13 Text Indexing – Searching entire genomes

2 February 2026

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg
version 2026-02-02 15:59 H

Learning Outcomes

Unit 13: ***Text Indexing***

1. Know and understand methods for text indexing: *inverted indexes*, *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.

Outline

13 Text Indexing

13.1 Inverted Indexes and Tries

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 indexes

≈ same as "indices"

- original 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 index:

- collect all words in T
  - can be as simple as splitting T at whitespaces
  - actual implementations typically support *stemming* of words
    `goes` $\longrightarrow$ `go`, `cats` $\longrightarrow$ `cat`

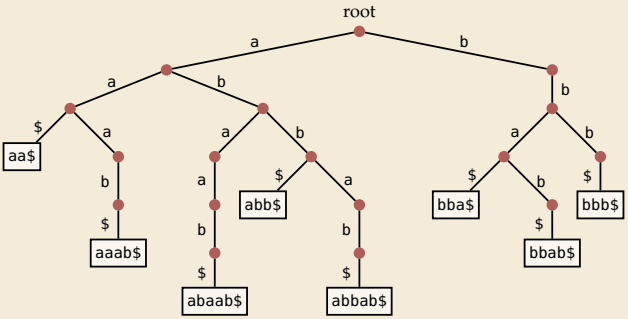
    language specific!
- store mapping from words to a list of occurrences $\rightsquigarrow$ *how?*

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

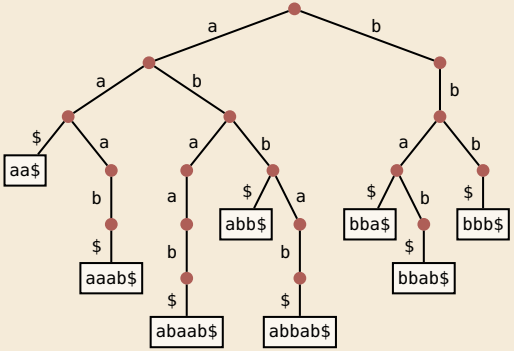


Compact tries

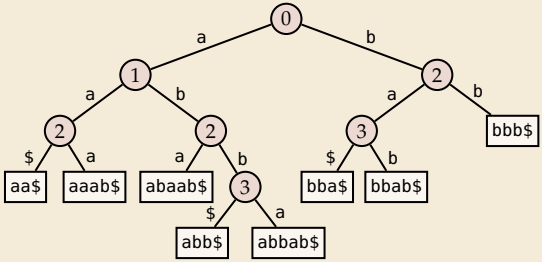
=1 child

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

standard trie



compact trie



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

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

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

```
0000001010100111101011100000111111000111111011111001101101000011100010011011110000010001101010
0110110000110101101100000001000000001110101100000100001111010111011001000110010110111011111111
1100010100010110010100000011101010100110000000011011000011001111110000101 0101011101111000011
1010111001001010101010000011111010011000000111100110101000000010010010000010110001100011011 1
```

- need new ideas

13.2 Suffix Trees

Suffix trees – A 'magic' data structure

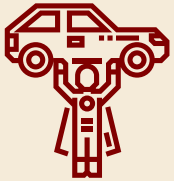
Appetizer: Longest common substring problem

- 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 ⇝ `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 $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

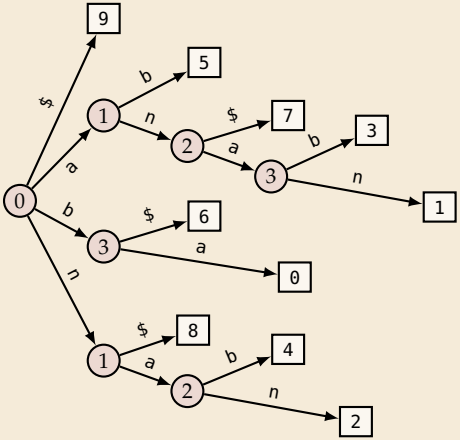
Example:

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!

