13 Text Indexing – Searching entire genomes

2 February 2026

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg
version 2026-02-02 16:02

Learning Outcomes

Unit 13: ***Text Indexing***

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

Outline

13 Text Indexing

13.1 Inverted Indexes and Tries

Text indexing

- *Text indexing* (also: *offline text search*):
  - case of string matching: find $P[0..m)$ in $T[0..n)$
  - but with *fixed* text $\rightsquigarrow$ preprocess T (instead of P)
  - $\rightsquigarrow$ expect many queries P, answer them without looking at all of T
  - $\rightsquigarrow$ essentially a data structuring problem: "building an *index* of T"

Latin: "one who points out"

- application areas
  - web search engines
  - online dictionaries
  - online encyclopedia
  - DNA/RNA data bases
  - . . . searching in any collection of text documents (that grows only moderately)

Inverted indexes

≈ same as "indices"

- original indexes in books: list of (key) words $\mapsto$ page numbers where they occur
- assumption: searches are only for **whole** (key) **words**

$\rightsquigarrow$ often reasonable for natural language text

Inverted indexes

≈ same as "indices"

- original indexes in books: list of (key) words $\mapsto$ page numbers where they occur
- assumption: searches are only for **whole** (key) **words**

$\rightsquigarrow$ often reasonable for natural language text

Inverted index:

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

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

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.

→ sli.do/cs566

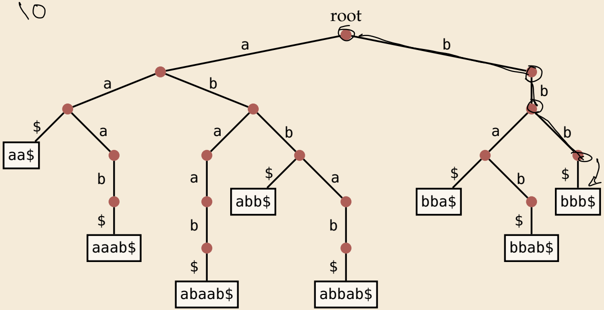
Tries

- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

- **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
  - strings of same length
  - strings have "end-of-string" marker $ (some character $\notin \Sigma$)

\0

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



Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
We now search for a query string Q with $|Q| = q$ (with $q \leq m$).
How many **nodes** in the trie are **visited** during this **query**?

A $\Theta(\log n)$

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

C $\Theta(m \cdot \log n)$

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

E $\Theta(m)$

F $\Theta(\log m)$

G $\Theta(q)$

H $\Theta(\log q)$

I $\Theta(q \cdot \log n)$

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

→ sli.do/cs566

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
We now search for a query string Q with $|Q| = q$ (with $q \le m$).
How many **nodes** in the trie are **visited** during this **query**?

A $\cancel{\Theta(\log n)}$

B $\cancel{\Theta(\log(nm))}$

C $\cancel{\Theta(m \cdot \log n)}$

D $\cancel{\Theta(m + \log n)}$

E $\cancel{\Theta(m)}$

F $\cancel{\Theta(\log m)}$

G $\Theta(q)$ ✓

H $\cancel{\Theta(\log q)}$

I $\cancel{\Theta(q \cdot \log n)}$

J $\cancel{\Theta(q + \log n)}$

→ sli.do/cs566

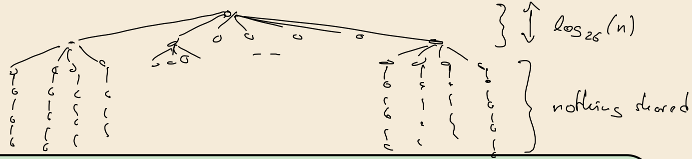
Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\mathtt{A}, \ldots, \mathtt{Z}\}$. Each stored string consists of m characters.
How many **nodes** does the trie have **in total** *in the worst case*?

- **A** $\Theta(n)$
- **B** $\Theta(n + m)$
- **C** $\Theta(n \cdot m)$
- **D** $\Theta(n \log m)$
- **E** $\Theta(m)$
- **F** $\Theta(m \log n)$

→ sli.do/cs566

Clicker Question



Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
How many **nodes** does the trie have **in total** *in the worst case*?

A $\cancel{\Theta(n)}$

B $\cancel{\Theta(n+m)}$

C $\Theta(n \cdot m)$ ✓ $=$ total # chars

D $\cancel{\Theta(n \log m)}$

E $\cancel{\Theta(m)}$

F $\cancel{\Theta(m \log n)}$

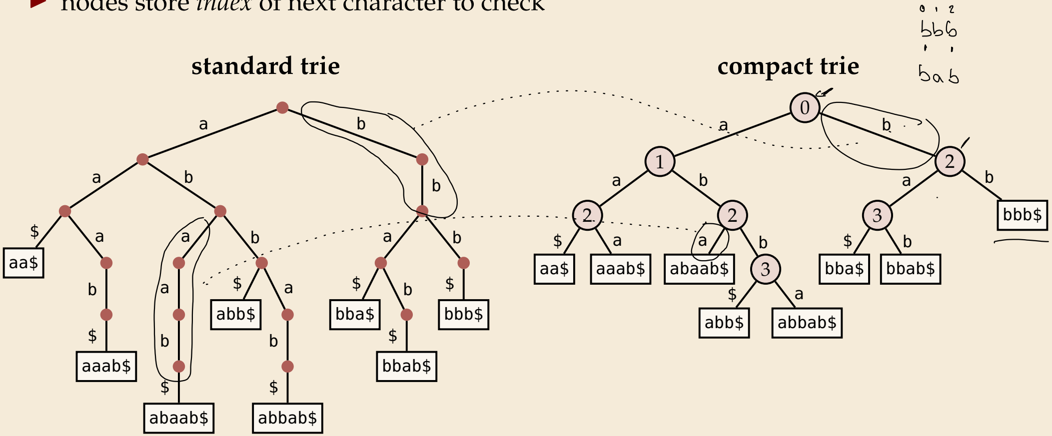
$n(m - \log n) + n$

node in trie needs array of child pointers
$\rightsquigarrow$ $\Theta(\sigma)$ space per node

→ sli.do/cs566

Compact tries

- compress paths of unary nodes into single edge
  =1 child
- nodes store *index* of next character to check



