Karlsruhe Institute of Technology

Algorithmen II

Peter Sanders, Thomas Worsch, Simon Gog

Übungen:

Demian Hespe, Yaroslav Akhremtsev

Institut für Theoretische Informatik, Algorithmik II

Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS17.php`

4 Stringology (Zeichenkettenalgorithmen)

- Strings sortieren
- Patterns suchen
  - Pattern vorverarbeiten
  - Text vorverarbeiten
    - Invertierte Indizes
    - Suffix Trees / Suffix Arrays
- Datenkompression
- Pattern suchen in komprimierten Indizes

Strings Sortieren

multikey quicksort / ternary quicksort

Function mkqSort(S : Sequence **of** String, $\ell : \mathbb{N}$) : Sequence **of** String
    assert $\forall e, e' \in S : e[1..\ell-1] = e'[1..\ell-1]$
    if $|S| \le 1$ **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$(\langle e \in S : e[\ell] > p[\ell]\rangle, \ell)$

- Laufzeit: $O(|S| \log |S| + \sum_{t \in s} |t|)$
- genauer: $O(|S| \log |S| + d)$ (d: Summe der eindeutigen Präfixe)
- Übung: in-place!

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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

Strings Sortieren

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$(S_>, \ell)$

Strings Sortieren – Laufzeitanalyse

Hauptarbeit in den Buchstabenvergleichen. Zwei Fälle:

- $e[\ell] = p[\ell]$: Ordne den Vergleich dem Zeichen $e[\ell]$ zu.
  - $e[\ell]$ wird danach nicht mehr betrachtet (Rekursion mit $\ell + 1$)
  - Maximale Vergleichsanzahl pro String e? Maximale Länge des längstes gemeinsamen Präfix von e mit $e' \in S$.
- $e[\ell] \neq p[\ell]$: Ordne den Vergleich dem String e zu.
  - e wird zu $S_<$ oder $S_>$ zugeordnet. Mit optimaler Pivotwahl sind beide Mengen höchstens $|S|/2$.
  - Nach höchstens $\log|S|$ Schritten ist e richtig sortiert.

Naives Pattern Matching

- Aufgabe: Finde alle Vorkommen von P in T
  - n: Länge von T
  - m: Länge von P
- naiv in $O(nm)$ Zeit

$i, j := 1$ // 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

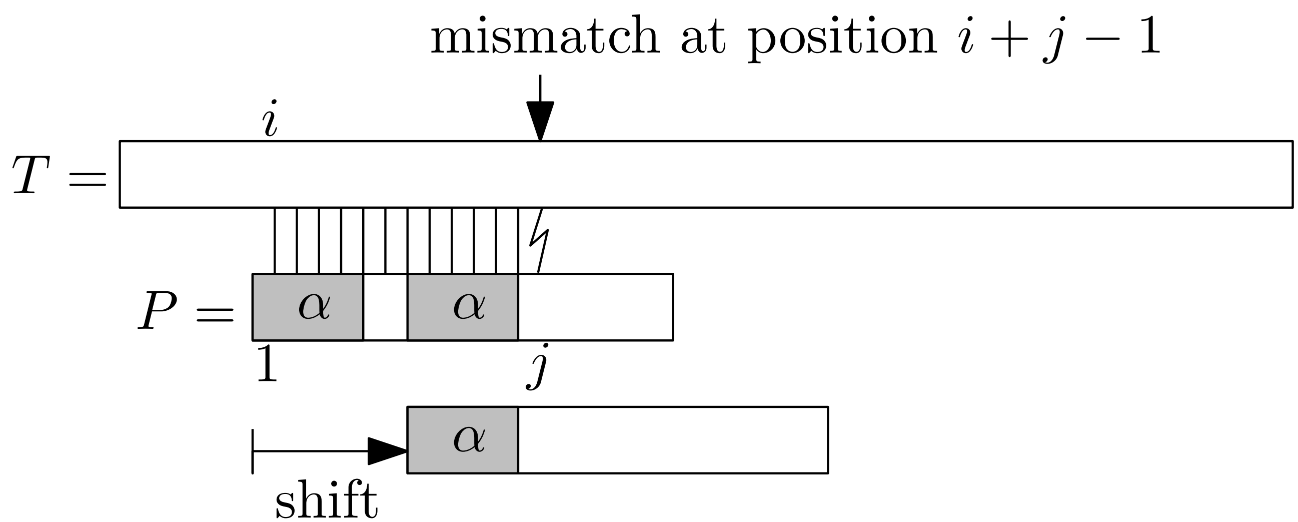
  if $j > m$ **then return** "P occurs at position i in T"

  i++ // advance in T

  $j := 1$ // restart

Knuth-Morris-Pratt (1977)

- besserer Algorithmus in $O(n+m)$ Zeit
- Idee: beachte bereits gematchten Teil



- $\text{border}[j] =$ längstes echtes Präfix von $P_{1\ldots j-1}$, das auch (echtes) Suffix von $P_{1\ldots j-1}$ ist

