Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

5. Übungsblatt zu Algorithmen I im SoSe 2017

`http://crypto.iti.kit.edu/index.php?id=799`
`{bjoern.kaidel,sascha.witt}@kit.edu`

Musterlösungen

Aufgabe 1 (*Sortieren,* $1+2+2=5$ *Punkte*)

Sortieren Sie die Ziffern Ihrer Matrikelnummer mittels den in den Teilaufgaben angegebenen Algorithmen aus der Vorlesung. (Falls Sie zu zweit abgeben, genügt eine von beiden.) Nehmen Sie dazu an, Ihre Matrikelnummer ist ein Array mit Feldern, die je eine einstellige Zahl enthalten (die Matrikelnummer 1229203 würde also dem Array $[1, 2, 2, 9, 2, 0, 3]$ entsprechen).

a) Geben Sie Ihre Matrikelnummer an und bestimmen Sie die Anzahl von Inversionen in Ihrer Matrikelnummer. Eine Inversion ist ein Paar von Indizes (i, j), sodass $i < j$ und $A[i] > A[j]$.

b) Benutzen Sie zur Sortierung Ihrer Matrikelnummer Insertionsort. Geben Sie den Zustand des Arrays nach jedem Einfüge-Schritt an.

c) Benutzen Sie Mergesort. Verwenden Sie das Schema aus der Vorlesung (siehe Vorlesungsfolien vom 22.05.2017, Beispiel Folie 6).

Musterlösung:
Beispiel mit Matrikelnummer 1229203.

a) Diese Matrikelnummer hat 7 Inversionen.

b)

1	2,2,9,2,0,3
1,2	2,9,2,0,3
1,2,2	9,2,0,3
1,2,2,9	2,0,3
1,2,2,2,9	0,3
0,1,2,2,2,9	3
0,1,2,2,2,3,9	

c) $\langle 1, 2, 2, 9, 2, 0, 3\rangle$
split
$\langle 1, 2, 2\rangle$ $\langle 9, 2, 0, 3\rangle$
split
$\langle 1\rangle$ $\langle 2, 2\rangle$ $\langle 9, 2\rangle$ $\langle 0, 3\rangle$
split
$\langle 1\rangle$ $\langle 2\rangle$ $\langle 2\rangle$ $\langle 9\rangle$ $\langle 2\rangle$ $\langle 0\rangle$ $\langle 3\rangle$
merge
$\langle 1\rangle$ $\langle 2, 2\rangle$ $\langle 2, 9\rangle$ $\langle 0, 3\rangle$
merge
$\langle 1, 2, 2\rangle$ $\langle 0, 2, 3, 9\rangle$
merge
$\langle 0, 1, 2, 2, 2, 3, 9\rangle$

Aufgabe 2 (*Listen von Studenten, $4 + 3 = 7$ Punkte*)

Angenommen, Sie haben zwei doppelt verkettete Listen von Studierenden, eine aus WebInscribe mit n Tupeln der Form $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}, \mathit{tutorium})$, die andere aus dem Praktomat mit m Tupeln der Form $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer})$ (also ohne $\mathit{tutorium}$). Beide Listen sollten eigentlich die gleichen Daten enthalten. Allerdings haben sich einige Studierende *nur* bei WebInscribe angemeldet und andere *nur* beim Praktomaten. In beiden Listen gibt es jeweils keine Duplikate.

a) Entwerfen Sie einen Algorithmus, der folgende drei Listen berechnet:

- die Liste B aus Tupeln $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}, \mathit{tutorium})$ der Studierenden, die sich in WebInscribe und im Praktomat mit den *gleichen Daten* registriert haben,
- die Liste W aus Tupeln $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}, \mathit{tutorium})$ der Studierenden, die in WebInscribe, aber nicht im Praktomat eingetragen sind, und
- die Liste P aus Tupeln $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer})$ der Studierenden, die sich im Praktomat, aber nicht im WebInscribe angemeldet haben.

Da die Listen sehr lang sind, soll der Algorithmus in erwarteter linearer Laufzeit (also erwartet $\mathcal{O}(n + m)$) arbeiten.

