Algorithmen / Algorithms II

Peter Sanders

Exercise:

Daniel Seemaier, Tobias Heuer

Institute of Theoretical Informatics

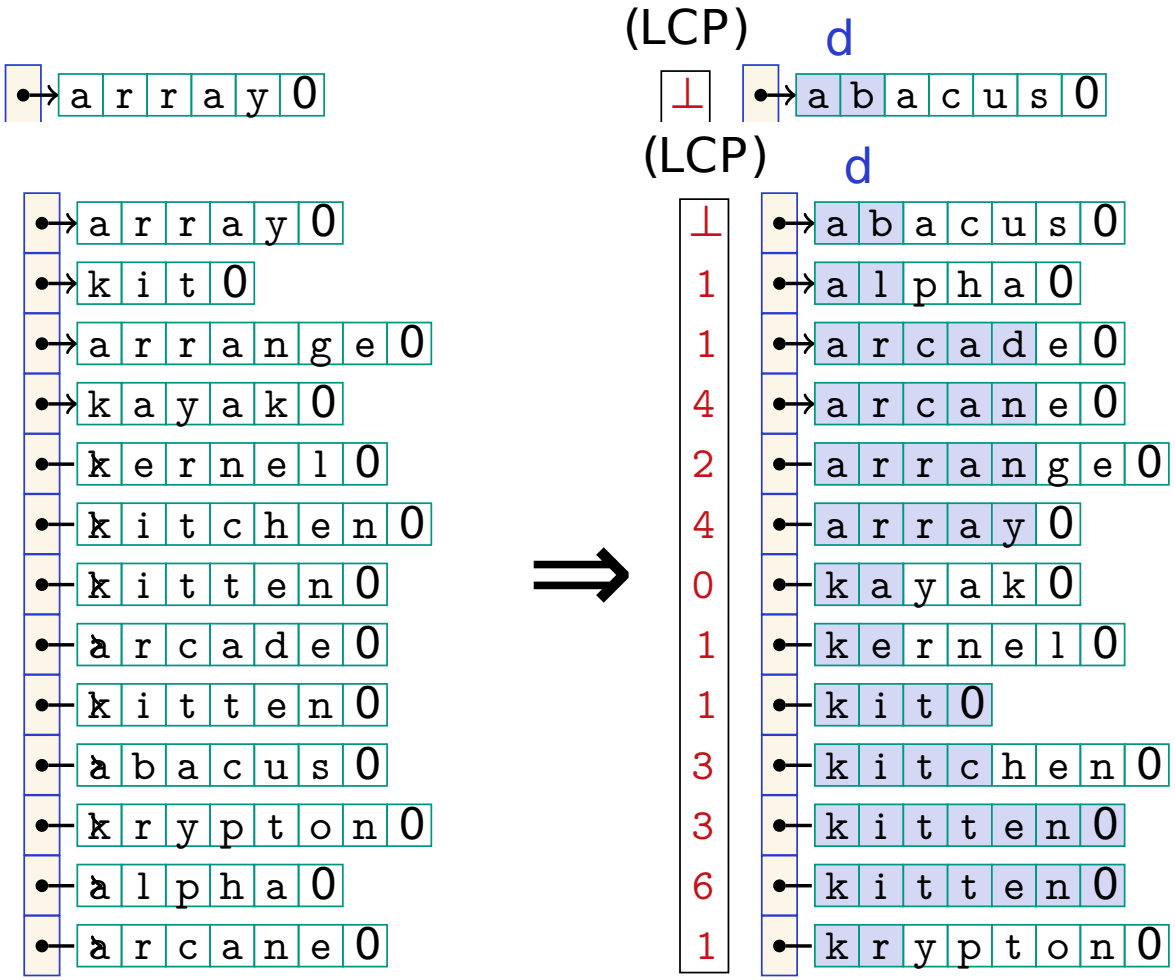
Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS20.php`

11 Stringology (String Algorithms)

- ☐ String Sorting
- ☐ Pattern Matching
  - Preprocess Pattern
  - Preprocess Text
    - Inverted Indices
    - Suffix Trees / Suffix Arrays
- ☐ Data Compression
- ☐ Pattern Matching in Compressed Indices

String Sorting



Input: n String with N characters

Sorting: Most Significant Digit Radix Sort

a r r a y
k i t
a r r a n g e
k a y a k
k e r n e l
k i t c h e n
k i t t e n
a r c a d e
k i t e
a b a c u s
k r y p t o n
a l p h a
a r c a i c

0		a		k		$\sigma-1$
0	···	6	·····	7	·····	0

Sorting: Most Significant Digit Radix Sort

a r r a y
k i t
a r r a n g e
k a y a k
k e r n e l
k i t c h e n
k i t t e n
a r c a d e
k i t e
a b a c u s
k r y p t o n
a l p h a
a r c a i c

0		a				k				$\sigma-1$
0	···	6	·····			7	·····			0

0	···	0	6	···	6	13	····	13	13

Sorting: Most Significant Digit Radix Sort

a r r a y
k i t
a r r a n g e
k a y a k
k e r n e l
k i t c h e n
k i t t e n
a r c a d e
k i t e
a b a c u s
k r y p t o n
a l p h a
a r c a i c

0		a		k		$\sigma-1$
0	···	6	·····	7	·····	0

0	···	0	6	···	6	13	····	13	13

Sorting: Most Significant Digit Radix Sort

```
a r r a y
a r r a n g e
a r c a d e
a b a c u s
a l p h a
a r c a i c
k i t
k a y a k
k e r n e l
k i t c h e n
k i t t e n
k i t e
k r y p t o n
```

0		a		k		σ−1
0	···	6	·····	7	·····	0

0	···	0	6	···	6	13	····	13	13

- Running Time: $O(d + r\sigma + n \log \sigma)$
- Variants: out-of-place and in-place

Sorting: Most Significant Digit Radix Sort

a r r a y
k i t
a r r a n g e
k a y a k
k e r n e l
k i t c h e n
k i t t e n
a r c a d e
k i t e
a b a c u s
k r y p t o n
a l p h a
a r c a i c

0		a		k		$\sigma-1$
0	···	6	·····	7	·····	0

0	···	0	6	···	6	13	····	13	13

p_1	0	···	2	·····	3	·····	0
p_2	0	···	2	·····	3	·····	0
p_3	0	···	2	·····	1	·····	0

Sorting: Most Significant Digit Radix Sort

a r r a y
k i t
a r r a n g e
k a y a k
k e r n e l
k i t c h e n
k i t t e n
a r c a d e
k i t e
a b a c u s
k r y p t o n
a l p h a
a r c a i c

	0		a			k			$\sigma-1$	
	0	···	6	·····		7	·····		0	
	0	···	0	6	···	6	13	····	13	13
p_1	0	···	2	·····		3	·····		0	
p_2	0	···	2	·····		3	·····		0	
p_3	0	···	2	·····		1	·····		0	
p_1	0	···	0	6	···	6	13	····	13	13
p_2	0	···	2	6	···	9	13	····	13	
p_3	0	···	4	6	···	12	13	····	13	

Sorting: Most Significant Digit Radix Sort

Algorithm : Sequential Radix Sort "CE0", adapted from [KR08; Ran07]

Function RadixSortCE0($\mathcal{S}, h$)

Input : $\mathcal{S} = [\, s_0, \ldots, s_{n-1} \,]$ an array of n strings with common prefix h.

$c := [\, 0, \ldots, 0 \,]$ // *Allocate* $|\Sigma|$ *integer counters initialized with zero,*

for $i = 0, \ldots, n-1$ **do** $c[s_i[h]]{+}{+}$ // *and count character occurrences.*

$b := [\, 0, \bot, \ldots, \bot \,]$ // *Calculate exclusive prefix sum of counters*

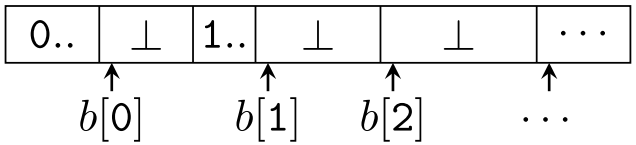
for $i = 1, \ldots, n-1$ **do** $b[i] := b[i-1] + c[i-1]$ // *as bucket pointers.*

$\mathcal{T} :=$ allocate(n, string pointer) // *Allocate temporary array for sorted output.*

for $i = 0, \ldots, n-1$ **do** // *Reorder*

  $\mathcal{T}[b[s_i[h]]] :=$ **move**(s_i) // *into*

  $b[s_i[h]]{+}{+}$ // *buckets*



copy($\mathcal{T} \to \mathcal{S}$), deallocate($\mathcal{T}, b$).

