Universität Karlsruhe
Fakultät für Informatik
ITI Wagner



# 1. Musterlösung

**Problem 1:** Average-case-Laufzeit vs. Worst-case-Laufzeit **

(a) Im schlimmsten Fall werden für jedes Element an der $i$-ten Stelle $i-1$ Vergleiche durchgeführt. Für $n$ Zahlen ergibt sich folgende Berechnung der Worst-case-Laufzeit $L(n)$

$$L(n) = \sum_{i=1}^{n} i - 1 = \left(\sum_{i=1}^{n} i\right) - n = \frac{n^2+n}{2} - n = \frac{n^2-n}{2} \in O(n^2).$$

(Anmerkung: Das Zurücklaufen entlang der sortierten Zahlen nach einer Folge von Tauschen wird hier vernachlässigt, da es asymptotisch keine Rolle spielt. Genau genommen sollte daher in obiger Gleichung ein zusätzlicher Faktor $c \geq 2$ auftreten.)

(b) Die höchste Laufzeit tritt genau dann auf, wenn das Array in der umgekehrten Reihenfolge sortiert ist, da in diesem Fall jedes Element $A[i]$ genau $i-1$ Mal vertauscht wird.

(c) Sei $X_i$ die Zufallsvariable, die die Anzahl der Tausche von $A[i]$ angibt. Unter der Annahme der Wahrscheinlichkeitsverteilung des Gartenzwergs ergibt sich

$$\begin{aligned}\frac{1}{2} \leq E(X_i) &= 0 \cdot \frac{1}{2} + 1 \cdot \left(\frac{1}{2} - \frac{1}{4}\right) + 2 \cdot \left(\frac{1}{4} - \frac{1}{8}\right) + 3 \cdot \left(\frac{1}{8} - \frac{1}{16}\right) + \dots \\ &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \\ &= \sum_{j=1}^{i-1} \frac{1}{2^j} \leq 1 \ , \ i = 2 \dots n.\end{aligned}$$

Die Average-case-Laufzeit $T^{\text{av}}(n)$ lässt sich wie folgt berechnen:

$$T^{\text{av}}(n) = \sum_{i=2}^{n} E(X_i).$$

Daraus ergibt sich folgende Abschätzung:

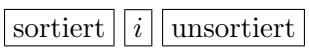
$$\frac{1}{2} \cdot (n-1) \leq T^{\text{av}}(n) \leq 1 \cdot (n-1).$$

Somit gilt:

$$T^{\text{av}}(n) \in \Theta(n).$$

(d) Das Resultat $\Theta(n)$ ist in Widerspruch mit der bewiesenen Schranke für die average-case-Laufzeit von Sortieralgorithmen, die in $\Omega(n \log n)$ liegt. Der Fehler des Gartenzwerges liegt in der falschen Annahme über die Wahrscheinlichkeit ($p = 1/2$) des Tausches eines Elements.

(e) Bei der Untersuchung des $i$-ten Elements hat das Array folgende Form erreicht:



Sei $j \in \{1, \ldots, i\}$. Aus der Menge der ersten $i$ Zahlen ist $A[i]$ mit Wahrscheinlichkeit $1/i$ das $j$-größte Element , da wir eine zufällige Menge von Zahlen betrachten. Dann muss $A[i]$ genau $j-1$ Mal vertauscht werden. Daraus ergibt sich

$$E(X_i) = \sum_{j=1}^{i} \frac{j-1}{i}.$$

Somit gilt für die average-case-Laufzeit $T^{\text{av}}(n)$ :

$$T^{\text{av}}(n) = \sum_{i=2}^{n} E(X_i) = \sum_{i=2}^{n} \sum_{j=1}^{i} \frac{j-1}{i} = \sum_{i=1}^{n} \frac{i-1}{2} + 0 = \frac{n^2-n}{4} \in \Theta(n^2).$$

**Problem 2:** Rekursive Suche *

Die Aufgabe lässt sich mittels einer binären Suche lösen. Der Algorithmus 1 zeigt eine mögliche Implementierung in rekursiver Form.

