Datenstrukturen und Algorithmen
Musterlösung Präsenzaufgaben*

Aufgabe 13:

- Beweis der Gleichheit der Definitionen von $\Theta(g)$:

$$\Theta_P(g) := \{f \mid \exists c_1, c_2 > 0, \exists n_0 > 0, \forall n > n_0 : \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

$$\Theta_V(g) := \{f \mid \exists c > 0, \exists n_0 > 0, \forall n > n_0 : \ \frac{1}{c} \cdot g(n) \leq f(n) \leq c \cdot g(n)\}$$

  Zu zeigen ist nun, das $\Theta_V(g) \subseteq \Theta_P(g)$ gilt und umgekehrt:

  - $\Theta_V(g) \subseteq \Theta_P(g)$
    Trivial: setze $c_2 = c$ und $c_1 = \frac{1}{c}$
  - $\Theta_V(g) \supseteq \Theta_P(g)$
    Da $0 < c_1 \leq c_2$ gilt, definieren wir:

$$c := max\{\frac{1}{c_1}, c_2\}$$

    Daraus folgt:

$$\frac{1}{c} \leq c_1 \leq c_2 \leq c$$

□

- Beweis der Gleichheit der Definitionen vom $o(g)$:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \quad (f(n) \geq 0, g(n) \geq 0)$$

  Daraus folgt:

$$\begin{aligned} & \forall \epsilon > 0, \exists n_0 > 0, \forall n > n_0 : \ 0 \leq \tfrac{g(n)}{f(n)} < \epsilon \\ \Leftrightarrow \quad & \forall c > 0, \exists n_0 > 0, \forall n > n_0 : \ 0 \leq g(n) < c \cdot f(n) \\ \Leftrightarrow \quad & g \in o(f) \end{aligned}$$

□

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

Aufgabe 14:

Zunächst soll die Behauptung vereinfacht werden, damit sie einfacher zu beweisen ist. Es gilt:

$$\Theta(g) \subseteq \Theta(h) \Longleftrightarrow [f \in \Theta(g) \Rightarrow f \in \Theta(h)]$$

Also gilt für die Behauptung insgesamt:

$$g \in \Theta(h) \Rightarrow [f \in \Theta(g) \Rightarrow f \in \Theta(h)]$$

Nach den Gesetzen der Schaltalgebra läßt sich nun umformen:

$$\begin{aligned} & A \Rightarrow (B \rightarrow C) \\ \Leftrightarrow \quad & \neg A \vee (\neg B \vee C) \\ \Leftrightarrow \quad & (\neg A \vee \neg B) \vee C \\ \Leftrightarrow \quad & \neg(A \vee B) \vee C \\ \Leftrightarrow \quad & A \wedge B \rightarrow C \\ \Leftrightarrow \quad & B \wedge A \rightarrow C \end{aligned}$$

Es gilt also:

$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$$

Im folgenden genügt es also, dieses zu beweisen.

Transitivität:

- für Θ:
  Sei Θ definiert, wie in Aufgabe 13. Die Gleichheit mit der Definition aus der Vorlesung wurde ebenfalls in Aufgabe 13 bereits bewiesen.
  Sei nun:

  $$f \in \Theta(g) \wedge g \in \Theta(h)$$

  und

  $$n \geq max\{n_1, n_2\} =: n_0$$

  Daraus folgt:

  $$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \wedge c_3 \cdot h(n) \leq g(n) \leq c_4 \cdot h(n)$$

  $$\Rightarrow f(n) \leq c_2 \cdot g(n) \leq c_2 \cdot c_4 \cdot h(n) \wedge f(n) \geq c_1 \cdot g(n) \geq c_1 \cdot c_3 \cdot h(n)$$

  Seien nun $c_5 = c_2 \cdot c_4$ und $c_6 = c_1 \cdot c_3$.

  $$c_5 \cdot h(n) \leq f(n) \leq c_6 \cdot h(n)$$

  also gilt:

  $$f \in \Theta(h)$$

  □

- für O:
  Sei O definiert, wie in Aufgabe 13. Sei nun:

  $$f \in O(g) \wedge g \in O(h)$$

  und

  $$n \geq max\{n_1, n_2\} =: n_0$$

Daraus folgt:

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

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

Sei nun $c = c_1 \cdot c_2$. Dann folgt

$$f(n) \leq c \cdot h(n)$$

also gilt:

$$f \in O(h)$$

□

- für Ω:
  Sei Ω definiert, wie in Aufgabe 13. Sei nun:

  $$f \in \Omega(g) \wedge g \in \Omega(h)$$

  und

  $$n \geq max\{n_1, n_2\} =: n_0$$

  Daraus folgt:

  $$f(n) \geq c_1 \cdot g(n) \wedge g(n) \geq c_2 \cdot h(n)$$

  $$\Rightarrow f(n) \geq c_1 \cdot g(n) \geq c_1 \cdot c_2 \cdot h(n)$$

  Sei nun $c = c_1 \cdot c_2$. Dann folgt

  $$f(n) \geq c \cdot h(n)$$

  also gilt:

  $$f \in \Omega(h)$$

  □

- für o:
  Sei o definiert, wie in Aufgabe 13. Sei nun:

  $$f \in o(g) \wedge g \in o(h)$$

  und

  $$n \geq max\{n_1, n_2\} =: n_0$$

  Daraus folgt:

  $$f(n) < c_1 \cdot g(n) \wedge g(n) < c_2 \cdot h(n)$$

  $$\Rightarrow f(n) < c_1 \cdot g(n) < c_1 \cdot c_2 \cdot h(n)$$

  Sei nun $c = c_1 \cdot c_2$. Dann folgt

  $$f(n) < c \cdot h(n)$$

  also gilt:

  $$f \in o(h)$$

  □

- für ω:
  Sei ω definiert, wie in Aufgabe 13. Nach der Umkehrsymmetrie gilt:

  $$f \in \omega(g) \Longleftrightarrow g \in o(f)$$

  Also gilt:

  $$f \in \omega(g) \wedge g \in \omega(h) \Longleftrightarrow g \in o(f) \wedge h \in o(g)$$

  Daraus folgt nach dem Transitivitätsbeweis für o:

  $$h \in o(f)$$

  also gilt wiederum nach Umkehrsymmetrie:

  $$f \in \omega(h)$$

□

Symmetrie:

$$\begin{aligned}
f \in \Theta(g) &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \frac{1}{c} g(n) \leq f(n) \leq c \cdot g(n) \\
&\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \frac{1}{c} g(n) \leq f(n) \wedge f(n) \leq c \cdot g(n) \\
&\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \wedge \frac{1}{c} f(n) \leq g(n) \\
&\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \frac{1}{c} f(n) \leq g(n) \leq c \cdot f(n) \\
&\Leftrightarrow g \in \Theta(f)
\end{aligned}$$

□

Reflexivität:

1.)

$$\begin{aligned}
\Theta(f) &= \{g \mid \exists c_1, c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : \ c_1 \cdot f(n) \leq g(n) \leq c2 \cdot f(n)\} \\
&\supseteq \{g \mid f(n) \leq g(n) \leq f(n), \forall n > 0\} \text{ mit } c_1 = c_2 = 1 \\
&= \{g \mid g(n) = f(n), \forall n > 0\} \\
&= \{f\} \\
&\Rightarrow f \in \Theta(f)
\end{aligned}$$

□

2.)

$$\begin{aligned}
O(f) &= \{g \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ g(n) \leq c \cdot f(n)\} \\
&\supseteq \{g \mid g(n) \leq f(n), \forall n > 0\} \text{ mit } c = 1 \\
&\supseteq \{f\} \\
&\Rightarrow f \in O(f)
\end{aligned}$$

□

3.)

$$\begin{aligned}\Omega(f) &= \{g \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ g(n) \geq c \cdot f(n)\} \\ &\supseteq \{g \mid g(n) \geq f(n), \forall n > 0\} \text{ mit } c = 1 \\ &\supseteq \{f\} \\ &\Rightarrow \ f \in \Omega(f)\end{aligned}$$

□

Umkehrsymmetrie:

1.)

$$\begin{aligned}f \in O(g) &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq f(n) \leq c \cdot g(n) \\ &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq \frac{1}{c} f(n) \leq g(n) \\ &\Leftrightarrow \exists c' > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq c' \cdot f(n) \leq g(n) \\ &\Leftrightarrow g \in \Omega(f)\end{aligned}$$

□

2.)

$$\begin{aligned}f \in o(g) &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq f(n) < c \cdot g(n) \\ &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq \frac{1}{c} f(n) < g(n) \\ &\Leftrightarrow \exists c' > 0, \exists n_0 > 0, \forall n \geq n_0 : \ 0 \leq c' \cdot f(n) < g(n) \\ &\Leftrightarrow g \in \omega(f)\end{aligned}$$

□

Bemerkungen und Gegenbeispiel:

Die gezeigten Eigenschaften deuten an, daß es sich um eine Art Vergleichsrelation handelt:

$$\begin{aligned}f \in O(g) &\approx a \leq b \\ f \in \Omega(g) &\approx a \geq b \\ f \in \Theta(g) &\approx a = b \\ f \in o(g) &\approx a < b \\ f \in \omega(g) &\approx a > b\end{aligned}$$

Während für reelle Zahlen a, b jedoch gilt, daß **genau** eine der drei Aussagen $a < b$, $a > b$ oder $a = b$ wahr ist, gilt dieses für Komplexitäten nicht ! **Es gilt nicht**:

$$f \notin o(g) \wedge f \notin \omega(g) \Rightarrow f \in \Theta(g)$$

Gegenbeispiel: Sei $f(n) = n^{1+sin(n)}$ und $g(n) = n$. Da der Grenzwert

$$\lim_{n \to \infty} \frac{n^{1+sin(n)}}{n} \longrightarrow \infty$$

ist, also nicht existiert, gilt: $f \notin O(g)$ und insbesondere $f \notin o(g)$. Für $f \in \Theta(g)$ gilt, daß $f \in O(g) \wedge f \in \Omega(g)$ ist. Da dieses für $f \in O(g)$ aber schon nicht gilt, gilt es insgesamt auch nicht für $f \in \Theta(g)$. Damit führt das zu einem Widerspruch auf der rechten Seite. Also gilt: $f \notin \Theta(g)$.

Aufgabe 15:

Abschätzen der Laufzeitkomplexität für den Algorithmus:

- $T(0) = c_1$
  (für Prozeduraufruf, if-Schleife und else-RETURN)
- $T(n) = c_2 + n \cdot (c_3 + T(n-1))$
  (c_2 für IF, s:=0, RETURN; der übrige Term schätzt die for-Schleife ab)

Daraus ergibt sich:

$$\begin{aligned} T(1) &= c_2 + c_3 + c_1 \\ T(2) &= c_2 + 2(c_3 + c_2 + c_3 + c_1) = 2c_1 + 3c_2 + 4c_3 \\ T(3) &= 6c_1 + 10c_2 + 15c_3 \\ &\dots \\ T(n) &= f_1(n) \cdot c_1 + f_2(n) \cdot c_2 + f_3(n) \cdot c_3 \end{aligned}$$

Die Funktionen f_1, f_2 und f_3 können wie folgt abgeschätzt werden:

$$\begin{aligned} f_1(n) &= n \cdot f_1(n-1) = n! \\ f_2(n) &= n \cdot f_2(n-1) + 1, \quad f_2(1) = 1 \\ f_3(n) &= n \cdot (f_3(n-1) + 1), \quad f_3(1) = 1 \end{aligned}$$

Behauptung: der Algorithmus hat eine Komplexität von $O((n+1)!)$.
Beweis: Die Behauptung gilt, wenn gilt: $f_1 \in O((n+1)!)$, $f_2 \in O((n+1)!)$ und $f_3 \in O((n+1)!)$.

- $f_1 \in O((n+1)!)$
  Trivial, die Aussage gilt ($n! \leq (n+1)!$)
- $f_2 \in O((n+1)!)$

  $$f_2(n) \leq (n+1)!$$

  - für $n = 1$:

    $$\text{gilt, da } 1 \leq 2$$

  - für $n > 1$:

    $$\begin{aligned} & f_2(n-1) && \leq n! \\ \Rightarrow\ & n \cdot f_2(n-1) + 1 && \leq n \cdot n! + 1 \\ \Rightarrow\ & n \cdot f_2(n-1) + 1 && \leq \frac{n}{n+1} \cdot (n+1)! + 1 \\ \Rightarrow\ & n \cdot f_2(n-1) + 1 && \leq (n+1)! \cdot \left[\frac{n}{n+1} + \frac{1}{(n+1)!}\right] \\ \Rightarrow\ & f_2(n) && \leq (n+1)! \cdot \left[\frac{n}{n+1} + \frac{1}{(n+1)!}\right] \leq (n+1)! \\ \Rightarrow\ & f_2(n) && \leq (n+1)! \end{aligned}$$

    □

- $f_3 \in O((n+1)!)$

  $$f_3(n) \leq (n+1)!$$

  - für $n = 1$:

    $$\text{gilt, da } 1 \leq 2$$

– für $n > 1$:

$$\begin{aligned}
& f_3(n-1) && \leq n! \\
\Rightarrow\ & n \cdot (f_3(n-1)+1) && \leq n \cdot (n!+1) \\
\Rightarrow\ & n \cdot (f_3(n-1)+1) && \leq \frac{n}{n+1} \cdot (n+1)! + n \\
\Rightarrow\ & n \cdot (f_3(n-1)+1) && \leq (n+1)! \cdot \left[\frac{n}{n+1} + \frac{n}{(n+1)!}\right] \\
\Rightarrow\ & f_3(n) && \leq (n+1)! \cdot \left[\frac{n}{n+1} + \frac{n}{(n+1)!}\right] \leq (n+1)! \\
\Rightarrow\ & f_3(n) && \leq (n+1)!
\end{aligned}$$

□

Damit gilt insgesamt:

$$T(n) \in O((n+1)!)$$

Speicherplatzkomplexität des Algorithmus:

Der Algorithmus benutzt nur Speicher für die lokalen Variablen. Dieser Speicherplatz wächst proportional zur Rekursionstiefe:

$$M(n) = c_4 \cdot (n+1)$$

Der gegebene Algorithmus berechnet die Fakultät:

$$f(n) = n \cdot f(n-1),\ f(0) = 1 \Rightarrow f(n) = n!$$

Die Berechnung der Fakultät geht mit folgendem Algorithmus auch einfacher:

```
PROCEDURE f(n:INTEGER) : INTEGER =
VAR s : INTEGER;
BEGIN
  s:=1;
  FOR i:=1 TO n DO s:=s*i; END;
  RETURN s;
END f;
```

Dieser Algorithmus hat nur die Laufzeitkomplexität $T \in O(n)$ und die Speicherplatzkomplexität $M \in O(1)$.

Aufgabe 16:

Hier handelt es sich um eine hübsche Fangfrage... Da die O-Notation auch alle Komplexitäten umfaßt, die langsamer wachsen, sagt die Aussage mit dem Terminus "mindestens" nichts aus. "mindestens" würde schließlich eine untere Schranke bedeuten, während O eine obere Schranke beschreibt.

Aufgabe 17:

Der Beweis ist zweigeteilt:

1.) Die eine Richtung ist logisch:

$$max\{f(n), g(n)\} \leq f(n) + g(n)$$

2.) Zunächst wird der Mittelwert gebildet:

$$\begin{aligned} & max\{f(n), g(n)\} && \geq \frac{1}{2} \cdot (f(n) + g(n)) \\ \Rightarrow \quad & max\{f(n), g(n)\} && \in \{h \mid \frac{1}{2} \cdot (f(n) + g(n)) \overset{\text{logisch}}{\leq} h \overset{\text{siehe 1.}}{\leq} f(n) + g(n),\ \forall n > 1\} \subseteq \Theta(f + g) \end{aligned}$$

Dieses gilt für $c_1 = \frac{1}{2}$, $c_2 = 1$, $n_0 = 1$.

Aus 1.) und 2.) folgt nun insgesamt:

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

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 1. Übung*

Aufgabe 1:

Für große n gilt, daß die Laufzeit (näherungsweise) proportional zur gegebenben Komplexität ist:

$$t_i(n) = c_i \cdot K_i(n)$$

Dabei ist $K_i(n)$ die laut Aufgabenstellung gegebene Komplexität für Algorithmus i. Daraus folgt:

$$c_i = \frac{t_i(n)}{K_i(n)}$$

Außerdem ist $t_i(1000) = 1$ h gemäß Aufgabenstellung.