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)!

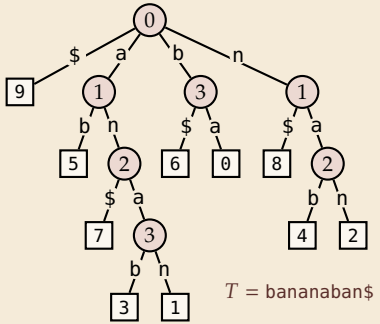
$\rightsquigarrow$ for now, take linear-time construction for granted. What can we do with them?

13.3 Applications

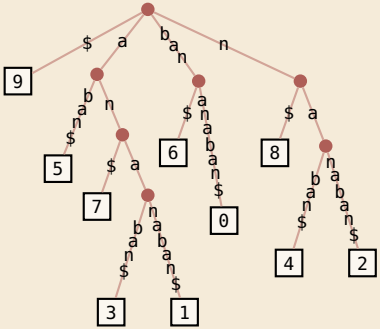
Applications of suffix trees

- 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 \$)

Application 1: Text Indexing / String Matching

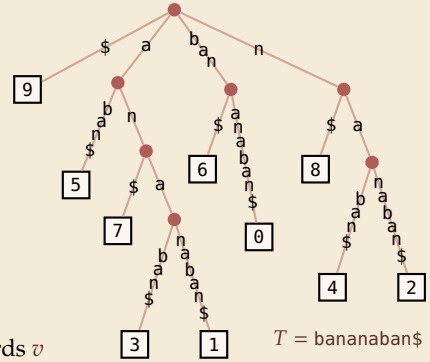
- P occurs in T $\iff$ P is a prefix of a suffix of T
- we have all suffixes in $\mathcal{T}$!

$\rightsquigarrow$ (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?*

Repeated substrings = shared paths in *suffix tree*

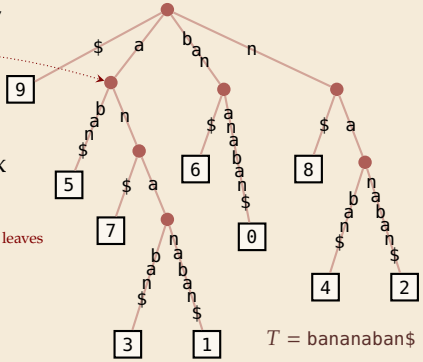
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

⇝ ∃ 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

13.4 Generalized Suffix Trees

Generalized suffix trees

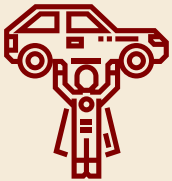
- longest *repeated* substring (of one string) feels very similar to
  longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
- can we solve that in the same way?
- could build the suffix tree for each $T^{(j)}$ ... but doesn't seem to help

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

Enter: ***generalized suffix tree***

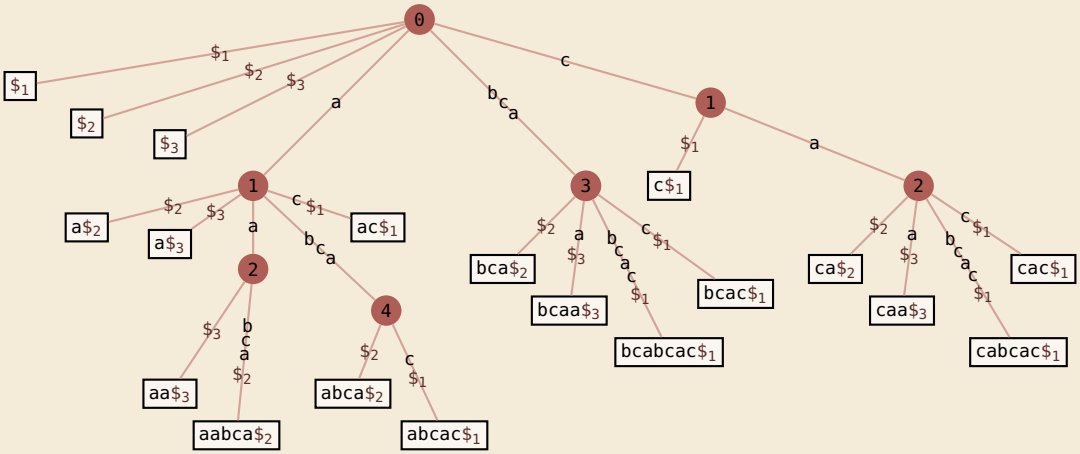
- Define $T := T^{(1)}\$_1 T^{(2)}\$_2 \cdots T^{(k)}\$_k$ for k new end-of-word symbols
- Construct suffix tree $\mathcal{T}$ for T

