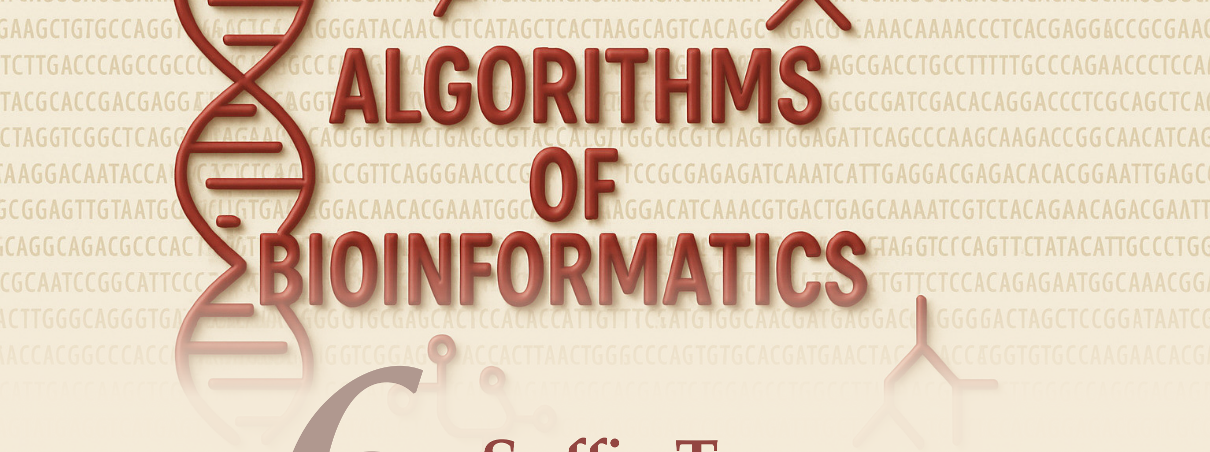


ALGORITHMS OF BIOINFORMATICS

6 Suffix Trees

11 December 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2025-12-19 18:44 H`

Outline

6 Suffix Trees

6.1 Suffix Trees

6.2 Direct Applications

6.3 Generalized Suffix Trees & Augmentation

6.4 Tandem Repeats

6.5 Longest Common Extensions

6.6 Suffix Arrays

6.7 Suffix Sorting: Induced Sorting and Merging

6.8 Suffix Sorting: The DC3 Algorithm

6.9 The LCP Array

6.10 LCP Array Construction

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.

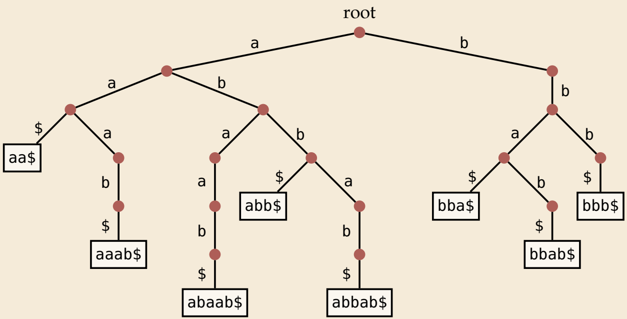
6.1 Suffix Trees

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$, bbb$}`

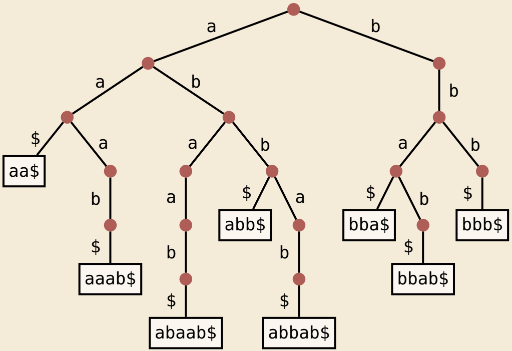


Compact tries

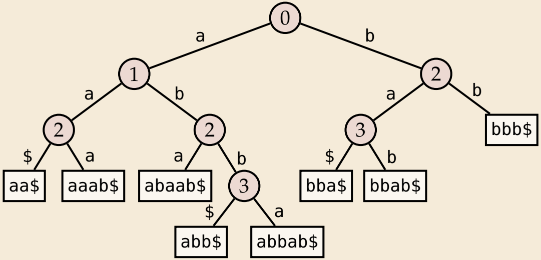
=1 child

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

standard trie



compact trie



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

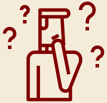
Suffix trees – A ‘magic’ data structure

Appetizer: Longest common substring problem

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

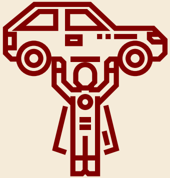
Example: S_1 = `superiorcalifornialives`, S_2 = `sealiver`

$\rightsquigarrow$ `alive`



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

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:

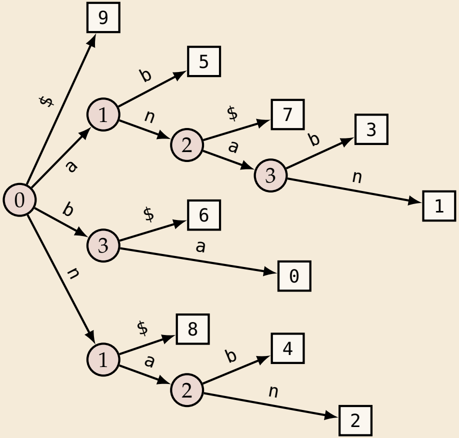
T = bananaban$

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

T =

0	1	2	3	4	5	6	7	8	9
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 $\Theta(n^2)$. ⇝ 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

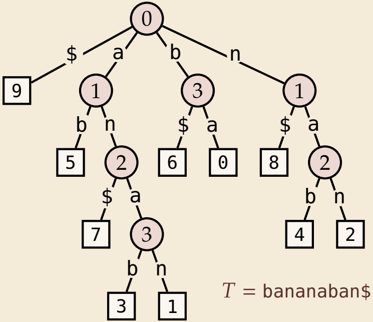
⇝ for now, take linear-time construction for granted. What can we do with them?