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**

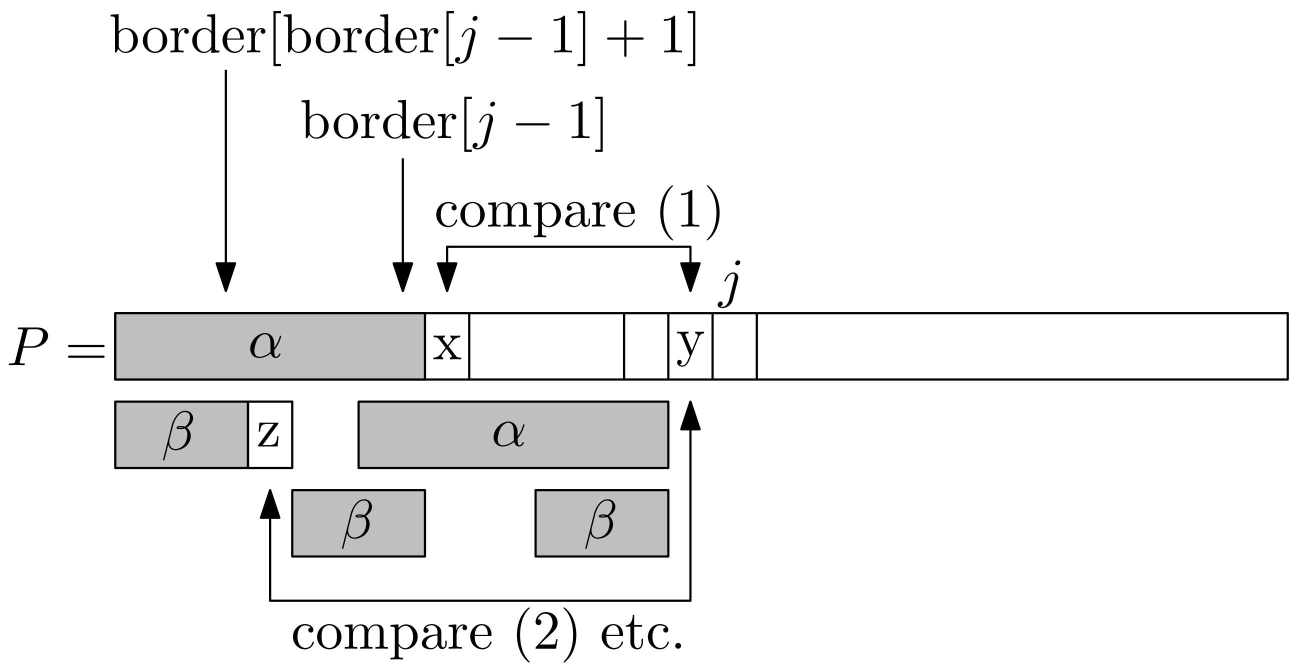
  return "P occurs at position i in T"

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

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

Berechnung des Border-Arrays

- seien die Werte bis zur Position $j-1$ bereits berechnet



Berechnung des Border-Arrays

- in $O(m)$ Zeit:

border$[1] := -1$

$i :=$ border$[1]$ // position in P

for $j = 2, \ldots, m$

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

  i++

  border$[j] := i$

Volltextsuche von Langsam bis Superschnell

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

Gesucht: Alle/erstes/nächstes Vorkommen von P in S

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

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

Mit Fehlern: ???

S vorverarbeiten: Textindizes. Erstes Vorkommen:

Invertierter Index: gute heuristik

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

Suffixtabellen

aus

Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

Etwas "Stringology"-Notation

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

von Buchstaben

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

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

0 ist kleiner als alle anderen Zeichen

Suffixe Sortieren

Sortiere die Menge $\{S_0, S_1, \ldots, S_{n-1}\}$
von Suffixen des Strings S der Länge n
(Alphabet $[1, n] = \{1, \ldots, n\}$)
in lexikographische Reihenfolge.

- 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

Anwendungen

- Volltextsuche
- Burrows-Wheeler Transformation (`bzip2` Kompressor)
- Ersatz für kompliziertere Suffixbäume
- Bioinformatik: Wiederholungen suchen,...

Volltextsuche

Suche Muster (pattern) $P[0..m)$ im Text $S[0..n)$

mittels Suffix-Tabelle SA of S.