$x := c[0]$ // *Track beginning of bucket as* x,

for $i = 1, \ldots, |\Sigma| - 1$ **do** // *recurse into every unfinished bucket,*

  StringSort($\mathcal{S}[x \,..\, x + c[i]], h + 1$) // *except for the first (zero-termination),*

  $x := x + c[i]$ // *which contains all fully sorted strings.*

Output : The array $\mathcal{S}$ is fully sorted lexicographically.

String Sorting: Multikey Quicksort

Also: MKQS / ternary quicksort

Function mkqSort(S : Sequence **of** String, $\ell : \mathbb{N}$) : Sequence **of** String

assert $\forall e, e' \in S : e[0..\ell-1] = e'[0..\ell-1]$

if $|S| \leq 1$ **then return** S // base case

pick $p \in S$ uniformly at random // pivot string

return concatenation of mkqSort($\langle e \in S : e[\ell] < p[\ell] \rangle, \ell$),

mkqSort($\langle e \in S : e[\ell] = p[\ell] \rangle, \ell + 1$), and

mkqSort($\langle e \in S : e[\ell] > p[\ell] \rangle, \ell$)

- Running Time: $O(|S| \log |S| + \sum_{t \in s} |t|)$
- More precisely: $O(|S| \log |S| + d)$ (d: Sum of unique prefixes)
- Exercise: in-place!

String Sorting: Multikey Quicksort

```
S A A L
B I E N E
E H R E
H A U S
A R M
M I E T E
T A S S E
M O R D
H A N D
S E E
H U N D
H A L L E
N A C H T
```

String Sorting: Multikey Quicksort

```
    SAAL
    BIENE
    EHRE
    HAUS
    ARM
    MIETE
    TASSE
    MORD
p   HAND
    SEE
    HUND
    HALLE
    NACHT
```

String Sorting: Multikey Quicksort

BIENE
EHRE
ARM

HAUS
HAND
HUND
HALLE

SAAL
TASSE
MIETE
MORD
SEE
NACHT

String Sorting: Multikey Quicksort

```
    BIENE
p   EHRE
    ARM

    HAUS
    HAND
    HUND
    HALLE

    SAAL
    TASSE
    MIETE
    MORD
    SEE
    NACHT
```

```
    ARM
    BIENE
    EHRE

    HAUS
p   HAND
    HUND
    HALLE

    SAAL
    TASSE
    MIETE
    MORD
    SEE
    NACHT
```

String Sorting: Multikey Quicksort

```
  ARM
  BIENE
  EHRE
  HAUS
p HAND
  HUND
  HALLE
  SAAL
  TASSE
  MIETE
  MORD
  SEE
  NACHT
```

```
  ARM
  BIENE
  EHRE
  HAUS
p HAND
  HALLE
  HUND
  SAAL
  TASSE
  MIETE
  MORD
  SEE
  NACHT
```

String Sorting: Multikey Quicksort

```
  ARM
  BIENE
  EHRE
  HAUS
p HAND
  HALLE
  HUND
  SAAL
  TASSE
  MIETE
  MORD
  SEE
  NACHT
```

```
  ARM
  BIENE
  EHRE
  HALLE
  HAND
  HAUS
  HUND
  SAAL
  TASSE
p MIETE
  MORD
  SEE
  NACHT
```

String Sorting: Multikey Quicksort

```
  ARM
  BIENE
  EHRE
  HALLE
  HAND
  HAUS
  HUND
  SAAL
  TASSE
p MIETE
  MORD
  SEE
  NACHT
```

```
  ARM
  BIENE
  EHRE
  HALLE
  HAND
  HAUS
  HUND
p MIETE
  MORD
  NACHT
  SAAL
  SEE
  TASSE
```

String Sorting: Multikey Quicksort

```
  ARM
  BIENE
  EHRE
  HALLE
  HAND
  HAUS
  HUND
p MIETE
  MORD
  NACHT
  SAAL
  SEE
  TASSE
```

```
  ARM
  BIENE
  EHRE
  HALLE
  HAND
  HAUS
  HUND
  MIETE
  MORD
  NACHT
p SAAL
  SEE
  TASSE
```

String Sorting (without terminal symbol)

Function mkqSort(S : Sequence **of** String, $\ell : \mathbb{N}$) : Sequence **of** String

if $|S| \leq 1$ **then return** S

$S_\perp \leftarrow \langle e \in S : |e| = \ell \rangle; S \leftarrow S \setminus S_\perp$

select pivot $p \in S$

$S_< \leftarrow \langle e \in S : e[\ell] < p[\ell] \rangle$

$S_= \leftarrow \langle e \in S : e[\ell] = p[\ell] \rangle$

$S_> \leftarrow \langle e \in S : e[\ell] > p[\ell] \rangle$

return concatenation of $S_\perp$,
mkqSort$(S_<, \ell)$,
mkqSort$(S_=, \ell + 1)$, and
mkqSort$(S_>, \ell)$

String Sorting – Running Time Analysis

Most work in character comparison. Two cases:

- $e[\ell] = p[\ell]$: Assign comparison to character $e[\ell]$.
  - $e[\ell]$ will be no longer considered (Recursion with $\ell + 1$)
  - Maximum number of comparisons for string e? Maximum length of longest common prefix of e with $e' \in S$.
- $e[\ell] \neq p[\ell]$: Assign comparison to string e.
  - e will be assigned to $S_<$ or $S_>$. With an optimal pivot choice both sets are at most of size $|S|/2$.
  - e is correctly sorted after at most $\log |S|$ steps.

String Sorting: Algorithm Overview

Sequential General Algorithms

- Radix Sort $\mathrm{O}(d + n \log \sigma)$ [McIlroy et al. '95]
- Multikey Quicksort $\mathrm{O}(d + n \log n)$ exp. [Bentley, Sedgewick '97]
- Burstsort $\mathrm{O}(d + n \log \sigma)$ exp. [Sinha, Zobel '04]
- Binary LCP-Mergesort $\mathrm{O}(d + n \log n)$ [Ng, Kakehi '08]

Theoretical Parallel Algorithms

- "Optimal Parallel String Algorithms: ..." [Hagerup '94]
  $\mathrm{O}(\log N / \log \log N)$ time and $\mathrm{O}(N \log \log N)$ work on CRCW PRAM

Practical Parallel and New General Algorithms

- Parallel Super Scalar String Sample Sort (pS^5) [B, Sanders, ESA'13]
- Parallel K-way LCP-aware Merge(sort) [B, et al. Algorithmica'17]

Comparison of Sequential Algorithms

Experiment [B'18]: Comparison of 39 sequential string sorting algorithms and highly-tuned variants on seven inputs and six machines.

Rank	Algorithm	Gmean	Rank	Algorithm	Gmean
1	KR.radixsort-CE6	1.27	10	B.Seq^5-UI	1.67
2	KRB.radixsort-CE3s	1.28	14	R.burstsort-vec	1.72
3	KR.radixsort-CE7	1.28	16	SZ.burstsortA	1.78
4	KRB.radixsort-CI3s	1.33	23	BS.mkqs	2.36
5	R.mkqs-cache8	1.34	28	NK.LCP-Mergesort	2.96
6	KR.radixsort-CE2	1.49	34	MBM.radixsort	4.82
7	KR.radixsort-DB	1.55	35	AN.ForwardRadix16	4.88

Results:

- Hardware-specific acceleration are very important
- Caching of characters reduce random-accesses but increase memory usage.

