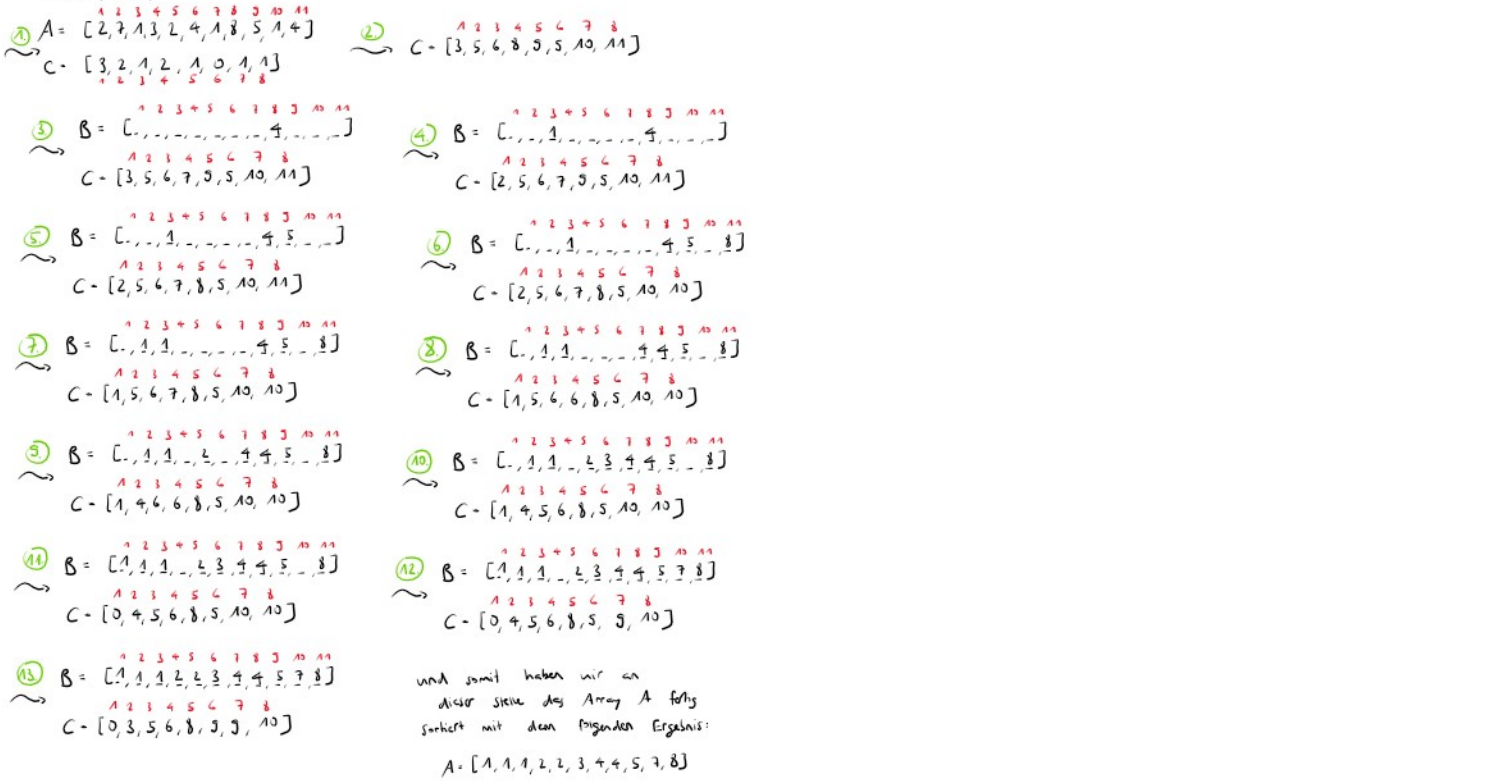
Serie 4

Sonntag, 20. März 2022 16:33

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)**



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)**

$A = [k_1, k_2, \ldots, k_n]$ $k_i \in \{0, 1, \ldots, k\}$

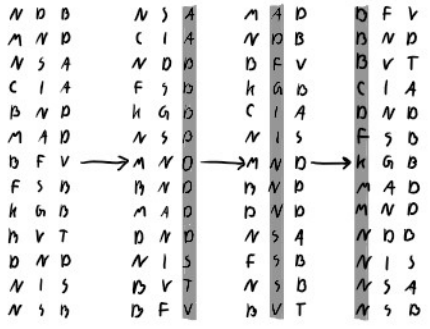
```
Preparation(A)  ← Analog zum Countingsort
C[k]
for i=0 to k        } kann vernachlässigt werden?
   C[i]=0
for i=1 to n        } Θ(n)
   C[A[i]]=(C[A[i]])+1
for i=1 to k        } Θ(k)   ⌐ Preparation ∈ Θ(n+k)
   C[i]=C[i]+C[i-1]
```

```
Numbers between (C, a, b)  ← wird nach Preparation durchgeführt

if a>0:
  n_a = C[a-1]  ← "werde <a"
else:
  n_a = 0
n_b = C[b]  ← "werde ≤b"

return n_b - n_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)**



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

   (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)

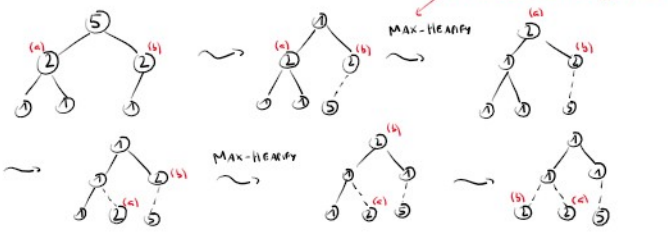
(a)

- Insertionsort ist ein stabiler Sortieralgorithmus. Wir sehen im Pseudocode aus der ersten Vorlesung, dass nur benachbarte Elemente vertauscht werden und nur, falls $A[i] > key$, d.h. ihre Schlüsselwerte müssen sich unterscheiden.
- Mergesort ist ebenfalls ein stabiler Sortieralgorithmus, wenn wir im Merge-Schritt (zwei sortierte Teilarrays verbinden) bei Gleichheit der Elemente immer diejenige in der linken Liste wählen. Dadurch wird sichergestellt, dass die Ordnung beibehalten wird, weil sich in der linken Liste die Elemente aus dem 1. Teilarray befinden.

Heapsort: Dieser Sortieralgorithmus ist nicht stabil. Dies liegt daran, dass die Reihenfolge der Elemente im ausgelegten Binärbaum nicht der Reihenfolge der Elemente im flachen Array entspricht. Das folgende Beispiel zeigt eine nicht-stabile Sortierung:

$A = [5, 2^{(a)}, 2^{(b)}, 1, 1, 1]$ (schon nach 1. MAX-HEAPIFY)

bei Gleichheit wird E.H. im linken Teilbaum hochgenommen



und somit haben wir im Ausgabearray: $A = [1, 1, 1, 2^{(b)}, 2^{(a)}, 5]$

⇒ nicht stabil!

Quicksort: Dieser Sortieralgorithmus ist nicht stabil, da in der Partition-Function die Ordnung von gleich grossen Schlüsselwerten gestört werden kann.

