ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger, Dr. Johannes Lengler
Übungsleitung: Florian Meier

HS 2018

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 4

Lösungsvorschlag zu Aufgabe 1

a) In einem ersten Schritt verwenden wir die normale Binäre Suche (Algorithmus 2.1 im Skript) um zu testen, ob das gegebene (sortierte) Array a überhaupt x als Element enthält. Falls dies nicht der Fall ist, sind wir offensichtlich fertig. Ansonsten ist die Zahl x im Array a mindestens einmal enthalten, und wir müssen die genaue Anzahl herausfinden. In diesem Fall gibt es $1 \leq \ell \leq r \leq n$ so dass $a[i] = x$ für alle $\ell \leq i \leq r$. Betrachten wir die Implementation der Binären Suche, so stellen wir fest, dass in diesem Fall der Index ℓ ausgegeben wird. Das heisst wir haben bereits die linke Grenze gefunden.

Um die rechte Grenze zu finden, führen wir erneut die Binäre Suche aus, dabei suchen wir jetzt aber nach dem ersten Element, das *grösser* als x ist. Dies lässt sich leicht dadurch erreichen, indem der Vergleich $x > a[m]$ durch den Vergleich $x \geq a[m]$ ersetzt wird. Damit finden wir entweder den Index $r+1$ des ersten Elementes, das grösser als x ist, und subtrahieren von diesem Index 1 um r zu erhalten, oder wir sehen dass $a[n] = x$ und setzen $r = n$. In beiden Fällen ist nun klar, wie oft x im Array enthalten ist, nämlich $r - \ell + 1$ mal.

Alternativ kann man r auch mit folgender Modifkation der Binären Suche finden. Wir ersetzen den Teil, in dem die BINÄRE SUCHE rekursiv aufgerufen wird durch:

$m \leftarrow \ell + \lceil \frac{r-\ell}{2} \rceil$
if $x \geq a[m]$ **then**
  return BINÄRESUCHE(a, m, r, x)
else
  return BINÄRESUCHE$(a, \ell, m-1, x)$
end if

Im Wesentlichen haben wir zweimal Binäre Suche durchgeführt, demzufolge ist die Gesamtlaufzeit $\mathcal{O}(2\log(n)) = \mathcal{O}(\log n)$.

b) Wir nehmen erst an, dass n ungerade ist. Da die Arrays a und b bereits sortiert vorliegen, können wir ihre Mediane $m_a = a[\frac{n+1}{2}]$ und $m_b = b[\frac{n+1}{2}]$ sofort angeben, d.h. in Zeit $\mathcal{O}(1)$. O.B.d.A. sei $m_a \leq m_b$. Alle Elemente in a, die kleiner als m_a sind, kommen als gemeinsamer Median sicher nicht in Frage, da es bereits mehr als n Elemente gibt, die größer sind (nämlich die $(n-1)/2$ Elemente in a, die größer sind als m_a, die $(n-1)/2$ Elemente in b, die größer sind als m_b, sowie die beiden Mediane m_a und m_b selbst). Analog findet man, dass alle Elemente in b, die größer sind als m_b, für den gemeinsamen Median nicht in Frage kommen.

Wir haben also mit einem Vergleich ("$m_a \leq m_b$") im wesentlichen die Hälfte der $2n$ Zahlen als Kandidaten für den Median ausgeschlossen. Die verbleibenden $n+1$ Zahlen haben wir in zwei sortierten Arrays $a[\frac{n+1}{2}..n]$ und $b[1..\frac{n+1}{2}]$ vorliegen, für die wir den Algorithmus rekursiv aufrufen können. Dies, weil der gemeinsame Median dieser zwei Arrays gleich dem gemeinsamen Median des ursprünglichen Problems ist: wir haben gleich viele Elemente entfernt, die grösser als der gesuchte Median sind, wie solche, die kleiner sind.

Wir können das Vorgehen also iterieren, bis das verbleibende Problem konstante Grösse, (z.B. $n \leq 2$) hat, wofür wir den gemeinsamen Median dann in Zeit $\mathcal{O}(1)$ angeben können. Da sich die Anzahl der Zahlen jedesmal im wesentlichen halbiert, brauchen wir $\mathcal{O}(\log n)$ Iterationen.

Da in jeder Iteration nur ein Vergleich angestellt wird, ist die Gesamtlaufzeit unseres Algorithmus damit $\mathcal{O}(\log n)$.