Binäre Suche: $\mathrm{O}(m\log n)$ gut für kurze Muster

Binäre Suche mit lcp: $\mathrm{O}(m+\log n)$ falls wir die

längsten gemeinsamen (common) Präfixe

zwischen verglichenen Zeichenketten vorberechnen

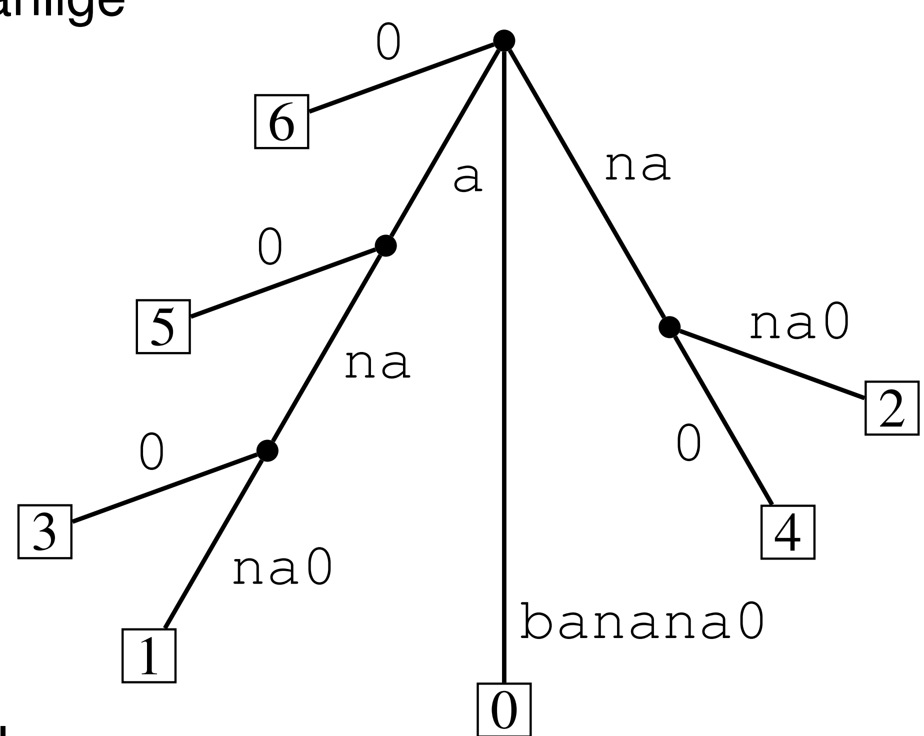
Suffix-Baum: $\mathrm{O}(n)$ kann aus SA berechnet werden

Suffix-Baum

[Weiner '73][McCreight '76]

- □ kompaktierter Trie der Suffixe
- + Zeit $\mathrm{O}(n)$ [Farach 97] für ganzzahlige Alphabete
- + Mächtigstes Werkzeug der Stringology?
- – Hoher Platzverbrauch
- – Effiziente direkte Konstruktion ist kompliziert
- □ kann aus SA in Zeit $\mathrm{O}(n)$ abgelesen werden

$S = \texttt{banana0}$



Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur verglichen werden

Konstante Alphabetgröße: endliche Menge

deren Größe nicht von n abhängt.

Ganzzahliges Alphabet: Alphabet ist $\{1,\ldots,\sigma\}$

für eine ganze Zahl $\sigma \geq 2$

Geordnetes $\rightarrow$ ganzzahliges Alphabet

Sortiere die Zeichen von S

Ersetze $S[i]$ durch seinen Rang

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

Verallgemeinerung: Lexikographische Namen

Sortiere die k-Tupel $S[i..i+k)$ für $i \in 1..n$

Ersetze $S[i]$ durch den Rang von $S[i..i+k)$ unter den Tupeln

Ein erster Teile-und-Herrsche-Ansatz

1. SA^1 =sort $\{S_i : i$ ist ungerade$\}$ (Rekursion)
2. SA^0 =sort $\{S_i : i$ ist gerade$\}$ (einfach mittels SA^1)
3. Mische SA^0 und SA^1 (schwierig)

Problem: wie vergleicht man gerade und ungerade Suffixe?

[Farach 97] hat einen Linearzeitalgorithmus für Suffix-Baum-Konstruktion entwickelt, der auf dieser Idee beruht. Sehr kompliziert.

Das war auch der einzige bekannte Algorithmus für Suffix-Tabellen (läßt sich leicht aus S-Baum ablesen.)

SA^1 berechnen

- Erstes Zeichen weglassen.

  $\texttt{banana} \rightarrow \texttt{anana}$

- Ersetze Buchstabenpaare durch Ihre lexikographischen Namen

  $\boxed{\texttt{an}}\boxed{\texttt{an}}\boxed{\texttt{a0}} \rightarrow \texttt{221}$

- Rekursion

  $\langle 1, 21, 221 \rangle$

- Rückübersetzen

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

Berechne SA^0 aus SA^1

1	anana	$\Longrightarrow$	5	a
3	ana		3	ana
5	a		1	anana

Ersetze S_i, $i \bmod 2 = 0$ durch $(S[i], r(S_{i+1}))$
mit $r(S_{i+1})$:= Rang von S_{i+1} in SA^1

