

APPLIED ALGORITHMS \$ APPLIED  
APPLIED ALGORITHMS \$  
CS \$ APPLIED ALGORITHMI  
DALGORITHMICS \$ APPLIE  
ED ALGORITHMS \$ APPLI  
GORITHMICS \$ APPLIEDAL  
HMICS \$ APPLIEDALGORIT  
ICS \$ APPLIEDALGORITHM  
IEDALGORITHMS \$ APPL  
ITHMICS \$ APPLIEDALGOR  
LGORITHMS \$ APPL  
MICS \$ APPLIEDALGOR  
OPTHMICS \$ APPLIEDALGO  
PPLIEDALGORITHMS \$ A  
RITHMICS \$ APPLIEDALGO  
S \$ APPLIEDALGORITHIC  
THMICS \$ APPLIEDALGORI

# 6 Text Indexing – Searching whole genomes

10 March 2022

Sebastian Wild

# Learning Outcomes

1. Know and understand methods for text indexing: *inverted indices*, *suffix trees*, *(enhanced) suffix arrays*
2. Know and understand *generalized suffix trees*
3. Know properties, in particular *performance characteristics*, and limitations of the above data structures.
4. Design (simple) *algorithms based on suffix trees*.
5. Understand *construction algorithms* for suffix arrays and LCP arrays.

## Unit 6: Text Indexing



# Outline

## 6 Text Indexing

- 6.1 Motivation
- 6.2 Suffix Trees
- 6.3 Applications
- 6.4 Longest Common Extensions
- 6.5 Suffix Arrays
- 6.6 Linear-Time Suffix Sorting
- 6.7 The LCP Array

## 6.1 Motivation

# Text indexing

- ▶ *Text indexing* (also: *offline text search*):


- ▶ case of string matching: find  $P[0..m]$  in  $T[0..n]$

- ▶ but with *fixed* text  $\rightsquigarrow$  preprocess  $T$  (instead of  $P$ )

- $\rightsquigarrow$  expect many queries  $P$ , answer them without looking at all of  $T$

- $\rightsquigarrow$  essentially a data structuring problem: “building an *index* of  $T$ ”

Latin: “one who points out”



- ▶ application areas

- ▶ web search engines

- ▶ online dictionaries

- ▶ online encyclopedia

- ▶ DNA/RNA data bases

- ▶ ... searching in any collection of text documents (that grows only moderately)

# Inverted indices

same as "indexes"

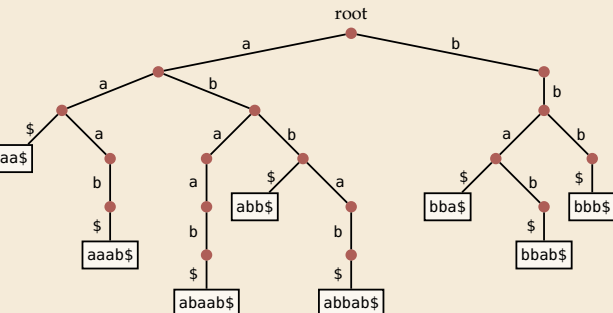
- ▶ original indices in books: list of (key) words  $\mapsto$  page numbers where they occur
  - ▶ assumption: searches are only for **whole** (key) **words**
- $\rightsquigarrow$  often reasonable for natural language text

## Inverted index:

- ▶ collect all words in  $T$ 
  - ▶ can be as simple as splitting  $T$  at whitespace
  - ▶ actual implementations typically support *stemming* of words  
goes  $\rightarrow$  go, cats  $\rightarrow$  cat
- ▶ store mapping from words to a list of occurrences  $\rightsquigarrow$  *how?*

# Tries

- ▶ efficient dictionary data structure for strings
- ▶ name from **re**trieval, but pronounced “try”
- ▶ tree based on symbol comparisons
- ▶ **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
  - ▶ strings of same length ✓
  - ▶ strings have “end-of-string” marker \$ ✓