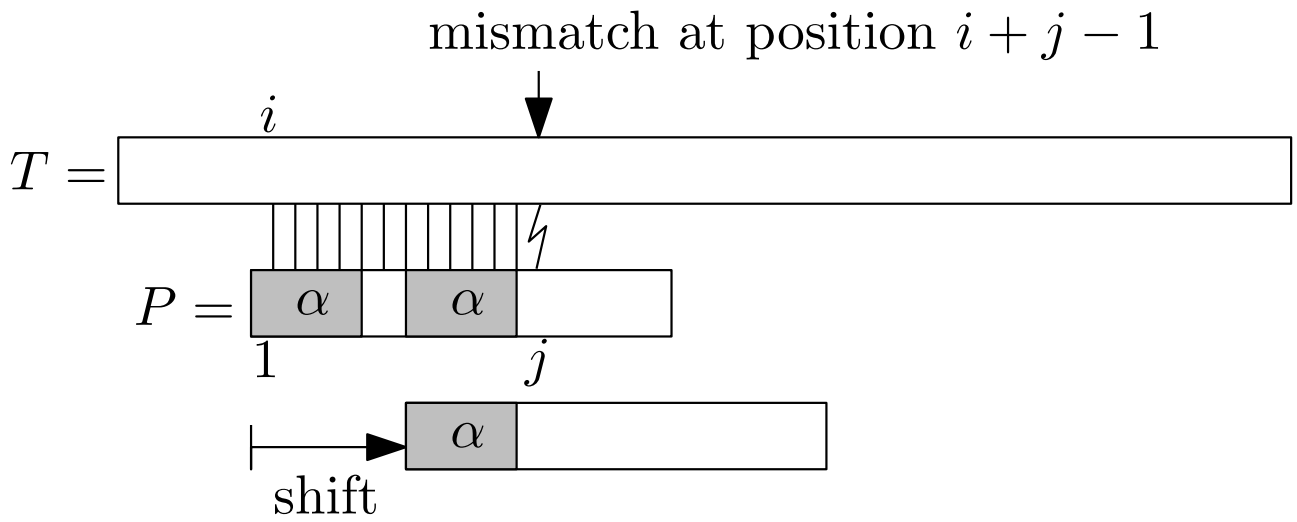
Naive Pattern Matching

- Task: Find all occurences of P in T
  - n: Length of T
  - m: Length of P
- naive in $O(nm)$ time

$i, j := 1$ // indexes in T and P
while $i \leq n - m + 1$
  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters
  if $j > m$ **then print** "P occurs at position i in T"
  i++ // advance in T
  $j := 1$ // restart

Knuth-Morris-Pratt (1977)

- better algorithm in $O(n+m)$ time
- Idea: consider already matched part of string



- $\text{border}[j]$ = length of prefix of $P_{1\ldots j-1}$ that is also suffix of $P_{1\ldots j-1}$. $\text{border}[1] := -1, \quad \text{border}[2] = 0.$

Knuth-Morris-Pratt (1977)

$i := 1$ // index in T

$j := 1$ // index in P

while $i \leq n - m + 1$

  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters

  if $j > m$ **then**

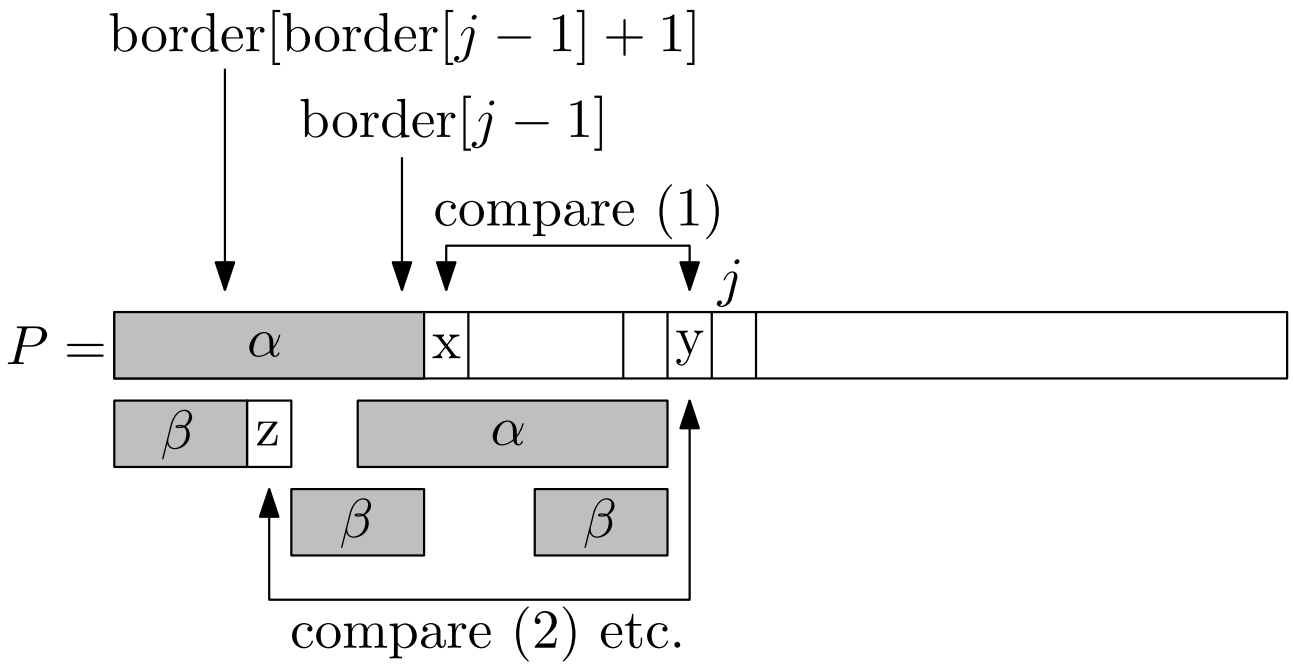
    print "P occurs at position i in T"

  $i := i + j - \text{border}[j] - 1$ // advance in T

  $j := \max\{1, \text{border}[j] + 1\}$ // skip first $\text{border}[j]$ characters of P

Calculation of the Border-Array

- assume the values up to position $j-1$ are already calculated



Calculation of the Border-Array

- in $O(m)$ time:

$\text{border}[1] := -1$

$i := \text{border}[1]$ // position in P

for $j = 2, \ldots, m+1$

  while $i \geq 0$ and $p_{i+1} \neq p_{j-1}$ **do** $i = \text{border}[i+1]$

  i++

  $\text{border}[j] := i$

Full-Text Search from Slow to Super Fast

Input: Text S (n:= $|S|$), Pattern P (m:= $|P|$), $n \gg m$

Output: All/First/Next occurence of P in S

naive: $\mathrm{O}(nm)$

Preprocess P: $\mathrm{O}(n+m)$

with errors: ???

Preprocess S: Text Indices. First occurence:

Inverted Index: Good Heuristic

Suffix Array: $\mathrm{O}(m \log n) \ldots \mathrm{O}(m)$

Inverted Index

```
1  The old night keeper keeps the keep in the town
2  In the big old house in the big old gown
3  The house in the town had the big old keep
4  Where the old night keeper never did sleep
5  The night keeper keeps the keep in the night
6  And keeps in the dark and sleeps in the light
```

term t	f_t	inverted list for t
and	1	(6,2)
big	2	(2,2), (3,1)
dark	1	(6,1)
did	1	(4,1)
gown	1	(2,1)
had	1	(3,1)
house	2	(2,1), (3,1)
in	5	(1,1), (2,2), (3,1), (5,1), (6,2)
keep	3	(1,1), (3,1), (5,1)
...	...	...

Some "Stringology"-Notation

Alphabet Σ: Set $\{a, b, c, \ldots\}$.

String S: Array $S[0..n) := S[0..n-1] := [S[0], \ldots, S[n-1]]$

of characters in Σ.