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

- all nodes ≥ 2 children ⇝ #nodes ≤ #leaves = #strings ⇝ linear space in #strings

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

    ```
    ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGC
    CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGC
    CAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCCAGCAATCCGCGCGCCGGGACAGAA
    TGCCCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
    ```

  - binary streams

    ```
    00000010101001111010111000000111110001111101111100110110100001110001001101111000001000110101 0
    0110110000110101101000000010000000011101011000001000011110101110110010001100101101110111111 1
    1100010100010110010100000011101010100110000000011011000011001111100001 01 0101011101111000011
    1010111001001010101010000011111010011000000111100110101000000010010010000010110001100011 0111
    ```

⇝ need new ideas

13.2 Suffix Trees

Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$

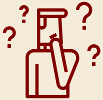
  Example: $S_1 = \underbrace{\texttt{superiorcaliforni}\texttt{alives}}$, $S_2 = \texttt{se}\underbrace{\texttt{aliver}}$

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

Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$ **Example:** $S_1 =$ `superiorcalifornialives`, $S_2 =$ `sealiver`
- Goal: find the longest substring that occurs in all k strings $\leadsto$ `alive`



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

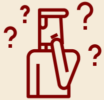
Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

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

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

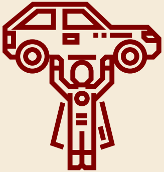
$\rightsquigarrow$ `alive`



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

Enter: ***suffix trees***

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



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

Suffix trees – Definition

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

Suffix trees – Definition

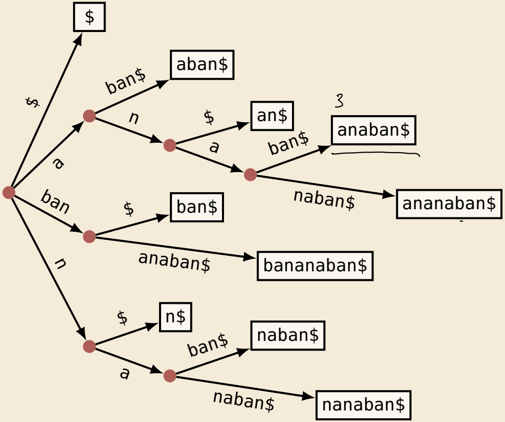
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ $=$ compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

Example:

T = bananaban$

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

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



Suffix trees – Definition

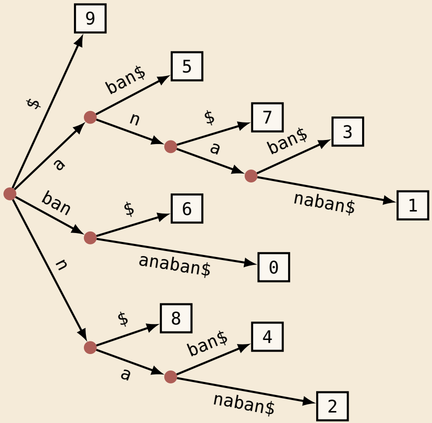
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

Example:

T = bananaban$

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

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



Suffix trees – Definition

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

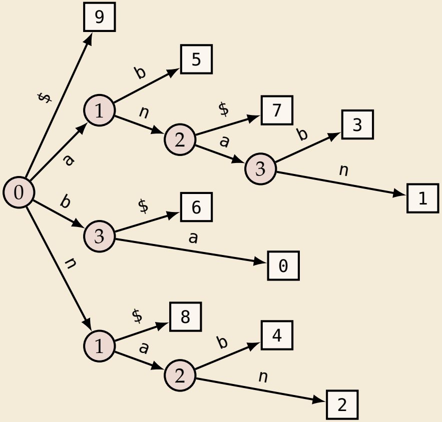
Example:

$T =$ bananaban$

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

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

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



Suffix trees – Construction

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

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 → EAA Book
- 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?

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.

→ sli.do/cs566

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.~~ $n+1$

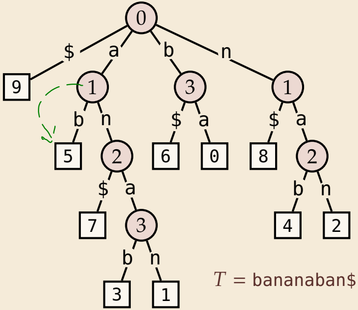
→ sli.do/cs566

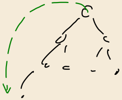
13.3 Applications

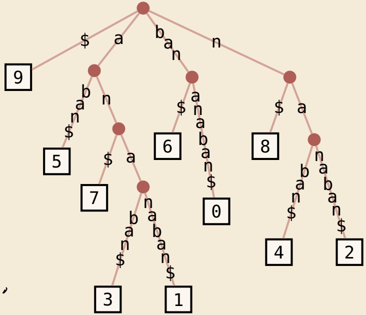
Applications of suffix trees

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

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

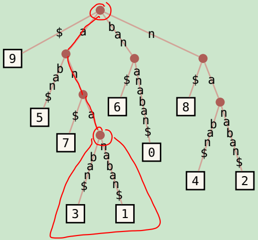






- Moreover: assume internal nodes store pointer to *leftmost leaf in subtree*.
- Notation: $T_i = T[i..n]$ (including $)

Clicker Question



What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = ana?

Check all that apply to this example.

A Nothing.

B P occurs in T.

C P does not occur in T.

D P occurs once in T.

E P occurs twice in T.

F P starts at index 0.

G P starts at index 1.

H P starts at index 2.

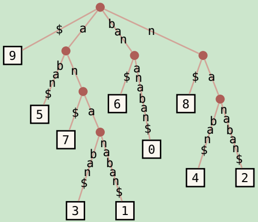
I P starts at index 3.

J P starts at index 4.

K P starts at index 7.

→ sli.do/cs566

Clicker Question



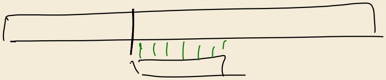
What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = ana?

Check all that apply to this example.

