

14 Algorithmen für Zeichenketten

hier nur ein Beispiel:

sehr schnelle Volltextsuche

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: $O(nm)$

P vorverarbeiten: $O(n + m)$ (siehe Vordiplom, endl. Automaten, KMP, Boyer Moore)

S vorverarbeiten: Textindizes. Erstes Vorkommen:

Invertierter Index: siehe Einführungsvorlesung

Suffix Array: $O(m \log n) \dots O(m)$

Suffixtabellen

aus

Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

Prüfungsrelevant: Abschnitte 2, 3

Etwas “Stringology”-Notation

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

von Buchstaben

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

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

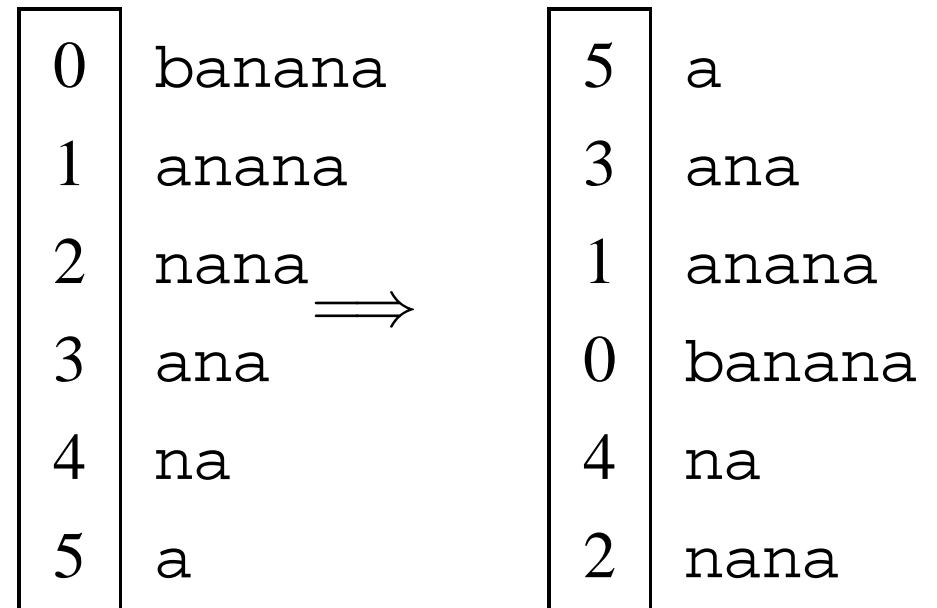
0 ist kleiner als alle anderen Zeichen

Suffixe Sortieren

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

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

$S = \text{banana}$:



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: $O(m \log n)$ gut für kurze Muster

Binäre Suche mit lcp: $O(m + \log n)$ falls wir die
längsten gemeinsamen (common) Präfixe
zwischen verglichenen Zeichenketten vorberechnen

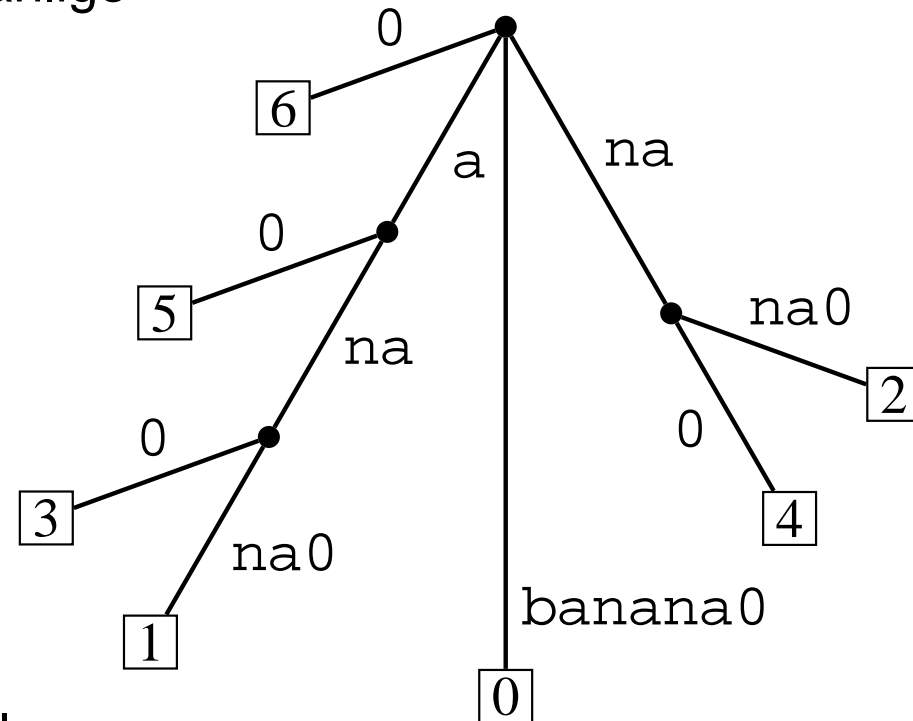
Suffix-Baum: $O(m)$ kann aus **SA berechnet werden**

