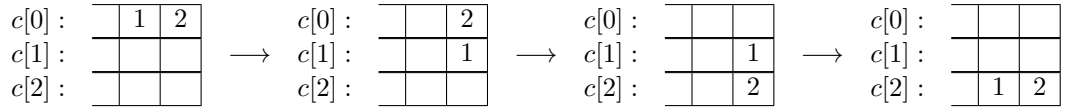
Prof. Dr. E. Ábrahám

F. Corzilius, S. Schupp, T. Ströder

Tutoraufgabe 1 (Algorithmenentwurf):

Entwerfen Sie einen iterativen (also nicht rekursiven) Algorithmus, der ein Array entgegennimmt, das zwei leere und einen gefüllten Container enthält. In den Containern liegen alle Elemente (paarweise verschiedene ganze Zahlen) **stets aufsteigend sortiert** (*) vor. Der Algorithmus soll die Elemente im gefüllten Container vollständig in einen der leeren Container verschieben, wobei alle Container so genutzt werden dürfen, dass (*) nie verletzt wird.
Beispiel (für a)):



Wir setzen voraus, dass der gefüllte Container der erste (`c[0]`) ist und alle Container beliebig viele Elemente enthalten dürfen.

a) Die Datenstruktur der Container ist ein **Stapelspeicher/Stack**, d. h. man darf immer nur ein Element von vorne (also das kleinste) wegnehmen oder ein neues (kleinstes) vorne anfügen. Außerdem kann man nur das oberste Element eines Stacks einsehen und prüfen, ob er leer ist.

b) Die Datenstruktur der Container ist eine **Warteschlange/Queue**, d. h. man darf immer nur ein Element von vorne (also das kleinste) wegnehmen oder ein neues (größtes) hinten anfügen. Außerdem kann man nur das oberste Element einer Queue einsehen und prüfen, ob sie leer ist.

Lösung:

a)
```
void swap(stack[] c)
{
    // Wiederhole Folgendes bis das Ziel erreicht ist:
    // Schiebe das kleinste Element, welches wegen (*) immer das vorderste
    // Element eines der drei Stacks ist, zum naechsten Stack (im Uhrzeigersinn) und
    // verschiebe danach ein anderes Element, wofuer es nur eine Moeglichkeit gibt
    // wenn man (*) beachtet.
    int min = 0; // Position des Stacks, der das kleinste Element enthaelt
    int next = 1; // Position des naechsten Stacks von min aus
    int nextnext = 2; // Position des uebernaechsten Stacks von min aus
    while( true )
    {
        c[next].push(c[min].pop());
        if(c[min].empty() && c[nextnext].empty()) // Ziel erreicht
        {
            return;
        }
        if(!c[min].empty() && (c[nextnext].empty() || c[min].top() < c[nextnext].top()))
        {
            c[nextnext].push(c[min].pop());
        }
        else
        {
            c[min].push(c[nextnext].pop());
        }
        int tmp = min;
        min = next;
        next = nextnext;
        nextnext = tmp;
    }
}
```

b)
```
void swap(queue[] c)
{
    while( !c[0].empty() )
    {
        c[1].push(c[0].pop());
    }
}
```

Aufgabe 2 (Algorithmenentwurf): (2 + 4 = 6 Punkte)

Entwerfen Sie einen iterativen (also nicht rekursiven) Algorithmus mit genau einer Schleife, der ermittelt, ob sich eine gegebene ganze Zahl in einer Reihe (Array) ganzer Zahlen mit Mindestlänge 3 befindet.

a) Die gegebene Reihe ist dabei **unsortiert** und der Algorithmus soll im schlimmsten Fall so viele Schleifen-Iterationen benötigen wie es Elemente in der gegebenen Reihe gibt.

b) Die gegebene Reihe ist dabei **sortiert** und der Algorithmus soll im schlimmsten Fall weniger Schleifen-Iterationen benötigen als es Elemente in der gegebenen Reihe gibt.