- **A** ~~Nothing.~~
- **B** P occurs in T.
- **C** ~~P does not occur in T.~~
- **D** ~~P occurs once in T.~~
- **E** P occurs twice in T.
- **F** ~~P starts at index 0.~~
- **G** P starts at index 1.
- **H** ~~P starts at index 2.~~
- **I** P starts at index 3.
- **J** ~~P starts at index 4.~~
- **K** ~~P starts at index 7.~~

→ sli.do/cs566

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

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

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

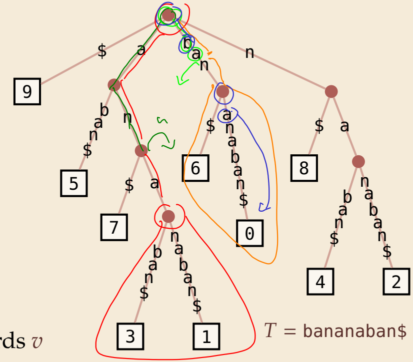
1. **we get stuck**
at internal node (no node with next character of P)
or *inside edge* (mismatch of next characters)
⇝ P does not occur in T

2. **we run out of pattern**
reach end of P at internal node v or inside edge towards v
⇝ P occurs at all leaves in subtree of v

3. **we run out of tree**
reach a leaf ℓ with part of P left ⇝ compare P to ℓ.

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

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



T = `bananaban$`

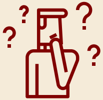
Examples:

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

Application 2: Longest repeated substring

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

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

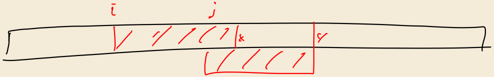
How can we efficiently check *all possible substrings?*

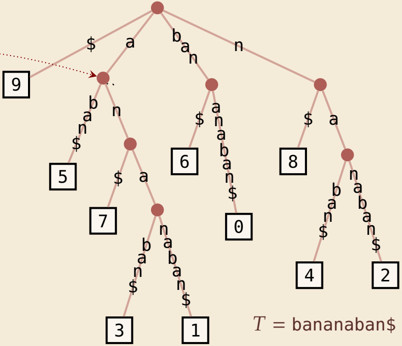
Repeated substrings = shared paths in *suffix tree*

- T_5 = aban\$ and T_7 = an\$ have *longest common prefix* ‘a’

$\rightsquigarrow$ $\exists$ internal node with path label ‘a’

here single edge, can be longer path





T = bananaban\$

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*

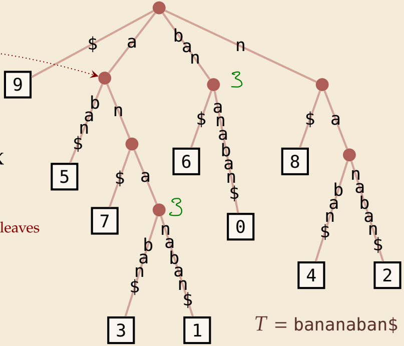
- T_5 = aban\$ and T_7 = an\$ have *longest common prefix* 'a'

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

here single edge, can be longer path

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

actually: adjacent leaves



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*

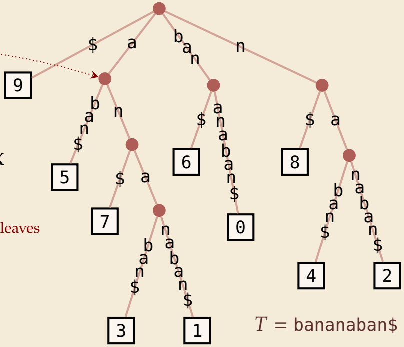
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

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

here single edge, can be longer path

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

actually: adjacent leaves



- 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

13.4 Generalized Suffix Trees

Generalized suffix trees

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

Generalized suffix trees

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

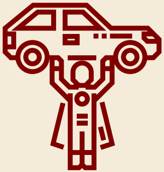
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts

Generalized suffix trees

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

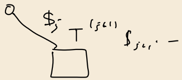
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts

Enter: ***generalized suffix tree***



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

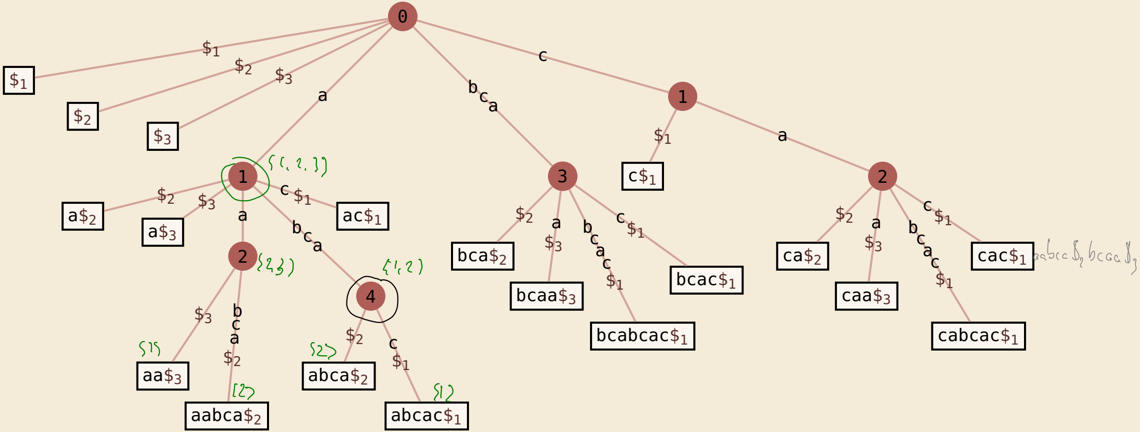
$\rightsquigarrow$ $\$_j$-edges always leads to leaves $\rightsquigarrow$ $\exists$ leaf $\underline{(j, i)}$ for each suffix $T_i^{(j)} = T^{(j)}[i..n_j]$