Suffix-Baum

[Weiner '73][McCreight '76]

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

$S = \text{banana0}$



Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur **verglichen** werden

Konstante Alphabetgröße: endliche Menge
deren Größe nicht von n abhängt.

Ganzzahliges Alphabet: Alphabet ist $\{1, \dots, \sigma\}$
für eine ganze Zahl $\sigma \geq 2$



Geordnetes \rightarrow ganzzahliges Alphabet

Sortiere die Zeichen von S

Ersetze $S[i]$ durch seinen Rang

012345 125024

banana \rightarrow aaabnn

213131 \leftarrow 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. $SA^1 = \text{sort} \{S_i : i \text{ ist ungerade}\}$ (Rekursion)
2. $SA^0 = \text{sort} \{S_i : i \text{ ist gerade}\}$ (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.

banana \rightarrow anana

- Ersetze Buchstabenpaare durch Ihre **lexikographischen Namen**

an	an	a0
----	----	----

 \rightarrow 221

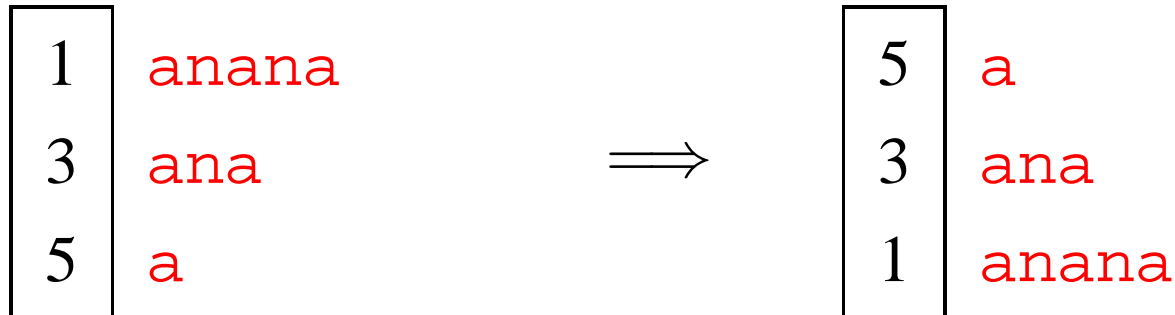
- Rekursion

$\langle 1, 21, 221 \rangle$

- Rückübersetzen

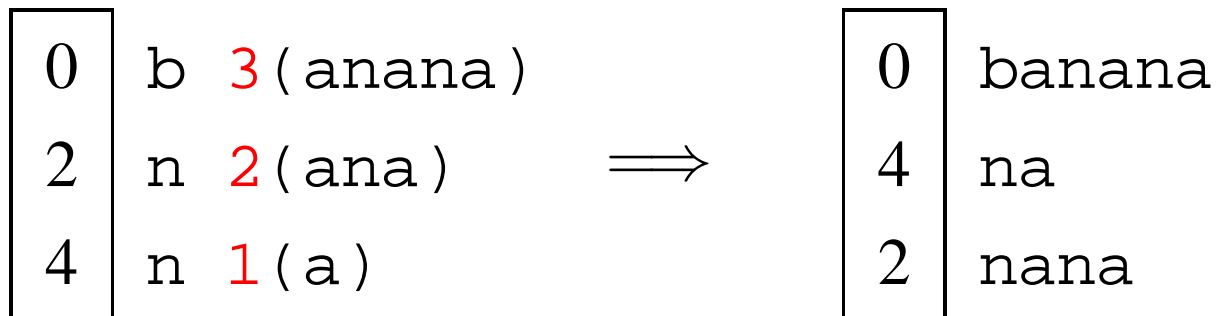
$\langle a, ana, anana \rangle$

Berechne SA^0 aus SA^1



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