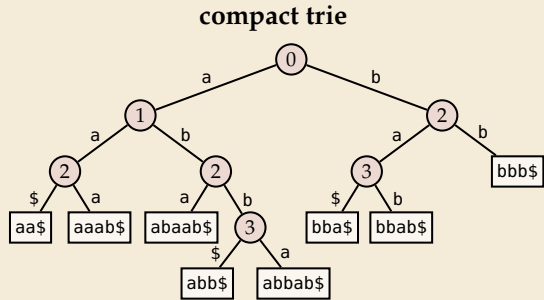
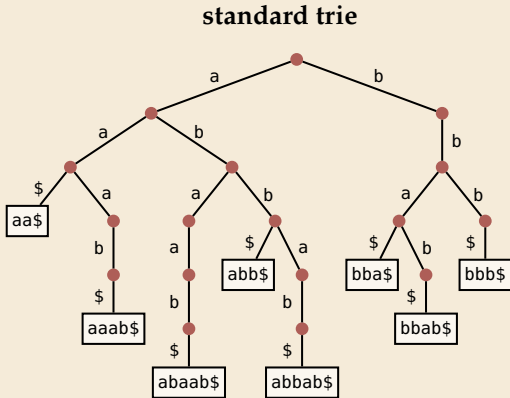
▶ **Example:**

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

# Compact tries

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

=1 child



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

- ▶ all nodes  $\geq 2$  children  $\rightsquigarrow$  #nodes  $\leq$  #leaves = #strings  $\rightsquigarrow$  linear space



# Tries as inverted index

👍 simple

👍 fast lookup

👎 cannot handle more general queries:

- ▶ search part of a word
- ▶ search phrase (sequence of words)

👎 what if the 'text' does not even have words to begin with?!

▶ biological sequences

```
ACAAGATGCCATTGTCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCTGCCCTGGAGGGTGGCCCCACCGGC  
CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCAGGC  
CAGTGCCGGGCCCCCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCAGCAATCCGCGCGCCGGGACAGAA  
TGCCCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCTGCAAATAAAACCTACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
```

▶ binary streams

```
00000010101001111010111000001111100011111011111001101101000011100010011011110000010001101010  
0110110000110101101000000010000000011101011000001000011110101110110010001100101101110111111  
11000101000101100101000000111010101001100000001101100001100111110000101 0101011101111000011  
1010111001001010101010000011111010011000000111100110101000000100100100000101100011000110111
```

↪ need new ideas

## 6.2 Suffix Trees

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

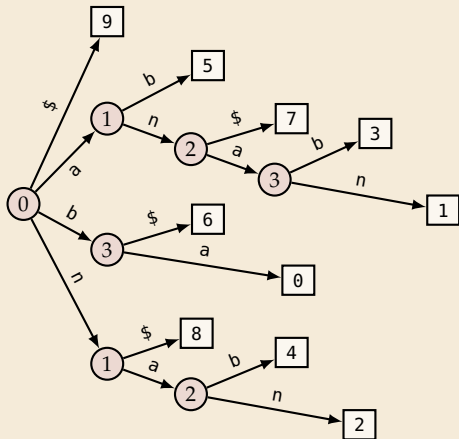
## Example:

$T = \text{bananaban\$}$

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  
 $T = \boxed{\text{b}} \boxed{\text{a}} \boxed{\text{n}} \boxed{\text{a}} \boxed{\text{n}} \boxed{\text{a}} \boxed{\text{b}} \boxed{\text{a}} \boxed{\text{n}} \boxed{\text{\$}}$

- ▶ also: edge labels like in compact trie
- ▶ (more readable form on slides to explain algorithms)



# Suffix trees – Construction

- ▶  $T[0..n)$  has  $n + 1$  suffixes (starting at character  $i \in [0..n)$ )
- ▶ We can build the suffix tree by inserting each suffix of  $T$  into a compressed trie. But that takes time  $\Theta(n^2)$ .  $\rightsquigarrow$  not interesting!



same order of growth as reading the text!

