

# 6 Text Indexing – Searching whole genomes

*9 March 2020*

Sebastian Wild

## 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  $P[0..m-1]$  in  $T[0..n-1]$

- ▶ 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**  $\parallel$
- $\rightsquigarrow$  often reasonable for natural language text

# Inverted indices

same as "indexes"

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

## Inverted index:

- ▶ collect all words in  $T$ 
    - ▶ can be as simple as splitting  $T$  at whitespace
    - ▶ actual implementations typically support *stemming* of words  
goes  $\rightarrow$  go, cats  $\rightarrow$  cat
  - ▶ store mapping from words to a list of occurrences  $\rightsquigarrow$  how? not here BST
- $go \mapsto \{5, 10, 20\}$   
 $cat \mapsto \{4, 21\}$

## Clicker Question



Do you know what a *trie* is?

- ☐ A A what? No!
- ☐ B I have heard the term, but don't quite remember.
- ☐ C I remember hearing about it in a module.
- ☐ D Sure.

[pingo.upb.de/622222](https://pingo.upb.de/622222)

# Tries

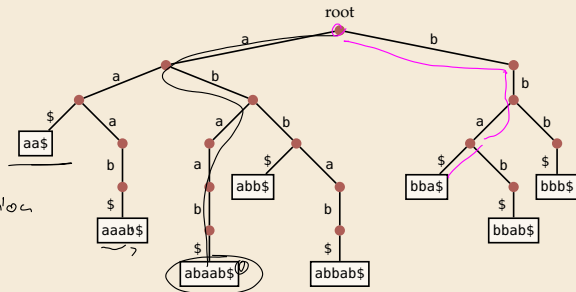
- ▶ efficient dictionary data structure for strings
- ▶ name from **re**trieval, but pronounced “try”
- ▶ tree based on symbol comparisons
- ▶ **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
  - ▶ strings of same length ✓
  - ▶ strings have “end-of-string” marker \$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

► Example:

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

- o construction: top-down  
independent of order of insertion

- query: (get)ex.lba\$

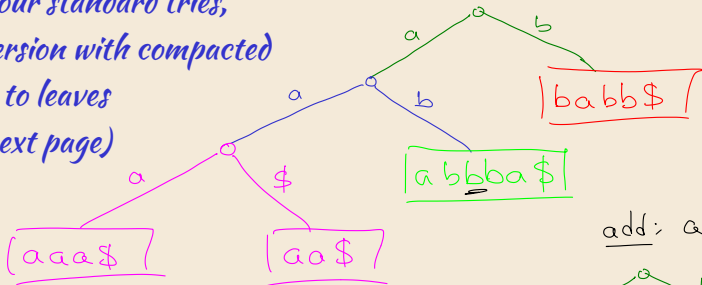

$$\{aa, a\}$$



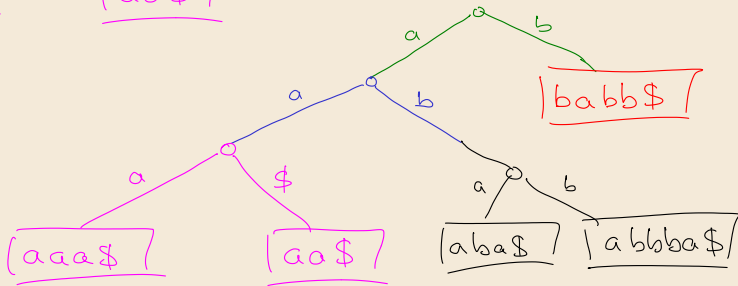

Construction

{aaa\$, bab\$, aa\$, abba\$}

NOT our standard tries,  
but version with compacted  
paths to leaves  
(see next page)



add: a ba\$

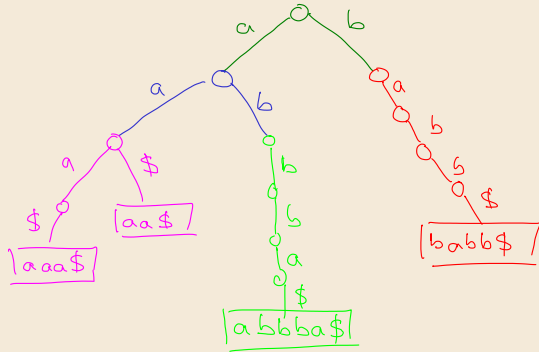
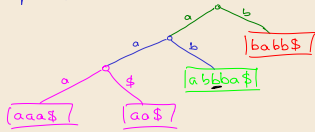


# Trie construction (correct version)

{aaa\$, babb\$, aa\$, abbbba\$}

## standard trie

trie with compacted  
paths to leaves



## Clicker Question

Suppose we have a trie that stores  $n$  strings over  $\Sigma = \{A, \dots, Z\}$ .  
Each stored string consists of  $m$  characters.

We now search for a query string  $Q$  with  $|Q| = q$ .

How many **nodes** in the trie are **visited** during this **query**?