0	b 3(anana)	$\Longrightarrow$	0	banana
2	n 2(ana)		4	na
4	n 1(a)		2	nana

Radix-Sort

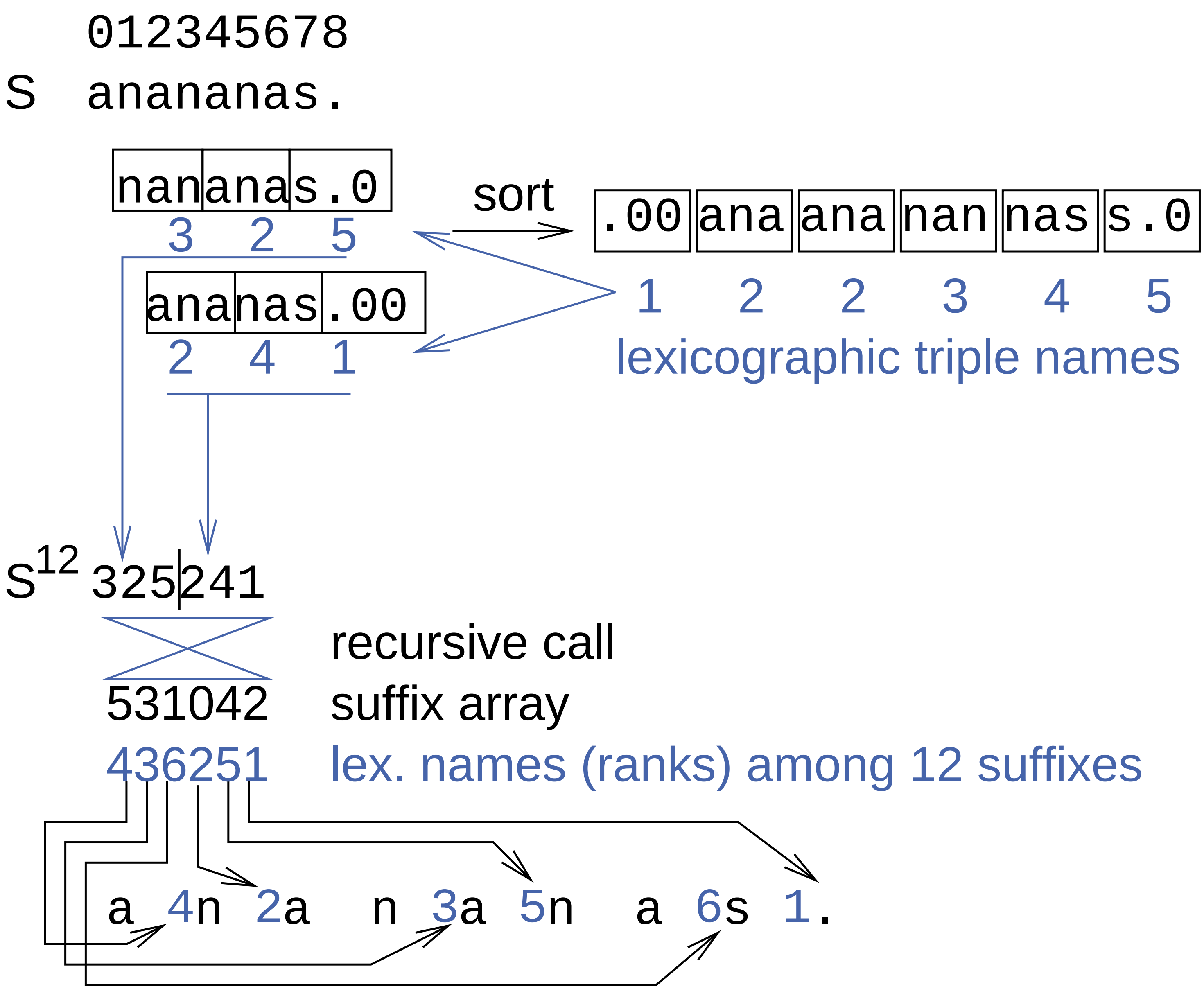
Asymmetrisches Divide-and-Conquer

1. SA^{12} =sort $\{S_i : i \bmod 3 \neq 0\}$ (Rekursion)
2. SA^{0} =sort $\{S_i : i \bmod 3 = 0\}$ (einfach mittels SA^{12})
3. Mische SA^{12} und SA^{0} (einfach!)

$$S = \texttt{banana}$$

5	a			5	a
		3	ana	3	ana
1	anana			1	anana
	+	0	banana $\Longrightarrow$	0	banana
4	na			4	na
2	nana			2	nana

Rekursion, Beispiel



Rekursion

- sortiere Tripel $S[i..i+2]$ für $i \bmod 3 \neq 0$
  (LSD Radix-Sortieren)

- Finde lexikographische Namen $S'[1..2n/3]$ der Tripel,
  (d.h., $S'[i] < S'[j]$ gdw $S[i..i+2] < S[j..j+2]$)

- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$,
  Suffix S^{12}_i von S^{12} repräsentiert S_{3i+1}
  Suffix $S^{12}_{n/3+i}$ von S^{12} repräsentiert S_{3i+2}

- Rekursion auf (S^{12}) (Alphabetgröße $\leq 2n/3$)

- Annotiere die 12-Suffixe mit ihrer Position in rek. Lösung

Least Significant Digit First Radix Sort

Hier: Sortiere n 3-Tupel von ganzen Zahlen $\in [0..n]$ in lexikographische Reihenfolge

