Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

# Algorithmen und Datenstrukturen
# Sommersemester 2024
# Musterlösung Übungsblatt 2

Abgabe: Dienstag, 30. April, 2024, 10:00 Uhr

## Aufgabe 1: $\mathcal{O}$-Notation *(9 Punkte)*

Beweisen oder widerlegen Sie die folgenden Aussagen anhand der *Mengendefinition* der $\mathcal{O}$-Notation (Vorlesungsfolien Woche 2, Folie 11 und 12).

(a) $2n^3 - 5 \cdot n^2 + 1 \in \mathcal{O}(n^4)$ *(1 Punkt)*

(b) $\log_3(n) \in o(\log_5(n))$ *(2 Punkte)*

(c) $n! \in \Omega(2^n)$ *(2 Punkte)*

(d) $\log_2\left(n^2\right) \in \omega((\log_2 n)^2)$ *(2 Punkte)*

(e) $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ für nicht negative Funktionen $f$ und $g$. *(2 Punkte)*

## Musterlösung

(a) Wahr. Sei $n_0 = 1$ und $c = 3$. Für $n \geq n_0$ gilt $n^4 \geq n^3 \geq 1$ und daher

$$2n^3 - 5 \cdot n^2 + 1 \leq 2n^3 + 1 \leq 2n^4 + n^4 = 3n^4 = cn^3$$

(b) Falsch. Durch die Anwendung der Logarithmengesetzte gilt für alle $n \geq 1$.

$$\log_3(n) = \frac{\log_5(n)}{\log_5(3)} > 1.46 \cdot \log_5(n) > 1 \cdot \log_5(n).$$

Das bedeutet aber dass wir für $c = 1$ kein $n_0 > 0$ finden können so dass die Aussage wahr ist.

(c) Wahr. Wähle $c = 1$ und $n_0 = 8$. Dann gilt für $n \geq n_0$

$$n! \geq \underbrace{n}_{\geq 4} \cdot \underbrace{(n-1)}_{\geq 4} \cdot \underbrace{(n-2)}_{\geq 4} \cdot \ldots \cdot \underbrace{\lfloor \frac{n}{2} \rfloor}_{\geq 4} \geq 4^{n/2} = 1 \cdot 2^n$$

*Hinweis:* Ein sehr ähnlicher Beweis ergibt sich unter Verwendung der (in der Vorlesung verwendeten) Abschätzung $n! \geq (n/2)^{n/2}$.

(d) Falsch. Für alle $n > 4$ gilt $2 < \log_2(n)$ und somit auch

$$\log_2(n^2) = 2\log_2(n) < \log_2(n) \cdot \log_2(n) = 1 \cdot (\log_2(n))^2.$$

Dies impliziert dass man für $c = 1$ kein $n_0 > 0$ wählen können so dass für alle $n \geq n_0$ die Ungleichung $\log_2(n^2) \geq c(\log_2 n)^2$ gilt (da wir eben schon bei $n = \max\{5, n_0\}$ einen Widerspruch bekommen).

(e) Wahr. Wähle $n_0 = 1$, $c_1 = \frac{1}{2}$ und $c_2 = 1$. Dann gilt für $n \geq n_0$

$$c_1 \cdot (f(n) + g(n)) \leq \max\{f(n), g(n)\} \overset{f,g \geq 0}{\leq} c_2(f(n) + g(n))$$

## Aufgabe 2: Sortieren nach Asymptotischem Wachstum *(4 Punkte)*

Sortieren Sie folgende Funktionen nach asymptotischem Wachstum. Schreiben Sie $g <_{\mathcal{O}} f$ falls $g \in \mathcal{O}(f)$ *und* $f \notin \mathcal{O}(g)$. Schreiben Sie $g =_{\mathcal{O}} f$ falls $f \in \mathcal{O}(g)$ *und* $g \in \mathcal{O}(f)$ (kein Beweis nötig).

| | | | |
|---|---|---|---|
| $\sqrt{n} \cdot n^{3/2}$ | $3^n$ | $\frac{1}{4} \cdot n!$ | $27 \cdot n$ |
| $4^{n/2}$ | $100 \cdot n^{100}$ | $\log_2(n^3)$ | $n^n$ |
| $12 \cdot \sqrt{\log_2(n)}$ | $21 \cdot \log_2(\sqrt{n})$ | $(n-1)!$ | $\log_2(n^n)$ |

## Musterlösung

$$12 \cdot \sqrt{\log_2(n)} <_{\mathcal{O}} 21 \cdot \log_2(\sqrt{n}) =_{\mathcal{O}} \log_2(n^3) <_{\mathcal{O}} 27 \cdot n <_{\mathcal{O}} \log_2(n^n) <_{\mathcal{O}} \sqrt{n} \cdot n^{3/2}$$
$$<_{\mathcal{O}} 100 \cdot n^{100} <_{\mathcal{O}} 4^{n/2} <_{\mathcal{O}} 3^n <_{\mathcal{O}} (n-1)! <_{\mathcal{O}} \frac{1}{4} n! <_{\mathcal{O}} n^n$$