- |                                   |                                   |
|-----------------------------------|-----------------------------------|
| <b>A</b> $\Theta(\log n)$         | <b>F</b> $\Theta(\log m)$         |
| <b>B</b> $\Theta(\log(nm))$       | <b>G</b> $\Theta(q)$              |
| <b>C</b> $\Theta(m \cdot \log n)$ | <b>H</b> $\Theta(\log q)$         |
| <b>D</b> $\Theta(m + \log n)$     | <b>I</b> $\Theta(q \cdot \log n)$ |
| <b>E</b> $\Theta(m)$              | <b>J</b> $\Theta(q + \log n)$     |

[pingo.upb.de/622222](https://pingo.upb.de/622222)

## Clicker Question

Suppose we have a trie that stores  $n$  strings over  $\Sigma = \{A, \dots, Z\}$ .  
Each stored string consists of  $m$  characters.

We now search for a query string  $Q$  with  $|Q| = q$ . <sup>successful</sup>  
How many nodes in the trie are **visited** during this **query**?



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

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

**B**  ~~$\Theta(\log(nm))$~~

**G**  $\Theta(q)$  ✓

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

**H**  ~~$\Theta(\log q)$~~

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

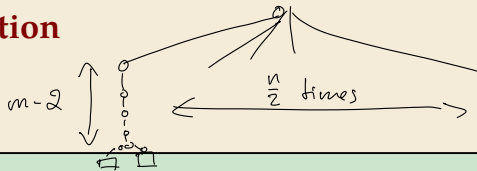
**I**  ~~$\Theta(q \log n)$~~

**E**  ~~$\Theta(m)$~~

**J**  ~~$\Theta(q + \log n)$~~

[pingo.upb.de/622222](https://pingo.upb.de/622222)

## Clicker Question



$$\begin{aligned} S_1 &= \$_1 \text{ a a a a o o c c b} \\ S_2 &= \$_1 \text{ c a c c c c c a} \\ S_3 &= \$_2 \\ S_4 &= \$_2 \dots \end{aligned}$$

Suppose we have a trie that stores  $n$  strings over  $\Sigma = \{A, \dots, Z\}$ . Each stored string consists of  $m$  characters.

How many **nodes** does the trie have **in total** in the worst case?



- |                              |                             |
|------------------------------|-----------------------------|
| <b>A</b> $\Theta(n)$         | <b>D</b> $\Theta(n \log m)$ |
| <b>B</b> $\Theta(n + m)$     | <b>E</b> $\Theta(m)$        |
| <b>C</b> $\Theta(n \cdot m)$ | <b>F</b> $\Theta(m \log n)$ |

[pingo.upb.de/622222](https://pingo.upb.de/622222)

## Clicker Question



Suppose we have a trie that stores  $n$  strings over  $\Sigma = \{A, \dots, Z\}$ . Each stored string consists of  $m$  characters.

How many **nodes** does the trie have **in total** *in the worst case*?

**A**  ~~$\Theta(n)$~~

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

**B**  ~~$\Theta(n + m)$~~

**E**  ~~$\Theta(m)$~~

**C**  $\Theta(n \cdot m)$  ✓

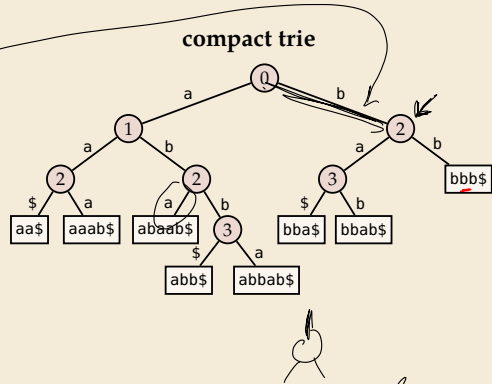
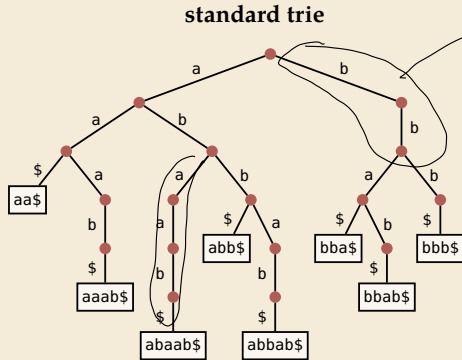
**F**  ~~$\Theta(m \log n)$~~

[pingo.upb.de/622222](https://pingo.upb.de/622222)

# Compact tries

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

o  $\triangle$  get needs extra check bab\$





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


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

$\Theta(n)$  not  $\Theta(mn)$

# Tries as inverted index

 simple

 fast lookup

 cannot handle more general queries:

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



# Tries as inverted index

👍 simple

👍 fast lookup

👎 cannot handle more general queries:

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

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

- ▶ biological sequences

```
ACAAGATGCCATTGTCCCCGGCCTCCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCTGGAGGGTGGCCCCACCGGC  
CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCAGGC  
CAGTGCCGGGCCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCAGGAGGCGACCCCCCAGCAATCCGCGCGCCGGGACAGAA  
TGCCCTGCAGGAACCTTCTTCTGGAAGACCTTCTCCTCGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
```

- ▶ binary streams

```
00000010101001111010111000001111100011111011111001101101000011100010011011110000010001101010  
0110110000110101101000000010000000011101011000001000011110101110110010001100101101110111111  
110001010001011001010000001110101010011000000001101100001100111110000101 010101110111000011  
10101110010010101010100000111110100110000001111001101010000000100100100000101100011000110111
```