Generalized Suffix Tree – Example

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

$T = bcabcac\$_1 aabca\$_2 bcaa\$_3$



Application 3: Longest common substring

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

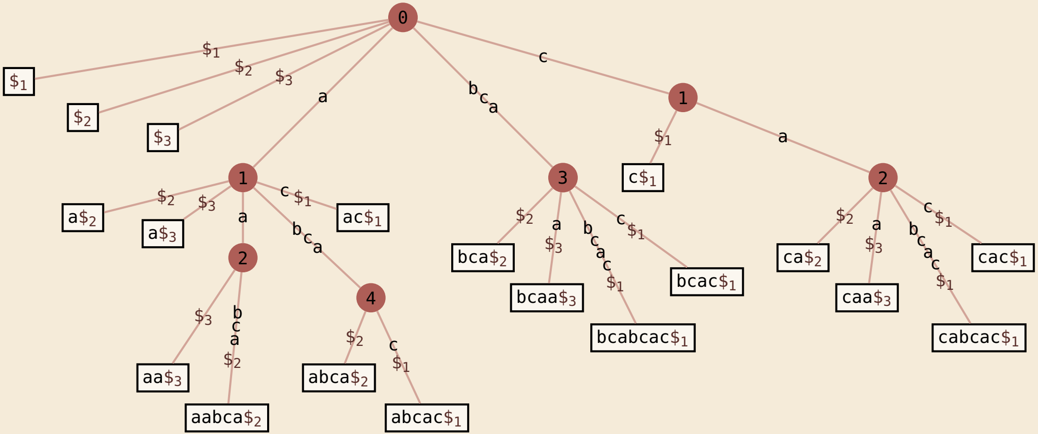
  1. Compute generalized suffix tree $\mathcal{T}$.
  2. Store with each node the *subset of strings* that contain its path label:
     2.1. Traverse $\mathcal{T}$ bottom-up.
     2.2. For a leaf (j, i), the subset is $\{j\}$.
     2.3. For an internal node, the subset is the union of its children.
  3. In top-down traversal, compute *string depths* of nodes. (as above)
  4. Report deepest node (by string depth) whose subset is $\{1, \ldots, k\}$.

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

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

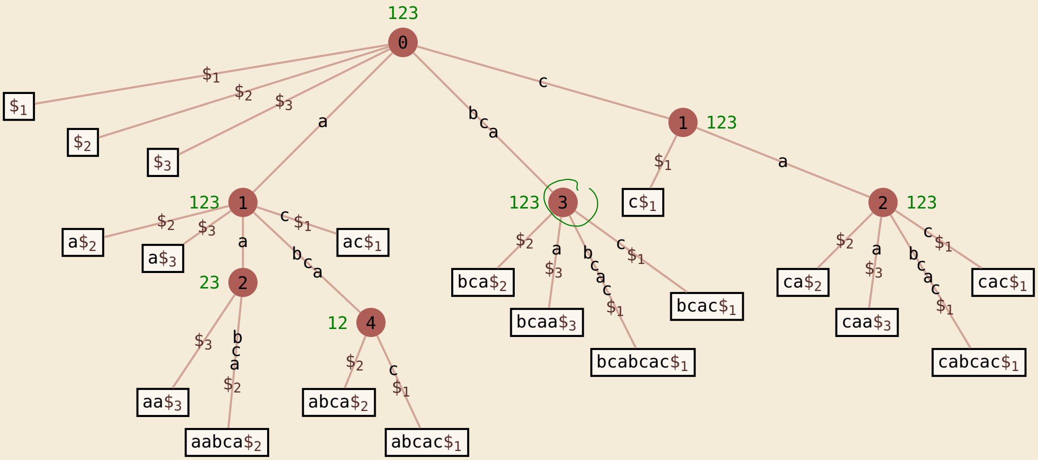
Longest common substring – Example

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



Longest common substring – Example

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



Further applications

Many more applications, in particular for problems on biological sequences

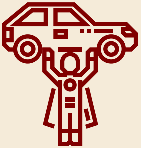
20+ described in Gusfield, *Algorithms on strings, trees, and sequences* (1999)

- key ingredient: longest common extensions

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

Suffix trees – Discussion

- Suffix trees were a threshold invention

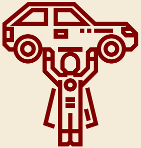


👍 linear time and space

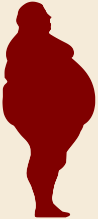
👍 suddenly many questions efficiently solvable in theory

Suffix trees – Discussion

- Suffix trees were a threshold invention



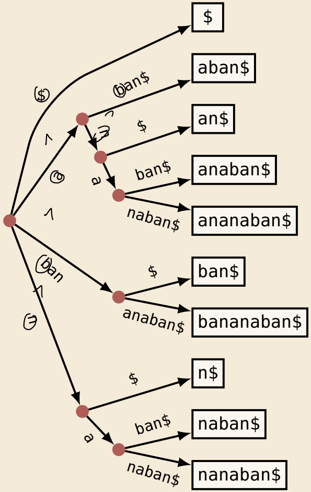
- linear time and space
- suddenly many questions efficiently solvable in theory



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

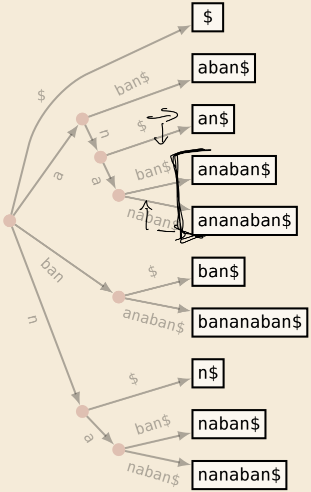
13.5 Suffix Arrays

Putting suffix trees on a diet



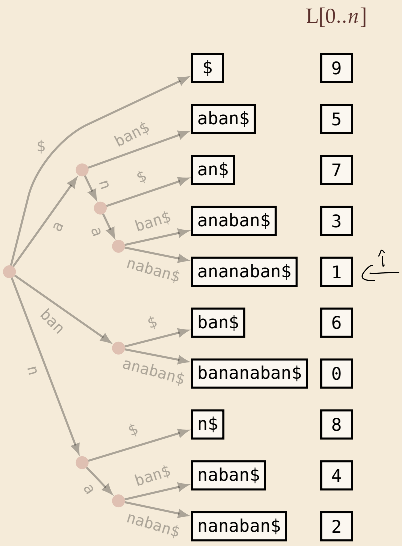
- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*