## Aufgabe 3: k-MergeSort *(7 Punkte)*

In Übungsblatt 1 ging es darum eine Variante des Mergesort Algorithmus zu implementieren, welche für einen gegebenen Parameter $k > 1$, das gegebene Array in $k$ Teile der Größen $\mathcal{O}(n/k)$ zerlegt, wobei $n$ die Größe des Arrays ist. Wir wollen hier nun die Laufzeit dieser Variante analysieren.

a) Sei $T(n, k)$ die Laufzeit für obigen Algorithmus mit Parametern $n$ und $k$. Geben Sie eine rekursive Formel für $T(n, k)$ an (analog zur Folie 24, Foliensatz 2). Zur Einfachheit können Sie annehmen dass der Algorithmus das Array in jedem rekursiven Schritt in $k$ Teile der Größe genau $n/k$ teilt. *(3 Punkte)*

b) Zeigen Sie mithilfe von vollständiger Induktion dass $T(n, k) = \mathcal{O}(\log_k(n) \cdot n \cdot k)$. *(3 Punkte)*

c) Setzen Sie die Werte $2, 3, \log_2(n), n/4$ für $k$ ein. Für welche dieser Werte ist die Laufzeit asymptotisch am besten? *(1 Punkt)*

## Musterlösung

a) Der Basisfall ist $T(1, k) = O(1)$ da ein einelementiges Array schon sortiert ist. Nun zum allgemeinen Fall mit $n$ Elementen: Wir teilen dass Array in $k$ teile mit jeweils $n/k$ elementen. Jedes dieser Teile kann in Zeit $T(n/k, k)$ sortiert werden. Nun müssen diese Teile sinnvoll "gemergt" werden. Hierfür speichert der Algorithmus $k$ Pointer (einer für jeden der $k$ Teile) und sucht das Minimum der Elemente auf die diese Pointer zeigen (und fügt dieses Minimum der Ausgabe hinzu). Dieser Pointer wird nun inkrementiert. Um das Minimum aus $k$ Werten zu finden benötigt man $O(k)$ Zeit. Dieser Vorgang wird $n$-mal wiederholt bis jedes der $n$ Elemente in unseren Ausgabearray kopiert wurden. Die Laufzeit für den *Merge* ist also $\mathcal{O}(n \cdot k)$. Die rekursive Formel ist demnach:

$$T(n, k) = k \cdot T\left(\frac{n}{k}, k\right) + \mathcal{O}(n \cdot k)$$

b) Sei $c$ die in der $\mathcal{O}$- Notation "versteckte" Konstante, also sei $T(1, k) \leq c$ und $T(n, k) \leq k \cdot T\left(\frac{n}{k}, k\right) + c \cdot n \cdot k$. Wir zeigen nun induktiv dass $T(n, k) \leq c \cdot (\log_k(n) + 1) \cdot n \cdot k$.
**Induktionsanfang** $n = 1$**:** $T(1, k) \leq c = c \cdot k = c \cdot (\underbrace{\log_k(1)}_{=0} + 1) \cdot 1 \cdot k$.

**Induktionsvorraussetzung:** Die Aussage gilt für alle Werte $< n$.

**Induktionsschritt:**

$$\begin{aligned}
T(n,k) &\leq k \cdot T\left(\frac{n}{k}, k\right) + c \cdot n \cdot k \\
&\leq k \cdot \underbrace{c \cdot \left(\log_k\left(\frac{n}{k}\right) + 1\right) \cdot \frac{n}{k} \cdot k}_{IV} + c \cdot n \cdot k \\
&= c \cdot k \cdot (\log_k(n) - \log_k(k) + 1) \cdot n + c \cdot n \cdot k \\
&= c \cdot k \cdot \log_k(n) \cdot n + c \cdot n \cdot k \\
&= c \cdot (\log_k(n) + 1) \cdot n \cdot k
\end{aligned}$$

c) Für konstante Werte wie $k = 2$ oder $k = 3$ beträgt die Komplexität des Algorithmus $\mathcal{O}(n \log(n))$ (also genau so wie die des Mergesorts aus der Vorlesung). Für $k = \log(n)$ bekommen wir eine Laufzeit von

$$\mathcal{O}(\log_{\log n}(n) \cdot n \cdot \log(n)) = \mathcal{O}\left(n \cdot \frac{\log^2(n)}{\log\log(n)}\right)$$

Diese Laufzeit ist schlechter als in den obigen Fällen, da wir hier den zusätzliche Faktor $\log(n)/\log(\log(n))$ in der Laufzeit haben.

Setzt man $k = n/4$ wird die Laufzeit zu $\mathcal{O}(n^2)$ und ist damit asymptotisch so gut wie InsertionSort oder SelectionSort aber schlechter als MergeSort (und schlechter als der $k = \log(n)$ Fall).