**Algorithmus 1** : Binärsuche

**Eingabe** : Sortiertes Array $A = (A[1], \ldots, A[n])$ von $n$ Zahlen und Zahl $T$.
**Ausgabe** : Falls $T$ in $A$ vorkommt, gebe **true** aus, ansonsten **false**.

```
function Binärsuche (A[1 · · n], T)
Wenn A nicht leer ist
    i ← ⌊n/2⌋
    Wenn T = A[i]
        return true
    sonst
        Wenn T < A[i]
            return Binärsuche (A[1 · · i − 1], T)
        sonst
            return Binärsuche (A[i + 1 · · n], T)
return false
```

Laufzeitanalyse:
Die Größe des zu durchsuchenden Arrays wird bei jeder Rekursionstiefe mindestens halbiert. Es können also höchstens $\log n$ Funktionsaufrufe durchgeführt werden. Da die Anzahl der Operationen pro Funktionsaufruf konstant ist, ergibt sich eine worst-case-Laufzeit von $O(\log n)$.

**Problem 3:** Gewichteter Median ***

Der Aufruf G-Median$(A, 0, 0)$ des folgenden Algorithmus liefert den gewichteten Median von $n$ Elementen.

Laufzeitanalyse:
Der Algorithmus bestimmt (wie bei SELECT, Schritte 1 bis 3) die Mengen $A_1$ und $A_2$. Die Berechnung der Summen $\sum_{x_i \in A_1} w_i$ und $\sum_{x_i \in A_2} w_i$ ist in $O(n)$ (wie im Schritt 4). Analog zum Schritt 5 benötigt der rekursive Aufruf von G-Median$(A_1, W_L, W_R)$ $T(7/10n+1)$ (nicht $T(7/10n+2)$, da wir $m$ nun explizit testen), wobei $T(n)$ die Laufzeit von G-Median für $n$ Elemente. Somit ergibt sich die selbe asymptotische Laufzeit von $O(n)$.

**Algorithmus 2** : G-MEDIAN$(A, W_L, W_R)$

**Eingabe** : Unsortiertes Array $A$ von $n$ Elementen $x_i$ mit Gewichten $w_i$ und ein Offset off
**Ausgabe** : Der gewichtete Median von $A$
$W \leftarrow 1/2(\sum_{x_i \in A} w_i + W_L + W_R)$
Wie im Algorithmus SELECT, wird das „mittlere der mittleren Elemente“ $m$ bestimmt.
$A_1$ : Elemente $x_i$ aus $A$ mit $x_i < m$
$A_2$ : Elemente $x_j$ aus $A$ mit $x_j > m$
**Wenn** $\sum_{x_i \in A_1} w_i + W_L \leq W$ *und* $\sum_{x_j \in A_2} w_i + W_R \leq W$
  gib $m$ aus
**sonst**
  **Wenn** $\sum_{x_i \in A_1} w_i + W_L \leq W$
    $W_L \leftarrow W_L + \sum_{x_i = m} w_i + \sum_{x_i \in A_1} w_i$
    **return** G-MEDIAN$(A_2, W_L, W_R)$
    **sonst**
      $W_R \leftarrow W_R + \sum_{x_i = m} w_i + \sum_{x_j \in A_2} w_j$
      **return** G-MEDIAN$(A_1, W_L, W_R)$

**Problem 4:** Laufzeit rekursiver Funktionen I *

Die Laufzeit $T(n)$ ist von der Form

$$T(n) = a \cdot T(n/b) + f(n)$$

mit $a = 4,\ b = 2,\ f(n) = n^3$.

Es gilt:

(a) $f(n) \in \Omega(n^{(\log_2 4)+\varepsilon})$ für $\varepsilon = 1$.

(b) $4 \cdot (n/2)^3 \leq c \cdot n^3$ für $c = 5/8 < 1$ und $n > n_0 = 1$.