(a)

Für die schnellere Maschine gilt: ${c_i}' = \frac{1}{2} \cdot c_i$. Also gilt für die jeweiligen Algorithmen:

A:

$$\begin{aligned} & {c_A}' \cdot log_2 n' && = c_A \cdot log_2 n \\ \Rightarrow \quad & log_2 n' && = 2 \cdot log_2 n \\ \Rightarrow \quad & n' && = 2^{2 \cdot log_2 n} = n^2 \\ \Rightarrow \quad & n' && = 10^6 \end{aligned}$$

B: Analog:

$$n' = 2 \cdot n = 2000$$

C: Analog:

$$n'^2 = 2 \cdot n^2 \Rightarrow n' = \sqrt{2} \cdot n \approx 1414$$

D: Analog:

$$2^{n'} = 2 \cdot 2^n = 2^{n+1} \Rightarrow n' = n + 1 = 1001$$

(b)

Es gilt wieder allgemein:

$$t_i(500) = c_i \cdot K_i(500) \text{ mit } c_i = \frac{t_i(1000)}{K_i(1000)} = \frac{1 \text{ h}}{K_i(1000)}$$

Also:

$$t_i(500) = \frac{K_i(500)}{K_i(1000)} \cdot 1 \text{ h}$$

Für die einzelnen Algorithmen ergibt sich dann folgende Zeitkomplexität für $n = 500$:

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

A:

$$t_A(500) = \frac{ln500}{ln1000} \cdot 1 \text{ h} \approx 0,90 \text{ h} \approx 54 \text{ min}$$

B:

$$t_B(500) = \frac{500}{1000} \cdot 1 \text{ h} = 0,5 \text{ h} = 30 \text{ min}$$

C:

$$t_C(500) = \frac{500^2}{1000^2} \cdot 1 \text{ h} = 0,25 \text{ h} = 15 \text{ min}$$

D:

$$t_D(500) = \frac{2^{500}}{2^{1000}} \cdot 1 \text{ h} = 2^{-500} \text{ h} \approx 0 \text{ sec}$$

Bei diesem Beispiel handelt es sich um ein ausgesprochen konstruiertes Beispiel, was eigentlich in keiner Weise der Realität entsprechen kann !

Aufgabe 2:

Zeige für beliebige $f, g : \mathbb{N} \to \mathbb{N}$:

$$g \in O(f) \iff f \in \Omega(g)$$

Es gilt:

$$\begin{aligned} g \in O(f) &\Leftrightarrow \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0 : \ g(n) \leq c \cdot f(n) \\ &\Leftrightarrow \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0 : \ \frac{1}{c} g(n) \leq f(n) \\ &\Leftrightarrow \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0 : \ c' \cdot g(n) \leq f(n) \\ &\Leftrightarrow f \in \Omega(g) \end{aligned}$$

□

Aufgabe 3:

(a)

Für $n \in \mathbb{N} \setminus \{0\}$ gilt

$$\begin{aligned} & \tfrac{g(n)}{f(n)} = \tfrac{4711n^2 - 42n + 5}{n^2} = 4711 - \tfrac{42}{n} + \tfrac{5}{n^2} \\ \Rightarrow \quad & \lim_{n \to \infty} \tfrac{g(n)}{f(n)} = 4711 \\ \Rightarrow \quad & g \notin o(f) \end{aligned}$$

Für $n \in \mathbb{N}$ gilt:

$$\begin{aligned} & 4711n^2 - 42n + 5 \leq 4711n^2 + 5n^2 \leq 4716n^2 \\ \Rightarrow \quad & g \in O(f) \end{aligned}$$

Dieses gilt mit $c = 4716$ und $n_0 = 0$. Außerdem gilt:

$$\begin{aligned} & 4711n^2 - 42n + 5 \geq n^2 \\ \Rightarrow \quad & g \in \Omega(f) \end{aligned}$$

Also folgt insgesamt $g \in \Theta(f)$

□

(b)

Für $n \in \mathbb{N} \setminus \{0\}$ gilt:

$$\begin{aligned} \lceil \sqrt[3]{n} \rceil &\leq \sqrt[3]{n} + 1 \\ &\leq \sqrt{n} + 1 \\ &\leq \sqrt{n} + \sqrt{n} \\ &\leq 2\lceil \sqrt{n} \rceil \end{aligned}$$

Damit gilt $g \in O(f)$ für $c = 2$ und $n_0 = 1$.

Angenommen, $g \in \Omega(f)$, dann gäbe es ein $c > 0$ und ein $n_0 \in \mathbb{N}$, so daß für alle $n > n_0$ gilt:

$$\lceil \sqrt[3]{n} \rceil \geq c \cdot \lceil \sqrt{n} \rceil$$

Dann gilt ebenso (nach oben) für $n > max\{n_0, 1\}$ und dieses $c > 0$:

$$2 \geq \frac{\lceil \sqrt[3]{n} \rceil}{\lceil \sqrt{n} \rceil} \geq c > 0$$

Das ist ein Widerspruch zu:

$$\lim_{n \to \infty} \frac{\lceil \sqrt[3]{n} \rceil}{\lceil \sqrt{n} \rceil} = 0$$

□

(c)

$$g(n) = \sum_{j=0}^{n} (n - j) = \sum_{j=0}^{n} j = \frac{n(n+1)}{2}$$

Also gilt:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n^2 + n}{2n^2} = \frac{1}{2} + \frac{1}{2n} = \frac{1}{2}$$

Damit gilt: $g \notin o(f)$.

Es gilt aber für $n \in \mathbb{N}$:

$$g(n) = \frac{n(n+1)}{2} \leq \frac{n^2 + n^2}{2} = n^2$$

Damit gilt $g \in O(f)$ mit $n_0 = 1$ und $c = 1$.

□

(d)

Für $n \in \mathbb{N} \setminus \{0, 1\}$ gilt:

$$g(n) = \sum_{j=0}^{n} 2^j = \frac{1 - 2^{n+1}}{1 - 2} = 2^{n+1} - 1 \leq 2^{n+1} \overset{n \geq 2}{\leq} 3^n$$

Damit gilt $g \in O(f)$ mit $n_0 = 1$ und $c = 1$.

Für die zweite Aussage ist noch zu zeigen, daß $g \in \Omega(f)$. Also muß für $n > max\{1, n_0\}$ folgendes gelten:

$$3^n \overset{s.o.}{\geq} 2^{n+1} - 1 \geq c \cdot 3^n$$

Also:

$$1 \geq \frac{2^{n+1}-1}{3^n} \geq c > 0$$

Das ist ein Widerspruch zu

$$\lim_{n\to\infty} \frac{2^{n+1}-1}{3^n} = 0$$

Damit gilt: $g \notin \Omega(f)$ und somit $g \notin \Theta(f)$.

□

(e)

Für $n \in \mathbb{N} \setminus \{0,1\}$ gilt:

$$\begin{aligned} g(n) &= \sum_{j=1}^{n} j \cdot 2^j \\ &\leq \sum_{j=1}^{n} n \cdot 2^j \\ &= n \cdot \left(\sum_{j=0}^{n} 2^j \right) - n \\ &= n \cdot (2^{n+1} - 2) \\ &= n \cdot 2(2^n - 1) \leq n \cdot n \cdot 2^n \\ &= n^2 \cdot 2^n \end{aligned}$$

Damit gilt: $g \in O(f)$ mit $n_0 = 1$ und $c = 1$.

□

(f)

Es gilt:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} \overset{L'Hospital}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2} \cdot \frac{1}{\sqrt{n}}} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

Damit gilt auch:

$$\lim_{n\to\infty} \frac{\log n + 1}{\sqrt{n}} = 0$$

Daraus folgt:

$$\lim_{n\to\infty} \frac{\lceil \log n \rceil}{\lceil \sqrt{n} \rceil} = 0$$

Also gilt:

$$\forall \epsilon > 0 \;\; \exists n_0 \in \mathbb{N} \;\; \forall n > n_0 : \frac{\lceil \log n \rceil}{\lceil \sqrt{n} \rceil} < \epsilon$$

Demnach gilt also auch:

$$\exists c > 0 \;\; \exists n_0 \in \mathbb{N} \;\; \forall n > n_0 : \lceil \log n \rceil < c \cdot \lceil \sqrt{n} \rceil$$

Damit gilt $\lceil \log n \rceil \in O(\lceil \sqrt{n} \rceil)$.

□

Aufgabe 4:

Es ergibt sich folgende Abschätzungskette:

$$42 \leq^{(0)} \left[\begin{array}{c} n \\ \text{und} \\ 2^{5+ldn} \end{array} \right]^{(1)} \leq^{(2)} n^2 \leq^{(3)} n^3 \leq^{(4)} (ldn)^{(ldn)} \leq^{(5)} \left(\frac{4}{3}\right)^n \leq^{(6)} n2^n \leq^{(7)} 3^n \leq^{(8)} n! \leq^{(9)} n^n$$

Die Beweise ergebn sich wie folgt:

(0) $\surd$

(1) Es gilt: $2^{ldn} = n^{ld2} = n$. Also gilt: $2^{5+ldn} = 2^5 \cdot n \in \Theta(n)$

(2) $\surd$

(3) $\surd$

(4) Es gilt: $(ldn)^{(ldn)} = n^{ld(ldn)}$. Also ist zu zeigen:

$$\begin{array}{rcl} & n^3 & \leq n^{ld(ldn)} \\ \Leftrightarrow & 3 & \leq ld(ldn) \\ \Leftrightarrow & 2^3 & \leq ldn \\ \Leftrightarrow & 8 & \leq ldn \\ \Leftrightarrow & 2^8 & \leq n \\ \Leftrightarrow & 256 & \leq n \end{array}$$

Damit gilt die Abschätzung für alle $n \geq 256$.

(5) Es gilt:

$$(ldn)^{ldn} = 2^{ld((ldn)^{ldn})} = 2^{ldn \cdot ld(ldn)}$$

Außerdem gilt:

$$\left(\frac{4}{3}\right)^n = 2^{ld(\left(\frac{4}{3}\right)^n)} = 2^{n \cdot ld(\frac{4}{3})}$$

Wegen der Monotonie des Logarithmus gilt dann:

$$ldn \cdot ld(ldn) \leq n \cdot ld(\frac{4}{3})$$

Es ergibt sich:

$$\lim_{n\to\infty} \frac{ldn \cdot ld(ldn)}{n \cdot ld(\frac{4}{3})} \overset{L'Hospital}{=} \lim_{n\to\infty} \frac{\frac{1}{n} ld(ldn) + ldn \cdot \left[\frac{1}{ldn} \cdot \frac{1}{n}\right]}{ld(\frac{4}{3})} = \lim_{n\to\infty} \frac{\frac{ld(ldn)}{n} + \frac{1}{n}}{ld(\frac{4}{3})} = 0$$

Damit ist die Richtigkeit der Abschätzung gezeigt.

(6) wegen $\left(\frac{4}{3}\right)^n \leq 2^n$ gilt insbesondere $\left(\frac{4}{3}\right)^n \leq n2^n$

(7) Es gilt: $n2^n = 2^{n+ldn}$ und $3^n = 2^{3^n} = 2^{nld3}$. Also gilt:

$$2^{n+ldn} \leq 2^{nld3} \Leftrightarrow n + ldn \leq n \cdot ld3$$

Folglich muß gelten:

$$\lim_{n\to\infty} \frac{n + ldn}{n \cdot ld3} \leq 1$$

Es ergibt sich:

$$\lim_{n\to\infty} \frac{n + ldn}{n \cdot ld3} \overset{l'Hospital}{=} \lim_{n\to\infty} \frac{1 + \frac{1}{n}}{ld3} = \frac{1}{ld3} < 1$$

Also folgt die Richtigkeit der Abschätzung.

(8) Beweis per Induktion:
$n = 7$:

$$\frac{3^7}{7!} = \frac{2187}{5040} < 1 \text{ also } \surd$$

$n \mapsto (n+1)$:

$$\frac{3^{(n+1)}}{(n+1)!} = \frac{3 \cdot 3^n}{(n+1) \cdot n!} = \frac{3}{n+1} \cdot \frac{3^n}{n!} < 1$$

Dieses gilt, da für $n \geq 7$ gilt:

$$\frac{3}{n+1} < 1$$

Außerdem gilt nach Induktionsvorraussetzung:

$$\frac{3^n}{n!} < 1$$

Damit gilt die Abschätzung.

(9) $\surd$

Aufgabe 5:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 6:

(a) und (b)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

(c)

Aufstellen der Rekursionsgleichung für die "Türme von Hanoi":
Es gilt offenbar als Anfangsbedingung:

$$H(0) = 0 \text{ und } H(1) = 1$$

Als Rekursionsgleichung ergibt sich aus dem Algorithmus:

$$\begin{aligned}
H(N) &= H(N-1) + 1 + H(N-1) \\
&= 2 \cdot H(N-1) + 1 \\
&= 2 \cdot [2 \cdot H(N-2) + 1] + 1 \\
&= 2 \cdot [2 \cdot H(N-2)] + 2 + 1 \\
&= 2 \cdot [2 \cdot \{2 \cdot H(N-3) + 1\}] + 2 + 1 \\
&= 2 \cdot [2 \cdot \{2 \cdot H(N-3)\}] + 4 + 2 + 1 \\
&= \ldots \\
&= 2^N \cdot H(0) + \sum_{i=0}^{N-1} 2^i
\end{aligned}$$

$$
\begin{aligned}
&= 0 + \frac{1 - 2^N}{1 - 2} \text{ (geometr. Reihe)} \\
&= \frac{2^N - 1}{1} \\
&= 2^N - 1
\end{aligned}
$$

Dadurch ergibt sich für die geforderten Problemgrößen:

$$
\begin{aligned}
H(3) &= 2^3 - 1 = 7 \\
H(7) &= 2^7 - 1 = 127 \\
H(10) &= 2^{10} - 1 = 1023
\end{aligned}
$$

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 2. Übung*

Aufgabe 7:

Die Behauptung, daß für Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ gilt:

$$f \in O(g) \wedge g \notin O(f) \quad \Rightarrow \quad f \in o(g)$$

gilt im Allgemeinen **nicht** !!!
Widerlegenung mit einem Gegenbeispiel:
Sei $f(n) := 1$ und $g(n) := n^{1+sin(n)}$. Dann gilt offensichtlich $f \in O(g)$ und $g \notin O(f)$, denn g wächst unbeschränkt. Nach Definition ist

$$f \in o(g) \quad \Longleftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Da der Grenzwert in unserem Beispiel aber nicht definiert ist (weil $sin(n)$ oszilliert, was man mit einem Plot der Funktion einmal verdeutlichen kann), folgt insbesondere, daß $f \notin o(g)$. Damit ist die Behauptung widerlegt !

□

Bemerkung 1:

Die Ordnung $\leq \subseteq \ (\mathbb{N} \to \mathbb{R}^+) \times (\mathbb{N} \to \mathbb{R}^+)$ ist eine partielle Ordnung, d.h. sie ist reflexiv, transitiv und antisymmetrisch:

1. Reflexivität: $\forall\, f \in \mathbb{N} \to \mathbb{R}^+ :\ f \leq f$
2. Transitivität: $\forall\, f, g, h \in \mathbb{N} \to \mathbb{R}^+ :\ f \leq g \ \wedge\ g \leq h \implies f \leq h$
3. Antisymmetrie: $\forall\, f, g \in \mathbb{N} \to \mathbb{R}^+ :\ f \leq g \ \wedge\ g \leq f \implies f = g$

Die Ordnung ist jedoch nicht total, d.h. je zwei beliebige Elemente $f, g \in \mathbb{N} \to \mathbb{R}^+$ lassen sich nicht notwendigerweise bzgl. $\leq$ anordnen. (Für eine totale Ordnung $\leq$ über einer nicht-leeren Menge M und zwei beliebige Elemente $a, b \in M$ würde dagegen stets eine der folgenden drei Relationen gelten: $a < b$ oder $a = b$ oder $a > b$.) Ist eine Ordnung $\leq$ auf einer Menge M nicht total, so darf man insbesondere aus $a \not\leq b$ **nicht** folgern, daß dann $a > b$ gilt.

Bemerkung 2:

Sei M eine nicht-leere Menge und $\leq \subseteq M \times M$. Dann heißt die Relation $<$ auch der strikte Anteil der Relation $\leq$ über der Menge M. Formal definiert man dabei $<$ wie folgt:

$$a < b \iff a \leq b \ \wedge\ a \neq b$$

In entsprechender Weise werden die Relationen $\geq$ und $>$ definiert.