$\rightsquigarrow$ $\$_j$-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

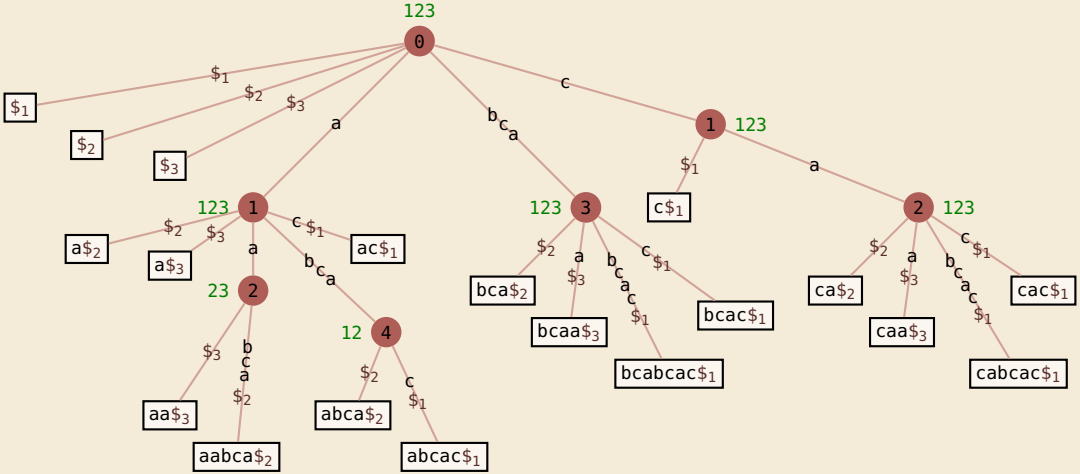
- With that new idea, we can find longest common substrings:
  1. Compute generalized suffix tree $\mathcal{T}$.
  2. Store with each node the *subset of strings* that contain its path label:
     - 2.1. Traverse $\mathcal{T}$ bottom-up.
     - 2.2. For a leaf (j, i), the subset is $\{j\}$.
     - 2.3. 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 $\{1, \ldots, k\}$.

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

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

Longest common substring – Example

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



Further applications

Many more applications, in particular for problems on biological sequences

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

- key ingredient: longest common extensions

If you want to see more, come to *Algorithms of Bioinformatics* 😉

Suffix trees – Discussion

- Suffix trees were a threshold invention


linear time and space


suddenly many questions efficiently solvable in theory

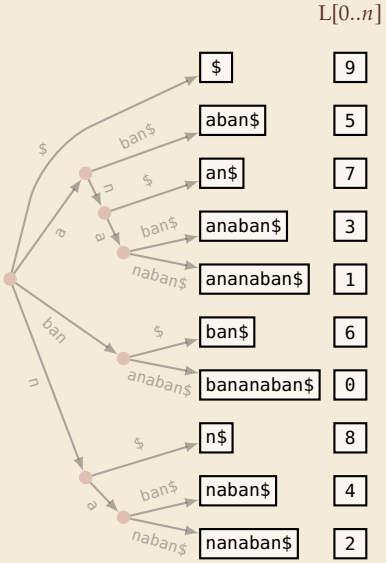

construction of suffix trees:
linear time, but significant overhead


construction methods fairly complicated


many pointers in tree incur large space overhead

13.5 Suffix Arrays

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

⇝ $L[0..n]$ 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

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?

