# Algorithmen und Datenstrukturen
## *Musterlösung 1*

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

16. März, zur Besprechung am 22. März

---

### Aufgabe 1) Warteschlangen

1. Inhalt der Warteschlange nach jedem Schritt:

   (a) `1`

   (b) `1, 2`

   (c) `1, 2, 3`

   (d) `2, 3`

   (e) `2, 3, 1`

   (f) `3, 1`

   Für die Variablen gilt anschließend `x = 1` und `y = 2`.

2. Java-Implementierung einer Warteschlange mit verketteten Listen:

```java
public class ListQueueSolution implements Queue {
    Node head = null; // head of queue
    Node last = null; // end of queue

    public void enqueue(int n) {
        if (head == null) {
            this.last = new Node(n, this.last);
            this.head = last;
        } else {
            this.last.next = new Node(n, this.last);
            this.last = this.last.next;
        }
    }

    public int dequeue() {
        if (this.head == null) {
            throw new RuntimeException("no element");
        } else {
```

```
                int n = this.head.value;
                this.head = this.head.next;
                if (this.head == null) {
                    this.last = null;
                }
                return n;
            }
        }
}
```

3. Die genaue Anzahl atomarer Operationen hängt von der Implementierung ab (und von der Definition von "atomarer Operation"), aber für obige Implementierung und die Betrachtung von Vergleichen und Zuweisungen als atomaren Operationen hat `enqueue` 3 atomare Operationen und `dequeue` 1, 3 oder 4 Operationen, je nach Ergebnissen der Vergleiche in den `if`-Bedingungen.

**Aufgabe 2) $\mathcal{O}$-Notation**

1. Enthalten in $\mathcal{O}(n^2)$ sind $a(n)$, $b(n)$, $c(n)$, $d(n)$, $e(n)$ und $g(n)$.

   Als Beispiel sie hier ein Beweis dieser Tatsache für $g(n)$ skizziert. Bedingung für $g(n) \in \mathcal{O}(n^2)$ ist:

   $$\exists n_0 \exists c > 0 \forall n \geq n_0 : n \log n \cdot \sqrt{n} \log n \leq cn^2$$

   Äquivalent dazu ist:

   $$\exists n_0 \exists c > 0 \forall n \geq n_0 : \log^2 n \leq c\sqrt{n}$$

   Bedingung für $g(n) \in \mathcal{O}(n^2)$ ist also $\log^2 n \in \mathcal{O}(\sqrt{n})$. Diese Bedingung ist erfüllt, da jede polynomielle Funktion $n^a$ mit $a > 0$ asymptotisch schneller wächst als jede polylogarithmische Funktion $\log^b n$ (Beweis siehe z.B. Cormen et al., "Introduction to Algorithms", Abschnitt 3.2).

2. Die Funktionen können in folgende Reihenfolge nach steigender Komplexität gebracht werden: $d(n)$ (konstant), $a(n)$ (linear), $e(n)$, $g(n)$, $c(n)$ (quadratisch), $b(n)$ (quadratisch), $f(n)$ (kubisch).

3. $\mathcal{O}(y(n)$ ist die Menge aller Funktionen, die höchstens so schnell wie $y(n)$ wachsen (siehe Folie 30 der Vorlesung), d.h. $c \cdot y(n)$ stellt eine asymptotische obere Schranke für die enthaltenen Funktionen dar.

   $\Omega(y(n))$ dagegen enthält die Funktionen, die mindestens so schnell wie $y(n)$ wachsen, d.h. $c \cdot y(n)$ ist eine asymptotische untere Schranke für diese Funktionen. So wächst $f(n) = 2n^3 + 3n^2$ asymptotisch schneller als $n^2$ und ist daher in $\Omega(n^2)$ enthalten, nicht aber in $\mathcal{O}(n^2)$.

   $\Theta(y(n))$ schließlich ist die Menge der Funktionen, die asymptotisch genauso schnell wachsen wie $y(n)$. Beispielsweise ist $0.5n$ in $\mathcal{O}(n^2)$ enthalten, nicht aber in $\Theta(n^2)$, da $n^2$ asymptotisch schneller wächst als $0.5n$.

## Aufgabe 3) Minimum-Maximum-Suche mit Divide & Conquer

1. Für Minimum und Maximum kann ein bestimmtes Element des Arrays, z.B. das erste, als initialer Lösungskandidat verwendet werden. Es muss dann mit den übrigen $n-1$ Elementen verglichen werden. Für Minimum und Maximum zusammen ergibt sich somit eine Gesamtzahl von $2n-2$ Vergleichen.

2. Java-Implementierung der Minimum-Maximum-Suche nach dem Prinzip Divide & Conquer:

```
public class MinMaxSolution {
    public static int[] minmax(int[] array) {
        // Aufruf einer Hilfsfunktion mit Suchbereich als
            zusaetzlichem
        // Eingabeparameter
        return minmaxRecursive(array, 0, array.length - 1);
    }

    private static int[] minmaxRecursive(int[] array, int left, int
        right) {
        int[] result = new int[2];

        if (left + 1 >= right) {
            // Basisfall: Nur noch zwei benachbarte Elemente
            //  => direkter Vergleich
            if (array[left] > array[right]) {
                result[0] = array[right];
                result[1] = array[left];
            } else {
                result[0] = array[left];
                result[1] = array[right];
            }
        } else {
            // Divide: Aufteilen des Arrays in zwei Haelften
            int middle = (left + right) / 2;
            // Rekursives Loesen der zwei Teilprobleme
            int[] resultL = minmaxRecursive(array, left, middle);
            int[] resultR = minmaxRecursive(array, middle + 1, right);
            // Conquer: Zusammenfuegen der Teilergebnisse zur
            // Gesamtloesung
            // Minimum ist das kleinere der beiden Teilminima
            if (resultL[0] < resultR[0]) {
                result[0] = resultL[0];
            } else {
                result[0] = resultR[0];
            }
            // Maximum ist das groessere der beiden Teilmaxima
            if (resultL[1] > resultR[1]) {
                result[1] = resultL[1];
            } else {
                result[1] = resultR[1];
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] array = { 9, 6, 64, 2, 42, 23, 7, 15 };

        System.out.print("Eingabearray: { ");
```

```
        for (int i = 0; i < array.length; i++) {
            if (i > 0) {
                System.out.print(", ");
            }
            System.out.print(array[i]);
        }
        System.out.print(" }\n");

        int[] result = minmax(array);
        System.out.println("Gefundenes Minimum: " + result[0]);
        System.out.println("Gefundenes Maximum: " + result[1]);
    }
}
```