~> need new ideas

## 6.2 Suffix Trees

# Suffix trees – A ‘magic’ data structure

$S[i..j]$

**Appetizer:** Longest common substring problem

- ▶ Given: strings  $S_1, \dots, S_k$       **Example:**  $S_1 = \text{superiorcaliforniallives}$ ,  $S_2 = \text{sealiver}$
- ▶ 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  $\Theta(|S_1| + \dots + |S_k|)$ ? 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  $T = T[0..n-1]$  = compact trie of all suffixes of  $T$ \$ (set  $T[n] := \$$ )

$n+1$  suffixes

# Suffix trees – Definition

- suffix tree  $\mathcal{T}$  for text  $T = T[0..n-1]$  = compact trie of all suffixes of  $T\$$  (set  $T[n] := \$$ )

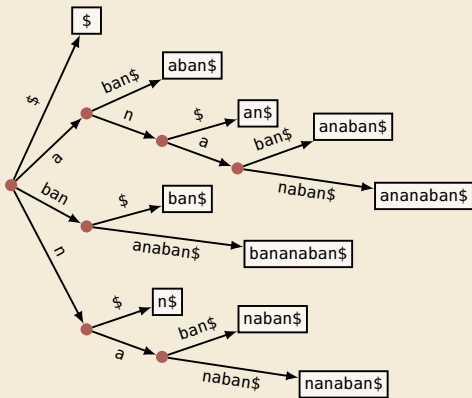
## Example:

$T = \text{bananaban\$}$

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  $T = T[0..n-1]$  = compact trie of all suffixes of  $T\$$  (set  $T[n] := \$$ )
- ▶ except: in leaves, store *start index* (instead of actual string)

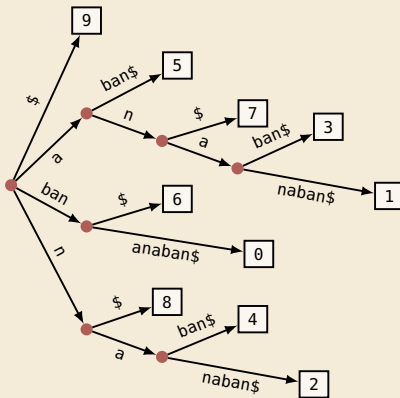
## Example:

$T = \text{bananaban\$}$

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
b	a	n	a	n	a	b	a	n	\$

$T =$





# Suffix trees – Definition

- ▶ suffix tree  $\mathcal{T}$  for text  $T = T[0..n-1]$  = compact trie of all suffixes of  $T\$$  (set  $T[n] := \$$ )
- ▶ except: in leaves, store *start index* (instead of actual string)

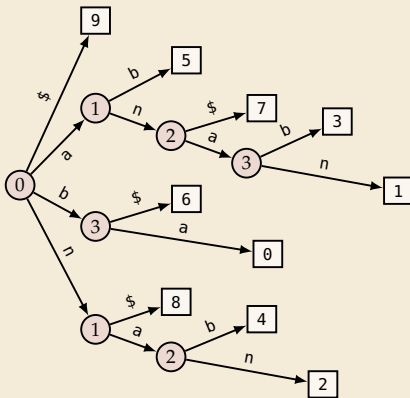
## Example:

$T = \text{bananaban\$}$

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
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

- ▶  $T[0..n-1]$  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!

Handwritten diagram illustrating a compressed trie node. The root node contains the string "banana\$". A diagonal line descends from the left side of the root, and a vertical line descends from the right side. The string "anana\$" is written between these two diagonal lines, representing the next node in the trie.

# Suffix trees – Construction

- ▶  $T[0..n-1]$  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  $\underline{\Theta(n)}$  time!

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

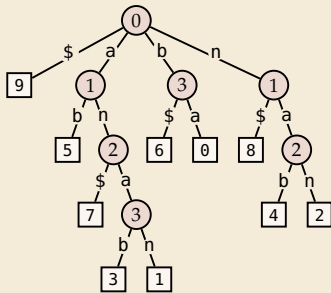
$\rightsquigarrow$  for now, take linear-time construction for granted. What can we do with them?

## 6.3 Applications

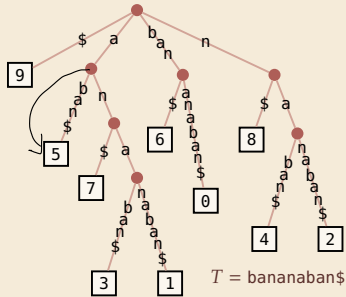
# Applications of suffix trees

- 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 \$)  
 $\downarrow$

## 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)

$\rightsquigarrow P$  does not occur in  $T$

2. we run out of pattern

reach end of  $P$  at internal node  $v$  or inside edge towards  $v$

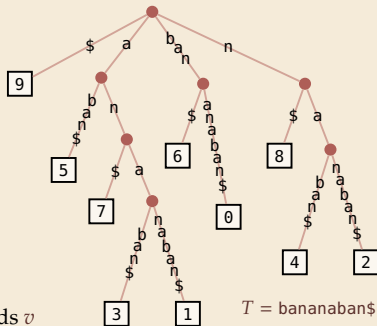
→  $P$  occurs at all leaves in subtree of  $v$

- ### 3. we run out of tree

reach a leaf  $\ell$  with part of  $P$  left  $\rightsquigarrow$  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!

