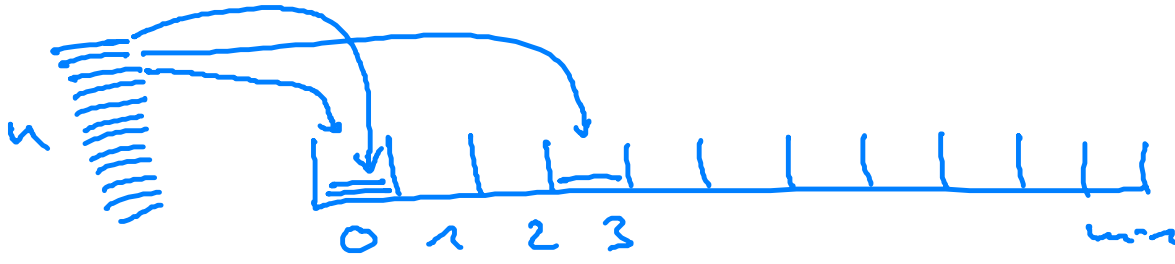


Ausage:

Tutorium von Judith fällt morgen (Mi: 10-12 Uhr) aus.

Bitte stattdessen diese Woche ein anderes Tutorium besuchen (z.B. Mona, Mi: 10-12 Uhr).

Bucket Sort



Array von Listen

Aufwand: $O(n + m)$

Bucket Sort zum Sortieren von Strings

Idee: Sortiere gegebene Strings lexikogr.

Algorithmus 1.2 (Bucketsort)

Input: Eine Liste L mit Strings A_1, A_2, \dots, A_n der Länge k mit $A_i = a_{i1}, a_{i2}, \dots, a_{ik}$ und $a_{ij} \in S = \{0, 1, \dots, m-1\}$

Output: Eine Permutation B_1, \dots, B_n von A_1, \dots, A_n mit $B_1 \leq_{\text{lex}} B_2 \leq_{\text{lex}} \dots \leq_{\text{lex}} B_n$

Methode:

1. Richte eine Queue Q ein und füge A_1, \dots, A_n in Q ein. ¹
2. Richte ein Array Bucket von m Buckets ein (wie beim einfachen Bucketsort)
3. **for** jede Stelle $r := k$ **downto** 1 **do**
 - 3.1 Leere alle Buckets Bucket[i]
 - 3.2 **while** Q nicht leer ist **do**
 - Sei A_j das erste Element in Q
 - Entferne A_j aus Q und füge es in Bucket[i] mit $i = a_{jr}$ ein
 - endwhile**
 - 3.3 Konkateniere die nichtleeren Buckets in die Queue Q
- endfor**

Beispiel:

Beispiel 1.3 Sei $S = \{0, 1\}$ und $0 < 1$. Seien $A_1 = 010, A_2 = 011, A_3 = 101, A_4 = 100$.

Wir sortieren zunächst nach der letzten Komponente: 010 011 101 100

Dadurch erhalten wir Folgendes:

$$\left. \begin{array}{l} 0: \underline{010} \ \underline{100} \\ 1: \underline{011} \ \underline{101} \end{array} \right\} \Rightarrow \underline{010} \ \underline{100} \ \underline{011} \ \underline{101}$$

Wir sortieren dann nach der zweitletzten Komponente: $\underline{010} \ \underline{100} \ \underline{011} \ \underline{101}$

Wir erhalten:

$$\left. \begin{array}{l} 0: \underline{100} \ \underline{101} \\ 1: \underline{010} \ \underline{011} \end{array} \right\} \Rightarrow \underline{100} \ \underline{101} \ \underline{010} \ \underline{011}$$

Nun die letzte Iteration, also Sortierung nach der ersten Komponente: $\underline{010} \ \underline{100} \ \underline{011} \ \underline{101}$

