Datenstrukturen und Algorithmen
Übung 4 – Sortieren II

Musterlösung

Theoretische Aufgaben

1. Erläutern Sie analog zu Abbildung 8.2 im Buch, wie COUNTING-SORT auf dem Feld

$$A = \langle 2,7,1,3,2,4,1,8,5,1,4 \rangle$$

arbeitet. **(1 Punkt)**

$C = \langle 0,3,2,1,2,1,0,1,1 \rangle$

Nach akkumulieren:

$C = \langle 0,3,5,6,8,9,9,10,11 \rangle$

Ablauf:

C									A (Eingabe)											B (Ausgabe)										
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8	9	10	11
0	3	5	6	8	9	9	10	11	2	7	1	3	2	4	1	8	5	1	4											
0	3	5	6	7	9	9	10	11	2	7	1	3	2	4	1	8	5	1									4			
0	2	5	6	7	9	9	10	11	2	7	1	3	2	4	1	8	5					1					4			
0	2	5	6	7	8	9	10	11	2	7	1	3	2	4	1	8						1					4	5		
0	2	5	6	7	8	9	10	10	2	7	1	3	2	4	1							1					4	5		8
0	1	5	6	7	8	9	10	10	2	7	1	3	2	4							1	1					4	5		8
0	1	5	6	6	8	9	10	10	2	7	1	3	2								1	1				4	4	5		8
0	1	4	6	6	8	9	10	10	2	7	1	3									1	1		2		4	4	5		8
0	1	4	5	6	8	9	10	10	2	7	1										1	1		2	3	4	4	5		8
0	0	4	5	6	8	9	10	10	2	7										1	1	1		2	3	4	4	5		8
0	0	4	5	6	8	9	9	10	2											1	1	1		2	3	4	4	5	7	8
0	0	3	5	6	8	9	9	10												1	1	1	2	2	3	4	4	5	7	8

2. Gegeben sind n ganze Zahlen zwischen 0 und k. Geben Sie einen Algorithmus an, der die Eingabe vorbearbeitet und dann jede Anfrage der Art, wie viele der n Zahlen in ein Intervall $[a \ldots b]$ fallen, in Zeit $\mathcal{O}(1)$ beantwortet. Ihr Algorithmus sollte für den Vorbearbeitungsschritt mit der Zeit $\mathcal{O}(n+k)$ auskommen. **(1 Punkt)**

Die Idee ist wie beim COUNTING-SORT die Elemente zu zählen und zu akkumulieren, um so in $\mathcal{O}(1)$ abfragen zu können, wie viele Elemente $\leq x$ sind. Mit zwei Abfragen kann man so die Grösse eines ganzen Bereichs zählen.

```
PREPARE(A)
 1  // B[i]: Anzahl Vorkommnisse des Werts i in A, mit 0 initialisiert
 2  B := new Array[0...k]
 3
 4  for i ∈ [1...A.length]
 5      B[A[i]] := B[A[i]] + 1
 6
 7  // C[i]: Anzahl Vorkommnisse von Werten ≤ i in A, mit 0 initialisiert
 8  C := new Array[0...k]
 9  C[0] := B[0]
10
11  for i ∈ [1...k]
12      C[i] = C[i − 1] + B[i]
13
14  return C

COUNT-BETWEEN(C, a, b)
1  // l_a: Anzahl Werte in A, die < a sind
2  if a > 0
3      l_a := C[a − 1]
4  else
5      l_a := 0
6
7  // le_b: Anzahl Werte in A, die ≤ b sind
8  le_b := C[b]
9  return le_b − l_a
```

3. Beschreiben Sie analog zu Abbildung 8.3 im Buch die Arbeitsweise von RADIX SORT angewendet auf die folgende Liste von Wörtern: *NDB, MND, NSA, CIA, BND, MAD, BFV, FSB, KGB, BVT, DND, NIS, NSB*. **(1 Punkt)**

N	D	B
M	N	D
N	S	A
C	I	A
B	N	D
M	A	D
B	F	V
F	S	B
K	G	B
B	V	T
D	N	D
N	I	S
N	S	B

N	S	A
C	I	A
N	D	B
F	S	B
K	G	B
N	S	B
M	N	D
B	N	D
M	A	D
D	N	D
N	I	S
B	V	T
B	F	V

M	A	D
N	D	B
B	F	V
K	G	B
C	I	A
N	I	S
M	N	D
B	N	D
D	N	D
N	S	A
F	S	B
N	S	B
B	V	T

B	F	V
B	N	D
B	V	T
C	I	A
D	N	D
F	S	B
K	G	B
M	A	D
M	N	D
N	D	B
N	I	S
N	S	A
N	S	B

4. (a) Welche der folgenden Sortieralgorithmen sind (in der Variante aus der Vorlesung) stabil: INSERTIONSORT, MERGESORT, HEAPSORT, QUICKSORT? Begründen Sie kurz. **(0.4 Punkte)**

   - INSERTIONSORT: **Stabil**: Der Code ist äquivalent dazu, Elemente nur mit ihren direkten Nachbarn zu tauschen — und das nicht, wenn sie den gleichen Wert haben. Wenn also zwei Elemente in ihrer Reihenfolge vertauscht wurden, mussten sie verschiedene Schlüssel-Werte haben.