Lösung:

a)
```
bool contains(int[] array, int elem)
{
    for(int i = 0; i <array.length(); i++)
    {
        if(elem == array[i])
        {
            return true;
        }
    }
    return false;
}
```

b)
```
bool contains(int[] array, int elem)
{
    int from = 0;
    int to = array.length() - 1;
    while(from <= to) // != ist falsch, da array leer sein kann
    {
        // abrunden (eigentlich schon implizit)
        int center = from + roundDown((to - from)/2);
        if(elem == array[center])
        {
            return true;
        }
        else if(elem > array[center])
        {
            from = center + 1;
        }
        else
        {
            to = center - 1;
        }
    }
    return false;
}
```

Tutoraufgabe 3 (Best- und Worst-Case):

Ein Algorithmus zum effizienten Potenzieren von der Zahl 5 (Annahme: $e \geq 0$):

```
int power5(int e)
{
    int res = 1;
    int tmp = 5;
    while(e > 0)
    {
        if(e.isOdd()) // Prüft, ob e ungerade ist.
        {
            e -= 1;
            res *= tmp;
        }
        e /= 2;
        tmp = tmp^2;
    }
    return res;
}
```

Den Hauptanteil an der Laufzeit für diesen Algorithmus nehmen dabei die nötigen **Multiplikationen** ein. Dabei ist das Quadrieren ebenfalls als Multiplikation zu zählen.

a) Geben Sie die Anzahl der nötigen Multiplikationen für den Parameter $e = 24$ an.

b) Gesucht sind zu obigem Algorithmus zwei Eingaben bestehend aus $0 < e_w < n, n < e_b < \infty$ zu einem beliebigen, aber festen $n \in \mathbb{N}$, welche die **Worst-Case** und die **Best-Case** Laufzeit erzeugen unter der Eingabe der Länge n. Die Eingabe für die Worst-Case Laufzeit soll dabei das kleinste mögliche $e < n$ sein, welches diese erzeugt. Die Eingabe für die Best-Case Laufzeit soll das kleinste mögliche $e > n$ sein, welches diese erzeugt. Erklären Sie bitte kurz (informell), wie Sie zu den angegebenen Lösungen gekommen sind.

Lösung:

a)

e	Binärdarstellung	res	tmp	#Multiplikationen
24	11000	1	5	0
12	1100	1	5^2	1
6	110	1	$(5^2)^2$	2
3	11	1	$((5^2)^2)^2$	3
2	10	$((5^2)^2)^2$	$((5^2)^2)^2$	4
1	1	$((5^2)^2)^2$	$(((5^2)^2)^2)^2$	5
0	0	$((5^2)^2)^2 \cdot (((5^2)^2)^2)^2$	$(((5^2)^2)^2)^2$	6
0	0	$((5^2)^2)^2 \cdot (((5^2)^2)^2)^2$	$((((5^2)^2)^2)^2)^2$	7

b) Worst-Case: $e = \begin{cases} 2^{\lceil log_2 n \rceil} - 1, \text{ falls } n < 2^{\lceil log_2 n \rceil} - \frac{2^{\lceil log_2 n \rceil}}{2} - 1 \\ 2^{\lceil log_2 n \rceil} - \frac{2^{\lceil log_2 n \rceil}}{2} - 1 \text{ sonst} \end{cases}$

Best-Case: $e = \begin{cases} 2^{l(n)+1} \text{ falls } n - 2^{l(n)} \geq 2^{l(n)-1} \\ 2^{l(n)} + 2^{l(n-2^{l(n)})+1} \text{ sonst} \end{cases}$

Die Anzahl der Multiplikationen setzt sich wie folgt zusammen:
$l(e) + p(e)$,
wobei $l(e)$ die Länge des Exponenten in Binärdarstellung sei und $p(e)$ die Anzahl der auf 1 gesetzten Bits.

Die Worst-Case Fallunterscheidung erklärt sich wie folgt:
Wenn die ersten beiden Bits in der Binärdarstellung von n gesetzt sind (Fall b), wird die Zahl, welche alle Bits bis auf die zweite Stelle der Binärdarstellung gesetzt hat als erste Zahl die Worst-Case Laufzeit erzeugen (Als Beispiel: $n = 11001 \rightarrow e = 10111$). Wenn das zweite Bit in der Binärdarstellung von n nicht gesetzt ist (Fall a), wird die kleinste mögliche Zahl, welche alle Bits gesetzt hat die Worst-Case Laufzeit erzeugen (Als Beispiel: $n = 10110 \rightarrow e = 01111$).

Die Best-Case Fallunterscheidung erklärt sich wie folgt:
Wenn die ersten beiden Bits in der Binärdarstellung von n auf 1 gesetzt sind, ist die nächste größere Zahl die nächst größere Zweierpotenz (Als Beispiel: $n = 11010 \rightarrow e = 100000$). Wenn das zweite Bit von der Binärdarstellung von n nicht gesetzt ist, wird die erste 1 nach der führenden 1 um eine Stelle nach links geshiftet und der Rest mit 0 gefüllt (Als Beispiel: $n = 10010 \rightarrow e = 10100$).