Suffix: $S_i := S[i..n)$

Terminal Symbol: $S[n] := S[n+1] := \cdots := 0$

0 is smaller than all other characters

$S =$ `banana`:

0	`banana`
1	`anana`
2	`nana`
3	`ana`
4	`na`
5	`a`

Sort Suffixes

Sort set $\{S_0, S_1, \ldots, S_{n-1}\}$
of suffixes from the string S of length n
(Alphabet $[1,n] = \{1,\ldots,n\}$)
in lexicographical order.

- suffix $S_i = S[i,n]$ für $i \in [0..n-1]$

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

$\Longrightarrow$

5	a
3	ana
1	anana
0	banana
4	na
2	nana

Applications

- Full-text search
- Burrows-Wheeler transformation (`bzip2` compression)
- Alternative for complex suffix tree
- Bioinformatics: Search for repetitions (in $\mathrm{O}(n)$ possible?) ,...

Full-Text Search

Search pattern $P[0..m)$ in text $S[0..n)$

with a suffix array SA of S.

Binary Search: $\mathrm{O}(m \log n) \rightarrow$ good for short patterns

Binary search with LCP: $\mathrm{O}(m + \log n)$ if we precompute the

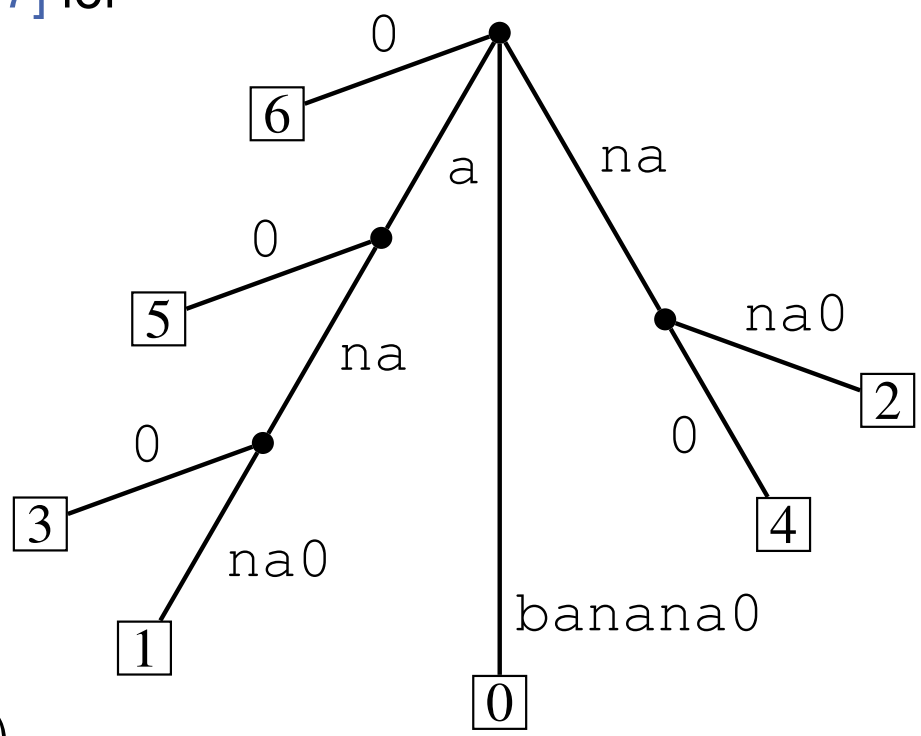
longest common prefixes for all suffixes

Suffix-Tree: constructable with suffix array in $\mathrm{O}(n)$ time

Suffix Tree [Weiner '73][McCreight '76][Ukkonen '95]

- ☐ Compressed trie of suffixes
- \+ Construction time $\mathrm{O}(n)$ [Farach 97] for integral alphabets
- \+ Most powerful tool of stringology?
- – High memory consumption
- – Efficient construction is complicated
- ☐ Computable with SA in time $\mathrm{O}(n)$

$S = \texttt{banana0}$



Search in Suffix Trees

- Search (all/one) occurence of $P_{1..m}$ in T:
- If we use $|\Sigma|$ space to store outgoing edges of a tree node:
  - $O(m)$ search time
  - $O(n|\Sigma|)$ space
- If we use #childs space to store outgoing edges of a tree node:
  - $O(m \log |\Sigma|)$ search time
  - $O(n)$ space

Alphabet Model

Ordered Alphabet: it is only possible to compare characters

Constant Alphabet Size: finite set

which size does not depend on n.

Integral Alphabet: Alphabet is $\{1, \ldots, \sigma\}$

for an integral number $\sigma \geq 2$

Ordered → Integral Alphabet

Sort characters of S

Replace $S[i]$ with its rank

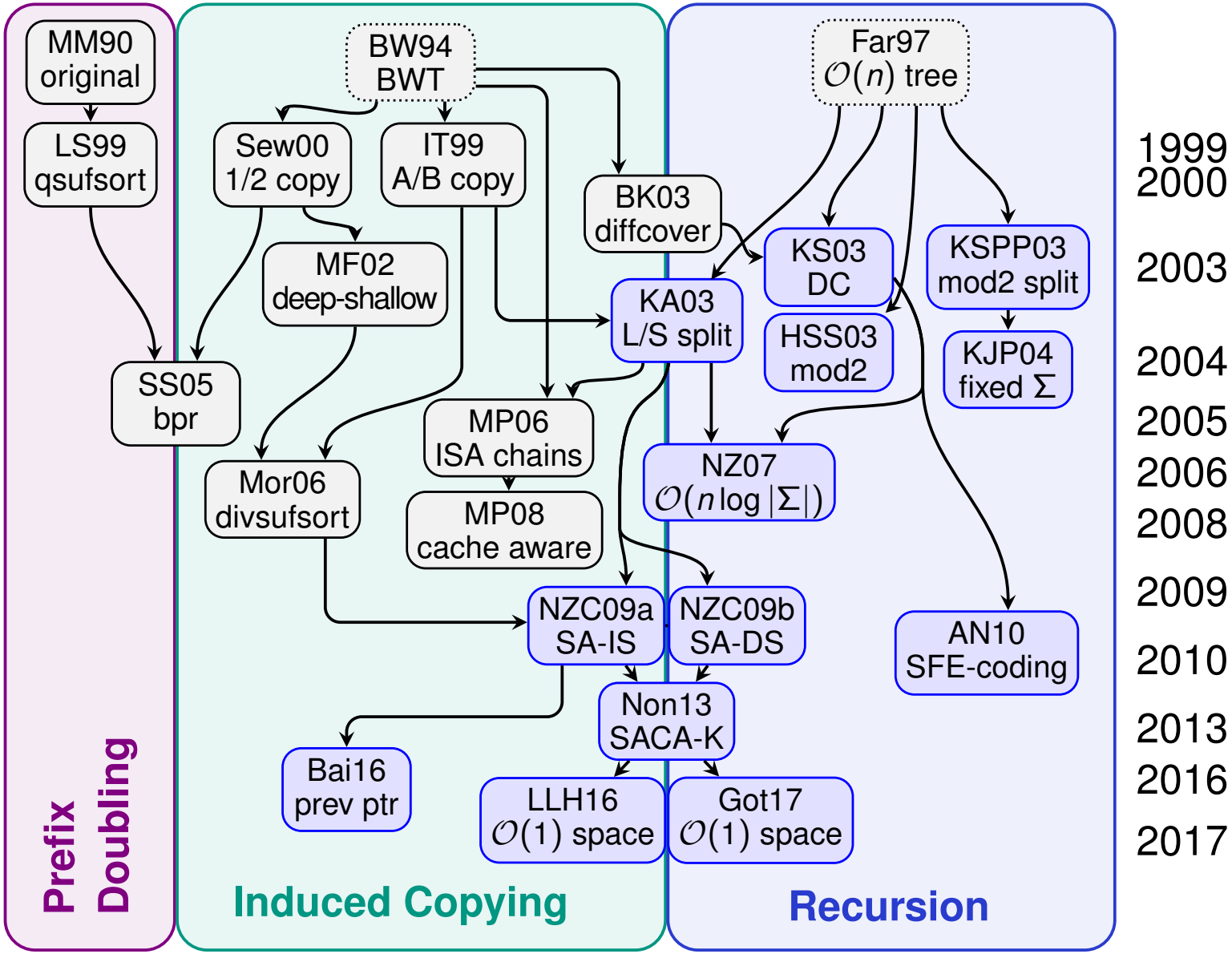
```
012345     135024
banana ->  aaabnn
213131 <-  111233
```

