Kapitel 4

Indizieren von Texten

Motivation

Es sei T ein langer (und unveränderlicher) Text.
Beispiele: Genom-Sequenzen, große Text-Datenbanken (Internet)

Ziel: Bei der Suche nach einem Muster P soll nicht der ganze Text T betrachtet werden.

Folgerung: Text muss vorverarbeitet (indiziert) werden.

Anforderungen an einen Index:
Suchprobleme deutlich schneller als in $\Theta(|T|)$ lösbar
relativ kompakt, schnell konstruierbar (linear in $|T|$)

Überblick

- Datenstrukturen für die Indizierung
  (Suffixbäume, Suffix-Arrays, q-gramme)
- Konstruktion von Suffixbäumen
- Konstruktion von Suffix-Arrays
- Anwendungen

4.1 Datenstrukturen für die Indizierung

Suffix-Tries

Definition. Es sei $S \in \Sigma^*$ ein Wort mit $|S| = n$ und $\# \notin \Sigma$ ein Sonderzeichen (Textende). Der Suffix-Suchwortbaum (Suffix-Trie) von S ist der Suchwortbaum $Trie(S_1, S_2, \ldots, S_n, S_{n+1})$ mit $S_i = S[i \ldots n]\#$.

Suche nach einem Wort P der Länge m in S entspricht der Suche nach einem Pfad der Länge m im Suffix-Trie von S;
Zeit: $O(m)$ bei bekanntem Suffix-Trie

Problem: Suffix-Trie hat quadratische Größe

Suffixbäume

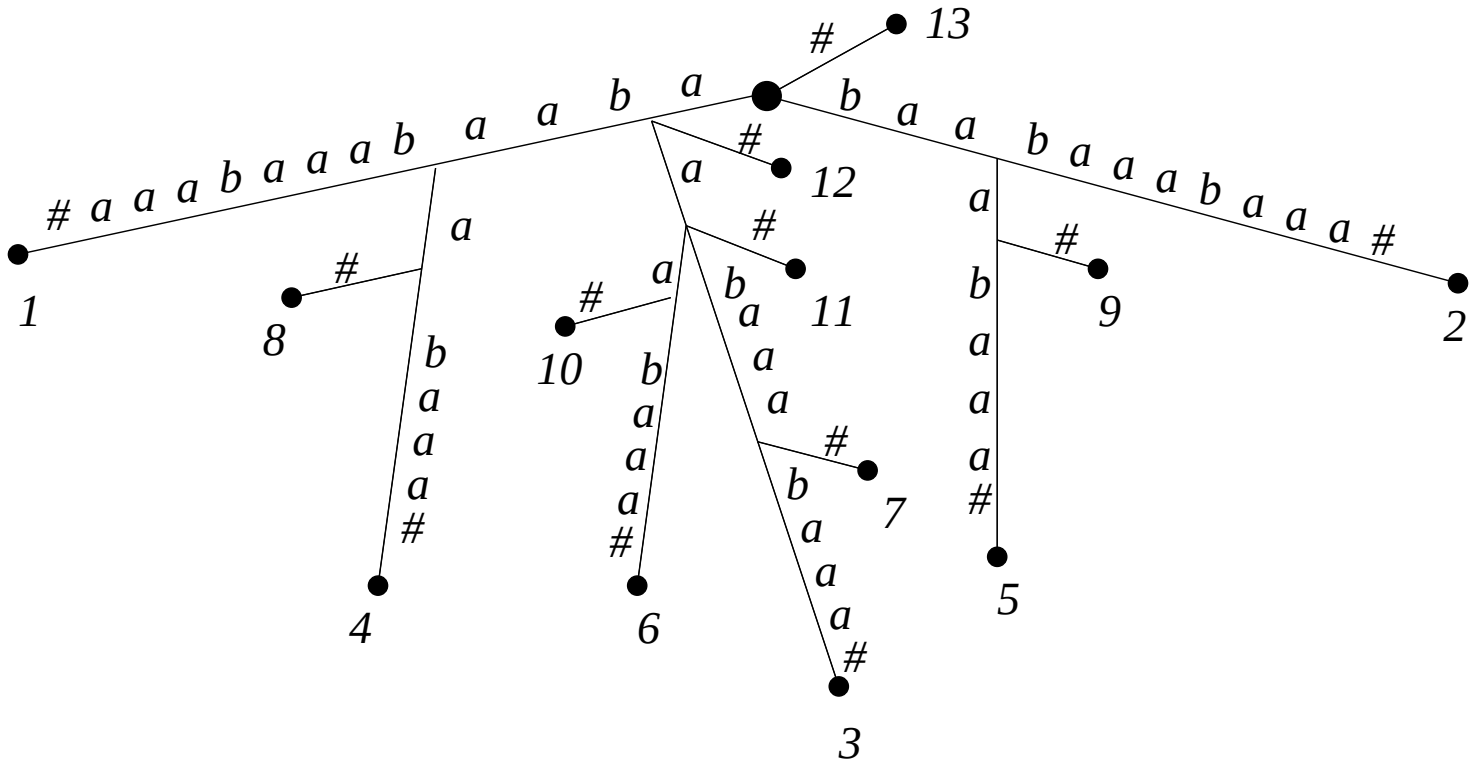
Definition. Es sei $S \in \Sigma^*$ ein Wort und $\# \notin \Sigma$ ein Sonderzeichen (Textende). Der Suffixbaum $\mathcal{T}(S)$ von S entsteht, indem man im Suffix-Trie von S jeden maximalen Weg, dessen innere Knoten einen Ausgangsgrad von 1 besitzen, durch eine Kante mit der gleichen Beschriftung ersetzt.

Lemma. Die Zahl der Knoten von $\mathcal{T}(S)$ ist höchstens $2n$ mit $n = |S|$.

Komprimierte Kantenbeschriftung: Die Beschriftung einer Kante im Suffixbaum ist ein Infix $S[i \ldots j]$ und wird durch $[i, j]$ dargestellt.
$\rightarrow$ Beschriftung einer Kante beansprucht konstanten Platz

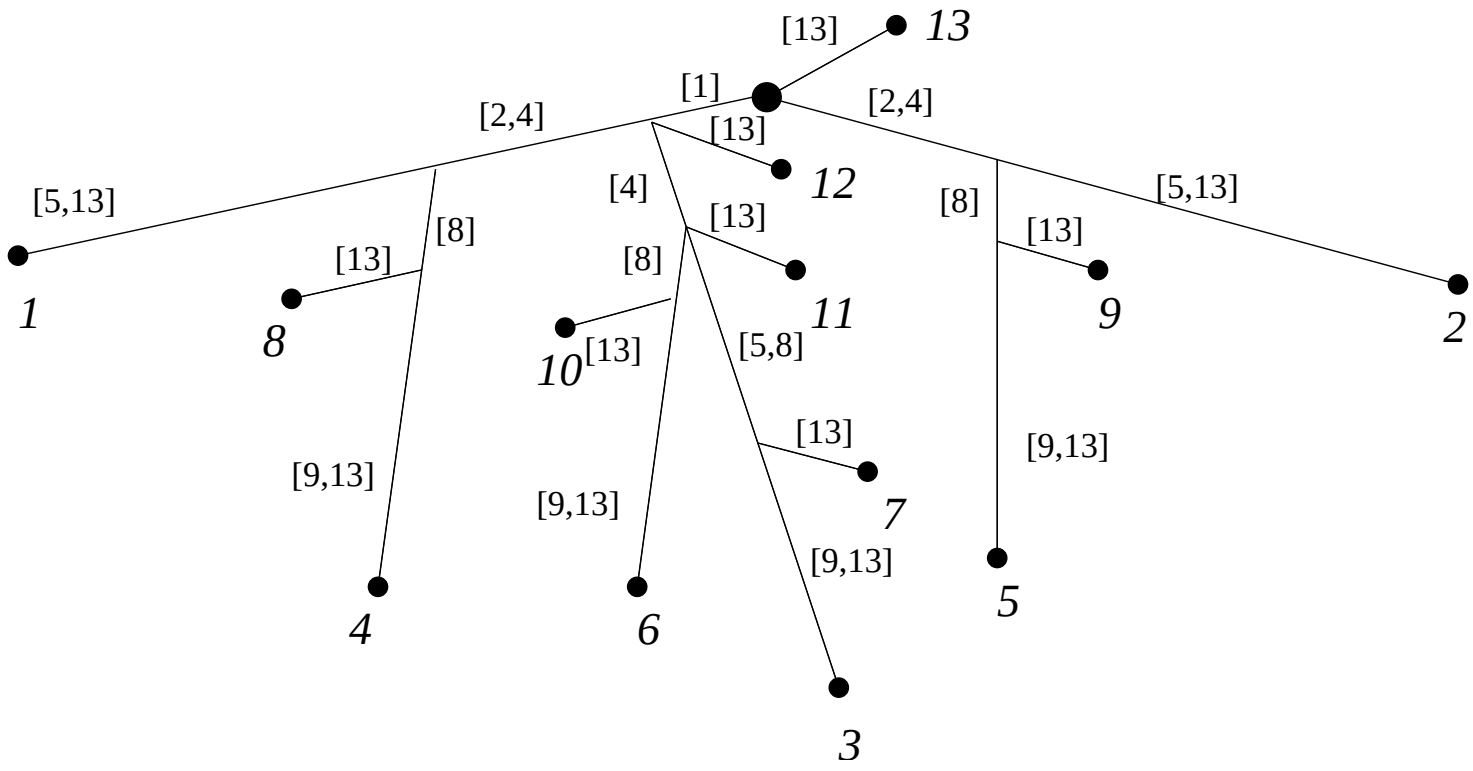
Beispiel

Der Suffixbaum für $S = abaabaaabaaa$ sieht wie folgt aus:



Beispiel

Suffixbaum für $S = abaabaaabaaa$ mit komprimierter Beschriftung (statt $[i, i]$ schreiben wir $[i]$):



Suffix-Arrays

Definition. Es sei $S \in \Sigma^*$ mit $|S| = n$. Auf $\Sigma \cup \{\#\}$ sei eine Ordnung mit dem kleinsten Element $\#$ definiert. Das Suffix-Array von S ist das $(n+1)$-dimensionale Feld A_S mit $A_S[i] = j \in \{1, \ldots, n+1\}$ genau dann, wenn $S[j \ldots n]$ das lexikographisch i-te Suffix ist.

Beispiel. Für $S = mississippi$ gilt $A_S = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$.

Vorteile von Suffix-Arrays: einfache Struktur, geringer Platzbedarf

Suche nach einem Wort P der Länge m in S durch binäre Suche im Suffix-Array
Zeit: $O(m \log n)$ bei bekanntem Suffix-Array (kann verbessert werden)

Array der q-gramme

Definition. Es sei $S \in \Sigma^*$ mit $|S| = n$, $|\Sigma| = \sigma$. Das Array der q-gramme von S ist ein Array über Σ^q, das für $\alpha \in \Sigma^q$ die (geordnete) Liste $L(\alpha)$ der Vorkommen von α in S enthält.