6.2 Direct Applications

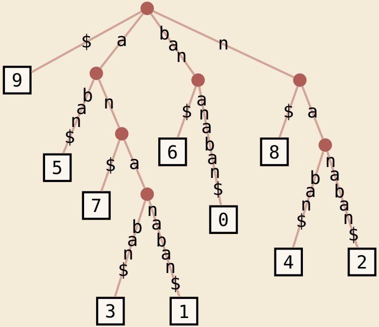
Applications of suffix trees

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

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



T = bananaban$



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

Application 1: Text Indexing / String Matching

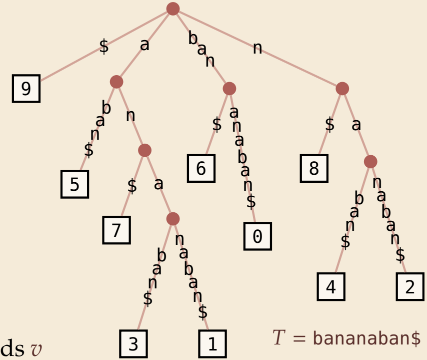
- $\boxed{P \text{ occurs in } T \iff P \text{ is a prefix of a suffix of } T}$
- we have all suffixes in $\mathcal{T}$!

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

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

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

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



T = bananaban$

Examples:

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

Application 2: Longest repeated substring

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

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

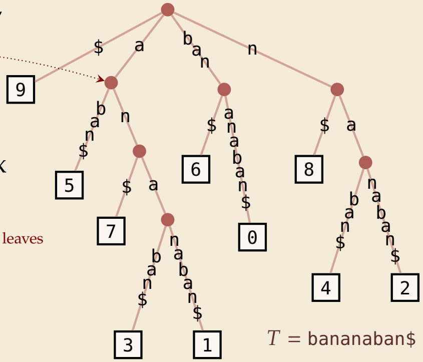
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

$\rightsquigarrow$ $\exists$ internal node with path label '`a`'

here single edge, can be longer path

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

actually: adjacent leaves



T = `bananaban$`

- Algorithm:
  1. Compute *string depth* (=length of path label) of nodes
  2. Find internal nodes with maximal string depth

- Both can be done in depth-first traversal $\rightsquigarrow$ $\Theta(n)$ time

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

⇝ need a *single/joint* suffix tree for *several* texts

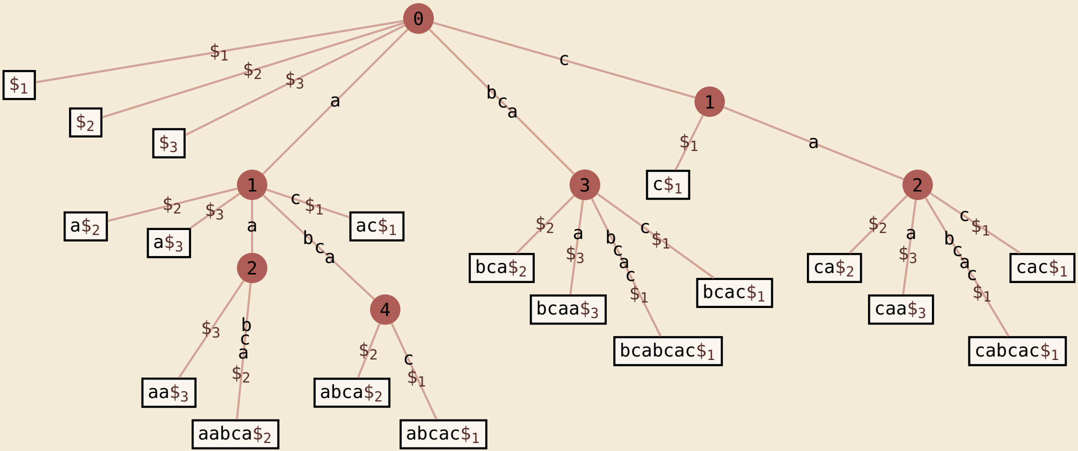
Enter: ***generalized suffix tree***

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

⇝ $\texttt{\$}_\texttt{j}$-edges always leads to leaves ⇝ $\exists$ 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