Sortiere nach 3. Position

Elemente sind nach Pos. 3 sortiert

Sortiere stabil nach 2. Position

Elemente sind nach Pos. 2,3 sortiert

Sortiere stabil nach 1. Position

Elemente sind nach Pos. 1,2,3 sortiert

Stabiles Ganzzahliges Sortieren

Sortiere $a[0..n)$ nach $b[0..n)$ mit $\mathrm{key}(a[i]) \in [0..n]$

$c[0..n] := [0,\ldots,0]$ Zähler

for $i \in [0..n)$ do $c[a[i]]{+}{+}$ zähle

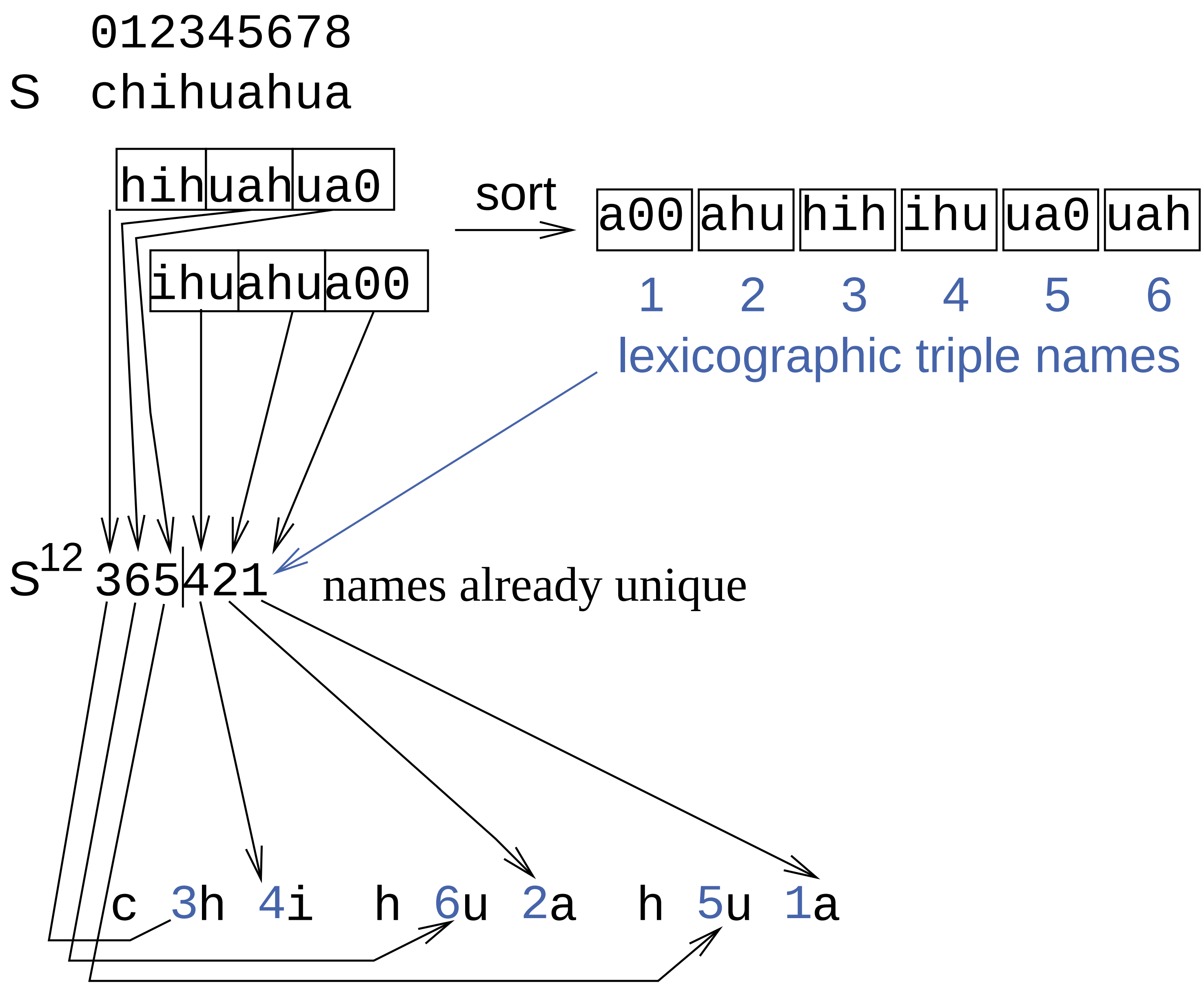
$s := 0$

for $i \in [0..n)$ do $(s, c[i]) := (s + c[i], s)$ Präfixsummen

for $i \in [0..n)$ do $b[c[a[i]]{+}{+}] := a[i]$ bucket sort

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

Rekursions-Beispiel: Einfacher Fall



Sortieren der mod 0 Suffixe

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	

Benutze Radix-Sort (LSD-Reihenfolge bereits bekannt)

Mische SA^{12} und SA^{0}

$0 < 1 \Leftrightarrow c_n < c_n$
$0 < 2 \Leftrightarrow cc_n < cc_n$

```
3: h 6u 2(ahua)       4: (6)u 2(ahua)
6: h 5u 1(a)          7: (5)u 1(a)
0: c 3h 4(ihuahua)    2: (4)i  h 6(uahua)
                      1: (3)h 4(ihuahua)
                      5: (2)a  h 5(ua)
                      8: (1)a 00 0(0)
```

$\Downarrow$

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

Analyse

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

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

   Tripel sortieren: $\mathrm{O}(n)$

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

   Lexikographisches benenennen: $\mathrm{O}(n)$ (scan)

   Rekursive Instanz konstruieren: $\mathrm{O}(n)$ (forall names do ...)