- 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	$alf␣eats␣alfalf**a**	16
lf␣eats␣alfalfa$a	1	␣alfalfa$alf␣eat**s**	8
f␣eats␣alfalfa$al	2	␣eats␣alfalfa$al**f**	3
␣eats␣alfalfa$alf	3	a$alf␣eats␣alfal**f**	15
eats␣alfalfa$alf␣	4	alf␣eats␣alfalfa**$**	0
ats␣alfalfa$alf␣e	5	alfa$alf␣eats␣al**f**	12
ts␣alfalfa$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$al**l**	2
lfalfa$alf␣eats␣a	10	fa$alf␣eats␣alfa**l**	14
falfa$alf␣eats␣al	11	falfa$alf␣eats␣a**l**	11
alfa$alf␣eats␣alf	12	lf␣eats␣alfalfa$**a**	1
lfa$alf␣eats␣alfa	13	lfa$alf␣eats␣alf**a**	13
fa$alf␣eats␣alfal	14	lfalfa$alf␣eats␣**a**	10
a$alf␣eats␣alfalf	15	s␣alfalfa$alf␣eat**t**	7
$alf␣eats␣alfalfa	16	ts␣alfalfa$alf␣e**a**	6

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

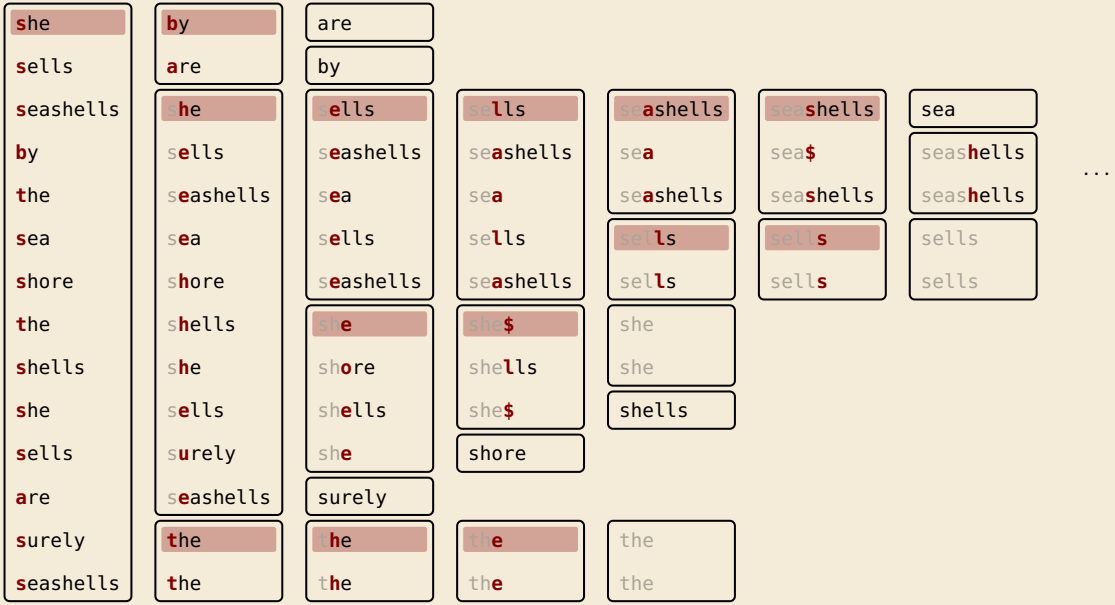
$\rightsquigarrow$ dedicated string sorting methods need to remember common prefixes between strings then we can avoid redoing these comparisons