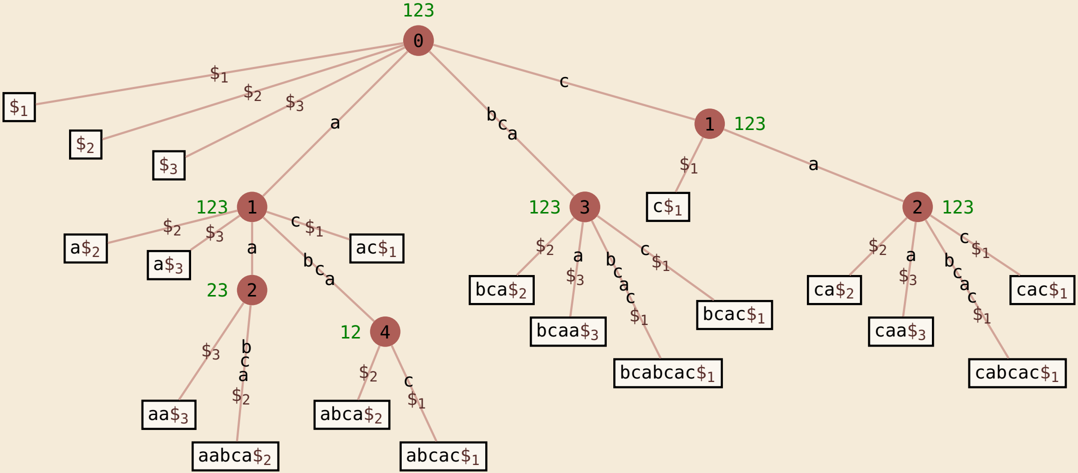
- With that new idea, we can find longest common substrings:

  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

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



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 $C[j]$.

- 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

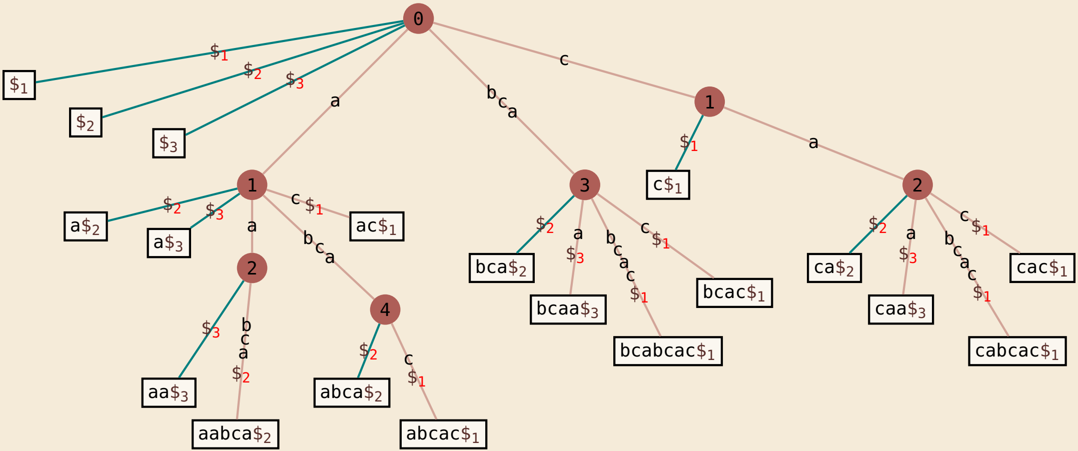
$\rightsquigarrow$ Linear in the total length of P and C.

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.
- ▶ 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
- ⇝ 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

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

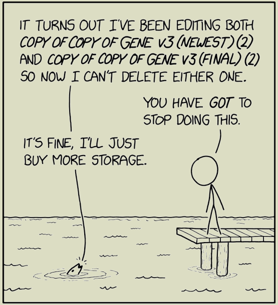


terminal edge

6.4 Tandem Repeats

Repetitions and “Junk DNA”

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

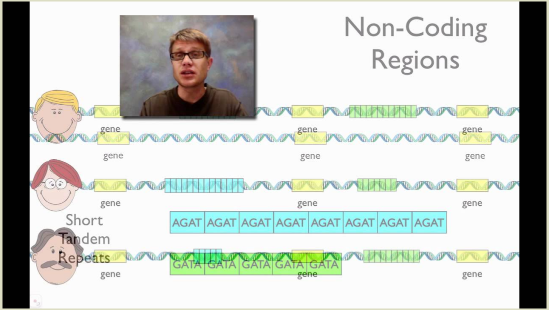

IT TURNS OUT I'VE BEEN EDITING BOTH *COPY OF COPY OF GENE V3 (NEWEST) (2)* AND *COPY OF COPY OF GENE V3 (FINAL) (2)* SO NOW I CAN'T DELETE EITHER ONE.
YOU HAVE *GOT* TO STOP DOING THIS.
IT'S FINE, I'LL JUST BUY MORE STORAGE.

WHY LUNGFISH HAVE SUCH ENORMOUS GENOMES

xkcd.com/3064/

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 $T[i..i+\ell) = T[i+\ell, i+2\ell)$

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

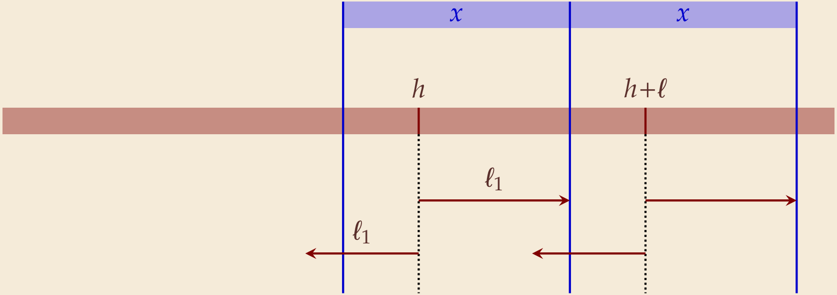
But remember the ***longest common extension (LCE)*** data structure:

- **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



Tandom Repeats – Code

```
procedure tandemRepeats($T[0..n)$)
    Build LCE data structure on $T$ and $T^R$
    // $T^R$ = reversed text
    // We abbreviate $\overrightarrow{e}(i,j) = \mathrm{LCE}_T(i,j)$
    // resp. $\overleftarrow{e}(i,j) = \mathrm{LCE}_{T^R}(n-j, n-i)$
    $tr := \emptyset$
    appendRepeats($T[0..n)$, $tr$)
    return $tr$
```

```
procedure appendRepeats($T[0..n)$, $tr$)
    if $n \le 1$ return
    $h := \lfloor n/2 \rfloor$
    appendRepeats($T[0..h)$)
    appendRepeats($T[h..n)$)
    // Add all repeats $xx$ with $h$ inside first copy of $x$
    for $\ell \in [1 .. n-h)$ // length of $x$
        $q := h + \ell$
        $\ell_1 := \overrightarrow{e}(h, q)$
        $\ell_2 := \overleftarrow{e}(h-1, q-1)$
        if $\ell_1 + \ell_2 \ge \ell$
            $tr := tr \cup \{(j, \ell) : j \in [h - \ell_2 \;..\; h + \ell_1 - \ell]\}$
    // Symmetrically for $h$ inside second copy of $x$
    for $\ell \in [1..h]$
        $q := h - \ell$
        $\ell_1 := \overrightarrow{e}(h, q)$
        $\ell_2 := \overleftarrow{e}(h-1, q-1)$
        if $\ell_1 + \ell_2 \ge \ell$
            $tr := tr \cup \{(j, \ell) : j \in [q - \ell_2 \;..\; q + \ell_1 - \ell]\}$
```

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

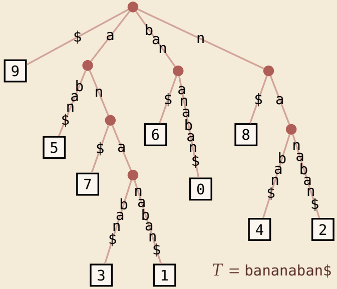
Longest Common Extensions & Suffix Trees

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

$\rightsquigarrow$ use suffix tree of T!



T = bananaban$

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

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

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

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 👎

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

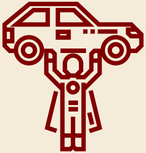
and suffix tree construction inside . . .

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

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

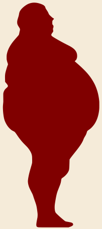
Suffix trees – Discussion

- Suffix trees were a threshold invention



👍 linear time and space

👍 suddenly many questions efficiently solvable in theory



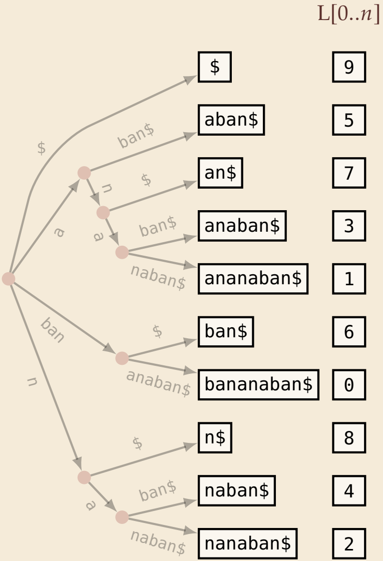
👎 construction of suffix trees:
linear time, but significant overhead

👎 construction methods fairly complicated

👎 many pointers in tree incur large space overhead

6.6 Suffix Arrays

Putting suffix trees on a diet



	L[0..n]
$	9
aban$	5
an$	7
anaban$	3
ananaban$	1
ban$	6
bananaban$	0
n$	8
naban$	4
nanaban$	2

- **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***:

$\boxed{L[r] = \text{(start index of) } r\text{th suffix in sorted order}}$

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

- 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 $d = 0$)

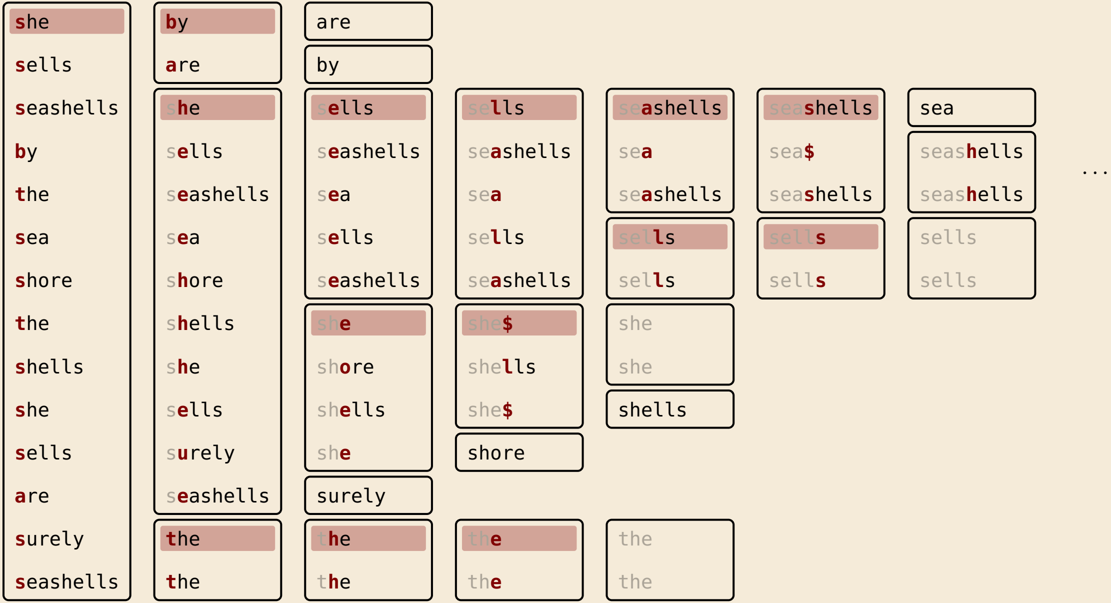
  $\rightsquigarrow$ 3 segments: smaller, equal, or larger than dth symbol of pivot

