



ALGORITHMS OF BIOINFORMATICS

6

Suffix Trees

11 December 2025

Prof. Dr. Sebastian Wild

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.

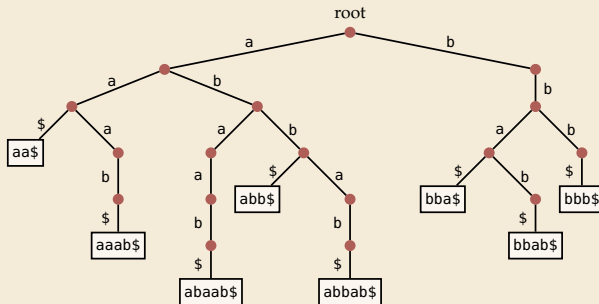
6.1 Suffix Trees

Recap: Tries

- ▶ efficient dictionary data structure for strings (or for Aho-Corasick automata 😊)
- ▶ name from **re**trieval, 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 ✓ some character $\notin \Sigma$
 - ▶ strings have “end-of-string” marker \$ ✓

▶ Example:

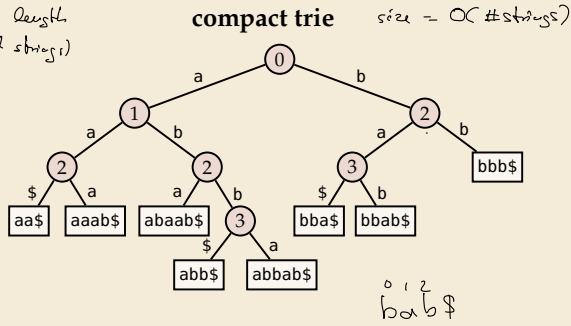
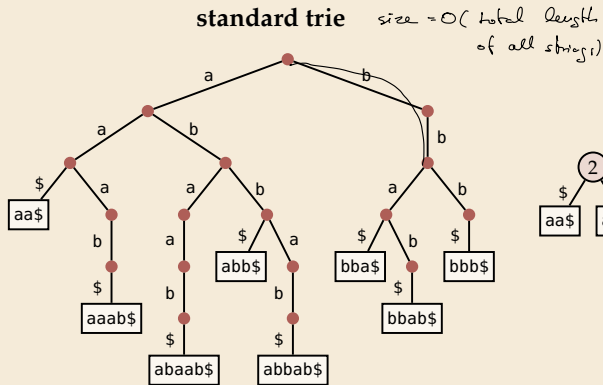
{aa\$, aaab\$, abaab\$, abb\$,
abbab\$, bba\$, bbab\$, bbb\$}



Compact tries

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

=1 child



- ▶ 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, \dots, S_k **Example:** $S_1 = \text{superiorcalifornialives}$, $S_2 = \text{sealiver}$
- ▶ 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, \dots, S_k

Example: $S_1 = \text{superiorcalifornialives}$, $S_2 = \text{sealiver}$

► Goal: find the longest substring that occurs in all k strings

\rightsquigarrow alive



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

Suffix trees – A ‘magic’ data structure

Appetizer: Longest common substring problem

▶ Given: strings S_1, \dots, S_k

Example: $S_1 = \text{superiorcalifornialives}$, $S_2 = \text{sealiver}$

▶ Goal: find the longest substring that occurs in all k strings

↪ alive



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

Enter: *suffix trees*

- ▶ versatile data structure for index with full-text search
- ▶ linear time (for construction) and linear space
- ▶ allows efficient solutions for many advanced string problems



“Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible.”

[Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]

Suffix trees – Definition

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

Suffix trees – Definition

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

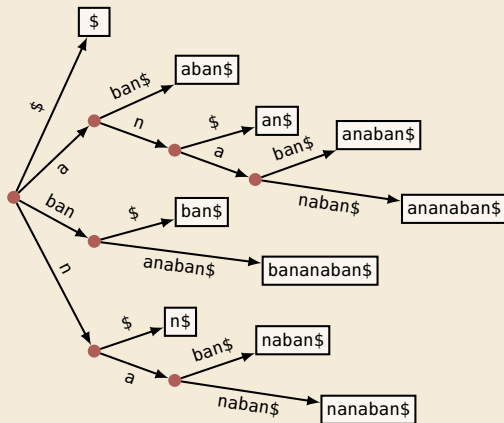
Example:

$T = \text{bananaban\$}$

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

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

$T =$



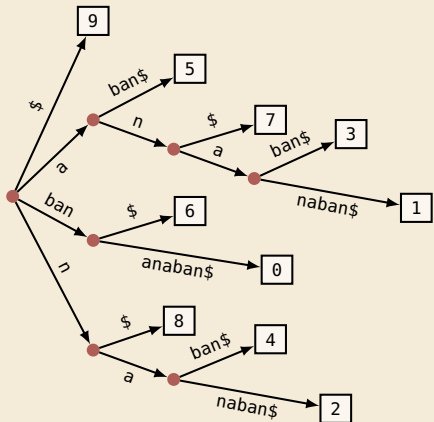
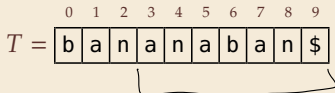
Suffix trees – Definition

- ▶ 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:

$T = \text{bananaban\$}$

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



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)

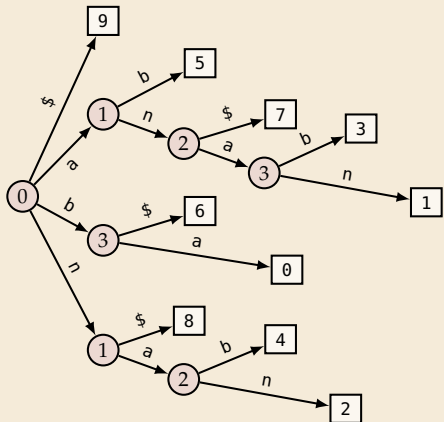
Example:

$T = \text{bananaban\$}$

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

	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 $\Theta(\overset{\text{same order of growth as reading the text!}}{n})$ time!

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

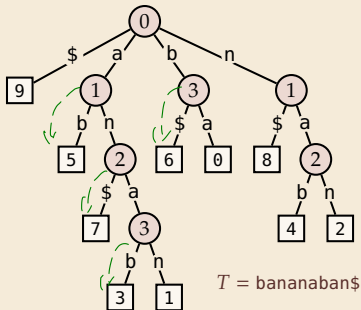
\rightsquigarrow for now, take linear-time construction for granted. What can we do with them?

6.2 Direct Applications

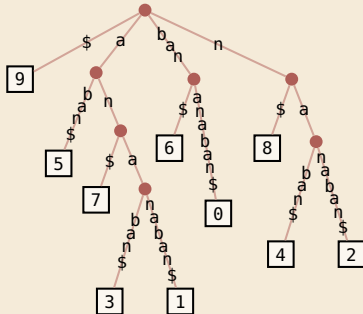
Applications of suffix trees

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

Recall: \mathcal{T} stored like this:



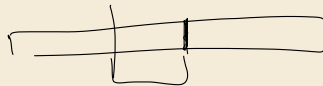
but think about this:



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

Application 1: Text Indexing / String Matching

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



Application 1: Text Indexing / String Matching

► P occurs in $T \iff P$ is a prefix of a suffix of T

► we have all suffixes in \mathcal{T} !

↪ (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

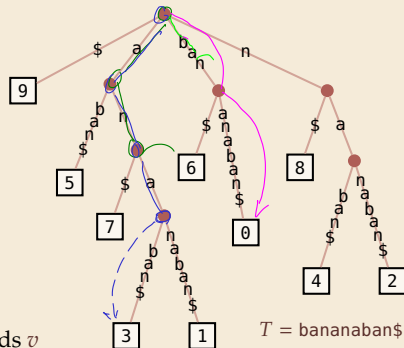
3. **we run out of tree**

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!



Examples:

► $P = \underline{\text{ann}}$

► $P = \underline{\text{baa}}$

► $P = \underline{\text{ana}}$

► $P = \text{ba}$

► $P = \underline{\text{briar}}$

Application 2: Longest repeated substring

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



How can we efficiently check *all possible substrings*?

Application 2: Longest repeated substring

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



How can we efficiently check *all possible substrings*?



Repeated substrings = shared paths in *suffix tree*



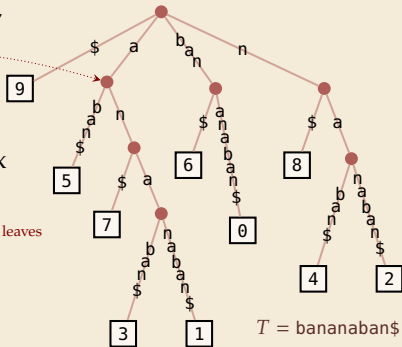
- $T_5 = \text{aban\$}$ and $T_7 = \text{an\$}$ have *longest common prefix* 'a'

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

here single edge, can be longer path

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

actually: adjacent leaves



Application 2: Longest repeated substring

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



How can we efficiently check *all possible substrings*?



