Datenstrukturen und Algorithmen
Übung 2 – Landau-Notation

Musterlösung

1. Zeigen Sie **formal** anhand der Definitionen aus der Vorlesung, dass die Laufzeit eines Algorithmus genau dann in $\Theta(g(n))$ ist, wenn seine Worst-Case-Laufzeit in $\mathcal{O}(g(n))$ und seine Best-Case-Laufzeit in $\Omega(g(n))$ ist. **(1 Punkt)**

   Die Aussage "Die Best-Case-Laufzeit ist $\Omega(g(n))$" heisst, es gilt für die Laufzeit die untere Schranke $T(n) = \Omega(g(n))$. Dies bedeutet gemäss Definition der Landau-Notation: Es existieren positive Konstanten c_1 und n_0, so dass $0 <= c_1 g(n) <= T(n)$ für alle $n >= n_0$.

   Die Aussage "Die Worst-Case-Laufzeit ist $\mathcal{O}(g(n))$" heisst, es gilt für die Laufzeit die obere Schranke $T(n) = \mathcal{O}(g(n))$. Dies bedeutet gemäss Definition der O-Notation: Es existieren positive Konstanten c_2 und n_0', so dass $0 <= T(n) <= c_2 g(n)$ für alle $n >= n_0'$.

   Falls für die Laufzeit gleichzeitig gilt $T(n) = \Omega(g(n))$ und $T(n) = O(g(n))$ bedeutet dies deshalb: Es existieren positive Konstanten c_1, c_2 und $n_0^* = \max\{n_0, n_0'\}$, so dass $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$ für alle $n >= n_0^*$. Dies entspricht aber genau der Definition der Notation, deshalb gilt also $T(n) = \Theta(g(n))$.

   Anmerkung: Anders sähe es aus, wenn die Best-Case-Laufzeit in $\mathcal{O}(g(n))$ und die Worst-Case-Laufzeit in $\Omega(g(n))$ wäre: dies macht keine besonders nützliche Aussage über das Laufzeitverhalten, da der Bestcase weitaus schneller als $g(n)$ und der Worst-Case weitaus schlechter als $g(n)$ sein könnte. Somit müsste in diesem Fall der Algorithmus nicht unbedingt in $\Theta(g(n))$ sein.

2. Verwenden Sie die Rechenregeln für Logarithmen...

   (a) ...um zu zeigen dass $a^{\log_b n} = n^{\log_b a}$.

$$
\begin{aligned}
& a^{\log_b n} = n^{\log_b a} \\
\iff & \log_a a^{\log_b n} = \log_a n^{\log_b a} \\
\iff & \log_b n = \log_a n^{\log_b a} \\
\iff & \log_b n = \log_b a \cdot \log_a n \\
\iff & \frac{\log_b n}{\log_b a} = \log_a n \\
\iff & \log_a n = \log_a n
\end{aligned}
$$

Oder

$$\begin{aligned} a^{\log_b n} &= n^{\log_n a^{\log_b n}} \\ &= n^{\log_b n \cdot \log_n a} \\ &= n^{\frac{\ln n}{\ln b} \cdot \frac{\ln a}{\ln n}} \\ &= n^{\frac{\ln a}{\ln b}} \\ &= n^{\log_b a} \end{aligned}$$

(b) ...um zu zeigen dass $\Theta(\log_a n) = \Theta(\log_b n)$.

Um zu zeigen, dass $\Theta(\log_a n) = \Theta(\log_b n)$, genügt es zu zeigen, dass es eine Konstante c gibt, so dass $\log_a n = c \cdot \log_b n$.

$$\begin{aligned} \log_a(n) &= \log_a \left(b^{\log_b n} \right) \\ &= \log_b n \cdot \underbrace{\log_a b}_{=:c} \end{aligned}$$

Alternativ:

$$\begin{aligned} \log_a n &= c \cdot \log_b n \\ \frac{\log_b n}{\log_b a} &= c \cdot \log_b n \\ \frac{1}{\log_b a} &= c \end{aligned}$$

Logarithmen mit unterschiedlicher Basis unterscheiden sich also nur um einen konstanten Faktor.

(c) Gilt auch $\Theta(a^n) = \Theta(b^n)$ wenn $0 < a < b$? Begründen Sie.