Wir erhalten:

$$\left. \begin{array}{l} 0: \underline{010} \ \underline{011} \\ 1: \underline{100} \ \underline{101} \end{array} \right\} \Rightarrow \underline{010} \ \underline{011} \ \underline{100} \ \underline{101}$$

Nach dem Durchlauf der for-Schleife stehen die Strings in folgender Reihenfolge in Q :

$r=3$	010	100	011	101	nach letzter Stelle sortiert
$r=2$	100	101	010	011	nach letzten 2 Stellen sortiert
$r=1$	010	011	100	101	nach letzten 3 Stellen sortiert

Es gilt die folgende Invariante:

Nach der i -ten Iteration sind die Strings nach den letzten i Stellen lexikographisch sortiert.

Beweis: Induktion über i

$i=1$: einfacher BucketSort liefert Beh.

Ind. Schluss: Betrachte zwei Fälle

a) Zwei Strings A und B , die in Iteration i im selben Bucket lagen

b) A und B in Iter. i in versch. Buckets.

Zu a) A und B haben dasselbe Zeichen an der i -letzten Stelle

$$l = k+1-i \quad a_1, \dots, a_{i-1}, a_i \quad b_1, \dots, b_{i-1}, b_i$$

Nach Induktion sind dann A und B

lexikogr. sortiert gemäß letztem $i-1$ Zeichen.
 \Rightarrow auch lex. korrekt sortiert gemäß
letztem i Stellen.

zu b) A und B haben verschiedene Zeichen
an Position $k+1-i \Rightarrow$ wurden in Iteration
 i so angeordnet, dass sie
lex. korrekt sortiert sind gemäß
letztem i Zeichen. \square

\Rightarrow Alg. arbeitet korrekt.

Aufwand: k mal Bucket Sort: $O(k \cdot (n+m))$

Untere Schranke für Sortieren mit paar-
weisen Vergleichen ist $\Omega(n \log n)$.

Frage: Ist $O(k \cdot (n+m))$ besser?

Falls wir Binärzahlen sortieren, die
paarweise verschieden sind, so ist

$$k \geq \log n$$

\Rightarrow Bucket Sort bringt hier nur Vorteile,
falls es viele identische Schlüssel

gibt.

Sortieren von Strings variabler Länge

Bsp: bab abc a

erste Idee: Verwende BucketSort für k -stellige Strings mit „unsichtbaren“ Zeichen an fehlenden Stellen in kurzen Strings.

$l_{\max} = 3$: bab abc auu mit $u < a, b, c$

Aufwand: $O(l_{\max} \cdot (n+m))$

in Bsp: $3 \cdot (3+4) = 21$

Es geht auch besser: $O(l_{\text{total}} + m)$

↑
Summe der
Längen aller
Strings.

in Bsp: $7+3=10$

Idee für besseren Algorithmus:

Sei ℓ_{max} max. Stringlänge

- 1) Sortiere Strings nach absteigender Länge
bab abc a
- 2) Verwende ℓ_{max} mal einfaches Bucket-Sort, aber betrachte in Iteration r nur Strings A_i , die an der r -ten Stelle ein Zeichen haben.
*r=3: bab, abc
r=2: " "*
- 3) Vermeide leere Buckets: Bestimme vorab die benötigten Buckets für jede Iteration und konkateniere dann nur diese.
*r=3: b, c
r=2: a, b
r=1: a, b*

Algorithmus 1.3 Input: Strings (Tupel) A_1, \dots, A_n

$$A_i = (a_{i1}, a_{i2}, \dots, a_{i\ell_i}), \quad a_{ij} \in \{0, \dots, m-1\}$$

(oder auch ein beliebiges anderes Alphabet)

$$\ell_{max} = \max_i \ell_i$$

Output: Permutation B_1, \dots, B_n von A_1, \dots, A_n mit $B_1 \leq_{lex} B_2 \leq_{lex} \dots \leq_{lex} B_n$

- NE**
1. Generiere ein Array von Listen NONEMPTY[] der Länge ℓ_{max} und für jedes $\ell, 1 \leq \ell \leq \ell_{max}$ eine Liste in NONEMPTY[ℓ], die angibt, welche Zeichen an einer der ℓ -ten Stellen vorkommen und welche Buckets daher in der (ℓ -ten) Iteration benötigt werden.