Repeated substrings = shared paths in *suffix tree*



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

↪ \exists internal node with path label 'a'

here single edge, can be longer path

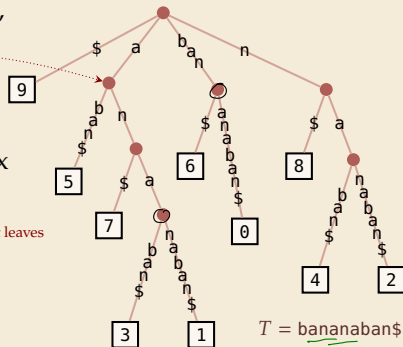
↪ longest repeated substring = longest common prefix (LCP) of two suffixes

actually: adjacent leaves

- **Algorithm:**

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

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



6.3 Generalized Suffix Trees & Augmentation

Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \dots, T^{(k)}$ 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

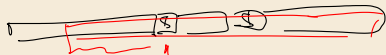
Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \dots, T^{(k)}$ 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

Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \dots, T^{(k)}$ 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*

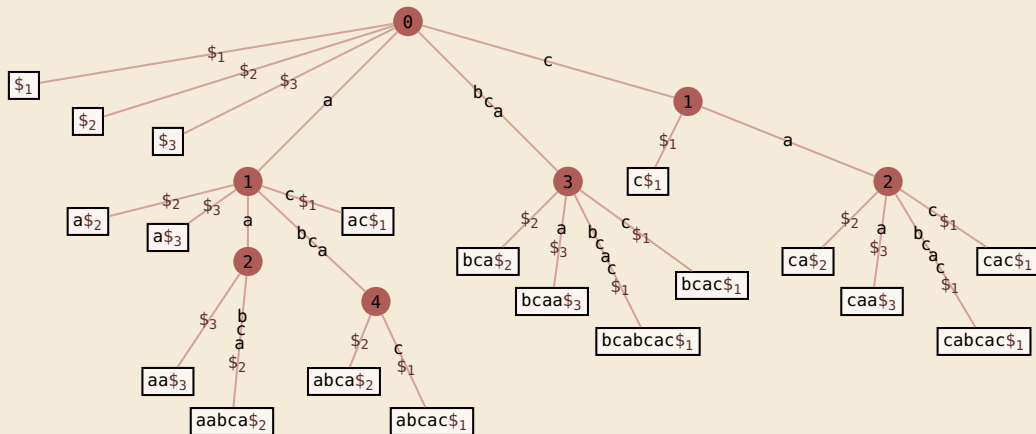
- ▶ Define $T := T^{(1)}\underline{\$}_1 T^{(2)}\underline{\$}_2 \dots T^{(k)}\underline{\$}_k$ for k new end-of-word symbols
- ▶ Construct suffix tree \mathcal{T} for T

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



Generalized Suffix Tree – Example

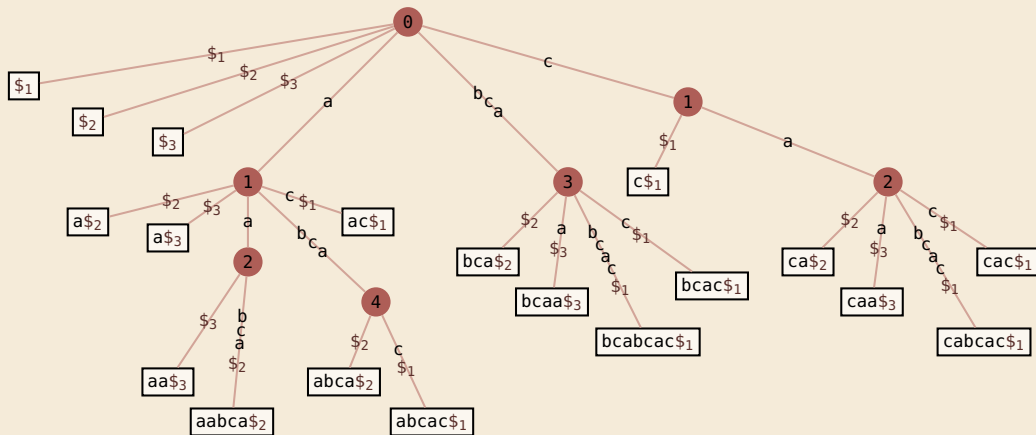
$T^{(1)} = \text{bcabcac}$, $T^{(2)} = \text{aabca}$, $T^{(3)} = \text{bcaa}$



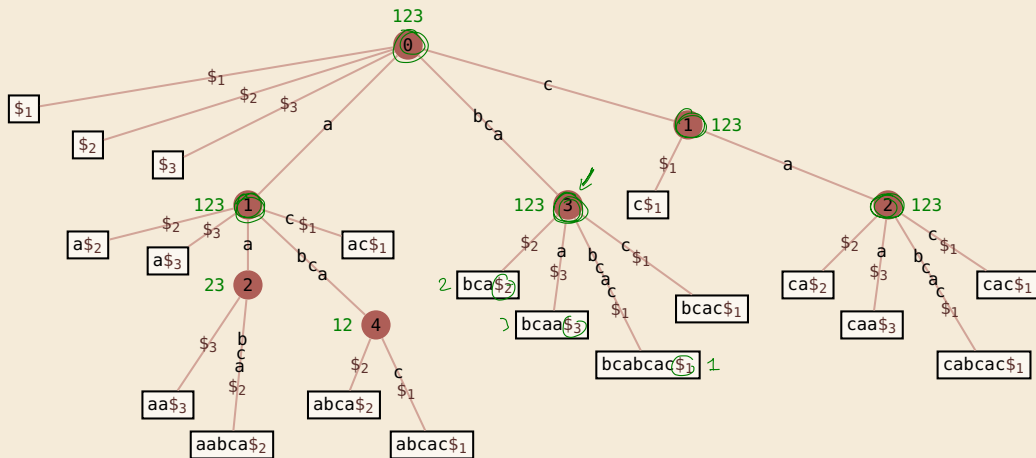
Application 3: Longest common substring

- ▶ 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 $\{1, \dots, k\}$.
- ▶ Steps 1, 3, 4 take time $\Theta(n)$ for $n = n_1 + \dots + n_k$ 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

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


Longest common substring – Example

$$T^{(1)} = \text{bcab}\underline{\text{cac}}, \quad T^{(2)} = \text{aab}\underline{\text{ca}}, \quad T^{(3)} = \underline{\text{bcaa}}$$


Application 4: DNA Sample Contamination

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

Application 4: DNA Sample Contamination

► **Given:** new reads $P[0..p)$, known contamination sources $C[0..c)$, length ℓ

► **Goal:** indices $I \subseteq [0..p)$ of reads that
do **not** have common substring of length $\geq \ell$
with **any** contamination source $C[j]$.

► Solution similar to longest common substring

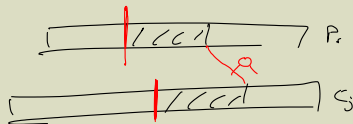
► Build generalized suffix tree \mathcal{T} of $P[0..p)$ and $C[0..c)$

► $P[r]$ has common substring with $C[j]$ of length $\geq \ell$
iff \exists node v in \mathcal{T} of string depth $\geq \ell$ and leaves from $P[r]$ and $C[j]$ below.

\rightsquigarrow Mark each node in \mathcal{T} in a bottom-up traversal if its subtree contains any leaves from any $C[j]$.

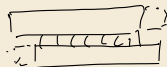
► For all marked nodes v of string depth $\geq \ell$, add r to I for all $P[r]$ with a leaf in subtree of v

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



Application 5: Computing the Overlap Graph

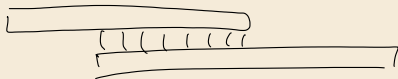
- Recall the genome sequencing problem with inexact reads



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

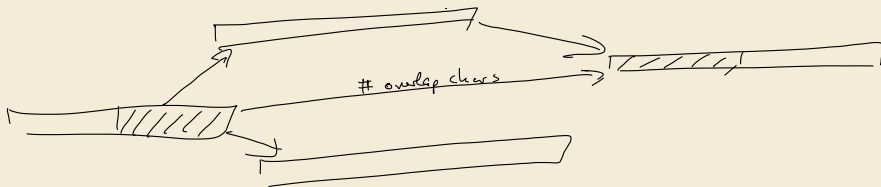
- Here: Overlap graph with edge weights = **length of exact suffix-prefix match**

↪ **All Pairs Suffix-Prefix Matching**



- **Given:** reads $P[0..p)$, $P[r] = P_r[0..m_r)$

- **Goal:** $O[0..p)[0..p)$ where $O[i][j]$ is length of longest suffix of P_i that is a prefix of P_j .



Application 5: Computing the Overlap Graph

- ▶ Recall the genome sequencing problem with inexact reads

↪ 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}

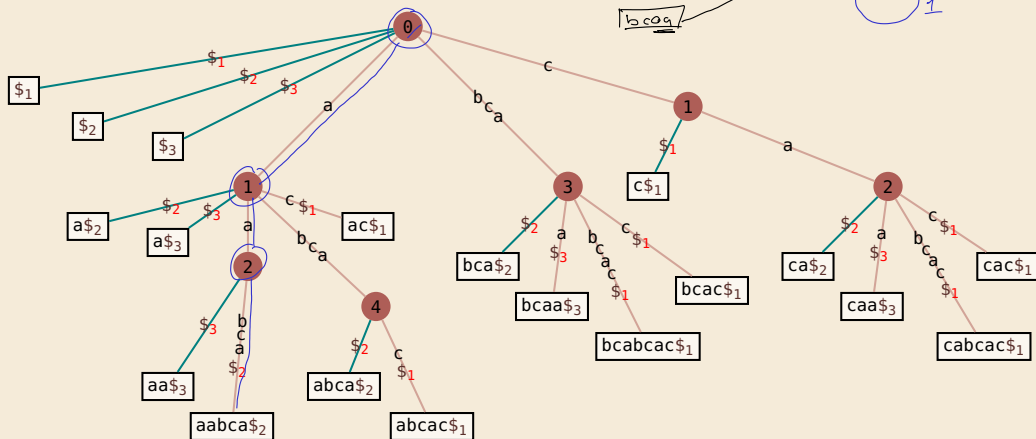
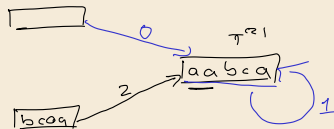
- ▶ Mark edges labeled only with $\$$ as *terminal edges*

- ▶ Suffix of P_i equals prefix of P_j iff traversal of P_j in \mathcal{T} reaches $\$$ 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

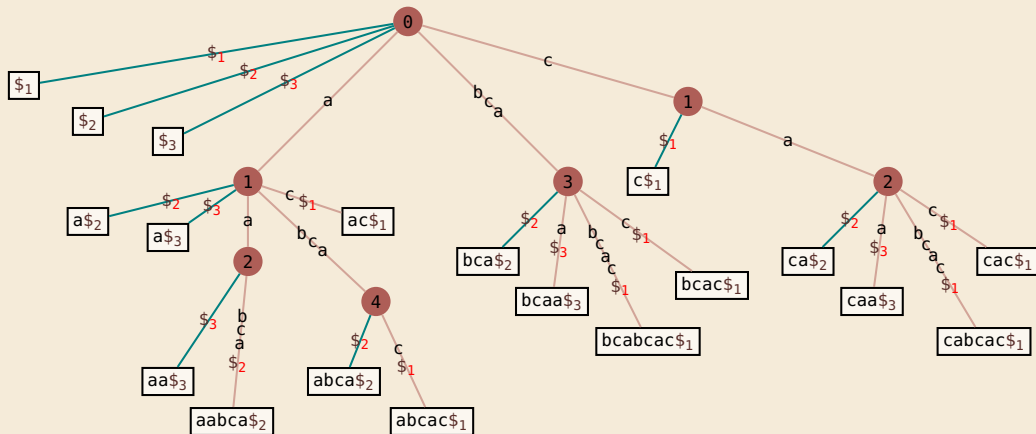
$T^{(1)} = \text{bcabcac}$, $T^{(2)} = \text{aabca}$, $T^{(3)} = \text{bcaa}$



terminal edge

All-Pairs Suffix-Prefix – Example

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



terminal edge

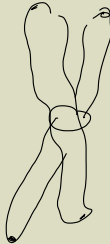
6.4 Tandem Repeats

Repetitions and “Junk DNA”