Nein, Gegenbeispiel: mit $a = 1$ und $b = 2$ ist $a^n = 1^n = 1$ konstant, während b^n exponentiell wächst.

Interessanter wird die Frage für $1 < a < b$: Angenommen, die Aussage sei wahr. Aus $\Theta(a^n) = \Theta(b^n)$ folgt $b^n \in \mathcal{O}(a^n)$. Dann gibt es ein c und n_0, so dass $b^n \leq c \cdot a^n$ für alle $n > n_0$. Umgeformt ergibt dies $c \geq \left(\frac{b}{a}\right)^n$. Wenn aber $b > a$, so ist $\frac{b}{a} > 1$, und für jedes feste c finden wir ein n, ab dem dieser Ausdruck nicht mehr gilt. Widerspruch zur Annahme, die Aussage kann also nicht wahr sein.

Folglich unterscheiden sich die exponentiellen Komplexitätsklassen je nach Exponent!

(1.5 Punkte)

3. Zeigen Sie mittels Induktion/der Substitutionsmethode, dass die Rekursionsgleichung

$$T(n) = 2T(\lceil n/4 \rceil + 12) + 3n$$

die Lösung $\mathcal{O}(n)$ hat. **(1 Punkt)**

Zunächst bemerken wir, dass erst für $n > 17$ der Rekursionsschritt zu einem kleineren n führt, also $\lceil n/4 \rceil + 12 < n$ gilt. Daher ist es hilfreich, ein $n_0 > 17$ zu wählen, wenn wir den Rekursions-Basisfall $T(n)$ für $n < n_0$ auf eine Konstante C festlegen.

Wir wollen zeigen:

$$\exists c, n_0 : T(n) \leq cn \quad \forall\, n > n_0$$

Wie in der Substitutionsmethode üblich beginnen wir mit dem Induktionsschritt:

Induktionsschritt Annahme: $T(n') \leq cn'$ für alle $n' < n$. Zu zeigen: $T(n) \leq cn$.

$$\begin{aligned}
T(n) &= 2T(\left\lceil \frac{n}{4} \right\rceil + 12) + 3n && \text{Definition} \\
&\leq 2c(\left\lceil \frac{n}{4} \right\rceil + 12) + 3n && \text{Induktionsannahme} \\
&\leq 2c(\frac{n}{4} + 13) + 3n && \text{Ausklammern, Aufrunden} \\
&= 26c + (\frac{1}{2}c + 3)n && \text{Umklammern} \\
&= \underbrace{cn}_{\text{zu zeigen}} + \underbrace{26c - (\frac{1}{2}c - 3)n}_{\text{Rest}} && \text{Ziel ausklammern} \\
&\leq cn && c > 6 \;\; n > \frac{26c}{0.5c - 3}
\end{aligned}$$

Im letzten Schritt wählen wir c erst so, dass der Restterm, der von n abhängig ist ($\left(\frac{1}{2}c - 3\right) n$) auf jeden Fall negativ ist (also $c > 6$). Da der Restterm für solche c immer kleiner wird, je grösser n gewählt wird, ist es ein leichtes, ein n zu wählen dass der ganze Rest negativ ist.

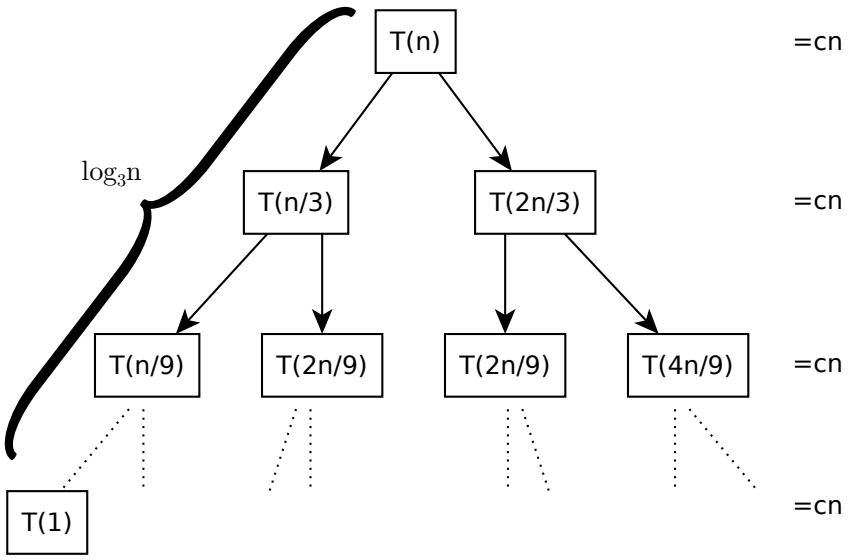
Induktionsverankerung Wir wählen $c = \max\{8, C\}$ und $n_0 = \lceil 26c/(0.5c - 3) \rceil$, und nehmen an, dass $T(n) = C$ für alle $n < n_0$. (Für $C = 1$ resultiert dies bspw. in $n_0 = 208$). Die 8 ist hier als beliebige Zahl > 6 gewählt - wichtig ist, dass $c \geq C$ und $c > 8$ gilt.