  - 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



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

by are

she sells seashells sea shore shells she sells surely seashells

the the

are

by

sells seashells sea sells seashells

she shore shells she

surely

the the

sells seashells sea sells seashells

she$ shells she$

shore

the the

seashells sea seashells

sells sells

she she

shells

the the

seashells sea$ seashells

sells sells

sea

seashells seashells

sells sells

. . .

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**

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., $T = \mathsf{a}^n$

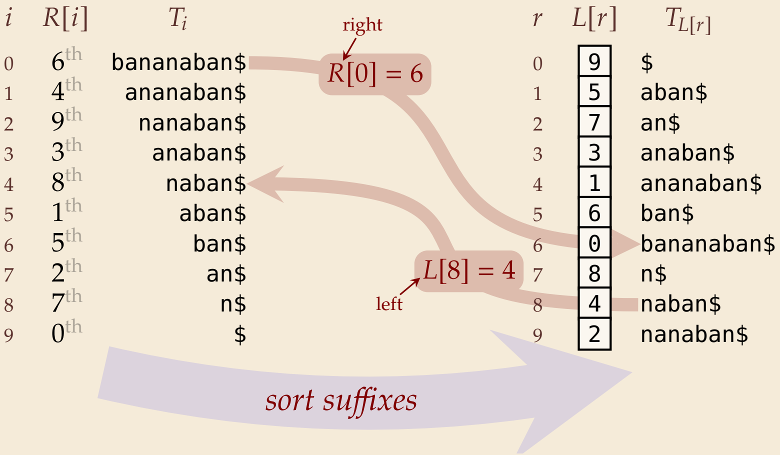
  Note: Not quicksort's fault! Any generic string sorting method must take $\Omega(n^2)$ time here

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

6.7 Suffix Sorting: Induced Sorting and Merging

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***



i	$R[i]$	T_i
0	6th	bananaban$
1	4th	ananaban$
2	9th	nanaban$
3	3th	anaban$
4	8th	naban$
5	1th	aban$
6	5th	ban$
7	2th	an$
8	7th	n$
9	0th	$

r	$L[r]$	$T_{L[r]}$
0	9	$
1	5	aban$
2	7	an$
3	3	anaban$
4	1	ananaban$
5	6	ban$
6	0	bananaban$
7	8	n$
8	4	naban$
9	2	nanaban$

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 $i \not\equiv \underset{\swarrow}{0} \pmod 3$ *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 $R_{1,2}$.

   $\rightsquigarrow$ rank array R for entire input

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

$\rightsquigarrow$ Total complexity is $n + \frac{2}{3}n + \left(\frac{2}{3}\right)^2 n + \left(\frac{2}{3}\right)^3 n + \cdots \;\le\; n \cdot \sum_{i \ge 0} \left(\frac{2}{3}\right)^i \;=\; 3n \;=\; \Theta(n)$

- ► **Note:** L can easily be computed from R in one pass, and vice versa.

  $\rightsquigarrow$ Can use whichever is more convenient.

DC3 / Skew algorithm – Step 2: Inducing ranks

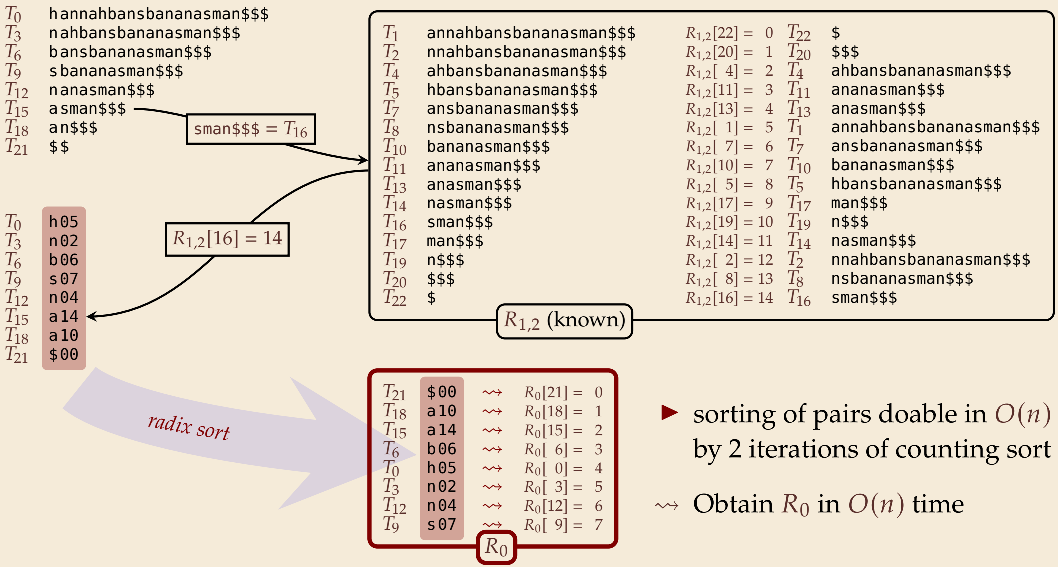
- **Assume:** rank array $R_{1,2}$ known:
  - $$R_{1,2}[i] = \begin{cases} \text{rank of } T_i \text{ among } T_1, T_2, T_4, T_5, T_7, T_8, \ldots & \text{for } i = 1, 2, 4, 5, 7, 8, \ldots \\ \text{undefined} & \text{for } i = 0, 3, 6, 9, \ldots \end{cases}$$
- **Task:** sort the suffixes $T_0, T_3,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