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

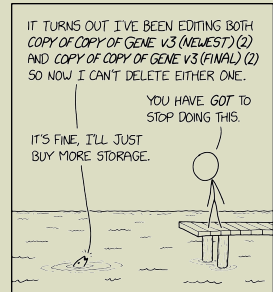
Repetitions and “Junk DNA”

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



Repetitions and “Junk DNA”

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

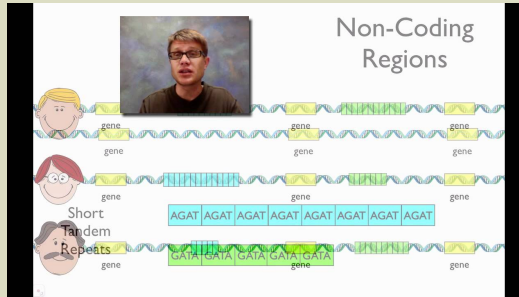


WHY LUNGFISH HAVE SUCH ENORMOUS GENOMES

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

Tandem Repeats

Formally, finding tandem repeats amounts to the following problem.

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

How can suffix trees help us here?

Tandem Repeats

Formally, finding tandem repeats amounts to the following problem.

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

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

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

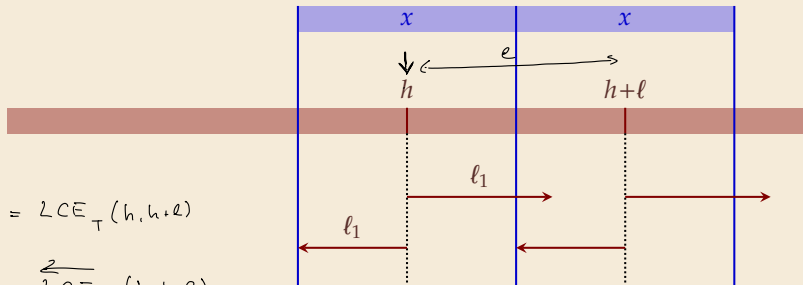
- ▶ **Given:** String $T[0..n)$
- ▶ **Goal:** Answer queries $\text{LCE}_T(i, j) = \max\{\ell : T[i..i + \ell) = T[j..j + \ell)\}$



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

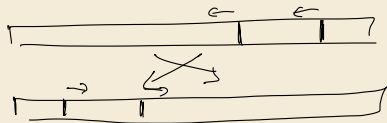
Tandem Repeats – Idea

- ▶ 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



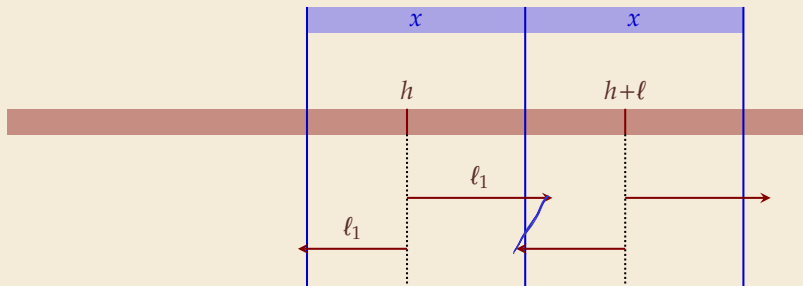
$$\ell_1 = \text{LCE}_T(h, h+\ell)$$

$$\ell_2 = \text{LCE}_T(h, h+\ell) = \text{LCE}_{TR}(n-h, n-h-\ell)$$

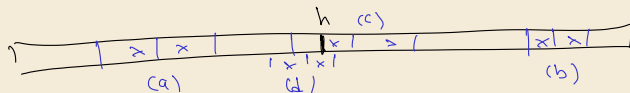


Tandem Repeats – Idea

- ▶ 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

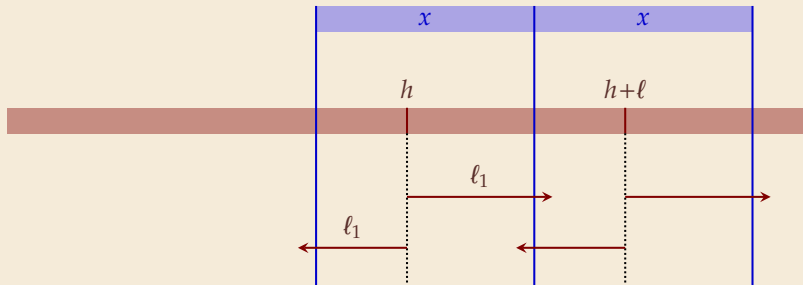


D & C!



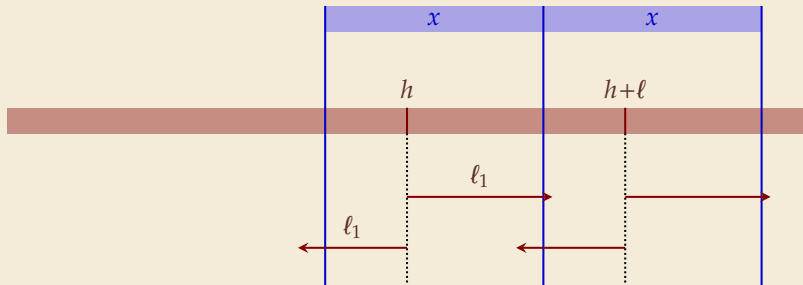
Tandem Repeats – Idea

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



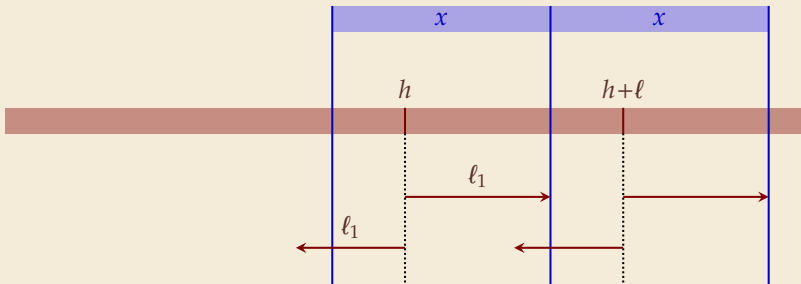
Tandem Repeats – Idea

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



Tandem Repeats – Idea

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



Tandom Repeats – Code

```
1 procedure tandemRepeats( $T[0..n]$ )
2   Build LCE data structure on  $T$  and  $T^R$ 
3   //  $T^R = \text{reversed text}$ 
4   // We abbreviate  $\vec{e}(i, j) = \text{LCE}_T(i, j)$ 
5   // resp.  $\tilde{e}(i, j) = \text{LCE}_{T^R}(n - j, n - i)$ 
6    $tr := \emptyset$ 
7   appendRepeats( $T[0..n]$ ,  $tr$ )
8   return  $tr$ 
```

```
1 procedure appendRepeats( $T[0..n]$ ,  $tr$ )
2   if  $n \leq 1$  return
3    $h := \lfloor n/2 \rfloor$ 
4   appendRepeats( $T[0..h]$ )
5   appendRepeats( $T[h..n]$ )
6   // Add all repeats  $xx$  with  $h$  inside first copy of  $x$ 
7   for  $\ell \in [1 .. n - h]$  // length of  $x$ 
8      $q := h + \ell$ 
9      $\ell_1 := \vec{e}(h, q)$ 
10     $\ell_2 := \tilde{e}(h - 1, q - 1)$ 
11    if  $\ell_1 + \ell_2 \geq \ell$ 
12       $tr := tr \cup \{(j, \ell) : j \in [h - \ell_2 .. h + \ell_1 - \ell]\}$ 
13    // Symmetrically for  $h$  inside second copy of  $x$ 
14    for  $\ell \in [1..h]$ 
15       $q := h - \ell$ 
16       $\ell_1 := \vec{e}(h, q)$ 
17       $\ell_2 := \tilde{e}(h - 1, q - 1)$ 
18      if  $\ell_1 + \ell_2 \geq \ell$ 
19         $tr := tr \cup \{(j, \ell) : j \in [q - \ell_2 .. q + \ell_1 - \ell]\}$ 
```

Tandom Repeats – Analysis

► Running time

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

$\rightsquigarrow O(n \log n + \text{output})$

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

Longest Common Extensions & Suffix Trees

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

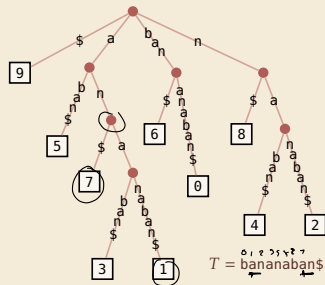
Recall *longest common extension (LCE)* data structure

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

↪ use suffix tree of T !



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

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



Efficient LCA

How to find lowest common ancestors?

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

Efficient LCA

How to find lowest common ancestors?

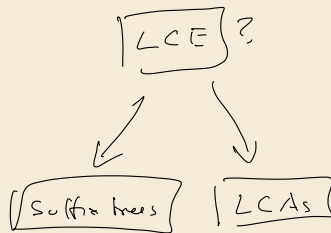
- ▶ Could walk up the tree to find LCA $\rightsquigarrow \Theta(n)$ worst case 🙄
- ▶ Could store all LCAs in big table $\rightsquigarrow \Theta(n^2)$ space and preprocessing 🙄



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

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

and suffix tree construction inside ...