Beispiel. Für $S = abaabaaabaaa$ und $q = 3$ erhalten wir folgendes Array der q-gramme.

α	aaa	aab	aba	abb	baa	bab	bba	bbb
$L(\alpha)$	$6, 10$	$3, 7$	$1, 4, 8$	$\emptyset$	$2, 5, 9$	$\emptyset$	$\emptyset$	$\emptyset$

Suche nach einem Wort $P = \alpha_1\alpha_2 \cdots \alpha_k$ mit $\alpha_i \in \Sigma^q$:
Suche in den Listen $L(\alpha_i)$ nach passenden Indizes.
Laufzeit: $O(\sum_{i=1}^{k} |L(\alpha_i)|)$; im Mittel: $O(k \log_\sigma n)$.

Vorteile von q-grammen:
Vorkommen sind geordnet; noch geringerer Platzbedarf als Suffix-Arrays

Teil-Indizes

Natürlichsprachige Texte bestehen aus Wörtern und Trennzeichen.
Gesucht sind in der Regel Folgen von Wörtern.

Der Index muß nur Positionen im Text erfassen, an denen ein Wort beginnt.
(z.B. partielles Suffix-Array, partieller Suffixbaum) $\rightarrow$ große Platzerparnis

Beispiel.

```
brautkleid bleibt brautkleid und blaukraut bleibt blaukraut
1          12     18          29  33         43     49
```

Wörter in lexikografischer Reihenfolge:
`blaukraut, bleibt, brautkleid, und`

Partielles Suffix-Array: $(49, 33, 43, 12, 1, 18, 29)$

Teil-Indizes sind auch für molekularbiologische Datenbanken anwendbar,
z.B. für DNA-Daten: Index nur an Positionen mit Symbol G.

Invertierter Index

Geeignet für natürlichsprachige Texte.
Jedem Wort wird die Liste seiner Vorkommen zugeordnet.

Beispiel.

```
brautkleid bleibt brautkleid und blaukraut bleibt blaukraut
1          12     18          29  33         43     49
```

`blaukraut`: 33, 49
`bleibt`: 12
`brautkleid`: 1, 18
`und`: 29

Häufig als Index über mehrere Dokumente;
oft nur Zuordnung der Dateien oder von Speicherbereichen (Blocks)

4.2 Konstruktion von Suffix-Bäumen

Gegeben: $S \in \Sigma^*$; $|S| = n$.
Gesucht: Suffixbaum von S

- Naiver Algorithmus: $O(n^2)$ Schritte im schlechtesten Fall, $O(n \log n)$ Schritte im mittleren Fall

- Algorithmus von McCreight: Verbesserung des Naiven Algorithmus
  Zeit: $O(n)$

- weitere Linearzeit-Algorithmen
  - Algorithmus von Weiner (historisch erster Linearzeit-Algorithmus)
  - Algorithmus von Ukkonen (Online-Variante des McCreight-Algorithmus)

Positionen im Suffixbaum

Position: entweder Knoten v oder
Paar (e, i) mit Kante $e = (v, \alpha, w)$ und $0 < i < |\alpha|$ (auf einer Kante)

Identifizierung einer Position p durch Beschriftung $\mathcal{L}(p)$ des Pfades von der Wurzel
Sprechweise: Position $\mathcal{L}(p)$

Nachfolger einer Position: Gilt $\mathcal{L}(p) = \beta$ und $\mathcal{L}(q) = \beta x$ mit $x \in \Sigma$, so nennt man q einen Nachfolger von p. Notation: $q = Next(p, x)$.

Tiefe einer Position p: graphentheoretische Tiefe (Anzahl der Kanten von der Wurzel).
Notation: $depth(p)$.

String-Tiefe einer Position p: Länge der Beschriftung $\mathcal{L}(p)$ (Tiefe im Suffix-Trie).
Notation: $strdepth(p)$.

Algorithmus 4.1 Konstruktion des Suffixbaumes (Naiver Algorithmus)

Eingabe: Wort S mit $|S| = n$
Ausgabe: Suffixbaum von S

(1) $\mathcal{T} \leftarrow (\{root\}, \emptyset)$;
(2) **for** $i \leftarrow 1$ **to** $n + 1$
(3) $\quad t \leftarrow i$; $p \leftarrow root$; $x \leftarrow S[t]$;
(4) $\quad$ **while** ($Next(p, x)$ existiert)
(5) $\quad\quad p \leftarrow Next(p, x)$; $t \leftarrow t + 1$; $x \leftarrow S[t]$;
(6) $\quad$ Füge bei p eine neue Kante mit der Beschriftung $S[t \ldots n + 1]$ zu einem neuen Blatt mit der Beschriftung i ein.
(7) **return** $\mathcal{T}$

Definition. $S[n + 1] := \#$

Laufzeit: $\Theta(n^2)$ im schlechtesten, $\Theta(n \log n)$ im mittleren Fall.

Einfügen einer Kante (bei komprimierter Beschriftung)

p sei die Position, die in Schritt 6 erreicht ist.

1. Fall: p ist ein Knoten u.
$\rightarrow$ neues Blatt b, neue Kante (u, b) mit Beschriftung $[t, n+1]$.

2. Fall: p ist im Inneren einer Kante, d.h. $p = ((v, w), r)$;
Beschriftung von (v, w) sei $[j, k]$.
$\rightarrow$ neuer Knoten u an der Position p.
Kante (v, w) wird ersetzt durch
Kante (v, u) mit Beschriftung $[j, j + r - 1]$ und
Kante (u, w) mit Beschriftung $[j + r, k]$.
neues Blatt b, neue Kante (u, b) mit Beschriftung $[t, n+1]$.

Algorithmus von McCreight

- fügt wie der Naive Algorithmus die Suffixe $S_1, S_2, \ldots, S_{n+1}$ ein
- Laufzeit wird verkürzt durch Nutzung zweier Heuristiken (Skip/Count-Trick und Suffix-Links)
- Laufzeit: $O(n)$.

Lemma. Gibt es vor dem Einfügen des Suffixes S_i eine Position $x\beta$ mit $x \in \Sigma, \beta \in \Sigma^*$, so gibt es nach dem Einfügen von S_i eine Position β.

Gibt es vor dem Einfügen des Suffixes S_i einen Knoten $x\beta$ mit $x \in \Sigma, \beta \in \Sigma^*$, so gibt es nach dem Einfügen von S_i einen Knoten β.

Skip/Count-Trick

Wenn im Voraus sicher ist, dass im Knoten v ein Pfad β beginnt, so braucht man bei der Suche nach der Endposition des Pfades nicht für jede Position, sondern nur für die Knoten auf dem Pfad nach der Fortsetzung zu suchen.

Gilt $\beta = \varepsilon$, so ist v die Endposition des Pfades.

Gilt $\beta \neq \varepsilon$, so bestimmt man die eindeutige Kante $e = (v, \alpha, w)$ mit $\alpha[1] = \beta[1]$ und läßt die $|\alpha|$ Vergleiche entlang der Kante aus (skip).

Gilt $|\alpha| > |\beta|$, so ist die Endposition $(e, |\beta|)$.

Gilt $|\alpha| \leq |\beta|$, so sucht man vom Knoten w nach $\beta[|\alpha| + 1, |\beta|]$ (count).

Skip/Count-Suche

Eingabe: Baum $\mathcal{T}$ mit Kantenbeschriftungen aus Σ^*, Knoten v,
$\beta \in \Sigma^*$, $|\beta| = m$, Pfad β ab v existiert

Ausgabe: Endposition des Pfades β ab v

SKIP-COUNT-SEARCH$(\mathcal{T}, v, \beta)$

(1) **if** $m = 0$ **then return** v;
(2) $t \leftarrow 1$; $u \leftarrow v$;
(3) **while** $t \leq m$
(4) Bestimme Kante $e = (u, \alpha, w)$ mit $\alpha[1] = \beta[t]$;
(5) **if** $t + |\alpha| = m + 1$ **then return** w;
(6) **if** $t + |\alpha| > m + 1$ **then return** $(e, m - t + 1)$;
(7) **if** $t + |\alpha| \leq m$ **then** $u \leftarrow w$; $t \leftarrow t + |\alpha|$;

Zeit für die Skip/Count-Suche nach dem Ende des Pfades β:
linear in graphentheoretischer Länge des Pfades (statt linear in $|\beta|$).

Suffix-Links

Definition. Es sei v ein Knoten im Suffixbaum mit der Beschriftung $x\alpha$, $x \in \Sigma, \alpha \in \Sigma^*$. Gibt es einen Knoten w mit der Beschriftung α, so nennen wir w das Suffix-Link von v, Bezeichnung $s[v]$.

Folgerung. Jeder Knoten der vor dem Einfügen des Suffixes S_i vorhanden ist, besitzt nach dem Einfügen von S_i ein Suffix-Link.

Algorithmus von McCreight – Einfügen von S_i

u: Knoten, in dem beim Einfügen von S_{i-1} die Kante zum neuen Blatt eingefügt wurde (Beschriftung dieser Kante: $[t, n+1]$)

v: letzter Knoten auf dem Weg nach u, der vor dem Einfügen von S_{i-1} vorhanden war
$u \neq v \iff u$ neu eingefügt.

1. Fall: u ist die Wurzel.
Füge S_i wie beim Naiven Algorithmus ein.

2. Fall: v ist nicht die Wurzel.
$\beta \leftarrow$ Beschriftung der Kante von v nach u; $p \leftarrow S[v]$.

3. Fall: v ist die Wurzel, u ist nicht die Wurzel.
$\beta \leftarrow$ Beschriftung der Kante von v nach u ohne das erste Zeichen. $p \leftarrow v$.

2. und 3. Fall: Verfolge von p aus unter Nutzung des Skip/Count-Tricks den Pfad β.
Die Endposition dieses Pfades sei q. Setzen des Suffix-Links: $s[u] \leftarrow q$.
Suche von q aus wie beim Naiven Algorithmus den Pfad $S[t \ldots n+1]$.