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

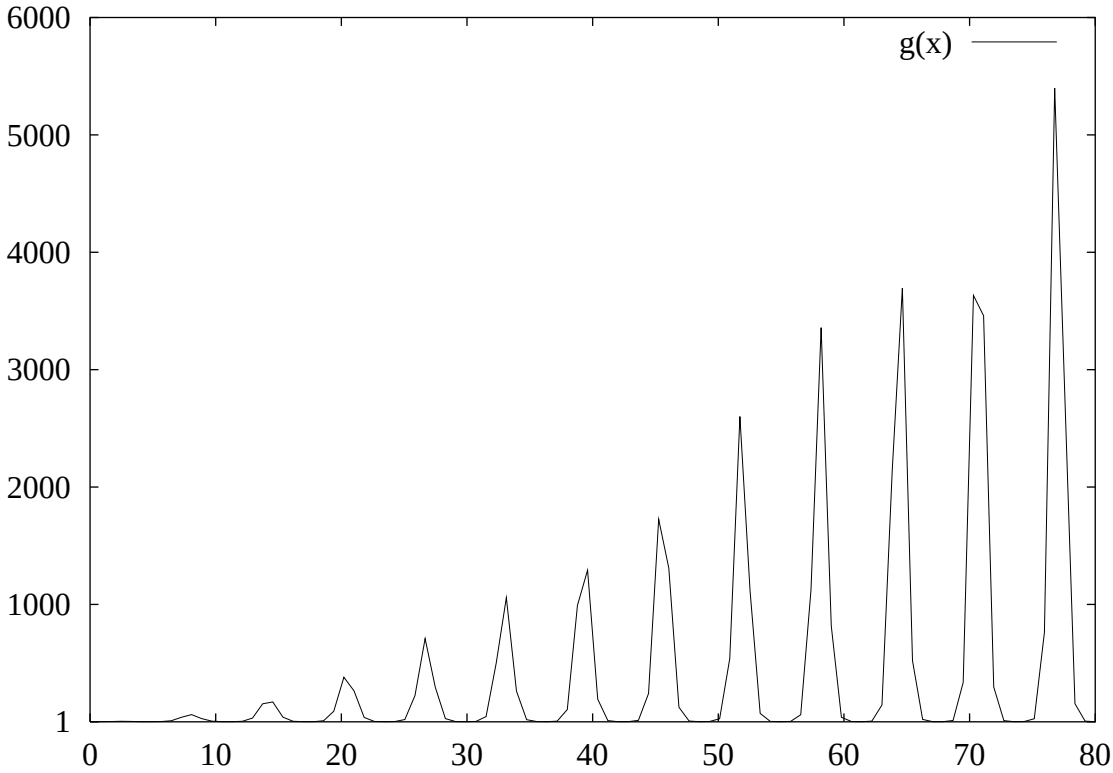


Abbildung 1: Plot von $f(n)$ und $g(n)$ aus Aufgabe 7

Aufgabe 8:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 9:

1.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Die Write-Prozedur entspricht dabei dem Inorder-Tiefendurchlauf.

2.)

Laufzeitkomplexitäten der Prozeduren: Sei h die Höhe des Baumes. Dann ist

- $T_{Insert} \in O(h)$
- $T_{Search} \in O(h)$
- $T_{Write} \in O(n)$

Nun ergibt sich für den

- besten Fall (best-case):
  Der Baum ist vollständig gefüllt und ausbalanciert:

  $$h = \lceil ld(n+1) \rceil - 1$$

  Daraus folgt:

  $$T_{Insert}, T_{Search} \in O(ldn)$$

- schlimmsten Fall (worst-case):
  Der Baum ist zur Liste entgeartet, also $h = n$. Daraus folgt:

  $$T_{Insert}, T_{Search} \in O(n)$$

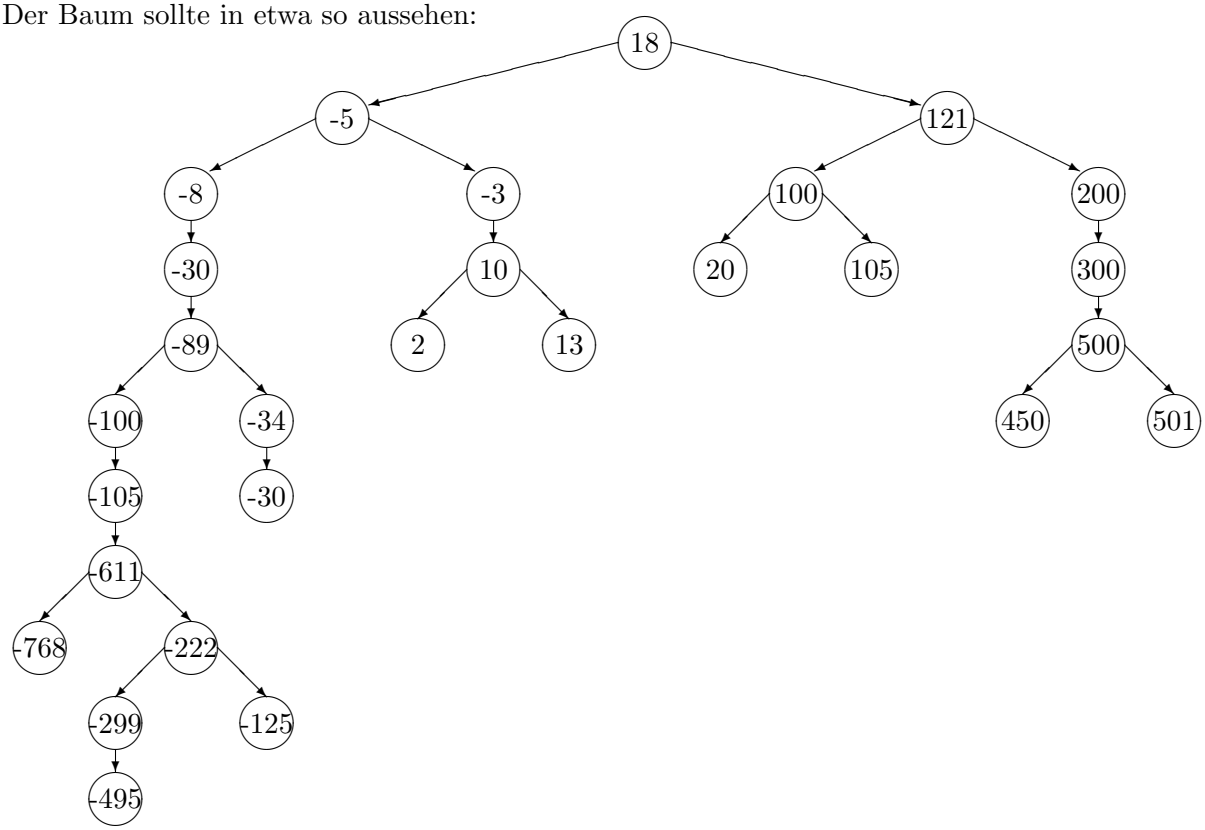
- durchschnittleichewn Fall (average-case):
  Auch hier ist $h \sim ldn$. Also folgt:

  $$T_{Insert}, T_{Search} \in O(ldn)$$

  Der Beweis folgt, wenn in der Vorlesung "Quick-Search" behandelt wurde.

3.)

Der Baum sollte in etwa so aussehen:



Laufzeitkomplexität des Suchverfahrens: Die Laufzeitkomplexität setzt sich zusammen aus der Komplexität zum Einfügen aller Elemente in den Baum und der Komplexität der Write-Prozedur. Also gilt:

$$T(n) = \sum_{i=0}^{n} T_{Insert}(i) \ + \ T_{Write}(n) \leq n \cdot T_{Insert}(n) + T_{Write}(n)$$

Damit ergibt sich

- für den schlimmsten Fall: $T \in O(n^2)$
- für den besten Fall: $T \in O(n \cdot ldn)$

Aufgabe 10:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 11:

1.)

Rekursiver Algorithmus zur Berechnung der Binomialkoeffizienten:

```
PROCEDURE BinKoeff(n, m : INTEGER) :  INTEGER =
BEGIN
  IF m = 0 THEN
    RETURN 1;
  ELSIF n = m THEN
    RETURN 1;
  ELSE
    RETURN BinKoeff(n-1, m-1) + BinKoeff(n-1, m);
  END;
END BinKoeff;
```

2.)

Bei der iterativen Implementierung gibt es 2 Möglichkeiten mit unterscheidlicher Speicherkomplexität.

- 1.Variante:

```
VAR a : ARRAY[0..Max] OF ARRAY [0..Max] OF INTEGER;

PROCEDURE BinKoeff(n, m : INTEGER) : INTEGER =
VAR i,j : INTEGER;
BEGIN
  FOR i := 0 TO n DO
    FOR j := 0 TO i DO
      IF j = 0 THEN
        a[i][j] := 1;
      ELSIF i = j THEN
        a[i][j] := 1;
      ELSE
        a[i][j] := a[i-1][j-1] + a[i-1][j];
      END;
    END;
  END;
  RETURN a[n][m];
END BinKoeff;
```

- 2.Variante
  Diese Variante ist schwerer zu lesen, hat dafür aber die bessere Speicherkomplexität.

```
VAR a : ARRAY[0..Max] OF INTEGER;
```

```
PROCEDURE BinKoeff(n, m : INTEGER) : INTEGER =
VAR i,j : INTEGER;
BEGIN
  FOR i := 0 TO n DO
    FOR j := i TO 0 BY -1 DO
      IF j = 0 THEN
        a[j] := 1;
      ELSIF i = j THEN
        a[j] := 1;
      ELSE
        a[j] := a[j-1] + a[j];
      END;
    END;
  END;
  RETURN a[m];
END BinKoeff;
```

3.)

Laufzeitkomplexität der Prozeduren:

1.) rekursiver Algorithmus:
Es gilt für den Aufruf, IF und RETURN:

$$T(n,0) = c_1$$

Ferner gilt für den Aufruf, 2·IF und RETURN:

$$T(n,n) = c_2$$

Nun gilt noch für $0 < m < n$:

$$T(n,m) = c_3 + T(n-1,m-1) + T(n-1,m)$$

Dabei bezeichne c_3 den Aufruf, 2·IF, "+" und RETURN. Der allgemeine Ansatz lautet nun:

$$T(n,m) = f_1(n,m) \cdot c_1 + f_2(n,m) \cdot c_2 + f_3(n,m) \cdot c_3$$

Wir setzen nun ein:

$$\begin{aligned} f_1(n,m) &= f_1(n-1,m-1) + f_1(n-1,m) \\ f_1(n,0) &= 1 \\ f_1(n,n) &= 0 \end{aligned}$$

Die Behauptung ist:

$$f_1(n,m) \leq 2^n$$

Beweis:

- $n = 0$:
  Da $n \geq m$ mit $n, m \in \mathbb{N}$, folgt daraus auch $m = 0$. Also:

$$1 \leq 2^0 = 1 \surd$$

- $n = 1$:
  D.h. $1 \geq m \geq 0$, also:

$$f_1(1,0) = 1 \leq 2 \text{ und } f_1(1,1) = 0 \leq 2 \surd$$

– $n > 1$:

$$\begin{array}{lcl} & f_1(n-1,m) & \leq 2^n \\ \Rightarrow & f_1(n-1,m) + f_1(n-1,m-1) & \leq 2^n + 2^n = 2^{n+1} \\ \Leftrightarrow & f_1(n,m) & \leq 2^{n+1} \end{array}$$

Also gilt:

$$f_1 \in O(2^n)$$

Analog führt dieser Beweis mit $f_2(n,n) = 1$ und $f_2(n,0) = 0$ zum Ergebnis:

$$f_2 \in O(2^n)$$

Nun für $f_3(n,m)$:

$$\begin{array}{rcl} f_3(n,m) & = & f_1(n-1,m-1) + f_1(n-1,m) + 1 \\ f_3(n,0) & = & 0 \\ f_3(n,n) & = & 0 \end{array}$$

Die Behauptung ist:

$$f_3(n,m) \leq 2^n - 1$$

Beweis:

– $n = 0$:
Da $n \geq m$ mit $n, m \in \mathbb{N}$, folgt daraus auch $m = 0$. Also:

$$0 \leq 2^0 - 1 = 0 \surd$$

– $n = 1$:
D.h. $1 \geq m \geq 0$, also gilt für beide Fälle:

$$0 \leq 2^1 - 1 = 1 \surd$$

– $n > 1$:

$$\begin{array}{lcl} & f_3(n-1,m) & \leq 2^{n-1} - 1 \\ \Rightarrow & f_3(n-1,m) + f_1(n-1,m-1) + 1 & \leq 2 \cdot (2^{n-1} - 1) + 1 \\ \Leftrightarrow & f_3(n,m) & \leq 2^n - 1 \end{array}$$

Also gilt:

$$f_3 \in O(2^n)$$

Damit gilt für den gesamten Algorithmus:

$$T \in O(2^n)$$

□

2.) iterativer Algorithmus:

$$\begin{array}{rcl} T(0) & = & c_1 + c_2 \\ T(1) & = & c_1 + c_2 + c_3 \\ T(2) & = & c_1 + c_2 + c_3 + c_4 \\ T(3) & = & c_1 + c_2 + c_3 + 2 \cdot c_4 \\ & & \dots \\ T(n) & = & c_1 + c_2 + c_3 + \left(\sum_{i=2}^{n} i - 1 \right) \cdot c_4 \end{array}$$

mit c_1 für den äußeren Schleifendurchlauf, c_2 für $m = 0$, c_3 für $m = n$ und c_4 für $0 < m < n$. Es gilt:

$$\sum_{i=2}^{n} i - 1 = \frac{1}{2}(n-1) \cdot n$$

Also folgt daraus:

$$T \in O(n^2)$$

Speicherplatzkomplexität der Prozeduren:

1.) rekursiver Algorithmus:
Es wird nur der Speicher für die Funktionsparameter benötigt, dieser aber wächst proportional zur Rekursionstiefe. Also gilt:

$$M(n, m) = c_4 \cdot (n + 1) \Rightarrow M \in O(n)$$

2.) iterativer Algorithmus:

- 1. Variante:
$$M = c_5 \cdot (n + 1)^2 \Rightarrow O(n^2)$$
- 2.Variante:
$$M = c_6 \cdot (n + 1) \Rightarrow O(n)$$

Aufgabe 12:

1.)

Es gibt folgende Arrays:

n	Arrays der Länge 1:	$[1], [2], \ldots, [n]$
$n-1$	Arrays der Länge 2:	$[1..2], [2..3], \ldots, [(n-1)..n]$
	$\ldots$	
2	Arrays der Länge $n-1$:	$[1..(n-1)], [2..n]$
1	Arrays der Länge n:	$[1..n]$

Also ergibt sich für die Anzahl der Teilfolgen nach der Gauss-Summe unter Addition des leeren Arrays:

$$A = \frac{n(n+1)}{2}$$

2.)

Da jedes Element als Element der maximalen Teilfolge in Frage kommt und demnach untersucht werden muß, ist der Aufwand mindestens linear in der Anzahl der Elemente, also:

$$T \in \Omega(n)$$

Da nur konstanter Speicherplatz benötigt wird (die Eingabe der Folge wird dabei nicht mitgerechnet), gilt für die Speicherplatzkomplexität:

$$M \in \Omega(1)$$

3.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Der Algorithmus funktioniert wie folgt:
Die gesamte Folge wird von vorne bis hinten durchlaufen und führt dabei folgende Schritte durch:

1. solange Werte positiv sind werden diese aufsummiert
2. ist das bisherige Maximum kleiner, als die in Schritt 1 ermittelte Summe, so setze das Maximum auf diese Summe
3. solange nun negative Werte folgen, werden diese aufsummiert
4. ist die Summe der negative und positiven Teilfolge positiv wird diese Summe als neue positive Summe gesetzt und direkt bei Schritt 1 fortgefahren, da durch eine nun folgende positive Teilfolge der maximale Wert ggf. noch gesteigert werden kann. Andernfalls macht es keinen Sinn, die negativen Werte in die maximale Teilfolge aufzunehmen, weil dieses nur zu einer Verringerung des Wertes führt; die positive Summe wird wieder auf 0 gesetzt und mit Schritt 1 versucht, eine neue Teilfolge zu finden, die eventuell größer als das Maximum ist.

Ist die Folge bis zum Ende durchlaufen, so steht der Wert der maximalen Teilfolge fest.

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 3. Übung*

Aufgabe 18:

1.)

Für $n \to \infty$ spielt der Summand 28 quasi keine Rolle mehr für die Größenabschätzung für $T(n)$. Also können wir auch betrachten:

$$T'(n) = 3T'(\lfloor \frac{n}{3} \rfloor) + n$$

Dann gilt: $a = 3$, $b = 3$ und $f(n) = n$

$$n^{log_b a} = n^{log_3 3} = n^1 = n$$

Daraus folgt:

$$f(n) = \Theta(n) = \Theta(n^{log_b a})$$

Also greift der 2. Punkt des Master-Theorems. Damit gilt:

$$T(n) = \Theta(n \cdot log_3 n)$$

2.)

Es gilt: $a = 1$, $b = \frac{3}{2}$ und $f(n) = 1$

$$n^{log_b a} = n^{log_{\frac{3}{2}} 1} = n^0 = 1$$

Daraus folgt:

$$f(n) = \Theta(1) = \Theta(n^{log_b a})$$

Also greift der 2. Punkt des Master-Theorems. Damit gilt:

$$T(n) = \Theta(log_{\frac{3}{2}} n)$$

3.)

Es gilt: $a = 3$, $b = 4$ und $f(n) = n \cdot logn$. Da $0 < log_4 3 < 1$, wähle $\epsilon = 1 - log_4 3 > 0$. Daraus folgt:

$$n^{log_b a + \epsilon} = n^{log_4 3 + \epsilon} = n^1 = n$$

Damit gilt:

$$f(n) = \Omega(n) \text{ , da } n < n \cdot logn$$

Da ferner gilt

$$\frac{3}{4} n \cdot log(\frac{1}{4} n) \leq c \cdot n \cdot logn \text{ mit } c = \frac{3}{4} < 1$$

gilt damit der 3. Fall des Master-Theorems. Also gilt:

$$T(n) = \Theta(f(n)) = \Theta(n \cdot logn)$$

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

4.)

