1 Sortieren durch Fachverteilung

In Abschnitt 1.1 besprechen wir das Verfahren BucketSort zum Sortieren einer gegebenen Folge von Zahlen (oder, allgemeiner, von Objekten einer linear geordneten Menge). In Abschnitt 1.2 bespreche wir das Konzept der lexikographischen Ordnung auf Strings über einem linear geordneten Alphabet. Diese Strings lassen sich auch als Zahlentupel auffassen. Abschnitt 1.3 ist dem Verfahren RadixSort zum Sortieren von Strings (oder Zahlentupeln) gleicher Länge gewidmet. In Abschnitt 1.4 besprechen wir zwei Varianten von RadixSort. Die erste Variante sortiert weiterhin Zahlentupel (oder Strings) gleicher Länge, erzielt aber eine bessere Laufzeit. Die zweite Variante erlaubt das effiziente Sortieren von Zahlentupeln (oder Strings) von i.A. verschiedener Länge.

1.1 Sortieren von Zahlen mit BucketSort

Eine linear geordnete Mange Σ der Größe m kann auf die offensichtliche Weise mit der Zahlenmenge $\{0, 1 \ldots, m-1\}$ identifiziert werden. Das Sortieren einer gegebenen Folge von Elementen aus M entspricht dann dem Sortieren einer Folge $a_1, \ldots, a_n$ von Zahlen aus dem Bereich $\{0, 1 \ldots, m-1\}$. Dies führt uns zu folgendem Sortierproblem:

Problem 1: Sortiere eine gegebene Folge $a_1, \ldots, a_n$ von Zahlen aus dem Bereich $\{0, 1, \ldots, m-1\}$.

Methode: BucketSort

1. Stelle leere Queues $Q_0, Q_1, \ldots, Q_{m-1}$, genannt „Buckets“, bereit (Zeit: $O(m)$).
2. Inspiziere die Folge $a_1, \ldots, a_n$ von links nach rechts und füge a_i in Q_{a_i} ein (Zeit: $O(n)$).
3. Produziere die sortierte Ausgabeliste durch Konkatenation von $Q_0, Q_1, \ldots, Q_{m-1}$ (Zeit: $O(m)$).

Offensichtlich gilt:

Satz 1.1 *BucketSort löst Problem 1 in $O(m+n)$ Schritten.*

Beachte, dass die Schranke $m + n$ asymptotisch unterhalb von $n \log(n)$ liegt, sofern $m = o(n \log(n))$.

Frage: Wieso steht Satz 1.1 nicht im Widerspruch zu der früher bewiesenen unteren Schranke $n/2 \log(n/2)$ für die Anzahl der zum Sortieren benötigten Schlüsselvergleiche?

1.2 Die lexikographische Ordnung auf Zahltupeln

Es sei „$<$“ eine (irreflexive) lineare Ordnung auf einem Alphabet[1] Σ. Weiter sei Σ^* die Menge aller Wörter (= endlichen, evtl. leeren, Folgen) über Σ. Dann können wir „$<$“ zu einer linearen Ordnung auf Σ^*, genannt *lexikographische Ordnung*, fortsetzen wie folgt:

Definition 1.2 *Die Folge $u = (u_1, \ldots, u_r)$ ist kleiner als die Folge $v = (v_1, \ldots, v_s)$, notiert als $u < v$, falls eine der folgenden beiden Bedingungen erfüllt ist:*

(i) u ist ein echter Präfix von v, d.h., $r < s$ und die ersten r Komponenten von u und v stimmen überein.

(ii) Es existiert ein $i \in \{1, \ldots, \min\{r, s\}\}$, so dass $u_i < v_i$ und die ersten $i - 1$ Komponenten von u und v stimmen überein.

Wenn Σ das lateinische Alphabet ist, versehen mit der natürlichen Ordnung von a bis z, dann liefert Definition 1.2 die in einem Lexikon verwendete Ordnung.