Da unser Induktionsschritt nicht von der Gültigkeit der Aussage für **ein** n ausgeht, sondern von der Gültigkeit für **alle** $n' < n$, muss unsere Induktionsverankerung dazu kompatibel sein, wir wollen also $T(n) \leq cn$ für alle $n \leq n_0$ zeigen. Für n_0 gilt dies nach Konstruktion. Da wir die Laufzeit C für alle n darunter festlegen, und nach Konstruktion $c \geq C$ gilt, ist auch $\forall n < n_0, n \geq 1 : \quad T(n) = C \leq c \leq cn$.

4. Zeichnen Sie einen Rekursionsbaum für die Gleichung

$$T(n) = T(n/3) + T(2n/3) + cn.$$

Erklären Sie anhand des Baumes, dass die Lösung der Gleichung in $\Omega(n \log n)$ ist. **(1 Punkt)**



Der Rekursionsbaum ist asymmetrisch: Wenn wir immer den $T(n/3)$-Rekursionsaufrufen (in der Skizze jeweils links) folgen, erhalten wir eine Tiefe von $\log_3 n$ Schritten; auf der rechten Seite jedoch eine Tiefe von $\log_{\frac{3}{2}} n$ Schritten (mehr als $\log_3 n$). Wenn wir die nichtrekursiven Anteile von cn auf jeder Ebene zusammenzählen, erhalten wir bis zur eingezeichneten Tiefe ($\log_3 n$) immer genau cn Elemente.

Wir betrachten also den Teilbaum mit dieser Tiefe - er enthält $\log_3 n$ Ebenen mit einer summierten Laufzeit von jeweils cn, also beträgt die Gesamtlaufzeit dieses Teilbaums $cn \log_3 n \in \Theta(n \log n)$. Der vollständige Rekursionsbaum kann nur eine höhere Laufzeit enthalten, somit liegt die Gesamtlaufzeit $T(n) \in \Omega(n \log n)$

5. Die Rekursionsgleichung für die Zeitkomplexität der binären Suche ist

$$T(n) = T(n/2) + \Theta(1).$$

Verwenden Sie die Mastermethode, um zu zeigen dass $T(n) = \Theta(\log n)$. **(1 Punkt)**

Die Mastermethode ist anwendbar auf Funktionen der Form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

wobei $a \geq 1$ und $b > 1$ gilt.

In unserem Fall sind $a = 1$, $b = 2$ und $f(n) \in \Theta(1)$ – Voraussetzungen erfüllt ✓.

Mit $\log_b a = \log_2 1 = 0$ gilt $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$, also können wir Fall 2 der Mastermethode anwenden. Demnach ist $T(n) \in \Theta(n^{\log_b a} \cdot \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$

6. Berechnen Sie die lösbare Problemgrösse in der gegebenen Zeit für Algorithmen mit verschiedener Zeitkomplexität, welche in der Tabelle gegeben sind. Nehmen Sie an, jede Operation dauere $0.01s$. **(2.5 Punkte)**

