Algorithmen und Datenstrukturen: Übung 3

Alexander Waldenmaier

26. November 2020

Aufgabe 3.1

Es handelt sich um den Algorithmus *select* unter Verwendung eines speziellen Pivotelements. Dazu wird die Grundmenge der Länge n in c regelmäßige Teilmengen aufgeteilt, innerhalb jeder dieser Teilmengen der Median bestimmt und anschließend der k/c-kleinste Median als Pivotelement verwendet.

a) Durch die Vergrößerung der Untermengen von 5 auf 7 Elemente ergeben sich folgende Änderungen:

- Die Anzahl an Vergleichen zur Bildung des Medians ist nun d, statt wie vorher 7.
- Die Anzahl an Elementen, die definitiv größer als das k-größte Element sind, beträgt nun $3 \cdot \frac{n}{7} \cdot \frac{1}{2} = \frac{2}{7}n$, weshalb im worst-case nun ein rekursiver Aufruf mit $1 - \frac{2}{7}n = \frac{5}{7}n$ stattfindet (vorher waren es $\frac{3}{10}n$).

$$V(n) \leq \underbrace{d\frac{n}{7}}_{\substack{\text{Medianbildung der} \\ \text{Untermengen}}} + \underbrace{V\left(\frac{n}{7}\right)}_{\substack{\text{k/7-kleinster} \\ \text{Median}}} + \underbrace{n-1}_{\substack{\text{Einordnung um} \\ \text{Pivotelement}}} + \underbrace{V\left(\frac{5}{7}n\right)}_{\substack{\text{Rekursiver} \\ \text{Aufruf}}}$$

Mit dem Master-Theorem lässt sich die Rekursionsgleichung abschätzen:

$$\begin{aligned} V(n) &\leq d\frac{n}{7} + V\left(\frac{n}{7}\right) + n - 1 + V\left(\frac{5}{7}n\right) \\ &= V\left(\frac{n}{7}\right) + V\left(\frac{5}{7}n\right) + n\left(\frac{d}{7} + 1\right) - 1 \text{ mit } \alpha_1 = \frac{1}{7}, \alpha_2 = \frac{5}{7}, m = 2, k = 1 \\ &\Rightarrow \sum_{i=1}^{m} \alpha_i^k = \frac{5}{7} + \frac{1}{7} = \frac{6}{7} < 1 \Rightarrow \text{Fall 1} \\ &\Rightarrow V_{wc}(n) \in \Theta(n) \end{aligned}$$

b) Im worst-case ist das Array vollständig „verkehrt“ herum sortiert und man wählt als Pivot-Element stets das letzte Element. In diesem Fall reduziert sich die Array-Größe in jedem rekursiven Aufruf gerade um 1: $\mathcal{O}(n^2)$.
Zur Wahl eines besseren Pivot-Elements wird der beschriebene Algorithmus verwendet. Im Fall einer Gesamtmenge mit Länge n und Untermengen der Länge 5 ergeben sich dabei stets $\frac{3}{10}n$ Elemente, die sicherlich größer als das Pivotelement sind. Diese können bei der Einordnung also automatisch „rechts“ des Pivotelements eingefügt werden. Genauso gibt es $\frac{3}{10}n$ Elemente, die sicherlich kleiner als das Pivotelement sind und somit automatisch „links“ des Pivotelements eingefügt werden müssen. Nur die verbleibenden $\frac{4}{10}n$ Elemente müssen nun mit dem Pivotelement verglichen und je entweder „links“ oder „rechts“ von diesem eingefügt werden. Im worst-case sind alle dieser verbleibenden $\frac{4}{10}n$ Elemente größer (oder kleiner) als das Pivotelement. In diesem Fall erfolgen anschließend zwei weitere rekursive

Aufrufe mit je $\frac{7}{10}n$ bzw. $\frac{3}{10}n$ Elementen. Die zugehörige Rekursionsgleichung lautet:

$$\begin{aligned} V(n) &\leq 7 \cdot \frac{n}{5} + V\left(\frac{n}{5}\right) + \frac{4}{10}n + V\left(\frac{7}{10}n\right) + V\left(\frac{3}{10}n\right) \\ &= V\left(\frac{n}{5}\right) + V\left(\frac{7}{10}n\right) + V\left(\frac{3}{10}n\right) + n\left(\frac{7}{5} + \frac{4}{10}\right) \\ &= V\left(\frac{n}{5}\right) + V\left(\frac{7}{10}n\right) + V\left(\frac{3}{10}n\right) + \frac{9}{5}n \text{ mit } \alpha_1 = \frac{1}{5}, \alpha_2 = \frac{7}{10}, \alpha_3 = \frac{3}{10}, m = 3, k = 1 \end{aligned}$$

$$\Rightarrow \sum_{i=1}^{m} \alpha_i^k = \frac{1}{5} + \frac{7}{10} + \frac{3}{10} = \frac{6}{5} > 1 \Rightarrow \text{Fall 3, wtf?!}$$

Eigentlich müsste hier Fall 2 rauskommen und damit $\Theta(n \log n)$. Irgendwie muss also meine Rekursionsgleichung falsch sein.

Aufgabe 3.2

i	l	r	a	Erläuterung
4	1	12	**13**,5,9,18,3,8,3,8,19,15,11,1	Anfangszustand
4	1	9	**5**,9,3,8,3,8,11,1,13,18,19,15	Pivotelement an Stelle 9, also $r = 9$
			3,3,1,5,9,8,8,11,13,18,19,15	Endzustand mit dem Pivotelement 5 an 4. Stelle

$\Rightarrow$ Das 4.-kleinste Element ist die 5.