Beispiel 1.3 *„aal<aalglatt“ (Bedingung (i)) und „tangente<tango“ (Bedingung (ii)).*

Bemerkung 1.4 *1. Wie früher bereits erwähnt, kann eine linear geordnete Menge Σ der Größe $|\Sigma| = m$ mit der Menge $\{0, 1, \ldots, m - 1\}$ identifiziert werden.*

2. Wörter aus Σ^k (feste Wortlänge k) sind interpretierbar als m-näre Zahlendarstellungen:

[1] Alphabet = endliche Menge.

(a) *Die kleinste darstellbare Zahl ist die Null.* $(0, \ldots, 0)$ *ist das zugehörige* k*-Tupel.*

(b) *Die größte darstellbare Zahl ist* $m^k - 1$*;* $(m-1, \ldots, m-1)$ *ist das zugehörige* k*-Tupel.*

(c) *Allgemein gilt: das* k*-Tupel* $A = (a_{k-1}, \ldots, a_1, a_0)$ *stellt die Zahl* $\sum_{i=0}^{k-1} a_i m^i$ *dar.*

Bei fester Wortlänge k *stimmt die lexikographische Ordnung überein mit der natürlichen Ordnung auf Zahlen.*

3. *Das Konzept der Lexikographischen Ordnung ist (auf die offensichtliche Weise) verallgemeinerbar auf den Fall verschiedener Alphabete für verschiedene Positionen in der Folge. S. dazu das folgende Beispiel 1.5.*

Beispiel 1.5 *Die Tripel (Jahr,Monat,Tag)*$\in \{1900, \ldots, 2100\} \times \{1, \ldots, 12\} \times \{1, \ldots, 31\}$ *benutzen in jeder Komponente ein anderes Alphabet.*

1.3 Sortieren von Zahltupeln mit RadixSort

In diesem Abschnitt beschäftigen wir uns mit dem folgenden

Problem 2: Sortiere gemäß der lexikographischen Ordnung eine gegebene Folge $A_1, \ldots, A_n$ von k-Tupeln mit Komponenten aus dem Bereich $\{0, 1, \ldots, m-1\}$.

Notation: $A_i = (a_{i1}, \ldots, a_{ik}) \in \{0, 1, \ldots, m-1\}^k$.

Plan: Wende BucketSort komponentenweise an in Richtung von rechts (also Komponente k) nach links (also Komponente 1).

Resultierende Methode: RadixSort

1. Initialisiere eine Queue Q mit $A_1, \ldots, A_n$.
2. Für $j = k, k-1, \ldots, 1$ mache Folgendes:
   (a) Stelle $Q_0, Q_1, \ldots, Q_{m-1}$ als (zunächst) leere Queues bereit.
   (b) Solange Q nicht leer ist, entnimm Q das vorderste Element, sagen wir A_i, und füge es in $Q_{a_{ij}}$ ein.
   (c) Rekonfiguriere Q neu als die Queue, die sich aus der Konkatenation von $Q_0, Q_1, \ldots, Q_{m-1}$ ergibt.

Beispiel 1.6 *Wir machen die folgenden „Schnappschüsse“ von einem Beispiellauf von RadixSort mit den Parametern $n = 8$, $m = 10$ und $k = 3$. Die Eingabe $A_1, \ldots, A_n$ ergibt sich aus der Initialisierung von Q (s. unten).*

Anfangs: *$Q_0, Q_1, \ldots, Q_{m-1}$ sind leer.*

$$Q = 087, 754, 750, 532, 501, 001, 539, 437 \ .$$

Nach Iteration j=3:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
750	501	532	–	754	–	–	087	–	539
	001						437		

$$Q = 750, 501, 001, 532, 754, 087, 437, 539 \ .$$

Nach Iteration j=2:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
501	–	–	532	–	750	–	–	087	–
001			437		754				
			539						

$$Q = 501, 001, 532, 437, 539, 750, 754, 087 \ .$$

Nach Iteration j=1:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
001	–	–	–	437	501	–	750	–	–
087					532		754		
					539				

$$Q = 001, 087, 437, 501, 532, 539, 750, 754 \ .$$

Dass sich im obigen Beispiellauf am Ende eine lexicographische Sortierung ergibt ist kein Zufall:

Satz 1.7 (Korrektheit von RadixSort) *Nach Terminieren von RadixSort befinden sich in Q die Eingangsdaten $A_1, \ldots, A_n$ in lexikographischer Sortierung.*

Beweis Die lexikographische Ordnung bezüglich der Komponenten $j, \ldots, k$ von Zahlen-k-Tupeln notieren wir als $<_j^k$. Beachte, dass $<_k^k$ mit der Ordnung bezüglich der k-ten Komponente und dass $<_1^k$ mit der gewöhnlichen lexikographischen Ordnung übereinstimmt. Nach jeder Iteration der Hauptschleife von RadixSort gilt die folgende Bedingung:

- Q enthält die Eingangsdaten $A_1, \ldots, A_n$ in Sortierung gemäß $<_j^k$.

Aus dieser Invarianzbedingung ergibt sich die Korrektheit: nach der k-ten Iteration mit $j = 1$ sind die Daten bezüglich aller Komponenten $1, \ldots, k$ lexikographisch sortiert. Die Bedingung selbst ist leicht verifizierbar mit Induktion über die Anzahl der Iterationen:

Induktionsanfang: Nach der ersten Iteration mit $j = k$ sind die Daten offensichtlich gemäß der k-ten Komponente (und somit gemäß $<_k^k$) sortiert.

Induktionsschritt: Wir betrachten eine Iteration mit $j < k$. Wir können induktiv voraussetzen, dass vor dieser Iteration die Daten gemäß $<_{j+1}^k$ sortiert sind. Die aktuelle Iteration hat den folgenden Effekt:

- Falls $A_i <_j^k A_{i'}$ wegen $a_{ij} < a_{i'j}$, dann landet in Phase 2(b) A_i in $Q_{a_{ij}}$ und $A_{i'}$ in $Q_{a_{i'j}}$. In Phase 2(c) wird somit A_i vor $A_{i'}$ in Q platziert.
- Falls $A_i <_j^k A_{i'}$ und $a_{ij} = a_{i'j}$, so gilt auch $A_i <_{j+1}^k A_{i'}$. Dann wird A_i in Phase 2(b) vor $A_{i'}$ in $Q_{a_{ij}}$ eingefügt und dementsprechend steht A_i nach Phase 2(c) vor $A_{i'}$ in Q.

Aus der obigen Diskussion ergibt sich die Korrektheit von RadixSort. •

RadixSort besteht im Wesentlichen aus k-maliger Anwendung von BucketSort. Anstatt k-Tupel in Queues einzufügen (oder gar Records R, bei denen das k-Tupel nur eine Komponente R.key darstellt) fügen wir stets nur einen Zeiger auf das betreffende k-Tupel ein. Daher kann eine Iteration der Hauptschleife in $O(n+m)$ Schritten durchgeführt werden und es ergibt sich folgendes Resultat:

Satz 1.8 *RadixSort benötigt zum lexikographischen Sortieren von n k-Tupeln mit Komponenten aus dem Bereich $\{0, 1, \ldots, m-1\}$ eine Rechenzeit der Größenordnung $O(k \cdot (n+m))$.*

Falls die Zahlen in den j-ten Komponenten der k-Tupel aus dem Bereich $\{0, 1, \ldots, m_j - 1\}$ stammen (vgl. mit Beispiel 1.5), dann benötigt die betreffende Iteration der Hauptschleife von RadixSort $O(n + m_j)$ Rechenschritte. Es egibt sich daher die folgende Verallgemeinerung von Satz 1.8:

Satz 1.9 *RadixSort benötigt zum lexikographischen Sortieren von n k-Tupeln mit j-ten Komponenten aus dem Bereich $\{0, 1, \ldots, m_j - 1\}$ eine Rechenzeit der Größenordnung $O\left(kn + \sum_{j=1}^{k} m_j\right)$.*

Eine Zahl $a \in \{0, 1, \ldots, n^k - 1\}$ kann via

$$a = \sum_{i=0}^{k-1} a_i n^i \text{ mit } 0 \leq a_i \leq n - 1$$

als k-Tupel $A = (a_{k-1}, \ldots, a_1, a_0)$ dargestellt werden. Zudem kann das k-Tupel A zu gegebenem a, oder umgekehrt a aus A, leicht in $O(k)$ Schritten berechnet werden. Damit ergibt sich folgende Anwendung von RadixSort:

Satz 1.10 *n Zahlen im Bereich $\{0, 1, \ldots, n^k - 1\}$ können in $O(kn)$ Schritten sortiert werden.*

1.4 Verbesserte Varianten von RadixSort

In diesem Abschnitt stellen wir uns zwei Ziele:

1. Laufzeit $O(kn + m)$ statt $O(k(n + m))$ bei der Sortierung von Strings fester Wortlänge k (also von k-Tupeln) .

2. Laufzeit $O(L + m)$ bei der Sortierung von Strings verschiedener Länge, wobei L die Gesamtlänge aller Strings bezeichnet.

Zu Ziel 1 merken wir Folgendes an:

Beobachtung: In den Phasen 2(a) und 2(c) verschwenden wir Rechenzeit durch die (sinnlose) Manipulation von leeren Buckets.

Plan: Berechne im Voraus für jedes $j \in \{0, 1, \ldots, m - 1\}$ eine sortierte Liste NONEMPTY[j], welche die Zahlen aus

$$\{a_{ij} |\ i = 1, \ldots, n\}$$

enthält. Das sind nämlich exakt die Adressen der in dieser Iteration nichtleeren Buckets.

Die Vorteile dieser Vorausberechnung sind, ein paar leichte Modifikationen in der Hauptsschleife vorausgesetzt, wie folgt:

1. In Phase 2(a) der Hauptschleife brauchen nur die nichtleeren Buckets der vorangegangenen Iteration (also die Q_i mit $i \in \text{NONEMPTY}[j-1]$) geleert werden. (Die anderen sind bereits leer.)

2. In Phase 2(c) der Hauptschleife werden nur die nichtleeren Buckets Q_i mit $i \in \text{NONEMPTY}[j]$ der Reihe nach konkateniert.

Eine Iteration dauert dann nur noch $O(n)$ statt $O(m)$ Schritte und wir gelangen zur Zeitschranke $O(kn)$ plus die Zeit zur Vorausberechnung der NONEMPTY-Listen. Das folgende Verfahren berechnet diese Listen in $O(kn+m)$ Rechenschritten:

1. Initialisiere in $O(kn)$ Schritten eine Queue Q' mit den Elementen der Multimenge
$$\{(j, a_{ij}) |\ 1 \leq i \leq n, 1 \leq j \leq k\}\ .$$

2. Sortiere in $O(kn+m)$ Schritten die Paare aus Q' lexikographisch mit Hilfe von RadixSort, so dass das (sortierte) Ergebnis wieder in Q' steht.

3. Arbeite in $O(kn)$ Schritten die Paare in Q' der Reihe nach ab und baue parallel dazu die sortierten NONEMPTY-Listen auf.

