irrelevant!
- BWT is essentially suffix sorting!
  - $B[i] = S[L[i] - 1]$
  - where $L[i] = 0$, $B[i] = \$$

$\rightsquigarrow$ Can compute B in $O(n)$ time from L

	r	$\downarrow$	$L[r]$
alf␣eats␣alfalfa$	0	$alf␣eats␣alfalfa	16
lf␣eats␣alfalfa$a	1	␣alfalfa$alf␣eats	8
f␣eats␣alfalfa$al	2	␣eats␣alfalfa$alf	3
␣eats␣alfalfa$alf	3	a$alf␣eats␣alfalf	15
eats␣alfalfa$alf␣	4	alf␣eats␣alfalfa$	0
ats␣alfalfa$alf␣e	5	alfa$alf␣eats␣alf	12
ts␣alfalfa$alf␣ea	6	alfalfa$alf␣eats␣	9
s␣alfalfa$alf␣eat	7	ats␣alfalfa$alf␣e	5
␣alfalfa$alf␣eats	8	eats␣alfalfa$alf␣	4
alfalfa$alf␣eats␣	9	f␣eats␣alfalfa$al	2
lfalfa$alf␣eats␣a	10	fa$alf␣eats␣alfal	14
falfa$alf␣eats␣al	11	falfa$alf␣eats␣al	11
alfa$alf␣eats␣alf	12	lf␣eats␣alfalfa$a	1
lfa$alf␣eats␣alfa	13	lfa$alf␣eats␣alfa	13
fa$alf␣eats␣alfal	14	lfalfa$alf␣eats␣a	10
a$alf␣eats␣alfalf	15	s␣alfalfa$alf␣eat	7
$alf␣eats␣alfalfa	16	ts␣alfalfa$alf␣ea	6

