Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
P. Bamberger
P. Schneider

Algorithmen und Datenstrukturen
Sommersemester 2020
Übungsblatt 3

Abgabe: Mittwoch, 03.06.2020, 16:00 Uhr.

Aufgabe 1: Bucket Sort *(7 Punkte)*

Bucketsort ist ein einfacher Algorithmus um ein Array $A[0..n-1]$ von n Datenelementen, deren zugehörige Sortierschlüssel nur Werte im Bereich $\{0,\dots,k\}$ annehmen, *stabil* zu sortieren. Dazu ordnet eine Funktion `key` jedem Datum $x \in A$ einen Schlüssel $\texttt{key}(x) \in \{0,\dots,k\}$ zu.

Zunächst wird ein Array $B[0..k]$ bestehend aus FIFO Queues erstellt, d.h., $B[i]$ stellt die i-te Queue dar. Dann iterieren wir durch das Array A. Wenn das aktuelle Element $A[j]$ den Schlüssel $i := \texttt{key}(A[j])$ hat reihen wir das Datum $A[j]$ in die Queue $B[i]$ mittels `enqueue` an.

Schließlich leeren wir alle Queues in $B[0],...,B[k]$ mittels `dequeue` und schreiben die Rückgabewerte *der Reihe nach* zurück in A. Danach ist A bezüglich `key` sortiert. Insbesondere bleiben Datenelemente $x, y \in A$ mit gleichem Schlüssel $\texttt{key}(x) = \texttt{key}(y)$ in gleicher relativer Reihenfolge.

Implementieren Sie *Bucketsort* auf Basis dieser Beschreibung. Sie *können* dazu die Vorlage `BucketSort.py` benutzen. Diese basiert auf einer Implementierung von FIFO Queues die wir Ihnen ebenfalls in den Dateien `Queue.py` und `ListElement.py` zur Verfügung stellen.[1]

Aufgabe 2: Radix Sort *(13 Punkte)*

Gegeben sei ein Array $A[0..n–1]$ der Größe n gefüllt mit ganzahligen Werten im Bereich $\{0,1,\dots,k\}$ für ein $k \in \mathbb{N}$. Unsere Implementierung von *Radixsort* benutzt den *stabilen* Sortieralgorithmus *Bucketsort* als Unterprogramm um A zu sortieren.

Das funktioniert wie folgt. Sei $m = \lfloor \log_b k \rfloor$. Jeder Schlüssel $x \in A$ wird als Zahl zur Basis b aufgefasst, d.h. $x = \sum_{i=0}^{m} c_i \cdot b^i$. Sie dürfen vereinfachend $b = 10$ wählen, dann entsprechen die c_i den Ziffern der Dezimalzahl x. Dann werden die Schlüssel in A zuerst nach ihrer niedrigst-wertigsten Stelle c_0 mittels `BucketSort` sortiert. Danach nochmals nach der Stelle c_1. Und so weiter bis die Schlüssel in A nach jeder Stelle c_i für $i = 0,\dots,m$ sortiert wurden.[2] Am Ende ist A sortiert.

(a) Implementieren Sie *Radixsort* auf Basis dieser Beschreibung. Dazu sollen Ihren Sortieralgorithmus *Bucketsort* als Unterprogramm benutzen. Falls Ihnen Aufgabe 1 nicht gelungen ist (und nur dann) dürfen Sie eine Bibliotheksfunktion (bspw. `sorted`) als Alternative benutzen. *(6 Punkte)*

(b) Vergleichen Sie die Laufzeit Ihrer Implementierungungen von *BucketSort* und *RadixSort*. Erstellen Sie dazu Arrays mit fester Größe $n = 10^4$ gefüllt mit *zufälligen* Schlüsseln im Bereich $\{0,1,\dots,k\}$. Tragen Sie die entsprechenden Laufzeiten für $k \in \{i \cdot 10^4 \mid i = 1,\dots,50\}$ (d.h. 50 Datenpunkte) in einem Schaubild auf. Diskutieren Sie Ihre Ergebnisse kurz in den `erfahrungen.txt`. *(4 Punkte)*

(c) Geben Sie die *asymptotische* Laufzeit Ihrer Implementierungen von *Bucketsort* und *Radixsort* abhängig von n und k an und begründen Sie diese. *(3 Punkte)*

[1] Vergessen Sie nicht sinnvolle Unit-Tests und angemessene Kommentierung ihres Quellcodes beizufügen.

[2] Hinweis: Die i-te Stelle c_i einer Zahl $x \in \mathbb{N}$ in b-adischer Darstellung $x = c_0 \cdot b^0 + c_1 \cdot b^1 + c_2 \cdot b^2 + \dots$ erhalten Sie mit der Formel $c_i = (x \texttt{ mod } b^{i+1}) \texttt{ div } b^i$, wobei `mod` die modulo-Operation und `div` die ganzzahlige Division ist.