## 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

- $\alpha_{nn}$  — at internal node (no node with next character of  $P$ )

- bb — or *inside edge* (mismatch of next characters)

$\rightsquigarrow P$  does not occur in  $T$

- 2. we run out of pattern

reach end of  $P$  at internal node  $v$  or inside edge towards  $v$

→  $P$  occurs at all leaves in subtree of  $v$

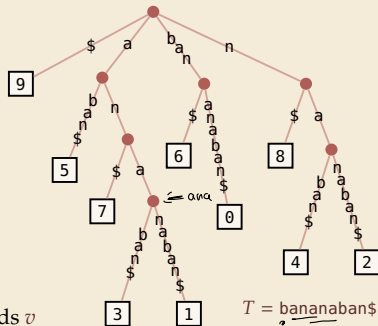
- ### 3. we run out of tree

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



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

- Finding <sup>one</sup> first match (or NO\_MATCH) takes  $O(|P|)$  time!



$T = \text{bananaban\$}$

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*?

## 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*?



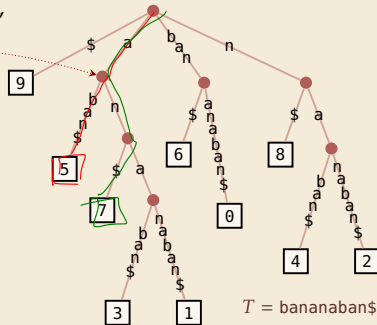
Repeated substrings = shared paths in *suffix tree*



- $T_5 = \text{aban\$}$  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



## 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*



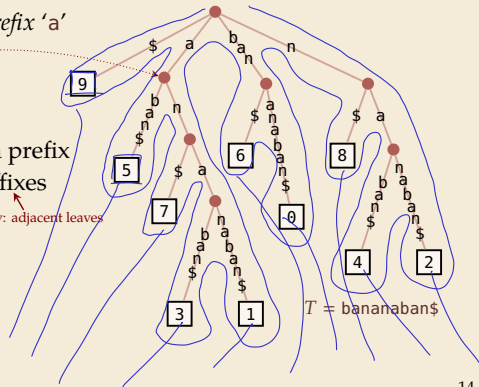
- $T_5 = \text{aban\$}$  and  $T_7 = \text{an\$}$  have *longest common prefix* 'a'

$\rightsquigarrow \exists$  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



## 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

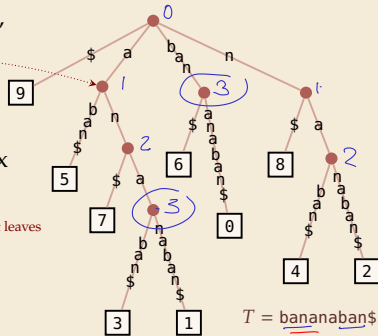
↪ longest repeated substring = longest common prefix (LCP) of two suffixes

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



## 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

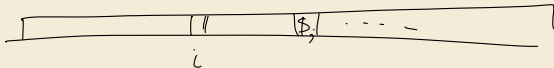
# Generalized suffix trees

- ▶ 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  $T := \underline{T^{(1)}\$_1 T^{(2)}\$_2 \dots T^{(k)}\$_k}$  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.

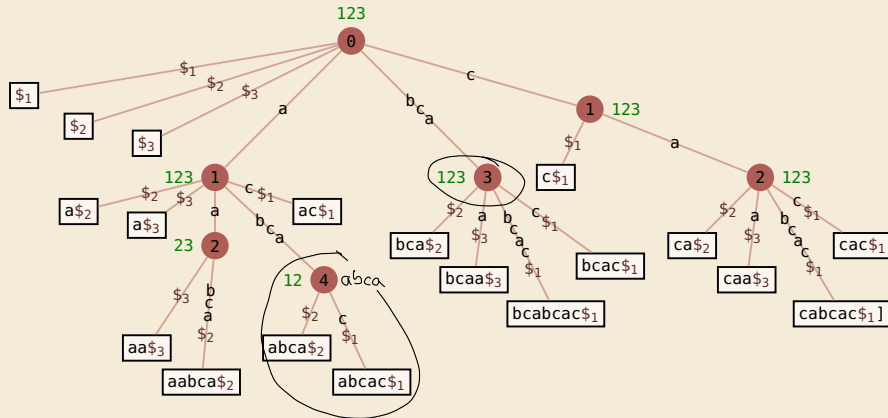
*“Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible.”* [Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]



# Longest common substring – Example

$T^{(1)} = bc\text{ab}\underline{c}ac$ ,  $T^{(2)} = aab\text{c}\underline{a}$ ,  $T^{(3)} = bcaa$

$T = bcab\text{c}ac\$, aab\text{c}a\$, bcaa\$_3$



## 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:

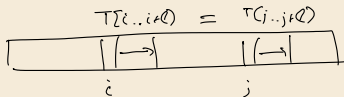
- ▶ **Given:** String  $T[0..n-1]$
- ▶ **Goal:** Answer LCE queries, i. e.,  
given positions  $i, j$  in  $T$ ,  
how far can we read the same text from there?  
formally:  $\text{LCE}(i, j) = \max\{\ell : T[i..i+\ell] = T[j..j+\ell]\}$

# Application 4: Longest Common Extensions

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

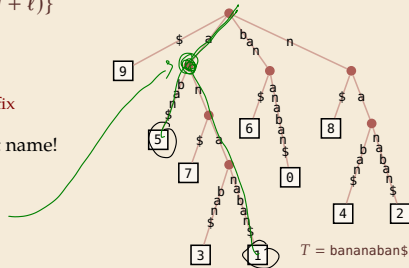
The *longest common extension (LCE)* data structure:

- **Given:** String  $T[0..n-1]$
- **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:  $LCE(i, j) = LCP(T_i, T_j) = \text{stringDepth}(\text{LCA}(\boxed{i}, \boxed{j}))$

$T_5$  aban\$  
 $T_1$  ananaban\$

# 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  $\rightsquigarrow \underline{\Theta(n^2)}$  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 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.