Beispiel

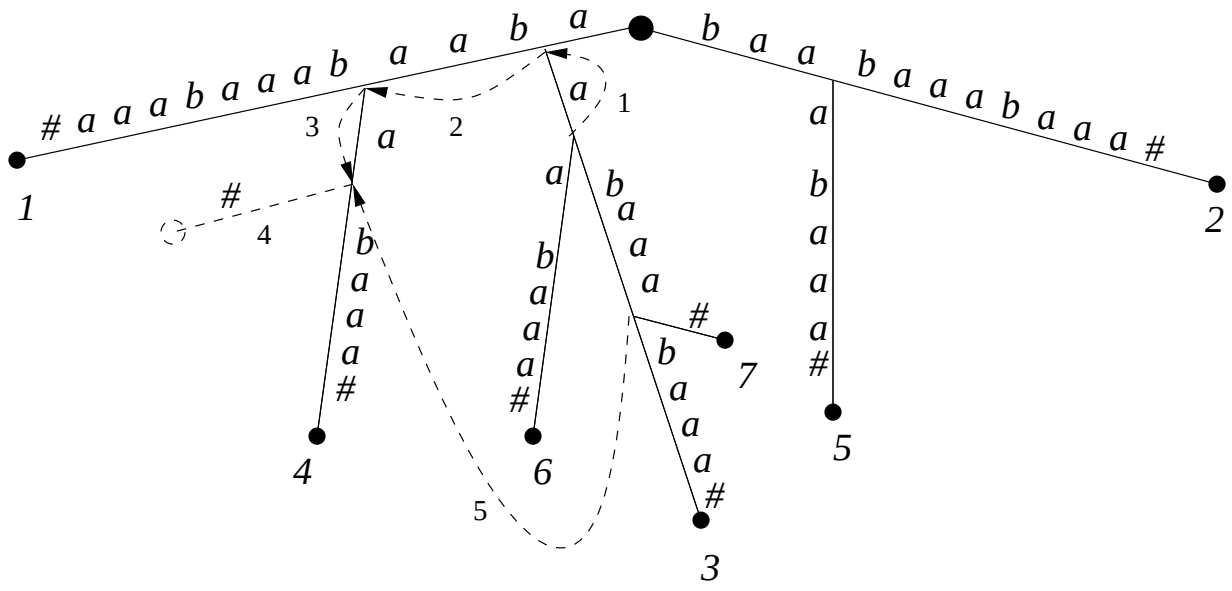
$S = abaabaaabaaa$. Betrachte Einfügen von $S_8 = abaaa$. $u = aabaaa$; $v = aa$; $\beta = baaa$;

1: Folge dem Suffix-Link nach $s[v]$.

2-3: Suche von $s[v]$ nach $baaa$ mittels Skip/Count-Suche.

4: Die Suche nach dem Wort $\#$ ist erfolglos; es werden ein neuer innerer Knoten, ein neues Blatt und eine Kante eingefügt.

5: Es wird das Suffix-Link vom Knoten u gesetzt.



Algorithmus 4.2 Konstruktion des Suffixbaumes (McCreight)

Eingabe: Wort S mit $|S| = n$
Ausgabe: Suffixbaum von S

(1) $\mathcal{T} \leftarrow (\{root\}, \emptyset)$; $v \leftarrow root$; $u \leftarrow root$;
(2) **for** $i \leftarrow 1$ **to** $n + 1$
(3) **if** $u = v$ **then** $\beta \leftarrow \varepsilon$;
(4) **else** $\beta \leftarrow$ Beschriftung der Kante von v nach u;
(5) **if** $u = root$ **then** $t \leftarrow i$;
(6) **else if** $v = root$ **then** $\beta \leftarrow \beta$ ohne erstes Zeichen;
(7) **else** $v \leftarrow s[v]$;
(8) $p \leftarrow$ SKIP-COUNT-SEARCH$(\mathcal{T}, v, \beta)$; $q \leftarrow p$;
(9) **while** ($Next(q, S[t])$ existiert)
(10) $q \leftarrow Next(q, S[t])$; $t \leftarrow t + 1$;
(11) $v \leftarrow$ letzter Knoten auf dem Pfad nach q;
(12) Füge bei q eine neue Kante mit der Beschriftung $S[t \ldots n + 1]$ zu einem neuen Blatt mit der Beschriftung i ein;
(13) **if** $u \neq root$ **then** $s[u] \leftarrow p$;
(14) $u \leftarrow q$;
(15) **return** $\mathcal{T}$;

Algorithmus von McCreight – Laufzeit

Lemma. Für jeden Knoten v, der ein Suffix-Link besitzt, gilt (zu jedem Zeitpunkt der Konstruktion)

$$depth(s[v]) \geq depth(v) - 1.$$

Satz. Der Algorithmus von McCreight ermittelt den Suffix-Baum von S mit einer Laufzeit von $\Theta(n)$.

Gemeinsame Suffixbäume für mehrere Wörter

gegeben: Wörter $S_1, S_2, \ldots, S_k$ mit $|S_j| = n_j$

Gemeinsamer Suffixbaum enthält alle Suffixe von $S_j\#$ für $1 \leq j \leq k$.
Blattbeschriftung (j, i) entspricht Pfad $S_j[i \ldots n_j]\#$.

Konstruktion des gemeinsamen Suffixbaumes durch sukzessives Einfügen der Wörter $S_1, S_2, \ldots, S_k$ – Verallgemeinerung des McCreight-Algorithmus möglich

Satz. Der gemeinsame Suffixbaum von $S_1, S_2, \ldots, S_k$ mit $|S_j| = n_j$ kann mit einem Aufwand von $O(\sum_{j=1}^{k} n_j)$ bestimmt werden.

Wotd-Algorithmus von Giegerich, Kurtz, Stoye

- Bei langen Texten passen Daten für den Suffixbaum nicht in den Hauptspeicher. Bei Konstruktion und Suche sind Zugriffe auf Sekundärspeicher (z.B. Festplatte) nötig.

- Wichtigstes Effizienzkriterium nicht mehr absolute Zahl der Schritte, sondern Zahl der Speicherzugriffe und benötigter Speicherplatz.

- Effiziente Algorithmen berücksichtigen Lokalität des Suffixbaumes (benachbarte Knoten auch im Speicher benachbart, bei Konstruktion Nachbarschaft "abarbeiten").

- Wotd-Algorithmus (write once top down) konstruiert Baum von der Wurzel von oben nach unten; dabei 2 Varianten:
  - Kompletter Baum wird konstruiert (eager).
  - Während der wiederholten Suche werden benötigte Teile konstruiert (lazy).

Wotd-Algorithmus: Speicherung des Suffixbaumes

- Speicherung der Knoten als Array $Tree$.
- Geschwisterknoten (mit gleichem Vorgänger) bilden zusammenhängendes Teil-Array von $Tree$; geordnet nach erstem Index der zugehörigen Suffixe.
- Für Blatt v Speicherung einer Zahl (Typ `int`):
  Wert l der impliziten Beschriftung $(l, n+1)$ der Kante nach v.
- Für inneren Knoten v Speicherung zweier Zahlen (Typ `int`):
  Wert l der impliziten Beschriftung (l, r) der Kante nach v;
  Index des ersten Kindes von v.
  (Wert r muss nicht gespeichert werden, denn erstes Kind speichert $r+1$)
- Für jeden Knoten zwei spezielle Bits: (Blatt?, Letztes Kind?)
- Suchvorgang von Knoten v aus: Teste für die Kinder von v der Reihe nach, ob ihre Kante mit dem nächsten Buchstaben beginnt.
  Wenige Zugriffe auf externen Speicher, da Geschwister in $Tree$ benachbart sind.

Wotd-Algorithmus: Konstruktion

- Zusätzliches Array $Suffix$ der Größe n: enthält Indizes der Suffixe
- Knoten v im Suffixbaum entspricht Intervall $[l_v, r_v]$ in $Suffix$
  Wurzel entspricht dem gesamten Array $Suffix$
- Innerer Knoten v ist zunächst nicht ausgewertet. Im Array $Tree$ ist nicht ausgewerteter Knoten v durch l_v, r_v repräsentiert.
- Auswertung eines Knoten v:
  - Entnimm l_v, r_v aus v und setze die endgültigen Werte von v fest:
    1. Zahl: Index an der Stelle l_v in $Suffix$.
    2. Zahl: Erstes freies Feld von $Tree$.
  - Bestimme für die Indizes des Intervalls $[l_v, r_v]$ das längste gemeinsame Präfix lcp. Erhöhe die Indizes um lcp.
  - Sortiere die Indizes des Intervalls $[l_v, r_v]$ nach dem zugehörigen Textbuchstaben mittels Counting Sort.

  - Für Buchstaben a entsteht Teilintervall von $[l_v, r_v]$. Dies entspricht einem Kind u von v; Beschriftung der Kante $v \to u$ beginnt mit a.
    Teilintervall mit dem niedrigsten Index entspricht dem ersten Kind von v.
  - Füge die Kinder von v in die ersten freien Felder von $Tree$ ein (erstes Kind als erstes!):
    Für ein Blatt füge endgültigen Wert ein.
    Innerer Knoten u wird als nicht ausgewertet durch l_u, r_u dargestellt.

- Vollständige Konstruktion des Suffixbaumes (eager construction):
  Werte alle Knoten von oben nach unten (top-down) aus.
  Platzbedarf für $Tree$ nimmt zu; kann durch abnehmenden Bedarf für $Suffix$ gedeckt werden.

- Auswertung eines Knoten nur bei Bedarf (lazy construction):
  Werte während der Suche einen Knoten nur aus, wenn nötig.
  Platzbedarf für $Tree$ bleibt bei moderater Anzahl von Suchvorgängen (z.B. $0.01n$) gering; Array $Suffix$ muss erhalten bleiben.

4.3 Konstruktion von Suffix-Arrays

- Konstruktion aus Suffixbaum
  Aufwand $O(n)$; aber unpraktikabel, da hoher Platzbedarf

- Konstruktion durch Verfeinerung (Manber, Myers 1992)
  Aufwand $O(n \log n)$

- Rekursiver Algorithmus (Kärkkäinen, Sanders 2003)
  Aufwand $O(n)$; geringer Platzaufwand

Konstruktion aus Suffixbaum

Konstruiere Suffixbaum von S, wobei für jeden inneren Knoten die ausgehenden Kanten lexikografisch geordnet sind.

Suffix-Array ergibt sich aus den Blättern in DFS-Reihenfolge.

Zeit: $O(n)$ (wenn Alphabetgröße als Konstante angesehen wird)

Problem: Platzbedarf für Suffixbaum etwa 4mal höher als für Suffix-Array

Konstruktion durch Verfeinerung

Gesucht: Suffix-Array von S, $|S| = n$.

Schritt 0**:** Ordne die Menge $\{i : 1 \leq i \leq n+1\}$ nach dem Symbol $S[i]$.

Schritt k**:** Ordne die Menge $\{i : 1 \leq i \leq n+1\}$ nach dem Teilwort der Länge 2^k an Position i.
(Ordnung nach Schritt k verfeinert die Ordnung nach Schritt $k-1$)

Anzahl der Schritte: $\lceil \log_2 n \rceil$.
Zeit für einen Schritt: $O(n)$ (Verwendung von Radix Sort).

Verfeinerung: Formalisierung

Definition. Es sei $S \in \Sigma^*$ mit $|S| = n$. Wir definieren für $k \geq 1$ die folgenden (von S abhängigen) binären Relationen auf $\{1, \ldots, n+1\}$.

$$\begin{aligned} i \equiv_k j \quad &:\Longleftrightarrow \quad S[i \ldots i+k-1] = S[j \ldots j+k-1] \\ i \prec_k j \quad &:\Longleftrightarrow \quad S[i \ldots i+k-1] <_{lex} S[j \ldots j+k-1] \end{aligned}$$

Jede Relation $\equiv_k$ ist eine Äquivalenzrelation.

Jede Relation $\prec_k$ ist transitiv und antireflexiv; definiert eine Ordnung auf den Äquivalenzklassen von $\equiv_k$.