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

- ▶ algorithms are a bit tricky to understand
- ▶ but were a theoretical breakthrough
- ▶ and they are efficient in practice (and heavily used)!

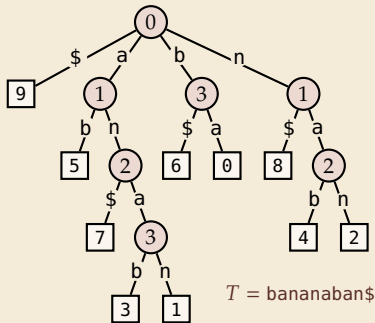
$\rightsquigarrow$  for now, take linear-time construction for granted. What can we do with them?

## 6.3 Applications

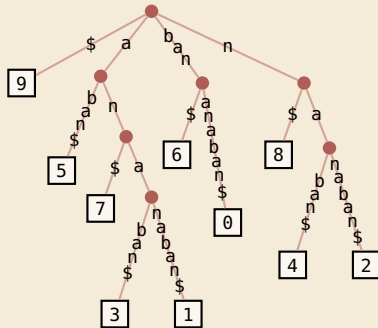
# Applications of suffix trees

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

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



but think about this:



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

# Application 1: Text Indexing / String Matching

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

▶ we have all suffixes in  $\mathcal{T}$ !

↪ (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  $v$

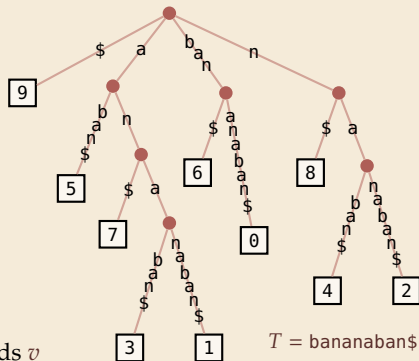
**3. we run out of tree**

reach a leaf  $\ell$  with part of  $P$  left ↪ compare  $P$  to  $\ell$ .



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:

▶  $P = \text{ann}$

▶  $P = \text{ana}$

▶  $P = \text{briar}$



## Application 2: Longest repeated substring

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

e.g. for compression  $\rightsquigarrow$  Unit 7



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

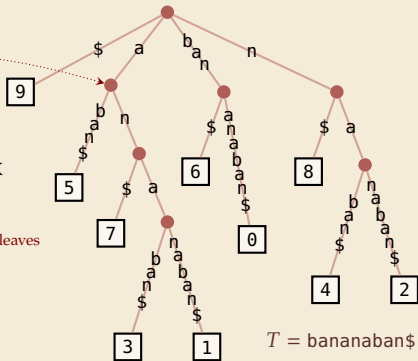
$\rightsquigarrow$  longest repeated substring = longest common prefix (LCP) of two suffixes

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  $\rightsquigarrow \Theta(n)$  time



$T = \text{bananaban\$}$

# 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  $T := T^{(1)}\$_1 T^{(2)}\$_2 \dots T^{(k)}\$_k$  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  $T_i^{(j)} = T^{(j)}[i..n_j]$



