Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2026
Musterlösung Übungsblatt 3

Abgabe: Dienstag, 12. Mai, 2026, 10:00 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 Element $x \in A$ einen Schlüssel $\texttt{key}(x) \in \{0,\dots,k\}$ zu.
Zunächst wird ein Array $B[0...k]$ bestehend aus FIFO Queues[1] 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 Element $A[j]$ 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 der Funktion `key` sortiert. Insbesondere bleiben Datenelemente $x, y \in A$ mit gleichem Schlüssel $\texttt{key}(x) = \texttt{key}(y)$ in gleicher relativer Reihenfolge (siehe *stabiles* sortieren).
Implementieren Sie *Bucketsort* auf Basis dieser Beschreibung (Vorlage: `BucketSort.py`). Diese basiert auf einer Implementierung von FIFO Queues die wir Ihnen ebenfalls in den Dateien `Queue.py` und `ListElement.py` zur Verfügung stellen.[2] Vergessen Sie nicht, Ihren Quellcode mit sinnvollen Unit-Tests zu versehen. Nutzen sie für die Tests eine möglichst einfache Funktion `key`(x), z.B. `key`(x) = x, damit die Tests übersichtlich bleiben.

Musterlösung

Siehe `BucketSort.py` im `public` Repository.

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*[3] 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. Anschließend werden die Schlüssel in A zuerst nach ihrer niedrig-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.[4] Am Ende ist A sortiert. Hierzu ein kleines **Beispiel**:

[1] Eine FIFO (*First In First Out*) Queue ist eine Datenstruktur, bei der die ersten eingefügten Elemente auch als erste entfernt werden. Details sind in den Vorlesungsfolien.

[2] Sie dürfen auch Bibiliotheksklassen nutzen, beachten Sie aber die abweichenden Methodenbenennungen.

[3] Ein stabiles Sortierverfahren ist ein Sortieralgorithmus, der die Reihenfolge der Daten (hier: Elemente im Array), deren Sortierschlüssel gleich sind, bewahrt. Siehe das Beispiel weiter unten.

[4] 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.

Angenommen $A = [329, 457, 657, 839, 436, 720, 355, 009]$ (und $b = 10$). Dann sortieren wir zuerst nach der niedrig-wertigsten Stelle c_0 (also die rechteste der Ziffern) und erhalten mit Bucketsort:

$$A = [720, 355, 436, 457, 657, 329, 839, 009]$$

Da Bucketsort stabil ist, bleiben die Elemente mit gleicher Ziffer in c_0 in der gleichen Reihenfolge, so muss zum Beispiel 839 vor 009 kommen, da 839 in der ursprünglichen Reihenfolge von A vor 009 war. Danach sortieren wir nach der Stelle c_1 und erhalten

$$A = [009, 720, 329, 436, 839, 355, 457, 657]$$

Letztlich sortieren wir nach der höchsten Stelle c_2 und erhalten das sortierte Array

$$A = [009, 329, 355, 436, 457, 657, 720, 839]$$

(a) Implementieren Sie *Radixsort* auf Basis dieser Beschreibung. Dazu sollen Ihren Sortieralgorithmus *Bucketsort* als Unterprogramm benutzen. *(7 Punkte)*

(b) Vergleichen Sie die Laufzeit Ihrer Implementierungungen von *BucketSort* und *RadixSort*. Die Eingabe sind Arrays der Größe $n = 10^4$ gefüllt mit *zufälligen* Schlüsseln im Bereich $\{0, 1, \ldots, k\}$. Tragen Sie die Laufzeiten für $k \in \{2i \cdot 10^4 \mid i = 1, \ldots, 60\}$ in einem Schaubild auf (Funktionen dafür sind in den Vorlagen). Diskutieren Sie Ihre Ergebnisse kurz in den `erfahrungen.md`. *(3 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)*

Musterlösung

(a) Siehe `RadixSort.py` im `public` Repository.