Aufgabe 4 (Best- und Worst-Case): (1 + 3 = 4 Punkte)

```
void sort(int[] _array)
{
    bool changed = false;
    do
    {
        changed = false;
        for(int i = 0; i <= _array.size()-2; ++i)
        {
            if(_array[i] > _array[i+1])
            {
                int tmp = _array[i];
                _array[i] = _array[i+1];
                _array[i+1] = tmp;
                changed = true;
            }
        }
```

```
    }
    while(changed)
}
```

a) Führen Sie den Algorithmus (auf Papier) für die Eingabe [1,5,2,4,3] aus.

b) Gesucht sind zu dem gegebenen Algorithmus zwei Eingaben, welche zu einem beliebigen, aber festen n, welches die Größe der Eingabe beschreibt, die Worst-Case und die Best-Case Laufzeit erzeugen. Erklären Sie kurz Ihre Lösung.

Lösung:

a)
15243
12543
12453
12435
12345

b) Es handelt sich bei dem Algorithmus um eine Variante von Bubble-Sort. Eine Eingabe, welche den Best-Case erzeugen würde, wäre ein bereits sortiertes Array. Dabei wird das gegebene Array maximal einmal durchlaufen. Bei einem umgekehrt sortierten Array wird dagegen die innnere Schleife $n-1$ (n sei die Länge der Eingabe) mal durchlaufen (es wird immer ein Element, welches nicht an der richtigen Stelle steht, verschoben).

Tutoraufgabe 5 ($\mathcal{O}$-Notation):

Beweisen oder widerlegen Sie die folgenden Aussagen. Hierbei sind f, g und h positive Funktionen von $\mathbb{N}$ nach $\mathbb{R}^{>0}$.

a) $2^{n+1} \in \Omega(2^n)$

b) $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$

c) $f(n) \in \mathcal{O}(g(n)) \wedge \lim_{n\to\infty} \frac{h(n)}{g(n)} = 0 \implies f(n) + h(n) \in \mathcal{O}(g(n))$

d) $0 \in \Theta(1)$

Lösung:

a) **Behauptung:** Die Aussage gilt.

Beweis:
Wir haben:

$$\begin{aligned} 2^{n+1} \in \Omega(2^n) &\iff \exists c \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : c \cdot 2^n \leq 2^{n+1} \\ &\iff \exists c \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : c \cdot 2^n \leq 2 \cdot 2^n \end{aligned}$$

Wähle also $c = 2$. Damit gilt $c \cdot 2^n = 2 \cdot 2^n$ und die Aussage ist bewiesen. □

b) **Behauptung:** Die Aussage gilt.

Beweis:
Zu zeigen ist:

$$\exists c_1, c_2 \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : c_1 \cdot (f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq c_2 \cdot (f(n) + g(n))$$

Wir beginnen mit $\max\{f(n), g(n)\} \leq c_2 \cdot (f(n) + g(n))$:
Da f und g positive Funktionen sind, gilt, $(f(n) + g(n)) \geq \max\{f(n), g(n)\}$. Aus dieser Abschätzung erhalten wir dann

$$\exists c_2 \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : \max\{f(n), g(n)\} \leq c_2 \cdot (f(n) + g(n))$$

für beliebige $c_2 \geq 1$ und $n \geq n_0$.

Um den zweiten Teil zu zeigen, schätzen wir den Ausdruck $f(n) + g(n)$ nach oben hin ab und erhalten

$$\begin{aligned} & f(n) + g(n) && \leq && 2 \cdot \max\{f(n), g(n)\} \\ \iff & \tfrac{1}{2} \cdot (f(n) + g(n)) && \leq && \max\{f(n), g(n)\} \end{aligned}$$

Hieraus ergibt sich

$$\exists c_1 \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : c_1 \cdot (f(n) + g(n)) \leq \max\{f(n), g(n)\}$$

für $c_1 = \frac{1}{2}$. □

c) **Behauptung:** Die Aussage gilt.