Putting suffix trees on a diet



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

Putting suffix trees on a diet



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

- Idea: only store list of leaves $L[0..n]$
- Sufficient to do efficient string matching!
  1. Use binary search for pattern P
  2. check if P is prefix of suffix after position found
- **Example:** P = ana $

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

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

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

Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

- **A** L can be very easily computed from B and T
- **B** B can be very easily computed from L and T
- **C** Both A and B
- **D** Neither A nor B

→ sli.do/cs566

Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

- **A** ~~L can be very easily computed from B and T~~
- **B** B can be very easily computed from L and T $\checkmark$
- **C** ~~Both A and B~~
- **D** ~~Neither A nor B~~

→ sli.do/cs566

Digression: Recall BWT

Burrows-Wheeler Transform

1. Take all cyclic shifts of S
2. Sort cyclic shifts
3. Extract last column

$S =$ `alf␣eats␣alfalfa$`

$B =$ `asff$f␣e␣lllaaata`

```
alf␣eats␣alfalfa$
lf␣eats␣alfalfa$a
f␣eats␣alfalfa$al
␣eats␣alfalfa$alf
eats␣alfalfa$alf␣
ats␣alfalfa$alf␣e
ts␣alfalfa$alf␣ea
s␣alfalfa$alf␣eat
␣alfalfa$alf␣eats
alfalfa$alf␣eats␣
lfalfa$alf␣eats␣a
falfa$alf␣eats␣al
alfa$alf␣eats␣alf
lfa$alf␣eats␣alfa
fa$alf␣eats␣alfal
a$alf␣eats␣alfalf
$alf␣eats␣alfalfa
```

sort

BWT

```
$alf␣eats␣alfalfa
␣alfalfa$alf␣eats
␣eats␣alfalfa$alf
a$alf␣eats␣alfalf
alf␣eats␣alfalfa$
alfa$alf␣eats␣alf
alfalfa$alf␣eats␣
ats␣alfalfa$alf␣e
eats␣alfalfa$alf␣
f␣eats␣alfalfa$al
fa$alf␣eats␣alfal
falfa$alf␣eats␣al
lf␣eats␣alfalfa$a
lfa$alf␣eats␣alfa
lfalfa$alf␣eats␣a
s␣alfalfa$alf␣eat
ts␣alfalfa$alf␣ea
```

Digression: Computing the BWT

How can we compute the BWT of a text efficiently?

Digression: Computing the BWT

How can we compute the BWT of a text efficiently?

- cyclic shifts S $\widehat{=}$ suffixes of S
  - comparing cyclic shifts stops at first \$
  - for comparisons, anything after \$ irrelevant!
- BWT is essentially suffix sorting!
  - $B[i] = S[L[i] - 1]$
  - where $L[i] = 0$, $B[i] = \$$

$\rightsquigarrow$ Can compute B in $O(n)$ time from L

	r		$L[r]$
alf␣eats␣alfalfa$	0	$alf␣eats␣alfalf**a**	16
lf␣eats␣alfalfa$a	1	␣alfalfa$alf␣eat**s**	8
f␣eats␣alfalfa$al	2	␣eats␣alfalfa$al**f**	3
␣eats␣alfalfa$alf	3	a$alf␣eats␣alfal**f**	15
eats␣alfalfa$alf␣	4	alf␣eats␣alfalfa**$**	0
ats␣alfalfa$alf␣e	5	alfa$alf␣eats␣al**f**	12
ts␣alfalfa$alf␣ea	6	alfalfa$alf␣eats**␣**	9
s␣alfalfa$alf␣eat	7	ats␣alfalfa$alf␣**e**	5
␣alfalfa$alf␣eats	8	eats␣alfalfa$alf**␣**	4
alfalfa$alf␣eats␣	9	f␣eats␣alfalfa$a**l**	2
lfalfa$alf␣eats␣a	10	fa$alf␣eats␣alfa**l**	14
falfa$alf␣eats␣al	11	falfa$alf␣eats␣a**l**	11
alfa$alf␣eats␣alf	12	lf␣eats␣alfalfa$**a**	1
lfa$alf␣eats␣alfa	13	lfa$alf␣eats␣alf**a**	13
fa$alf␣eats␣alfal	14	lfalfa$alf␣eats␣**a**	10
a$alf␣eats␣alfalf	15	s␣alfalfa$alf␣ea**t**	7
$alf␣eats␣alfalfa	16	ts␣alfalfa$alf␣e**a**	6

Suffix arrays – Construction

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

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

Suffix arrays – Construction

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

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

Excursion: String sorting

- when sorting strings, “blind” comparisons can cost $\Theta(n)$ character comparisons
- happens iff strings share long prefix!

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

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 ∉ exam
- Option 2: Fat-pivot radix quicksort
  - **partition** based on d**th character** only (initially $d = 0$)
  - ⇝ 3 segments: smaller, equal, or larger than dth 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	**b**y
sells	**a**re
seashells	s**h**e
by	s**e**lls
the	s**e**ashells
sea	s**e**a
shore	s**h**ore
the	s**h**ells
shells	s**h**e
she	s**e**lls
sells	s**u**rely
are	s**e**ashells
surely	**t**he
seashells	**t**he

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	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	
seashells	the	

Fat-pivot radix quicksort – Example

she	by	are
sells	are	by
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

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

she	by	are	
sells	are	by	
seashells	she	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	she	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

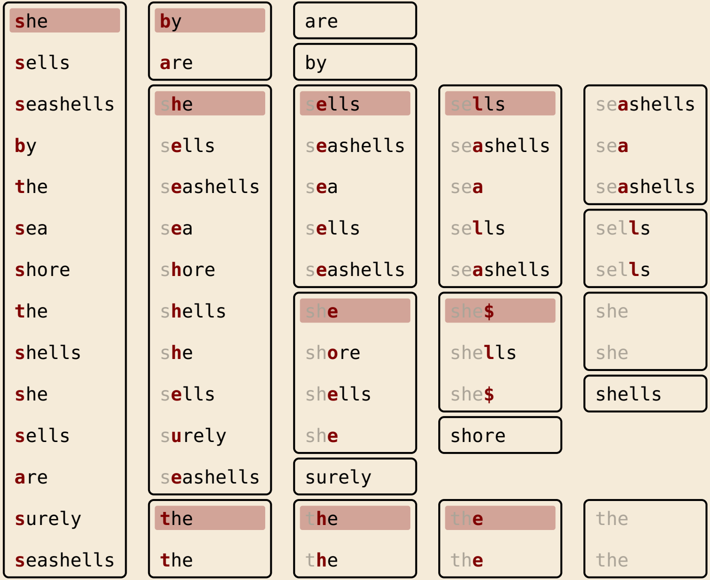
Fat-pivot radix quicksort – Example

she	by	are		
sells	are	by		
seashells	she	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$	
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

she	by	are		
sells	are	by		
seashells	she	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	
seashells	the	the	the	

