# Seqs Zusammenfassung

November 14, 2023

# Contents

# 1 Suffix Arrays

Sei $S = c_1c_2...c_n$ ein String. der $i$-te Suffix $S_i$ ist definiert als der String $c_ic_{i+1}...c_n$. Ein Suffix Array nimmt die Menge aller $n$ Suffixe und sortiert sie lexikographisch. Dafür ist eine Ordnung $<$ auf dem Alphabet $\Sigma$ gegeben. Die naive Implementierung, die Substrings anhand ihrer ersten Character vergleicht, hat eine Laufzeit von $\mathcal{O}(n^2)$

## 1.1 Skew Algorithmus

Dieser ist ein Beispiel für einen linear-Zeit Algorithmus. Dieser wird im Folgenden am Beispiel $S = ctaataatg$ erklärt.
Schritt 1. Als erstes teilt der Algorithmus den String $S$ in zwei Teile. Sei $S_i$ der Substring beginnend an der Stelle $i$. Man definiert die Menge $S_L = \{S_i : \ i \mod 3 \neq 1\}$ und $S_R = \{S_i : \ i \mod 3 = 1\}$. Im Beispiel besteht $S_L$ aus

- taataatg
- aataatg
- aatg
- tg
- g

und $S_R$ aus

- ctaataatg
- ataatg
- atg

Die Suffixmenge von $S_L$ wird mittels Radixsort rekursiv in linearer Zeit sortiert, dabei wird nur bezüglich der ersten drei Stellen aller Suffixe sortiert. Dadurch erhält man

- aat ($S_3$)
- aat ($S_6$)
- g\$\$ ($S_9$)
- taa ($S_2$)
- taa ($S_5$)

- tg$ ($S_8$)

Wären nun alle dieser Tripel echt verschieden, so wäre diese Liste bereits total sortiert, was in diesem Beispiel aber nicht der Fall ist. Die Tripel werden anschließend von oben nach unten nummeriert, wobei gleiche Tripel die gleiche Nummer erhalten. Diese werden als lexikografische Namen bezeichnet. Man definiert $\overline{S}$ als der String der lexikografischen Namen der Substrings $S_i$ mit $i \mod 3 = 2$ konkateniert mit dem String der lexikografischen Namen der Substrings $S_i$ mit $i \mod 3 = 0$. Im Beispiel ist $\overline{S} = 334112$. Auf $\overline{S}$ wird der Algorithmus rekursiv angewandt. Von $\overline{S}$ erhalten wir die Suffixe

1. 334112
2. 34112
3. 4112
4. 112
5. 12
6. 2

Diese Liste wird im Sinne eines Suffixarrays sortiert, wodurch man

- 112
- 12
- 2
- 334112
- 34112
- 4112

erhält. Für jeden dieser Suffixe erhält man einen dazugehörigen Suffix vom ursprünglichen String $S$ und seine Position in $i$ in $S$. In der Tat erhält man die Zuordnungen: $\overline{S}_1 \equiv S_2$, $\overline{S}_2 \equiv s_5$, $\overline{S}_3 \equiv S_8$, ...
Im Allgemeinen gilt $\overline{S}_{\frac{i+1}{3}} \equiv S_i$ wenn $i = 3 \mod 2$. Für $i = 0 \mod 3$ hingegen gilt $\overline{S}_{\frac{n+i}{3}} \equiv S_i$. Dafür wird die Notation

$$\tau(i) = \begin{cases} \frac{i+1}{3}, & i = 2 \mod 3 \\ \frac{n+i}{3}, & i = 0 \mod 3 \end{cases}$$

Da wir aber die Richtung von $\overline{S}$ zu $S$ benötigen, brauchen wir die Umkehrabbildung von $\tau$:

$$\tau^{-1}(j) = \begin{cases} 3j-1, \ 1 \leq j \leq \frac{n}{3} \\ 3j-n, \ \frac{n}{3} < j \leq \frac{2n}{3} \end{cases}$$

Mittels $\tau^{-1}$ können nun die Suffixe von $\overline{S}$ in jene von $S$ umgerechnet werden, die dadurch sortiert sind. Schritt 2: In diesem Teil müssen die Suffixe in $S_R$ sortiert werden. In diesem Schritt wird jeder Suffix von seinem Anfangsbuchstaben getrennt. Man erhält