Es wird $t := \log_3 n$ substituiert. Dann gilt wegen $n = 3^t$:

$$\sqrt[3]{3^t} = 3^{t/3}$$

$T(n)$ läßt sich nun wie folgt schreiben:

$$T(n) = T(3^t) = 3T(3^{t/3}) + t$$

Sei nun $S(t) = T(3^t)$, dann folgt:

$$S(t) = T(3^t) = 3 \cdot S\left(\frac{t}{3}\right) + t$$

Das läßt sich nach dem 2. Fall des Mastertheorems auflösen. Daraus folgt dann:

$$S(t) \in \Theta(t \log_3 t)$$

Damit folgt (mit Resubstitution):

$$T(n) = T(3^t) = S(t) = \Theta(t \cdot log_3 t) \overset{t=log_3 n}{=} \Theta(\log_3(n) \cdot \log_3(\log_3 n))$$

Aufgabe 19:

1.)

Es gilt für die Rekursionsgleichung:

$$T(0) = \alpha, T(1) = \beta$$

$$T(n) = (1 + T(n-1))/T(n-2)$$

Zunächst berechnen wir einmal ein paar Werte:

$$\begin{aligned}
T(1) &= \beta \\
T(2) &= \frac{1 + T(1)}{T(0)} = \frac{1+\beta}{\alpha} \\
T(3) &= \frac{\frac{\alpha+(1+\beta)}{\alpha}}{\beta} = \frac{1+\alpha+\beta}{\alpha\beta} \\
T(4) &= \frac{1+\alpha+\beta+\alpha\beta}{(1+\beta)\cdot\beta} = \frac{1+\alpha}{\beta} \\
T(5) &= \frac{(1+\alpha+\beta)\cdot(\alpha\beta)^{-1}}{\beta(1+\alpha+\beta)} = \alpha \\
T(6) &= \frac{1+\alpha}{(1+\alpha)\cdot\beta^{-1}} = \beta
\end{aligned}$$

Offensichtlich ist wegen $T(5) = T(0)$ und $T(6) = T(1)$ die gesuchte Rekursionsgleichung $T(n)$ periodisch in n mit $T(n) = T(n + 5k)$ für $k \in \mathbb{N}$. Die Lösung lautet daher:

$$T(n) = \begin{cases} \alpha & n = 5k \\ \beta & n = 5k+1 \\ \frac{1+\beta}{\alpha} & n = 5k+2 \\ \frac{1+\alpha\beta}{\alpha\beta} & n = 5k+3 \\ \frac{1+\alpha}{\beta} & n = 5k+4 \end{cases}$$

2.)

Hier gibt es zwei Lösungsmöglichkeiten, die im folgenden beide vorgestellt werden sollen.

- Zunächst setzt man sukzessive ein:

$$\begin{aligned}
T(n) &= 2^{-1} \cdot nT(n-1) + 3 \cdot 2^{-1} \cdot n! \\
&= 2^{-1} \cdot n \left(2^{-1} \cdot (n-1) \cdot T(n-2) + 3(n-1)! \cdot 2^{-1}\right) + 2^{-1} \cdot 3n! \\
&= 2^{-2} \cdot n(n-1) \cdot T(n-2) + 2^{-2} \cdot 3 \cdot (n-1)!\, n + 2^{-1} \cdot 3 \cdot n! \\
&\vdots \\
&= 2^{-(n-1)} \cdot n(n-1) \cdot \ldots \cdot T(1) + \\
&\quad 2^{-(n-1)} \cdot 3n! + 2^{-(n-2)} \cdot 3n! + \ldots + 2^{-1} \cdot 3n! \\
&= 2^{-(n-1)} \cdot n! \cdot 4 + 3n! \cdot (2^{-1} + \ldots + 2^{-(n-1)}) \text{ mit } 4 = \mathrm{T}(1) \\
&= \frac{4n!}{2^{n-1}} + 3n! \cdot \frac{2^{n-1} - 1}{2^{n-1}} \\
&= \frac{3 \cdot 2^{n-1} + 1}{2^{n-1}} \cdot n!
\end{aligned}$$

  Wir erhalten somit als Lösung:

$$T(n) = \frac{3 \cdot 2^{n-1} + 1}{2^{n-1}} \cdot n! = 3n! + \frac{n!}{2^{n-1}}$$

- Zu dem gleichen Resultat gelangt man auch mittels Substitution.

$$\begin{aligned}
2T(n) &= n \cdot T(n-1) + 3n! \qquad \Big| \cdot \frac{2^{n-1}}{n!} \\
\frac{2^n \cdot T(n)}{n!} &= \frac{2^{n-1}}{(n-1)!} \cdot T(n-1) + 3 \cdot 2^{n-1} \\
S(n) &:= 2^n \cdot T(n)/n! \\
S(n) &= S(n-1) + 3 \cdot 2^{n-1} \\
S(n) &= \ldots = T(0) + \sum_{i=1}^{n} 3 \cdot 2^{i-1} \\
&= 5 + 3 \cdot 2^n - 3 \\
&= 2 + 3 \cdot 2^n \\
T(n) &= \frac{n!}{2^n} \cdot (3 \cdot 2^n + 2) = 3n! + \frac{n!}{2^{n-1}}
\end{aligned}$$

Aufgabe 20:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Es gibt insgesamt für $N = 5$ 10 Möglichkeiten und für $N = 8$ 92 Möglichkeiten, die Damen auf dem Bord zu platzieren.

Das Prinzip des Algorithmus ist recht einleuchtend: man betrachtet das Board und läuft spaltenweise von der ersten Spalte an durch. Für die aktuelle Spalte wird nun zeilenweise das Board durchlaufen und in jeder Zeile geprüft, ob hier eine Dame sicher platziert werden kann, d.h. in der Zeile steht noch keine Dame und das aktuelle Feld ist auf keiner Diagonalen einer anderen Dame. Dann wird die Dame gesetzt und in der nächsten Spalte fortgefahren, da in derselben Spalte keine weitere Dame stehen darf. Können keine N Damen auf dem Feld platziert werden, gint das Programm nichts aus, anderfalls gibt es alle Möglichkeiten aus, die Damen auf dem Brett zu platzieren.

Aufgabe 21:

(1.)

Greedy berechnet für jedes Exponat den Quotienten $\frac{Wert}{Gewicht}$ und packt als erstes die "Venus von Milo" ein, da sie den höchsten Quotienten 6 aufweist. Anschließend ist im Rucksack kein Platz mehr für weitere Gegenstände. Also erzielt Greedy einen Gesamtwert von 150 TDM.

(2.), (4.) und (5.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Anmerkung: Es wurde nicht verlangt, den Inhalt der Zusammenstellung anzugeben. Für die volle Punktzahl reichte es, jeweils ein Programm, das nur den Wert der Zusammenstellung bestimmt, zu schreiben. Es ergibt sich jeweils ein Gesamtgewicht von 30 kg mit einem Wert von 176 TDM.

(3.)

$$W(i,g) = max\{W(i-1,g), W(i-1,g-g_i) + w_i\}$$

Dabei ist

- g_i das Gewicht des Exponats i,
- w_i der Wert des Exponats i.

Außerdem gilt:

- $W(i-1,g) \stackrel{\wedge}{=}$ Exponat i gehört nicht zur besten Zusammenstellung,
- $W(i-1,g-g_i) + w_i \stackrel{\wedge}{=}$ Exponat i gehört zur besten Zusammenstellung.

Falls $g < g_i$ ist, heißt das, daß Exponat i nicht mehr in den Rucksack paßt und somit gilt: $W(i,g) = W(i-1,g)$.

(6.)

Sei im folgenden N die Zahl der Gegenstände und G das zulässige Gesamtgewicht.

- **Backtracking**
  In jeder Instanz der Rekursion entsteht ein Aufwand $O(N)$. Im schlimmsten Fall ist die Rekursionstiefe gleich N. Also gilt für den worst-case: $T \in O(n^n)$. Das ist hier allerdings nicht ganz so "schlimm", da nicht mehr als 4 Gegenstände in den Rucksack passen, also die Rekusionstiefe kleiner oder gleich 5 ist.
- **NoMemoization**
  $$T \in O(G \cdot 2^N)$$
- **Dynamic Programming**
  $$T \in O(N \cdot G)$$
- **Memoization**
  $$T \in O(N \cdot G)$$

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 4. Übung*

Aufgabe 22:

1.)

Für 4 Teilprobleme der Größe $\frac{N}{2}$ ergibt sich folgende Rekursionsgleichung:

$$T(N) = 4 \cdot T\left(\frac{N}{2}\right) \qquad \text{mit} \qquad T(1) = 1$$

Diese läßt sich wie folgt lösen:

$$\begin{aligned} T(N) &= 4 \cdot T\left(\frac{N}{2}\right) \\ &= 16 \cdot T\left(\frac{N}{4}\right) \\ &\vdots \\ &= 4^{ldN} \quad = \quad 2^{2 \cdot ldN} \quad = \quad 2^{ldN^2} \\ &= N^2 \end{aligned}$$

Also gilt für dieses Divide-And-Conquer-Verfahren:

$$T(N) \in \Theta(N^2)$$

Eine **Alternative** ist die Lösung über den 1. Fall des Mastertheorems mit $a = 4$, $b = 2$ und $\epsilon > 0$.

2.)

Da das "naive" Verfahren ebenfalls in $\Theta(N^2)$ liegt, ergibt sich keine Verbesserung durch die Divide-And-Conquer-Strategie.

3.)

Für nun 3 Teilprobleme der Größe $\frac{N}{2}$ ergibt sich, ähnlich wie oben, die folgende Rekursionsgleichung:

$$T(N) = 3 \cdot T\left(\frac{N}{2}\right) \qquad \text{mit} \qquad T(1) = 1$$

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

Diese läßt sich dann wie folgt lösen:

$$\begin{aligned} T(N) &= 3 \cdot T\left(\frac{N}{2}\right) \\ &= 9 \cdot T\left(\frac{N}{4}\right) \\ &\vdots \\ &= 3^{ldN} \quad = \quad 2^{(ld3)\cdot(ldN)} \quad = \quad 2^{ld(N^{ld3})} \\ &= N^{ld3} \quad \approx \quad N^{1,58496...} \end{aligned}$$

Also gilt für dieses Divide-And-Conquer-Verfahren:

$$T(N) \in \Theta(N^{ld3})$$

Eine **Alternative** ist wiederum die Lösung über den 1. Fall des Mastertheorems mit $a = 3$, $b = 2$ und $\epsilon > 0$.

Fazit: Das Verfahren mit 3 Teilproblemen ist das bessere, da $ld3 < 2$ ist.

Aufgabe 23:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Das Verfahren ist auch als "binäre Suche" bekannt und benötigt als Laufzeit $O(ldn)$, da mit jedem Schritt das Problem halbiert wird.

Aufgabe 24:

1.) und 2.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

3.)

Die Anzahl der skalaren Multiplikationen beträgt bei der einfachen Multiplikation von links nach rechts 2835 Multiplikationen, mit optimaler Klammerung ergibt sich eine Anzahl von 1710 Multiplikationen.

4.)

Gemessen werden sollte die Laufzeit, nicht die Zahl der Schritte. Die Abtragungen der Laufzeit finden sich in Abb. 1 und Abb. 2.

Aufgabe 25:

Zunächst ist die Rekursionsgleichung des Problems von Prof. Dr. Theo Retisch zu aufzustellen. Es ergibt sich:

$$T(n) = 80 \cdot T(\left\lceil \frac{n}{3} \right\rceil) + w(n)$$

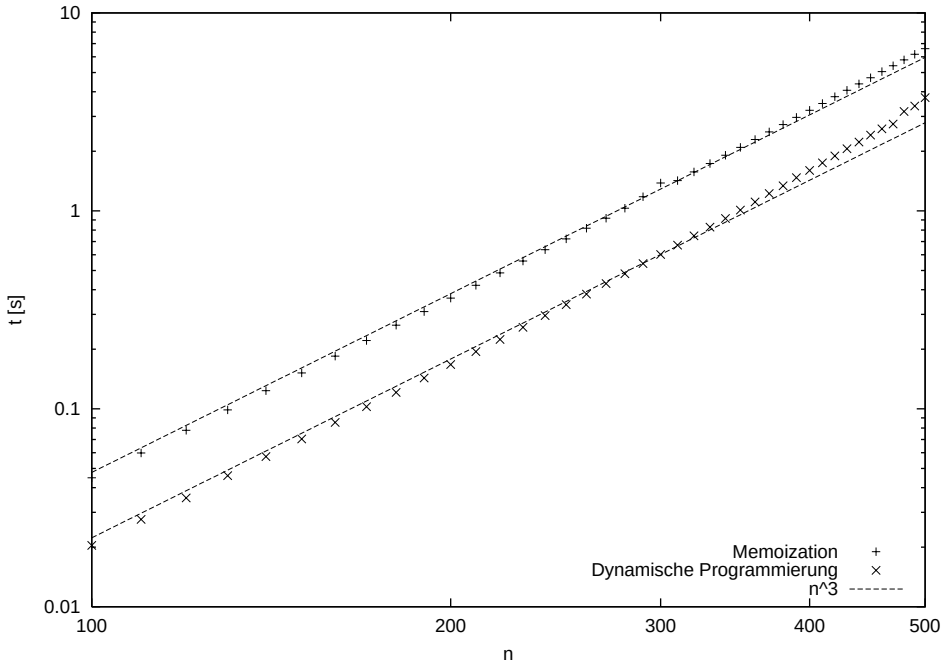


Abbildung 1: Laufzeit nach dynamischer Programmerung und Memoization

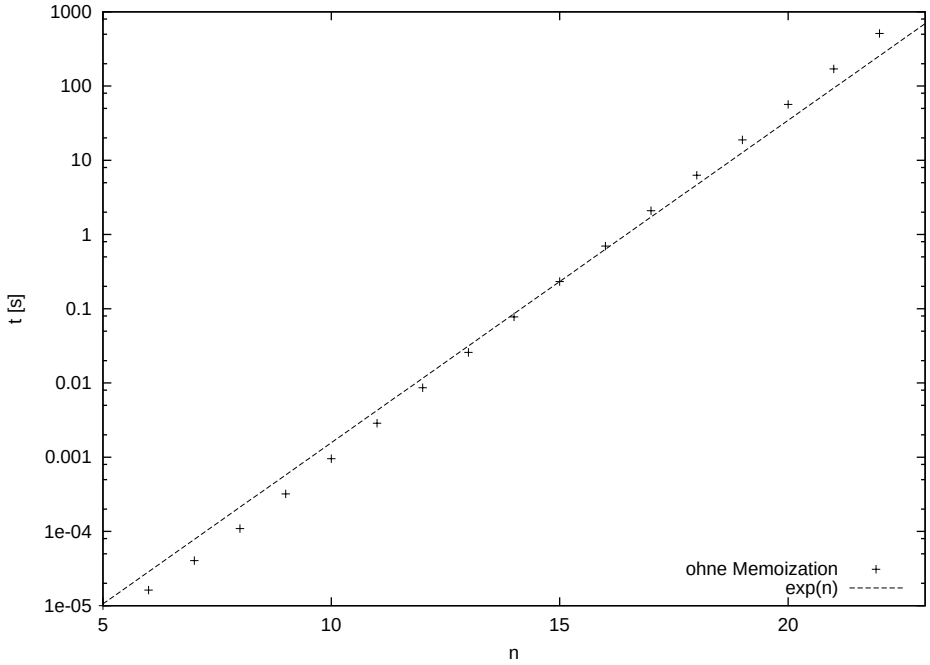


Abbildung 2: Laufzeit ohne Memoization

Um die Rekursionsgleichung zu lösen, muß zunächst die Rekursionsgleichung $w(n)$ gelöst werden.

$$w(n) = 26 \cdot w(\left\lceil \frac{n}{3} \right\rceil)$$

Die Gleichung $w(n)$ läßt sich nach dem Mastertheorem lösen. Mit $a = 26$, $b = 3$, $f(n) = 0$ und $\epsilon > 0$ gilt:

$$f(n) = 0 \in O(n^{log_3 26 - \epsilon})$$

Also gilt nach dem 1. Fall des Mastertheorems:

$$w(n) \in \Theta(n^{log_3 26}) \subseteq O(n^{log_3 27}) = O(n^3)$$

Nun können wir uns der Rekursionsgleichung $T(n)$ zuwenden. Auch hier kommt das Mastertheorem zum Einsatz. Mit $a = 80$, $b = 3$ und $\epsilon > 0$ gilt:

$$w(n) \in O(n^3) \subseteq O(n^{log_3 80 - \epsilon})$$

Also ist der 1. Fall des Mastertheorems anzuwenden. Es gilt:

$$T(n) \in \Theta(n^{log_3 80}) \approx \Theta(n^{3,98869\ldots})$$

Wenn nun das Verfahren von Prof. Dr. P.Raktisch nicht nur in $O(n^4)$, sondern sogar in $\Theta(n^4)$ liegt, dann ist das Verfahren von Prof. Dr. Theo Retisch das minimal schnellere hinsichtlich der Laufzeit. Dieses kann sich durch sie Art der Implementierung schon wieder aufgehoben werden. Eine genauere Untersuchung wäre somit von Nöten.