(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 $d = 0$)
  - $\rightsquigarrow$ 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

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

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
   - duplicates only reduce # 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
   - $\rightsquigarrow$ #LCP comparisons $=$ #comparisons when inserting all strings into a **trie**

Fat-pivot radix quicksort – Discussion


simple to code


efficient for sorting many lists of strings

random string

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


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

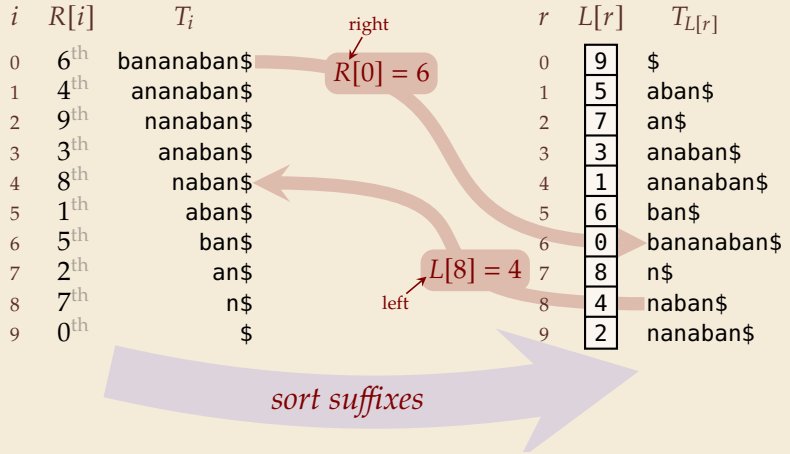
Note: Not quicksort's fault! Any generic string sorting method must take $\Omega(n^2)$ time here

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

13.6 Linear-Time Suffix Sorting: Inducing Order

Inverse suffix array: going left & right

- to understand the fastest algorithm, it is helpful to define the *inverse suffix array*:
  - $R[i] = r \iff L[r] = i$ $\quad L$ = *leaf array*
    $\iff$ there are r suffixes that come before T_i in sorted order
    $\iff$ T_i has (0-based) *rank* r $\rightsquigarrow$ call $R[0..n]$ the ***rank array***



Linear-time suffix sorting

DC3 / Skew algorithm

not a multiple of 3

1. Compute rank array $R_{1,2}$ for suffixes T_i starting at $i \not\equiv 0 \pmod 3$ *recursively.*
2. Induce rank array R_3 for suffixes $T_0, T_3, T_6, T_9, \ldots$ from $R_{1,2}$.
3. Merge $R_{1,2}$ and R_0 using $R_{1,2}$.
   ⇝ rank array R for entire input

- We will show that steps 2. and 3. take $\Theta(n)$ time

⇝ Total complexity is $n + \frac{2}{3}n + \left(\frac{2}{3}\right)^2 n + \left(\frac{2}{3}\right)^3 n + \cdots \leq n \cdot \sum_{i \geq 0} \left(\frac{2}{3}\right)^i = 3n = \Theta(n)$

- **Note:** L can easily be computed from R in one pass, and vice versa.
  ⇝ Can use whichever is more convenient.