Für alle k ist $\equiv_{k+1}$ eine Verfeinerung von $\equiv_k$ und $\prec_{k+1}$ eine Verfeinerung von $\prec_k$.

Lemma. Für $S \in \Sigma^*$ mit $|S| = n$ gilt:

1. $i \equiv_{k_1+k_2} j \iff (i \equiv_{k_1} j) \wedge (i + k_1 \equiv_{k_2} j + k_1)$,
2. $i \equiv_n j \iff i = j$,
3. $i \prec_{k_1+k_2} j \iff i \prec_{k_1} j \vee ((i \equiv_{k_1} j) \wedge (i + k_1 \prec_{k_2} j + k_1))$,
4. $i \prec_n j \iff S[i \ldots n] <_{lex} S[j \ldots n]$.

Idee des Algorithmus: Unter Nutzung des Lemmas (Punkte 1,3)

bestimme die Relationen $\equiv_{2^k}$ und $\prec_{2^k}$ (beginnend mit $k = 0$),
bis jede Äquivalenzklasse einelementig ist (wegen 2,4 höchstens bis $k = \lceil \log_2 n \rceil$)

Ordnung von Tupeln und Radix Sort

$(M, <)$ sei eine geordnete Menge

natürliche Ordnung $<_k$ auf M^k:
$(a_1, a_2, \ldots, a_k) <_k (b_1, b_2, \ldots, b_k) : \Longleftrightarrow \exists i(\forall j(j < i \rightarrow a_j = b_j) \wedge a_i < b_i)$

Radix Sort für eine Menge $A \subseteq M^k$, $|A| = n$:

for $i \leftarrow k$ **downto** 1
  $A \leftarrow A$ stabil sortiert nach i-ter Komponente;

Gilt $M = \{0, 1, \ldots, r\}$, so kann für die stabile Sortierung Counting Sort angewendet werden $\rightarrow$ Gesamtaufwand $O(k(n + r))$

Verfeinerung – Details

Definition. Es sei $S \in \Sigma^*$ mit $|S| = n$. Das inverse Suffix-Array von S ist das $(n+1)$-dimensionale Feld $\overline{A_S}$ mit $\overline{A_S}[j] = i$ genau dann, wenn $A_S[i] = j$.

Konstruktion benutzt Arrays

A ($\{1, 2, \ldots, n+1\}$ geordnet nach $\prec_{2^k}$, am Ende A_S) und
$\overline{A}$ (Äquivalenzklasse in $\equiv_{2^k}$ geordnet nach $\prec_{2^k}$, am Ende $\overline{A_S}$)

Konstruktion des Suffix-Arrays durch Verfeinerung: Schritt 0

(1) $A \leftarrow$ RADIX SORT$(\{1, 2, \ldots, n+1\})$ nach Schlüssel $S[j]$;
(2) $\overline{A}[A[1]] \leftarrow 1$;
(3) **for** $i \leftarrow 2$ **to** $n+1$
(4) **if** $S[A[i]] = S[A[i-1]]$ **then** $\overline{A}[A[i]] \leftarrow \overline{A}[A[i-1]]$;
(5) **else** $\overline{A}[A[i]] \leftarrow \overline{A}[A[i-1]] + 1$;

Verfeinerung – Fortsetzung

Konstruktion des Suffix-Arrays durch Verfeinerung: Schritt $k \geq 1$

(1) $A \leftarrow$ RADIX SORT$(\{1, 2, \ldots, n{+}1\})$ nach Schlüssel $(\overline{A}[j], \overline{A}[j{+}2^{k-1}])$;
(2) $B[A[1]] \leftarrow 1$;
(3) **for** $i \leftarrow 2$ **to** $n+1$
(4) **if** $(\overline{A}[A[i]] = \overline{A}[A[i-1]]$ **and** $\overline{A}[A[i]+2^{k-1}] = \overline{A}[A[i-1]+2^{k-1}])$
(5) **then** $B[A[i]] \leftarrow B[A[i-1]]$;
(6) **else** $B[A[i]] \leftarrow B[A[i-1]]+1$;
(7) $\overline{A} \leftarrow B$;

Laufzeit pro Schritt: $O(n)$, da Radix Sort (mit 2 Durchläufen) benutzt wird.

Verfeinerung – Beispiel

$S =$ `mississippi`

Schritt 0:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	2	5	8	11	1	9	10	3	4	6	7
$\overline{A}$	3	2	5	5	2	5	5	2	4	4	2	1

Schritt 1:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	2	5	1	10	9	4	7	3	6
$\overline{A}$	5	4	9	8	4	9	8	3	7	6	2	1

Verfeinerung – Beispiel (Fortsetzung)

Schritt 2:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	2	5	1	10	9	7	4	6	3
$\overline{A}$	5	4	11	9	4	10	8	3	7	6	2	1

Schritt 3:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	5	2	1	10	9	7	4	6	3
$\overline{A}$	6	5	12	10	4	11	9	3	8	7	2	1

Skew Algorithmus von Kärkkäinen und Sanders

Idee

1. Sortiere die Suffixe $S_j = S[j \ldots n]$ mit $j \, mod \, 3 \neq 1$.
Zeit: $O(n)$ + rekursiver Aufruf für Problem der Größe $\lceil 2n/3 \rceil$

2. Sortiere die Suffixe S_j mit $j \, mod \, 3 = 1$.
Zeit: $O(n)$

3. Konstruiere aus den beiden Teilen das Suffix-Array (Merge). Zeit: $O(n)$

Gesamte Laufzeit: $O(n)$

Skew Algorithmus – Schritt 1

Der Einfachheit halber: $|S| = n = 3m$.

Sortiere $\{j : 2 \leq j \leq n \wedge j \, mod \, 3 \neq 1\}$ nach Schlüssel $(S[j], S[j+1], S[j+2])$ mittels RADIX SORT.

Entsprechend der lexikografischen Ordnung von $(S[j], S[j+1], S[j+2])$ erhält jedes j mit $j \, mod \, 3 \neq 1$ einen Namen $N[j] \in \{1, \ldots, 2n/3\}$.

Sind die $N[j]$ paarweise verschieden, so ist die gewünschte Ordnung erreicht.

Anderenfalls konstruiere Wort $N = N[2]N[5] \ldots N[3m-1]N[3]N[6] \ldots N[3m]$. Ordnung ergibt sich aus Suffix-Array von N (rekursiv bestimmen).

Skew Algorithmus – Schritt 2

Ordnung der Suffixe S_j, $j \, mod \, 3 = 1$, ergibt sich aus Ordnung bezüglich $(S[j], S_{j+1})$.

Ordnung der S_{j+1} ist aus Schritt 1 bekannt.

Das heißt: $\textsc{Radix Sort}$ der nach S_{j+1} geordneten Indizes $(j+1)$ nach Schlüssel $S[j]$ ergibt Ordnung der Suffixe S_j, $j \, mod \, 3 = 1$.

Skew Algorithmus – Schritt 3

Beim Mischen Vergleiche von Suffixen der Form S_{3i+1} mit Suffixen der Form S_{3j+2} oder S_{3j}

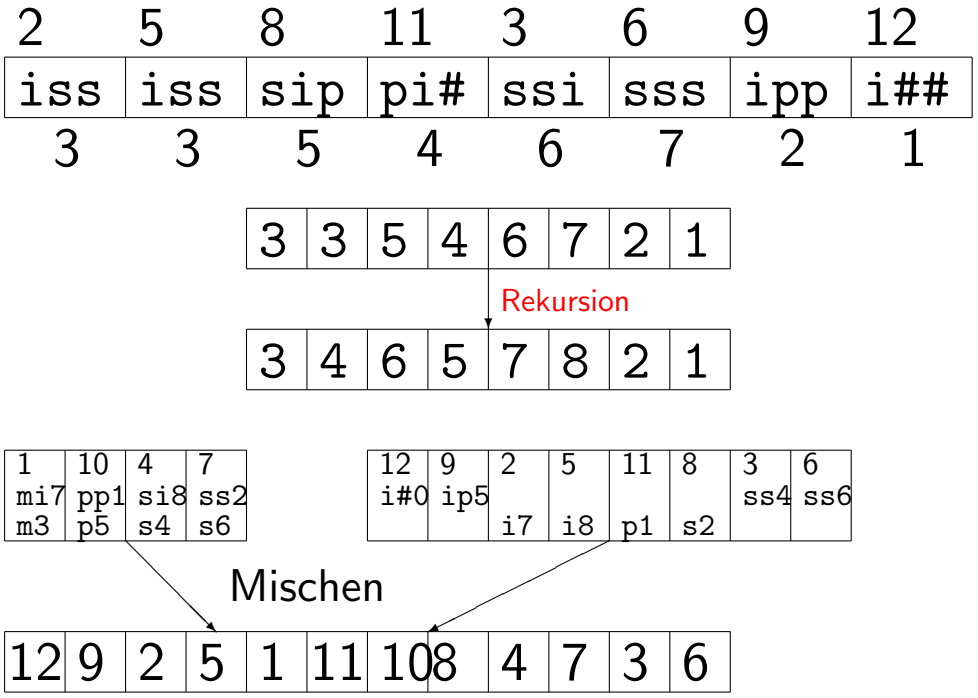
Vergleich eines Suffixes S_{3i+1} mit einem Suffix S_{3j+2}:
Vergleiche die Paare $(S[3i+1], S_{3i+2})$ und $(S[3j+2], S_{3j+3})$.
Reihenfolge von S_{3i+2} und S_{3j+3} ist bekannt aus Schritt 1.

Vergleich eines Suffixes S_{3i+1} mit einem Suffix S_{3j}:
Vergleiche die Tripel $(S[3i+1], S[3i+2], S_{3i+3})$ und $(S[3j], S[3j+1], S_{3j+2})$.
Reihenfolge von S_{3i+3} und S_{3j+2} ist bekannt aus Schritt 1.

Zeit für einen Vergleich: $O(1)$; gesamt $O(n)$.

Skew Algorithmus – Beispiel

$S =$ `mississsippi`



Tripel
Namen

neues Wort N

inverses Suffix-Array von N
(sortierte Suffixe mod 2,0)

links: sortierte Suffixe mod 1
rechts: sortierte Suffixe mod 2,0

Suffix-Array

Verbessertes Suffix-Array: LCP-Werte

Definition. Es sei S ein Wort der Länge n mit dem Suffix-Array A_S.
Das LCP-Array von S ist das Feld LCP_S der Länge $(n+1)$, wobei $LCP_S[i]$ für $1 \leq i \leq n$ das längste gemeinsame Präfix der Suffixe $S[A_S[i] \ldots n]$ und $S[A_S[i+1] \ldots n]$ ist.
Wir definieren $LCP_S[n+1] = -1$.