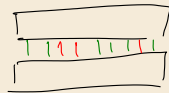
$\rightsquigarrow$  After linear preprocessing (time & space), we can find LCEs in  $O(1)$  time.

## Application 5: Approximate matching

### $k$ -mismatch matching:

- ▶ **Input:** text  $T[0..n-1]$ , pattern  $P[0..m-1]$ ,  $k \in [0..m)$
- ▶ **Output:**
  - ▶ smallest  $i$  so that  $T[i..i+m)$  and  $P$  differ in at most  $k$  characters
  - ▶ or NO\_MATCH if there is no such  $i$

"Hamming distance  $\leq k$ "



Hamming  
distance 5

↪ searching with typos

- ▶ Assume longest common extensions in  $T\$_1P\$_2$  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

## Clicker Question



What is the Hamming distance between heart and beard?

*[pingo.upb.de/622222](https://pingo.upb.de/622222)*



# 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  $i$ 
```

brute force

$O(n \cdot m)$

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

↪ very efficient for small  $k$

► State of the art

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

## Application 6: Matching with wildcards

- ▶ Allow a wildcard character in pattern

stands for arbitrary (single) character

unit*	P
in_unit5_we_will	T

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

generalize  $k$ -mismatches to  $k$  edit distance (+ ins dels)

\*   \*   \*

Many more applications, in particular for problems on biological sequences

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

# Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention

👍 linear time and space

👍 suddenly many questions efficiently solvable in theory



# Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention

👍 linear time and space  $\Theta(n)$

👍 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



storing tries in Java •  $\Sigma = \text{ASCII}$  (128 characters)

① array[0..127] of children

② dictionary:  $\Sigma \mapsto \text{child}$   
e.g. BSTs

## 6.5 Suffix Arrays

## Clicker Question



**Recap:** Check all correct statements about suffix tree  $\mathcal{T}$  of  $T[0..n)$ .

- ☐ **A** We require  $T$  to end with \$.
- ☐ **B** The size of  $\mathcal{T}$  can be  $\Omega(n^2)$  in the worst case.
- ☐ **C**  $\mathcal{T}$  is a standard trie of all suffixes of  $T\$$ .
- ☐ **D**  $\mathcal{T}$  is a compact trie of all suffixes of  $T\$$ .
- ☐ **E** The leaves of  $\mathcal{T}$  store (a copy of) a suffix of  $T\$$ .
- ☐ **F** Naive construction of  $\mathcal{T}$  takes  $\Omega(n^2)$  (worst case).
- ☐ **G**  $\mathcal{T}$  can be computed in  $O(n)$  time (worst case).
- ☐ **H**  $\mathcal{T}$  has  $n$  leaves.

[\*pingo.upb.de/622222\*](https://pingo.upb.de/622222)

## Clicker Question



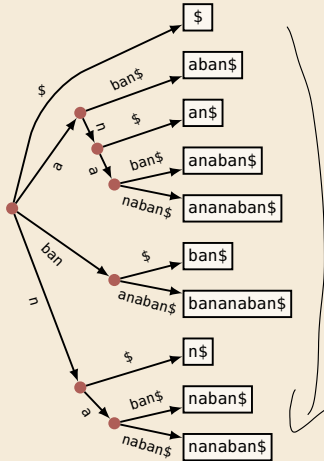
**Recap:** Check all correct statements about suffix tree  $\mathcal{T}$  of  $T[0..n)$ .

- ☒ **A** We require  $T$  to end with  $\$$ . ✓
- ☐ **B** ~~The size of  $\mathcal{T}$  can be  $\Omega(n^2)$  in the worst case.~~
- ☐ **C**  ~~$\mathcal{T}$  is a standard trie of all suffixes of  $T\$$ .~~
- ☒ **D**  $\mathcal{T}$  is a compact trie of all suffixes of  $T\$$ . ✓
- ☐ **E** ~~The leaves of  $\mathcal{T}$  store (a copy of) a suffix of  $T\$$ .~~
- ☒ **F** Naive construction of  $\mathcal{T}$  takes  $\Omega(n^2)$  (worst case). ✓
- ☒ **G**  $\mathcal{T}$  can be computed in  $O(n)$  time (worst case). ✓
- ☐ **H**  ~~$\mathcal{T}$  has  $n$  leaves.~~

[pingo.upb.de/622222](https://pingo.upb.de/622222)

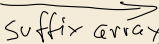
# Putting suffix trees on a diet

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



suffixes in leaves  
are sorted

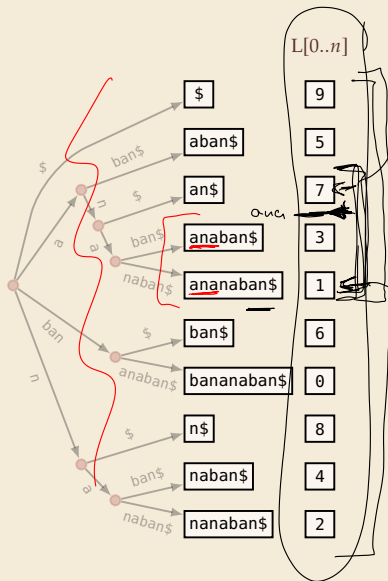




1. Use binary search for pattern  $P$

- check if  $P$  is prefix of suffix after found position

# Putting suffix trees on a diet



$L[0..n]$

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

$\underline{ana} \succ an\$$

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

- Enough to do efficient string matching!

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

- **Example:**  $P = \underline{ana}$   $\underline{ana} \succ \underline{an}anab\$$

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