## Application 3: Longest common substring

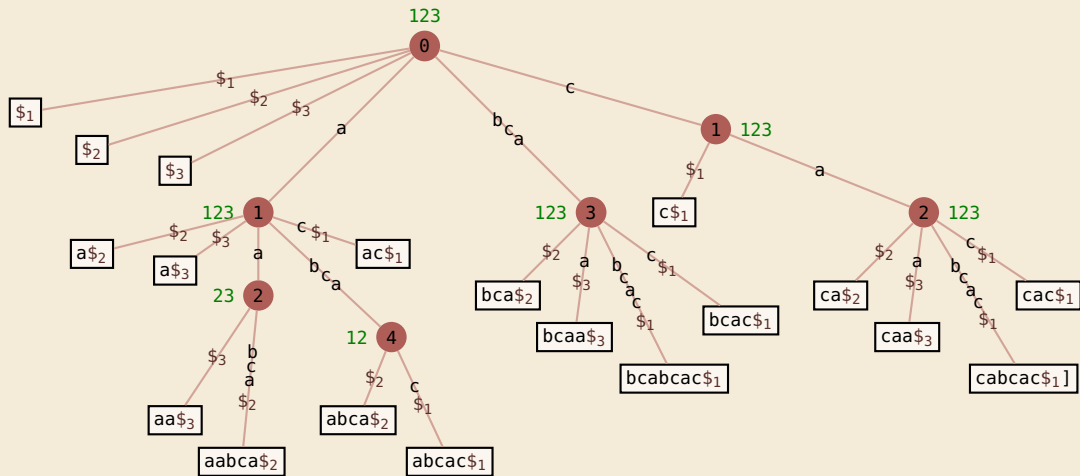
- ▶ With that new idea, we can find longest common superstrings:
  1. Compute generalized suffix tree  $\mathcal{T}$ .
  2. Store with each node the *subset of strings* that contain its path label:
    - 2.1. Traverse  $\mathcal{T}$  bottom-up.
    - 2.2. For a leaf  $(j, i)$ , the subset is  $\{j\}$ .
    - 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  $\{1, \dots, k\}$ .
  
- ▶ Each step takes time  $\Theta(n)$  for  $n = n_1 + \dots + n_k$  the total length of all texts.

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

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

# Longest common substring – Example

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



## **6.4 Longest Common Extensions**

## Application 4: Longest Common Extensions

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

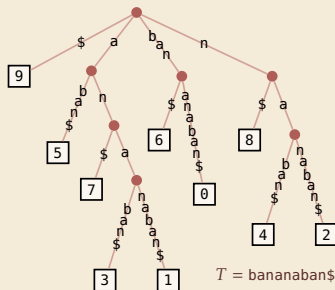
The *longest common extension (LCE)* data structure:

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

↪ 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 



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

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

and suffix tree construction inside ...



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

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

## Application 5: Approximate matching

*k*-mismatch matching:

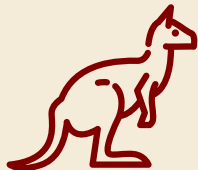
- ▶ **Input:** text  $T[0..n)$ , pattern  $P[0..m)$ ,  $k \in [0..m)$
- ▶ **Output:** “Hamming distance  $\leq k$ ”
  - ▶ smallest  $i$  so that  $T[i..i+m)$  are  $P$  differ in at most  $k$  characters
  - ▶ or NO\_MATCH if there is no such  $i$

↪ searching with typos

- ▶ Assume longest common extensions in  $T_1 P_2$  can be found in  $O(1)$ 
  - ↪ generalized suffix tree  $\mathcal{T}$  has been built
  - ↪ string depths of all internal nodes have been computed
  - ↪ constant-time LCA data structure for  $\mathcal{T}$  has been built



# Kangaroo Algorithm for approximate matching



---

```
1 procedure kMismatch( $T[0..n - 1], P[0..m - 1]$ )
2   // build LCE data structure
3   for  $i := 0, \dots, n - m - 1$  do
4     mismatches := 0;  $t := i$ ;  $p := 0$ 
5     while mismatches  $\leq k \wedge p < m$  do
6        $\ell := \text{LCE}(t, p)$  // jump over matching part
7        $t := t + \ell + 1$ ;  $p := p + \ell + 1$ 
8       mismatches := mismatches + 1
9     if  $p == m$  then
10      return  $i$ 
```

---

► **Analysis:**  $\Theta(n + m)$  preprocessing +  $O(n \cdot k)$  matching

↪ very efficient for small  $k$

► State of the art

- $O(n \frac{k^2 \log k}{m})$  possible with complicated algorithms
- extensions for edit distance  $\leq k$  possible

## Application 6: Matching with wildcards

- ▶ Allow a wildcard character in pattern

stands for arbitrary (single) character

unit*	<i>P</i>
in_unit5_we_will	<i>T</i>

- ▶ similar algorithm as for  $k$ -mismatch  $\rightsquigarrow O(n \cdot k + m)$  when  $P$  has  $k$  wildcards