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

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

Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention

👍 linear time and space

👍 suddenly many questions efficiently solvable in theory



Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention



linear time and space



suddenly many questions efficiently solvable in theory



construction of suffix trees:
linear time, but significant overhead



construction methods fairly complicated



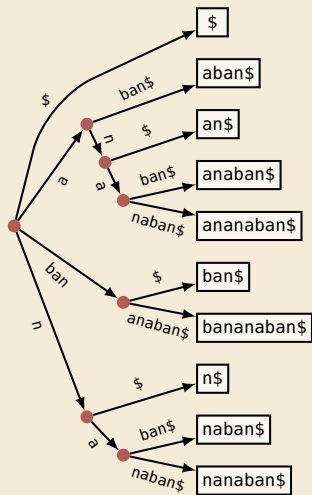
many pointers in tree incur large space overhead



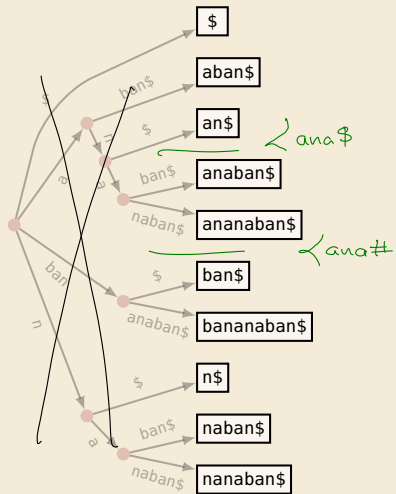
6.6 Suffix Arrays

Putting suffix trees on a diet

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



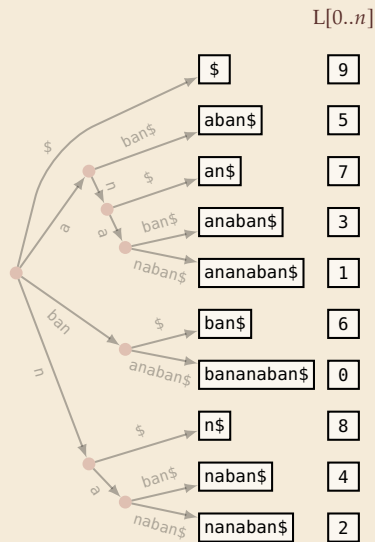
Putting suffix trees on a diet



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

$\text{ana\$}$
 ana\#

Putting suffix trees on a diet



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

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

► Sufficient to do efficient string matching!

1. Use binary search for pattern P
2. check if P is prefix of suffix after position found

► **Example:** $P = \text{ana}$

↪ $L[0..n]$ is called *suffix array*:

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


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

Suffix arrays – Construction


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

- ▶ from suffix tree


- ▶ possible with traversal . . .

-  but we are trying to avoid constructing suffix trees!

- ▶ sorting the suffixes of T using general purpose sorting method

-  trivial to code!

- ▶ but: comparing two suffixes can take $\Theta(n)$ character comparisons


-  $\Theta(n^2 \log n)$ time in worst case

Suffix arrays – Construction


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

- ▶ from suffix tree


- ▶ possible with traversal ...

-  but we are trying to avoid constructing suffix trees!

- ▶ sorting the suffixes of T using general purpose sorting method

-  trivial to code!

- ▶ but: comparing two suffixes can take $\Theta(n)$ character comparisons

-  $\Theta(n^2 \log n)$ time in worst case

- ▶ We can do better!

Excursion: String sorting

- ▶ when sorting strings, “blind” ^{string} comparisons can cost $\Theta(n)$ character comparisons
- ▶ happens iff strings share long prefix!
- ↪ dedicated string sorting methods need to remember common prefixes between strings
then we can avoid redoing these comparisons

Excursion: String sorting

- ▶ when sorting strings, “blind” comparisons can cost $\Theta(n)$ character comparisons
- ▶ happens iff strings share long prefix!
- ↪ dedicated string sorting methods need to remember common prefixes between strings then we can avoid redoing these comparisons

(length of) longest common prefix

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

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

Fat-pivot radix quicksort – Example

she

sells

seashells

by

the

sea

shore

the

shells

she

sells

are

surely

seashells

Fat-pivot radix quicksort – Example

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

Fat-pivot radix quicksort – Example

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

Fat-pivot radix quicksort – Example

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

Fat-pivot radix quicksort – Example

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

Fat-pivot radix quicksort – Example

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

Fat-pivot radix quicksort – Example



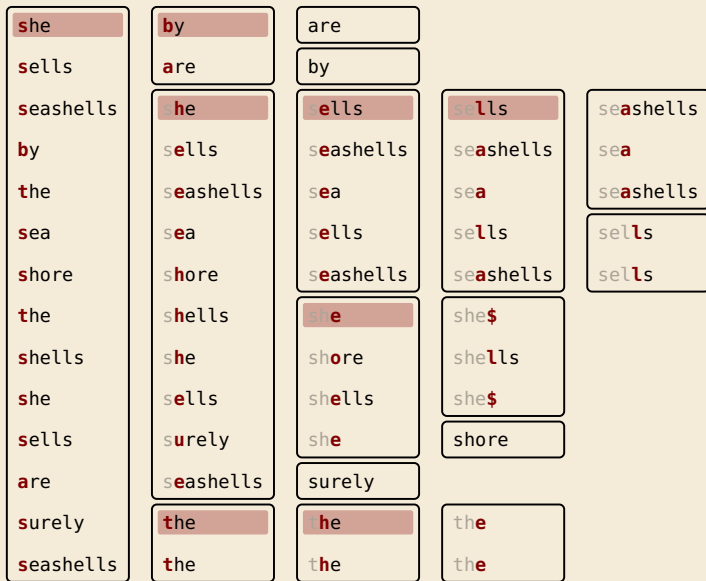
Fat-pivot radix quicksort – Example

she	by	are	
sells	are	by	
seashells	he	sells	sells
by	sells	seashells	seashells
the	seashells	sea	sea
sea	sea	sells	sells
shore	shore	seashells	seashells
the	shells	she	she\$
shells	she	shore	shells
she	sells	shells	she\$
sells	surely	she	shore
are	seashells	surely	
surely	the	the	
seashells	the	the	

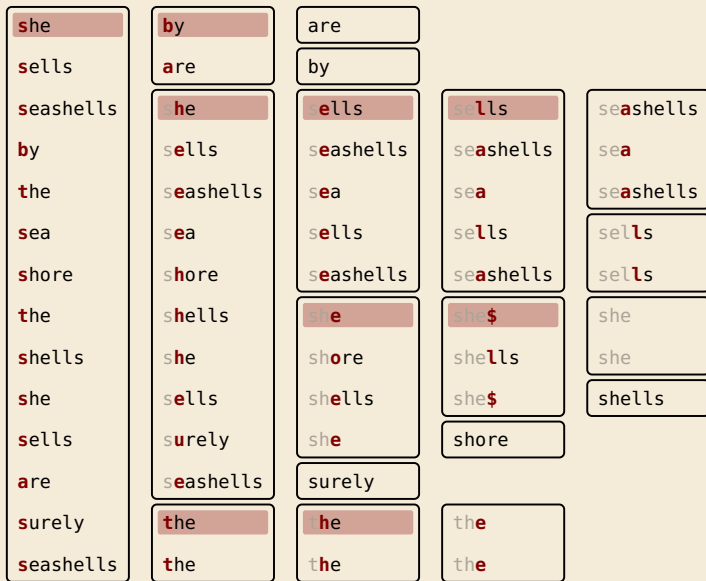
Fat-pivot radix quicksort – Example

she	by	are	
sells	are	by	
seashells	he	sells	sells
by	sells	seashells	seashells
the	seashells	sea	sea
sea	sea	sells	sells
shore	shore	seashells	seashells
the	shells	she	she\$
shells	she	shore	shells
she	sells	shells	she\$
sells	surely	she	shore
are	seashells	surely	
surely	the	the	the
seashells	the	the	the

Fat-pivot radix quicksort – Example



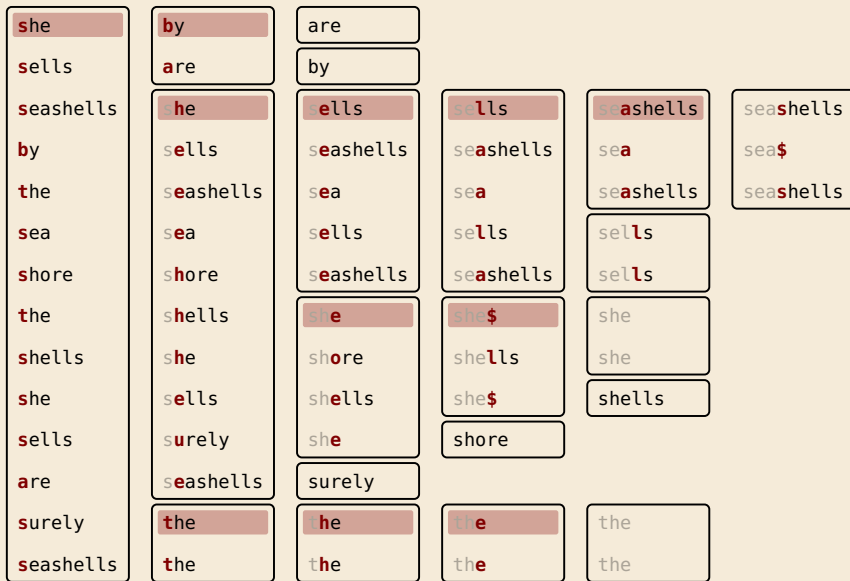
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example

she	by	are		
sells	are	by		
seashells	he	sells	sells	seashells
by	sells	seashells	seashells	sea
the	seashells	sea	sea	seashells
sea	sea	sells	sells	sells
shore	shore	seashells	seashells	sells
the	shells	she	she\$	she
shells	she	shore	shells	she
she	sells	shells	she\$	shells
sells	surely	she	shore	
are	seashells	surely		
surely	the	the	the	the
seashells	the	the	the	the