Generalisation: Lexicographic Names

Sort k-tuples $S[i..i+k)$ for $i \in 1..n$

Replace $S[i]$ with the rank of $S[i..i+k)$ within the k-tuples

Suffix Array Construction Algorithms



Manber Myers Prefix Doubling Algorithm

for $j := 0$ **to** $\log n$ **do**

sort pairs $p_i := (S[i], S[i+2^j])$ for $i \in 1..n$

replace $S[i]$ with the rank of p_i within the pairs

for $i := 0$ **to** $n-1$ **do** $SA[S[i]-1] := i$

```
012345      3  2  4  2  4  1
banana -> ba an na an na a0
           4  3  6  2  5  1
324241 -> 34 22 44 21 40 10

436251 -> ... no change
SA= 531042}
```

Manber Myers Prefix Doubling Algorithm

for $j := 0$ **to** $\log n$ **do**

sort pairs $p_i := (S[i], S[i+2^j])$ for $i \in 1..n$

replace $S[i]$ with the rank of p_i within the pairs

for $i := 0$ **to** $n-1$ **do** $SA[S[i]-1] := i$

Running time $\mathrm{O}(n \log n)$ with radix sort.

Suffix Arrays

from

Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Pages 1–19, Number 6, Volume 53.

A first Divide-And-Conquer Approach

1. $SA^1 =$sort $\{S_i : i$ is odd$\}$ (Recursion)
2. $SA^0 =$sort $\{S_i : i$ is even$\}$ (simple with SA^1)
3. Merge SA^0 und SA^1 (complicated)

Problem: How to compare even and odd suffixes?

[Farach 97] developed a linear time algorithm for suffix tree construction, which is based on that idea.

It is also the only known divide-and-conquer algorithm for suffix arrays

Compute SA^1

- Remove first character.

  `banana` $\rightarrow$ `anana`

- Replace pairs of characters with its lexicographicals names

  | an | an | a0 | $\rightarrow$ 221

- Recursion

  $\langle 1, 21, 221 \rangle$

- back transformation

  $\langle \mathtt{a}, \mathtt{ana}, \mathtt{anana} \rangle$

Compute SA^0 with SA^1

1	anana
3	ana
5	a

$\Longrightarrow$

5	a
3	ana
1	anana

Replace S_i, $i \bmod 2 = 0$ with $(S[i], r(S_{i+1}))$

with $r(S_{i+1})$:= rank of S_{i+1} in SA^1

0	b	3 (anana)
2	n	2 (ana)
4	n	1 (a)

$\Longrightarrow$

0	banana
4	na
2	nana

Radix-Sort

Asymmetrical Divide-and-Conquer

1. $SA^{12} =$sort $\{S_i : i \bmod 3 \neq 0\}$ (Recursion)
2. $SA^{0} =$sort $\{S_i : i \bmod 3 = 0\}$ (simply with SA^{12})
3. Merge SA^{12} and SA^{0} (easy!)

$$S = \texttt{banana}$$

5	a
1	anana
4	na
2	nana

$+$

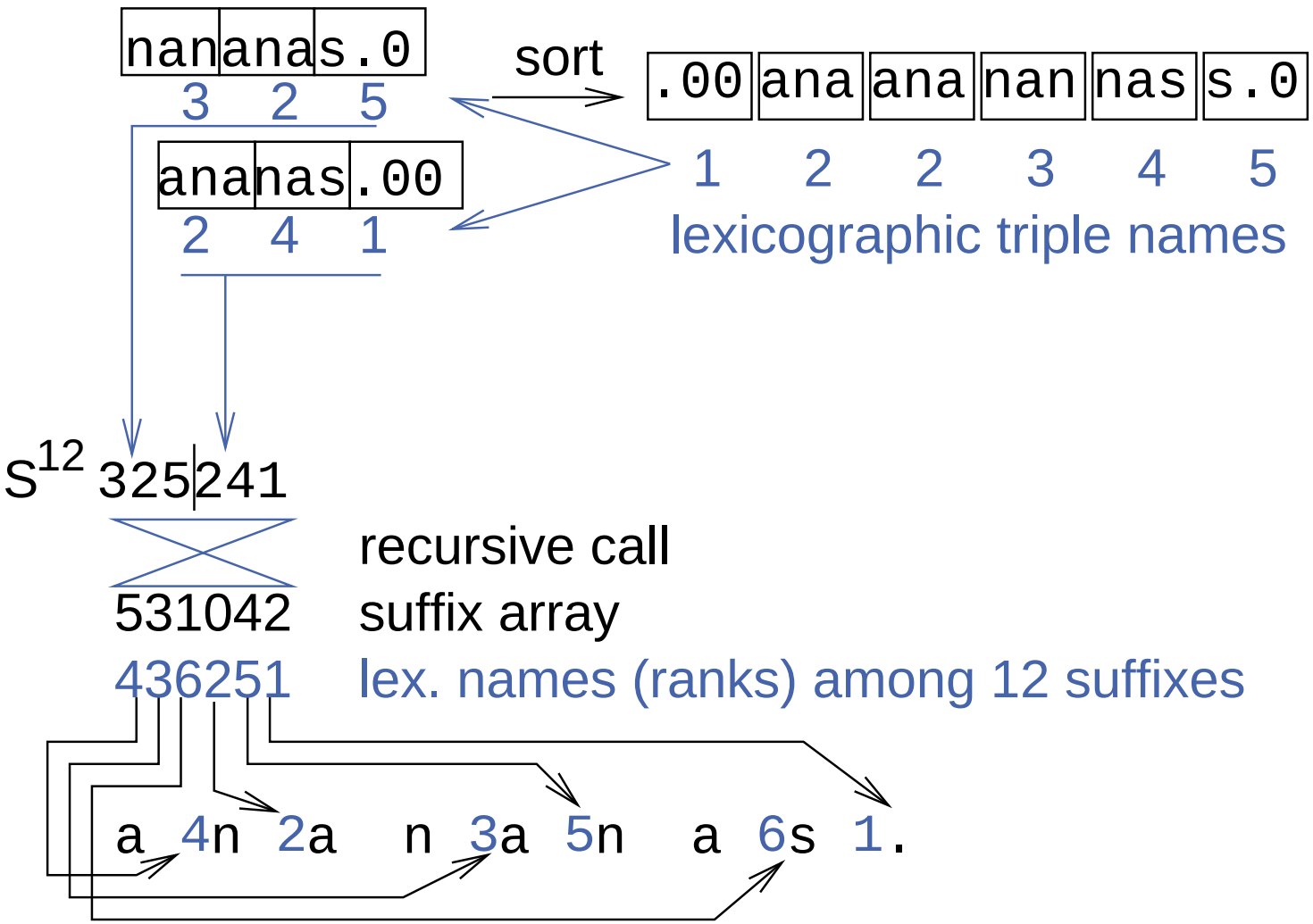
3	ana
0	banana

$\Longrightarrow$

5	a
3	ana
1	anana
0	banana
4	na
2	nana

Recursion, Example

```
   012345678
S  anananas.
```



Recursion

- sort triples $S[i..i+2]$ for $i \bmod 3 \neq 0$
  (LSD Radix-Sort)
- Find lexicographic names $S'[1..2n/3]$ of triples,
  (that is $S'[i] < S'[j] \Leftrightarrow S[i..i+2] < S[j..j+2]$)
- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$,
  Suffix S^{12}_i of S^{12} represents S_{3i+1}
  Suffix $S^{12}_{n/3+i}$ of S^{12} represents S_{3i+2}
- Recursion on (S^{12}) (Alphabet size $\leq 2n/3$)
- Annotate the 12-suffixes with its position in the recursive calculated solution