Algorithm 1 MEDIAN(sortierte Arrays $a[1..n], b[1..n]$)

```
if n ≤ 2 then
  return MEDIANOFFOUR(a[1..n], b[1..n])
end if
if a[⌊(n+1)/2⌋] ≤ b[⌈(n+1)/2⌉] then
  MEDIAN(a[⌊(n+1)/2⌋..n], b[1..⌈(n+1)/2⌉])
else
  MEDIAN(a[1..⌊(n+1)/2⌋], b[⌈(n+1)/2⌉..n])
end if
```

In der Pseudocode-Implementation MEDIAN dieses Algorithmus wird mittels Gauss-Klammern berücksichtigt, dass n i.A. nicht ungerade ist. Dabei wird einmal auf- und einmal abgerundet, damit die beiden kleineren Arrays gleiche Länge haben.

Lösungsvorschlag zu Aufgabe 2

(a) Die aufsteigend sortierte Liste $(1, 2, \ldots, n)$ hat keine Inversionen. Da die Anzahl Inversionen nicht negativ sein kann, ist das minimal.

In der absteigend sortierten Liste $(n, n-1, \ldots, 1)$ ist jedes Paar (a_i, a_j) mit $i \neq j$ eine Inversion und es gibt somit $\frac{n\cdot(n-1)}{2}$ Inversionen. Da es nicht mehr Inversionen als solche Paare geben kann, ist das die maximal mögliche Anzahl.

(b) Der Algorithmus verwendet einen Zähler c, der anfangs 0 ist. Dann geht er alle $\frac{n\cdot(n-1)}{2}$ Paare (a_i, a_j) mit $i \neq j$ durch und prüft für jedes ob es eine Inversion ist. Falls ja, setzt er $c := c + 1$ und sonst ändert er c nicht. Am Ende wird c ausgegeben. Die Korrektheit ist offensichtlich und die Laufzeit ist $O(\frac{n\cdot(n-1)}{2}) = O(n^2)$.

(c) Seien $L_\ell := (a_1, a_2, \ldots, a_{\lceil \frac{n}{2} \rceil})$ die linke und $L_r := (b_1, b_2, \ldots, b_{\lfloor \frac{n}{2} \rfloor})$ die rechte Hälfte der Liste L, und beide Hälften der Liste seien aufsteigend sortiert. Wir modifizieren den Merge-Schritt von Mergesort, um eine aufsteigend sortierte Liste aller Elemente L' zu erhalten und gleichzeitig die Anzahl der Inversionen in L zu zählen. Zur Erinnerung: Mergesort verwendet zwei Pointer i für L_ℓ und j für L_r. Zu Beginn ist $i := 1$ und $j := 1$ und L' die leere Liste. Falls nun $a_i \leq b_j$ ist, wird a_i an L' angehängt und $i := i + 1$ gesetzt. Andernfalls wird b_j an die Liste angehängt und $j := j + 1$ gesetzt. Dies wird wiederholt, bis alle Elemente in L' eingeordnet wurden.

Wir verwenden nun zusätzlich einen Zähler c, der die Inversionen zählt. Im Fall $a_i \leq b_j$ ändern wir c nicht, aber im Fall $a_i > b_j$ erhöhen wir c um die Anzahl der verbleibenden Elemente in L_ℓ, d.h. $c := c + (\lceil \frac{n}{2} \rceil - i + 1)$.

Korrektheit: Wir argumentieren, dass jede Inversion genau einmal gezählt wird. Immer wenn ein b_j an L' angehängt wird, stellt b_j mit allen verbleibenden a_k mit $k \geq i$ eine Inversion dar. Für die a_k mit $k < i$ stellt b_j keine Inversion dar. Somit werden alle Inversionen in denen b_j vorkommt gezählt. Das geschieht für jedes b_j genau einmal, da es genau einmal an die Liste L' angehängt wird.

Laufzeit: Da das erhöhen von c jeweils nur einen Schritt benötigt, bleibt die Laufzeit des Merge-Schritts $O(|L|)$, wobei $|L|$ die Länge der Liste L bezeichnet.

(d) Wir verwenden den Mergesort Algorithmus und modifizieren den Merge-Schritt wie in (c) beschrieben. Daraus erhalten wir den Algorithmus MERGE-AND-COUNT, der für zwei sortierte Listen (L_ℓ, L_r) die Anzahl Inversionen c und die sortierte Liste L' zurückgibt.

Algorithm 2 SORT-AND-COUNT(L)

```
if |L| = 1 then
  return (0, L)
end if
if L > 1 then
  Teile die Liste in zwei Hälften L_ℓ and L_r mit |L_ℓ| = ⌈n/2⌉ und |L_r| = ⌊n/2⌋
  (c_ℓ, L_ℓ) = SORT-AND-COUNT(L_ℓ)
  (c_r, L_r) = SORT-AND-COUNT(L_r)
  (c, L') = MERGE-AND-COUNT(L_ℓ, L_r)
end if
return (c + c_ℓ + c_r, L')
```

Korrektheit: Da der Algorithmus MERGE-AND-COUNT gemäss (c) in jedem Merge-Schritt die Inversionen der sortierten Listen korrekt zählt, und man die Inversionen zählen kann indem man zuerst die Inversionen der beiden Hälften zählt, diese sortiert und dann die Anzahl Inversionen mit je einem Element in der linken und rechten Hälfte dazuadiert, folgt die Korrektheit.