Fat-pivot radix quicksort – Example



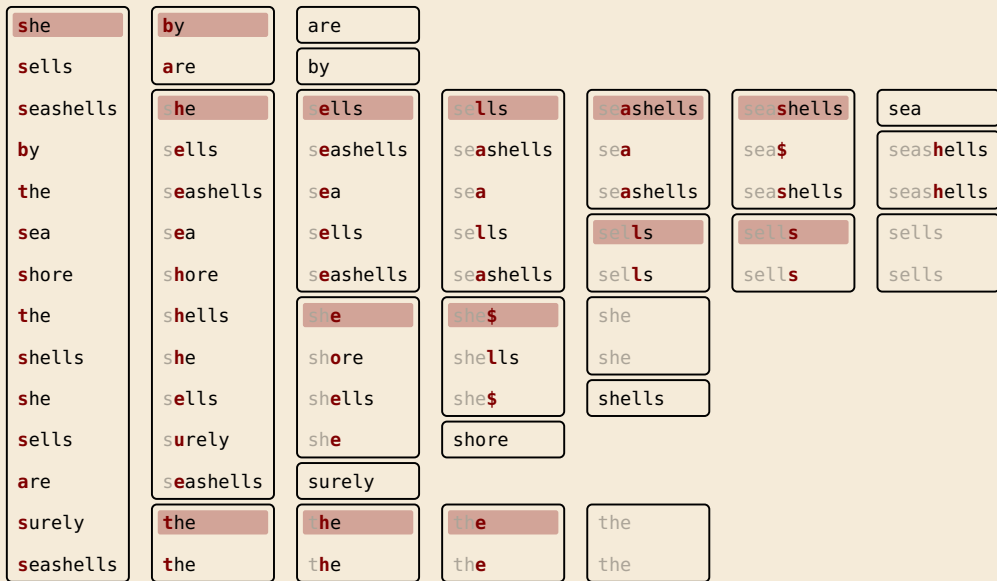
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example



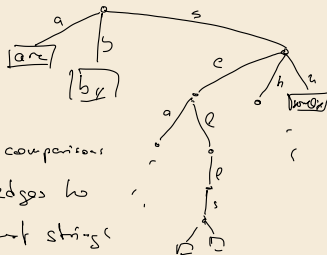
Fat-pivot radix quicksort – Example



Fat-pivot radix quicksort – Example

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

LCP comparisons
= # edges to insert strings



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 $\sim 2 \ln(2) \cdot n \lg n \approx 1.39n \lg n$ decisive comparisons
 - ▶ duplicates only reduce # comparisons

Fat-pivot radix quicksort – Analysis

Separately analyze character comparisons **by outcome**

1. *“Decisive Comparisons:”* character comparisons with outcome “<” or “>”

- ▶ can have at most one in any **string comparison** (afterwards done!)
- ↪ Same number of decisive comparisons as in standard quicksort (just delayed)
- ↪ expected $\sim 2 \ln(2) \cdot n \lg n \approx 1.39n \lg n$ 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
- ↪ #LCP comparisons = #comparisons when inserting all strings into a **trie**

w/ unused paths
↳ leaves collapsed

Fat-pivot radix quicksort – Discussion

👍 simple to code

👍 efficient for sorting many lists of strings

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

random string



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

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

Fat-pivot radix quicksort – Discussion

👍 simple to code

👍 efficient for sorting many lists of strings

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

random string



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

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*:

- $R[i] = r \iff L[r] = i$ $L = \text{leaf array}$
 - \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		r	$L[r]$	$T_{L[r]}$
0	6 th	bananaban\$	<div>right $R[0] = 6$</div>	0	9	\$
1	4 th	ananaban\$		1	5	aban\$
2	9 th	nanaban\$		2	7	an\$
3	3 th	anaban\$		3	3	anaban\$
4	8 th	naban\$		4	1	ananaban\$
5	1 th	aban\$		5	6	ban\$
6	5 th	ban\$		6	0	bananaban\$
7	2 th	an\$		7	8	n\$
8	7 th	n\$	<div>left $L[8] = 4$</div>	8	4	naban\$
9	0 th	\$		9	2	nanaban\$

sort suffixes

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 $T_0, T_3, T_6, T_9, \dots$ from $R_{1,2}$.
3. Merge $R_{1,2}$ and R_0 using $R_{1,2}$.
 \rightsquigarrow rank array R for entire input

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 $T_0, T_3, T_6, T_9, \dots$ 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

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

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 $T_0, T_3, T_6, T_9, \dots$ 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 $n + \frac{2}{3}n + \left(\frac{2}{3}\right)^2 n + \left(\frac{2}{3}\right)^3 n + \dots \leq n \cdot \sum_{i \geq 0} \left(\frac{2}{3}\right)^i = 3n = \Theta(n)$

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

\rightsquigarrow Can use whichever is more convenient.

DC3 / Skew algorithm – Step 2: Inducing ranks

- ▶ **Assume:** rank array $R_{1,2}$ known:

$$\text{▶ } R_{1,2}[i] = \begin{cases} \text{rank of } T_i \text{ among } T_1, T_2, T_4, T_5, T_7, T_8, \dots & \text{for } i = 1, 2, 4, 5, 7, 8, \dots \\ \text{undefined} & \text{for } i = 0, 3, 6, 9, \dots \end{cases}$$

- ▶ **Task:** sort the suffixes $T_0, T_3, T_6, T_9, \dots$ in linear time (!)

DC3 / Skew algorithm – Step 2: Inducing ranks

- ▶ **Assume:** rank array $R_{1,2}$ known:

$$\text{▶ } R_{1,2}[i] = \begin{cases} \text{rank of } T_i \text{ among } T_1, T_2, T_4, T_5, T_7, T_8, \dots & \text{for } i = 1, 2, 4, 5, 7, 8, \dots \\ \text{undefined} & \text{for } i = 0, 3, 6, 9, \dots \end{cases}$$

- ▶ **Task:** sort the suffixes $T_0, T_3, T_6, T_9, \dots$ 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

$T = \text{hannahbansbananasman}\$ \$ \$$

(append 3 \$ markers)

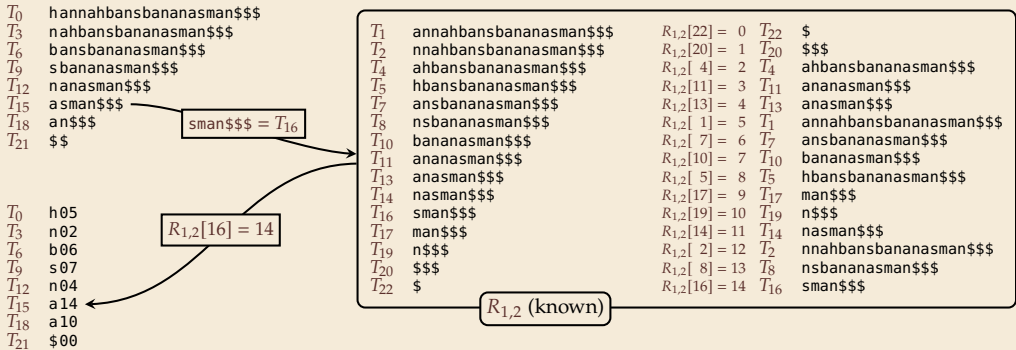
T_0 hannahbansbananasman\$\$\$
 T_3 nahbansbananasman\$\$\$
 T_6 bansbananasman\$\$\$
 T_9 sbanasman\$\$\$
 T_{12} nanasman\$\$\$
 T_{15} asman\$\$\$
 T_{18} an\$\$\$
 T_{21} \$\$

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} asman\$\$\$
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)

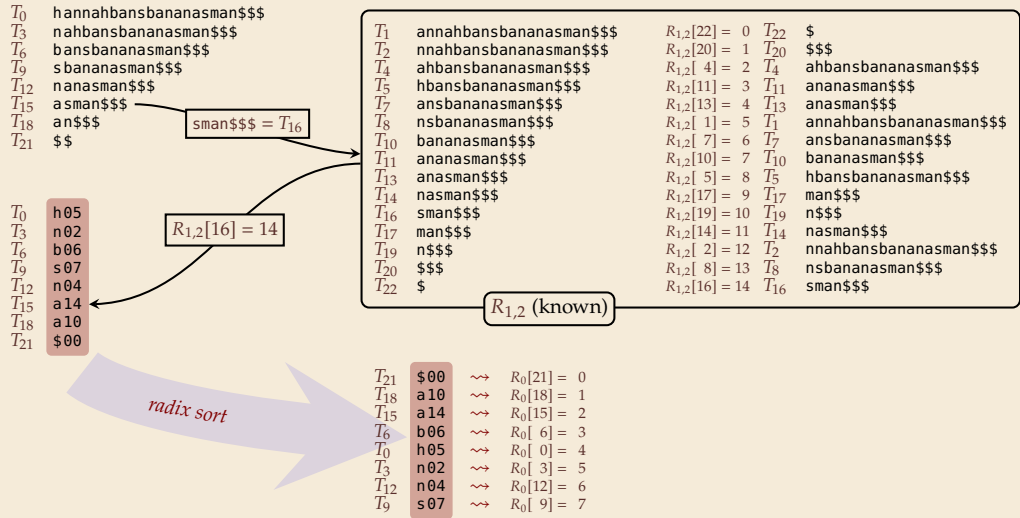
DC3 / Skew algorithm – Inducing ranks example

$T = \text{hannahbansbananasman}\$ \$ \$$ (append 3 \$ markers)



DC3 / Skew algorithm – Inducing ranks example

$T = \text{hannahbansbananasman}\$ \$ \$$ (append 3 \$ markers)



DC3 / Skew algorithm – Inducing ranks example

$T = \text{hannahbansbananasman}\$ \$ \$$

(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} \$\$

smans\$\$ = T_{16}

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

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_1 annahbansbananasman\$\$\$
 T_2 nnahbansbananasman\$\$\$
 T_4 ahbansbananasman\$\$\$
 T_5 hbansbananasman\$\$\$
 T_7 ansbananasman\$\$\$
 T_8 nsbananasman\$\$\$
 T_{10} bananasman\$\$\$
 T_{11} ananasman\$\$\$
 T_{13} anasman\$\$\$
 T_{14} nasman\$\$\$
 T_{16} sman\$\$\$
 T_{17} man\$\$\$
 T_{19} n\$\$\$
 T_{20} \$\$\$
 T_{22} \$

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

$R_{1,2}$ (known)

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

DC3 / Skew algorithm – Inducing ranks example

$T = \text{hannahbansbananasman}\$ \$ \$$

(append 3 \$ markers)