- MERGESORT: **Stabil**, wenn wie in Buch/Vorlesung implementiert. Essentiell dafür ist, dass im Merge-Schritt bei Gleichheit der Schlüssel das Element aus dem linken Teilfeld gewählt wird.
- HEAPSORT: **Nicht stabil**: Im HEAPIFY-Schritt können zwei Elemente mit gleichem Schlüssel die Position wechseln
- QUICKSORT: So wie vorgestellt **Nicht stabil**, das Einordnen des Pivotelements kann einen Tausch verursachen. Nur mit zusätzlichem Speicheraufwand ist Stabilisierung möglich.

(b) Geben Sie ein einfaches Schema an, nach dem *beliebige* vergleichende Sortieralgorithmen stabilisiert werden können. Wie viel zusätzliche Zeit und wie viel zusätzlicher Speicherplatz zieht Ihr Schema nach sich? **(0.6 Punkte)**

Eine mögliche Variante ist es, bei einer Eingabe $A[i]$ mit Schlüsseln $k[i]$ einen Schlüssel $k'[i] := \langle k[i], i\rangle$ zu definieren, der lexikalisch vergleichen wird. So gilt für $i < j$ auch bei $k[i] = k[j]$ trotzdem noch $k'[i] < k'[j]$, ansonsten ist die Ordnung aber äquivalent. Nun kann man anhand von k' sortieren und erhält immer ein stabiles Verfahren. Der zusätzliche Speicher für dieses Verfahren ist $\Theta(n \lg n)$ (zusätzlicher Schlüssel-Speicher, n Elemente zu je $\lg n$ bits). Es fallen ebenfalls $\Theta(n \lg n)$ an Zeitaufwand für das Erstellen dieser Schlüssel an. Beim Ausführen des Algorithmus kommt pro Vergleich ein konstanter Zeitaufwand hinzu, an der asymptotischen Aufzeit ändert das jedoch nichts.

5. Erläutern Sie analog zu Abbildung 8.4 die Arbeitsweise von BUCKET-SORT angewendet auf das Feld

$$A = \langle 0.79, \quad 0.13, \quad 0.16, \quad 0.64, \quad 0.39, \quad 0.20, \quad 0.89, \quad 0.53, \quad 0.71, \quad 0.42\rangle$$

(1 Punkt)

Nach sortieren in Kübel:

[0.0,0.1)	[0.1,0.2)	[0.2,0.3)	[0.3,0.4)	[0.4,0.5)	[0.5,0.6)	[0.6,0.7)	[0.7,0.8)	[0.8,0.9)	[0.9,1.0)
	0.16	0.2	0.39	0.42	0.53	0.64	0.71	0.89	
	0.13						0.79		

Nach sortieren der einzelnen Kübel:

[0.0,0.1)	[0.1,0.2)	[0.2,0.3)	[0.3,0.4)	[0.4,0.5)	[0.5,0.6)	[0.6,0.7)	[0.7,0.8)	[0.8,0.9)	[0.9,1.0)
	0.13	0.2	0.39	0.42	0.53	0.64	0.71	0.89	
	0.16						0.79		

Sortiertes Ergebnis (konkatenierte Kübel):

$A = \langle 0.13, 0.16, 0.2, 0.39, 0.42, 0.53, 0.64, 0.71, 0.79, 0.89\rangle$

6. Gegeben sei ein Feld mit ganzzahligen Werten, wobei die einzelnen Zahlen unterschiedlich viele Stellen haben können. Die Gesamtanzahl der Stellen *aller* Zahlen des Feldes hat jedoch den festen Wert n. Zeigen Sie, wie das Feld in der Zeit $\mathcal{O}(n)$ sortiert werden kann. **(1 Punkt)**

Wir gehen ohne Beschränkung der Allgemeinheit davon aus, dass die Zahlen in einer festen Basis (z.B. 10) gegeben sind, keine führenden Nullen enthalten und positiv sind.

Mit RADIX-SORT können wir n Zahlen mit bis zu k Ziffern in $\mathcal{O}(n \cdot k)$ sortieren. Eine direkte Anwendung ist also nicht genug, im Worst-Case wäre die Laufzeit damit Θn^2!

Ebenfalls wissen wir, dass eine nichtnegative Ganzzahlen (ohne führende Nullen) mit weniger Ziffern immer kleiner sind als eine mit mehr Ziffern.

Die maximale Länge k einer Zahl in der Eingabe ist durch n nach oben begrenzt.