(b) Wir können in dem zu sortierenden Array A die Schlüssel $k[i]$ zum Wert $A[i]$ mit einem neuen Schlüssel $k'[i] = [k[i], i]$ behaften. Dieser neue Schlüssel wird nun der Reihenfolge von links nach rechts sortiert. Dadurch hat $k[i]$ immer noch die höhere Präzedenz. Jedoch wird durch ein 1. Sortieren der zweiten Komponente gewährt, dass gleiche Schlüssel nicht vertauscht werden. Es kommt nun zu $\Theta(n)$ mehr Speicheraufwand von den zusätzlichen Schlüsseln und auch $\Theta(n)$ mehr Zeitaufwand für das Generieren der Schlüssel.

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

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

	A	Q	B
1	.79	0	
2	.73	1	→ .13 → .16
3	.16	2	.20
4	.64	3	.39
5	.39	4	.42
6	.20	5	.53
7	.89	6	.64
8	.53	7	→ .71 → .79
9	.71	8	.89
10	.42	9	

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

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 $O(n)$ sortiert werden kann. (1 Punkt)

Wir haben somit ein Array mit n verschiedenen Werten, die alle in Base-n angegeben sind und nicht-negativ sind. Wir können nun zunächst das Array mit Counting-Sort nach der Grösse sortieren. Dies benötigt $\Theta(n)$. Anschliessend führen wir Radix-Sort über die Elt. durch, die genügend gross sind. Weil auch dieser Radix-Sort in $\Theta(n)$ ist, wird die Gesamtlaufzeit in $\Theta(n)$ sein.

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)**

Code:
27 · (generate empty queues) ← $\theta(1)$
for right to left ← $\theta(d)$
  initialize empty queues ← $\theta(1)$
  write character arrays to queues ← $\theta(n)$

⇒ Laufzeit $\in \theta(n \cdot d)$

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)**

Proof of work:

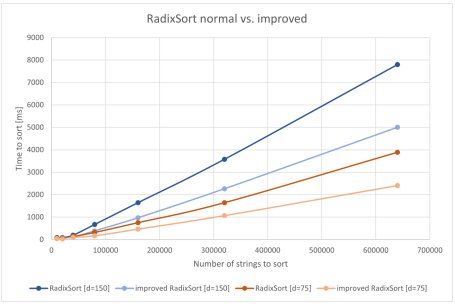
argma	al	a
ay	dtgeq	egxhb
jqq	iad	inkgh
ot	jeveh	leetp
r	jmhra	lhu
rzda	m	me
tikk	mbdte	om
wxcd	nis	p
z	ov	qc
z	p	r

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)**

Erwartung Laufzeit RadixSort: $\theta(n \cdot d)$
Erwartung Laufzeit improved RadixSort: $\theta(n \cdot d)$
Aber average case:
da nur "genug lange" strings sortiert werden und strings bei normalverteilung ca. $\frac{d}{2}$ lang im Mittel:

$$\text{Laufzeit} = \sum_{i=0}^{d} \frac{n \cdot i}{d} = n \cdot \sum_{i=0}^{d} \frac{i}{d} = n \cdot \frac{d(d+1)}{2d} = \frac{n \cdot (d+1)}{2} \geq \frac{n \cdot d}{2}$$

⇒ Laufzeit improved RadixSort $\in \theta(\frac{n \cdot d}{2})$



Beim vergleich fällt auf, dass beide $\in \theta(n \cdot d)$ wobei die "linearität in d" sehr schön repräsentiert wird, und die linearität in n erst für grosse n ≥ 70'000 ersichtlich wird.