\* \* \*

Many more applications, in particular for problems on biological sequences

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

## Suffix trees – Discussion

► Suffix trees were a threshold invention

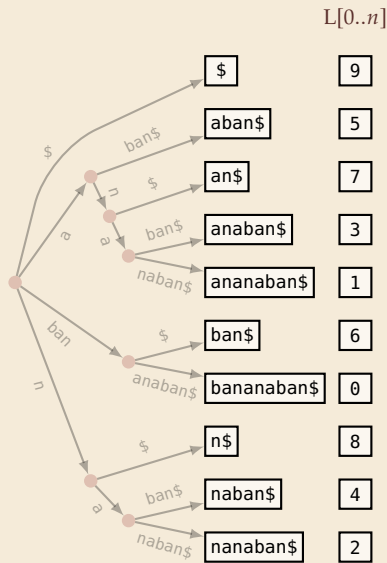
- 👍 linear time and space
- 👍 suddenly many questions efficiently solvable in theory

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



## 6.5 Suffix Arrays

# Putting suffix trees on a diet



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

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

► Enough to do efficient string matching!

1. Use binary search for pattern  $P$

2. check if  $P$  is prefix of suffix after position found

► **Example:**  $P = ana$

↪  $L[0..n]$  is called *suffix array*:

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

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

# Suffix arrays – Construction

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

- ▶ from suffix tree
  - ▶ possible with traversal . . .
  - 👎 but we are trying to avoid constructing suffix trees!
  
- ▶ sorting the suffixes of  $T$  using general purpose sorting method
  - 👍 trivial to code!
  - ▶ but: comparing two suffixes can take  $\Theta(n)$  character comparisons
  - 👎  $\Theta(n^2 \log n)$  time in worst case
  
- ▶ We do better!

# Fat-pivot radix quicksort – Example

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

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

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

sells  
seashells  
sea  
sells  
seashells  
she\$  
shells  
she\$  
shore  
the  
the

seashells  
sea  
seashells  
sells  
sells  
she  
she  
shells  
the  
the

seashells  
sea\$  
seashells  
sells  
sells

sea  
seashells  
seashells  
sells  
sells

...

# Fat-pivot radix quicksort

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

- ▶ **partition** based on  $d$ th character only (initially  $d = 0$ )
- ↪ 3 segments: smaller, equal, or larger than  $d$ th symbol of pivot
- ▶ recurse on smaller and large with same  $d$ , on equal with  $d + 1$ 
  - ↪ never compare equal prefixes twice

↪ can show:  $\sim 2 \ln(2) \cdot n \lg n \approx 1.39n \lg n$  character comparisons on average <sup>for random strings</sup>

👍 simple to code

👍 efficient for sorting many lists of strings

- ▶ fat-pivot radix quicksort finds suffix array in  $O(n \log n)$  expected time <sup>random string</sup>

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



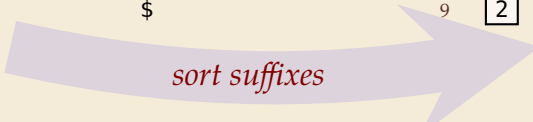
## 6.6 Linear-Time Suffix Sorting

# 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 <sup>th</sup>	bananaban\$	right $R[0] = 6$	0	9	\$
1	4 <sup>th</sup>	ananaban\$		1	5	aban\$
2	9 <sup>th</sup>	nanaban\$	left $L[8] = 4$	2	7	an\$
3	3 <sup>th</sup>	anaban\$		3	3	anaban\$
4	8 <sup>th</sup>	naban\$		4	1	ananaban\$
5	1 <sup>th</sup>	aban\$		5	6	ban\$
6	5 <sup>th</sup>	ban\$		6	0	bananaban\$
7	2 <sup>th</sup>	an\$		7	8	n\$
8	7 <sup>th</sup>	n\$		8	4	naban\$
9	0 <sup>th</sup>	\$		9	2	nanaban\$



# Linear-time suffix sorting

## DC3 / Skew algorithm

*not a multiple of 3*

1. Compute rank array  $R_{1,2}$  for suffixes  $T_i$  starting at  $i \not\equiv 0 \pmod{3}$  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  $R_3$  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 (!)