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

`sman$$$` = T_{16}

T_1	`annahbansbananasman$$$`	$R_{1,2}[22] = 0$	T_{22}	`$`
T_2	`nnahbansbananasman$$$`	$R_{1,2}[20] = 1$	T_{20}	`$$$`
T_4	`ahbansbananasman$$$`	$R_{1,2}[\ 4] = 2$	T_4	`ahbansbananasman$$$`
T_5	`hbansbananasman$$$`	$R_{1,2}[11] = 3$	T_{11}	`ananasman$$$`
T_7	`ansbananasman$$$`	$R_{1,2}[13] = 4$	T_{13}	`anasman$$$`
T_8	`nsbananasman$$$`	$R_{1,2}[\ 1] = 5$	T_1	`annahbansbananasman$$$`
T_{10}	`bananasman$$$`	$R_{1,2}[\ 7] = 6$	T_7	`ansbananasman$$$`
T_{11}	`ananasman$$$`	$R_{1,2}[10] = 7$	T_{10}	`bananasman$$$`
T_{13}	`anasman$$$`	$R_{1,2}[\ 5] = 8$	T_5	`hbansbananasman$$$`
T_{14}	`nasman$$$`	$R_{1,2}[17] = 9$	T_{17}	`man$$$`
T_{16}	`sman$$$`	$R_{1,2}[19] = 10$	T_{19}	`n$$$`
T_{17}	`man$$$`	$R_{1,2}[14] = 11$	T_{14}	`nasman$$$`
T_{19}	`n$$$`	$R_{1,2}[\ 2] = 12$	T_2	`nnahbansbananasman$$$`
T_{20}	`$$$`	$R_{1,2}[\ 8] = 13$	T_8	`nsbananasman$$$`
T_{22}	`$`	$R_{1,2}[16] = 14$	T_{16}	`sman$$$`

$R_{1,2}$ (known)

$R_{1,2}[16] = 14$

T_0	`h05`
T_3	`n02`
T_6	`b06`
T_9	`s07`
T_{12}	`n04`
T_{15}	`a14`
T_{18}	`a10`
T_{21}	`$00`

radix sort

T_{21}	`$00`	⇝	$R_0[21] = 0$
T_{18}	`a10`	⇝	$R_0[18] = 1$
T_{15}	`a14`	⇝	$R_0[15] = 2$
T_6	`b06`	⇝	$R_0[\ 6] = 3$
T_0	`h05`	⇝	$R_0[\ 0] = 4$
T_3	`n02`	⇝	$R_0[\ 3] = 5$
T_{12}	`n04`	⇝	$R_0[12] = 6$
T_9	`s07`	⇝	$R_0[\ 9] = 7$

R_0

- 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 $R_{1,2}$

  ⇝ $O(n)$ time for merge

Compare T_{15} to T_{11}

Idea: try same trick as before

$$\begin{aligned} T_{15} &= \texttt{asman\$\$\$} \\ &= \texttt{asman\$\$\$} \\ &= \texttt{a}T_{16} \end{aligned}$$

can't compare T_{16} and T_{12} either!

$$\begin{aligned} T_{11} &= \texttt{ananasman\$\$\$} \\ &= \texttt{ananasman\$\$\$} \\ &= \texttt{a}T_{12} \end{aligned}$$

⇝ Compare T_{16} to T_{12}

$$\begin{aligned} T_{16} &= \texttt{sman\$\$\$} \\ &= \texttt{sman\$\$\$} \\ &= \texttt{s}T_{17} \end{aligned}$$

always at most 2 steps then can use $R_{1,2}$!

$$\begin{aligned} T_{12} &= \texttt{nanasman\$\$\$} \\ &= \texttt{aanasman\$\$\$} \\ &= \texttt{a}T_{13} \end{aligned}$$

6.8 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* $\boxed{\texttt{abc}}$

$T = \texttt{bananaban\$\$\$}$

⇝ $T^{\square} = \boxed{\texttt{ban}}\boxed{\texttt{ana}}\boxed{\texttt{ban}}\boxed{\texttt{\$\$\$}}$

$\boxed{\texttt{ana}}\boxed{\texttt{ban}}\boxed{\texttt{\$\$\$}}$

$\boxed{\texttt{ban}}\boxed{\texttt{\$\$\$}}$

$\boxed{\texttt{\$\$\$}}$

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

- $T' = T[1..n)^{\square}\,\boxed{\texttt{\$\$\$}}\,T[2..n)^{\square}\,\boxed{\texttt{\$\$\$}}$ ⇜⇝ 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!
  - ⇝ (Eventually) cannot use linear-time sorting anymore!