Wir fassen das Ergebnis zusammen:

Satz 1.11 *Die oben beschriebene verbesserte Variante von RadixSort sortiert n k-Tupel mit Komponenten aus $\{0, 1, \ldots, m-1\}$ in $O(kn+m)$ Schritten.*

Beispiel 1.12 *Gegeben seien die folgenden 8 Tripel mit Komponenten aus $\{0, 1, \ldots, 9\}$ (die selben Eingabedaten wie in Beispiel 1.6):*

$$087, 754, 750, 532, 501, 001, 539, 437$$

Diese Liste können wir von links nach rechts durchlesen und parallel dazu die entsprechenden Paare in Q' aufnehmen:

$$\begin{aligned} Q' \ = \ & (1,0),(2,8),(3,7),(1,7),(2,5),(3,4),(1,7),(2,5),(3,0),(1,5),(2,3),(3,2), \\ & (1,5),(2,0),(3,1),(1,0),(2,0),(3,1),(1,5),(2,3),(3,9),(1,4),(2,3),(3,7) \end{aligned}$$

Nach Anwendung von RadixSort enthält Q' die selben Paare wie zuvor, aber in sortierter Form:

$$\begin{aligned} Q' \;=\; & (1,0),(1,0),(1,4),(1,5),(1,5),(1,5),(1,7),(1,7),(2,0),(2,0),(2,3),(2,3),\\ & (2,3),(2,5),(2,5),(2,8),(3,0),(3,1),(3,1),(3,2),(3,4),(3,7),(3,7),(3,9) \end{aligned}$$

Hieraus ergeben sich die folgenden NONEMPTY-Listen:

$$\begin{aligned} \mathit{NONEMPTY}[1] &= 0,4,5,7\\ \mathit{NONEMPTY}[2] &= 0,3,5,8\\ \mathit{NONEMPTY}[3] &= 0,1,2,4,7,9 \end{aligned}$$

Wenden wir uns nun dem zweiten Ziel dieses Abschnittes zu: der lexikographischen Sortierung von Strings bzw. Zahlentupeln verschiedener Länge.

Problem 3: Sortiere gemäß der lexikographischen Ordnung eine gegebene Folge $A_1,\ldots,A_n$ von Tupeln der Längen $\ell_1,\ldots,\ell_n$ mit Komponenten aus dem Bereich $\{0,1,\ldots,m-1\}$. (Somit gilt $A_i \in \{0,1\ldots,m-1\}^{\ell_i}$.)

Wir setzen $k := \max_{i\in[n]} \ell_i$. Für die Gesamtlänge $L := \sum_{i\in[n]} \ell_i$ aller Tupel gilt dann $L \leq kn$, aber L kann i.A. sehr viel kleiner als kn sein (zum Beispiel, wenn sich unter den A_i ein sehr langes und viele kleine Tupel befinden).

Bemerkung 1.13 *Eine Sortierung mit RadixSort bzw. verbessertem RadixSort würde in $O(k(n+m))$ bzw. $O(kn+m)$ Schritten ablaufen.*

Dies ist jedoch im Falle $L << kn$ nicht zufriedenstellend. Wir streben vielmehr eine Laufzeit $O(L+m)$ an. Dies kann erreicht werden

- durch Verwendung der uns bereits bekannten NONEMPTY-Listen,
- durch Verwendung von LENGTH-Listen, die uns helfen werden, für Tupel einer Länge ℓ in den Iterationen mit $j = k, k-1, \ldots, \ell+1$ keine Zeit aufzuwenden. Vielmehr werden diese Tupel erst in der Iteration mit $j = \ell$ ins Spiel gebracht.

Diese Überlegungen führen zu der folgenden Variante von RadixSort:

1. Bestimme zu jedem Tupel A_i seine Länge ℓ_i und baue anschließend die LENGTH-Listen auf, wobei LENGTH[j] alle A_i mit $\ell_i = j$ enthält.[2] Dies lässt sich in $O(L)$ Schritten bewerkstelligen.

2. Initialisiere in $O(L)$ Schritten eine Queue Q' mit den Elementen der Multimenge
$$\{(j, a_{ij}) |\ 1 \leq i \leq n, 1 \leq j \leq \ell_i\}\ .$$
(Diese Multimenge hat die Größe L.)

3. Sortiere in $O(L + m)$ Schritten die Paare aus Q' lexikographisch mit Hilfe von RadixSort, so dass das (sortierte) Ergebnis wieder in Q' steht.