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

sman\$\$\$ = T_{16}

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

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_1 annahbansbananasman\$\$\$
 T_2 nnahbansbananasman\$\$\$
 T_4 ahbansbananasman\$\$\$
 T_5 hbansbananasman\$\$\$
 T_7 ansbananasman\$\$\$
 T_8 nsbananasman\$\$\$
 T_{10} bananasman\$\$\$
 T_{11} ananasman\$\$\$
 T_{13} anasman\$\$\$
 T_{14} nasman\$\$\$
 T_{16} sman\$\$\$
 T_{17} man\$\$\$
 T_{19} n\$\$\$
 T_{20} \$\$\$
 T_{22} \$

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

$R_{1,2}$ (known)

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

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

\rightsquigarrow 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\$\$\$

► Have:

- sorted 1,2-list:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

- sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

- use standard merging method from Mergesort
- but speed up comparisons using $R_{1,2}$

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}	anasman\$\$\$
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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

► sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

► use standard merging method from Mergesort

► but speed up comparisons using $R_{1,2}$

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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

► sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

- use standard merging method from Mergesort
- but speed up comparisons using $R_{1,2}$

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} = \text{asman}$$$
 $= \text{asman}$$$
 $= aT_{16}$
 $T_{11} = \text{ananasman}$$$
 $= \text{ananasman}$$$
 $= aT_{12}$$$$$

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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

- sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

- use standard merging method from Mergesort
- but speed up comparisons using $R_{1,2}$

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} = \text{asman}$$$
 $= \text{asman}$$$$ can't compare T_{16}
 $= aT_{16}$ and T_{12} either!
 $T_{11} = \text{ananasman}$$$
 $= \text{ananasman}$$$$
 $= aT_{12}$$$

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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

- sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

- use standard merging method from Mergesort
- but speed up comparisons using $R_{1,2}$

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} = \text{asman}$$$
 $= \text{asman}$$$$ can't compare T_{16}
 $= aT_{16}$ and T_{12} either!
 $T_{11} = \text{ananasman}$$$
 $= \text{ananasman}$$$$
 $= aT_{12}$$$

↪ Compare T_{16} to T_{12}

$T_{16} = \text{sman}$$$$
 $= \text{sman}$$$$
 $= sT_{17}$
 $T_{12} = \text{nanasman}$$$$
 $= \text{aananasman}$$$$
 $= aT_{13}$

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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

- sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► Task: Merge them!

- use standard merging method from Mergesort
- but speed up comparisons using $R_{1,2}$

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} = \text{asman}$$$
 $= \text{asman}$$$ can't compare T_{16}
 $= aT_{16}$ and T_{12} either!
 $T_{11} = \text{ananasman}$$$
 $= \text{ananasman}$$$
 $= aT_{12}$$$$$

↪ Compare T_{16} to T_{12}

$T_{16} = \text{sman}$$$
 $= \text{sman}$$$ always at most 2 steps
 $= sT_{17}$ then can use $R_{1,2}$!
 $T_{12} = \text{nanasman}$$$
 $= \text{aananasman}$$$
 $= aT_{13}$$$$$

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:

$T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \dots$

► sorted 0-list:

$T_0, T_3, T_6, T_9, \dots$

► 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} = \text{asman}$$$
 $= \text{asman}$$$ can't compare T_{16}
 $= aT_{16}$ and T_{12} either!
 $T_{11} = \text{ananasman}$$$
 $= \text{ananasman}$$$
 $= aT_{12}$$$$$

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

$T_{16} = \text{sman}$$$
 $= \text{sman}$$$ always at most 2 steps
 $= sT_{17}$ then can use $R_{1,2}$!
 $T_{12} = \text{nanasman}$$$
 $= \text{nanasman}$$$
 $= aT_{13}$$$$$

6.8 Suffix Sorting: The DC3 Algorithm

DC3 / Skew algorithm – Fix Step 1: Recursive call

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

DC3 / Skew algorithm – Fix Step 1: 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!



DC3 / Skew algorithm – Fix Step 1: 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?

DC3 / Skew algorithm – Fix Step 1: 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{abc}

↪ suffixes of $T^\square \iff T_0, T_3, T_6, T_9, \dots$

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

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

$T = \text{bananaban} \$ \$ \$$
↪ $T^\square = \boxed{\text{ban}} \boxed{\text{ana}} \boxed{\text{ban}} \boxed{\$ \$ \$}$
 $\boxed{\text{ana}} \boxed{\text{ban}} \boxed{\$ \$ \$}$
 $\boxed{\text{ban}} \boxed{\$ \$ \$}$
 $\boxed{\$ \$ \$}$

DC3 / Skew algorithm – Fix alphabet explosion

- ▶ Still does not quite work!

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!

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{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 $\sigma \leq n$ without affecting order of suffixes.

DC3 / Skew algorithm – Example Step 1: Recursive call

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

► $T = \text{hannahbansbananasman\$}$

DC3 / Skew algorithm – Example Step 1: Recursive call

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

► $T = \text{hannahbansbananasman\$}$ $T_2 = \text{nna hba nsb ana nas man \$}$

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

DC3 / Skew algorithm – Example Step 1: Recursive call

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

► $T = \text{hannahbansbananasman\$}$ $T_2 = \text{nnaahbansbananasman\$}$

$T' =$ annahbansbananasman\$ \$ \$ \$ nnahbansbananasman \$ \$ \$

► Occurring triples:

annahbansbananasman\$ \$ \$ \$ nnahbansb nasman

DC3 / Skew algorithm – Example Step 1: Recursive call

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

► $T = \text{hannahbansbananasman\$}$ $T_2 = \text{nnahbansbananasman\$}$

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

► Occurring triples:

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

► Sorted triples with ranks:

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

DC3 / Skew algorithm – Example Step 1: Recursive call

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

► $T = \text{hannahbansbananasman\$}$ $T_2 = \text{nnahbansbananasman\$}$

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

► Occurring triples:

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

► Sorted triples with ranks:

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

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

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

DC3 – Overall algorithm

Recall:

DC3 / Skew algorithm

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 $T_0, T_3, T_6, T_9, \dots$ from $R_{1,2}$.
3. Merge $R_{1,2}$ and R_0 using $R_{1,2}$.
Return R , rank array for entire input

Implementation notes:

- Robust and efficient implementations are available as libraries

↪ Here focus on simplicity:

Key elements of clever implementation from original inventors of DC3 algorithm



Kärkkäinen, Sanders, Burkhardt: *Linear work suffix array construction*, JACM 2006

Radix Pass with Indirect Addressing

Key building block of Kärkäinen et al:

one round of **radix sort** with *indirect addressing* and *indirect keys*

```
1 procedure radixPass( $A[0..n)$ ,  $V[0..m)$ ,  $k$ )
2   // Return indices in  $A$  sorted by  $V[A[i]]$  as keys
3   // Assumes all  $A[i] \in [0..m)$  and  $V[j] \in [0..k)$ 
4   // Step 1: Count occurrences of values  $[0..k)$ 
5    $C[0..k) := 0$ 
6   for  $i \in [0..n)$  do  $C[V[A[i]]] := C[V[A[i]]] + 1$  end for
7   // Step 2: Compute Prefix sums  $P[i] = \sum C[0..i)$ , (starting indices for blocks)
8    $P[0..k] := 0$ 
9   for  $j := 1, \dots, k$  do  $P[j] := P[j - 1] + C[j - 1]$  end for
10  // Step 3: Sort
11   $B[0..n) := 0$  // output
12  for  $i := 0, \dots, n - 1$ 
13     $v := V[A[i]]$ 
14     $B[C[v]] := A[i]$ 
15     $C[v] := C[v] + 1$ 
16  return  $B[0..n)$ 
```

- ↪ can sort non-contiguous subarrays without copying
- ↪ can pass as $V[0..n)$ a shifted version of $T[1..n + 1)$ to compare next characters

DC3 – Final algorithm

```
1 procedure DC3( $T[0..n], \sigma$ )
2   // Returns rank array (inverse suffix array) for  $T[0..n]$ 
3   // Assumes  $T[0..n] \in [1..\sigma]^*$ ,  $T[n] = T[n+1] = T[n+2] = \$ = 0$ 
4    $n_0 := \lfloor \frac{n+2}{3} \rfloor$ ;  $n_1 := \lfloor \frac{n+1}{3} \rfloor$ ;  $n_2 := \lfloor \frac{n}{3} \rfloor$ ;  $n_{02} := n_0 + n_2$ 
5    $R_{12} := [i : i \in [0..n+(n_0-n_1)) \wedge i \bmod 3 \neq 0]$  // indices for  $R_{12}$ 
6   // Step 1: Recursively compute  $R_{12}$ ; Step 1a: Sort triples in  $T'$ 
7    $tmp[0..n_{02}] := \text{radixPass}(R_{12}[0..n_{02}], T[2..n+2], \sigma)$ 
8    $R_{12}[0..n_{02}] := \text{radixPass}(tmp[0..n_{02}], T[1..n+1], \sigma)$ 
9    $tmp[0..n_{02}] := \text{radixPass}(R_{12}[0..n_{02}], T[0..n+0], \sigma)$ 
10  // Step 1b: Rank-reduce alphabet, write into  $R_{12}$ 
11   $c := 0$ ;  $prev := \text{NULL}$ 
12  for  $j := 0, \dots, n_{02} - 1$ 
13     $i := tmp[j]$ 
14    if  $T[i..i+3] \neq prev$  // new triple
15       $c := c + 1$ ;  $prev := T[i..i+3]$ 
16       $R_{12}[\lfloor j/3 \rfloor + [j \bmod 3 == 2] \cdot n_0] := c$  // Write  $c$  to first/second half of  $R_{12}$  based on  $j \bmod 3$ 
17  // Step 1c: Recurse iff names not yet unique (otherwise: triple names = ranks!)
18  if  $c < n_{02}$  then  $R_{12} := \text{DC3}(R_{12}[0..n_{02}], c)$ 
19  // ... merging and induced sorting as before
20  return  $R$ 
```

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

Clicker Question

Which feature of suffix **trees** did we use to find the *length* of a longest repeated substring?



- ☐ **A** order of leaves
- ☐ **B** path label of internal nodes
- ☐ **C** string depth of internal nodes
- ☐ **D** constant-time traversal to child nodes
- ☐ **E** constant-time traversal to parent nodes
- ☐ **F** constant-time traversal to leftmost leaf in subtree



→ *sli.do/cs594*

Clicker Question



Which feature of suffix **trees** did we use to find the *length* of a longest repeated substring?