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

⇝ Before recursion:

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

⇝ Maintains $\sigma \le n$ without affecting order of suffixes.

DC3 / Skew algorithm – Step 3. Example

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

- T = `hannahbansbananasman$` T_2 = `nnahbansbananasman$`

  $T' = \boxed{\texttt{ann}}\boxed{\texttt{ahb}}\boxed{\texttt{ans}}\boxed{\texttt{ban}}\boxed{\texttt{ana}}\boxed{\texttt{sma}}\boxed{\texttt{n\$\$}}\;\boxed{\texttt{\$\$\$}}\;\boxed{\texttt{nna}}\boxed{\texttt{hba}}\boxed{\texttt{nsb}}\boxed{\texttt{ana}}\boxed{\texttt{nas}}\boxed{\texttt{man}}\boxed{\texttt{\$\$\$}}$

- Occurring triples:

  $\boxed{\texttt{ann}}\boxed{\texttt{ahb}}\boxed{\texttt{ans}}\boxed{\texttt{ban}}\boxed{\texttt{ana}}\boxed{\texttt{sma}}\boxed{\texttt{n\$\$}}\;\boxed{\texttt{\$\$\$}}\;\boxed{\texttt{nna}}\boxed{\texttt{hba}}\boxed{\texttt{nsb}}\qquad\boxed{\texttt{nas}}\boxed{\texttt{man}}$

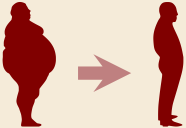
- Sorted triples with ranks:

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

- $T' = \boxed{\texttt{ann}}\boxed{\texttt{ahb}}\boxed{\texttt{ans}}\boxed{\texttt{ban}}\boxed{\texttt{ana}}\boxed{\texttt{sma}}\boxed{\texttt{n\$\$}}\;\boxed{\texttt{\$\$\$}}\;\boxed{\texttt{nna}}\boxed{\texttt{hba}}\boxed{\texttt{nsb}}\boxed{\texttt{ana}}\boxed{\texttt{nas}}\boxed{\texttt{man}}\boxed{\texttt{\$\$\$}}$

  $T'' = \boxed{\texttt{03}}\;\boxed{\texttt{01}}\;\boxed{\texttt{04}}\;\boxed{\texttt{05}}\;\boxed{\texttt{02}}\;\boxed{\texttt{12}}\;\boxed{\texttt{08}}\;\;\boxed{\texttt{00}}\;\;\boxed{\texttt{10}}\;\boxed{\texttt{06}}\;\boxed{\texttt{11}}\;\boxed{\texttt{02}}\;\boxed{\texttt{09}}\;\boxed{\texttt{07}}\;\boxed{\texttt{00}}$

Suffix array – Discussion

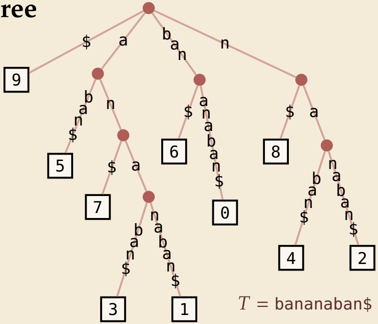


- 👍 sleek data structure compared to suffix tree
- 👍 simple and fast $O(n \log n)$ construction
- 👍 more involved but optimal $O(n)$ construction
- 👍 supports efficient string matching

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

6.9 The LCP Array

String depths of internal nodes



T = bananaban$

- Recall algorithm for longest repeated substring in **suffix tree**

  1. Compute *string depth* of nodes
  2. Find *path label to node* with maximal string depth

- Can we do this using **suffix *arrays***?

- Yes, by **enhancing** the suffix array with the ***LCP array***!

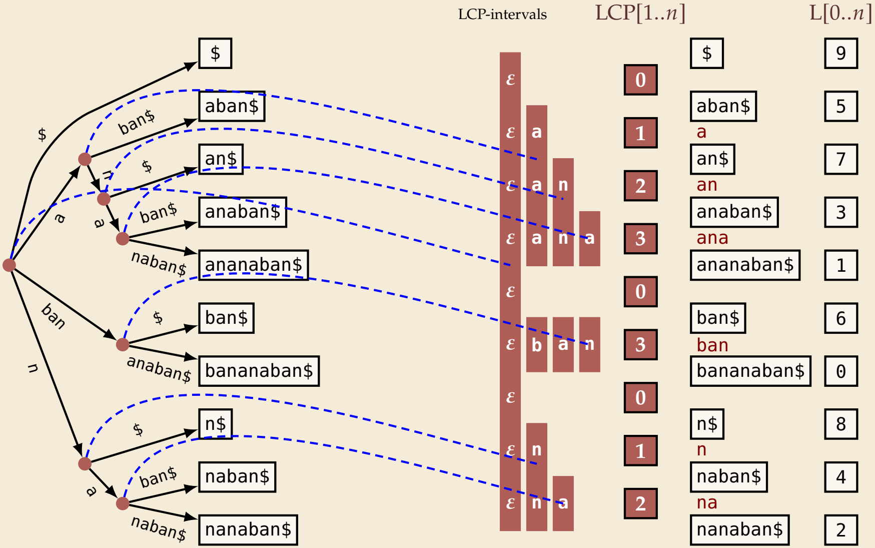
  $\mathrm{LCP}[1..n]$

  $\mathrm{LCP}[r] \;=\; \mathrm{LCP}(T_{L[r]}, T_{L[r-1]})$

length of longest common prefix of suffixes of rank r and $r-1$

⇝ longest repeated substring = find maximum in $\mathrm{LCP}[1..n]$

LCP array and internal nodes



⇝ Leaf array *L*[0..*n*] plus LCP array LCP[1..*n*] encode full tree!