$T(n)$	Problemgrösse lösbar in 10s	Problemgrösse lösbar in 1000s
$10n$	$10 \overset{!}{\geq} .01T(n) = .01 \cdot 10n$ $\iff \boxed{n \leq 100}$	$1000 \overset{!}{\geq} .01T(n) = .01 \cdot 10n$ $\iff \boxed{n \leq 10'000}$
$2n^3$	$10 \overset{!}{\geq} .01T(n) = 0.02n^3 \iff 500 \geq n^3 \iff \sqrt[3]{500} \geq n \iff \boxed{n \leq 7}$	$1000 \overset{!}{\geq} .01T(n)$ $\iff \sqrt[3]{50000} \geq n \iff \boxed{n \leq 36}$
$n^{2.5}$	$10 \overset{!}{\geq} .01T(n)$ $\iff 1'000 \geq n^{2.5} \iff \sqrt[2.5]{1000} > n$ $\iff \boxed{n \leq 15}$	$1000 \overset{!}{\geq} .01T(n)$ $\iff \sqrt[2.5]{100'000} \geq n$ $\iff \boxed{n \leq 100}$
$2\log_2(8n)$	$10 \overset{!}{\geq} .01T(n)$ $\iff 500 \geq \log_2(8n)$ $\iff 2^{500} \geq 8n \iff \boxed{n \leq 2^{497}}$	$1000 \overset{!}{\geq} .01T(n)$ $\iff 50'000 \geq \log_2(8n)$ $\iff 2^{5'0000} \geq 8n \iff \boxed{n \leq 2^{49997}}$
2^{2n}	$10 \overset{!}{\geq} .01T(n) \iff 2^{2n} < 1000$ $\iff 2n \leq \log_2 1000 \iff \boxed{n \leq 4}$	$1000 \overset{!}{\geq} .01T(n) \iff 2^{2n} < 100'000$ $\iff 2n \leq \log_2 100'000 \iff \boxed{n \leq 8}$

Hinweis: Die Äquivalenzumformungen beruhen auf $n \in \mathbb{N}^{>0}$ sowie auf dem monotonen Wachstum von $n \mapsto n^\alpha$, $n \mapsto \log_\alpha n$ und $n \mapsto \alpha^n$ für $\alpha > 0$

7. Geben Sie die asymptotische Laufzeit dieses Algorithmus in Abhängigkeit von n an. Verwenden Sie die Θ-Notation. Geben Sie eine Summenformel für die Laufzeit an. Hinweis: Verwenden Sie die Partialsummenformel für geometrische Reihen ($\sum_{k=0}^{n} a_0 q^k = a_0 \frac{q^{n+1}-1}{q-1}$). **(1 Punkt)**

```
1  i = 1
2  while i < n
3      j = 0
4      while j ≤ i
5          j = j + 1
6      i = 3i
```

Für hinreichend grosses n gilt:

$$\begin{aligned} T(n) &= \sum_{k=0}^{\lfloor \log_3 n \rfloor - 1} 3^k \\ &\overset{(\star)}{=} \frac{3^{\lfloor \log_3 n \rfloor - 1 + 1} - 1}{3 - 1} \\ &\approx \frac{3^{\log_3 n} - 1}{2} \\ &= \frac{n-1}{2} \\ &\in \Theta(n) \end{aligned}$$

$(\star)$: Anwendung der Partialsummenformel der geometrischen Reihe

8. **Implementierungsdetails von MERGE-SORT**

   In der ersten Übungsserie hatten wir Ihnen eine Java-Implementierung von MERGE-SORT zur Verfügung gestellt.

Einer Ihrer Mitstudierenden hat sich die Zeit genommen, diese Implementierung mithilfe von *Property-Based Testing*[1] auf Fehler zu untersuchen und konnte einen Bug identifizieren. Die folgende Eingabe verursacht eine **IndexOutOfBoundsException**.

```
1 public void brokenExample() {
2     int[] arr = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2147483647, 0,
      ↪  0, -71, -2660, -1, 219, -115374, -5, 5294, -5939825,
      ↪  2147483647};
3     Sorting.mergeSort(arr, 0, arr.length-1);
4 }
```

(a) Finden und erklären Sie den Bug. Welche undokumentierte Annahme liegt dem Fehler zugrunde, d.h. für welche Eingaben arbeitet er korrekt? **(1 Punkt)**

(b) Beheben Sie den Bug[2]. Achten Sie darauf, dass ihre Implementierung weiterhin in Linearzeit arbeitet. Erarbeiten Sie Ihre Lösung selbstständig! **(1 Bonuspunkt)**

Siehe Slides/Video

[1] mit dem Tool *jquik* (https://jqwik.net/)

[2] Die undokumentierte Annahme zu dokumentieren, gilt nicht als Fix – der Code sollte für alle Eingaben funktionieren