- ▶ 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  $R_{1,2}[4]$ !

↪  $T_0$  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\$\$\$  
 $T_9$  sbananasman\$\$\$  
 $T_{12}$  nanasman\$\$\$  
 $T_{15}$  asman\$\$\$  
 $T_{18}$  an\$\$\$  
 $T_{21}$  \$\$

smans\$\$\$ =  $T_{16}$

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

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

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

$R_{1,2}$  (known)

radix sort

$T_{21}$	\$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\$\$\$

$T_{22}$  \$  
 $T_{21}$  \$\$  
 $T_{20}$  \$\$\$  
 $T_4$  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  $R_{1,2}$

↪  $O(n)$  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}$$$$$

↪ 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}$$$  
 $= \text{aanasman}$$$  
 $= aT_{13}$$$$$

## DC3 / Skew algorithm – Fix recursive call

- ▶ both step 2. and 3. doable in  $O(n)$  time!
  - ▶ But: we cheated in 1. step! “compute rank array  $R_{1,2}$  recursively”
    - ▶ Taking a *subset* of suffixes is *not* an instance of the same problem!
- ↪ Need a single *string*  $T'$  to recurse on, from which we can deduce  $R_{1,2}$ .



How can we make  $T'$  “skip” some suffixes?



redefine alphabet to be *triples of characters* `abc`

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

▶  $T' = T[1..n]^\square \text{ $$$ } T[2..n]^\square \text{ $$$} \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$$$}$   
↪  $T^\square = \text{ban ana ban $$$}$   
 $\text{ana ban $$$}$   
 $\text{ban $$$}$   
 $\text{$$$}$

## DC3 / Skew algorithm – Fix alphabet explosion

▶ Still does not quite work!

▶ Each recursive step *cubes*  $\sigma$  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  $\Sigma$ ).

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



## DC3 / Skew algorithm – Step 3. Example

$$T' = T[1..n) \square \square \square T[2..n) \square \square \square$$

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

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

▶ Occurring triples:

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

▶ Sorted triples with ranks:

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

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

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

## Suffix array – Discussion

- 👍 sleek data structure compared to suffix tree
- 👍 simple and fast  $O(n \log n)$  construction
- 👍 more involved but optimal  $O(n)$  construction
- 👍 supports efficient string matching
- 👎 string matching takes  $O(m \log n)$ , not optimal  $O(m)$
- 👎 Cannot use more advanced suffix tree features  
e. g., for longest repeated substrings



## 6.7 The LCP Array

# 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**?

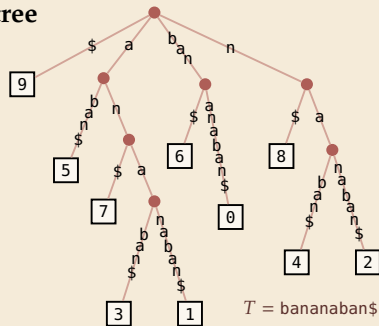
- ▶ Yes, by **enhancing** the suffix array with the **LCP array**!