3. Als Basisfall kann der Fall mit einem einzelnen verbliebenen Element gewählt werden. Es ergibt sich dann genau wie beim Standardansatz eine Gesamtzahl von $2n - 2$ Vergleichen (der Beweis bleibt dem Leser überlassen).

   Mit einer kleinen Änderung lässt sich aber eine geringere Zahl an notwendigen Schlüsselvergleichen erreichen: Wenn als Basisfall der Fall mit zwei verbleibenden Elementen gewählt wird, können diese beiden mit einem Schlüsselvergleich direkt miteinander verglichen werden und als Teillösungen für Minimum bzw. Maximum zurückgegeben werden (siehe obige Implementierung). Für zwei Elemente kommt man also mit einem Schlüsselvergleich statt mit zweien aus. Im Rekursionsfall können jeweils die Minima und die Maxima der Teilprobleme miteinander verglichen werden, es finden also 2 Schlüsselvergleiche statt. Damit ergibt sich für Werte $n = 2^k, k \in \mathbb{N}$ die Rekursionsgleichung

$$T(2^k) = \begin{cases} 1 & \text{falls } k \leq 1 \\ 2T(2^{k-1}) + 2 & \text{sonst} \end{cases}$$

   Das Master Theorem liefert hierfür eine asymptotische Zahl an Schlüsselvergleichen von $T(n) = \Theta(n)$. Ein exakteres Ergebnis lässt sich durch Raten & Induktion erhalten:

   **Induktionshypothese** $T(2^k) = 1.5 \cdot 2^k - 2$

   **Induktionsanfang** $T(2^1) = 1.5 \cdot 2^1 - 2 = 3 - 2 = 1$

   **Induktionsschritt**

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2 \\ \text{(IH)} &= 2(1.5 \cdot 2^k - 2) + 2 \\ &= 1.5 \cdot 2^{k+1} - 4 + 2 \\ &= 1.5 \cdot 2^{k+1} - 2 \end{aligned}$$

**Aufgabe 4) Master Theorem**

- – $T_1(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ mit $a = 4$, $b = 2$, $f(n) = n$.
  – Es liegt Fall 1 des Master Theorems vor, da $f(n) = n = \mathcal{O}(n^{log_2(4)-\varepsilon}) = \mathcal{O}(n^{2-\varepsilon})$.
  – Die Lösung ist demzufolge $T_1(n) = \Theta(n^{log_2(4)}) = \Theta(n^2)$.
- – $T_2(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ mit $a = 6$, $b = 3$, $f(n) = n^2$.
  – Es liegt möglicherweise Fall 3 des Master Theorems vor, da $f(n) = n^2 = \Omega(n^{log_3(6)+\varepsilon})$ für $\varepsilon = 0.001$.
  – Zu prüfen ist die Nebenbedingung $\exists c < 1 \exists n_0 \forall n \geq n_0 : a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$:

$$\begin{aligned} & 6\left(\frac{n}{3}\right)^2 \leq cn^2 \\ \Leftrightarrow\ & \frac{6}{9}n^2 \leq cn^2 \\ \Leftrightarrow\ & \frac{6}{9} \leq c < 1 \end{aligned}$$

  – Die Bedingung ist erfüllt, die Lösung ist demzufolge $T_2(n) = \Theta(f(n)) = \Theta(n^2)$.
- – $T_3(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ mit $a = 16$, $b = 4$, $f(n) = n^2 \log(n)$.
  – Da $log_b(a) = 2$ ist, kommt offensichtlich nur Fall 3 des Master Theorems in Frage. Da jedoch $n^2 \log(n) \notin \Omega(n^{2+\varepsilon})$ für irgendein $\varepsilon > 0$ gilt, ist auch Fall 3 nicht anwendbar.
- Das Master Theorem ist auf $T_4(n)$ nicht anwendbar, da das Argument für den rekursiven Aufruf von $T_4(n)$ kein Quotient, sondern eine Differenz ist. Bei näherer Betrachtung lässt sich aber feststellen, dass $T_4(n) = T_4(n-1) + n$ mit $T_4(1) = 1$ eine Rekursionsgleichung für die arithmetische Reihe $\sum_{i=1}^{n} i$ darstellt, für die die Lösung $T_4(n) = \frac{n(n+1)}{2}$ angegeben werden kann. Ein Beweis ist z.B. durch Induktion möglich:

  **Induktionshypothese** $T_4(n) = \frac{n(n+1)}{2}$

  **Induktionsanfang** $T_4(1) = \frac{1(1+1)}{2} = 1$

  **Induktionsschritt**

$$\begin{aligned} T_4(n+1) &= T_4(n) + n + 1 \\ \text{(IH)} &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{n(n+1) + 2n + 2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$