Fat-pivot radix quicksort – Example



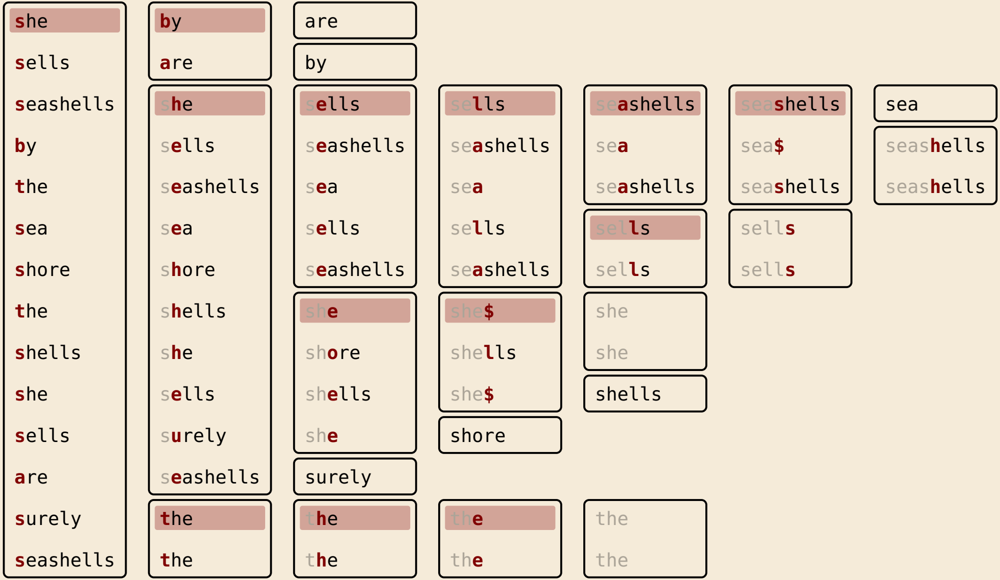
Fat-pivot radix quicksort – Example

she	by	are			
sells	are	by			
seashells	she	sells	sells	seashells	seashells
by	sells	seashells	seashells	sea	sea$
the	seashells	sea	sea	seashells	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	

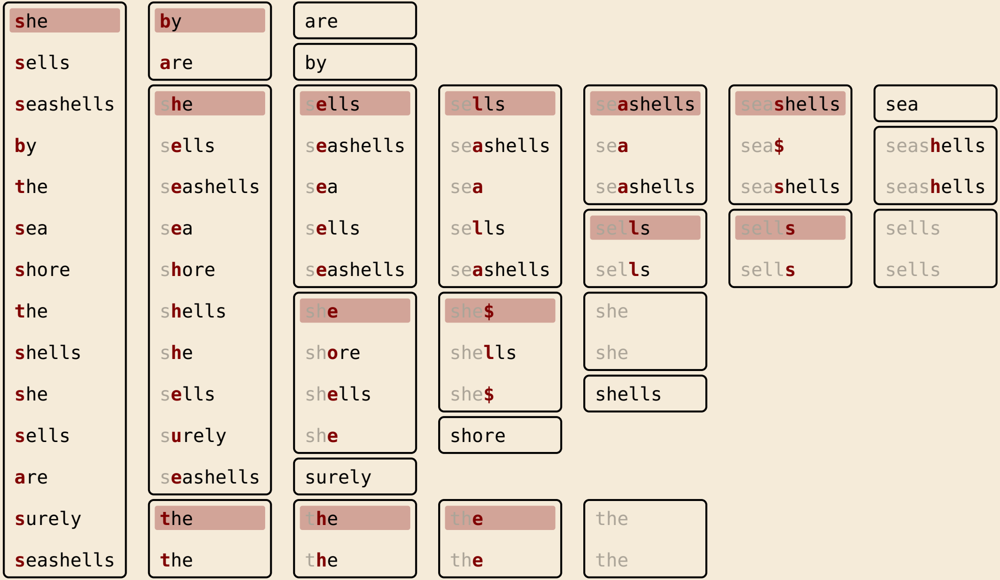
Fat-pivot radix quicksort – Example

she	by	are			
sells	are	by			
seashells	she	sells	sells	seashells	seashells
by	sells	seashells	seashells	sea	sea$
the	seashells	sea	sea	seashells	seashells
sea	sea	sells	sells	sells	sells
shore	shore	seashells	seashells	sells	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

she	by	are					
sells	are	by					
seashells	she	sells	sells	seashells	seashells	sea	
by	sells	seashells	seashells	sea	sea$	seashells	…
the	seashells	sea	sea	seashells	seashells	seashells	
sea	sea	sells	sells	sells	sells	sells	
shore	shore	seashells	seashells	sells	sells	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 – Analysis

Separately analyze character comparisons **by outcome**

1. ***"Decisive Comparisons:"*** character comparisons with outcome "$<$" or "$>$"
   - can have at most one in any **string comparison** (afterwards done!)
   - $\rightsquigarrow$ Same number of decisive comparisons as in standard quicksort (just delayed)
   - $\rightsquigarrow$ expected $\sim 2\ln(2) \cdot n \lg n \;\approx\; 1.39 n \lg n$ decisive comparisons
   - duplicates only reduce # comparisons