$LCP[1..n]$

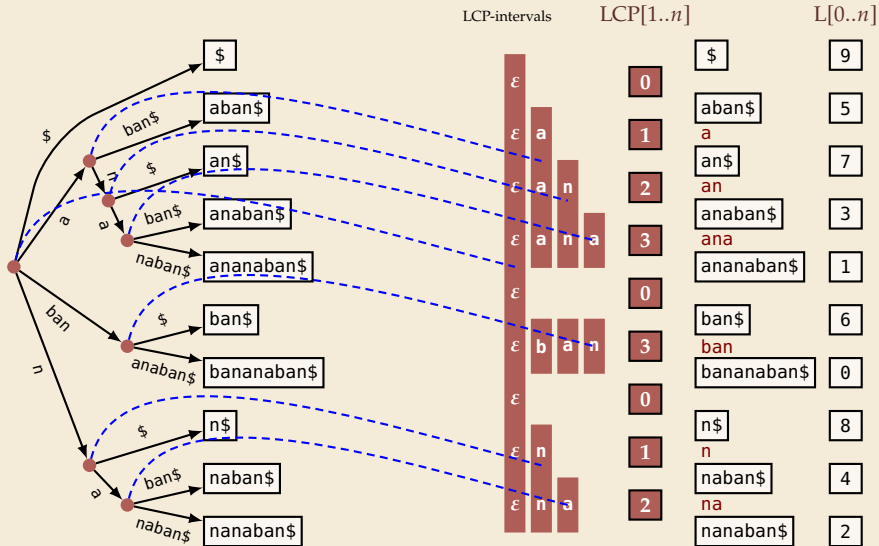
$$LCP[r] = LCP(T_{L[r]}, T_{L[r-1]})$$

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

↪ longest repeated substring = find maximum in  $LCP[1..n]$



# LCP array and internal nodes



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

# 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:

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

$$T_{L[1]} = \text{alo\_buffalo\$}$$

$$T_{L[2]} = \text{alo\_buffalo\_buffalo\$}$$

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

$$T_{L[3]} = \text{alo\_buffalo\_buffalo\_buffalo\$}$$

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

$$T_{L[?]} = \text{lo\_buffalo\_buffalo\$}$$

$$\text{lo\_buffalo\_buffalo} \rightsquigarrow LCP[?] = 18$$

$$T_{L[?]} = \text{lo\_buffalo\_buffalo\_buffalo\$}$$

↑  
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 <sup>th</sup>	bananaban\$	0	9	\$	–
1	4 <sup>th</sup>	ananaban\$	1	5	aban\$	0
2	9 <sup>th</sup>	nanaban\$	2	7	an\$	1
3	3 <sup>th</sup>	anaban\$	3	3	anaban\$	2
4	8 <sup>th</sup>	naban\$	4	1	ananaban\$	3
5	1 <sup>th</sup>	aban\$	5	6	ban\$	0
6	5 <sup>th</sup>	ban\$	6	0	bananaban\$	3
7	2 <sup>th</sup>	an\$	7	8	n\$	0
8	7 <sup>th</sup>	n\$	8	4	naban\$	1
9	0 <sup>th</sup>	\$	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  $i := 0, \dots, n - 1$ 
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  $\ell$  of induced common prefix
- ▶ use  $L$  to get start index of suffixes

## Analysis:

- ▶ dominant operation:  
character comparisons
  - ▶ separately count those with  
outcomes “=” resp. “≠”
  - ▶ each ≠ ends iteration of for-loop  
↪  $\leq n$  cmps
  - ▶ each = implies increment of  $\ell$ ,  
but  $\ell \leq n$  and  
decremented  $\leq n$  times  
↪  $\leq 2n$  cmps
- ↪  $\Theta(n)$  overall time

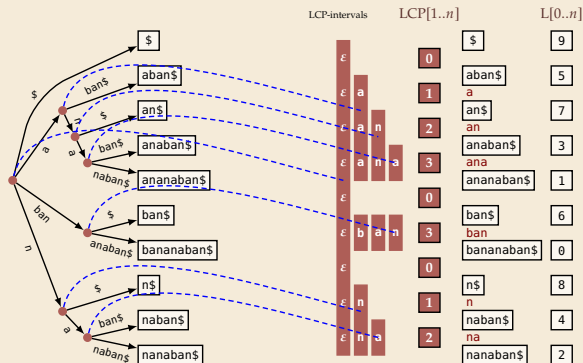


# Back to suffix trees

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



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



## Conclusion

▶ *(Enhanced) Suffix Arrays* are the modern version of suffix trees

👎 can be harder to reason about

👍 can support same algorithms as suffix trees

👍 but use much less space

👍 simpler linear-time construction