6.10 LCP Array Construction

LCP array construction

- computing LCP$[1..n]$ naively too expensive
  - each value could take $\Theta(n)$ time
  - 👎 $\Theta(n^2)$ in total
- but: seeing one large (= costly) LCP value ⇝ can find another large one!
- Example: T = `Buffalo␣buffalo␣buffalo␣buffalo$`
  - first few suffixes in sorted order:

$T_{L[0]}$ = `$`
$T_{L[1]}$ = `alo␣buffalo$`
$T_{L[2]}$ = `alo␣buffalo␣buffalo$`
`alo␣buffalo␣buffalo` ⇝ LCP[3] = **19**
$T_{L[3]}$ = `alo␣buffalo␣buffalo␣buffalo$`

⇝ **Removing first character** from $T_{L[2]}$ and $T_{L[3]}$ gives two new suffixes:

$T_{L[?]}$ = `lo␣buffalo␣buffalo$`
`lo␣buffalo␣buffalo` ⇝ LCP[**?**] = **18**
$T_{L[?]}$ = `lo␣buffalo␣buffalo␣buffalo$`
unclear where. . .

⚠ Shortened suffixes might *not* be *adjacent* in sorted order!
⇝ no LCP entry for them!

Kasai's algorithm – Example

- Kasai et al. used above observation systematically
- Key idea: *compute* LCP values in ***text order***
- Dropping first character of adjacent suffixes might not lead to *adjacent* shorter suffixes, but LCP entry can only be *longer*.

i	$R[i]$	T_i
0	6th	`bananaban$`
1	4th	`ananaban$`
2	9th	`nanaban$`
3	3th	`anaban$`
4	8th	`naban$`
5	1th	`aban$`
6	5th	`ban$`
7	2th	`an$`
8	7th	`n$`
9	0th	`$`

r	$L[r]$	$T_{L[r]}$	$\mathrm{LCP}[r]$
0	9	`$`	–
1	5	`aban$`	0
2	7	`an$`	1
3	3	`anaban$`	2
4	1	`ananaban$`	3
5	6	`ban$`	0
6	0	`bananaban$`	3
7	8	`n$`	0
8	4	`naban$`	1
9	2	`nanaban$`	2

Kasai's algorithm – Code

```
procedure computeLCP($T[0..n]$, $L[0..n]$, $R[0..n]$):
    // Assume $T[n] = \$$, $L$ and $R$ are suffix array and inverse
    $\ell := 0$
    for $i := 0, \ldots, n-1$ // Consider $T_i$ now
        $r := R[i]$
        // compute $\mathrm{LCP}[r]$; note that $r > 0$ since $R[n] = 0$
        $i_{-1} := L[r-1]$
        while $T[i+\ell] == T[i_{-1}+\ell]$ do
            $\ell := \ell + 1$
        $\mathrm{LCP}[r] := \ell$
        $\ell := \max\{\ell - 1, 0\}$
    return $\mathrm{LCP}[1..n]$
```

- remember length ℓ of induced common prefix
- use L to get start index of suffixes

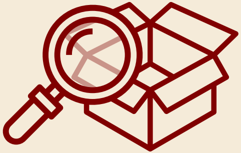
Analysis:

- dominant operation:
  character comparisons
- separately count those with
  outcomes "$=$" resp. "$\neq$"
- each $\neq$ ends iteration of for-loop
  $\rightsquigarrow$ $\le n$ cmps
- each $=$ implies increment of ℓ,
  but $\ell \le n$ and
  decremented $\le n$ times
  $\rightsquigarrow$ $\le 2n$ cmps

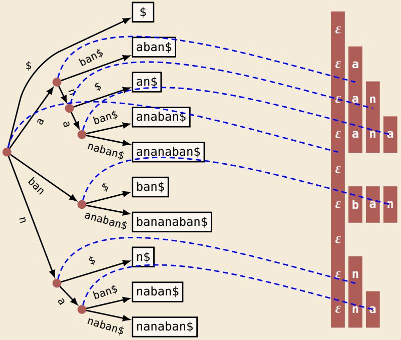
$\rightsquigarrow$ $\Theta(n)$ overall time

Back to suffix trees

We can finally look into the black box of linear-time suffix-array construction!



1. Compute suffix array for T.
2. Compute LCP array for T.
3. Construct $\mathcal{T}$ from suffix array and LCP array.



LCP[1..n]			L[0..n]
		$	9
0		aban$	5
1	a	an$	7
2	an	anaban$	3
3	ana	ananaban$	1
0		ban$	6
3	ban	bananaban$	0
0		n$	8
1	n	naban$	4
2	na	nanaban$	2

Conclusion

- *(Enhanced) Suffix Arrays* are 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

👍 but use much less space

👍 simpler linear-time construction

Outlook:

- enhanced suffix arrays still need original text T to work
- a *self-index* avoids that
  - can store T in *compressed* form **and** support operations like string matching

$\rightsquigarrow$ *Advanced Data Structures*