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

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

## Clicker Question



**Recap:** Check all correct statements about suffix array  $L[0..n]$  and suffix tree  $\mathcal{T}$  of text  $T[0..n]$ .

- ☐ **A**  $L[0..n]$  lists the start indices of leaves of  $\mathcal{T}$  in left-to-right order.
- ☐ **B**  $T[L[r]..n]$  is the path label in  $\mathcal{T}$  to the leaf storing  $r$ .
- ☐ **C**  $T[L[r]..n]$  is the path label to the  $r$ th leaf in  $\mathcal{T}$ .
- ☐ **D**  $T_{L[r]}$  is the  $r$ th smallest suffix of  $T$  (lexicographic order).
- ☐ **E** In terms of  $\Theta$ -classes,  $\mathcal{T}$  needs more space than  $L$ .
- ☐ **F**  $L$  (and  $T$ ) suffice to solve the text indexing problem.

[\*pingo.upb.de/622222\*](https://pingo.upb.de/622222)

# Clicker Question



**Recap:** Check all correct statements about suffix array  $L[0..n]$  and suffix tree  $\mathcal{T}$  of text  $T[0..n]$ .

- A**  $L[0..n]$  lists the start indices of leaves of  $\mathcal{T}$  in left-to-right order. ✓
- B**  ~~$T[L[r]..n]$  is the path label in  $\mathcal{T}$  to the leaf storing  $r$ .~~
- C**  $T[L[r]..n]$  is the path label to the  $r$ th leaf in  $\mathcal{T}$ . ✓
- D**  $T_{L[r]}$  is the  $r$ th smallest suffix of  $T$  (lexicographic order). ✓
- E** ~~In terms of  $\Theta$  classes,  $\mathcal{T}$  needs more space than  $L$ .~~
- F**  $L$  (and  $T$ ) suffice to solve the text indexing problem. ✓

assuming

$|\Sigma| = \sigma = O(1)$

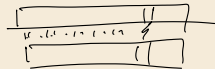
[pingo.upb.de/622222](https://pingo.upb.de/622222)

but searching  
is a bit slower

# 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 sort
  - 👍 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

general string sorting

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

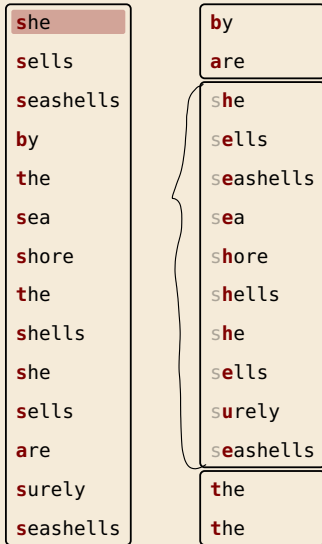
sells

are

surely

seashells

## Fat-pivot radix quicksort – Example





## Fat-pivot radix quicksort – Example



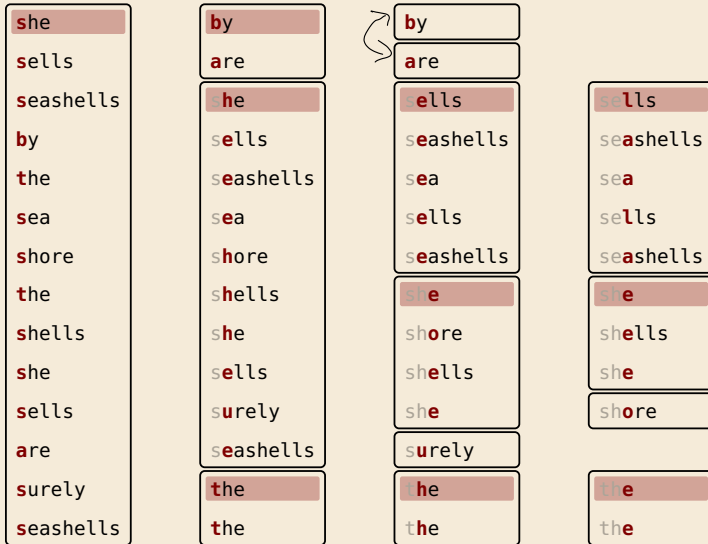
# Fat-pivot radix quicksort – Example



## Fat-pivot radix quicksort – Example

she	by	by
sells	are	are
seashells	she	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

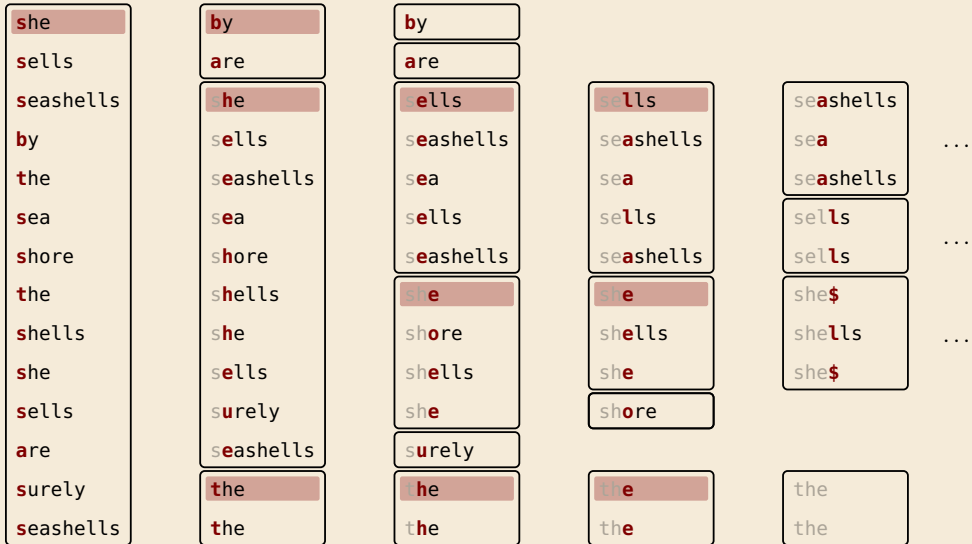
## Fat-pivot radix quicksort – Example



## Fat-pivot radix quicksort – Example



# Fat-pivot radix quicksort – Example



# Fat-pivot radix quicksort

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

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

# Fat-pivot radix quicksort

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

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

↪ 3 segments: smaller, equal, or larger than  $d$ 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 in expectation

👍 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 pivots





# 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 in expectation

👍 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 pivots



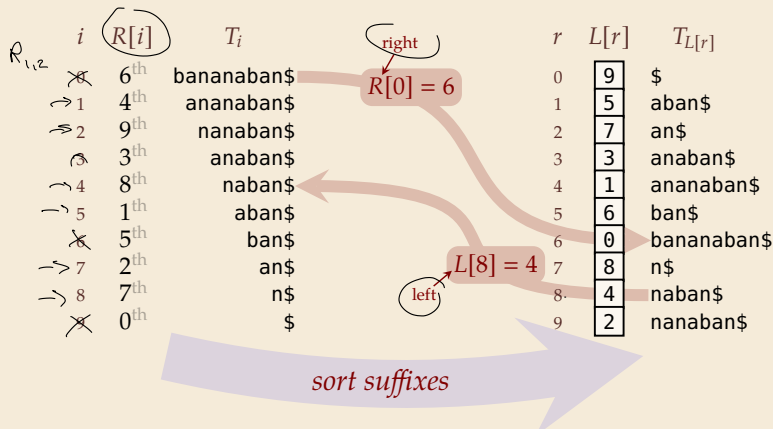
*but we can do  $O(n)$  time worst case!*

## 6.6 Linear-Time Suffix Sorting

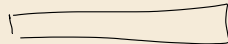
# Inverse suffix array: going left & right

► to understand the fastest algorithm, it is helpful to define the *inverse suffix array*:

- $R[i] = r \iff \boxed{L[r] = i} \quad L = \text{leaf array}$   
 $\iff$  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



# Linear-time suffix sorting



## DC3 / Skew algorithm

1. Compute rank array  $R_{1,2}$  for suffixes  $T_i$  starting at  $i \not\equiv 0 \pmod{3}$  *not a multiple of 3* recursively.  $\frac{2}{3}n$
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

1. Compute rank array  $R_{1,2}$  for suffixes  $T_i$  starting at  $i \not\equiv 0 \pmod{3}$  *not a multiple of 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 
$$n + \frac{2}{3}n + \left(\frac{2}{3}\right)^2 n + \left(\frac{2}{3}\right)^3 n + \dots \leq n \cdot \sum_{i \geq 0} \left(\frac{2}{3}\right)^i = \underline{3n} = \Theta(n)$$
$$\Theta(\quad)$$