Beispiel. Für $S = mississippi$ ergibt sich folgendes LCP-Array:

i	1	2	3	4	5	6	7	8	9	10	11	12
$A[i]$	12	11	8	5	2	1	10	9	7	4	6	3
$LCP[i]$	0	1	1	4	0	0	1	0	2	1	3	-1

Konstruktion des LCP-Arrays

Lemma. Es sei S ein Wort der Länge n mit dem inversen Suffix-Array $\overline{A}_S$. Dann gilt $LCP_S[\overline{A}_S[i+1]] \geq LCP_S[\overline{A}_S[i]] - 1$ für $1 \leq i \leq n$.

Satz. Das LCP-Array kann mit linearem Aufwand konstruiert werden.

Speicherung des LCP-Arrays in der Praxis
Für LCP-Werte < 255 (d.h. für die meisten): 1 Byte
Für LCP-Werte ≥ 255: Eintrag 255 in LCP-Array, Speicherung des tatsächlichen Wertes in Extra-Tabelle (gesamt: 9 Bytes)
$\longrightarrow$ Gesamter Speicherbedarf in der Regel etwas mehr als n Bytes.

LCP-Intervalle

Definition. Es sei S ein Wort der Länge n mit dem LCP-Array LCP.
Ein Intervall $[lb \ldots rb]$ mit $1 \leq lb < rb \leq n+1$ heißt LCP-Intervall mit Wert ℓ (kurz ℓ-Intervall), wenn

1. $LCP[lb-1] < \ell$, $LCP[rb] < \ell$,
2. $LCP[k] \geq \ell$ für alle $lb \leq k < rb$,
3. $LCP[k] = \ell$ für mindestens ein $lb \leq k < rb$.

Beispiel. $S = mississippi$ hat das LCP-Array

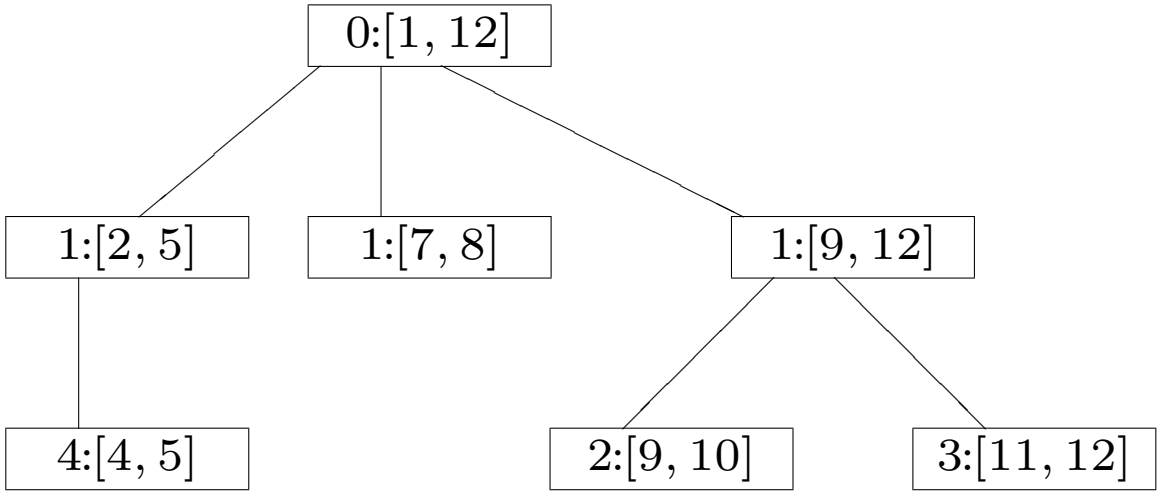
i	1	2	3	4	5	6	7	8	9	10	11	12
$LCP[i]$	0	1	1	4	0	0	1	0	2	1	3	-1

und damit folgende ℓ-Intervalle:
0-Intervall: $[1 \ldots 12]$; 1-Intervalle: $[2 \ldots 5]$, $[7 \ldots 8]$, $[9 \ldots 12]$; 2-Intervall: $[9 \ldots 10]$;
3-Intervall: $[11 \ldots 12]$; 4-Intervall: $[4 \ldots 5]$.

Baum der LCP-Intervalle

LCP-Intervall entspricht innerem Knoten des Suffixbaumes. Damit ist das Hasse-Diagramm der Teilmengenrelation der LCP-Intervalle der Suffixbaum ohne Blätter.

Beispiel. Baum der LCP-Intervalle für $S = mississippi$:



Bedeutung des Baumes: Konzeptuell (wird nicht tatsächlich konstruiert).
Traversierungen des Suffixbaumes werden mittels der LCP-Intervalle nachvollzogen.

Traversierung der LCP-Intervalle

LCP-Intervalle werden in der Postorder-Reihenfolge des Baumes ermittelt.
Intervall wird durch Tripel (ℓ, lb, rb) dargestellt.
Aktuell bearbeitete Intervalle werden auf einem Stack gehalten.
Rechtes Ende rb ist vor Abschluss eines Intervalls unbekannt (Wert '?').
Traversierung erfolgt durch einmaligen Lauf über das LCP-Array.

---

(1) $push((0, 1, ?))$;
(2) **for** $i \leftarrow 2$ **to** $n + 1$
(3) $\quad lb \leftarrow i$;
(4) $\quad$ **while** $LCP[i] < top.\ell$
(5) $\quad\quad top.rb \leftarrow i$;
(6) $\quad\quad intervall \leftarrow pop$;
(7) $\quad\quad$ Ausgabe: $intervall$;
(8) $\quad\quad lb \leftarrow intervall.lb$;
(9) $\quad$ **if** $LCP[i] > top.\ell$ **then** $push((LCP[i], lb, ?))$;

4.4 Anwendungen von Suffixbäumen und Suffix-Arrays

- exakte Suche in unveränderlichen Texten
- inexakte Suche in unveränderlichen Texten
- Finden von Regelmäßigkeiten (z.B. längste Wiederholungen)
- Finden von Gemeinsamkeiten in Texten (z.B. längstes gemeinsames Teilwort)
- Datenkompression (Lempel-Ziv, Burrows-Wheelers)

Anwendung von Suffixbäumen: Methoden

- Durchsuchung von der Wurzel nach unten (top-down)
  Beispiel: Suche nach einem Wort

- Durchsuchung von den Blättern nach oben (bottom-up)
  Beispiel: Suche nach gemeinsamen Teilwörtern

- Ermittlung des letzten gemeinsamen Vorfahren (least common ancestor – LCA)
  Beispiel: Suche nach Palindromen

- Verwendung von Suffix-Links (seltener)
  Beispiel: Konstruktion des DAWG

- Effizienter ist in der Regel die Verwendung von erweiterten Suffix-Arrays (mit LCP-Array und weiteren Tabellen)
  Abouelhoda, Kurtz, Ohlebusch: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2 (2004), 53-86.

Exakte Suche im Suffixbaum

Gegeben: unveränderlicher Text T, $|T| = n$.
Nach einem Präprozessing mit Aufwand $O(n)$ kann man für ein Suchwort P der Länge m folgende Fragen mit einem Aufwand von $O(m)$ beantworten:

Kommt P in T vor?

Suche im Suffixbaum von T nach einem Pfad mit der Beschriftung P.

Wie oft kommt P in T vor?

Präprozessing: Ermittle im Suffixbaum von T von den Blättern beginnend (bottom-up) für jeden Knoten die Anzahl der Blätter im Unterbaum. (Zeit $O(n)$)
Die Anzahl der Vorkommen von P kann man am nächsten Knoten nach dem Pfad P ablesen.

Welches ist das erste Vorkommen von P in T?

Präprozessing: Ermittle im Suffixbaum von T von den Blättern beginnend (bottom-up) für jeden Knoten das erste Vorkommen des zugehörigen Wortes. (Zeit $O(n)$)
Das erste Vorkommen von P kann man am nächsten Knoten nach dem Pfad P ablesen.

Exakte Suche im Suffix-Array

Gegeben: unveränderlicher Text T, $|T| = n$ durch Suffix-Array A_T.

Suche Wort P mit $|P| = m$ mittels binärer Suche:
Suchintervall sei $[L, R]$; am Anfang $L = 1$, $R = n + 1$.
Vergleiche P mit Suffix T_j an Position $j = A_T[M]$ mit $M = \lceil (L + R)/2 \rceil$.
1. Fall: P ist Präfix von $T_j \rightarrow$ P kommt in T vor.
2. Fall: $P <_{lex} T_j$. Falls $L = M$, so kommt P nicht in T vor; anderenfalls Suche in $[L, M]$.
3. Fall: $P >_{lex} T_j$. Falls $R = M$, so kommt P nicht in T vor; anderenfalls Suche in $[M, R]$.

Menge der Vorkommen von P entspricht Teilintervall von A_T.
Bestimmung der Intervallgrenzen durch binäre Suche.

Laufzeit der (naiven) binären Suche $O(m \cdot \log n)$.

Beschleunigte Suche im Suffix-Array

Sei bekannt, dass P mit den Suffixen an den Stellen $A_T[L]$ bzw. $A_T[R]$ in den ersten l bzw. r Positionen übereinstimmt.
Dann stimmt P mit dem Suffix an der Stelle $A_T[M]$ in den ersten $\min\{l, r\}$ Positionen überein.

Beschleunigte binäre Suche: Lasse die ersten $\min\{l, r\}$ Vergleiche aus.

Laufzeit der beschleunigten Suche: $O(m \cdot \log n)$ im schlechtesten Fall, $O(m + \log n)$ im Mittel.
Problem: Nur Übereinstimmung von $\min\{l, r\}$ Zeichen garantiert.

Schlechter Fall: $T = a^{n-1}c$, $P = a^{m-1}b$.
In den meisten Phasen gilt: $l = m - 1$, $r = 0 \rightarrow m$ Vergleiche notwendig.

Suche mit verallgemeinerten LCP-Werten

Definition. Es sei A_T das Suffix-Array von T mit $|T| = n$. Für $1 \leq i < j \leq n+1$ ist $LCP(i, j)$ die Länge des längsten gemeinsamen Präfixes der Suffixe $T[A_T[i] \ldots n]$ und $T[A_T[j] \ldots n]$.

Binäre Suche mit bekannten verallgemeinerten LCP-Werten:
1. Fall: $l > r$.
1.1. $LCP(L, M) > l$: P ist lexikografisch größer als $T[A_T[M] \ldots n]$ bei Übereinstimmung l, d.h. $L \leftarrow M$.
1.2. $LCP(L, M) < l$: P ist lexikografisch kleiner als $T[A_T[M] \ldots n]$ bei Übereinstimmung $LCP(L, M)$, d.h. $R \leftarrow M$, $r \leftarrow LCP(L, M)$.
1.3. $LCP(L, M) = l$: P hat mit $T[A_T[M] \ldots n]$ mindestens Übereinstimmung l, d.h. Vergleich ab Position $l + 1$.
2. Fall: $l < r$. Analog zum 1. Fall unter Verwendung von $LCP(M, R)$.
3. Fall: $l = r$. Wie bisher Vergleich von P und $T[A_T[M] \ldots n]$ ab Position $l + 1$.

Maximale Anzahl von Vergleichen: $O(m + \log n)$.