Laufzeit: Da MERGE-AND-COUNT(L_ℓ, L_r) in Zeit $O(|L_\ell| + |L_r|)$ läuft folgt aus der Laufzeitanalyse von Mergesort, dass die Laufzeit von SORT-AND-COUNT $O(n \log n)$ ist.

Lösungsvorschlag zu Aufgabe 3

a) Wir unterteilen das Array A der Länge $2m$ in m Zweiergruppen und Vergleichen die beiden Zahlen jeder Zweiergruppe. Falls die beiden Zahlen gleich sind, so fügen wir eine eine Zahl vom selben Wert ins Array B ein. Da wir pro Zweiergruppe maximal eine Zahl in B einfügen, hat B Länge $n' \leq m$. Falls x omnipräsent in A ist, so gibt es mehr Zweiergruppen in denen x zweimal vorkommt als Zweiergruppen in denen x nicht vorkommt. Daraus folgt, dass x auch omnipräsent in B ist. Die Anzahl benötigter Vergleiche um B zu erzeugen ist m.

b) Falls $n = 1$ so geben wir diese Zahl aus, ansonsten unterteilen wir das Array A der Länge $n = 2m + 1$ in m Zweiergruppen und eine 1-er Gruppe. Zuerst vergleichen wir die Zahl y der 1-er Gruppe mit allen anderen Zahlen. Falls y omnipräsent in A ist, so geben wir y aus. Ansonsten gehen wir mit den m Zweiergruppen analog zu a) vor. Wir wissen dass eine in A omnipräsente Zahl x öfter als m mal in den m Zweiergruppen vorkommen muss und sie deshalb analog zu a) auch omnipräsent im neu erzeugten Array B vorkommt. Die Anzahl benötigter Vergeleiche ist $3m$.

c) Wir gehen rekursiv vor um einen Kandidaten für x zu finden. Falls die Anzahl Elemente in A gerade ist, benutzen wir den Algorithmus aus a) ansonsten den aus b). Wir fahren mit dem neu erhaltenen Array fort bis der Algorithmus aus b) einen Kandidaten für x liefert (entweder weil das aktuelle Array Länge 1 hat oder weil die Zahl aus der 1-er Gruppe omnipräsent im aktuellen Array ist). In diesem Fall testen wir ob der Kandidat omnipräsent in A ist (durch Vergleichen mit allen Elementen). Falls der Kandidat omnipräsent ist, geben wir ihn aus, ansonsten geben wir aus, dass keine solche Zahl existiert. Falls das Array Länge 0 erreicht ohne, dass ein Kandidat gefunden wurde, so geben wir aus, dass keine solche Zahl existiert.

Korrektheit: Wenn obiger Algorithmus eine Zahl ausgibt, ist diese sicherlich omnipräsent, da genau dies vor der Ausgabe getestet wurde. Für die Korrektheit bleibt zu zeigen, dass auch die Ausgabe 'Es gibt keine omnipräsente Zahl' korrekt ist. Dies ist äquivalent zur Aussage: Wenn eine Zahl x omnipräsent in A ist, wird diese vom Algorithmus gefunden und ausgegeben. Wir haben in a) und b) gezeigt, dass in diesem Fall x entweder vom Algorithmus b) als Kandidat entdeckt wird oder x im neu erzeugten Array omnipräsent ist. Das Zweite im-

pliziert insbesondere, dass der neu erzuegte Array nicht Länge 0 hat. Da das Array immer kürzer wird, muss x irgendwann vom Algorithmus b) als Kandidat gefunden werden.

Laufzeit: Sei S_n die Worst-case Anzahl benötigter Vergleiche um für ein Array der Länge n einen Kandidaten zu finden. Wir zeigen per Induktion über n, dass $S_n \leq 3n$ gilt. Für $n = 1$ werden keine Vergleiche benötigt. Für den Induktionsschritt sei $k \leq \frac{n}{2}$ die Länge des neu erzeugten Arrays B. Wir haben oben gezeigt, dass wir maximal $\frac{3n}{2}$ Vergleiche benötigen um B zu erzeugen. Es folgt dass

$$S_n \leq \frac{3n}{2} + S_k \leq \frac{3n}{2} + 3k \leq 3n \,,$$

wobei die zweite Ungleichung aus der Induktionsannahme und die dritte aus $k \leq \frac{n}{2}$ folgt. Anschliessend wird der gefundene Kandidat mit allen Elementen verglichen. Für die gesamte Anzahl Vergleiche L gilt somit $L \leq 4n$.