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 5. Übung*

Aufgabe 26:

- Sorten (Datentypen):

$$\begin{aligned} down : Queue &\mapsto Queue \\ up : Element \times Queue &\mapsto Queue \end{aligned}$$

- Operationen :

$$\begin{aligned} initqueue : &\mapsto Queue \\ enqueue : Element \times Queue &\mapsto Queue \\ dequeue : Queue &\mapsto Element \times Queue \\ isempty : Queue &\mapsto \{True, False\} \end{aligned}$$

- Axiome:

$$\begin{aligned} dequeue(enqueue(x,q)) &= dequeue(down(enqueue(x,q))) \\ down(enqueue(x,q)) &= enqueue(x, down(q)) \\ down(enqueue(x, initqueue)) &= up(x, initqueue) \\ enqueue(x, up(y,q)) &= up(y, enqueue(x,q)) \\ dequeue(up(x,q)) &= (x,q) \\ isempty(q) &= True, \quad \text{wenn } q = initqueue \\ &= False, \quad \text{sonst} \end{aligned}$$

Alternative:

Die folgende Spezifikation ist in einer Gofer-ähnlichen Notation formuliert. (Gofer ist eine funktionale Programmiersprache, die bis auf das Fehlen des Modul-Systems äquivalent zu Haskell ist). Queues können i.a. als Listen über einer Datentyp-Variable a spezifiziert werden. Zur Vereinfachung werden hier konkret Listen über natürlichen Zahlen betrachtet. Die übliche Darstellung für Listen ist:

`[ ]`	leere Liste
`h:[ ]`	Liste mit *einem* Element; äquivalent zu `[h]`
`h:t`	Liste mit Head = `h` und Tail (Listenrest) = `t`

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

Dabei bezeichnet „:“ (gespr. *cons*) den Operator zum linksseitigen Einfügen eines Elementes in die Liste. Eine Folge $a_1, a_2, a_3, \ldots, a_n$ wird dann wie folgt als Liste repräsentiert:

$$a_1 : a_2 : a_3 : \ldots : a_n : []$$

- Spezifikation des Datentyps Queue als Liste:

```
Queue -> [ ] | Nat : Queue
```

- Zugriffsoperationen:

```
enqueue :: (Nat, Queue) -> Queue
enqueue     n, [ ]      -> [n]
enqueue n, (h : t)      -> h : enqueue(n, t)

dequeue :: Queue  -> (Queue, Nat)
dequeue     [n]   -> ([ ], n)
dequeue n : h : t -> (h : t, n)

isempty :: Queue -> Bool
isempty      [ ] -> True
isempty   h : t  -> False
```

- Axiome:

```
enqueue(x, [ ]) = [x]
enqueue(x, h : t) = h : enqueue(x, t)
dequeue(enqueue(x, h : t)) = dequeue(h : enqueue(x, t))
dequeue(x : y) = (y, x)
isempty([ ]) = True
isempty(h : t) = False
```

Aufgabe 27:

a.)

Beim Sortieren mit Bubble-Sort werden nur benachbarte Elemente getauscht, falls ein Element echt größer als das Nachfolgeelement ist, d.h. gleiche Element "bubblen" sich von hinten heran, werden aber nie mit einem gleichen Element vertauscht.

```
IF a[j-1] > a[j] THEN
  Swap(a[j-1],a[j]);
```

Also ist der Bubble-Sort stabil !

b.)

Shell-Sort ist im Allgemeinen nicht stabil, da nicht nur benachbarte Elemente miteinander verglichen und vertauscht werden. Gegenbeispiel:
Gegeben sei folgende Folge:

$$1 \quad 5 \quad 3_1 \quad 3_2$$

D.h. es ist $n = 4$ und $incr = 2$.

- 1. Vergleich:
  1 und 3_1 werden verglichen: keine Aktion.

$$1 \quad 5 \quad 3_1 \quad 3_2$$

- 2. Vergleich:
  5 und 3_2 werden verglichen: Vertauschung der beiden Elemente

$$1 \quad 3_2 \quad 3_1 \quad 5$$

Damit ist die Ordnung geändert worden und Shell-Sort somit kein stabiler Sortieralgorithmus !

Aufgabe 28:

Im folgenden sei der Begriff *"Bewegung"* definiert als **ein Ringtausch**. Bei anderer Interpretation ist die Bewegungszahl mit 3 (für die Anzahl der Bewegungen im Ringtausch) zu multiplizieren.

- best-case:
  Da die Folge bereits sortiert ist, läuft lediglich einmal die FOR-Schleife von 1 bis $n-1$ durch. Da das aktuelle Element immer kleiner als das nächste ist, erfolgt keine Vertauschung. Damit gilt insgesamt:

$$0 \text{ Bewegungen, } n-1 \text{ Vergleiche } (V_{bv} \in O(n))$$

- worst-case:
  Die Folge ist bereits in umgekehrter Reihenfolge sortiert. Für **jedes** Element werden also $n-1$ Vergleiche benötigt. Es gilt insgesamt:

$$\tfrac{n\cdot(n-1)}{2} \text{ Bewegungen } (M_{wc} \in O(n^2)),\ n\cdot(n-1) \text{ Vergleiche } (V_{wc} \in O(n^2))$$

- average-case:
  Es liegt eine beliebige Folge vor. Aus worst-case und best-case ergibt sich für den average-case:

$$\tfrac{n\cdot(n-1)}{4} \text{ Bewegungen } (M_{ac} \in O(n^2)),\ \tfrac{n^2-1}{2} \text{ Vergleiche } (V_{ac} \in O(n^2))$$

Aufgabe 29:

a.)

16	5	2	7	4	8	10	12	11	6	13	14	15	3	9	|1|
1	5	2	7	4	8	10	12	11	6	13	14	15	3	9	|16|
	5	2	7	4	8	10	12	11	6	13	14	15	3	|9|	**16**
	5	2	7	4	8	3	|6|	**9**							
	5	2	3	|4|	**6**										
	3	|2|	**4**												
	2					7	|8|								
							8		12	13	14	15	10	|11|	
										11	14	15	12	|13|	
												13	14	|15|	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

b.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	|16|
1	2	3	4	5	6	7	8	9	10	11	12	13	14	|15|	**16**
1	2	3	4	5	6	7	8	9	10	11	12	13	|14|	**15**	
1	2	3	4	5	6	7	8	9	10	11	12	|13|	**14**		
1	2	3	4	5	6	7	8	9	10	11	|12|	**13**			
1	2	3	4	5	6	7	8	9	10	|11|	**12**				
1	2	3	4	5	6	7	8	9	|10|	**11**					
1	2	3	4	5	6	7	8	|9|	**10**						
1	2	3	4	5	6	7	|8|	**9**							
1	2	3	4	5	6	|7|	**8**								
1	2	3	4	5	|6|	**7**									
1	2	3	4	|5|	**6**										
1	2	3	|4|	**5**											
1	2	|3|	**4**												
1	|2|	**3**													
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

c.)

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	|1|
1	15	14	13	12	11	10	9	8	7	6	5	4	3	2	|16|
	15	14	13	12	11	10	9	8	7	6	5	4	3	|2|	**16**
	2	14	13	12	11	10	9	8	7	6	5	4	3	|15|	
		14	13	12	11	10	9	8	7	6	5	4	|3|	**15**	
		3	13	12	11	10	9	8	7	6	5	4	|14|		
			13	12	11	10	9	8	7	6	5	|4|	**14**		
			4	12	11	10	9	8	7	6	5	|13|			
				12	11	10	9	8	7	6	|5|	**13**			
				5	11	10	9	8	7	6	|12|				
					11	10	9	8	7	|6|	**12**				
					6	10	9	8	7	|11|					
						10	9	8	|7|	**11**					
						7	9	8	|10|						
							9	|8|	**10**						
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Aufgabe 30:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 31:

Wenn ein sortiertes Feld vorliegt, überprüft man zunächst die Summe aus dem kleinsten ($l[min]$) und dem größten Wert ($l[max]$). Ist diese Summe größer als der gesuchte Wert, dann kann $l[max]$ offensichtlich nicht eine der gesuchten Zahlen sein. Folglich kann man die Untersuchung auf $l[min..max-1]$ beschränken. Ist die Summe kleiner, dann kann $l[min]$ nicht Teil des Paares sein, und die Suche kann auf $l[min+1..max-1]$ beschränkt werden. Offensichtlich ist die Gesamtlaufzeit also $O(n)$.

Der angegebene Algorithmus verwendet die globalen Variablen p und q, um die beiden gefundenen Zahlen zu speichern, sofern sie denn im Feld existieren.

```
ALGORITHMUS Pair_Search(l:ARRAY[1..n] OF INTEGER; x,min,max:INTEGER):BOOLEAN =
VAR i : INTEGER;
BEGIN
  IF min = max THEN RETURN False;
  IF l[min]+l[max] = x THEN BEGIN
    p := l[min];
    q := l[max];
  ELSIF l[min]+l[max] > x THEN
    RETURN Pair_Search(l,x,min,max-1);
  ELSE
    RETURN Pair_Search(l,x,min+1,max);
  END;
END Pair_Search;
```

Der Source-Code der Implememententierung in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 6. Übung*

Aufgabe 32:

Die gegebene Folge genügt offensichtlich nicht den Heap-Bedingungen. Also werden zunächst die Heap-Bedingungen durch Versickern der Elemente $k_{\lfloor \frac{n}{2} \rfloor}$,$k_{\lfloor \frac{n}{2} \rfloor - 1}$, ... , k_1 hergestellt, in diesem Fall also die Elemente k_8,...,k_1.

- $k_8 = 12$ kann nicht weiter versickert werden
- $k_7 = 10$ kann nicht weiter versickert werden
- $k_6 = 8$ versickern:

16	5	2	7	4	15	10	12	11	6	13	14	8	3	9	1

- $k_5 = 4$ versickern:

16	5	2	7	13	15	10	12	11	6	4	14	8	3	9	1

- $k_4 = 7$ versickern:

16	5	2	12	13	15	10	7	11	6	4	14	8	3	9	1

- $k_3 = 2$ versickern:

16	5	15	12	13	14	10	7	11	6	4	2	8	3	9	1

- $k_2 = 5$ versickern:

16	13	15	12	6	14	10	7	11	5	4	2	8	3	9	1

- $k_1 = 16$ kann nicht weiter versickert werden

Damit sind nun die Heap-Bedingungen hergestellt und das Sortieren kann beginnen. Die bereits sortierte Liste wird hinten im Feld abgelegt.

1. Tauschen:

1	13	15	12	6	14	10	7	11	5	4	2	8	3	9	16

   Versickern:

15	13	14	12	6	8	10	7	11	5	4	2	1	3	9	16

2. Tauschen:

9	13	14	12	6	8	10	7	11	5	4	2	1	3	15	16

   Versickern:

14	13	10	12	6	8	9	7	11	5	4	2	1	3	15	16

3. Tauschen:

3	13	10	12	6	8	9	7	11	5	4	2	1	14	15	16

   Versickern:

13	12	10	11	6	8	9	7	3	5	4	2	1	14	15	16

4. Tauschen:

1	12	10	11	6	8	9	7	3	5	4	2	13	14	15	16

   Versickern:

12	11	10	7	6	8	9	1	3	5	4	2	13	14	15	16

5. Tauschen:

2	11	10	7	6	8	9	1	3	5	4	12	13	14	15	16

   Versickern:

11	7	10	3	6	8	9	1	2	5	4	12	13	14	15	16

6. Tauschen:

4	7	10	3	6	8	9	1	2	5	11	12	13	14	15	16

   Versickern:

10	7	9	3	6	8	4	1	2	5	11	12	13	14	15	16

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

7. Tauschen:	5	7	9	3	6	8	4	1	2	10	11	12	13	14	15	16
Versickern:	9	7	8	3	6	5	4	1	2	10	11	12	13	14	15	16
8. Tauschen:	2	7	8	3	6	5	4	1	9	10	11	12	13	14	15	16
Versickern:	8	7	5	3	6	2	4	1	9	10	11	12	13	14	15	16
9. Tauschen:	1	7	5	3	6	2	4	8	9	10	11	12	13	14	15	16
Versickern:	7	6	5	3	1	2	4	8	9	10	11	12	13	14	15	16
10. Tauschen:	4	6	5	3	1	2	7	8	9	10	11	12	13	14	15	16
Versickern:	6	4	5	3	1	2	7	8	9	10	11	12	13	14	15	16
11. Tauschen:	2	4	5	3	1	6	7	8	9	10	11	12	13	14	15	16
Versickern:	5	4	2	3	1	6	7	8	9	10	11	12	13	14	15	16
12. Tauschen:	1	4	2	3	5	6	7	8	9	10	11	12	13	14	15	16
Versickern:	4	3	2	1	5	6	7	8	9	10	11	12	13	14	15	16
13. Tauschen:	1	3	2	4	5	6	7	8	9	10	11	12	13	14	15	16
Versickern:	3	1	2	4	5	6	7	8	9	10	11	12	13	14	15	16
14. Tauschen:	2	1	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Versickern:	2	1	3	4	5	6	7	8	9	10	11	12	13	14	15	16
15. Tauschen:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Versickern:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

An dieser Stelle bricht die Repeat-Schleife ab, da $N \leq 1$ erfüllt ist. Die Folge ist fertig sortiert.

Aufgabe 33:

1.)

Ein modifinzierter Quicksort, der bei weniger als k Elementen abbricht, hat eine mittlere Rekursionstiefe von $ld(\frac{n}{k})$. Also folgt daraus für die Laufzeit:

$$T_{QS'} \in O(n \cdot ld(\frac{n}{k}))$$

Insertionsort, angewandt auf die "fast sortierte" Folge, muß jedes falschstehende Element maximal k Plätze weiter nach links verschieben. Es folgt also für die Laufzeit:

$$T_{IS} \in O(n \cdot k)$$

Für das gesamte Verfahren gilt daher:

$$T \in O(n \cdot k + n \cdot ld(\frac{n}{k}))$$

Zusätzliche Betrachtung:
Seien α und β der relative Aufwand für Quicksort bzw. Insertionsort, also

$$T \approx \alpha \cdot n \cdot ld(\frac{n}{k}) + \beta \cdot n \cdot k$$

Minimiere T bezüglich k:

$$\begin{aligned} \frac{dT}{dk} &= \alpha \cdot n \cdot \frac{k}{n} \cdot \left(-\frac{n}{k^2}\right) + \beta \cdot n \quad \stackrel{!}{=} \quad 0 \\ \Rightarrow \quad -\frac{\alpha}{k} + \beta &= 0 \\ \Rightarrow \quad \frac{\alpha}{\beta} &= k \end{aligned}$$

Also ist $k = \frac{\alpha}{\beta}$ die optimale Wahl für k, d.h. je aufwendiger die Quicksort-Implementierung und je schneller die Insertionsort-Implementierung, desto größer muß k gewählt werden.

2.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

3.)

Das Optimum für k liegt bei $k \approx 50$ (vgl. Abb.1).

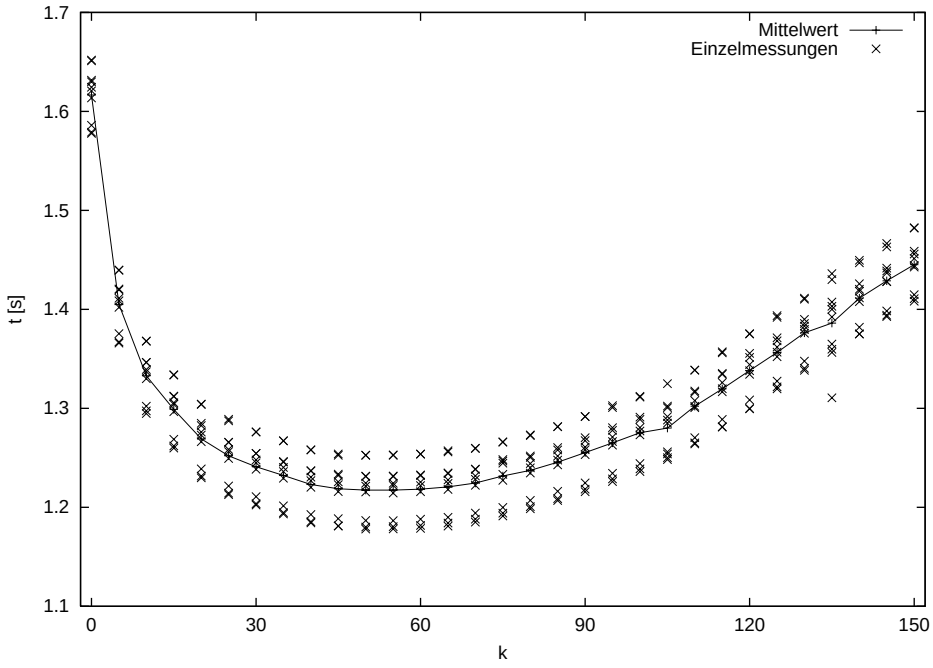


Abbildung 1: Laufzeit für modifizierten Quicksort abhängig von k

Aufgabe 34:

1.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

2.)