Berechnung der verallgemeinerten LCP-Werte

Der Einfachheit halber sei n eine Zweierpotenz.

Benötigte verallgemeinerte LCP-Werte:
$LCP(i, i + 2^k)$, falls $i + 2^k \leq n + 1$ und $i - 1$ Vielfaches von 2^k ist.
Anzahl der benötigten LCP-Werte: $2n$.

Berechnung iterativ mit wachsendem k:
$LCP(i, i + 1) = LCP[i]$;
$LCP(i, i + 2^{k+1}) = \min\{LCP(i, i + 2^k), LCP(i + 2^k, i + 2^{k+1})\}$.

Zeitaufwand: konstant je verallgemeinertem LCP-Wert, gesamt $O(n)$.

Suche mit LCP-Intervallen

Zusätzlich zum LCP-Array verwende Array $child$.
(Zugriff auf Kinder im Baum der LCP-Intervalle)

Simulation der Suche im Suffixbaum ohne Zeitverlust.
Laufzeit: $O(m)$ bei geringerem Platzbedarf und wenigen Zugriffen auf Sekundärspeicher.

Details in: Abouelhoda, Kurtz, Ohlebusch. Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2 (2004), 53-86.

Inexakte Suche in einem unveränderlichen Text

Gegeben: unveränderlicher Text T, Suchwort P;
Gesucht: Inexakte Vorkommen von P in T (z.B. bis Abstand k)

Lösungsansätze

- Alignment am Suffixbaum von T (ähnlich zum Alignment am Trie)
- Filtermethode: Zerlege $P = P_1 P_2 \cdots P_k$ und suche exakte Vorkommen der P_i als Kandidaten.

Generelles Problem: Große Teile des Suffixbaumes müssen durchsucht werden.

Wiederholungen

Definition. Es sei $S \in \Sigma^*$ ein Wort der Länge n. Ein Tripel (α, i, j) mit $\alpha \in \Sigma^*$, $1 \leq i < j \leq n$ heißt Wiederholung in S,
wenn $\alpha = S[i \ldots i + |\alpha| - 1] = S[j \ldots j + |\alpha| - 1]$ gilt.
Eine Wiederholung (α, i, j) heißt maximal, wenn zusätzlich $S[i-1] \neq S[j-1]$ und $S[i + |\alpha|] \neq S[j + |\alpha|]$ gilt.
Ein Wort α heißt supermaximale Wiederholung, wenn α mehrfach vorkommt, aber kein Wort mehrfach vorkommt, für das α ein echtes Teilwort ist.

Beispiel. Das Wort $abcaabcbaabca$ enthält u.a. die Wiederholungen $(abc, 1, 5), (abc, 1, 10), (abc, 5, 10)$, von denen nur $(abc, 1, 5)$ maximal ist. Supermaximale Wiederholungen sind $abca$ und $aabc$.

Typische Fragestellungen: Anzahl maximaler Wiederholungen einer Mindestlänge; Bestimmung der supermaximalen Wiederholungen; Bestimmung der längsten Wiederholung.

Wiederholungen und Suffixbäume

Eine Wiederholung der Form (α, i, j) existiert genau dann, wenn der Pfad α im Suffixbaum in einem inneren Knoten oder auf der Kante zu einem inneren Knoten endet, in dessen Unterbaum sich die Blätter i und j befinden.

Eine maximale Wiederholung der Form (α, i, j) existiert genau dann, wenn der Pfad α im Suffixbaum in einem inneren Knoten v endet, die Blätter i und j sich in den Unterbäumen zweier verschiedener Kinder von v befinden und $S[i-1] \neq S[j-1]$ gilt.

Eine supermaximale Wiederholung α existiert genau dann, wenn der Pfad α im Suffixbaum in einem inneren Knoten v endet, dessen Kinder sämtlich Blätter sind, und $S[i-1] \neq S[j-1]$ für alle Kinder $i \neq j$ von v gilt.

Bestimmung der Anzahl maximaler Wiederholungen

Bestimme für jeden Knoten v die Anzahl $maxrep(v)$ der maximalen Wiederholungen des Wortes zum Knoten v.
Weitere Werte: Anzahl der Blätter $leaves(v)$,
Anzahl der Blätter mit "linker Fortsetzung" $a \in \Sigma \cup \{\#\}$: $leaves_a(v)$.

Berechnung erfolgt bottom-up:
Beginne mit den Blättern; für innere Knoten verwende Werte der Kinder.

Für ein Blatt v, zum Suffix $S[i \ldots n]$ gehörend:

$$maxrep(v) = 0;\ leaves(v) = 1;\ leaves_a(v) = \begin{cases} 1, & a = S[i-1] \\ 0, & \text{sonst} \end{cases};$$

Anzahl maximaler Wiederholungen: Fortsetzung

Für einen inneren Knoten v:

(1) $maxrep(v) \leftarrow 0$; $leaves(v) \leftarrow 0$;
(2) **foreach** a
(3) $\quad leaves_a(v) \leftarrow 0$;
(4) **foreach** Kind u von v
(5) $\quad$ **foreach** a
(6) $\quad\quad maxrep(v) \leftarrow maxrep(v) + leaves_a(v) \cdot (leaves(u) - leaves_a(u))$;
(7) $\quad\quad leaves_a(v) \leftarrow leaves_a(v) + leaves_a(u)$;
(8) $\quad\quad leaves(v) \leftarrow leaves(v) + leaves(u)$;

Längstes gemeinsames Teilwort

Gegeben: Wörter S_1, S_2.
Gesucht: Längstes Wort α, das Teilwort von S_1 und S_2 ist.

Konstruiere den gemeinsamen Suffixbaum von S_1 und S_2.

Wort α ist gemeinsames Teilwort, wenn der Pfad α in einem Knoten oder in der Kante zu einem Knoten endet, der Blätter zu S_1 und zu S_2 enthält.

Bestimme bottom-up für jeden Knoten v einen Bitvektor $(b_1, b_2)[v]$, wobei b_i genau dann 1 ist, wenn sich im Unterbaum von v ein Blatt zu S_i befindet.
Induktionsschritt: $b_i[v] = \bigvee_{u \text{ Kind von } v} b_i[u]$.

Längstes gemeinsames Teilwort gegeben durch Knoten mit Bitvektor $(1, 1)$ und größter String-Tiefe.

Lempel-Ziv-Algorithmus (LZ77)

Eingabe: Text T, $|T| = n$,
Ausgabe: Komprimierte Z.

Idee der Lempel-Ziv-Komprimierung:
Initialisierung: $Z \leftarrow \varepsilon$; $i \leftarrow 0$; (i ist Position im Text)
Längstes Präfix α von $T[i+1 \ldots n]$, das an einer Stelle $p_i \leq i$ vorkommt, habe die Länge ℓ_i.
Falls $\ell_i = 0$, hänge $(0, T[i+1])$ an Z an; setze $i \leftarrow i+1$.
Falls $\ell_i > 0$, hänge (p_i, ℓ_i) an Z an; setze $i \leftarrow i + \ell_i$.

Unterschied zu LZW: Das Vorkommen von α an p_i muss nicht notwendig bis zur Stelle i enden.

Dekompression ist sehr einfach mit Aufwand $O(n)$.

Beispiel für LZ77-Kompression

$\texttt{abaababaabaab} \rightarrow (0,\texttt{a})(0,\texttt{b})(1,1)(1,3)(2,5)(1,2)$.

$\texttt{ababababababa} \rightarrow (0,\texttt{a})(0,\texttt{b})(1,11)$.

Bei hoher Periodizität (2. Beispiel) bessere Kompression als LZW.

LZ77 und Suffixbäume

Bestimme bottom-up für jeden Knoten v das erste Vorkommen $First(v)$ des zu v gehörigen Teilwortes.

Bestimmung des längsten Präfixes von $T[i+1 \ldots n]$:
Verfolge von der Wurzel den Pfad der Beschriftung $T[i+1 \ldots n]$ so lange, wie der $First$-Wert des nächsten Knotens höchstens i ist.
ℓ_i ist Länge des Pfades; p_i ist $First$-Wert des letzten Pfad-Knotens.

Einfacher: ℓ_i ergibt sich automatisch bei Einfügen von Suffix $T[i+1 \ldots n]$;
$First$-Wert eines Knotens ergibt sich bei dessen Einfügung;
d.h. LZ77-Kompression simultan zu Suffixbaum-Konstruktion möglich.

Gesamtaufwand der Berechnung der LZ77-Komprimierten: $O(n)$.

Die Probleme LCA und RMQ

Letzter gemeinsamer Vorfahre (Least Common Ancestor – LCA).
Gegeben: Baum $\mathcal{T}$, Knoten u, v.
Gesucht: letzter Knoten, der auf den Pfaden von der Wurzel nach u und v liegt.

Bereichsminimum-Anfragen (Range Minimum Query – RMQ).
Gegeben: Array A der Größe n, Indizes $1 \leq i < j \leq n$.
Gesucht: Index k im Intervall $[i, j]$ mit minimalem Wert $A[k]$.

Beide Probleme können nach einem Präprozessing mit Aufwand $O(n)$ mit konstantem Aufwand gelöst werden.
Erster Algorithmus: Harel/Tarjan, 1984.
Einfacher Algorithmus: Bender/Farach-Colton, 2000. (im folgenden dargestellt)

Reduktion von LCA auf RMQ

Ein Baum $\mathcal{T}$ mit n Knoten wird in linearer Zeit in ein Array A der Größe $2n-1$ (und zusätzliche Arrays E bzw. F der Größen $2n-1$ bzw. n) überführt:
E enthält die Knoten von $\mathcal{T}$ in der Reihenfolge der DFS-Traversierung.
$A[i]$ enthält die Tiefe des Knotens $E[i]$.
F enthält für jeden Knoten das erste Auftreten in E.

Beantwortung einer LCA-Anfrage $LCA(u, v)$:
1. Bestimme $i = \min\{F[u], F[v]\}$, $j = \max\{F[u], F[v]\}$.
2. Antwort auf RMQ für (A, i, j) sei k.
3. LCA von u und v ist $E[k]$.

Beachte: Reduktion erfolgt auf Spezialfall von RMQ, da die Differenz benachbarter Werte in A immer den Betrag 1 hat: ± 1-RMQ.

Reduktion von RMQ auf LCA

Definition. Für ein Array A der Größe n ist der kartesische Baum $C(A)$ mit den Knoten $1, 2, \ldots n$ wie folgt definiert:
Die Wurzel von $C(A)$ ist die Zahl i, für die $A[i]$ minimal wird.
Die Unterbäume der Wurzel sind die kartesischen Bäume von $A[1 \ldots i-1]$ und $A[i+1 \ldots n]$.

Lemma. Der kartesische Baum kann in linearer Zeit konstruiert werden.

Beantwortung einer Anfrage $RMQ(A, i, j)$:
Bestimme in $C(A)$ den LCA von i und j.

Folgerung. Kann ± 1-RMQ nach $O(n)$-Präprozessing in Zeit $O(1)$ gelöst werden, so auch LCA und RMQ.