Least Significant Digit First Radix Sort

Here: Sort n 3-tuples of integral numbers $\in [0..n]$ in lexicographical order

Sort after third position

Elements are sorted after Pos. 3

Sort stable after second position

Elements are sorted after Pos. 2, 3

Sort stable after first position

Elements are sorted after Pos. 1, 2, 3

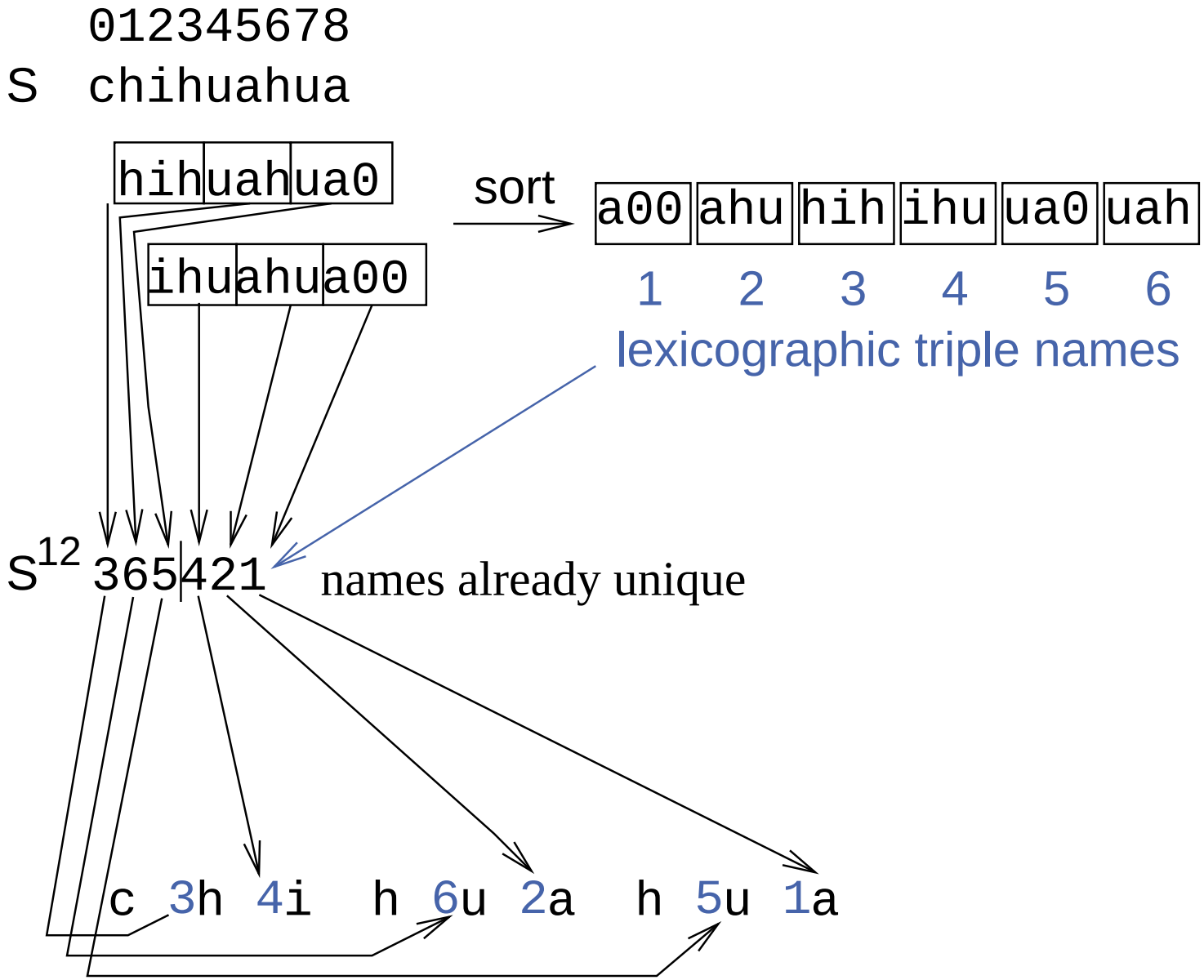
Stable Integral Sorting

Sort $a[0..n)$ after $b[0..n)$ with $\mathrm{key}(a[i]) \in [0..n]$

$c[0..n] := [0,\ldots,0]$	Counter
for $i \in [0..n)$ do $c[a[i]]{+}{+}$	count
$s := 0$	
for $i \in [0..n)$ do $(s, c[i]) := (s + c[i], s)$	prefix sum
for $i \in [0..n)$ do $b[c[a[i]]{+}{+}] := a[i]$	bucket sort

Time $\mathrm{O}(n)$!

Recursion Example: Easy Case



Sorting of mod 0 Suffixes

0	c 3 (h 4 i h 6 u 2 a h 5 u 1 a)
1	
2	
3	h 6 (u 2 a h 5 u 1 a)
4	
5	
6	h 5 (u 1 a)
7	
8	

Use Radix-Sort (LSD-Order already known)

Merge SA^{12} and SA^{0}

$0 < 1 \Leftrightarrow c\,{}^{n} < c\,{}^{n}$

$0 < 2 \Leftrightarrow cc\,{}^{n} < cc\,{}^{n}$

```
3:   h 6u 2(ahua)
6:   h 5u 1(a)
0:   c 3h 4(ihuahua)
```

$\Downarrow$

```
4:   (6)u 2(ahua)
7:   (5)u 1(a)
2:   (4)i  h 6(uahua)
1:   (3)h 4(ihuahua)
5:   (2)a  h 5(ua)
8:   (1)a 0 0 0(0)
```

```
8:   a
5:   ahua
0:   chihuahua
1:   hihuahua
6:   hua
3:   huahua
2:   ihuahua
7:   ua
4:   uahua
```

Analysis

1. Recursion: $T(2n/3)$ plus

   Extract Triples: $\mathrm{O}(n)$ (forall $i, i \bmod 3 \neq 0$ do ...)

   Sort Triples: $\mathrm{O}(n)$

   (e.g., LSD-first radix sort — 3 Iterations)

   Find lexicographic names of triples: $\mathrm{O}(n)$ (scan)

   Construct instance for recursion: $\mathrm{O}(n)$ (forall names do ...)

2. $SA^0 =$sort $\{S_i : i \bmod 3 = 0\}$: $\mathrm{O}(n)$

   (1 Radix-Sort Iteration)

3. merge SA^{12} and SA^0: $\mathrm{O}(n)$

   (usual merge with strange comparison function)

In Total: $T(n) \leq cn + T(2n/3)$

$\Rightarrow T(n) \leq 3cn = \mathrm{O}(n)$

Implementation: Comparison-Operator

```
inline bool leq(int a1, int a2,   int b1, int b2) {
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3) {
  return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));
}
```

Implementation: Radix-Sort

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                          // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;         // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;    // count occurences
  for (int i = 0, sum = 0;  i <= K;  i++) { // exclusive prefix sums
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}
```

Implementation: Sort Triples

```
void suffixArray(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);
```

Implementation: Names of Triples

```
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0;  i < n02;  i++) {
  if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
    name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
  }
  if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; } // left half
  else                  { s12[SA12[i]/3 + n0] = name; } // right half
```

Implementation: Recursion

```
// recurse if names are not yet unique
if (name < n02) {
  suffixArray(s12, SA12, n02, name);
  // store unique names in s12 using the suffix array
  for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
  for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;
```

Implementation: Sorting of the mod 0 suffixes

```
for (int i=0, j=0;  i < n02;  i++)
  if (SA12[i] < n0) s0[j++] = 3*SA12[i];
radixPass(s0, SA0, s, n0, K);
```

Implementation: Merge