DC3 / Skew algorithm – Step 2: Inducing ranks

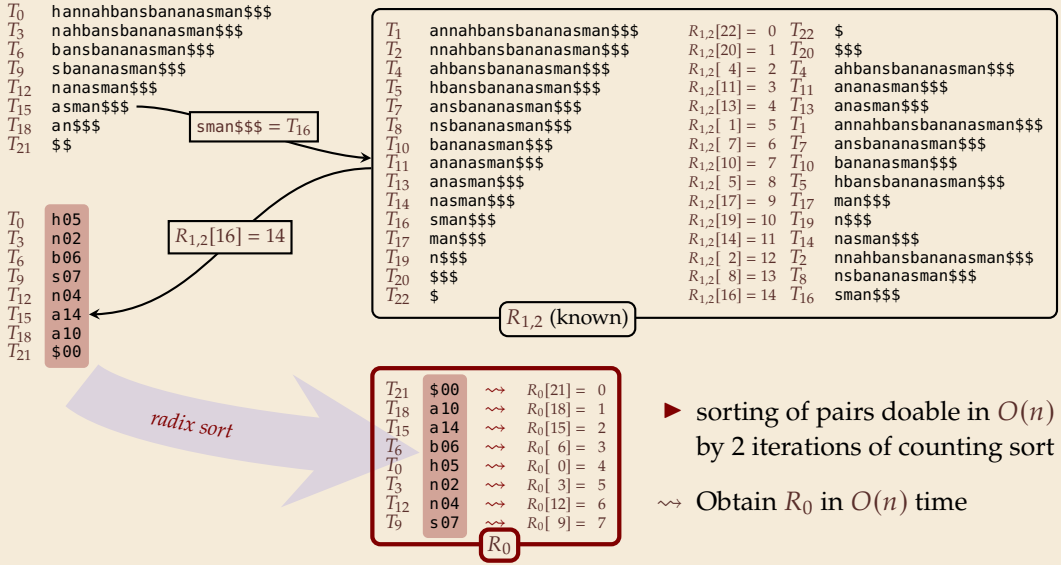
- **Assume:** rank array $R_{1,2}$ known:
  - $R_{1,2}[i] = \begin{cases} \text{rank of } T_i \text{ among } T_1, T_2, T_4, T_5, T_7, T_8, \ldots & \text{for } i = 1, 2, 4, 5, 7, 8, \ldots \\ \text{undefined} & \text{for } i = 0, 3, 6, 9, \ldots \end{cases}$
- **Task:** sort the suffixes T_0, T_3, T_6, T_9, ... in linear time (!)

- Suppose we want to compare T_0 and T_3.
  - Characterwise comparisons too expensive
  - but: after removing first character, we obtain T_1 and T_4
  - these two can be compared in *constant time* by comparing $R_{1,2}[1]$ and $R_{1,2}[4]$!

$\rightsquigarrow$ T_0 comes before T_3 in lexicographic order
iff pair $(T[0], R_{1,2}[1])$ comes before pair $(T[3], R_{1,2}[4])$ in lexicographic order

DC3 / Skew algorithm – Inducing ranks example

T = hannahbansbananasman$$$ (append 3 $ markers)



- sorting of pairs doable in $O(n)$ time by 2 iterations of counting sort

⇝ Obtain R_0 in $O(n)$ time

DC3 / Skew algorithm – Step 3: Merging

T_{21} $$
T_{18} an$$$
T_{15} asman$$$
T_6 bansbananasman$$$
T_0 hannahbansbananasman$$$
T_3 nahbansbananasman$$$
T_{12} nanasman$$$
T_9 sbananasman$$$

T_{22} $
T_{20} $$$
T_4 ahbansbananasman$$$
T_{11} ananasman$$$
T_{13} anasman$$$
T_1 annahbansbananasman$$$
T_7 ansbananasman$$$
T_{10} bananasman$$$
T_5 hbansbananasman$$$
T_{17} man$$$
T_{19} n$$$
T_{14} nasman$$$
T_2 nnahbansbananasman$$$
T_8 nsbananasman$$$
T_{16} sman$$$

T_{22} $
T_{21} $$
T_{20} $$$
T_4 ahbansbananasman$$$
T_{18} an$$$

- Have:
  - sorted 1,2-list:
    $T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \ldots$
  - sorted 0-list:
    $T_0, T_3, T_6, T_9, \ldots$
- Task: Merge them!
  - use standard merging method from Mergesort
  - but speed up comparisons using $R_{1,2}$
  - $\rightsquigarrow$ $O(n)$ time for merge