- ☐ **A** ~~order of leaves~~
- ☐ **B** ~~path label of internal nodes~~
- ☒ **C** string depth of internal nodes ✓
- ☐ **D** ~~constant time traversal to child nodes~~
- ☐ **E** ~~constant time traversal to parent nodes~~
- ☐ **F** ~~constant time traversal to leftmost leaf in subtree~~



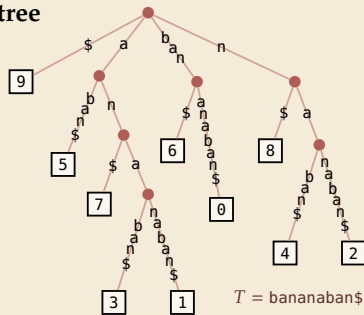
→ *sli.do/cs594*

String depths of internal nodes

- 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**?



String depths of internal nodes

- 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**?

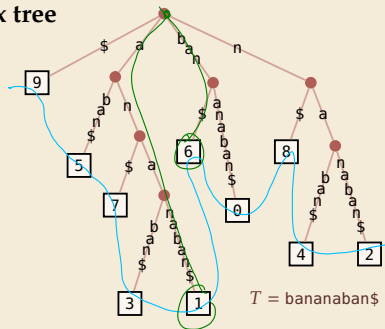
- Yes, by **enhancing** the suffix array with the **LCP array**!

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

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

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

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



LCP array and internal nodes

$L[0..n]$

9

5

7

3

1

6

0

8

4

2

LCP array and internal nodes

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

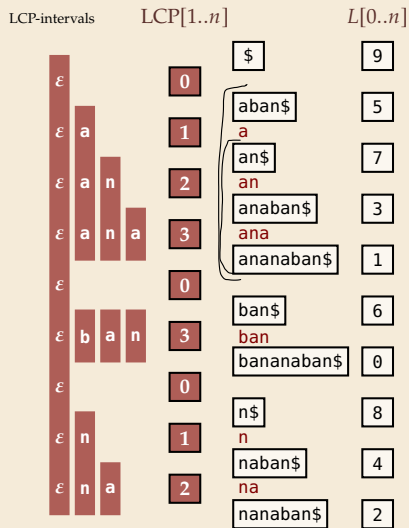
LCP array and internal nodes

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

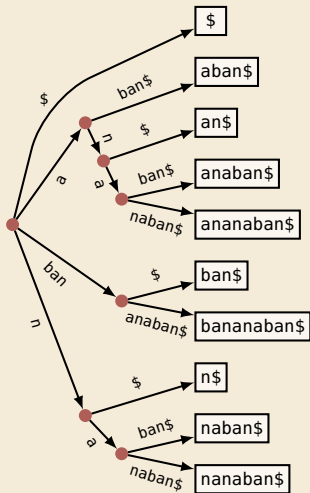
LCP array and internal nodes

	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

LCP array and internal nodes



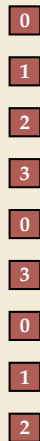
LCP array and internal nodes



LCP-intervals



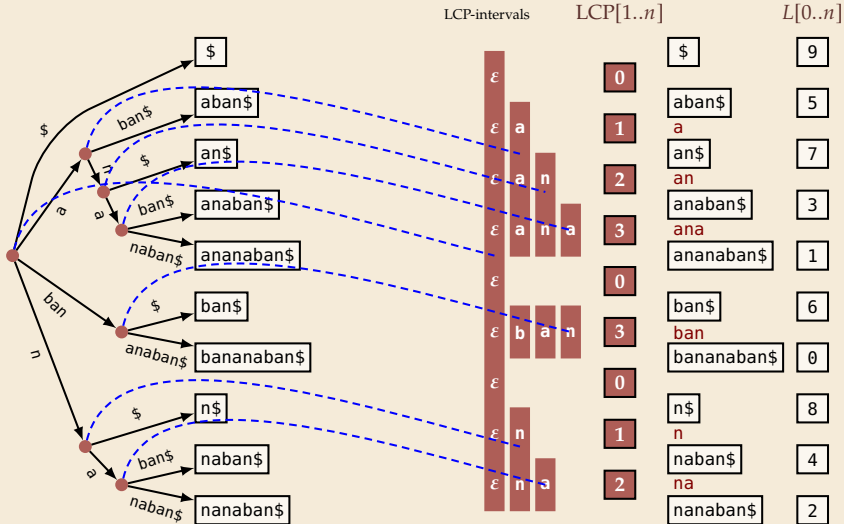
LCP[1..n]



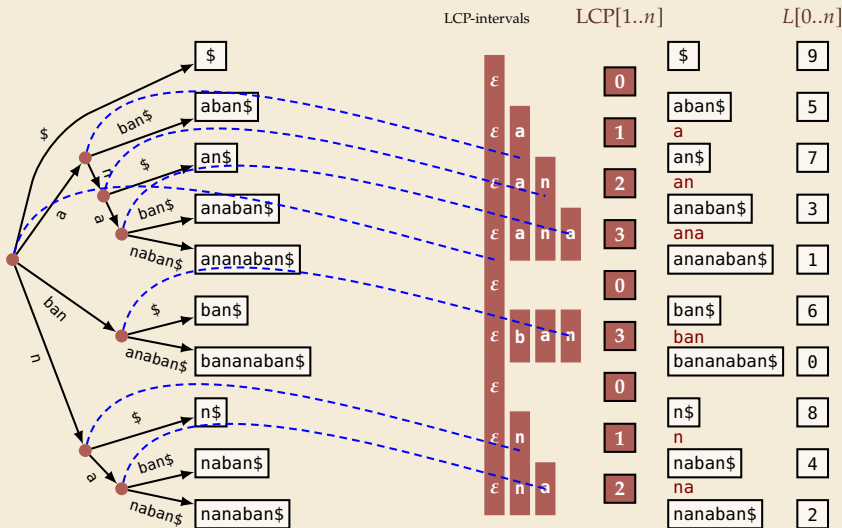
L[0..n]

\$	9
aban\$	5
a	7
an	3
anaban\$	1
anaban\$	6
an	8
n	4
naban\$	2
nanaban\$	

LCP array and internal nodes



LCP array and internal nodes



↪ Leaf array $L[0..n]$ plus LCP array $LCP[1..n]$ encode full tree!

Back to suffix trees



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

Back to suffix trees



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

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

Back to suffix trees



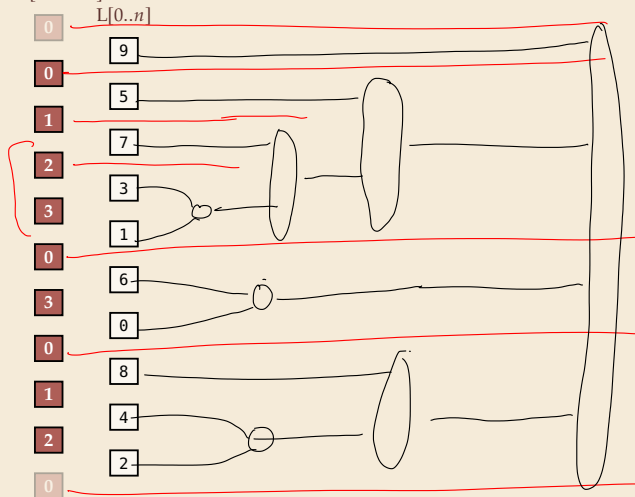
We can finally look into the black box of linear-time suffix-array construction!

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

Simple recursive method:

- Find min LCP values
- recurse between them
(add fictitious min at boundaries)
- attach resulting subtrees
to a new root

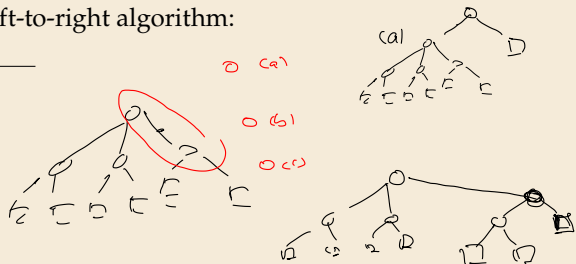
LCP[0 .. n+1]



Linear-time tree reconstruction

Recursive method is equivalent to the following left-to-right algorithm:

```
1 procedure suffixTreeFrom(LCP[1..n], L[0..n])
2    $v_0 := \text{new Node}(\text{stringDepth: } -1)$  // dummy root / sentinel
3    $v_0.\text{addChild}(\text{Leaf}(L[0]))$ 
4    $S := \text{new Stack}$ 
5    $S.\text{push}(v_0)$  //  $S$  always contains right spine of  $\mathcal{T}$ 
6   for  $j := 1, \dots, n + 1$ 
7      $d := \text{LCP}[j]$ 
8     // Insert new node into right spine
9     while  $S.\text{top}().\text{stringDepth} > d$  do  $S.\text{pop}()$  end while
10     $p := S.\text{top}()$ 
11     $u := \text{new Node}(\text{stringDepth: } d)$ 
12     $u.\text{addChild}(p.\text{removeRightmostChild}())$ 
13     $u.\text{addChild}(\text{Leaf}(L[j]))$ 
14     $p.\text{addChild}(u)$ 
15     $S.\text{push}(u)$ 
16  return  $v_0.\text{rightmostChild}()$ 
```



Linear-time tree reconstruction

Recursive method is equivalent to the following left-to-right algorithm:

```
1 procedure suffixTreeFrom(LCP[1..n], L[0..n])
2    $v_0 := \text{new Node}(\text{stringDepth: } -1)$  // dummy root / sentinel
3    $v_0.\text{addChild}(\text{Leaf}(L[0]))$ 
4    $S := \text{new Stack}$ 
5    $S.\text{push}(v_0)$  //  $S$  always contains right spine of  $\mathcal{T}$ 
6   for  $j := 1, \dots, n + 1$ 
7      $d := \text{LCP}[j]$ 
8     // Insert new node into right spine
9     while  $S.\text{top}().\text{stringDepth} > d$  do  $S.\text{pop}()$  end while
10     $p := S.\text{top}()$ 
11     $u := \text{new Node}(\text{stringDepth: } d)$ 
12     $u.\text{addChild}(p.\text{removeRightmostChild}())$ 
13     $u.\text{addChild}(\text{Leaf}(L[j]))$ 
14     $p.\text{addChild}(u)$ 
15     $S.\text{push}(u)$ 
16  return  $v_0.\text{rightmostChild}()$ 
```