Aufgabe 3.3

a) Zu Beginn wird die Variable $m = 0$ gesetzt. Anschließend wird das Array a durchiteriert und für jedes a_i festgestellt, ob dieses größer als m ist. Wenn ja, wird $m = a_i$ gesetzt. Dadurch findet sich das Maximum aller Elemente in a, der sich nun in der Variable m gespeichert befindet.

Nun wird ein Array b der Länge $m + 1$ erstellt und mit Nullen initialisiert. Anschließend wird das Array a erneut durchiteriert und jeder Wert a_i nun als Index des Arrays b verstanden. Nach Durchlaufen der Schleife entspricht jeder Wert b_j nun der Anzahl der Vorkommnisse der Zahl j im Array a. Anders gesagt: Das Intervall $[0, m]$ wurde damit in Teilintervalle (Buckets) der Länge 1 unterteilt und die Werte aus a nun in diese Buckets hineingelegt. Manche Buckets erhalten mehrmals einen Wert aus a, manche keinen.

Nun wird ein Array c erstellt, das wie a die Länge n besitzt und zunächst nicht initialisiert wird. Dann werden alle möglichen Indizes $i \in [0, m]$ von b durchiteriert und, und für jedes i so häufig i in aufeinanderfolgende Zellen von c geschrieben, bis die Zahl b_i erreicht ist (also bis alle Zahlen im Bucket „aufgebraucht“ wurden). Ist $b_i = 0$, wird dieses i nicht in c erscheinen. Das Resultat ist das Array a in sortierter Reihenfolge.

b) Die erste for-Schleife benötigt stets n Durchläufe zum Finden des Maximums. Auch die zweite for-Schleife benötigt nur n Schritte zum Zählen der Vorkommnisse einer Zahl i in a.

In der letzen for-while-Schleife kann jedoch ein unerwartet hoher Rechenaufwand entstehen: Angenommen, in a befinden sich zunächst die Ziffern 0-9 und an 11. Stelle die Zahl 1000, dann werden die ersten zwei Schleifen noch immer n Durchläufe benötigen (mit $n = 11$). Die letzte for-Schleife wird allerdings 1001 Mal durchlaufen werden, wobei in den ersten 10 Fällen die while-Bedingung einmalig erfüllt wird, dann 990 Mal nicht erfüllt wird (da $b_i = 0$ für diese i) und schließlich noch einmal erfüllt wird (bei $i = 1000$). Insgesamt entstehen in der letzten Schleife also mehr als n Durchläufe, und zwar $\sim \max(a)$ Durchläufe.

Der Algorithmus benötigt also nur dann $\mathcal{O}(n)$ Rechenschritte, wenn die in a vorkommenden Zahlen

„homogen“ sind, also keine großen Lücken (idealerweise keine Lücken) zwischen den individuellen Zahlen vorhanden sind.

Aufgabe 3.4

a)

$$\text{GaU} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \begin{cases} 1 & \text{wenn } a_i > a_j \\ 0 & \text{sonst} \end{cases}$$

Insgesamt gibt es $(n-1)+(n-2)+...+1+0 = n\frac{n-1}{2} = \frac{n^2-n}{2}$ mögliche Paarungen. Wenn jede davon in der obigen Gleichung „eine 1 ergibt“, folgt für den worst-case (den super-GaU):

$$\text{GaU}_{wc} = \frac{n^2-n}{2}$$

b) Angenommen, ein Array ist bereits vollständig sortiert - lediglich die letzten zwei Elemente sind vertauscht. Der GaU vor der Vertauschung ist 1, da lediglich das vorletzte Element gepaart mit dem letzten Element „eine 1 ergibt“ (alle anderen Paare ergeben eine Null, da die Sortierung bereits stimmt). Somit ändert sich der GaU um -1.

Ein zweites Beispiel: Wenn das Array vollständig sortiert ist und sich ledglich das größte Element ganz vorne im Array der Länge n befindet, ergibt sich der GaU zu $n-1$ (da jede Paarung des ersten Elements mit allen nachfolgenden die Bedingung $a_0 > a_j$ erfüllt). Nach der Vertauschung der ersten beiden Elemente verringert sich der GaU zu $n-2$ (da jede Paarung des zweiten Elements mit allen nachfolgenden die Bedingung $a_1 > a_j$ erfüllt). Erneut verringert sich der GaU lediglich um 1.

Das ist in der Tat immer der Fall, da eine Vertauschung nur einen Einfluss auf die betrachtete Paarung hat, nicht aber auf alle anderen Paarungen. Schließlich ist es egal, ob zwei Elemente „weiter hinten“ im Array ihren Platz tauschen - die Anzahl der Paarungen die eine „1 bzw. 0 ergeben“ bleibt für alle unbeteiligten Elemente die selbe.

c) Ist ein Array in umgekehrter Reihenfolge sortiert, sind also alle Elemente größer als ihre Vorgänger, so liegt ein maximaler GaU vor. In diesem Fall ergibt jede Paarung eine 1 und der GaU beträgt wie bereits in a) beschrieben $\frac{n^2-n}{2}$.

d)

```
1: procedure BERECHNEGAU(a, n)
2:     gau ← 0
3:     for i ← 0 to n − 2 do
4:         for j ← i + 1 to n − 1 do
5:             if a[i] > a[j] then
6:                 gau ← gau + 1
7:     return gau
```

Aufgabe 3.5

Abgabe in DOMjudge. Teamname: "test"