1. Sortiere die Eingabe mit Hilfe von COUNTING-SORT aufsteigend nach ihrer Länge in k Teilfelder der Längen $m_1, \ldots, m_k$.
   Es gelten $k \leq n$ und $\sum_{i=1}^{k} i \cdot m_i = n$.
   Laufzeit: $\mathcal{O}(n+k) = \mathcal{O}(n)$
2. Sortiere jedes Teilfeld aufsteigend mit RADIX-SORT.
   Laufzeit: $\sum_{i=1}^{k} \mathcal{O}(k \cdot m_i) = \mathcal{O}\left(\sum_{i=1}^{k} k \cdot m_i\right) = \mathcal{O}(n)$.

Anmerkung: Falls wir auch negative Zahlen in der Eingabe zulassen wollen, können wir die Eingabe zunächst in $\mathcal{O}(n)$ Schritten in drei Teilfelder (negativ/0/nicht-negativ) aufteilen, jeweils einzeln den Algorithmus anwenden und danach die sortierten Ergebnisse wieder in $\mathcal{O}(n)$ Schritten zusammenfügen.

Praktische Aufgaben

In dieser Aufgabe geht es darum, Zeichenketten mit RADIX-SORT zu sortieren. Sie sollen einen verbesserten RADIX-SORT implementieren, der effizient Zeichenketten *unterschiedlicher Länge* sortiert. Wir stellen eine Implementation von RADIX-SORT auf *Ilias* zur Verfügung, auf welcher Ihre Verbesserung aufbauen soll.

1. Studieren Sie die Funktion `radixSort` in `RadixSort.java`. Was ist die Komplexität dieses Algorithmus gegeben die Anzahl Zeichenketten n und die Länge der längsten Zeichenkette d? **(1 Punkt)**

   Die Zeitkomplexität des Algorithmus ist:

Java-Code	Zeit
// for all positions from left to right	d
// init empty queues	27
//place each char in queue	$n \cdot 1 \cdot 1$
//traverse queues	$27 \cdot n$

   Total : $O(d \cdot (27 + n \cdot 1 \cdot 1 + 27 \cdot n)) = \mathcal{O}(d \cdot n)$

2. Entwicklen Sie einen verbesserten RADIX-SORT Algorithmus, dessen Laufzeit linear in der Summe aller Buchstaben in allen Zeichenketten ist. Die Idee ist, dass die Zeichenketten zuerst ihrer Länge nach sortiert werden. In jedem Schritt von RADIX-SORT sollen dann nur diese Zeichenketten sortiert werden, die genug lang sind. Das heisst, die Zeichenketten haben an der entsprechenden Position keinen "Leerbuchstaben". Zeigen Sie mit einem kleinen Beispiel (zehn Zeichenketten, maximale Länge fünf Buchstaben), dass Ihr Algorithmus korrekt sortiert. **(2 Punkte)**

```java
import java.util.ArrayList;
import java.util.LinkedList;

public class RadixSort {
    public static void radixSort(ArrayList<String> A, int d)
    {
        ArrayList<LinkedList<String» queues = new ArrayList<>();
        // 27 queues for 26 characters plus 'empty' character
        for (int i=0; i<27; ++i) {
            queues.add(new LinkedList<String>());
        }
```

```
12
13          // sort/split A by length (linear in number of strings):
14          ArrayList<ArrayList<String» A_by_len = new
             ↪  ArrayList<>(d+1);
15          ArrayList<Integer> num_shorter_than = new
             ↪  ArrayList<>(d+1);
16          for(int i=0; i < d + 1; i++) {
17              A_by_len.add(i, new ArrayList<>());
18          }
19          for(int i=0; i < A.size(); i++) {
20              int len = A.get(i).length();
21              A_by_len.get(len).add(A.get(i));
22          }
23          for(int len=0, n=0; len <= d; len++) {
24              var B = A_by_len.get(len);
25              num_shorter_than.add(len, n);
26              for (int i = 0; i < B.size(); ++i) {
27                  A.set(n + i, B.get(i));
28              }
29              n += B.size();
30          }
31
32
33          // for all positions from right to left
34          for(int j=d-1; j>=0; j-)
35          {
36              // initialize empty queues
37              for(int i=0; i<27; i++) queues.get(i).clear();
38
39              // place each character array in correct queue
40              int startidx = num_shorter_than.get(j+1);
41              for(int i=startidx; i<A.size(); i++)
42              {
43                  // characters 'a'-'z'
44                  queues.get(A.get(i).charAt(j)-'a'+1)
45                      .addLast(A.get(i));
46              }
47
48              // traverse queues
49              int n = 0;
50              for(int i=0; i<27; i++)
51              {
52                  while(queues.get(i).size() > 0)
53                  {
54                      A.set(startidx+n,
                         ↪  queues.get(i).removeFirst());
55                      n++;
56                  }
57              }
58          }
59      }
60  }
```

3. Vergleichen Sie Ihren verbesserten Algorithmus mit der ursprünglichen Version. Erstellen Sie

eine Grafik, welche Zeitmessungen für beide Algorithmen und verschiedene Parameter n und d zusammenfasst. Erläutern Sie, ob Ihre Messungen der Theorie entsprechen. **(1 Punkt)**