Es ist kein Pseudocode zum Lösen der Aufgabe nötig, es genügt, wenn Sie ihr Vorgehen eindeutig beschreiben.

b) Begründen Sie detailliert die Laufzeit der Schritte in Ihrem Algorithmus.

Musterlösung:

a) Man nehme eine Hashtabelle H mit verketteten Listen der Größe $\Theta(|\text{WebInscribe-Liste}|)$ und einer zufälligen Hashfunktion h für Zeichenketten aus einer universellen Familie.

Wir durchlaufen die Liste aus WebInscribe und fügen die *vollständigen* Tupel in H in der Zelle mit dem Hashwert $h((\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}))$ ein, wobei h einen guten Hashwert berechnen muss. Wir fügen also das Tupel $(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}, \mathit{tutorium})$ an die Stelle $h((\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer}))$ ein. Somit fungieren die ersten drei Einträge des Tupels quasi als Schlüssel/Key. Diese Zuordnung ist eindeutig, da die Matrikelnummer eindeutig ist.

Dann durchlaufen wir die Liste aus dem Praktomat, und durchsuchen für jedes enthaltene Tupel die verkettete Liste in H an der Zelle $h(\mathit{vorname}, \mathit{nachname}, \mathit{matrikelnummer})$ nach einem in den ersten drei Komponenten exakt passenden Tupel. Finden wir ein passendes Tupel, können wir den vollständigen Eintrag samt *tutorium* aus H in die Liste B eintragen, und den Eintrag aus der Hashtabelle entfernen, indem wir ihn aus der Liste entfernen. Finden wir kein passendes Tupel wird der Eintrag aus Praktomat in P eingetragen.

Nach Durchlaufen der Praktomat-Liste gehen wir die Hashtabelle nochmals durch und sammeln alle übrig gebliebenen Einträge in der Liste W.

b) Sei n die Anzahl der Einträge in der WebInscribe-Liste und m die Anzahl in der Praktomat-Liste.

Das Initialisieren der Hashtabelle kostet $\mathcal{O}(n)$ Zeit und Platz. Durchlaufen der Liste WebInscribe kostet lineare Arbeit in n. Jede Hashtabelle-Einfüge Operation dauert nur erwartet $\mathcal{O}(1)$ Laufzeit, da die Hashtabelle $\Omega(n)$ Einträge hat. Der Aufbau der Hashtabelle mit Einträgen aus WebInscribe hat also insgesamt erwartet lineare Laufzeit in n.

Durchlaufen der Liste des Praktomat kostet lineare Arbeit in m, jeder *lookup* in der Hashtabelle dauert wieder nur erwartet $\mathcal{O}(1)$ Zeit, Vergleich der Paare ist konstante Arbeit, eventuelle Ausgabe ebenfalls. Löschen aus der Hashtabelle ist auch konstant viel Arbeit, da nur aus der doppelt verketten Liste entfernt werden muss.

Sammeln der übrigen Tupeln in W braucht Zeit linear in n.

Insgesamt haben wir also ein Laufzeit erwartet in $\mathcal{O}(n+m)$.

Aufgabe 3 ($\textit{Sortieren, } 4+2=6 \textit{ Punkte}$)

Betrachten Sie folgenden Sortieralgorithmus.

Procedure sort(A : Array $[1,\ldots,n]$ of $\mathbb{N}$, $links : \mathbb{N}$, $rechts : \mathbb{N}$)
  assert $1 \leq links \leq rechts \leq n$
  if $A[links] > A[rechts]$ **then**
    temp := $A[links]$ — vertausche $A[links]$ und $A[rechts]$
    $A[links] := A[rechts]$
    $A[rechts] := temp$
  if $links + 1 < rechts$ **then**
    $k := \lfloor (rechts - links + 1)/3 \rfloor$
    sort$(A, links, rechts - k)$ — sortiere die ersten zwei Drittel
    sort$(A, links + k, rechts)$ — sortiere die letzten zwei Drittel
    sort$(A, links, rechts - k)$ — sortiere die ersten zwei Drittel nochmal

a) Beweisen Sie mit vollständiger Induktion, dass $sort(A, 1, \text{length}(A))$ das Array $A[1 \ldots \text{length}(A)]$ korrekt sortiert. Sie dürfen dabei vereinfachend annehmen, dass Zahlen nicht doppelt vorkommen.

b) Geben Sie eine scharfe asymptotische Schranke im Θ-Kalkül für die Laufzeit des Algorithmus an. Beweisen Sie, dass die gegebene asymptotische Schranke tatsächlich eine obere Schranke der Laufzeit ist. Sie dürfen für die Laufzeitanalyse davon ausgehen, dass length(A) eine Dreierpotenz ist. Ist der Algorithmus asymptotisch schneller oder langsamer als Mergesort? Begründen Sie ihre Antwort!