Präprozessing für RMQ

Volle RMQ-Matrix: Bestimme für jedes Paar (i, j) die Antwort auf $RMQ(A, i, j)$
Zeit und Platzbedarf: $O(n^2)$

Spärliche RMQ-Matrix: Bestimme für jedes i und jede Zweierpotenz 2^q mit $i + 2^q \leq n$ die Antwort auf $RMQ(A, i, i + 2^q)$
Zeit und Platzbedarf: $O(n \log n)$.

Beantwortung einer Anfrage $RMQ(A, i, j)$ mittels der spärlichen Matrix:
Sei $q = \lfloor \log_2(j - i) \rfloor$. Bestimme $k_1 = RMQ(A, i, i + 2^q)$, $k_2 = RMQ(A, j - 2^q, j)$.
Ermittle $\min\{A[k_1], A[k_2]\}$, Argument des Minimums ist gesuchter Wert.

Noch zu tun: Ersparnis der Faktors $\log n$. Möglich für ± 1-RMQ.

Fortsetzung: Präprozessing für ± 1-RMQ

Zerlege A in Blöcke der Größe $\frac{1}{2}\log n$.
Konstruiere Arrays A' und B der Größe $2n/\log n$.
$A'[i]$: Wert des Minimums im i-ten Block von A.
$B[i]$: Argument des Minimums im i-ten Block von A.

Bestimme für A' die spärliche RMQ-Matrix (Aufwand $O(n)$).
Bestimme für jeden Block von A die volle RMQ-Matrix (Aufwand $O(n)$).

Beantwortung einer Anfrage $RMQ(A, i, j)$:
Bestimme Blocknummern i' bzw. j' für i bzw. j.
Falls $i' = j'$, lies $RMQ(A, i, j)$ aus der vollen Matrix für den Block i' ab.
Falls $i' < j'$, bestimme 3 Minima k_1, k_2, k_3:

$k_1 = RMQ(A, i, r)$, wobei r rechtes Ende des Blockes i' ist,
$k_2 = B[k']$, wobei $k' = RMQ(A', i' + 1, j' - 1)$ ist,
$k_3 = RMQ(A, l, j)$, wobei l linkes Ende des Blockes j' ist

und ermittle $\min\{A[k_1], A[k_2], A[k_3]\}$.

Aufwand für Berechnung der Block-RMQ-Matrizen

Beobachtung: RMQ-Matrix für ein Array $A[1 \ldots q]$ nur abhängig vom Offsetvektor $D[1 \ldots q-1]$ mit $D[i] = A[i+1] - A[i]$.

Anzahl der Offsetvektoren bei ± 1-RMQ: 2^{q-1}.
Für Blöcke der Länge $1/(2 \log n)$ gibt es $O(\sqrt{n})$ Offsetvektoren.

Konstruktion der vollen RMQ-Matrizen für die Blöcke von A:

1. Bestimme für jeden Offsetvektor die volle RMQ-Matrix; Aufwand: $O(\sqrt{n} \cdot \log^2 n)$.
2. Bestimme für jeden Block den zugehörigen Offsetvektor; Aufwand: $O(n)$.

Satz. Die Probleme LCA und RQM sind nach einem Präprozessing von linearem Aufwand in konstanter Zeit lösbar.

Beispiel für LCA-Anfragen – längste gemeinsame Erweiterung

Definition. Es sei S ein Wort der Länge n und $1 \leq i < j \leq n$. Die längste gemeinsame Erweiterung für i und j ist die größte Zahl l mit $S[i \ldots i+l-1] = S[j \ldots j+l-1]$.

Anwendungen der längsten gemeinsamen Erweiterung, z.B.
Suche mit wild cards in Text und Muster
Bestimmung von Tandem-Wiederholungen, d.h. Teilwörter $\alpha\alpha$
Bestimmung maximaler Palindrome

Ermittlung der längsten gemeinsamen Erweiterung für i, j:
String-Tiefe des LCA der Blätter i, j.

Im Suffix-Array Ersetzung der LCA-Anfrage durch RMQ im LCP-Array:
$RMQ(LCP, k_1, k_2 - 1)$ mit $k_1 = \min\{\overline{A}[i], \overline{A}[j]\}, k_2 = \max\{\overline{A}[i], \overline{A}[j]\}$

Beispiel für LCA-Anfragen – Palindrome

Gegeben: Wort S
Gesucht: längstes Palindrom in S

Ein Teilwort von S heißt ungerades Palindrom mit Mittelpunkt i, wenn es ein Palindrom ist und die Form $S[i-k \ldots i+k]$ hat.

Konstruiere gemeinsamen Suffixbaum von S und Spiegelbild S^R.
Blatt habe Bezeichnung b_i, falls zum Suffix $S[i \ldots n]$ gehörig, bzw.
b_{i^R}, falls zum Suffix $S[i]S[i-1] \cdots S[1]$ von S^R gehörig.

Längstes ungerades Palindrom mit Mittelpunkt i ergibt sich aus $LCA(b_i, b_{i^R})$.

Bestimmung des längsten ungeraden Palindroms durch n LCA-Anfragen möglich, also mit Gesamtaufwand $O(n)$.

Für längste gerade Palindrome bestimme $LCA(b_{i+1}, b_{i^R})$.

Palindrome: Lösung mit Suffix-Arrays

Konstruiere gemeinsames Suffix-Array und gemeinsames LCP-Array von S und S^R.
Führe RMQ im LCP-Array durch.

Beispiel:

1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
A	0^R	12	11	2^R	8	11^R	5^R	5	8^R	2	1^R	1	10	9^R	9	10^R	3^R	7	6^R	4	4^R	6	7^R	3
LCP	0	0	4	1	4	1	4	4	7	0	1	0	2	1	3	0	2	2	5	1	3	3	4	-1

Längstes gerades Palindrom mit Mitte $(3, 4)$ ergibt sich aus minimalem LCP-Wert im Intervall $[k_1, k_2 - 1]$ mit $k_1 = \min\{\overline{A}[3^R], \overline{A}[4]\}$, $k_2 = \max\{\overline{A}[3^R], \overline{A}[4]\}$.
Minimaler LCP-Wert: 2; also maximale Palindromlänge 4.

Burrows-Wheeler-Transformation

- transformiert ein Wort $S\#$ eineindeutig in ein Wort S' gleicher Länge
- Grundlage: lexikografische Ordnung der Suffixe von S
- Transformation und Rücktransformation in linearer Zeit
- Hauptanwendung Kompression: S' enthält oft lange Blöcke mit einem Symbol $\rightarrow S'$ kann gut komprimiert werden, implementiert in `bzip2`.
- weitere Anwendung: Bestimmung maximaler Wiederholungen mit Suffix-Arrays

BW-Transformierte: Definition

Definition. Es sei $S \in \Sigma^*$ ein Wort mit $|S| = n$ und $\# \notin \Sigma$ ein Sonderzeichen. Die Burrows-Wheeler-Transformierte von S ist das Wort S' der Länge $(n+1)$ mit $S'[j] = S[A_S[j] - 1]$, wobei $S[0] := \#$.

Beispiel. Für $S = abcabca$ erhalten wir:

$$\begin{aligned} A_S &= (8, 7, 4, 1, 5, 2, 6, 3), \\ S' &= \texttt{a c c \# a a b b} \end{aligned}$$

Satz. Die Burrows-Wheeler-Transformierte von S kann in $O(n)$ Schritten ermittelt werden.

Interpretation der BW-Transformierten

Schreibe die zyklischen Permuationen von $S\#$ in lexikografischer Ordnung untereinander. Die letzte Spalte ergibt S'.

Beispiel. $S = abcabca$

```
#abcabc   a
a#abcab   c
abca#ab   c
abcabca   #
bca#abc   a
bcabca#   a
ca#abca   b
cabca#a   b
```

BW-Transformierte: Rücktransformation

Satz. S kann aus S' eindeutig bestimmt werden.

Beweis.

Von der Matrix der zyklischen Permutationen sind
die erste Spalte $\overline{S}$ (Symbole von S' in lexikografischer Ordnung) und
die letzte Spalte (S') bekannt.

Bestimme induktiv die Symbole $S[i]$ (verwende Zeiger j):
$i = 1$: Sei j die Position von $\#$ in S'. Dann ist $S[1] = \overline{S}[j]$.
$i \leftarrow i + 1$: Die Zeile mit der Permutation $S[i \ldots n+1]S[1 \ldots i-1]$ sei die k-te Zeile, die mit $S[i]$ beginnt.
Dann ist die Zeile mit der Permutation $S[i+1 \ldots n+1]S[1 \ldots i]$ die k-te Zeile, die mit $S[i]$ endet. Sei j die Nummer dieser Zeile. Dann ist $S[i+1] = \overline{S}[j]$.

Rücktransformation in Linearzeit

Definiere Array P über $\Sigma \cup \{\#\}$ mit $P[a] = \sum_{b<a} |S'|_b$.
($P[a]+1$ ist erstes Vorkommen von a in erster Spalte.)

Definiere für $a \in \Sigma \cup \{\#\}$ das Array V_a der Größe $|S'|_a$ mit
$V_a[i] = j :\Longleftrightarrow\ S'[j] = a \wedge |S'[1 \ldots j]|_a = i$.
($V_a[i]$ ist i-tes Vorkommen von a in letzter Spalte.)

P und V_a können in linearer Zeit bestimmt werden.

In Schritt i sei Zeile j erreicht. $\rightarrow S[i] = \overline{S}[j] = a$;
Dann wird im Schritt $i+1$ die Zeile $V_a[j - P[a]]$ erreicht.

Rücktransformation: Beispiel

$S' = \texttt{acc\#aabb}$

$P[\#] = 0; \quad P[a] = 1; \quad P[b] = 4; \quad P[c] = 6$
$V_\# = (4); \quad V_a = (1, 5, 6); \quad V_b = (7, 8); \quad V_c = (2, 3)$

$\overline{S}$	#	a	a	a	b	b	c	c
S'	a	c	c	#	a	a	b	b
	8	7	4	1	5	2	6	3

Die Rücktransformation ergibt damit $S = abcabca$.

Anwendung der BWT für die Kompression (`bzip2`)

1. Konstruktion der Burrows-Wheeler-Transformierten
   Dadurch entstehen große Blöcke mit einem oder wenigen Zeichen.

2. Move-to-Front-Transformation der BW-Transformierten
   Die MTF-Transformierte enthält die ersten Zeichen des Alphabets sehr häufig (ASCII: `\0`, `\1`).

3. Huffman-Kodierung der MTF-Transformierten
   Die Verteilung der Zeichen in der MTF-Transformierten sorgt für ein gutes Kompressionsergebnis.

Bestimmung von Tandem-Wiederholungen

- **Gesucht:** Teilwörter der Form $\alpha\alpha$

- Bedeutung von Tandem-Wiederholungen:
  - machen großen Teil des Erbgutes aus,
  - starke individuelle Variation (genetischer Fingerabdruck).