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

   (1 Radix-Sort Durchgang)

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

   (gewöhnliches Mischen mit merkwürdiger Vergleichsfunktion)

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

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

Implementierung: Vergleichs-Operatoren

```
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));
}
```

Implementierung: Radix-Sortieren

```
// 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;
}
```

Implementierung: Tripel Sortieren

```
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);
```

Implementierung: Lexikographisches Benennen

```
// 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
```

Implementierung: Rekursion

```
// 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;
```

Implementierung: Sortieren der mod 0 Suffixe

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

Implementierung: Mischen

```
  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; }
```

Verallgemeinerung: Differenzenüberdeckungen

Ein Differenzenüberdeckung D modulo v ist eine Teilmenge von $[0,v)$,
so dass $\forall i \in [0,v) : \exists j,k \in D : i \equiv k - j \pmod{v}$.

Beispiel:

$\{1,2\}$ ist eine Differenzenüberdeckung modulo 3.

$\{1,2,4\}$ ist eine Differenzenüberdeckung modulo 7.

- ☐ Führt zu platzeffizienterer Variante
- ☐ Schneller für kleine Alphabete

Verbesserungen / Verallgemeinerungen

- tuning
- größere Differenzenüberdeckungen
- Kombiniere mit den besten Alg. für einfache Eingaben

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

Suffixtabellenkonstruktion: Zusammenfassung

- einfache, direkte, Linearzeit für Suffixtabellenkonstruktion
- einfach anpassbar auf fortgeschrittene Berechnungsmodelle
- Verallgemeinerung auf Diff-Überdeckungen ergibt platzeffiziente Implementierung

Suche in Suffix Arrays

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

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 $[s, l]$

- Zeit $O(m \log n)$ (geht auch: $O(m + \log n)$)

LCP-Array

speichert Längen der längsten gemeinsamen Präfixe lexikographisch benachbarter Suffixe!

$S =$ banana:

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

SA =

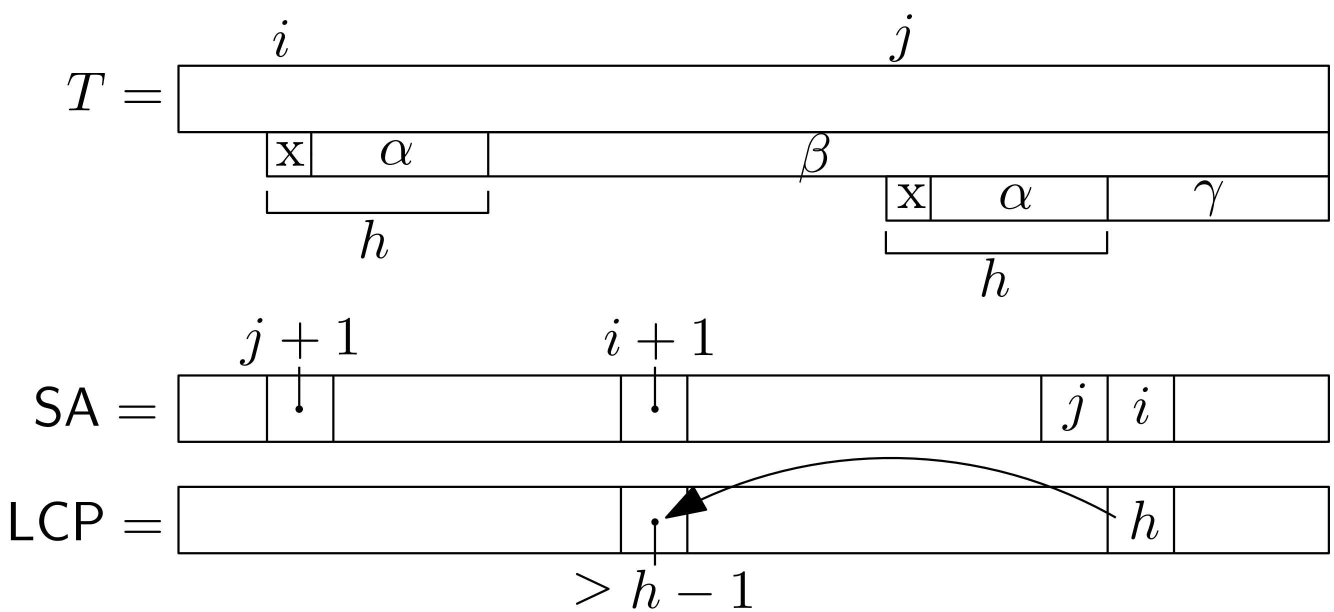
5	`a`
3	`ana`
1	`anana`
0	`banana`
4	`na`
2	`nana`

LCP =

0	`a`
1	**`a`**`na`
3	**`ana`**`na`
0	`banana`
0	`na`
2	**`na`**`na`

LCP-Array: Berechnung

- naiv $O(n^2)$
- inverses Suffix-Array: $\mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$ (wo steht 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: Berechnung