- $S_1 = cS_2$
- $S_4 = aS_5$
- $S_7 = aS_8$

Wendet man nun Radixsort auf die ersten Buchstaben an, so werden diese Strings automatisch sortiert.
Schritt 3: In diesem Schritt werden die beiden Sublisten wieder zu einem Suffixarray zusammengefügt. Wenn das in der naiven Merge-Sort Implementierung durchgeführt wird, so würde das in quadratischer Laufzeit resultieren. Deswegen wird wieder die Strategie von Schritt 2 verwendet; die Liste $S_R$ wird wieder aufgespalten in erste Buchstaben und Suffixe von $S_L$. Dadurch sind die beiden Suffixe aber nicht zwingend in der selben Gruppe, weswegen auch noch der zweite Buchstabe abgespalten werden kann. Der Merge Teil kann nun zuerst auf Basis der ersten beiden Buchstaben entschieden werden, außer wenn die beiden Buchstaben der zu vergleichenden Suffixe gleich sind. Dieses Verfahren funktioniert für Substrings $S_i$ aus $S_L$ mit $i = 0 \mod 3$. Für Substrings $S_i$ aus $S_L$ mit $i = 2 \mod 3$ hingegen würde die abgespaltenen Suffixe wieder in verschiedenen Listen liegen. Deshalb wird in diesem Fall nur ein Buchstabe abgespalten.
Der Vergleich der beiden abgespaltenen Suffixe funktioniert aber nur in linearer Zeit. Deswegen wird hierfür das sogenannte inverse Suffixarray eingeführt.

**Definition 1.1.1** (inverses Suffixarray (ISA))**.** Für ein Suffixarray $SA$ mit $k \to SA[k]$ ist das inverse Suffixarray definiert durch $ISA[SA[k]] = k$.

Allgemein gilt dann $S_i < S_j \Leftrightarrow ISA[\tau(i)] < ISA[\tau(j)]$.

## 1.2 Longest common prefixes

Das LCP-Array ist definiert auf der Sortierung des Suffix-Arrays. $LCP[i]$ speichert die Länge des längsten gemeinsamen Präfixes von $S_{SA[i-1]}$ und $S_{SA[i]}$. Insbesondere ist daher $LCP[1]$ undefiniert und wird daher auf -1 gesetzt. Algorithmus:

1. setze $LCP[1] = 1$ und $LCP[n+1] = -1$
2. konstruiere ISA aus SA (linear)
3. man beginnt mit dem längsten Suffix (bekommt man konstant aus ISA) und vergleicht mit dem Suffix in der Zeile oben
4. dieser Wert $l$ wird eingeschrieben und man geht zum zweitlängsten Suffix
5. Wieder vergleichen mit dem nächst-höheren Suffix im SA. Dort ist die Länge mindestens $l - 1$, daher erst ab dem $l$-ten Buchstaben vergleichen.
6. iterativ fortsetzen

### 1.3 LCP-Intervalle

Ein Intervall $[i, i]$ wird als Singleton bezeichnet. Ein Intervall $[i, j]$ in einem LCP Array mit $1 \leq i < j \leq n$ heißt LCP-Intervall mit LCP-Wert $l$, wenn

1. $LCP[i] \leq l$
2. $LCP[k] \geq l$ für alle $k$ mit $i + 1 \leq k \leq j$
3. $LCP[k] = l$ für mindestens ein $i + 1 \leq k \leq j$
4. $LCP[j + 1] < l$

Wenn man jeden String mit einem Dummy-Wert (z.B. \$), dann ist das Intervall $[1, n]$ immer ein LCP-Intervall.

**Lemma 1.3.1.** *Zwei LCP-Intervalle können sich nicht überlappen, d.h. es muss einer der folgenden Fälle eingetreten sein:*

1. *Intervall $[i, j]$ ist echt in $[p, q]$ enthalten oder*
2. *Intervalle $[i, j]$ und $[p, q]$ sind disjunkt*

**Definition 1.3.2.** Ein $m$-Intervall ist in einem $l$-Intervall enthalten, falls höchstens einer der Ränder gleich ist und $m > l$ gilt.
Das $l$-Intervall $[i, j]$ ist das umschließende Intervall von $[p, q]$, wenn $[p, q]$ in $[i, j]$ enthalten ist und es kein anderes in $[i, j]$ enthaltenes Intervall gibt, das $[p, q]$ enthält. Diese Intervalle werden Eltern und Kinder-Intervalle genannt.
Ein Intervall der Form $[k, k]$ wird Singleton genannt. Das Elternintervall ist das kleinste LCP-Intervall, dass den Singleton enthält.