Nach dem Aufteilungs-Beschleunigungssatz ergibt sich $T(n) \in \Theta(f(n)) = \Theta(n^3)$.

**Problem 5:** Laufzeit rekursiver Funktionen II ***

Betrachte die zweite Laufzeit mit Hilfe des allgemein formulierten Master-Theorems aus der Vorlesung. Es gilt:

$$\sum_{i=1}^{m} \alpha_i^k = \sum_{i=1}^{7} \left(\frac{1}{2}\right)^2 = \frac{7}{4} > 1$$

Somit gilt der dritte Fall:

$$T_A(n) \in \Theta(n^c) \quad \text{wobei für } c \text{ gilt:} \quad \sum_{i=1}^{7} \left(\frac{1}{2}\right)^c = 1 \quad \Rightarrow \quad \left(\frac{1}{2}\right)^c = \frac{1}{7}$$

$$\text{und somit:} \quad 2^c = 7 \quad \Rightarrow \quad c = \log_2 7$$

Für die erste Laufzeit hingegen gilt analog:

$$\sum_{i=1}^{m} \alpha_i^k = \sum_{i=1}^{a} \left(\frac{1}{4}\right)^2 = \frac{a}{16}$$

Somit gilt zunächst folgende Fallunterscheidung:

$$\begin{array}{lll} T_B(n) \in \Theta(n^2) & \text{falls} & a < 16 \\ T_B(n) \in \Theta(n^2 \log n) & \text{falls} & a = 16 \\ T_B(n) \in \Theta(n^c) & \text{falls} & a > 16 \quad \text{wobei wieder gilt} \quad \sum_{i=1}^{a} \left(\frac{1}{4}\right)^c = 1 \end{array}$$

Für $a \leq 16$ ist die Laufzeit $T_B(n)$ demnach geringer als $T_A(n)$. Beachte, dass wegen $\sum_{i=1}^{a} \alpha_i^k > 1 = \sum_{i=1}^{a} \alpha_i^c$ stets gilt $c > k$. Für $a > 16$ ist eine weitere Fallunterscheidung nötig, $c$ muss für alle Werte von $a$ berechnet werden. Berechne zunächst $c$ für $T_B(n)$:

$$\sum_{i=1}^{a} \left(\frac{1}{4}\right)^c = 1 \quad \Rightarrow \quad \left(\frac{1}{4}\right)^c = \frac{1}{a}$$

$$\text{somit gilt:} \quad 4^c = a \quad \Rightarrow \quad c = \log_4 a$$

Vergleiche nun, wann für $T_A(n)$ und $T_B(n)$ derselbe Wert $c$ gilt (Gleichstand):

$$\log_4 a = \log_2 7 \quad \Leftrightarrow$$

$$a = 4^{\log_2 7} = 2^{2 \log_2 7} = (2^{\log_2 7})^2 = 7^2 = 49$$

Somit ist im $\Theta$-Kalkül ein Algorithmus mit der Laufzeit $T_B(n)$ für $a < 49$ asymptotisch schneller als ein Algorithmus mit der Laufzeit $T_A(n)$.

## **Problem 6:** Amortisierte Analyse **

Beim Zählen bis $n$ benötigt der Zähler $\lceil \log n \rceil$ Bits.

(a) Im schlimmsten Fall müsseren en wir jedesmal beim Inkrementilog $n$ Bits kippen ($0111\ldots111 \longrightarrow 1000\ldots000$).
Somit gilt für die worst-case-Laufzeit: $T^{\text{worst}}(n) \in O(n \log n)$.