Average-Case:
Gegeben ist ein Teilarray der Länge $n = r - l + 1$. Der Aufwand für die Partitionierung ist $O(n)$. Danach wird der Median des größeren der beiden Teilstücke berechnet. Sei i die Position, an der das Pivot-Element zu liegen kommt, dann ist die Länge des nächsten zu bearbeitenden Arrays

$$n' = max\{i - l, r - i\} = max\{j, n - j - 1\} \quad \text{mit} \quad j := i - l$$

Da die Position des Pivot-Elementes zufällig zwischen l und r ist (j ist gleichverteilt zwischen 0 und $n - 1$), ist der erwartete Wert von n':

$$\begin{aligned}
n' &= \frac{1}{n} \cdot \sum_{j=0}^{n-1} max\{j, n - j - 1\} \\
&= \frac{1}{n} \cdot \left[\left(\sum_{j=0}^{\frac{n-1}{2}} n - j - 1 \right) + \left(\sum_{j=\frac{n-1}{2}+1}^{n-1} j \right) \right] \\
&= \frac{1}{n} \cdot \left[\left(\sum_{j=0}^{\frac{n-1}{2}} n - j - 1 \right) + \left(\sum_{j=0}^{\frac{n-1}{2}} j + \frac{n-1}{2} + 1 \right) \right] \\
&= \frac{1}{n} \cdot \left(\sum_{j=0}^{\frac{n-1}{2}} n - j - 1 + j + \frac{n-1}{2} + 1 \right) \\
&= \frac{1}{n} \cdot \left(\sum_{j=0}^{\frac{n-1}{2}} n + \frac{n-1}{2} \right) \\
&= \frac{1}{n} \cdot \left[\left(\frac{n-1}{2} + 1 \right) \cdot \left(n + \frac{n-1}{2} \right) \right] \\
&= \frac{1}{n} \cdot \left(\frac{n+1}{2} \cdot \frac{3n-1}{2} \right) \\
&= \frac{1}{n} \cdot \left(\frac{3n^2 + 2n - 1}{4} \right) \\
&= \frac{3}{4}n + \frac{1}{4n} + \frac{1}{2}
\end{aligned}$$

Da der Term $\frac{1}{4n} + \frac{1}{2}$ für große n vernachlässigbar ist, gilt für die Laufzeit folgende Rekursionsgleichung:

$$T(n) = T\left(\frac{3}{4}n\right) + n$$

Dabei stellt die Addition von n in jedem Schritt die Laufzeit für die in jedem Schritt notwendige Partitionierung (s.o.) dar. Nach Master-Theorem ergibt sich dann als Lösung:

$$T(n) \in \Theta(n)$$

Alternativ kann man auch berechnen:

$$T(n) = n \cdot \sum_{k=0}^{n} \left(\frac{3}{4}\right)^k = n \cdot \frac{1 - \left(\frac{3}{4}\right)^{n+1}}{1 - \frac{3}{4}} \quad \overset{n \to \infty}{\longrightarrow} \quad 4n$$

Worst-Case:
Im worst-case hat dieses Verfahren natürlich die Laufzeit des worst-case des QuickSort, da die Liste im ungünstigsten Fall komplett durchsortiert werden muß. Also: $T(n) \in O(n^2)$.

Aufgabe 35: (*Bonusaufgabe*)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 36: (*Bonusaufgabe*)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 37: (*Bonusaufgabe*)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 7. Übung*

Aufgabe 37: (*g* schon wieder)[1]

1.)

unsortierte Liste: Neue Elemente können irgendwo eingefügt werden, z.B. am Ende, daher Aufwand $O(1)$. Alle anderen Operationen erfordern komplettes Durchsuchen der Liste, daher $O(n)$.

sortierte Liste: Minimum kann in $O(1)$ aufgefunden werden, das Einfügen von Elementen erfordert jedoch $O(n)$ Schritte um die richtige Position zu finden.

binärer Suchbaum: Das kleinste Element befindet sich im „linkesten“ Knoten. Alle Operationen erfordern im „Normalfall“ $O(h)$ Schritte, wobei h die Höhe des Baums ist. Im Mittel ist $h \in O(ld\ n)$. Im schlimmesten Fall entartet der Baum zu einer absteigend sortierten Liste, d.h. jeder Knoten hat nur einen linken Sohn. Dann sind alle Operationen $O(n)$. Entartet der Baum zu einer aufsteigend sortierten Liste, so ist immerhin der Zugriff auf das Minimum in $O(1)$ Schritten möglich.

Heap: Der Heap erlaubt direkten Zugriff auf das kleinste Element da dies am Anfang des Arrays gespeichert ist. GetMin und ReplaceMin werden dadurch realisiert, dass man die Heap-Bedingung vorübergehend verletzt und anschließend durch „Versickern“ wieder herstellt mit Aufwand $O(ld\ n)$. Put arbeitet nach dem gleichen Prinzip, jedoch in umgekehrter Richtung („aufsteigen lassen“) und benötigt ebenfalls $O(ld\ n)$ Schritte (siehe Teil 3).
Die Definition der Heap-Eigenschaft muß man an dieser Stelle etwas flexibler auffassen. Es hindert einen schließlich niemand daran, es genau andersherum zu machen als beim Heap-Sort, wenn das im konkreten Anwendungsfall sinnvoll ist. Hier möchte man direkten Zugriff auf das Minimum, also fordert man, daß jedes Element **kleiner** als seine Nachfolger sein soll, also $a\left[\left\lfloor \frac{i}{2} \right\rfloor\right] < a[i]$ für alle i.

Übersicht:

	Put	Min	GetMin	ReplaceMin
unsortierte Liste	$O(1)$	$O(n)$	$O(n)$	$O(n)$
sortierte Liste	$O(n)$	$O(1)$	$O(1)$	$O(n)$
binärer Suchbaum	$O(ld\ n)$	$O(ld\ n)$	$O(ld\ n)$	$O(ld\ n)$
zu aufsteigender Liste entartet	$O(n)$	$O(1)$	$O(1)$	$O(n)$
zu absteigender Liste entartet	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Heap	$O(ld\ n)$	$O(1)$	$O(ld\ n)$	$O(ld\ n)$

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

[1]Das war ein Numerierungsfehler auf dem Aufgabenblatt, der nun der Einfachheit halber übernommen wird.

2.) und 3.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

4.)

Erst GetMin und dann Put aufzurufen würde zuerst das letzte Element des Array nach oben kopieren, dieses mittels DownHeap versichern lassen und anschliessend das neue Element an Ende des Array ablegen und mittel UpHeap aufsteigen lassen. Einfacher ist es, das erste Element gleich mit dem neuen zu überschreiben und dieses dann versickern zu lassen (vgl. Programmtext).

Aufgabe 38:

1.) und 2.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

3.)

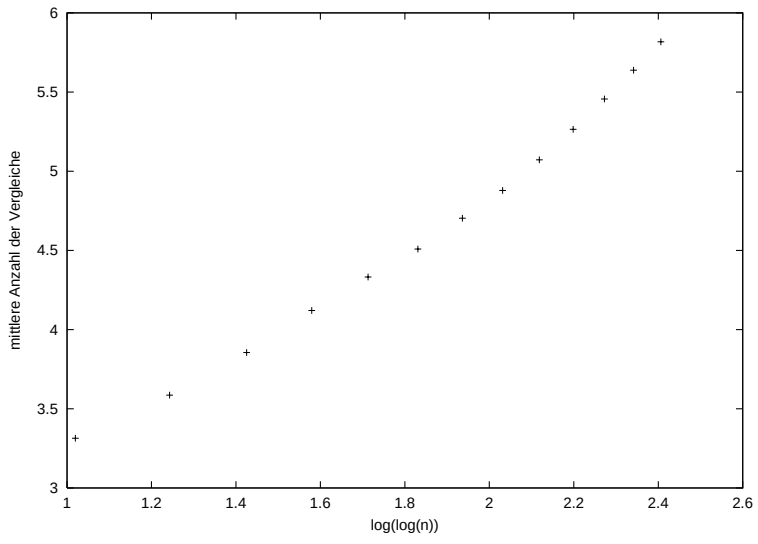


Abbildung 1: Experimentelle Verifikation der Laufzeit

Aufgabe 39:

Der Beweis wird als Widerspruchsbeweis geführt. Angenommen, es gibt Zahlen $x, y \in \mathbb{N}$ mit $x \neq y$ und

$$x \cdot A - \lfloor x \cdot A \rfloor = y \cdot A - \lfloor y \cdot A \rfloor$$

Dann gilt:

$$\begin{aligned} & x \cdot A - \lfloor x \cdot A \rfloor &=& \quad y \cdot A - \lfloor y \cdot A \rfloor \\ \Leftrightarrow \quad & \underbrace{(x-y)}_{\neq 0} \cdot A &=& \quad \underbrace{\lfloor x \cdot A \rfloor}_{\in \mathbb{N}} - \underbrace{\lfloor y \cdot A \rfloor}_{\in \mathbb{N}} \\ \Leftrightarrow \quad & A &=& \quad \frac{\lfloor x \cdot A \rfloor - \lfloor y \cdot A \rfloor}{x - y} \end{aligned}$$

Also ist A rational, was ein Widerspruch ist zur Voraussetzung, A sei irrational ($A \in \mathbb{R} \setminus \mathbb{Q}$).

Aufgabe 40:

Durchführung des Hashings mit

1. der Quersumme
2. der Division-Rest-Methode
3. der multiplikativen Methode

als Hashfunktion.

a.)

Lösen der Platzkollisionen durch ***lineares Sondieren***:

$h(k)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1.)	17	01	01	12	08	41	15	52	52	44	37	83	91	82	95	63	54	19	88
2.)	95	19	01	41	15	01	44	83	82	63	17	08	12	88	52	52	91	54	37
3.)	17	08	52	52	44	83	91	12	54	41	88	01	82	95	19	15	37	63	01

b.)

Lösen der Platzkollisionen durch ***quadratisches Sondieren***:

Variante 1 **ohne** Verfeinerung:

$h(k)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1.)	82	01	01	12	08	41	15	52	52	44	37	83	17	63	91	95	19	88	54
2.)	95	19	01	63	41	15	44	83	01	17	82	08	12	88	52	52	91	54	37
3.)	08	01	52	52	44	83	41	12	91	15	17	54	82	95	19	01	37	63	88

Variante 2 **mit** Verfeinerung:

$h(k)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1.)	19	01	15	12	41	54	82	52	52	44	37	83	17	63	91	95	88	01	08
2.)	95	19	01	41	01	82	44	83	17	08	63	15	12	88	52	52	91	54	37
3.)	08	88	52	52	44	83	54	12	91	15	41	01	82	95	19	01	37	63	17

Für die ganz heißen Fans des Hashings hier auch noch die Übersicht über die Anzahl der Kollisisonen bei obigem Beispiel (berechnet mit dem Modula-3-Programm zu Aufgabe 40 im Netz)):

Hashverfahren	lin. Sond.	quadr. Sond. 1	quadr. Sond. 2
Quersummen-Methode	77	122630	44
Division-Rest-Methode	38	46375	30
multiplikative Methode	35	103646	43

Bitte keine Nachfragen: die Zahlen in der 2.Spalte kommen mir auch etwas hoch vor, scheinen aber tatsächlich der Realität zu entsprechen.

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 8. Übung*

Aufgabe 41:

1.)

Gegeben die Zeichenfolge $s = s_1, s_2, \ldots, s_n$. Ordne jedem Zeichen seine Position im ISO-Zeichensatz zu: $s_i \in \{0, \ldots 255\}$ (A = 65, B = 66, ..., a = 97, b = 98, ...). In Modula-3 geschieht dies mit der Funktion ORD. Dann gibt es mehrere Möglichkeiten eine Hashfunktion zu erstellen:

Methode 1

Wir fassen die Zeichenfolge als Zahlendarstellung in der Basis $b = 256$ auf und verwenden die Divisions-Rest-Methode:

$$h(s) = \sum_{i=1}^{n} s_i b^{i-1} \; mod \; m$$

Die Summe ist in der Regel zu groß um als Ganzzahlvariable dargestellt zu werden, deswegen formen wir den Ausdruck folgendermaßen um:

$$h(s) = s_1 + b(s_2 + b(s_3 + b(s_4 + \ldots + s_n)\ldots)) \; mod \; m$$

Die Modulo-Operation kann in das Produkt hineingezogen werden, so daß kein Überlauf entsteht. Implementierung siehe Teil 2.

Methode 2 (Zusatz)

P. J. Weinberg hat die folgende Hash-Funktion für Strings vorgeschlagen, die etwas kompliziert aussieht, sich aber in der Praxis sehr gut bewährt hat (siehe 'Compilerbau" von A. V. Aho, R. Sethi und J. D. Ullman Seite 535, sogenanntes „Dragon Book"). Sie berechnet aus einem Eingabestring eine 28-bit Zahl, die dann z.B. mit der Divisions-Rest-Methode auf den Bereich 0,..., m-1 abgebildet wird.
Implementierung in C:

```
unsigned int string_hash(const char* cs) {
    const char* p;
    unsigned int i = 0, j;

    if (!cs) return 0 ;

    for (p = cs; *p != '\0'; p++) {
        i = (i << 4) + *p ;
        j = i & 0xf0000000 ;
```

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

```
            if (j) {
                i = i ^ (j >> 24) ;
                i = i ^ j ;
            }
        }
        return i ;
    }
```

Erläuterung:

1. Die `for`-Schleife iteriert über alle Zeichen im String. Dabei ist `*p` das jeweils aktuelle Zeichen.
2. `i << 4` ist der Wert von `i` um 4 Bit nach links verschoben (entspricht einer Multiplikation mit 16).
3. `i & 0xf0000000` sind die obersten 4 Bit von `i`.
4. `i = i ^ (j >> 24)` führt eine XOR-Verknüpfung durch: Die obersten 4 Bit von `i` werden unten wieder drangexodert.
5. `i = i ^ j` setzt die obersten 4 Bit von `i` auf Null.

2.)

Zur Verwaltung der Felder (Zellen) der Hash-Tabelle:
Delete darf eine Zelle nicht einfach als frei markieren, da sonst nachfolgende Search-Operationen die Sondierung zu früh abbrechen, wenn es bereit zu einer Kollision mit dem zu löschenden Element gekommen ist.
Daher wird jeder Zelle einer von *drei* möglichen Statuswerten zugeordnet:

unbenutzt (free): Insert verwendet Zelle zum Einfügen eines neün Eintrags. Search und Delete brechen (erfolglos) Sondierung ab.

belegt (used): Eintrag ist gültig. Sondierung wird fortgesetzt, falls Schlüssel nicht uberein-stimmt.

gelöscht (deleted): Insert verwendet Zelle zum Einfügen eines neuen Eintrags. Search und Delete setzten Sondierung fort.

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 42:

1.)

Idee:
Durchsuche den Baum mittels InOrder-Traversierung. (Andere Traversierungsarten sind auch möglich. InOrder hat den Vorteil, daß die Elemente sortiert ausgegeben werden.)
Als Optimierung werden Teilbäume, die komplett außerhalb des gesuchten Bereichs liegen, nicht traversiert.
Wenn der Schlüssel k des aktuellen Knotens kleiner als die untere Grenze i ist, so müssen alle Schlüssel im linken Teilbaum ebenfalls kleiner als i sein (Transitivität der Ordnungsrelation). Also braucht der linke Teilbaum nicht weiter traversiert zu werden.

Analoges gilt für den rechten Teilbaum und die obere Grenze.

Algorithmus:

```
Bereichssuche(n : Knoten) =
  IF n <> z THEN
    IF n^.key >= i THEN
      Bereichssuche(n^.left)
    IF i <= n^.key <= j THEN
      n ausgeben
    IF n^.key <= j THEN
      Bereichssuche(n^.right)
```

2.)

Sei $h \approx ld\, N$ die Höhe des Baums und m sie Anzahl der Schlüssel, die ausgegeben werden. Der Laufzeitbedarf ist $O(m+h)$, da

1. alle Knoten, die im abgefragten Bereich liegen eine Aufwand $O(1)$ verursachen,
2. es $O(h)$ Knoten gibt, die besucht aber nicht nicht ausgegeben werden. Diese liegen als Saum um die auszugebenden Knoten.

Das heißt also, um den Bereichsanfang zu finden, brauche ich $O(h)$. Jeder dann bearbeitete Knoten braucht $O(1)$. Da m Knoten in dem Bereich liegen folgt also $O(m)$. Faßt man beides zusammen, erhält man $O(m+h)$. **Bemerkung:** Wenn der Bereich keine Knoten enthält so ist der Aufwand $O(h)$. Wenn der Bereich alle Knoten enthält ist der Aufwand $O(N)$.

3.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 43:

Es sei $n(h)$ die *minimale* Zahl von Knoten, die in einem Baum der Höhe h enthalten sind, der den genannten Bedinungen genügt.
Ein Baum der Höhe $h+1$ hat (bezogen auf die Wurzel) mindestens einen Teilbaum der Höhe h mit mindestens $n(h)$ Knoten. Nach Voraussetzung hat der andere Teilbaum mindestens $\left\lceil \frac{n(h)}{2} \right\rceil$ Knoten. Daher gilt:

$$n(h+1) = 1 + n(h) + \left\lceil \frac{n(h)}{2} \right\rceil \geq \frac{3}{2} n(h)$$

Mit $n(0) = 1$ folgt

$$n(h) \geq \left(\frac{3}{2}\right)^h$$

und die Umkehrung

$$h(n) \leq \log_{3/2} n$$

Also ist $h \in O(\log n)$ und die Höhe des Baum wächst höchstens logarithmisch in der Anzahl der Knoten.