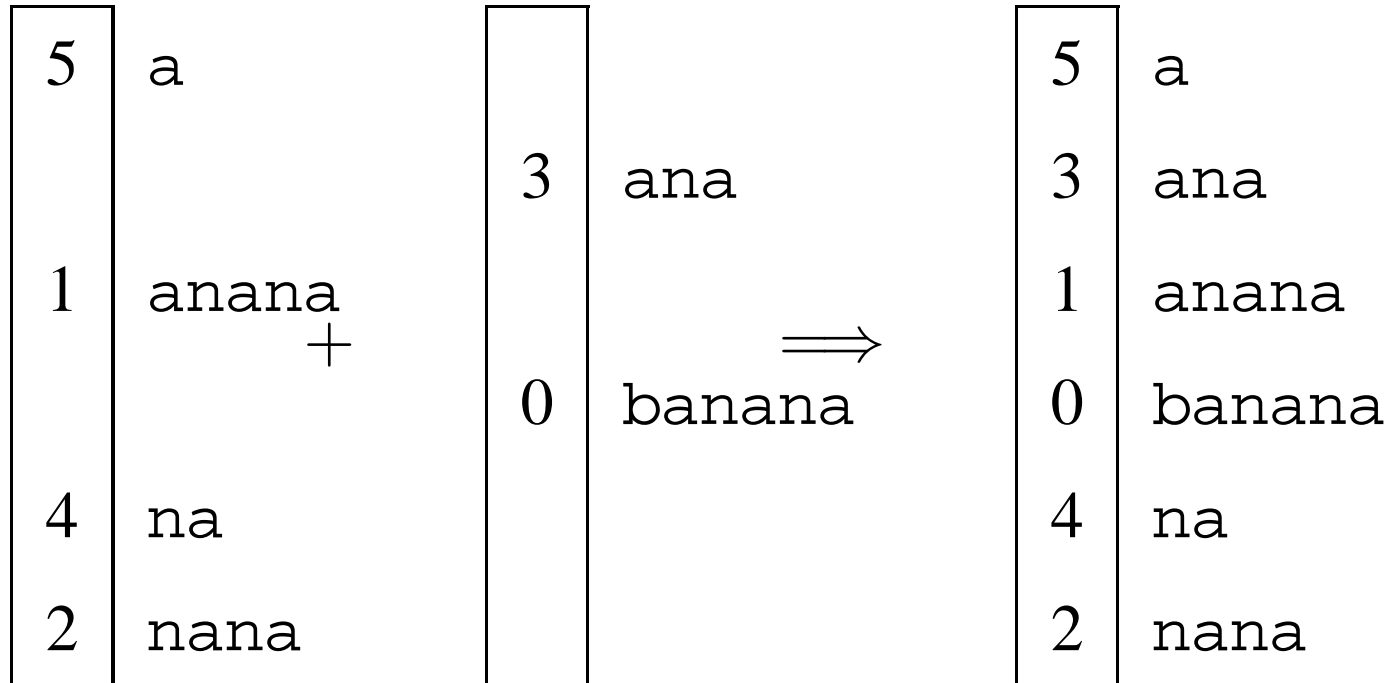
Radix-Sort



Asymmetrisches Divide-and-Conquer

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

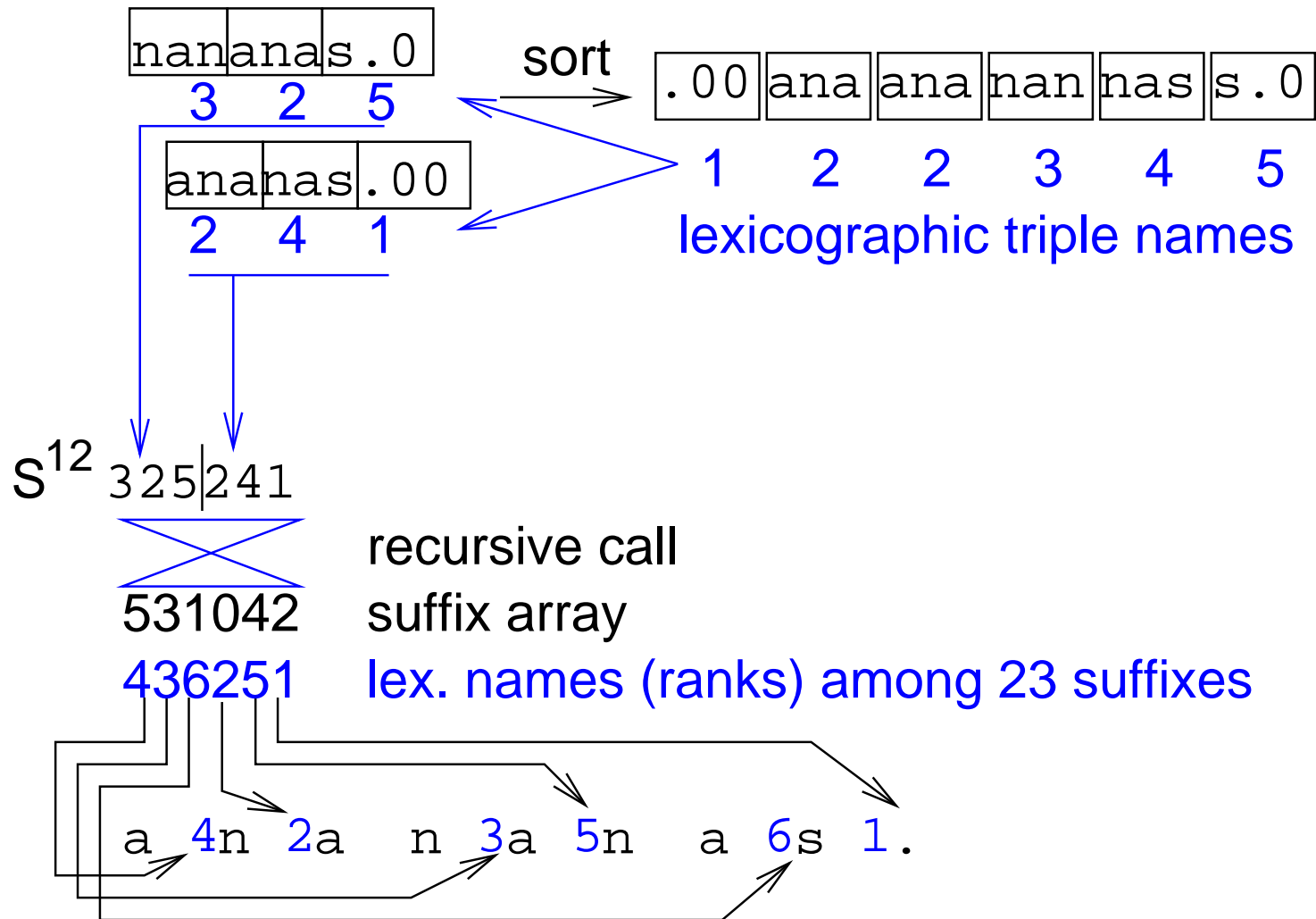
$S = \text{banana}$



Rekursion, Beispiel

012345678

S anananas.





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_i^{12} von S^{12} repräsentiert S_{3i+1}
Suffix $S_{n/3+i}^{12}$ von S^{12} repräsentiert S_{3i+2}
- **Rekursion** auf (S^{12}) (Alphabetgröße $\leq 2n/3$)
- Annotiere die 23-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 $\text{key}(a[i]) \in [0..n]$