Fat-pivot radix quicksort – Analysis

Separately analyze character comparisons **by outcome**

1. *"Decisive Comparisons:"* character comparisons with outcome "<" or ">"
   - can have at most one in any **string comparison** (afterwards done!)
   - $\rightsquigarrow$ Same number of decisive comparisons as in standard quicksort (just delayed)
   - $\rightsquigarrow$ expected $\sim 2\ln(2) \cdot n \lg n \;\approx\; 1.39 n \lg n$ decisive comparisons
   - duplicates only reduce # comparisons
2. *LCP comparisons:* character comparisons that return "="
   - must be all remaining character comparisons
   - every such comparison contributes to common prefix between strings, never compare same characters again
   - every string sort must discover longest common prefixes in sorted order
   - $\rightsquigarrow$ #LCP comparisons $=$ #comparisons when inserting all strings into a **trie**

Fat-pivot radix quicksort – Discussion

- simple to code
- efficient for sorting many lists of strings
  - fat-pivot radix quicksort finds suffix array in $O(n \log n)$ expected time
    random string
- worst case remains $\Omega(n^2)$, i. e., $T = \mathtt{a}^n$

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

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

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

13.6 Linear-Time Suffix Sorting: Inducing Order

Inverse suffix array: going left & right

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

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

right
$R[0] = 6$

$L[8] = 4$
left

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

sort suffixes

Clicker Question

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

- **A** L lists the leaves of $\mathcal{T}$ in left-to-right order.
- **B** R lists the leaves of $\mathcal{T}$ in right-to-left order.
- **C** R lists starting indices of suffixes in lexciographic order.
- **D** L lists starting indices of suffixes in lexciographic order.
- **E** $L[r] = i$ iff $R[i] = r$
- **F** L stands for leaf
- **G** L stands for left
- **H** R stands for rank
- **I** R stands for right

→ sli.do/cs566

Clicker Question

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

- **A** L lists the leaves of $\mathcal{T}$ in left-to-right order. ✓
- **B** ~~R lists the leaves of $\mathcal{T}$ in right-to-left order.~~
- **C** ~~R lists starting indices of suffixes in lexciographic order.~~
- **D** L lists starting indices of suffixes in lexciographic order. ✓
- **E** $L[r] = i$ iff $R[i] = r$ ✓
- **F** L stands for leaf ✓
- **G** L stands for left ✓
- **H** R stands for rank ✓
- **I** R stands for right ✓

→ sli.do/cs566

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 $i \not\equiv 0 \pmod 3$ $\underbrace{\textit{recursively}}$.

2. Induce rank array R_3 for suffixes $T_0, T_3, T_6, T_9, \ldots$ from $R_{1,2}$.

3. Merge $R_{1,2}$ and R_0 using $R_{1,2}$.
   $\rightsquigarrow$ rank array R for entire input

Linear-time suffix sorting

DC3 / Skew algorithm

not a multiple of 3

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

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

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

Linear-time suffix sorting

DC3 / Skew algorithm

not a multiple of 3

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

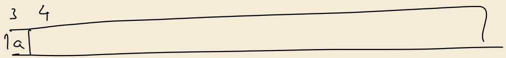
- We will show that steps 2. and 3. take $\Theta(n)$ time

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

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

DC3 / Skew algorithm – Step 2: Inducing ranks

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



DC3 / Skew algorithm – Step 2: Inducing ranks

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

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

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

DC3 / Skew algorithm – Inducing ranks example

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

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

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

$R_{1,2}$ (known)

DC3 / Skew algorithm – Inducing ranks example

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

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

sman$$$ = T_{16}

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

$R_{1,2}$ (known)

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

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

DC3 / Skew algorithm – Inducing ranks example

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

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

`sman$$$` = T_{16}

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

$R_{1,2}$ (known)

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

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

radix sort

T_{21}	`$00`	$\rightsquigarrow$	$R_0[21] = 0$
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 = hannahbansbananasman$$$ (append 3 $ markers)

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

sman$$$ = T_{16}

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

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

$R_{1,2}$ (known)

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

radix sort

R_0

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 = `hannahbansbananasman$$$` (append 3 \$ markers)

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

`sman$$$` = T_{16}

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

$R_{1,2}$ (known)

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

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

radix sort

T_{21}	`$00`	$\rightsquigarrow$	$R_0[21] = 0$
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:
    $T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \ldots$
  - sorted 0-list:
    $T_0, T_3, T_6, T_9, \ldots$
- Task: Merge them!
  - use standard merging method from Mergesort
  - but speed up comparisons using $R_{1,2}$

DC3 / Skew algorithm – Step 3: Merging

T_{21}	`$$`
T_{18}	`an$$$`
T_{15}	`asman$$$`
T_{6}	`bansbananasman$$$`
T_{0}	`hannahbansbananasman$$$`
T_{3}	`nahbansbananasman$$$`
T_{12}	`nanasman$$$`
T_{9}	`sbananasman$$$`

T_{22}	`$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
T_{11}	`ananasman$$$`
T_{13}	`anasman$$$`
T_{1}	`annahbansbananasman$$$`
T_{7}	`ansbananasman$$$`
T_{10}	`bananasman$$$`
T_{5}	`hbansbananasman$$$`
T_{17}	`man$$$`
T_{19}	`n$$$`
T_{14}	`nasman$$$`
T_{2}	`nnahbansbananasman$$$`
T_{8}	`nsbananasman$$$`
T_{16}	`sman$$$`

T_{22}	`$`
T_{21}	`$$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
T_{18}	`an$$$`

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

DC3 / Skew algorithm – Step 3: Merging

T_{21}	`$$`
T_{18}	`an$$$`
T_{15}	`asman$$$`
T_{6}	`bansbananasman$$$`
T_{0}	`hannahbansbananasman$$$`
T_{3}	`nahbansbananasman$$$`
T_{12}	`nanasman$$$`
T_{9}	`sbananasman$$$`

T_{22}	`$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
T_{11}	`ananasman$$$`
T_{13}	`anasman$$$`
T_{1}	`annahbansbananasman$$$`
T_{7}	`ansbananasman$$$`
T_{10}	`bananasman$$$`
T_{5}	`hbansbananasman$$$`
T_{17}	`man$$$`
T_{19}	`n$$$`
T_{14}	`nasman$$$`
T_{2}	`nnahbansbananasman$$$`
T_{8}	`nsbananasman$$$`
T_{16}	`sman$$$`

T_{22}	`$`
T_{21}	`$$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
T_{18}	`an$$$`