Aufgabe 44:

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 45: (*Bonusaufgabe*)

1.)

Der Algorithmus verfolgt ein einfaches Prinzip: wenn eine n-elementige Folge mit $n > 2^d$ Zahlen aus dem Bereich $0, \ldots, 2^d - 1$ enthält, dann ist offensichtlich, daß mindestens eine Zahl doppelt ist. An dieser Stelle macht man sich das Binärsystem zu Nutze. Wenn man im Binärsystem $n = 2^d$ setzt, dann ist über die ganze Folge aufsummiert die Anzahl der Nullen und Einsen an jeder Stelle der Zahlen identisch. Ist nun eine Zahl doppelt, so finden sich entweder mehr Nullen oder mehr Einsen an einer Stelle. Aus dieser Tatsache ergibt sich der folgende Algorithmus:

1. Schreibe alle Zahlen als Binärzahlen in ein Feld.
2. Prüfe für jede Stelle i der Folgen die Anzahl der Einsen und Nullen an der i-ten Stelle der verschiedenen Zahlen. Sind mehr Einsen vorhanden, dann hat die doppelte Zahl eine 1 an der Stelle i, ansonsten eine 0.
3. Betrachte für die Stelle $i + 1$ nur noch die Zahlen, die von hinten bis zur Stelle i bereits der konstruierten Zahl entsprechen.
4. Wenn das Ende der Folge noch nicht erreicht ist, fahre bei 2. fort, ansonsten ist eine doppeltvorhandene Zahl konstruiert.

Bei einem geraden $n > 2^d$ sind entweder 2 Zahlen doppelt oder eine Zahl dreifach vorhanden. Also kann gefahrlos eine Zahl entfernt werden bevor der Algorithmus durchläuft. Der Algorithmus arbeitet auch in diesem Sonderfall korrekt.
Der Algorithmus wird vielleicht im nächsten Aufgabenteil deutlich.

2.)

Wähle als Beispiel $d = 3$ und $n = 9$. Gegeben sei die Folge $\{001, 010, 110, 111, 011, 000, 100, 101, 010\}$. Alle **fettgedruckten** Zahlen werden in dem jeweiligen Schritt noch betrachtet.

	1.Schritt	2.Schritt	3.Schritt
betrachtete Stelle:	3	2	1
Feld:	**001** **010** **110** **111** **011** **000** **100** **101** **010**	001 **010** **110** 111 011 **000** **100** 101 **010**	001 **010** **110** 111 011 000 100 101 **010**
Anzahl 0:	5	2	2
Anzahl 1:	4	3	1
bereits konstruiert:	`xx0`	`x10`	`010`

$\Longrightarrow$ `010` ist in der Folge doppelt vorhanden

An diesem Beispiel wird auch ersichtlich, daß das Verfahren mit einer Laufzeitkomplexität von $O(n \cdot d)$ und einem konstanten Speicherbedarf auskommt.

3.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 9. Übung*

Aufgabe 46:

Zunächst ergibt sich aus der Aufstellung der Häufigkeiten:

α_0	β_0	α_1	β_1	α_2	β_2	α_3	β_3	α_4	β_4	α_5
4	8	1	3	2	5	4	10	1	4	3

Es gilt:

$$w(i,j) = \sum_{k=i}^{j} \alpha_k + \sum_{k=i+1}^{j} \beta_k$$

Ferner:

$$c(i,i) = 0 \qquad \text{für alle } i$$

$$c(i,j) = w(i,j) + \min_{i<k\leq j} [c(i,k-1) + c(k,j))] \qquad \text{für alle } i < j$$

Sowie:

$$r(i,j) = arg \min_{i<k\leq j} [c(i,k-1) + c(k,j))]$$

Aus diesen 3 Formeln kann man nun durch Einsetzen der Häufigkeiten aus obiger Tabelle die Matrizen zur Aufstellung des optimalen Suchbaums berechnen:

- Gewichtungsmatrix $w(i,j)$

i / j	1	2	3	4	5
0	13	18	27	38	45
1		6	15	26	33
2			11	22	29
3				15	22
4					8

- Pfadlängenmatrix $c(i,j)$

i / j	0	1	2	3	4	5
0	0	13	24	48	77	99
1		0	6	21	47	62
2			0	11	33	48
3				0	15	30
4					0	8
5						0

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

- Entscheidungmatrix $r(i,j)$

i / j	0	1	2	3	4	5
0	0	1	1	1	3	3
1		0	2	3	3	4
2			0	3	4	4
3				0	4	4
4					0	5
5						0

Damit ergibt sich nun aus der Matrix $r(i,j)$ der optimale Suchbaum wie folgt:

- Beginne beim optimalen Pfad $(0,5)$, da dieses die Wurzel darstellt. In diesem Fall ist die Wurzel also das Element mit dem Index $k=3$, d.h. k_3 bildet die Wurzel.
- Betrachte nun den linken Teilbaum: Da 3 bereits die Wurzel ist, muß für den linken Teilbaum nur noch der optimale Pfad für $(i,k-1)$, also $(0,2)$ betrachtet werden. Aus der Matrix ergibt sich für den linken Nachfolger der Wurzel das Element mit dem Index $k=1$, also k_1.
- Wende das gleiche Verfahren für den rechten Teilbaum an, allerdings im Bereich (k,j), also $(3,5)$. Es ergibt sich für den rechten Nachfolger das Element mit dem Index $k=4$, also k_4.
- Fahre für die weiteren Teilbäume genauso fort. Ist der Index $k=0$, so gibt es dort keinen weiteren Teilbaum mehr.

Ein Bilde des Baums folgt in Kürze...

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

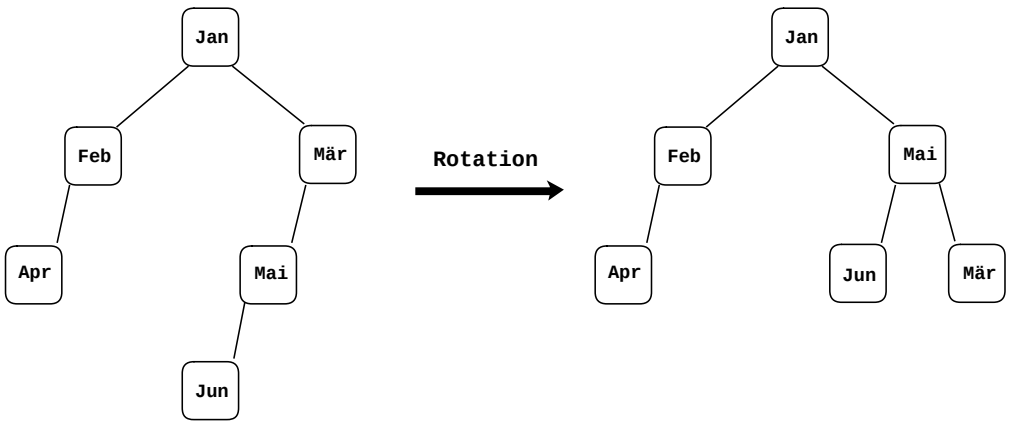
`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 47:

1.)

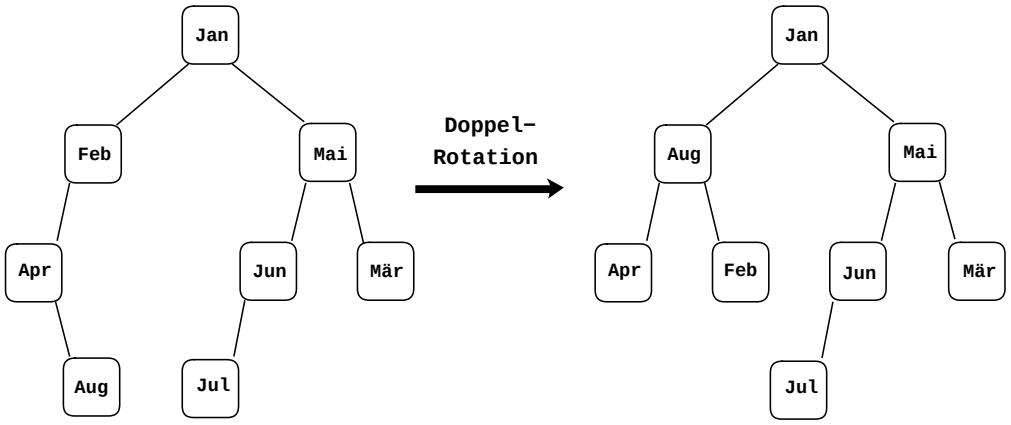
Einsortieren der Monatsnamen in Reihenfolge eines Jahres in einen AVL-Baum[1]:

- Januar, Februar, März, April, Mai, Juni einfügen, anschließend Rotation:

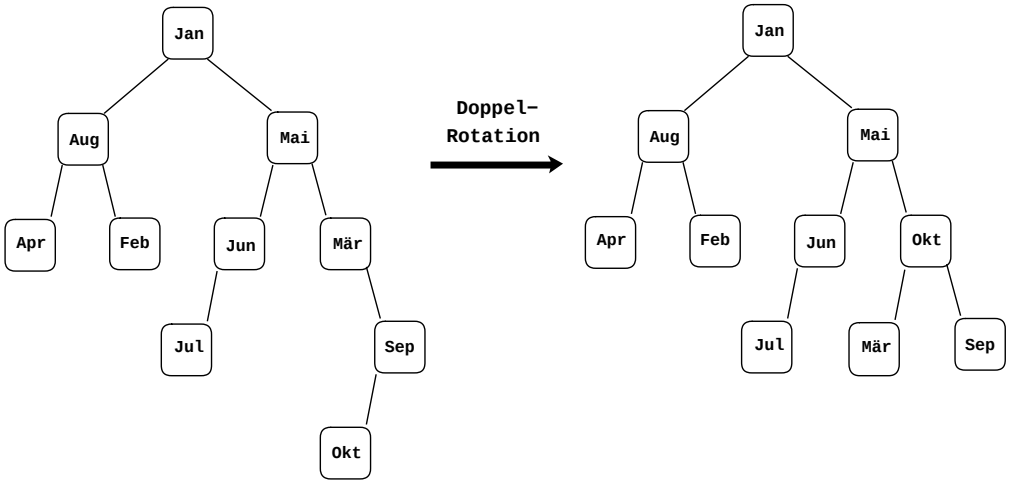


[1] Aus Arbeitsersparnisgründen sind hier nicht alle Zwischenschritte aufgeführt, die der Übersicht halber aber bei Klausuren, etc. unbedingt mit angegeben werden sollten...

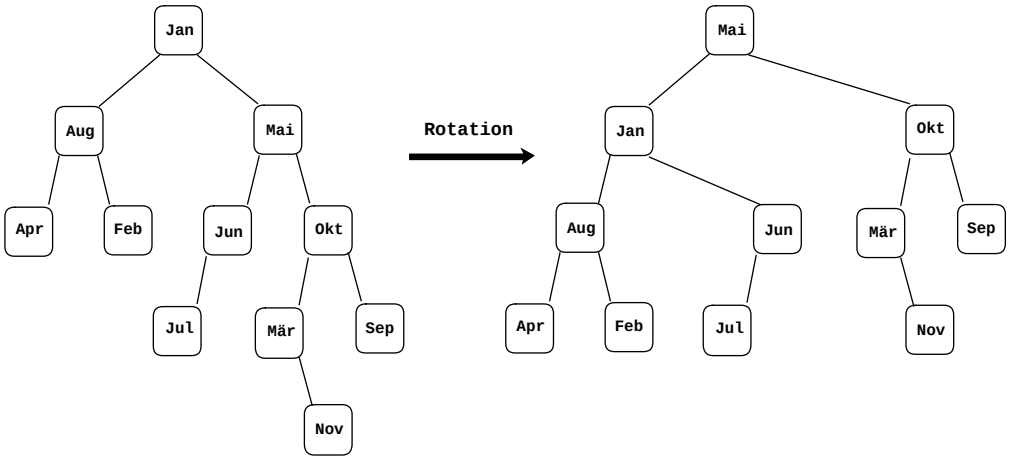
- Juli, August einfügen, anschließend Doppel-Rotation:



- September, Oktober einfügen, anschließend Doppel-Rotation[2]:

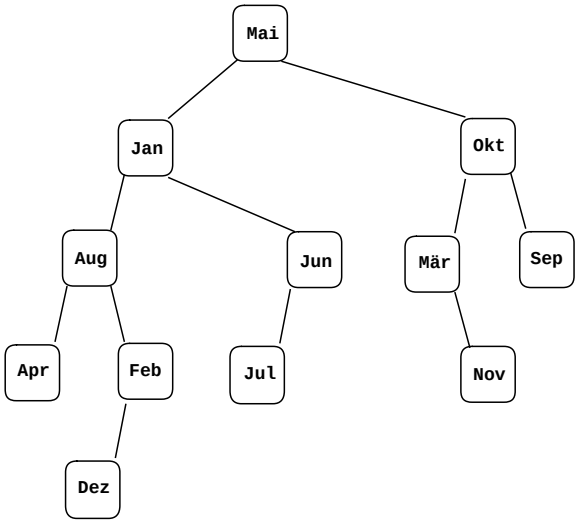


- November einfügen, anschließend Rotation:



[2]Die Doppelrotationen sind etwas unübersichtlich, allerdings ohne Zwischenschritte einfacher in LaTeX zu setzen *g*, außerdem wird so der Ausdruck der Musterlösung kürzer...

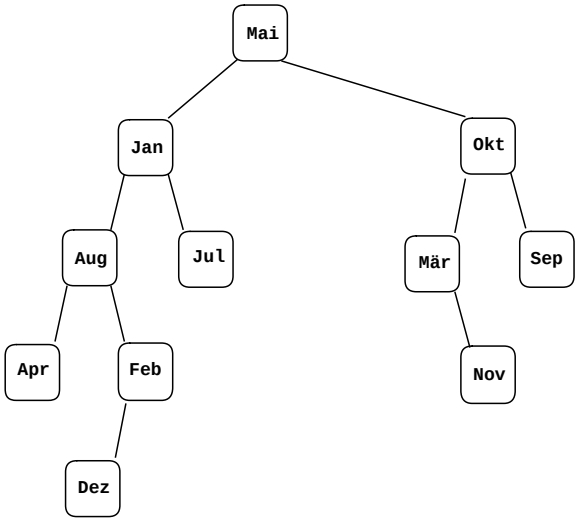
- Dezember einfügen:



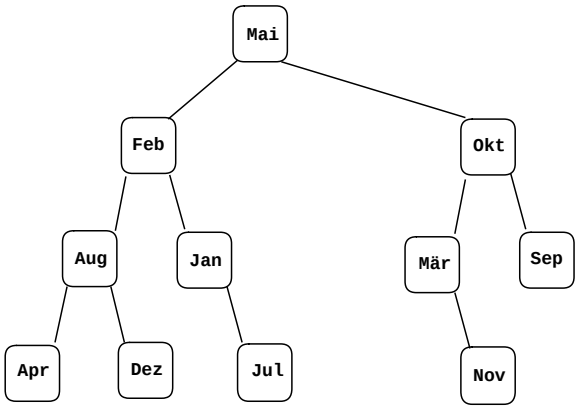
Geschafft, das war's !

2.)

- Entfernen des Juni:



- Doppelrotation links-rechts; zunächst nach links um den August, dann rechts um den Januar:



Nun ist auch das überstanden !

Aufgabe 48:

1.)

Ein B-Baum der Höhe h besteht aus $h+1$ Ebenen. Die Wurzel hat mindestens zwei Nachfolger, alle anderen Knoten bis auf die Blätter haben mindestens a Nachfolger. Daher befindet sich auf Ebene $i = 1$ genau ein Knoten (die Wurzel), auf Ebene $i = 2$ befinden sich mindestens 2 Knoten und auf den Ebenen $i > 2$ mindestens $2a^{i-2}$ Knoten. Die Wurzel enthält mindestens einen Schlüssel, alle anderen Knoten mindestens $a - 1$ Schlüssel. Daraus folgt:

$$\begin{aligned} N &\geq 1 + 2(a-1) + \sum_{i=3}^{h+1} 2a^{i-2}(a-1) \\ &= 1 + 2(a-1)\sum_{i=2}^{h+1} a^{i-2} = 1 + 2(a-1)\sum_{i=0}^{h-1} a^i \\ &= 1 + 2(a-1)\frac{a^h - 1}{a-1} = 2a^h - 1 \end{aligned}$$

Dies ist offenbar äquivalent zu:

$$\frac{N+1}{2} \geq 2^h$$

Wir nehmen auf beiden Seiten den Logarithmus zur Basis a und erhalten die Bahauptung:

$$h \leq \log_a \frac{N+1}{2}$$

2.)