Fat-pivot radix quicksort – Example

she

sells

seashells

by

the

sea

shore

the

shells

she

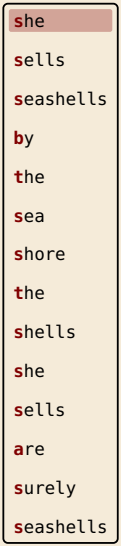
sells

are

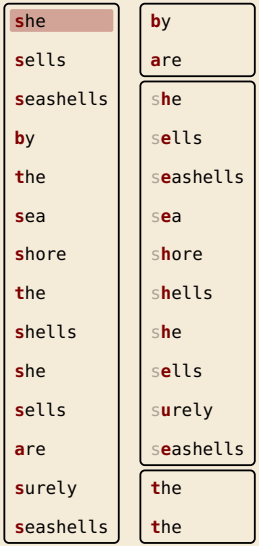
surely

seashells

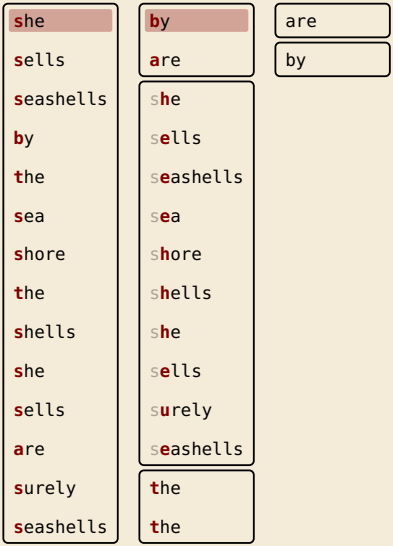
Fat-pivot radix quicksort – Example



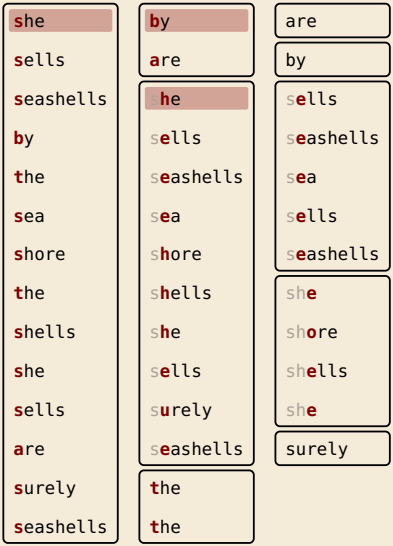
Fat-pivot radix quicksort – Example



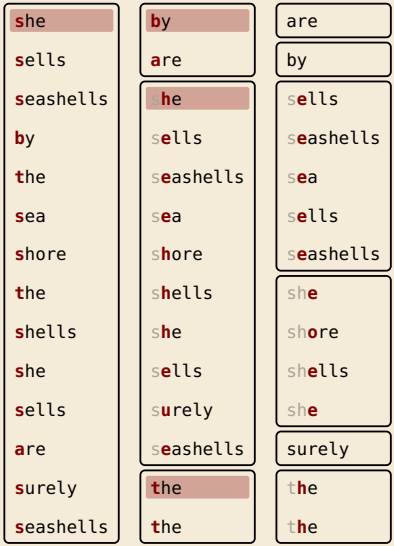
Fat-pivot radix quicksort – Example



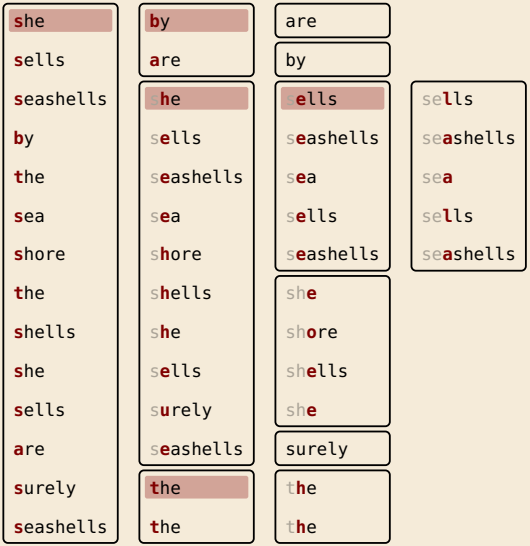
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example

