technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Jonas Ellert
Mart Hagedoorn
Guangping Li

Sommersemester 2022
4. Mai

DAP2 Praktikum – Blatt 6

Abgabe: ab 16. Mai

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

Radixsort

In der Vorlesung haben Sie *Radixsort* kennengelernt. Die Vorlesungsmaterialien finden Sie im Moodle-Raum der Vorlesung. Natürlich können Sie auch ein gängiges Lehrbuch (z.B. “Introduction to Algorithms” von Cormen et al.) verwenden.

Radixsort wird oft damit motiviert, dass eine Liste von mehrdimensionalen Schlüsseln sortiert werden soll. Tatsächlich ist Radixsort aber auch gut dazu geeignet, eine Liste von Ganzzahlen zu sortieren. Dabei wird jeder `int`-Wert als ein Schlüssel mit vier Komponenten interpretiert, wobei jede Komponente eines der vier Bytes ist, welche den `int`-Wert bilden. Das höchstwertigste Byte (Most-Significant-Digit, MSD) ist die erste Komponente, und das niederwertigste Byte (Least-Significant-Digit, LSD) ist die letzte Komponente. Beispiel:

Wert in Basis-10-Darstellung:	1661914940
Wert in 32-Bit-Binärdarstellung:	01100011 00001110 11001111 00111100
Wert in Byte-Komponenten:	$\underset{\text{Komp. 1}}{99}$ $\underset{\text{Komp. 2}}{14}$ $\underset{\text{Komp. 3}}{207}$ $\underset{\text{Komp. 4}}{60}$

Jedes Byte entspricht dabei einer Stelle in der Basis-256-Darstellung des Wertes:

$$1661914940 = 99 \cdot 256^3 + 14 \cdot 256^2 + 207 \cdot 256^1 + 60 \cdot 256^0.$$

In dieser Aufgabe sollen Sie zwei Varianten von Radixsort für Ganzzahlen implementieren: Least-Significant-Digit-First (LSD) und Most-Significant-Digit-First (MSD). Für beide Varianten ist es wichtig, dass Sie ein Array nach dem Wert eines bestimmten Bytes sortieren können.

Langaufgabe 6.1: Sortieren nach einem Byte (3 Punkte)

Implementieren Sie die Methode

`public static void sortByByte(int[] input, int l, int r, int b)`.

Diese bekommt ein Array `input` übergeben, und soll das Arrayintervall `input[l, r]` *absteigend und stabil* nach dem `b`-niederwertigsten Byte sortieren (für `b` $= 0$ nach dem niederwertigsten Byte, für $b = 3$ nach dem höchstwertigsten). Dabei sollen Sie den Mechanismus von Counting-Sort verwenden (siehe Blatt 5), und ein Frequenzarray mit 256 Einträgen benutzen. Sie dürfen ein Hilfsarray der Länge `r`$-$`l`$+1$ verwenden, und davon ausgehen, dass nur positive Ganzzahlen eingegeben werden.

Tipp: Wie Sie oben gesehen haben, ist das `b`-niederwertigste Byte eines Integers `a` der Wert der `b`-niederwertigsten Stelle in Basis 256. Mathematisch entspricht dies $(\texttt{a}/256^{\texttt{b}}) \bmod 256$. Praktisch berechnen Sie die Division durch einen Bitshift, und den Modulo durch die Verundung mit einer Maske: `(a >> (8 * b)) & 0xFF`.

Langaufgabe 6.2: LSD-Radixsort (1 Punkt)

Implementieren Sie die Methode

`public void lsdRadix(int[] data)`,

welche die Ganzzahlen in `data` absteigend sortiert. Diese Methode sortiert die Zahlen zuerst nach dem niederwertigsten Byte mittels `sortByByte` (Aufgabe 6.1). Dann sortiert der Prozess die Zahlen nach dem zweit-niederwertisten Byte, dem dritt-niederwertigsten Byte, und schließlich nach dem höchstwertigsten (viert-niederwertigsten) Byte.

Abschließend sollen Sie eine geeignete `main`-Methode schreiben, sodass Ihr Porgramm eine Liste von Ganzzahlen via Standard-In erhält, und dann die Liste *absteigend* sortiert ausgibt. Wie immer sollten Sie bei ungültigen Eingaben passende Fehlermeldungen ausgeben! Sie dürfen davon ausgehen, dass nur positive Ganzzahlen eingegeben werden.

Überprüfen Sie die Korrektheit des Ergebnis mit einer geeigneten Assertion!