Dazu: $O(\ell_{total} + m)$

- 1.1 Erschaffe für jedes $a_{i\ell}, 1 \leq i \leq n, 1 \leq \ell \leq \ell_i$ ein Paar $(\ell, a_{i\ell})$ (das bedeutet: das Zeichen $a_{i\ell}$ kommt an ℓ -ter Stelle in einem der Strings vor)
- 1.2 Sortiere die Paare lexikographisch mit Algorithmus 1.2, indem man sie als zweistellige Strings betrachtet.
- 1.3 Durchlaufe die sortierte Liste der $(\ell, a_{i\ell})$ und generiere im Array NONEMPTY[], sortierte Listen, wobei das Array NONEMPTY[ℓ], $1 \leq \ell \leq \ell_{max}$ eine sortierte Liste aller $a_{i\ell}$ enthält. Dabei lassen sich auch gleich auf einfache Weise eventuell auftretende Duplikate entfernen.

*NE[1]: a, b
NE[2]: a, b
~~NE[3]: b, c~~*

2. Bestimme Länge ℓ_i jedes Strings und generiere Listen LENGTH[ℓ] aller Strings mit Länge ℓ (nur Referenzen auf die Strings in NONEMPTY[ℓ] speichern, daher nur $O(1)$ für Referenzen verhängen)

L $O(m + \ell_{max})$

Referenzen auf die Strings in LENGTH[l] verwalten, daher nur $O(1)$ für Referenzen umhängen)

3. Sortiere Strings analog zu Algorithmus 1.2.3, beginnend mit ℓ_{max} . Aber: $O(\ell_{total})$

- nach der r -ten Phase enthält Q nur die Strings der Länge $\geq \ell_{max} - r + 1$; diese sind lexikographisch korrekt sortiert bezüglich der letzten r Komponenten.
- NONEMPTY[] wird benutzt, um die Listen in BUCKET[] neu zu generieren und außerdem zur schnelleren Konkatenation der Einzellisten. Dies ist nötig, weil wir nur die nichtleeren Buckets verwalten wollen.
- vor dem $r + 1$ -ten Durchlauf wird LENGTH[$\ell_{max} - r$] am Anfang² der Queue Q eingefügt. Die kurzen Strings stehen dann am Anfang und damit am lexikographisch richtigen Platz, falls sie mit anderen im selben Bucket landen.

Bsp: $r=1$: Q : bab abc
 LE[3] Buckets

~~b: bab~~ ~~c: abc~~ $\xrightarrow{\text{konkat.}}$ Q : bab, abc
 NE[3]

$r=2$: Q : bab, abc
 LE[2]

~~a: bab~~ ~~b: abc~~ $\xrightarrow{\text{konkat.}}$ Q : bab, abc
 NE[2]

$r=2$: Q : a, bab, abc
 LE[1]

~~a: a, abc~~ ~~b: bab~~ $\xrightarrow{\text{konkat.}}$ Q : a, abc, bab
 NE[1]

Korrektheit des Alg. folgt aus Korrektheit von Algor. 2 und der Tatsache, dass die neuen (kurzen) Strings jeweils am Beginn der Liste eingefügt werden.

Aufwand: Vorbereitung: $O(\ell_{total}) + O(\ell_{total} + m)$

Sortierphase: An der Stelle ℓ :

u_e Strings

w_e Buckets

→ Aufwand $O\left(\sum_{e=1}^{l_{\max}} (u_e + w_e)\right)$

$= O\left(\underbrace{\sum_{e=1}^{l_{\max}} u_e}_{l_{\text{total}}}\right)$

$= O(l_{\text{total}})$