- ▶ $O(n)$ time overall
(every node pushed and popped once)
- ▶ algorithm computes shape only
 \rightsquigarrow second traversal for labels

Linear-time tree reconstruction

Recursive method is equivalent to the following left-to-right algorithm:

```
1 procedure suffixTreeFrom(LCP[1..n], L[0..n])
2    $v_0 := \text{new Node}(\text{stringDepth: } -1)$  // dummy root / sentinel
3    $v_0.\text{addChild}(\text{Leaf}(L[0]))$ 
4    $S := \text{new Stack}$ 
5    $S.\text{push}(v_0)$  //  $S$  always contains right spine of  $\mathcal{T}$ 
6   for  $j := 1, \dots, n + 1$ 
7      $d := \text{LCP}[j]$ 
8     // Insert new node into right spine
9     while  $S.\text{top}().\text{stringDepth} > d$  do  $S.\text{pop}()$  end while
10     $p := S.\text{top}()$ 
11     $u := \text{new Node}(\text{stringDepth: } d)$ 
12     $u.\text{addChild}(p.\text{removeRightmostChild}())$ 
13     $u.\text{addChild}(\text{Leaf}(L[j]))$ 
14     $p.\text{addChild}(u)$ 
15     $S.\text{push}(u)$ 
16  return  $v_0.\text{rightmostChild}()$ 
```

- ▶ $O(n)$ time overall
(every node pushed and popped once)
- ▶ algorithm computes shape only
 \rightsquigarrow second traversal for labels
- ▶ similar to known algorithms
 - ▶ Cartesian trees
 - ▶ Powersort

6.10 LCP Array Construction

LCP array construction

- ▶ computing $\text{LCP}[1..n]$ naively too expensive
 - ▶ each value could take $\Theta(n)$ time
- 👎 $\Theta(n^2)$ in total


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 \rightsquigarrow can find another large one!
- ▶ Example: $T = \text{Buffalo_buffalo_buffalo_buffalo\$}$

LCP array construction

- ▶ computing $\text{LCP}[1..n]$ naively too expensive

- ▶ each value could take $\Theta(n)$ time

-  $\Theta(n^2)$ in total

- ▶ but: seeing one large (= costly) LCP value \rightsquigarrow can find another large one!

- ▶ Example: $T = \text{Buffalo_buffalo_buffalo_buffalo\$}$

- ▶ first few suffixes in sorted order:

$T_L[0] = \$$

$T_L[1] = \text{alo_buffalo\$}$

$T_L[2] = \text{alo_buffalo_buffalo\$}$

alo_buffalo_buffalo $\rightsquigarrow \text{LCP}[3] = 19$

$T_L[3] = \text{alo_buffalo_buffalo_buffalo\$}$

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 \rightsquigarrow can find another large one!

- ▶ Example: $T = \text{Buffalo_buffalo_buffalo_buffalo\$}$

- ▶ first few suffixes in sorted order:

$T_L[0] = \$$

$T_L[1] = \text{alo_buffalo\$}$

$T_L[2] = \text{alo_buffalo_buffalo\$}$

alo_buffalo_buffalo $\rightsquigarrow LCP[3] = 19$

$T_L[3] = \text{alo_buffalo_buffalo_buffalo\$}$

- \rightsquigarrow **Removing first character** from $T_L[2]$ and $T_L[3]$ gives two new suffixes:

$T_L[?] = \text{lo_buffalo_buffalo\$}$

lo_buffalo_buffalo $\rightsquigarrow LCP[?] = 18$

$T_L[?] = \text{lo_buffalo_buffalo_buffalo\$}$
 \uparrow
unclear where...

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 \rightsquigarrow can find another large one!

- ▶ Example: $T = \text{Buffalo_buffalo_buffalo_buffalo\$}$

- ▶ first few suffixes in sorted order:

$T_{L[0]} = \$$

$T_{L[1]} = \text{alo_buffalo\$}$

$T_{L[2]} = \text{alo_buffalo_buffalo\$}$

alo_buffalo_buffalo $\rightsquigarrow LCP[3] = 19$

$T_{L[3]} = \text{alo_buffalo_buffalo_buffalo\$}$

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

$T_{L[?]} = \text{lo_buffalo_buffalo\$}$

lo_buffalo_buffalo $\rightsquigarrow LCP[?] = 18$

$T_{L[?]} = \text{lo_buffalo_buffalo_buffalo\$}$

↑
unclear where...



Shortened suffixes might *not* be *adjacent* in sorted order!

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

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

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

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

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

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

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
→ 0	6 th	bananaban\$	0	9	\$	–
1	4 th	ananaban\$	1	5	aban\$	
2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	anaban\$	
4	8 th	naban\$	4	1	ananaban\$	
5	1 th	aban\$	5	6	<u>ban</u> \$	
6	5 th	ban\$	→ 6	0	<u>ban</u> anaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	naban\$	
9	0 th	\$	9	2	nanaban\$	

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

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
0	6 th	bananaban\$	0	9	\$	–
→ 1	4 th	ananaban\$	1	5	aban\$	
2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	anaban\$	
4	8 th	naban\$	4	1	ananaban\$	
5	1 th	aban\$	5	6	<u>b</u> an\$	
6	5 th	ban\$	6	0	<u>b</u> anaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	naban\$	
9	0 th	\$	9	2	nanaban\$	

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

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

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

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

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

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
0	6 th	bananaban\$	0	9	\$	–
1	4 th	ananaban\$	1	5	aban\$	
→ 2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	a <u>n</u> aban\$	
4	8 th	naban\$	4	1	a <u>n</u> anaban\$	3
5	1 th	aban\$	5	6	ban\$	
6	5 th	ban\$	6	0	bananaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	naban\$	
9	0 th	\$	9	2	nanaban\$	

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
0	6 th	bananaban\$	0	9	\$	–
1	4 th	ananaban\$	1	5	aban\$	
→ 2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	a <u>n</u> aban\$	
4	8 th	naban\$	4	1	a <u>n</u> anaban\$	3
5	1 th	aban\$	5	6	ban\$	
6	5 th	ban\$	6	0	bananaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	naban\$	
9	0 th	\$	→ 9	2	nanaban\$	

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
0	6 th	bananaban\$	0	9	\$	–
1	4 th	ananaban\$	1	5	aban\$	
→ 2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	a <u>n</u> aban\$	
4	8 th	naban\$	4	1	a <u>n</u> anaban\$	3
5	1 th	aban\$	5	6	ban\$	
6	5 th	ban\$	6	0	bananaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	<u>n</u> aban\$	
9	0 th	\$	→ 9	2	<u>n</u> anaban\$	

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	r	$L[r]$	$T_{L[r]}$	$LCP[r]$
0	6 th	bananaban\$	0	9	\$	–
1	4 th	ananaban\$	1	5	aban\$	
→ 2	9 th	nanaban\$	2	7	an\$	
3	3 th	anaban\$	3	3	a <u>n</u> aban\$	
4	8 th	naban\$	4	1	a <u>n</u> anaban\$	3
5	1 th	aban\$	5	6	ban\$	
6	5 th	ban\$	6	0	bananaban\$	3
7	2 th	an\$	7	8	n\$	
8	7 th	n\$	8	4	<u>n</u> aban\$	
9	0 th	\$	→ 9	2	<u>n</u> anaban\$	2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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



Kasai's algorithm – Code

```
1 procedure computeLCP( $T[0..n]$ ,  $L[0..n]$ ,  $R[0..n]$ ):  
2   // Assume  $T[n] = \$$ ,  $L$  and  $R$  are suffix array and inverse  
3    $\ell := 0$   
4   for  $i := 0, \dots, n - 1$  // Consider  $T_i$  now  
5      $r := R[i]$   
6     // compute LCP[ $r$ ]; note that  $r > 0$  since  $R[n] = 0$   
7      $i_{-1} := L[r - 1]$   
8     while  $T[i + \ell] == T[i_{-1} + \ell]$  do  
9        $\ell := \ell + 1$   
10    LCP[ $r$ ] :=  $\ell$   
11     $\ell := \max\{\ell - 1, 0\}$   
12 return LCP[ $1..n$ ]
```

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

Kasai's algorithm – Code

```
1 procedure computeLCP( $T[0..n]$ ,  $L[0..n]$ ,  $R[0..n]$ ):  
2   // Assume  $T[n] = \$$ ,  $L$  and  $R$  are suffix array and inverse  
3    $\ell := 0$   
4   for  $i := 0, \dots, n - 1$  // Consider  $T_i$  now  
5      $r := R[i]$   
6     // compute  $\text{LCP}[r]$ ; note that  $r > 0$  since  $R[n] = 0$   
7      $i_{-1} := L[r - 1]$   
8     while  $T[i + \ell] == T[i_{-1} + \ell]$  do  
9        $\ell := \ell + 1$   
10     $\text{LCP}[r] := \ell$   
11     $\ell := \max\{\ell - 1, 0\}$   
12  return  $\text{LCP}[1..n]$ 
```

- ▶ 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. “≠”
- ▶ each \neq ends iteration of for-loop
 $\rightsquigarrow \leq n$ cmps
- ▶ each $=$ implies increment of ℓ ,
but $\ell \leq n$ and
decremented $\leq n$ times
 $\rightsquigarrow \leq 2n$ cmps


$\rightsquigarrow \Theta(n)$ overall time

Enhanced Suffix Arrays

- ▶ *Enhanced suffix array*: L , R and LCP array
- ▶ modern version of suffix trees

 can be harder to reason about

 but use much less space

 simple(r) linear-time construction

Enhanced Suffix Arrays

- ▶ *Enhanced suffix array*: L , R and LCP array
- ▶ modern version of suffix trees

👎 can be harder to reason about

👍 but use much less space

👍 simple(r) linear-time construction

- ▶ **Goal:** directly simulate any suffix tree operations on L and LCP arrays

- ▶ string matching ✓
- ▶ string matching in $O(m)$ time ✗ (= you have not yet seen how to do this \neq not possible!)
- ▶ string depth of internal nodes = LCP values ✓
- ▶ internal suffix tree node = LCP interval ✓
 - ↪ storing information per node ✓
 - ▶ bottom-up traversal via enclosing LCP intervals ✓
- ▶ longest common extension queries ✗
- ▶ suffix links ✗