```
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    { // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) { // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  { // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0; }
```

Generalisation: Difference Cover

A difference cover D modulo v is a subset of $[0,v)$ such that
$\forall i \in [0,v) : \exists j,k \in D : i \equiv k - j \pmod{v}$.

Example:

$\{1,2\}$ is a difference cover modulo 3.

$\{1,2,4\}$ is a difference cover modulo 7.

- ☐ Leads to a more space efficient approach
- ☐ Faster for smaller alphabets

Improvement / Generalisation

- tuning
- larger difference cover
- Combine with best algorithm for simple inputs

  [Manzini Ferragina 02, Schürmann Stoye 05, Yuta Mori 08]

Suffix Array Construction: Summary

- easy, direct, linear time construction for suffix arrays
- easy customizable for advanced computational models
- Generalisation of difference cover leads to a more space-efficient implementation

Search in Suffix Arrays

Given: T, SA, P.

$l := 1; r := n+1$

while $l < r$ **do** // search left index

  $q := \lfloor \frac{l+r}{2} \rfloor$

  if $P >_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

    then $l := q+1$ **else** $r := q$

$s := l; l{-}{-}; r := n$

while $l < r$ **do** // search right index

  $q := \lceil \frac{l+r}{2} \rceil$

  if $P =_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

    then $l := q$ **else** $r := q-1$

return $[s,l]$

- Time $O(m \log n)$ (also possible: $O(m + \log n)$)

LCP-Array

stores length of longest common prefix of lexicographical adjacent suffixes!

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

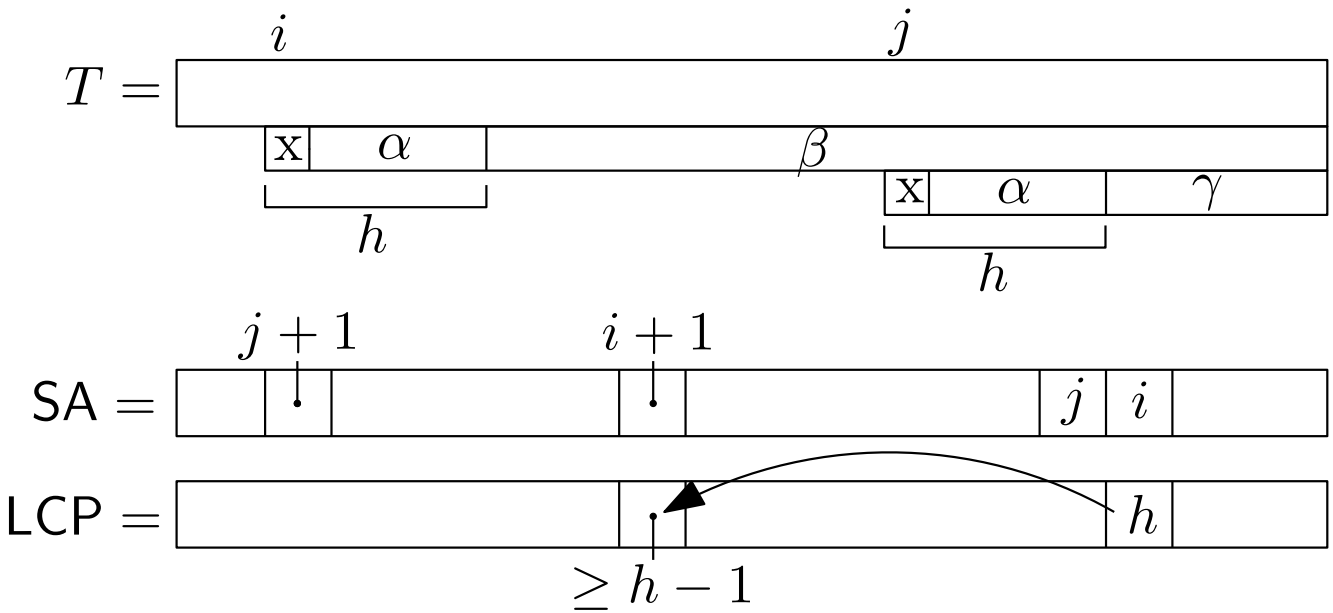
5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

$\perp$	a
1	**a**na
3	**ana**na
0	banana
0	na
2	**na**na

LCP-Array: Computation

- naive $O(n^2)$
- Inverse Suffix-Array: $\mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$ (position of i in SA?)
- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.



LCP-Array: Computation

[KLAAP '01]

- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.

$h := 0, \mathrm{LCP}[1] := 0$

for $i = 1, \ldots, n$ **do**

  if $\mathrm{SA}^{-1}[i] \neq 1$ **then**

    while $t_{i+h} = t_{\mathrm{SA}[\mathrm{SA}^{-1}[i]-1]+h}$ **do** $h++$

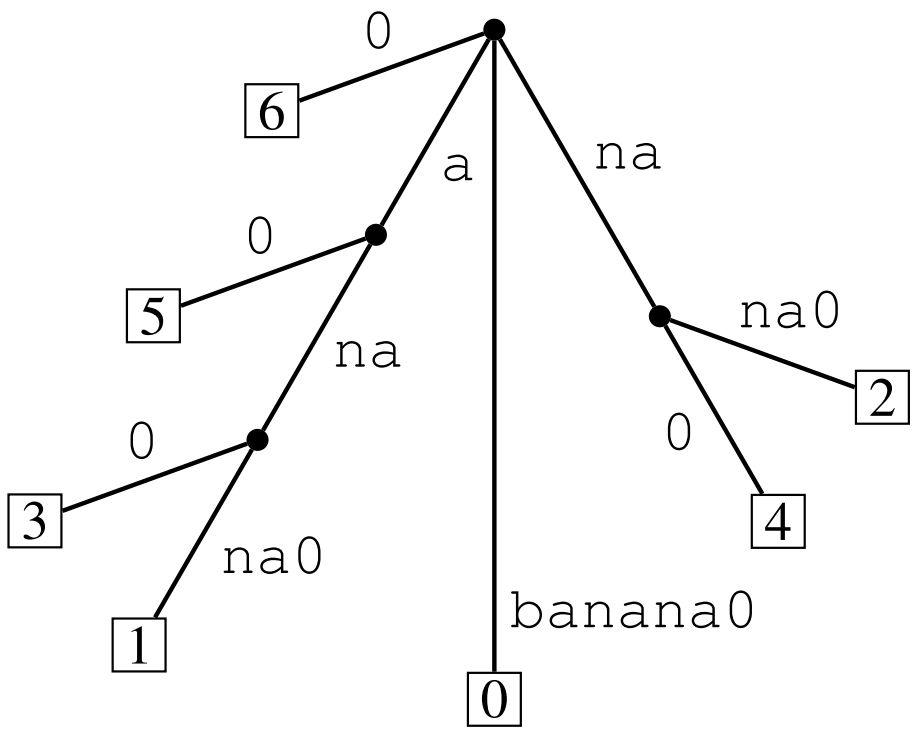
    $\mathrm{LCP}[\mathrm{SA}^{-1}[i]] := h$

    $h := \max(0, h-1)$

- Time: $O(n)$

Suffix Tree Computation with SA and LCP

$S = \texttt{banana0}$



- naive: $O(n^2)$
- with Suffix-Array: in lexicographical order

Suffix Tree Computation with SA and LCP

- LCP-Array helps!
- Consider only rightmost path!
- Find deepest node with string-depth $\leq \mathrm{LCP}[i] \rightsquigarrow$ Insertion Point!

	0	1	2	3	4	5	6
$\mathrm{SA} =$	6	5	3	1	0	4	2
$\mathrm{LCP} =$	0	0	1	3	0	0	2

- Time $O(n)$

Data Compression

Problem: with naive storage, data consumes often a lot of space / communication bandwidth. $\Rightarrow$ can be often reduced significantly

Variants:

- Lossy (mp3, jpg, ...) / **Loss-Free** (Text, Files, Search Engines, Databases, Medical Image Processing, Professional Photography, ...)
- **1D** (text, sequences of numbers,...) / 2D (Images) / 3D (Movies)
- **only storage** / with operations ($\rightsquigarrow$ succinct data structures)

Loss-Free Text Compression

Given: Alphabet Σ

String $S = \langle s_1, \ldots, s_n \rangle \in \Sigma^*$

Text compression algorithm $f : S \to f(S)$ with $|f(S)|$ (e.g. measured in bits) as small as possible.

Theory Loss-Free Text Compression

Information theory. For example:

Entropy: $H(S) = \sum_{c \in \Sigma} p(c) \log(1/p(c))$ where

$p(c) = |\{s_i : s_i = c\}|/n$ is the relative frequency of c.