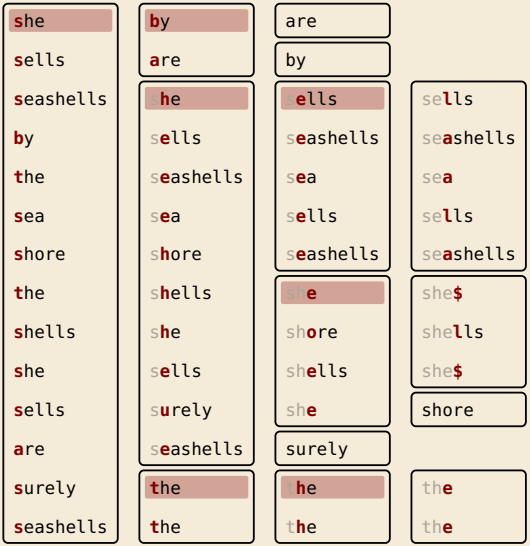


Fat-pivot radix quicksort – Example

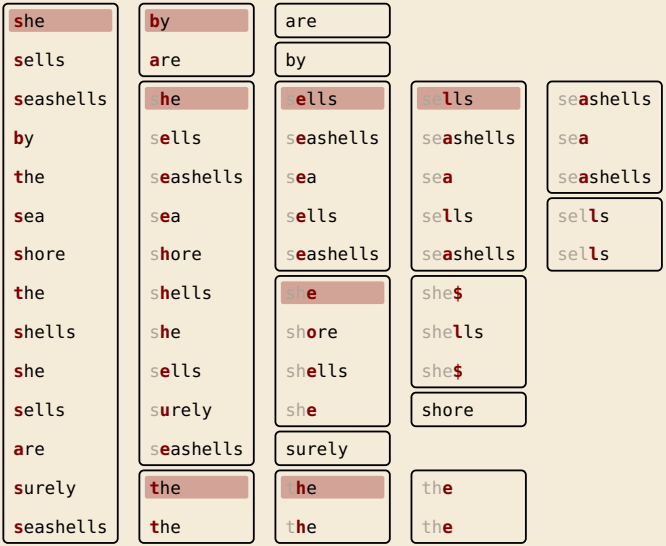


Fat-pivot radix quicksort – Example

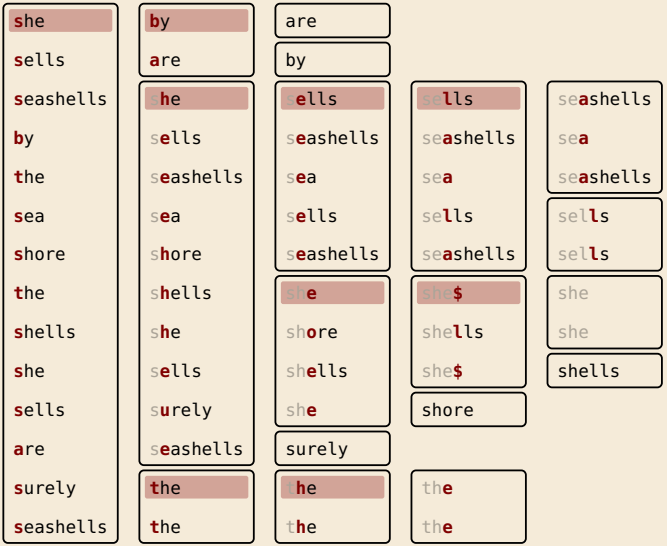
Fat-pivot radix quicksort – Example



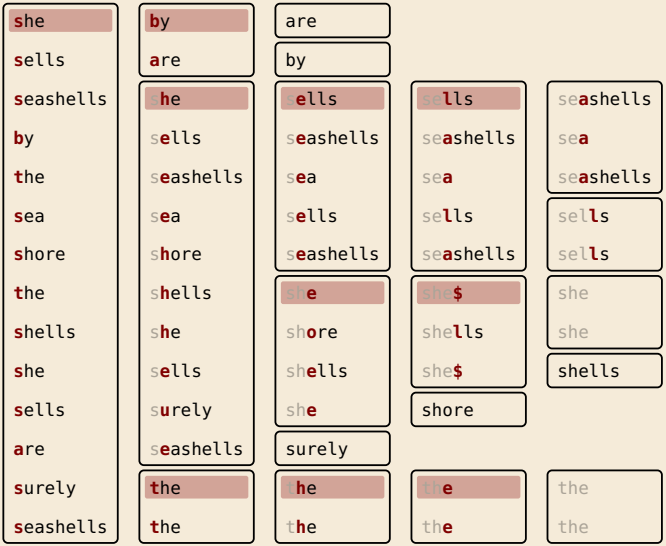
Fat-pivot radix quicksort – Example



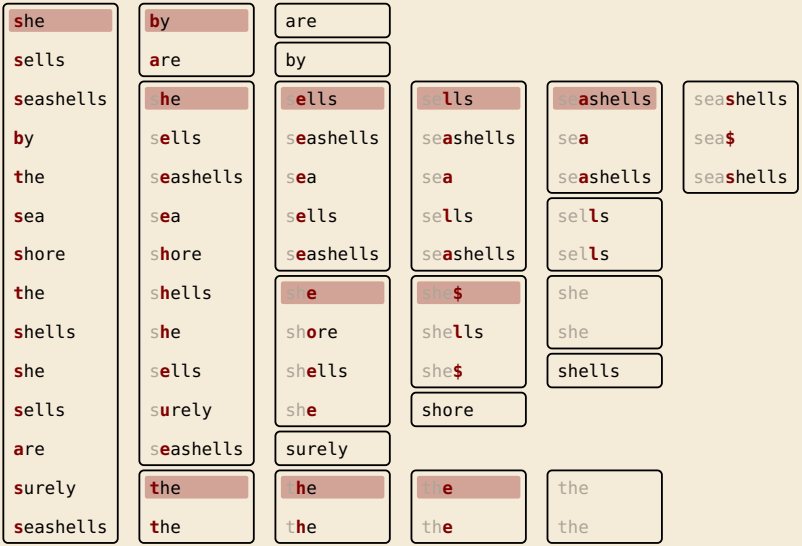
Fat-pivot radix quicksort – Example



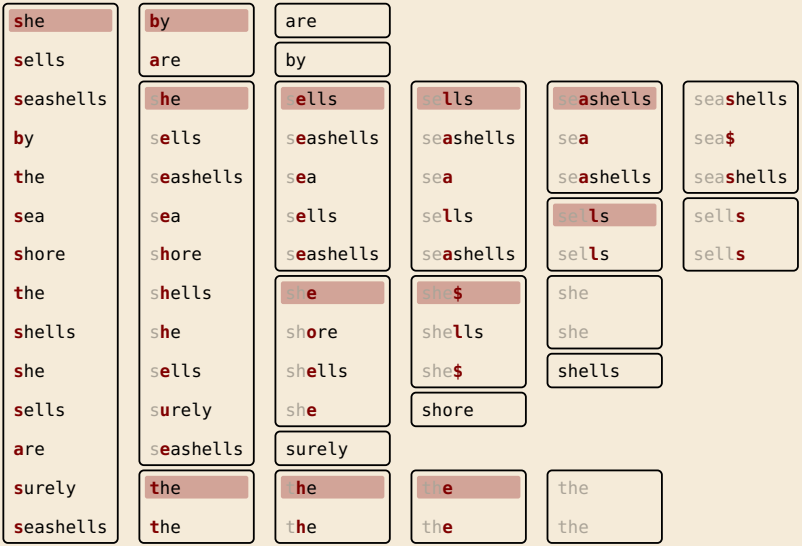
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example



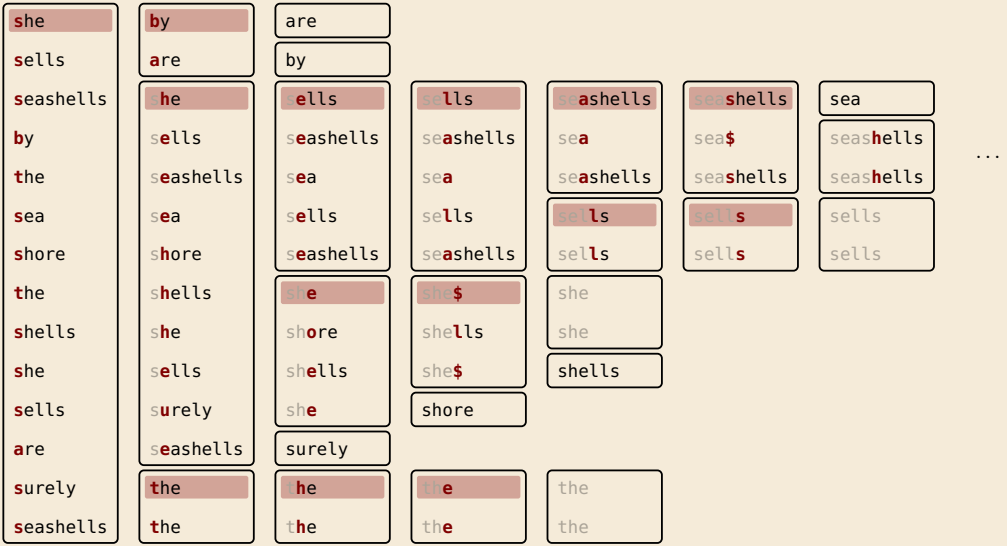
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example

Fat-pivot radix quicksort – Example

Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort

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

- **partition** based on d**th character** only (initially $d = 0$)

$\rightsquigarrow$ 3 segments: smaller, equal, or larger than dth symbol of pivot

- recurse on smaller and large with same d, on equal with $d + 1$

  $\rightsquigarrow$ never compare equal prefixes twice

Fat-pivot radix quicksort

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

- **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

⇝ can show: $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ character comparisons on average (for random strings)


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)

Fat-pivot radix quicksort

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

- **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

⇝ can show: $\sim 2 \ln(2) \cdot n \lg n \approx 1.39 n \lg n$ character comparisons on average (for random strings)


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)

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