# Linear-time suffix sorting

## DC3 / Skew algorithm

1. Compute rank array  $R_{1,2}$  for suffixes  $T_i$  starting at  $i \not\equiv 0 \pmod{3}$  *not a multiple of 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:**  $\underline{L}$  can easily be computed from  $\underline{R}$  in one pass, and vice versa.

$\rightsquigarrow$  Can use whichever is more convenient.

## DC3 / Skew algorithm – Step 2: Inducing ranks

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

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

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

$$R_{1,2}[3] = \text{undef.}$$

## DC3 / Skew algorithm – Step 2: Inducing ranks

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

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

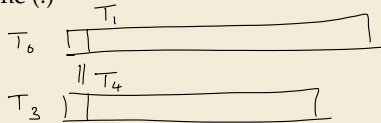
- **Task:** sort the suffixes  $T_0, T_3, T_6, T_9, \dots$  in linear time (!)

- 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

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

(append 3 \$ markers)

Step 1

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

$T_1$	annahbansbananasman\$\$\$	$R_{1,2}[22] = 0$	$T_{22}$	\$
$T_2$	nnaahbansbananasman\$\$\$	$R_{1,2}[20] = 1$	$T_{20}$	\$\$\$
$T_4$	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\$\$\$	$R_{1,2}[17] = 9$	$T_{17}$	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\$\$\$	$R_{1,2}[2] = 12$	$T_2$	nnaahbansbananasman\$\$\$
$T_{20}$	\$\$\$	$R_{1,2}[8] = 13$	$T_8$	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)

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

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

$T_0$  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\$\$\$  
 $T_2$  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)

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

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

$T_0$  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\$\$\$  
 $T_2$  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)

*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  $\rightsquigarrow R_0[9] = 7$

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

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

$T_0$  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\$\$\$  
 $T_2$  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\$\$\$  
 $T_{20}$  \$\$\$  
 $T_{22}$  \$

$R_{1,2}[22] = 0$   
 $R_{1,2}[20] = 1$   
 $R_{1,2}[4] = 2$   
 $R_{1,2}[11] = 3$   
 $R_{1,2}[13] = 4$   
 $R_{1,2}[1] = 5$   
 $R_{1,2}[7] = 6$   
 $R_{1,2}[10] = 7$   
 $R_{1,2}[5] = 8$   
 $R_{1,2}[17] = 9$   
 $R_{1,2}[19] = 10$   
 $R_{1,2}[14] = 11$   
 $R_{1,2}[2] = 12$   
 $R_{1,2}[8] = 13$   
 $R_{1,2}[16] = 14$

$T_{22}$  \$  
 $T_{20}$  \$\$\$  
 $T_4$  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\$\$\$

$R_{1,2}$  (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  $\rightsquigarrow R_0[9] = 7$

$R_0$

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

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

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

$T_0$  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\$\$\$	$R_{1,2}[22] = 0$	$T_{22}$	\$
$T_2$	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\$\$\$	$R_{1,2}[17] = 9$	$T_{17}$	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\$\$\$	$R_{1,2}[2] = 12$	$T_2$	nahbansbananasman\$\$\$
$T_{20}$	\$\$\$	$R_{1,2}[8] = 13$	$T_8$	nsbananasman\$\$\$
$T_{22}$	\$	$R_{1,2}[16] = 14$	$T_{16}$	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

$R_0$

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

$R_{1,2}$

$T_{22}$	\$
$T_{20}$	\$\$\$
$T_4$	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\$\$\$

### ► 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\$\$\$  
 $T_9$  sbanasman\$\$\$

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

# 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\$\$\$

$T_{22}$  \$  
 $T_{20}$  \$\$\$  
 $T_1$  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\$\$\$

Compare  $T_{15}$  to  $T_{11}$

Idea: try same trick as before

$T_{15} = \text{asman}$$$  
 $\quad = \underline{a}\text{sm}an$$$  
 $\quad = \underline{a}T_{16}$   
 $T_{11} = \text{ananasman}$$$  
 $\quad = \text{ananasman}$$$  
 $\quad = aT_{12}$$$$$

## ► 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