$c[0..n] := [0, \dots, 0]$

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

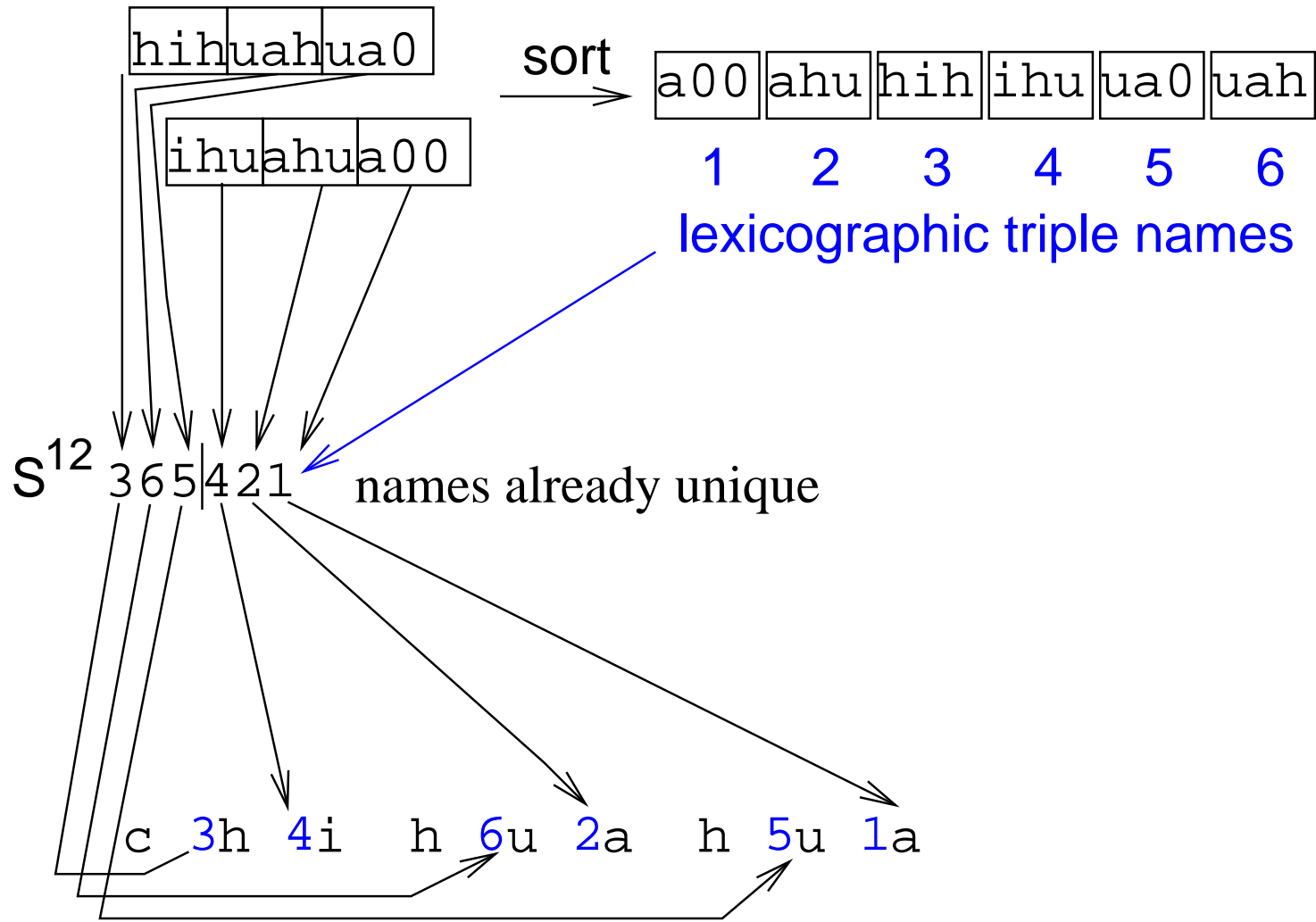
Zeit $O(n)$!



Rekursions-Beispiel: Einfacher Fall

012345678

S chihuahua



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

$0 < 1 \Leftrightarrow c_n < c_n$	4:	$(6)u_2(ahua)$
$0 < 2 \Leftrightarrow cc_n < cc_n$	7:	$(5)u_1(a)$
3: $h_6u_2(ahua)$	2:	$(4)i_h_6(uahua)$
6: $h_5u_1(a)$	1:	$(3)h_4(ihuahua)$
0: $c_3h_4(ihuahua)$	5:	$(2)a_h_5(ua)$
	8:	$(1)a_{00_0}(0)$

⇓

- 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: $O(n)$ (forall $i, i \bmod 3 \neq 0$ do ...)

Tripel sortieren: $O(n)$

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

Lexikographisches benennen: $O(n)$ (scan)

Rekursive Instanz konstruieren: $O(n)$ (forall names do ...)

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

(1 Radix-Sort Durchgang)

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

(gewöhnliches Mischen mit merkwürdiger Vergleichsfunktion)

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

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

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 occurrences
  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 $\forall i \in [0, v) : \exists j, k \in D : i \equiv k - j \pmod{v}$.