(b) Beim Zählen von 0 bis $n$ kippt das $i$-te Bit genau $\lfloor n/2^{i-1} \rfloor$ mal. Die Ganzheitsmethode ergibt somit für die $n$ Operationen beim Zählen bis $n$:

$$T^{\text{worst}}(n) = \sum_{i=1}^{\lceil \log n \rceil} \left\lfloor \frac{n}{2^{i-1}} \right\rfloor \leq \sum_{i=1}^{\lceil \log n \rceil} \frac{n}{2^{i-1}} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2 \cdot n \in O(n).$$

da das erste Bit jedoch stets $n$ mal kippt gilt auch: $T(n) \geq n$
$\Rightarrow T(n) \in \Theta(n)$

Mit Hilfe der Buchungsmethode und Gebühren 2 pro Operation lässt sich dies sehr ähnlich zeigen.

## **Problem 7:** Small-tree-Modifikation von Weighted-Union-Find ***

Die Laufzeit des Union-Find-Algorithmus wird hauptsätzlich von dem Aufwand der Union- und der Find-Operationen bestimmt. Bei der Small-tree-Modifikation verbessert sich die Laufzeit von Find von $O(\log n)$ zu $O(1)$. Allerdings erhöht sich der Aufwand für Union von $O(1)$ zu $\Omega$(Größe der kleineren Menge), da wir jede Komponente einzeln an die Wurzel hängen. Bei einer beliebigen Folge von $n$ Union- und Find-Operationen ändert sich die asymptotische Laufzeit nicht. Folgendes Beispiel in Abbildung 1 zeigt einen Aufwand von $\Omega(n \log n)$ :

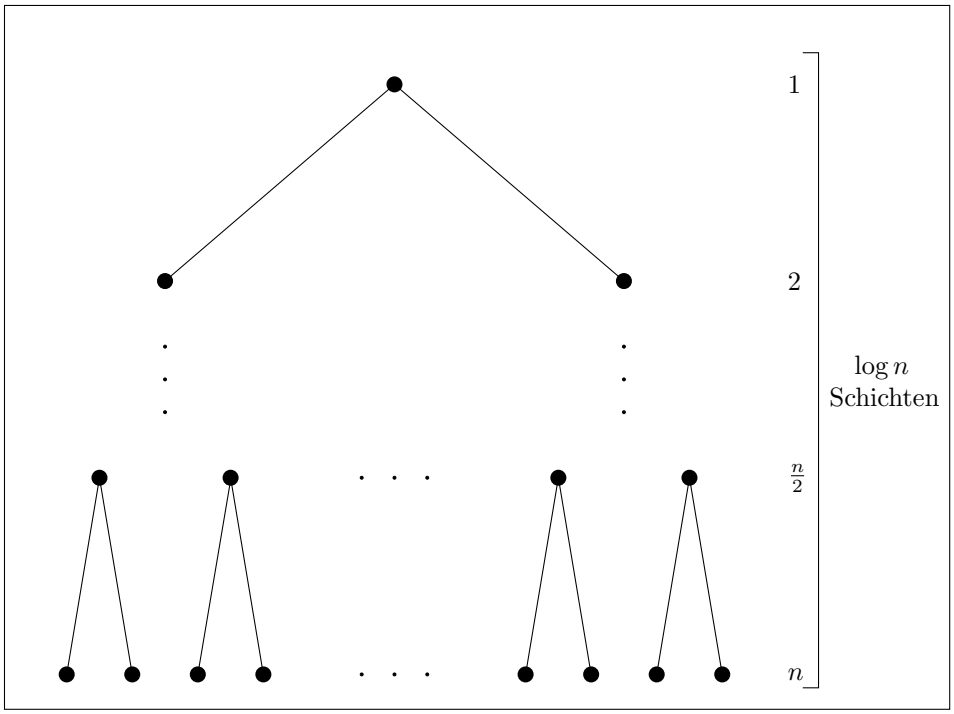



Abbildung 1: Beispiel für Aufwand $\Omega(n \log n)$.

Im ersten Schritt werden $n/2$ UNION-Operationen durchgeführt mit Kosten jeweils 2. Im zweiten Schritt werden dann $n/4$ Operationen mit Kosten jeweils 4 durchgeführt usw. Bei $\log n$ Schichten ergibt sich eine Gesamtlaufzeit von $\Omega(n \log n)$.