4. Arbeite die Paare in Q' der Reihe nach ab und baue parallel dazu in $O(L)$ Schritten die NONEMPTY-Listen auf (in der gleichen Weise wie bei der uns schon bekannten Verbesserung von RadixSort).

Die gesamte Vorausberechnung der LENGTH- und NONEMPTY-Listen kann in Zeit $O(L + m)$ abgeschlossen werden.

Die Vorteile dieser Vorausberechnung sind, ein paar leichte Modifikationen in den Phasen 1 und 2 von RadixSort vorausgesetzt, wie folgt:

1. In Phase 1 von RadixSort wird Q als leere Queue initialisiert.

2. In Phase 2(a) der Hauptschleife brauchen nur die nichtleeren Buckets der vorangegangenen Iteration (also die Q_i mit $i \in$ NONEMPTY[$j-1$]) geleert werden. (Die anderen sind bereits leer.)

3. In Phase 2(b) der Hauptschleife fügen wir die in LENGTH[j] enthaltenen Tupel A_i *vorne* in Q ein. Q enthält zu diesem Zeitpunkt die Daten A_i mit $\ell_i = j$, gefolgt von den Daten A_i mit $\ell_i \geq j + 1$ (letztere in der Sortierung gemäß „$<^k_{j+1}$“)[3].

4. In Phase 2(c) der Hauptschleife werden nur die nichtleeren Buckets, also die Buckets Q_i mit $i \in$ NONEMPTY[j], der Reihe nach konkateniert.

[2] In einer realen Implementierung würde man die Adressen der A_i enthaltenden Rekords statt der A_i in die LENGTH-Listen einfügen.

[3] Da Tupel einer Länge j in den Komponenten $j+1, \ldots, k$ eine leere Sequenz repräsentieren, und diese kleiner ist als jede nichtleere Sequenz, sind *alle* in Q befindlichen Daten (also auch die aus LENGTH[j] am Anfang von Q) vor Phase 2(c) gemäß „$<^k_{j+1}$“ sortiert. Diese Beobachtung wäre wichtig für einen Korrektheitsbeweis.

Satz 1.14 (Korrektheit der „Problem 3“-Variante von RadixSort)
Nach Terminieren von RadixSort stehen die Daten $A_1, \ldots, A_n$ in Q in lexikographischer Sortierung.

Den Beweis, der ähnlich zum Beweis von Satz 1.7 verläuft, lassen wir aus.

Satz 1.15 (Laufzeit der „Problem 3“-Variante von RadixSort)
RadixSort terminiert nach $O(L + m)$ Rechenschritten.

Beweis Wir hatten bereits früher angemerkt, dass die Vorausberechnung der NONEMPTY- und LENGTH-Listen in Zeit $O(L + m)$ erfolgen kann. Die weitere Anzahl der Rechenschritte wird durch Phase 2 (die Hauptschleife) von RadixSort dominiert. Der Gesamtaufwand für die Phasen 2(a) und 2(c) lässt sich abschätzen durch die Gesamtlänge der NONEMPTY-Listen. Da jedes Tupel A_i einen Beitrag von maximal ℓ_i zu dieser Gesamtlänge leistet, erhalten wir die Schranke $O(L)$. Um den Gesamtaufwand für die Phase 2(b) abzuschätzen, verwenden wir folgende Buchhaltung: wann immer ein A_i aus Q inspiziert wird (um es in $Q_{a_{ij}}$ einzufügen), belasten wir A_i mit Kosten 1. Da ein A_i nur während der Iterationen $j = \ell_i, \ldots, 2, 1$ in Q verweilt, wird jedes A_i mit Kosten ℓ_i belastet. Daher beträgt der Gesamtaufwand für die Phase 2(b) ebenfalls $O(L)$. Aus dieser Diskussion ergibt sich die Gesamtlaufzeit $O(L + m)$. •