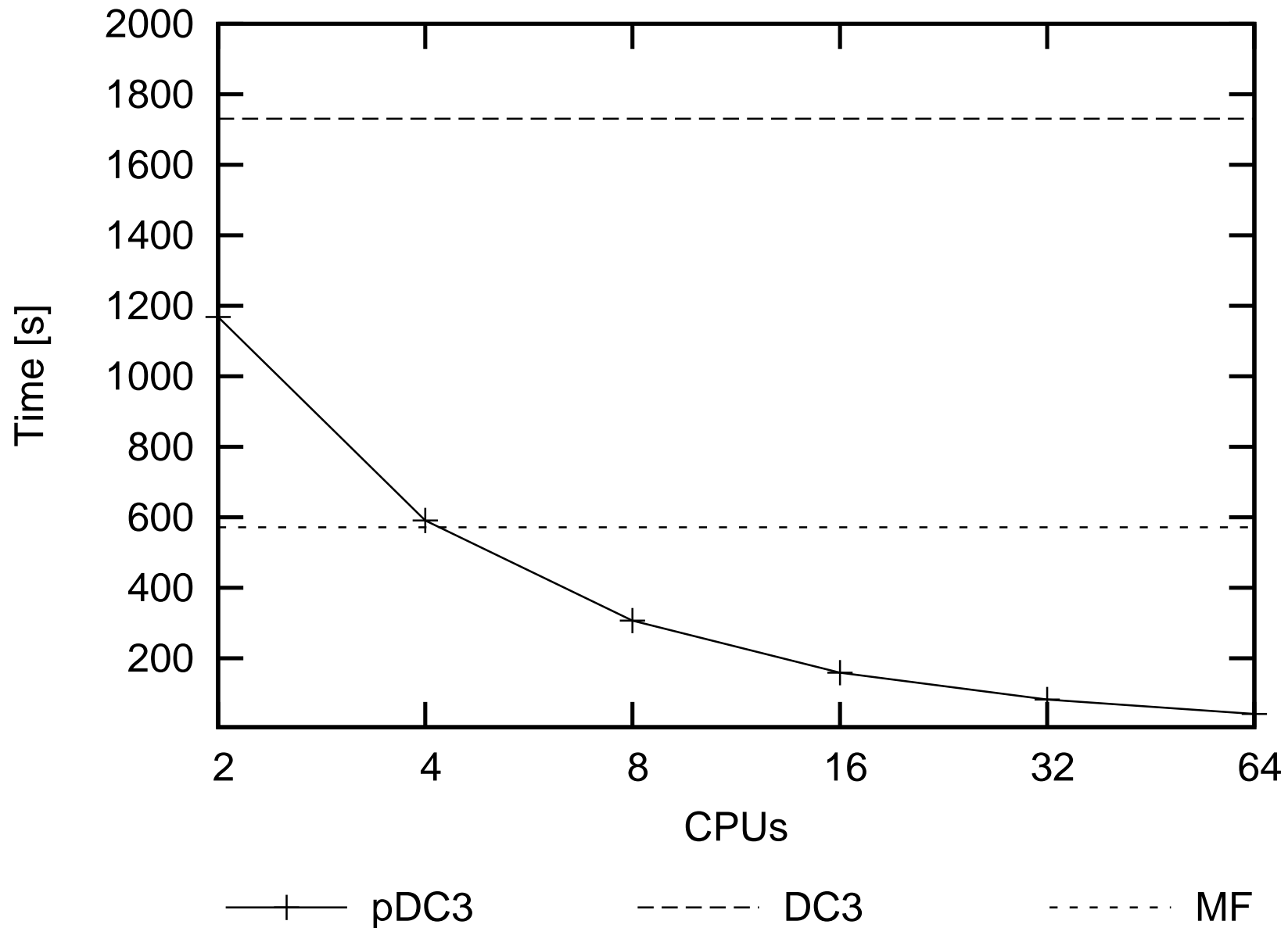
Beispiel:

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

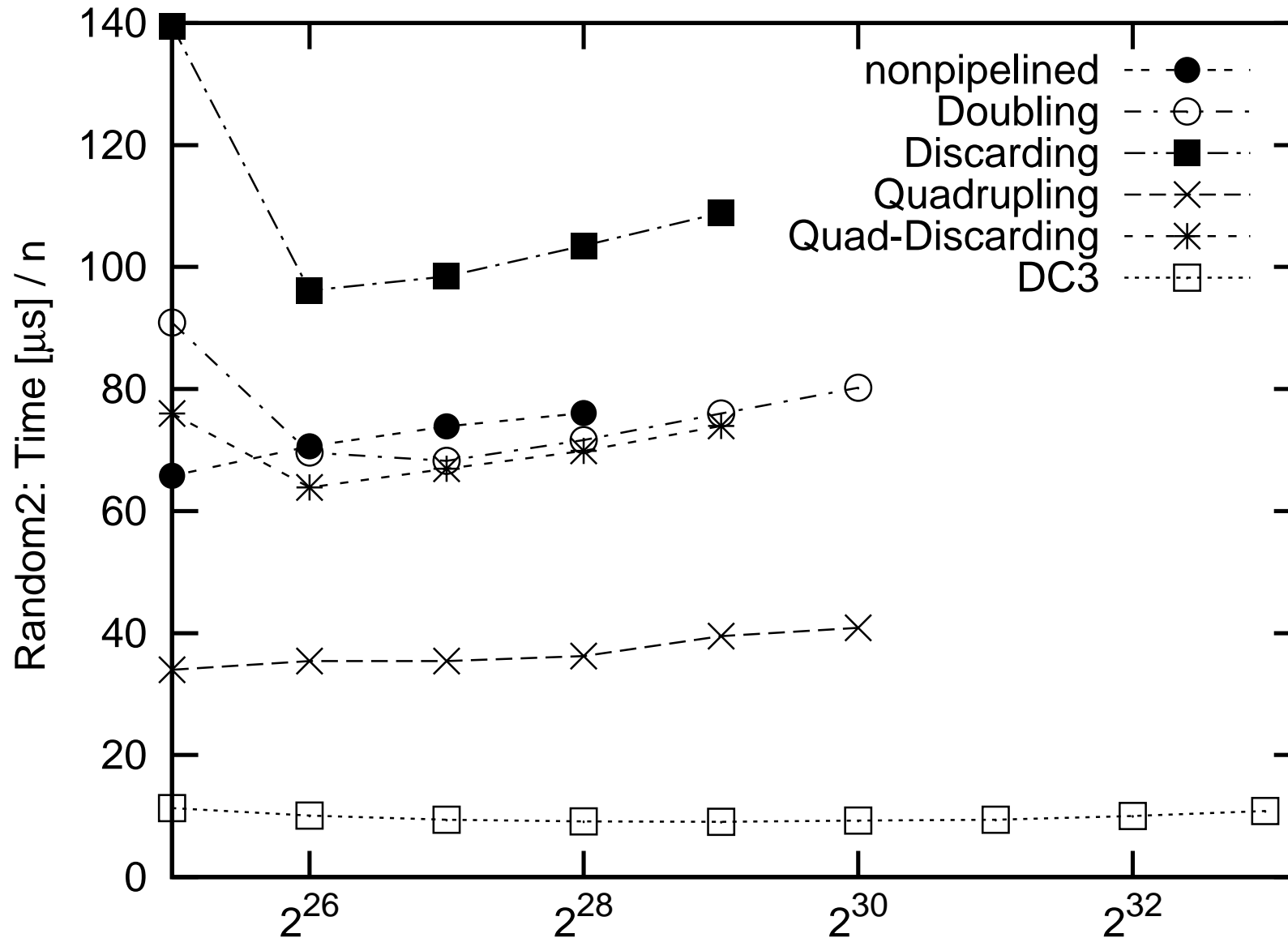
$\{1, 2, 4\}$ ist eine Differenzenüberdeckung modulo 7.

- Führt zu platzeffizienterer Variante
- Schneller für kleine Alphabete

Parallele Implementierung (522 MByte Source-Code)



Externe Implementierung(4 GByte schwierige Zufallseingabe)





Verbesserungen / Verallgemeinerungen

- tuning
- größere **Differenzenüberdeckungen**
- Kombiniere mit den besten Alg. für einfache Eingaben
[Manzini Ferragina 02, Schürmann Stoye 05]

Suffixtabellenkonstruktion: Zusammenfassung

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