Langaufgabe 6.3: MSD-Radixsort (3 Punkte)

In der letzten Aufgabe haben Sie Radixsort nach dem Konzept Least-Significant-Digit-First implementiert. Dieses Konzept geht gewissermaßen gegen die menschliche Intuition: Es scheint natürlicher, die Zahlen ersteinmal nach der höchstwertigsten Stelle zu gruppieren.

Dies sollen Sie nun in der Implementierung von MSD-Radixsort umsetzen. Der Algorithmus sortiert die Zahlen zunächst nach dem höchstwertigsten Byte. Dadurch wird das Array in Intervalle partitioniert, von denen jedes Intervall genau die Werte enthält, die das gleiche höchstwertigste Byte haben. Nun wird jedes Intervall rekursiv nach dem nächsten Byte sortiert, und rekursiv fortgefahren, bis entweder nach allen vier Bytes sortiert wurde, oder das betrachtete Intervall klein genug ist, um es mit einem naiven Algorithmus zu sortieren.

Implementieren Sie die Methode

`public void msdRadix(int[] data, int l, int r, int b)`,

welche wie folgt funktioniert:

- Wenn die Methode aufgerufen wird, dann ist `data[l, r]` bereits nach den höchstwertigsten $(4-(\texttt{b}+1))$ Bytes sortiert. Der initiale Aufruf erfolgt mit $\texttt{l} = 0$, $\texttt{r} = \texttt{data.length} - 1$ und $\texttt{b} = 3$.
- Wenn das Intervall bereits sehr kurz ist, genauer gesagt $\texttt{r} - \texttt{l} + 1 \leq 32$, dann sortieren Sie das Intervall `data[l, r]` absteigend mit dem naiven Algorithmus Insertion-Sort[1].
- Ansonsten sortieren Sie `data[l, r]` absteigend nach dem `b`-niederwertigsten Byte ($\texttt{b} = 0$ entspricht dem niederwertigsten Byte). Dazu können Sie die Methode `sortByByte` aus Aufgabe 6.1 verwenden. Damit Sie rekursiv fortfahren können, müssen Sie die Grenzen der Subintervalle mit gleichem `b`-niederwertigsten Byte bestimmen. Dazu können Sie ein Array C der Länge 257 berechnen, sodass $\texttt{data}[C[i], C[i+1]-1]$ genau die Werte enthält, deren `b`-tes Byte den Wert i hat.

  Nun sortieren Sie jedes Subintervall $\texttt{data}[C[i], C[i+1]-1]$ rekursiv weiter.

  Tipp: Das Array C lässt sich im Aufruf von `sortByByte` leicht aus dem Frequenzarray berechnen. Sie können `sortByByte` modifizieren, sodass C zurückgegeben wird.

  Würden Sie statt Bytes die Dezimalstellen der Zahlen verwenden, so sähe ein Sortierschritt etwa so aus:

  `[123, 134, 234, 345, 222, 225]-> [123, 134, 234, 222, 225, 345]`

  (`123, 134`: rekursiv sortieren; `234, 222, 225`: rekursiv sortieren; `345`: rekursiv sortieren)

Schreiben Sie außerdem eine Methode `public void msdRadix(int[] data)` für den initialen Aufruf, und eine `main`-Mehthode mit der üblichen Funktionalität. Sie dürfen davon ausgehen, dass nur positive Ganzzahlen eingegeben werden.

Wie immer sollten Sie bei ungültigen Eingaben passende Fehlermeldungen ausgeben!

Überprüfen Sie die Korrektheit des Ergebnis mit einer geeigneten Assertion!

[1] siehe z.B. `https://www.geeksforgeeks.org/insertion-sort/`

Langaufgabe 6.4: Laufzeitmessung (1 Punkt)

Messen Sie die Laufzeit der zwei Varianten von Radixsort auf zufälligen Sequenzen. Welcher Algorithmus ist schneller? Vergrößern Sie oder verändern Sie die Struktur der Eingabe, bis Sie einen klaren Unterschied feststellen können. Achten Sie bei der Generierung der Eingabe darauf, welche Bytes wirklich von den Eingabewerten benutzt werden!

Weitere Hinweise zur Laufzeitmessung finden Sie auf Blatt 2, Aufgabe 2.4.

Achten Sie darauf, dass Sie nur die Zeit für die Sortierung messen, also nicht etwa die Zeit zum Lesen der Eingabe oder zum Prüfen von Assertions!