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

Compare T_{15} to T_{11}

Idea: try same trick as before

$$T_{15} = \texttt{asman\$\$\$} = \texttt{asman\$\$\$} = \texttt{a}T_{16}$$

$$T_{11} = \texttt{ananasman\$\$\$} = \texttt{ananasman\$\$\$} = \texttt{a}T_{12}$$

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$$$`
T_{16}	`sman$$$`

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

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} = `asman$$$`
= `asman$$$`
= `a`T_{16}

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

T_{11} = `ananasman$$$`
= `ananasman$$$`
= `a`T_{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:
    $T_1, T_2, T_4, T_5, T_7, T_8, T_{10}, T_{11}, \ldots$
  - sorted 0-list:
    $T_0, T_3, T_6, T_9, \ldots$
- Task: Merge them!
  - use standard merging method from Mergesort
  - but speed up comparisons using $R_{1,2}$

Compare T_{15} to T_{11}

Idea: try same trick as before

T_{15} = `asman$$$` = `asman$$$` = `a`T_{16}

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

T_{11} = `ananasman$$$` = `ananasman$$$` = `a`T_{12}

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

T_{16} = `sman$$$` = `sman$$$` = `s`T_{17}

T_{12} = `nanasman$$$` = `aanasman$$$` = `a`T_{13}

DC3 / Skew algorithm – Step 3: Merging

T_{21}	`$$`
T_{18}	`an$$$`
$\boxed{T_{15}}$	`asman$$$`
T_{6}	`bansbananasman$$$`
T_{0}	`hannahbansbananasman$$$`
T_{3}	`nahbansbananasman$$$`
T_{12}	`nanasman$$$`
T_{9}	`sbananasman$$$`

T_{22}	`$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
$\boxed{T_{11}}$	`ananasman$$$`
T_{13}	`anasman$$$`
T_{1}	`annahbansbananasman$$$`
T_{7}	`ansbananasman$$$`
T_{10}	`bananasman$$$`
T_{5}	`hbansbananasman$$$`
T_{17}	`man$$$`
T_{19}	`n$$$`
T_{14}	`nasman$$$`
T_{2}	`nnahbansbananasman$$$`
T_{8}	`nsbananasman$$$`
T_{16}	`sman$$$`

T_{22}	`$`
T_{21}	`$$`
T_{20}	`$$$`
T_{4}	`ahbansbananasman$$$`
T_{18}	`an$$$`

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

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} =$ `asman$$$`
$\quad =$ `asman$$$`
$\quad =$ `a`T_{16}

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

$T_{11} =$ `ananasman$$$`
$\quad =$ `ananasman$$$`
$\quad =$ `a`T_{12}

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

$T_{16} =$ `sman$$$`
$\quad =$ `sman$$$`
$\quad =$ `s`T_{17}

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

$T_{12} =$ `nanasman$$$`
$\quad =$ `aanasman$$$`
$\quad =$ `a`T_{13} ·

DC3 / Skew algorithm – Step 3: Merging

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

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

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

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

$\rightsquigarrow$ $O(n)$ time for merge

Compare T_{15} to T_{11}

Idea: try same trick as before

$T_{15} =$ `asman$$$`
$=$ `asman$$$`
$=$ `a`T_{16}

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

$T_{11} =$ `ananasman$$$`
$=$ `ananasman$$$`
$=$ `a`T_{12}

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

$T_{16} =$ `sman$$$`
$=$ `sman$$$`
$=$ `s`T_{17}

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

$T_{12} =$ `nanasman$$$`
$=$ `aanasman$$$`
$=$ `a`T_{13}

continue 15:16

13.7 Linear-Time Suffix Sorting: The DC3 Algorithm

DC3 / Skew algorithm – Fix recursive call

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

DC3 / Skew algorithm – Fix recursive call

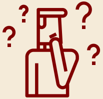
- both step 2. and 3. doable in $O(n)$ time!

- 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!

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



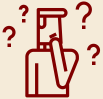
How can we make T' "skip" some suffixes?

DC3 / Skew algorithm – Fix 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!

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



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

redefine alphabet to be *triples of characters* $\boxed{\texttt{abc}}$

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

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

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

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

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

$\rightsquigarrow$ suffixes of $T^{\square}$ $\iff$ $\underline{T_0, T_3, T_6, T_9, \ldots}$

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

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

DC3 / Skew algorithm – Fix alphabet explosion

- Still does not quite work!

DC3 / Skew algorithm – Fix alphabet explosion

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

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

$\leadsto$ Before recursion:

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

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

DC3 / Skew algorithm – Step 3. Example

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

- T = `hannahbansbananasman$`

DC3 / Skew algorithm – Step 3. Example

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

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

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

DC3 / Skew algorithm – Step 3. Example

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

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

DC3 / Skew algorithm – Step 3. Example

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

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

  T' = `ann` `ahb` `ans` `ban` `ana` `sma` `n$$` `$$$` `nna` `hba` `nsb` `ana` `nas` `man` `$$$`

- Occurring triples:

  `ann` `ahb` `ans` `ban` `ana` `sma` `n$$` `$$$` `nna` `hba` `nsb` `nas` `man`

- Sorted triples with ranks:

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

DC3 / Skew algorithm – Step 3. Example

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

- T = `hannahbansbananasman$` T_2 = `nnahbansbananasman$`
  T' = `ann` `ahb` `ans` `ban` `ana` `sma` `n$$` `$$$` `nna` `hba` `nsb` `ana` `nas` `man` `$$$`

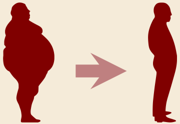
- Occurring triples:
  `ann` `ahb` `ans` `ban` `ana` `sma` `n$$` `$$$` `nna` `hba` `nsb` `nas` `man`

- Sorted triples with ranks:

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

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

Suffix array – Discussion



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

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

Outlook: Enhanced Suffix Arrays

- suffix array by itself somewhat less powerful,
  but can be augmented with the LCP array

  longest common prefix

⇝ *(Enhanced) Suffix Arrays*

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

👎 can be harder to reason about

👍 can support same algorithms as suffix trees

👍 use less space

👍 simpler linear-time construction

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