lower bound for # bits per character if text originate from a random data source.

$\rightsquigarrow$ Huffman encoding is approximately optimal! (entropy encoding)

????

Alternatively:

Entropy of higher order consider substrings of fixed length

"Ultimatively": Kolmogorov complexity. Unfortunately not calculable.

Theory Loss-Free Text Compression

Entropy of higher order: Given a text S of length n over an alphabet Σ. We define $N(\omega, S)$ as the concatenation of all characters, which follows in S on occurences of $\omega \in \Sigma^k$. We define the empirical entropy of order k as follows:

$$H_k(S) = \sum_{\omega \in \Sigma^k} \frac{|N(\omega, S)|}{n} H(N(\omega, S))$$

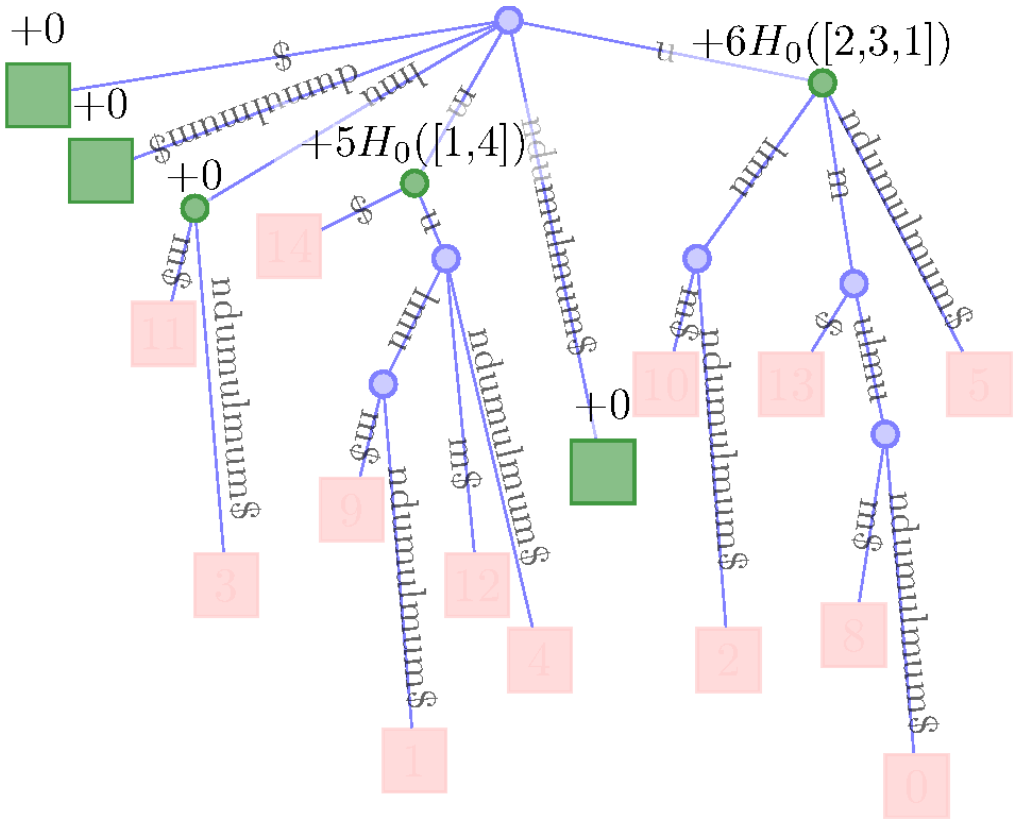
Example: $S = \texttt{ananas}$, $k = 2$

$N(\texttt{an}, S) = \texttt{aa}$, $N(\texttt{na}, S) = \texttt{ns}$, $N(\texttt{as}, S) = \varepsilon$.

$H_2(\texttt{ananas}) = \frac{2}{6} H(\texttt{ns}) = \frac{1}{3}$ bits

Theory Loss-Free Text Compression

H_k: Calculation via suffix tree (Example: H_1)



Theory Loss-Free Text Compression

Values of the empirical entropy in practice

$H_k(S)$ in bits und (# unambiguous context/$|S|$ in percent)

k	dblp.xml		DNA		english		proteins	
0	5.26	0.0	1.97	0.0	4.53	0.0	4.20	0.0
1	3.48	0.0	1.93	0.0	3.62	0.0	4.18	0.0
2	2.17	0.0	1.92	0.0	2.95	0.0	4.16	0.0
3	1.43	0.1	1.92	0.0	2.42	0.0	4.07	0.0
4	1.05	0.4	1.91	0.0	2.06	0.3	3.83	0.1
5	0.82	1.3	1.90	0.0	1.84	1.0	3.16	1.7
6	0.70	2.7	1.88	0.0	1.67	2.7	1.50	17.4

Dictionary-based Text Compression

General Idea: Choose $\Sigma' \subseteq \Sigma^*$ and replace $S \in \Sigma^*$ with $S' = \langle s'_1, \ldots, s'_k \rangle \in \Sigma'^*$ such that $S = s'_1 \cdot s'_2 \cdots s'_k$. (with '$\cdot$'= Character concatenation).

Space $n \lceil \log \Sigma \rceil \to k \lceil \log \Sigma' \rceil$.

With entropy encoding of Σ' even $k\text{Entropy}(S')$

Problem: requires additional space for dictionary.

OK for big data.

Dictionary-based Text Compression – Example

Full-text search engines often use alphabets $\Sigma' :=$ where the words of the underlying natural language are seperated by spaces (etc.).

Special treatment of seperator symbols etc.

`Gallia est omnis divisa in partes tres,...`

$\rightarrow$ gallia | est | omnis | divisa | in | partes | tres ...

Lempel-Ziv Compression (LZ) [LZ'78]

Idea: build dictionary "on the fly" during encoding and decoding.

Without extra memory!

Naive Lempel-Ziv Compression (LZ) [LZ'78]

Procedure naiveLZCompress$(\langle s_1, \ldots, s_n \rangle, \Sigma)$

$D := \Sigma$ // Init Dictionary

$p := s_1$ // current string

for $i := 2$ **to** n **do**

  if $p \cdot s_i \in D$ **then** $p := p \cdot s_i$

  else

    output code for p

    $D := D \cup p \cdot s_i$

    $p := s_i$

output code for p

Naive LZ Decompression

[LZ'78]

Procedure naiveLZDecode$(\langle c_1, \ldots, c_k \rangle)$

$D := \Sigma$

output decode(c_1)

for $i := 2$ **to** k **do**

  if $c_i \in D$ **then**

    $D := D \cup$ decode$(c_{i-1}) \cdot$ decode$(c_i)[1]$

  else // where $s[1]$ is the first letter.

    $D := D \cup$ decode$(c_{i-1}) \cdot$ decode$(c_{i-1})[1]$

  output decode(c_i)

LZ Example: abracadabra

#	p	output	input	$D \cup =$
1	$\perp$	-	a	a,b,c,d,r
2	a	a	b	ab
3	b	b	r	br
4	r	r	a	ra
5	a	a	c	ac
6	c	c	a	ca
7	a	a	d	ad
8	d	d	a	da
9	a	-	b	-
10	ab	ab	r	abr
11	r	-	a	-
-	ra	ra	-	-

LZ Fine Tuning

- Restrict dictionary size, e.g. $|D| \leq 4096 \rightsquigarrow$ 12bit codes.
- Restart in case dictionary is full $\rightsquigarrow$ works blockwise
- Encoding with variable number of bits (e.g., Huffman, arithmetic encoding)
- Delete rarely used dictionary entries ???
- Implement dictionary efficiently:
  (universal) hashing
- used in zip/gzip with Huffman encoding