```
T21 $$  
T18 an$$$  
T15 asman$$$  
T16 bansbananasman$$$  
T0 hannahbansbananasman$$$  
T3 nahbansbananasman$$$  
T12 nanasman$$$  
T9 sbanasman$$$
```

```
T22 $  
T20 $$$  
T1 ahbansbananasman$$$  
T11 ananasman$$$  
T13 anasman$$$  
T1 annahbansbananasman$$$  
T7 ansbananasman$$$  
T10 bananasman$$$  
T5 hbansbananasman$$$  
T17 man$$$  
T19 n$$$  
T14 nasman$$$  
T2 nnahbansbananasman$$$  
T8 nsbananasman$$$  
T16 sman$$$
```

```
T22 $  
T21 $$  
T20 $$$  
T4 ahbansbananasman$$$  
T18 an$$$
```

Compare  $T_{15}$  to  $T_{11}$

Idea: try same trick as before

```
T15 = asman$$$  
      = asman$$$      can't compare T16  
      = aT16          and T12 either!  
  
T11 = ananasman$$$  
      = ananasman$$$  
      = aT12
```

### ► 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\$\$\$  
 $T_9$  sbanasman\$\$\$

$T_{22}$  \$  
 $T_{20}$  \$\$\$  
 $T_1$  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}$$$  
 $= \text{asman}$$$  
 $= aT_{16}$       can't compare  $T_{16}$   
    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}$$$  
 $= \text{aanasman}$$$  
 $= 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\$\$\$

$T_{22}$  \$  
 $T_{20}$  \$\$\$  
 $T_1$  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}$$$  
 $= \text{aanasman}$$$  
 $= 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\$\$\$  
 $T_9$  sbanasman\$\$\$

$T_{22}$  \$  
 $T_{20}$  \$\$\$  
 $T_1$  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}$$$  
 $= \text{aanasman}$$$  
 $= aT_{13}$$$$$

## DC3 / Skew algorithm – Fix recursive call

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

## 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!



## 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?