- 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** $h{+}{+}$

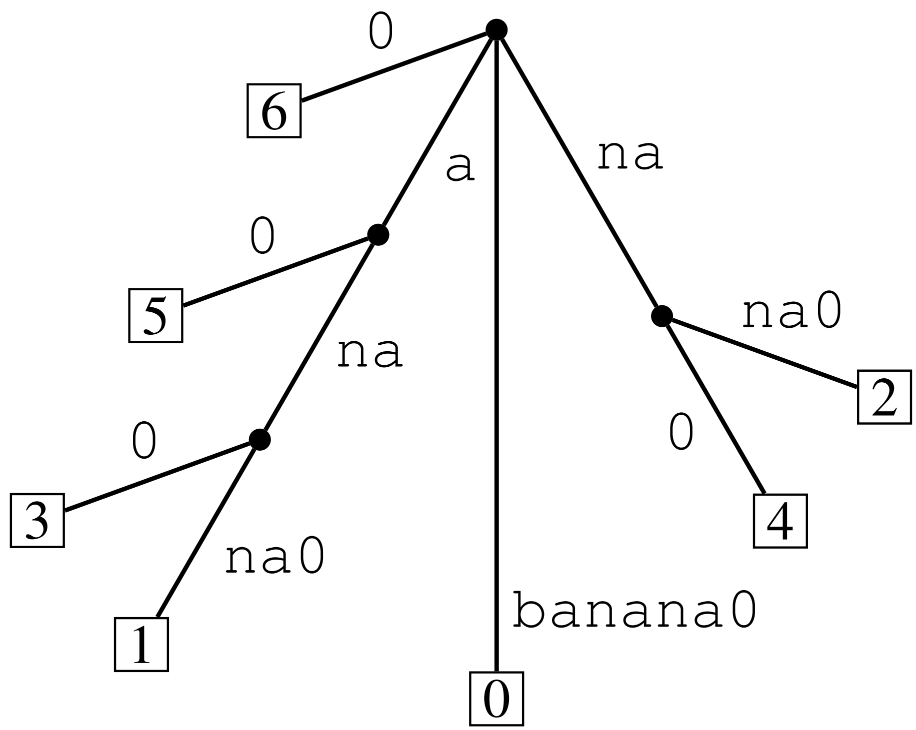
    $\mathrm{LCP}[\mathrm{SA}^{-1}[i]] := h$

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

- Zeit: $O(n)$

Suffix-Baum aus SA und LCP

$S = \texttt{banana0}$



- naiv: $O(n^2)$
- mit Suffix-Array: in lexikographischer Reihenfolge

Suffix-Baum aus SA und LCP

- LCP-Werte helfen!
- Betrachte nur rechtesten Pfad!
- Finde tiefsten Knoten mit String-Tiefe $\leq \mathrm{LCP}[i] \rightsquigarrow$ Einfügepunkt!

	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

- Zeit $O(n)$

Suche in Suffix-Bäumen

- Suche (alle/ein) Vorkommen von $P_{1..m}$ in T:
- Wenn ausgehende Kanten Arrays der Größe $|\Sigma|$:
  - $O(m)$ Suchzeit
  - $O(n|\Sigma|)$ Gesamtplatz
- Wenn ausgehende Kanten Arrays der Größe prop. zur #Kinder:
  - $O(m \log |\Sigma|)$ Suchzeit
  - $O(n)$ Platz

Datenkompression

Problem: bei naiver Speicherung verbrauchen Daten sehr viel Speicherplatz / Kommunikationsbandbreite. Das läßt sich oft reduzieren.

Varianten:

- Verlustbehaftet (mp3, jpg, . . .) / **Verlustfrei** (Text, Dateien, Suchmaschinen, Datenbanken, medizin. Bildverarbeitung, Profifotografie, . . .)
- **1D** (text, Zahlenfolgen,. . .) / 2D (Bilder) / 3D (Filme)
- **nur Speicherung** / mit Operationen ($\leadsto$ succinct data structures)

Verlustfreie Textkompression

Gegeben: Alphabet Σ

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

Textkompressionsalgorithms $f : S \to f(S)$ mit $|f(S)|$ (z.B. gemessen in bits) möglichst klein.

Theorie Verlustfreier Textkompression

Informationstheorie. Zum Beispiel

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

$p(c) = |\{s_i : s_i = c\}|/n$ die relative Häufigkeit von c ist.

untere Schranke für # bits pro Zeichen falls Text einer Zufallsquelle entspränge.

$\rightsquigarrow$ Huffman-Coding ist annähernd optimal! (Entropiecodierung) ????

Schon eher:

Entropie höherer Ordnung betrachte Teilstrings fester Länge

“Ultimativ”: Kolmogorov Komplexität. Leider nicht berechenbar.

Theorie Verlustfreier Textkompression

Entropie höherer Ordnung: Gegeben ein Text S der Länge n über dem Alphabet Σ. Wir definieren $N(\omega, S)$ als Konkatenation aller Zeichen, die in S auf Vorkommen von $\omega \in \Sigma^k$ folgen. Wir definieren die empirische Entropie der Ordung k wie folgt:

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

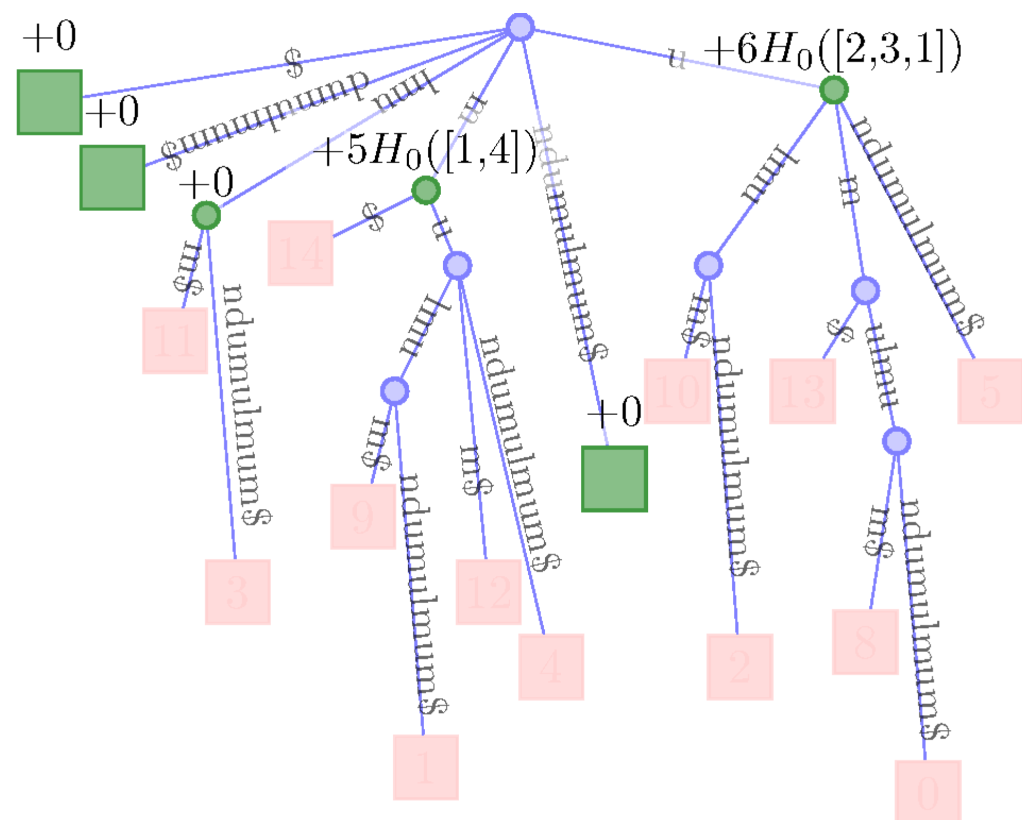
Beispiel: $S = \texttt{ananas}, k = 2$

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

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

Theorie Verlustfreier Textkompression

H_k: Berechnung mittels Suffixbaumes (Beispiel: H_1)



Theorie Verlustfreier Textkompression

Werte der empirischen Entropie in der Praxis

$H_k(S)$ in bits und (# eindeutiger Kontexte/$|S|$ in Prozent)

k	dblp.xm		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

Wörterbuchbasierte Textkompression

Grundidee: wähle $\Sigma' \subseteq \Sigma^*$ und ersetze $S \in \Sigma^*$ durch $S' = \langle s'_1, \ldots, s'_k \rangle \in \Sigma'^*$, so dass $S = s'_1 \cdot s'_2 \cdots s'_k$. (mit '$\cdot$'$=$ Zeichenkettenkonkatenation.

Platz $n \lceil \log \Sigma \rceil \to k \lceil \log \Sigma' \rceil$ mit Entropiecodierung der Zeichen aus Σ' sogar kEntropie(S')

Problem: zusätzlicher Platz für Wörterbuch.

OK für sehr große Datenbestände.

Wörterbuchbasierte Textkompression – Beispiel

Volltextsuchmaschinen verwenden oft $\Sigma' :=$ durch Leerzeichen (etc.) separierte Wörter der zugrundeliegenden natürlichen Sprache.

Spezialbehandlung von Trennzeichen etc.

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

$\rightarrow$ $\boxed{\text{gallia}}$ $\boxed{\text{est}}$ $\boxed{\text{omnis}}$ $\boxed{\text{divisa}}$ $\boxed{\text{in}}$ $\boxed{\text{partes}}$ $\boxed{\text{tres}}$...

Lempel-Ziv Kompression (LZ)

Idee: baue Wörterbuch “on the fly” bei Codierung und Decodierung.

Ohne explizite Speicherung!

Naive Lempel-Ziv Kompression (LZ)

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 Dekompression

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**

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

else

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

output decode(c_i)

LZ Beispiel: 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-Verfeinerungen

- Wörterbuchgröße begrenzen, z.B. $|D| \leq 4096 \rightsquigarrow$ 12bit codes.
- Von vorn wenn Wörterbuch voll $\rightsquigarrow$ Blockweise arbeiten
- Kodierung mit variabler Zahl Bits (z.B. Huffman, arithmetic coding)
- Selten benutzte Wörterbucheinträge löschen ???
- Wörterbuch effizient implementieren:

  (universelles) hashing

LCP zwischen beliebigen Suffixen

- Wie berechnet man die Länge des längste gemeinsame Präfix zweier Suffixe S_x und S_y mit $(x \neq y)$?
- Falls LCP Array schon vorliegt:
  - Sei $\ell = min(SA^{-1}[x], SA^{-1}[y])$ und $r = max(SA^{-1}, SA^{-1}[y])$
  - $LCP(S_x, S_y) = min_{\ell < i \leq r}\{LCP[i]\}$
  - Frage: Wie bestimmt man die Position des Minimums in einem Array A effizient?

Motivation für Range-Minimum-Query (RMQ) Datenstrukturen.