Technische Universität Dortmund
Fakultät Statistik
Lehrstuhl Computergestützte Statistik

Dr. Uwe Ligges
M.Sc. Daniel Horn
M.Sc. Hendrik van der Wurp
Steffen Maletz

Übung zur Vorlesung

Computergestützte Statistik

Wintersemester 2018/2019

Übungsblatt Nr. 2

Abgabe ist Montag der 22.10.2018 an CS-abgabe@statistik.tu-dortmund.de oder Briefkasten 138

Aufgabe 1 **(4 Punkte)**

Implementieren und Testen Sie den aus der Vorlesung bekannten Sortieralgorithmus Bubble-Sort.

a) (1 Punkt) Schreiben Sie einen `R`-Funktion `bubbleSort`. Diese soll als Eingabe einen numerischen Vektor `a` erhalten, Ausgabe ist die sortierte Version von `a`.

b) (0.5 Punkte) Dokumentieren Sie Ihre Funktion

c) (0.5 Punkte) Testen Sie Ihre Funktion. Hierbei empfiehlt sich der Einsatz der `R`-Funktion `is.unsorted`.

d) (1 Punkt) Erweiteren Sie Ihre Funktion um den logischen Eingabeparameter `decreasing`. Dieser soll steuern, ob `a` aufsteigend oder absteigend sortiert wird. Passen Sie Ihre Dokumentation und Ihre Tests entsprechend an.

e) (1 Punkt) Erweitern Sie Ihre Funktion, so dass diese die Anzahl der durchgeführten Vergleiche zählt. Rückgabe Ihrer Funktion soll nun eine Liste mit 2 Elementen sein, dem sortierten Vektor und der Anzahl vergleiche. Passen Sie Ihre Dokumentation und Ihre Tests entsprechend an.

Hinweis: Bitte geben Sie nur die Implementierung Ihre finalen Funktion aus Aufgabenteil e) ab.

Aufgabe 2 **(4 Punkte)**

Es existieren weitaus mehr Sortieralgorithmen als in der Vorlesung vorgestellt werden. Einer davon ist Bucket-Sort. Bucket-Sort sortiert zunächst die Beobachtungen in eine Anzahl Töpfe (engl., *buckets*) vor und sortiert anschließend jeden Topf mit einem einfachen Sortieralgorithmus, wie z.B. mit Bubble-Sort.

Require: $\mathbf{a} \in \mathbb{R}^n$ (list of inputs), $\texttt{n.bucket} \in \mathbb{N}$

for $i \in 1, ..., \text{length}(\mathbf{a})$ **do**

$\quad \texttt{bucket}[i] \leftarrow \left\lceil \texttt{n.bucket} \cdot \frac{\mathbf{a}[i] - \min(\mathbf{a})}{\max(\mathbf{a}) - \min(\mathbf{a})} \right\rceil$

end for

for $j \in 1, ..., \texttt{n.bucket}$ **do**

$\quad \texttt{res} \leftarrow \text{c}(\texttt{res}, \text{sort}(\mathbf{a}[\texttt{bucket} == j]))$

end for

return `res`

---

a) (1 Punkt) Geben Sie für jede Zeile des Pseudo-Codes an, wie viele Vergleiche durchgeführt werden. Geben Sie ein geeignetes Landau-Symbol an, dass die Gesamtzahl Vergleiche des Algorithmus beschreibt.

b) (2 Punkte) Implementieren Sie Bucket-Sort in einer Funktion `bucketSort`. Eingabe soll neben dem zu sortierenden Vektor die Anzahl Töpfe `n.bucket` sein. Verwenden Sie in Zeile 6Ihre Implementierung von Bubble-Sort aus Aufgabe 1. Achten Sie auf eine ordentliche Dokumentation. Testen Sie Ihre Implementierung mit Ihrer Testfunktion aus Aufgabe 1 c). Auch Ihre Bucket-Sort Implementierung soll zählen, wie viele Vergleiche durchgeführt werden.

c) (1 Punkt) Implementieren Sie eine kleine Simulation, um Ihr theoretisches Ergebnis aus c) zu untermauern. Erzeugen Sie dazu dazu 100 Vektoren der Länge 1000 und sortieren diese mit Bucket-Sort für `n.bucket` $\in \{1, ..., 100\}$. Stellen Sie die Ergebnisse geeignet grafisch dar. Inwiefern arbeitet Bucket-Sort *unfair*?

Hinweise: Sollten Sie an der Implementierung von Bubble-Sort gescheitert sein, dürfen Sie auch die `sort`-Funktion aus `R` verwenden. In diesem Fall ist die Bearbeitung von Aufgabenteil c) leider nicht möglich.

Aufgabe 3 **(4 Punkte)**

In dieser Aufgabe soll der Sortieralgorithmus *Merge-Sort* analysiert werden. Dieser verfolgt das verbreitete *divide et impera*-Prinzip (*divide-and-conquer*). Dieses beruht darauf das gesamte Problem zunächst in einfache Teilprobleme zu unterteilen, diese zu lösen und aus den Teilproblemen die Lösung des gesamten Problems abzuleiten.

Die Idee von *Merge-Sort* ist es, den Eingabevektor zunächst zu halbieren und jeden der beiden Teilvektoren erneut (rekursiv) mit *Merge-Sort* zu sortieren. Aus den beiden sortierten Teilvektoren kann nun mit der Hilfsfunktion Merge in $b{\cdot}n$ Vertauschungen das sortierte Endergebnis erzeugt werden.

Die genaue Realisierung dieser Hilfsfunktion Merge ist für uns nicht von Interesse. Außerdem wird verwendet, dass Vektoren der Länge 1 bereits sortiert sind.

Require: $\mathbf{a} \in \mathbb{R}^n$ (list of inputs), where $n = 2^k$ with $k \in \mathbb{N}$
Ensure: sorted $\vec{a}$
if length($\mathbf{a}$) $= 1$ **then**
  return $\mathbf{a}$
end if
$\mathbf{a}_1 \leftarrow \text{Recall}\left(\mathbf{a}_{1:\frac{n}{2}}\right)$
$\mathbf{a}_2 \leftarrow \text{Recall}\left(\mathbf{a}_{(\frac{n}{2}+1):n}\right)$
return Merge $(\mathbf{a}_1, \mathbf{a}_2)$

---

a) (3 Punkte) Zeigen Sie formal, dass für die Average-Case Laufzeit von MergeSort gilt: $C_{mean}(n) \in O(n \log_2 n)$. Nehmen Sie hier der Einfachheit halber an, dass die Länge n des Eingabevektors stets eine Zweierpotenz ist, d. h. $\exists k \in \mathbb{N} : n = 2^k$.

b) (1 Punkt) Begründen Sie, warum die Laufzeitanalyse aus a) für alle n gilt, obwohl wir für den Beweis die Einschränkung $\exists k \in \mathbb{N} : n = 2^k$ getroffen haben. Hier ist kein formaler Beweis vonnöten, sondern lediglich eine semi-formale Begründung.

Tipp: Der Beweis in a) lässt sich durch Induktion führen. Überlegen Sie sich dazu zunächst wie $C_{mean}(n)$ mit den oben genannten Annahmen aussieht.