- Algorithmus von Gusfield und Stoye markiert in Linearzeit die Enden aller Tandem-Wiederholungen im Suffixbaum.

  Verwendete Techniken:
  - LZ77-Zerlegung,
  - LCA-Anfragen (längste gemeinsame Erweiterung),
  - Suffix-Links.

Tandem-Wiederholungen: Notationen

Sei $S[i \ldots i+l-1] = \alpha\alpha$.

Repräsentation (Tandem-Paar): (i, l)
Typ der Tandem-Wiederholung (Tandem-Typ): $\alpha\alpha$.
Zentrum von (i, l): $i + \frac{l}{2} - 1$. (letzte Position der ersten Hälfte)

Intervall $[i, \ldots, k]$ heißt l-Lauf von Tandem-Wiederholungen,
wenn $(i, l), (i+1, l), \ldots, (k, l)$ jeweils Tandem-Paare sind.
$[i, \ldots, k]$ ist l-Lauf $\iff$ $S[i \ldots k+l-1] = \alpha^{2+r}\alpha'$, $|\alpha\alpha| = l$, α' Präfix von α.
Das Tandem-Paar (j, l) mit $i \leq j \leq k$ wird von (i, l) überdeckt.
Wird (j, l) von (i, l) überdeckt und ist (i, l) vom Typ $\alpha\alpha$ mit $\alpha = \alpha_1\alpha_2$ und $\alpha_1 = j - i$, so ist (j, l) vom Typ $\alpha'\alpha'$ mit $\alpha' = \alpha_2\alpha_1$.

Menge P von Tandem-Paaren heißt Links-Überdeckung, wenn für jeden Tandem-Typ das erste Vorkommen überdeckt ist.

Tandem-Wiederholungen: Beispiel

$S = abaabaabbaaabaaba$.

Tandem-Typen: aa, $aabaab$, $abaaba$, $baabaa$, bb

Tandem-Paare:
Typ aa: $(3, 2)$, $(6, 2)$, $(10, 2)$, $(11, 2)$, $(14, 2)$.
Typ $aabaab$: $(3, 6)$, $(11, 6)$.
Typ $abaaba$: $(1, 6)$, $(12, 6)$.
Typ $baabaa$: $(2, 6)$.
Typ bb: $(8, 2)$.

Läufe: $6 - [1, 2, 3]$, $2 - [10, 11]$, $6 - [11, 12]$.

Linksüberdeckung: $\{(1, 6), (3, 2), (8, 2)\}$.

Anzahl der Tandem-Typen

Satz. An jeder Position i starten für höchstens 2 Tandem-Typen die rechtesten Vorkommen.

Folgerung. Die Anzahl der verschiedenen Tandem-Typen in einem Wort der Länge n ist beschränkt durch $2n$.

Idee des Algorithmus

- Phase I: Konstruktion einer Links-Überdeckung
  Nutzung der LZ-Zerlegung und von LCA-Anfragen

- Phase II: Markierung einer "hinreichenden" Menge von Tandem-Typen
  Bottom-up-Traversierung des Suffixbaumes

- Phase III: Bestimmung aller Tandem-Typen
  Traversierung des Suffixbaumes mit Suffix-Links

Tandem-Wiederholungen und LZ-Zerlegung

LZ-Zerlegung: Zerlegung in Faktoren (Blöcke) entsprechend der LZ77-Kompression

Beispiel einer LZ-Zerlegung

a	b	a	a b a a b	b a a	a b a a b a

Lemma. Die 2. Hälfte einer Tandem-Wiederholung berührt höchstens 2 Blöcke.

Lemma. Das erste Vorkommen einer Tandem-Wiederholung berührt mindestens 2 Blöcke.

Folgerung. Betrachte das erste Paar (i, l) vom Typ $\alpha\alpha$. Liegt das Zentrum von (i, l) in Block b, so gilt entweder A oder B.

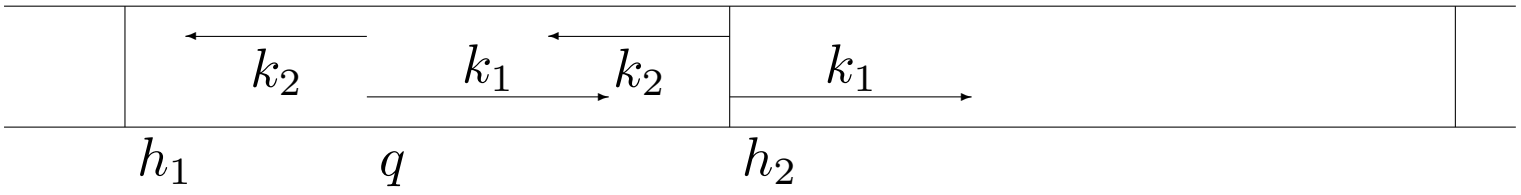
A: Das linke Ende liegt in Block b und das rechte in Block $b + 1$.

B: Das linke Ende liegt vor Block b (und das rechte in b oder $b + 1$).

Phase Ia: Betrachtung von Fall A

Betrachte den Block $b = [h_1, h_2 - 1]$. Die folgende Prozedur ermittelt für $k \leq h_2 - h_1$ ein Tandem-Paar $(i, 2k)$, das alle Tandem-Paare $(j, 2k)$ mit Zentrum in Block b, linkem Ende in Block b und rechtem Ende in Block $b + 1$ überdeckt.

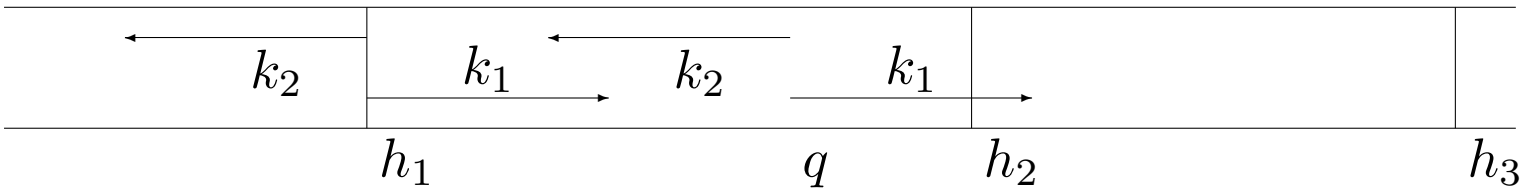
(1) $q \leftarrow h_2 - k$;
(2) $k_1 \leftarrow LCE(h_2, q)$;
(3) $k_2 \leftarrow LCE^R(h_2 - 1, q - 1)$; `// LCE-Werte rückwärts`
(4) **if** $k_1 + k_2 \geq k$ **and** $k_1 > 0$
(5) $\quad i \leftarrow \max\{q - k_2, q - k + 1, h_1\}$;
(6) $\quad$ **return** $(i, 2k)$;



Phase Ib: Betrachtung von Fall B

Betrachte die Blöcke $b = [h_1, h_2 - 1]$ und $b+1 = [h_2, h_3 - 1]$ Die folgende Prozedur ermittelt für $k \leq h_3 - h_1$ ein Tandem-Paar $(i, 2k)$, das alle Tandem-Paare $(j, 2k)$ mit Zentrum in Block b, rechtem Ende in Block b oder Block $b+1$ und linkem Ende vor Block b überdeckt.

(1) $q \leftarrow h_1 + k$;
(2) $k_1 \leftarrow LCE(h_1, q)$;
(3) $k_2 \leftarrow LCE^R(h_1 - 1, q - 1)$; `// LCE-Werte rückwärts`
(4) **if** $k_1 + k_2 \geq k$ **and** $k_1 > 0$ **and** $k_2 > 0$
(5) $\quad i \leftarrow \max\{h_1 - k_2, h_1 - k + 1\}$;
(6) $\quad$ **if** $i + k < h_2$ **then return** $(i, 2k)$;



Phase I: Analyse

Satz. Die Ausgaben der Phasen Ia und Ib ergeben eine Links-Überdeckung aller Tandem-Paare. Die Laufzeit ist linear.

Korrektheit: Folgt aus Eigenschaften des ersten Vorkommens eines Tandem-Typs.

Laufzeit: Für jeden Wert von k konstant. Für jeden Block linear bezüglich seiner Länge. Insgesamt linear wegen Disjunktheit der Blöcke.

Lemma. Man kann in linearer Zeit Listen $P(i)$ für alle Positionen i erstellen, wobei $P(i)$ die Tandem-Paare (i, l) geordnet nach fallender Länge l enthält.

Phase II: Markierung einer Teilmenge von Tandem-Typen

Bottom-up erhält jeder Knoten v eine Liste $P(v)$ zugeordnet:

Blatt zum Suffix i erhält Liste $P(i)$.
Für inneren Knoten u mit String-Tiefe d führe folgende Prozedur aus:

(1) **foreach** Kind v von u
(2) **while** $P(v) \neq \emptyset$ **and** $l > d$ für erstes Element (i, l) von $P(v)$
(3) Entferne (i, l) aus $P(v)$ und markiere Position $(u, v, l - d)$;
(4) $P(u) \leftarrow P(First(u))$;

Laufzeit: linear, da Zuweisung einer Liste nur Setzen eines Zeigers ist.

Phase II: Analyse

Lemma. Ist das Paar (i, l) zum Tandem-Typ $\alpha\alpha$ und wird (i, l) nicht durch ein Paar (i', l) mit $i' < i$ überdeckt, so wird die Position $\alpha\alpha$ des Suffixbaumes in Phase II markiert.

Ein Tandem-Typ $\alpha\alpha$ heißt nach Phase II erreichbar, wenn

1. die Position $\alpha\alpha$ im Suffixbaum markiert ist **oder**
2. das erste Vorkommen (i, l) von $\alpha\alpha$ durch ein Vorkommen (i', l) eines erreichbaren Tandem-Typs $\alpha'\alpha'$ überdeckt wird.

Satz. Alle im Wort S vorkommenden Tandem-Typen sind nach Phase II erreichbar.

Phase III: Markierung aller Tandem-Typen

Starte mit den in Phase II erhaltenen Markierungen von Tandem-Typen.

Für jeden markierten Tandem-Typ $x\beta x\beta$ ($x \in \Sigma$) suche nach Tandem-Typ $\beta x\beta x$.
Falls $\beta x\beta x$ existiert und noch nicht markiert ist, so setze Markierung.

Beschleunigung der Laufzeit durch Suffix-Links und Skip/Count-Trick:
Existenz von Pfad $\beta x\beta$ ist garantiert.
Sei u letzter Knoten auf dem Pfad $x\beta x\beta$ und γ das Wort von u bis zum Endpunkt von $x\beta x\beta$, so führe von $S[u]$ Skip/Count-Suche nach γ durch (und erreiche damit Position $\beta x\beta$).

Phase III: Analyse

Korrektheit

1. In Phase III werden nur Enden von Tandem-Typen markiert.
2. Jeder nach Phase II erreichbare Tandem-Typ wird in Phase III markiert.

Laufzeit

Jede Kante des Suffixbaumes wird in Phase III während der Skip/Count-Suche höchstens 2σ-mal benutzt.