Compare T_{15} to T_{11}

Idea: try same trick as before

T_{15} = asman$$$
= asman$$$ — can't compare T_{16} and T_{12} either!
= aT_{16}

T_{11} = ananasman$$$
= ananasman$$$
= aT_{12}

$\rightsquigarrow$ Compare T_{16} to T_{12}

T_{16} = sman$$$
= sman$$$ — always at most 2 steps then can use $R_{1,2}$!
= sT_{17}

T_{12} = nanasman$$$
= aanasman$$$
= aT_{13}

13.7 Linear-Time Suffix Sorting: The DC3 Algorithm

DC3 / Skew algorithm – Fix recursive call

- both step 2. and 3. doable in $O(n)$ time!

- But: we cheated in 1. step! *"compute rank array $R_{1,2}$ recursively"*
  - Taking a *subset* of suffixes is *not* an instance of the same problem!

  ⇝ Need a single *string* T' to recurse on, from which we can deduce $R_{1,2}$.

How can we make T' "skip" some suffixes?

redefine alphabet to be *triples of characters* abc

T = bananaban$$$

⇝ $T^{\square}$ = ban ana ban $$$

ana ban $$$

ban $$$

$$$

⇝ suffixes of $T^{\square}$ ⟺ $T_0, T_3, T_6, T_9, \ldots$

- $T' = T[1..n)^{\square}$ $$$ $T[2..n)^{\square}$ $$$ ⟺ T_i with $i \not\equiv 0 \pmod 3$.

⇝ Can call suffix sorting recursively on T' and map result to $R_{1,2}$

DC3 / Skew algorithm – Fix alphabet explosion

- Still does not quite work!
  - Each recursive step *cubes* σ by using triples!
  - $\rightsquigarrow$ (Eventually) cannot use linear-time sorting anymore!

- But: Have at most $\frac{2}{3}n$ different triples $\boxed{\texttt{abc}}$ in T'!

$\rightsquigarrow$ Before recursion:

1. Sort all occurring triples. (using counting sort in $O(n)$)
2. Replace them by their *rank* (in Σ).

$\rightsquigarrow$ Maintains $\sigma \leq n$ without affecting order of suffixes.

DC3 / Skew algorithm – Step 3. Example

$T' = T[1..n)^{\square}\,\boxed{\$\$\$}\,T[2..n)^{\square}\,\boxed{\$\$\$}$

- T = hannahbansbananasman$ T_2 = nnahbansbananasman$
  $T' =$ ann ahb ans ban ana sma n$$ $$$ nna hba nsb ana nas man $$$
- Occurring triples:
  ann ahb ans ban ana sma n$$ $$$ nna hba nsb nas man
- Sorted triples with ranks:

Rank	00	01	02	03	04	05	06	07	08	09	10	11	12
Triple	$$$	ahb	ana	ann	ans	ban	hba	man	n$$	nas	nna	nsb	sma

- $T' =$ ann ahb ans ban ana sma n$$ $$$ nna hba nsb ana nas man $$$
  $T'' =$ 03 01 04 05 02 12 08 00 10 06 11 02 09 07 00

Suffix array – Discussion

- sleek data structure compared to suffix tree
- simple and fast $O(n \log n)$ construction
- more involved but optimal $O(n)$ construction
- supports efficient string matching

- string matching takes $O(m \log n)$, not optimal $O(m)$
- Cannot use more advanced suffix tree features
  e. g., for longest repeated substrings

Outlook: Enhanced Suffix Arrays

- suffix array by itself somewhat less powerful, but can be augmented with the LCP array (longest common prefix)

⇝ *(Enhanced) Suffix Arrays*

  - the modern version of suffix trees
  - directly simulate suffix tree operations on L and LCP arrays

👎 can be harder to reason about

👍 can support same algorithms as suffix trees

👍 use less space

👍 simpler linear-time construction

⇝ Basis for modern *compressed self-indexes* such as *FM index*