### 1.3.1 Range Minimum Queries

Gegeben ist ein Array $A$ von $n$ ganzen Zahlen. Ein Range Minimum Query gibt für das Array $A$ und Indices $i, j$ ein $k$ zurück, sodass

$$A[k] = \min_{i \leq m \leq j} A[m]$$

So ein Query wird im Folgenden als $RMQ_A(i, j)$ bezeichnet. Man kann zeigen, dass $A$ in linearer Zeit bearbeitet werden kann, sodass das RMQ in konstanter Zeit ausgeführt werden kann.

**Definition 1.3.3.** Aus der Definition eines LCP-Arrays folgt, dass es einen Index $k$ gibt, an dessen Stelle der LCP-Wert $l$ angenommen wird. All diese Indizes werden in Folge als $l$-Index bezeichnet.

### 1.3.2 Kind-Intervalle finden

**Lemma 1.3.4.** *Sei $[i, j]$ ein $l$-Intervall. Seien $i_1 < i_2 < ... < i_k$ die $l$-Indizes in aufsteigender Reihenfolge. Dann sind die Kind-Intervalle gerade*

$$[i, i_1 - 1], [i_1, i_2 - 1], \ldots [i_k, i_{k+1} - 1], \ldots [i_k, j]$$

**Definition 1.3.5.** Für $2 \leq i \leq n$ definiere

$$PSV[i] = \max\{j | 1 \leq j < i \text{ und } LCP[i] < LCP[j]\}$$

und

$$NSV[i] = \min\{j | i < j \leq n + 1 \text{ und } LCP[i] < LCP[j]\}$$

**Lemma 1.3.6.** *Sei $2 \leq k \leq n$ und $LCP[k] = l$. Dann ist $[PSV[k], NSV[k] - 1]$ ein LCP Intervall mit LCP-Wert $l$*

**Lemma 1.3.7.** *Sei $[i, j] \neq [1, n]$ ein LCP-Intervall mit $LCP[i] = p$ und $LCP[j + 1] = q$.*

- *Ist $p = q$, dann*
  - *ist das Elternintervall $[PSV[i], NSV[i] - 1]$*
  - *hat das Elternintervall den LCP-Wert $p = q$*
  - *sind $i$ und $j + 1$ aufeinander folgende $p$-Indizes des Elternintervall*
- *Ist $p > q$, dann*

  - *ist* $[PSV[i], j]$ *das Elternintervall*
  - *hat das Elternintervall den LCP-Wert* $p$
  - *ist* $i$ *der letzte* $p$*-Index vom Elternintervall*

- *Ist* $p < q$*, dann*
  - *ist das Elternintervall* $[i, NSV[j+1]-1]$
  - *hat das Elternintervall den LCP-Wert* $q$
  - *ist* $j+1$ *der letzte* $q$*-Index des Elternintervalls*

**Lemma 1.3.8.** *Der LCP-Wert des Elternintervall von* $[i, j]$ *ist* $\max\{LCP[i], LCP[j+1]\}$.

## 1.4 Bottom-up Konstruktion des LCP-Baums

Man definiere $\langle p, a, b, A\rangle$ als das Viertupel mit LCP-Wert $p$, Intervallanfang $a$, -ende $b$ und die Liste der Kindintervalle $A$. Algorithmus 5 aus dem Skript funktioniert in linearer Zeit.

**Theorem 1.4.1.** *Sei top das oberste Intervall auf dem Stack und* $top_{-1}$ *das darunter für ein* $k$ *der For-Schleife. Bemerke* $top.lcp > top_{-1}.lcp$*. Dann, bevor top gepoppt wird, gilt*

- *Ist* $LCP[k] \leq top_{-1}.lcp$*, dann ist top ein Kindintervall von* $top_{-1}$.
- *Ist* $LCP[k] > top_{-1}.lcp$*, dann ist top ein Kindintervall des ersten LCP-Intervall mit LCP-Wert* $LCP[k]$*, das* $k$ *enthält. Genauer gesagt, ist top ein Kindintervall von* $[top.lb, NSV[k]-1]$.