(b) Siehe Abbildung 1 für die entsprechenden Datenpunkte. Uns fällt auf das *Bucketsort* einen klar linearen Trend in k hat. Bei *Radixsort* scheint die Siutation nicht so klar, auf den ersten Blick ist die Laufzeit konstant. Auf den zweiten Blick fallen uns Stufen auf die auf $k = 10^5, 10^6$ fallen. Das liegt daran dass *Radixsort* für jede vorkommende Ziffer in der Eingabe einen Durchlauf von *Bucketsort* startet. Das ist auch der Grund warum Bucketsort für kleine k schneller ist, da k zuerst ein Vielfaches von n sein muss damit die Laufzeiten etwa gleich sind (grob muss k so sein dass $n \log_{10}(k) = n + k$).

(c) *Bucketsort* durchläuft das ganze Eingabearray A zwei mal, einmal um alle Werte aus A in die Buckets zu schreiben und einmal um die Werte aus den Buckets zurück nach A zu schreiben. Das hat eine Laufzeit von $\mathcal{O}(n)$ (denn das Eintragen *eines* Wertes in ein Bucket und das zurückschreiben *eines* Wertes am Anfang eines Buckets nach A dauert jeweils nur $\mathcal{O}(1)$). Zusätzlich muss *Bucketsort* noch k leere Listen erstellen und in ein Array der Größe k eintragen. Das benötigt $\mathcal{O}(k)$ Zeit. Insgesamt also $\mathcal{O}(n + k)$.

RadixSort startet einen Durchlauf von *Bucketsort* für jede Ziffer. Da unsere Schlüssel nur $m \in \mathcal{O}(\log k)$ Ziffern haben, starten wir $\mathcal{O}(\log k)$ Durchläufe von *BucketSort*. Ein Durchlauf von *Bucketsort* dauert jetzt außerdem nur $\mathcal{O}(n)$ da die Anzahl unterschiedlicher Schlüssel die *Bucketsort* berücksichtigt, konstant ist (es gibt nur 10 unterschiedliche Möglichkeiten für eine Ziffer). Insgesamt haben wir also $\mathcal{O}(n \log k)$.

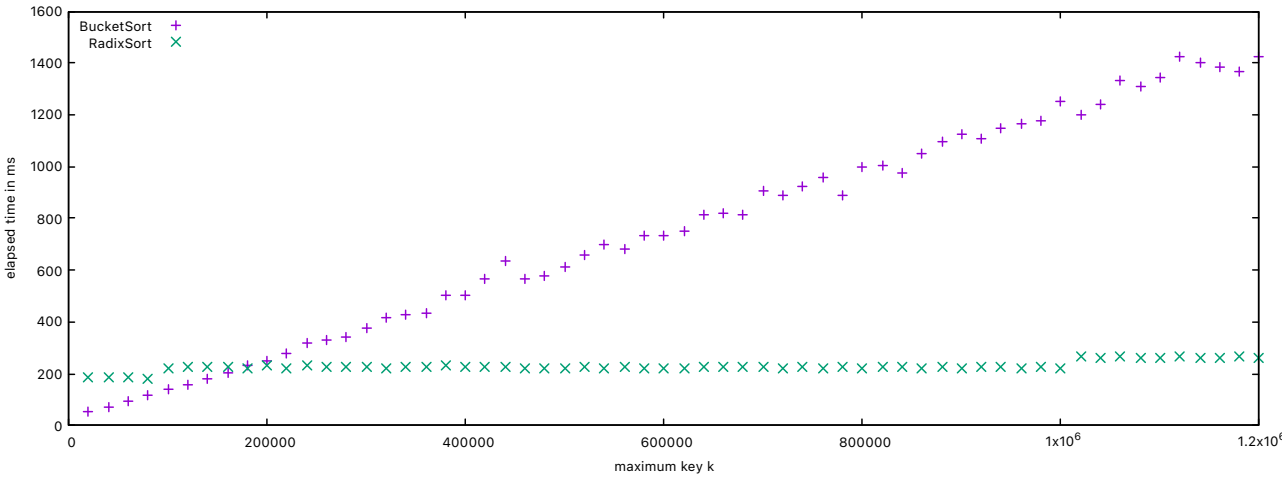

Abb. 1: Plot zu Aufgabe 2 (b).