Beweis:
Aus $\lim_{n\to\infty} \frac{h(n)}{g(n)} = 0$ folgt $h(n) \in \mathcal{O}(g(n))$, also

$$\exists c_1 \in \mathbb{R}^{>0}, \exists n_1 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_1 : h(n) \leq c_1 \cdot g(n).$$

Weiterhin bedeutet $f(n) \in \mathcal{O}(g(n))$, dass

$$\exists c_2 \in \mathbb{R}^{>0}, \exists n_2 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_2 : f(n) \leq c_2 \cdot g(n).$$

Damit gilt

$$f(n) + h(n) \leq c_2 \cdot g(n) + c_1 \cdot g(n) = (c_1 + c_2) \cdot g(n)$$

für alle $n \geq \max\{n_1, n_2\}$, woraus $f(n) + h(n) \in \mathcal{O}(g(n))$ mit $c = c_1 + c_2$ und $n_0 = \max\{n_1, n_2\}$ folgt. □

d) **Behauptung:** Die Aussage gilt nicht.

Beweis:
Wir haben $\lim_{n\to\infty} \frac{0}{1} = 0$ und damit auch $\liminf_{n\to\infty} \frac{0}{1} = 0$, aber $0 \in \Omega(1) \iff \liminf_{n\to\infty} \frac{0}{1} > 0$ und $0 \in \Theta(1) \iff 0 \in \Omega(1) \wedge 0 \in \mathcal{O}(1)$. □

Aufgabe 6 ($\mathcal{O}$-Notation): (3 + 3 + 3 + 3 = 12 Punkte)

Beweisen oder widerlegen Sie die folgenden Aussagen. Hierbei sind f, g und h positive Funktionen von $\mathbb{N}$ nach $\mathbb{R}^{>0}$.

a) $n^2 + \log(n) \in \mathcal{O}(\frac{n^5}{n^2+n})$

b) $\log(n) \in \Omega(\sqrt{n})$

c) $f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n)) \implies f(n) \in \mathcal{O}(h(n))$

d) $\log_a(n) \in \Theta(\log_b(n))$ für zwei beliebige Konstanten $a, b > 1$

Lösung:

a) **Behauptung:** Die Aussage gilt.

Beweis:
Es gilt $\lim_{n\to\infty} \frac{n^2+\log(n)}{\frac{n^5}{n^2+n}} = \lim_{n\to\infty} \frac{1+\frac{\log(n)}{n^2}}{\frac{n}{1+\frac{1}{n}}} = 0$. Damit gilt auch $\limsup_{n\to\infty} \frac{n^2+\log(n)}{\frac{n^5}{n^2+n}} = 0$ und die Aussage gilt per Definition. □

b) **Behauptung:** Die Aussage gilt nicht.

Beweis:
Wir haben $\lim_{n\to\infty} \frac{\log(n)}{\sqrt{n}} \overset{\text{L'Hôpital}}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\cdot\sqrt{n}}} = \lim_{n\to\infty} \frac{2\cdot\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$. Damit gilt auch $\liminf_{n\to\infty} \frac{\log(n)}{\sqrt{n}} = 0$, aber $\log(n) \in \Omega(\sqrt{n}) \iff \liminf_{n\to\infty} \frac{\log(n)}{\sqrt{n}} > 0$. □

c) **Behauptung:** Die Aussage gilt.

Beweis:
Angenommen wir haben

$$\exists c_1 \in \mathbb{R}^{>0}, \exists n_1 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_1 : f(n) \leq c_1 \cdot g(n)$$

und

$$\exists c_2 \in \mathbb{R}^{>0}, \exists n_2 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_2 : g(n) \leq c_2 \cdot h(n).$$

Dann gilt

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

für alle $n \geq \max\{n_1, n_2\}$, woraus $f(n) \in \mathcal{O}(h(n))$ mit $c = c_1 \cdot c_2$ und $n_0 = \max\{n_1, n_2\}$ folgt. □

d) Behauptung: Die Aussage gilt.

Beweis:
Zu zeigen ist:

$$\exists c_1, c_2 \in \mathbb{R}^{>0}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N} \text{ mit } n \geq n_0 : c_1 \cdot \log_b(n) \leq \log_a(n) \leq c_2 \cdot \log_b(n)$$

Aus den Logarithmengesetzen folgt $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$. Wähle also $c_1 = c_2 = \frac{1}{\log_b(a)}$. Dies ist möglich, da $a, b > 1$ gilt und damit auch $\frac{1}{\log_b(a)} > 0$. Damit erhalten wir

$$c_1 \cdot \log_b(n) = \log_a(n) = c_2 \cdot \log_b(n)$$

und damit ist die Aussage bewiesen. □