Musterlösung:

a) Der Algorithmus funktioniert folgendermaßen: Zunächst werden (rekursiv) die ersten zwei Drittel des Arrays sortiert, dann die hinteren zwei Drittel und danach nochmals die ersten zwei Drittel des Arrays. Wir beweisen nun induktiv die Korrektheit. Genauer sei A ein Array, $links \geq 1$, $rechts \leq \text{length}(A)$ und $l := rechts - links + 1$. Dann liegt das Teilarray $A[links \ldots rechts]$ nach Aufruf von sort$(A, links, rechts)$ sortiert vor. Denn:

- **Induktionsanfang:** Für $l = 1$ und $l = 2$ liefert der Algorithmus das sortierte Teilarray: Der erste Fall ist „automatisch“ sortiert (da das Array nur ein Element enthält) und im zweiten Fall sorgt das mögliche Vertauschen (falls nötig) in der 3. Zeile für ein sortiertes Teilarray.
- **Induktionsvoraussetzung:** Wir nehmen an, dass für ein $l \in \mathbb{N}$ der *sort* Algorithmus für Intervalle bis zur Größe l korrekt sortierte Teilarrays liefert.
- **Induktionsschritt $l \rightsquigarrow l+1$:** Wir teilen nun zunächst die Elemente des Arrays in Typen. Die $l/3$ kleinsten Elemente ordnen wir dem Typ T_1 zu, die nächstgrößeren $l/3$ Elemente dem Typ T_2, entsprechend das nächstgrößere Drittel der Elemente dem Typ T_3. Da wir davon ausgehen, dass keine Zahl mehrmals vorkommt, ist diese Zuordnung eindeutig.

  Nach dem Aufruf sort$(A, links, rechts - k)$ kommen im *ersten* Drittel von $A[links \ldots rechts]$ keine T_3-Elemente mehr vor, da dass Array nach Induktionsvoraussetzung korrekt sortiert wird (da wir ein „kleineres“ Array übergeben greift die Induktionsvorraussetzung!). Die T_2 und T_3 Elemente aus dem ersten Drittel liegen nun also alle „in der Mitte“ des Arrays. Entsprechend kommen nach dem Aufruf von sort$(A, links + k, rechts)$ im letzten Drittel von $A[links \ldots rechts]$ keine T_1-Elemente mehr vor und alle T_3-Element befinden sich in sortierter

Reihenfolge im letzten Drittel. Im „mittleren Drittel“ liegen nun also noch unsortierte T_1 und T_2 Elemente. Wir sortieren also nochmal die ersten zwei Drittel des Arrays. Nach dem erneuten Aufruf von $\mathrm{sort}(A, links, rechts - k)$ stehen nach Induktionsvoraussetzung nun also auch alle T_1- und T_2-Elemente in sortierter Reihenfolge im Array, da dieser Aufruf wieder die „forderen“ zwei Drittel des Arrays nach Induktionsvorrausetzung richtig sortiert.

b) Für die Laufzeit ergibt sich für $n = \mathrm{length}(A) \geq 3$ die Rekurrenzgleichung

$$T(n) = \begin{cases} 1 & \text{für } n < 3, \\ 3 \cdot T\left(\frac{2n}{3}\right) + a_2 & \text{für } n \geq 3 \end{cases}$$

Nach Master-Theorem (mit $d = 3$ und $b = 3/2$) folgt, dass die Laufzeit des Algorithmus in $\Theta\left(n^{\log_{3/2}(3)}\right) = \Theta(n^{2.71\cdots})$. Damit ist der Algorithmus asymptotisch langsamer als Mergesort, dessen Laufzeit in $\mathcal{O}(n \log n)$ ist.