Idee: Innerhalb der Knoten binäre Suche. Wird der Schlüssel nicht gefunden, so ergibt sich dabei der Teilbaum in dem der Schlüssel liegen müsste (wenn er vorhanden ist). Dort rekursiv weitersuchen, falls der Teilbaum vorhanden ist, sonst Suche erfolglos abbrechen. Formal:

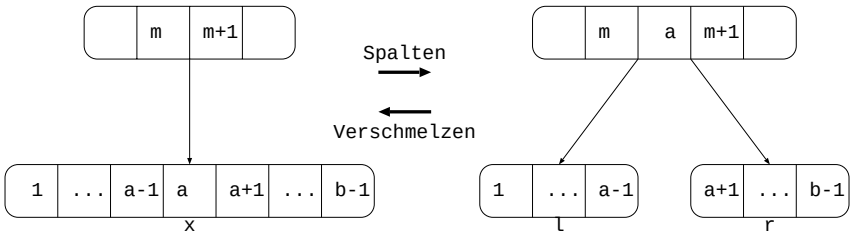
```
Search(k : KeyType ; x : Node ; i, j: [1..b − 1]) : Node, [1..b − 1] =
    IF i ≤ j THEN
        m := ⌊(i+j)/2⌋
        IF k = x.key[m] THEN
            RETURN x, m (gefunden!)
        ELSIF k < x.key[m] THEN
            RETURN Search(v, x, i, m − 1)
        ELSE
            RETURN Search(v, x, m + 1, j)
    ELSIF x.child[i] ≠ NIL THEN
        RETURN Search(v, x.child[i], 1, x.child[i].n)
    ELSE
        RETURN nicht gefunden!
```

Aufruf erfolgt mit Search(k, root, 1, root.n). Zurückgeliefert wird der Knoten und die Position innerhalb des Knotens. Anzahl der Vergleiche ist $O(ld\ N)$. Bei Speicherung der Knoten auf einem externen Medium sind $O(\log_a N)$ Zugriffe nötig.

3.)

Ein voller Knoten hat $b-1$ Schlüsselelemente. Wegen $b = 2a$ sind dies genug Schlüssel um daraus zwei Knoten mit je $a-1$ Schlüsseln und einen Schlüssel über zu behalten ($b-1 = 2\cdot(a-1)+1$). Da der Vaterknoten nach Voraussetzung nicht voll ist, kann der übrig gebliebene Schlüssel dort

eingefügt werden. Dadurch können dann die beiden neu entstandenen Knoten Söhne des Vaters werden.



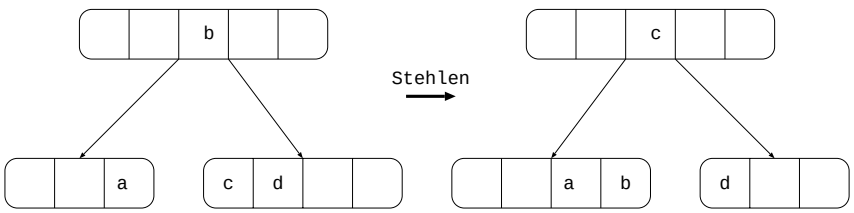
Algorithmus zum Einfügen: Steige wie beim Suchen von der Wurzel abwärts bis der richtige Einfügeplatz gefunden ist. Wenn dabei ein voller Knoten betreten wird, spalte diesen. Dies gilt auch für die Wurzel: Wenn die Wurzel voll ist, spalte diese und lege eine neue Wurzel an mit dem übrig gebliebenen Schlüssel als einzigem Element. Damit ist sichergestellt, daß der Knoten, in den der neue Schlüssel eingefügt werden muß, nicht voll ist, und daß falls ein Knoten gespalten werden muß, dessen Vater nicht voll ist.

Insert(k : KeyType ; x : Node) =
    IF x ist voll THEN
        Spalte x wie beschrieben: $(x) \rightarrow (l)a(r)$
        IF $k < a$ THEN
            Insert(k, l)
        ELSE
            Insert(k, r)
    ELSE
        Finde die Position i, an der k eingefügt werden muß (mittels binärer Suche)
        IF x hat Nachfolger THEN
            Insert(k, x.child[i])
        ELSE
            Füge k an der Stelle i in x ein.

Neue Schlüssel werden immer auf der untersten Ebene eingefügt. Dabei werden gegebenenfalls Knoten gespalten und Schlüssel nach oben verschoben. (Der Baum wächst sozusagen von unten nach oben.) Die Tiefe aller Knoten bleibt dabei erhalten, es sei denn die Wurzel wird gespalten, wobei sich die Tiefe aller Knoten um eins erhöht. Daher bleiben insbesondere alle Blattknoten auf der gleichen Ebene.

4.)

Zwei minimal gefüllte Geschwisterknoten haben zusammen $b - 2$ Schlüssel. Verschmilzt man sie, so hat ihr (gemeinsamer) Vater einen Nachfolger zu wenig, daher muß ein Schlüssel aus dem Vaterknoten nach unten in den neuen verschmolzenen Knoten wandern. Dies ist genau der umgekehrte Vorgang des Spaltens. Man kann Schlüssel zwischen direkt benachbarten Geschwisterknoten verschieben, ohne die geforderte Ordnungsbeziehung zu verletzen, indem man den trennenden Schlüssel des Vaterknoten mit einbezieht:



Diese Operation wird als *Stehlen* bezeichnet. Algorithmus zum Löschen: Steige wie beim Suchen von der Wurzel abwärts bis der zu entfernende Schlüssel gefunden ist. Wenn dabei ein minimal

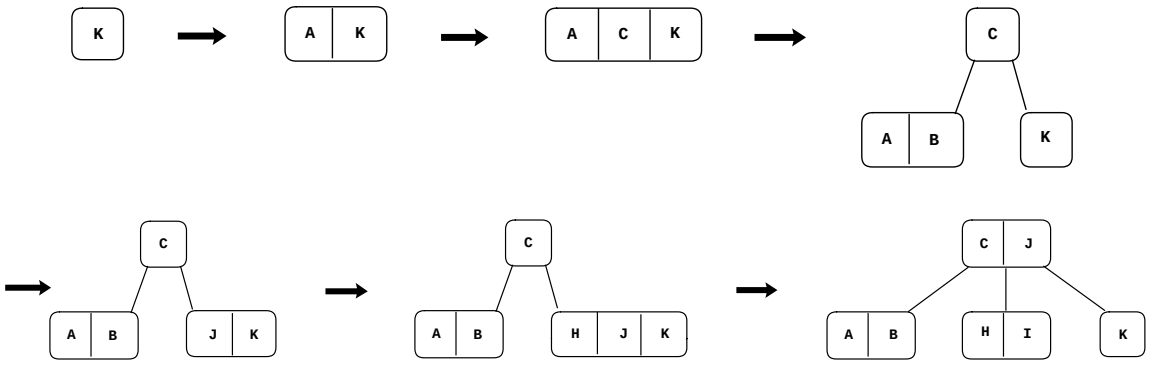
gefüllter Knoten betreten wird, sorge durch Stehlen oder Verschmelzen dafür, das dieser nicht mehr minimal gefüllt ist. Befindet sich der zu löschende Schlüssel k nicht in einem Blattknoten, so entferne den Schlüssel r, der in dem Teilbaum links von k am weistesten rechts liegt und ersetzte k durch l.

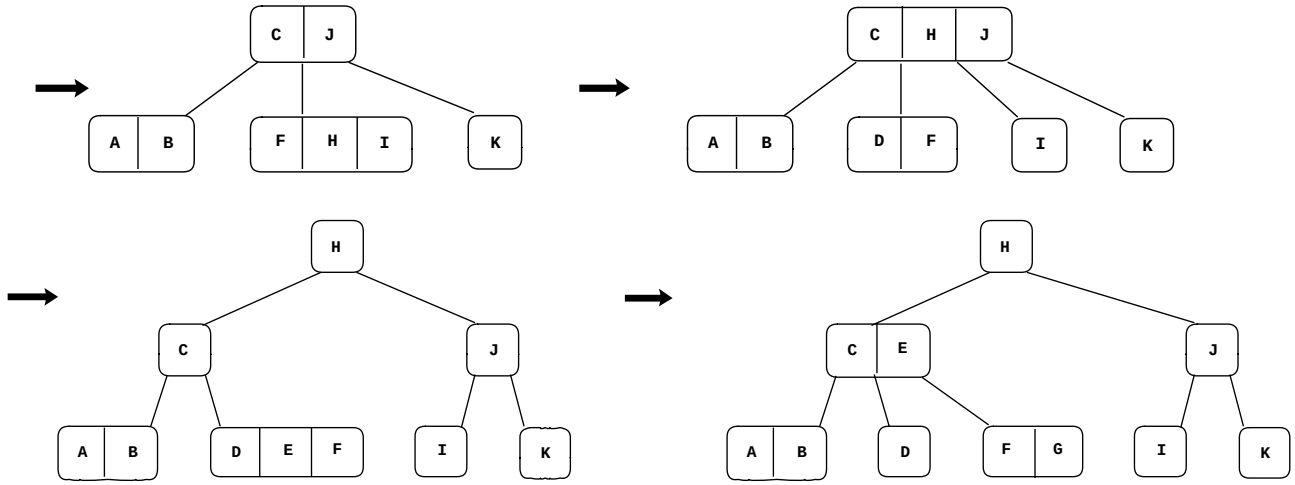
Delete(k : KeyType ; x : Node) =
    IF x ist minimal gefüllt THEN
        IF x hat einen nicht minimal gefüllten Geschwisterknoten y THEN
            Stehle einen Schlüssel von y.
        ELSIF x hat einen rechten Geschwisterknoten THEN
            Verschmelze x mit seinem rechten Geschwisterknoten.
        ELSE
            Verschmelze x mit seinem linken Geschwisterknoten.
    Finde die Position i an der sich k befinden müsste (mittels binärer Suche).
    IF $k = x$.key[i] THEN
        IF x hat Nachfolger THEN
            x.key[i] := DeleteRightmost(x.child[i])
        ELSE
            Entferne den i-ten Schlüssel aus x.
    ELSE
        IF x hat Nachfolger THEN
            Delete(k, x.child[i])
        ELSE
            k ist nicht vorhanden.

DeleteRightmost(x : Node) =
    IF x ist minimal gefüllt THEN
        Wie bei Delete...
    IF x hat Nachfolger THEN
        RETURN DeleteRightmost(x.child[$x.n + 1$])
    ELSE
        $r := x$.key[$x.n$]
        $x.n := x.n - 1$
        RETURN r

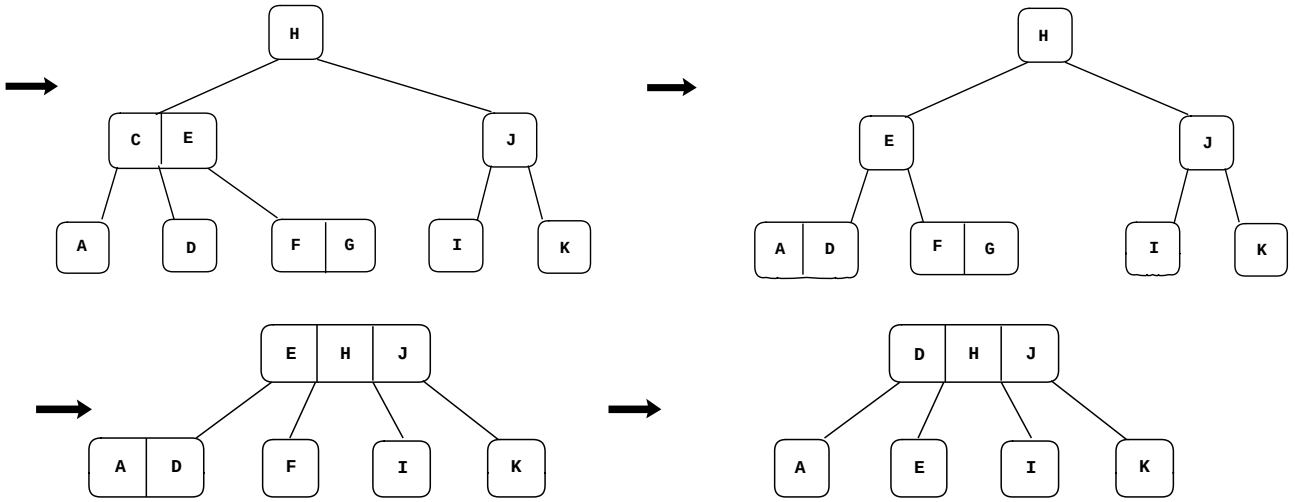
5.)

Praktisch angewandt sieht der Algorithmus beim Einfügen der gegebenen Zeichenkette dann so aus:





Nun werden die Elemente `B C G F` in genau dieser Reihenfolge entfernt:



Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de

Datenstrukturen und Algorithmen
Musterlösung 10. Übung*

Aufgabe 49:

1.)

Satz: Zu $G = (V, E)$ existiert eine topologische Sortierung genau dann, wenn G azyklisch ist.

Beweis:

- „$\Rightarrow$“:
  Sei $\mathsf{ord} : V \to \{1, \ldots, n\}$ eine topologische Sortierung von G. Angenommen, G enthält einen Zyklus $v_0 \to_E v_1 \to_E \ldots \to_E v_0$. Dann ist $\mathsf{ord}(v_0) < \mathsf{ord}(v_1) < \ldots < \mathsf{ord}(v_0)$. Dies ist ein Widerspruch.
- „$\Leftarrow$“:
  Es sei G azyklisch. Wir zeigen die Behauptung durch Induktion über $|V| = n$.

$n = 1$: $V = \{v_1\}$. Da G azyklisch ist, gibt es keine Kante $(v_1, v_1) \in E$. $\curvearrowright$ $\mathsf{ord} : v_1 \to 1$ ist eine topologogische Sortierung.

$n \mapsto n + 1$: $|V| = n + 1$.

1. Es gibt ein $v \in V$ mit $\mathsf{indeg}(v) = 0$, denn: Sei $v_0 \in V$ beliebig. Entweder ist $\mathsf{indeg}(v_0) = 0$ oder es gibt $v_1 \in V$ mit $v_1 \to_E v_0$. Nun ist wieder $\mathsf{indeg}(v_1) = 0$ oder es gibt ein $v_2 \in V$ mit $v_2 \to_E v_1 \to_E v_0$.
   Ist stets $\mathsf{indeg}(v_i) \neq 0$, so erhält man eine unendliche Folge. Da V endlich ist, gibt es ein $i < j$ mit $v_i = v_j$. Damit ist aber
   $$v_j \to_E v_{j-1} \to_E \ldots \to v_i = v_j$$
   ein Zyklus.
2. Es sei nun v ein Knoten mit $\mathsf{indeg}(v) = 0$.
   Definiere $G \setminus v := (V', E')$ mit
   - $V' := V \setminus \{v\}$
   - $E' := E \cap (V' \times V')$

   Offenbar ist $G \setminus v$ azyklisch und $|V'| = n$. Nach Induktion existiert $\mathsf{ord'} : V' \to \{1, \ldots, n\}$, so daß
   $$(w, w') \in E \quad \curvearrowright \quad \mathsf{ord'}(w) < \mathsf{ord'}(w')$$
   Definiere nun $\mathsf{ord} : V \to \{1, \ldots, n+1\}$ durch $\mathsf{ord}(v) := 1$ und $\mathsf{ord}(v') := \mathsf{ord'}(v') + 1$ für alle $v' \in V'$. Da $\mathsf{indeg}(v) = 0$ ist, folgt leicht, daß ord eine topologische Sortierung von G ist.

*Keine Garantie für Richtigkeit, wurde von mir nur für meine Übungsgruppe (Gruppe 9 - Achim Lücking) in LaTeX gesetzt...

Bemerkung: Der Beweis ist konstruktiv, d.h. er zeigt, wie man ord tatsächlich berechnen kann bzw. wie man einen Zyklus feststellt.

- **Algorithmus:** Topologisches Sortieren
- **Eingabe:** azyklischer, gerichteter Graph $G = (V, E)$
- **Ausgabe:** Sortierung der Knoten, so daß die durch $\rightarrow \subseteq V \times V$ beschriebene Halbordnung zu einer totalen Ordnung erweitert wird.

2.)

Algorithmus *Topologische Sortierung*

```
var lfd.NR.: 1..knotenzahl;
    Gradnull: stack of knotentyp;
    Eingrad: array[knotentyp] of 0..knotenzahl-1;
begin
  Eingrad[v]:=indeg(v) in G f"ur alle v in V;
  f"uge alle v mit Eingrad[v]=0 in Gradnull ein;
  lfd.NR.:=0;
  while Gradnull≠ ∅ do begin
    w"ahle v aus Gradnull und entferne es daraus;
    lfd.NR.:=lfd.NR.+1;
    ord[v]:=lfd.NR.;
    p:=adjazenzliste[v];
    while p≠nil do begin (* Neuer Eingrad[w] in G\v *)
      w:=p↑.endknoten;
      Eingrad[w]:=Eingrad[w]-1;
      if Eingrad[w]=0 then f"uge w in Gradnull ein;
    end
  end
  if lfd.Nr.=knotenzahl  then
    G azyklisch
  else
    G enh"alt Zyklus
  end
end
```

3.)

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 50:

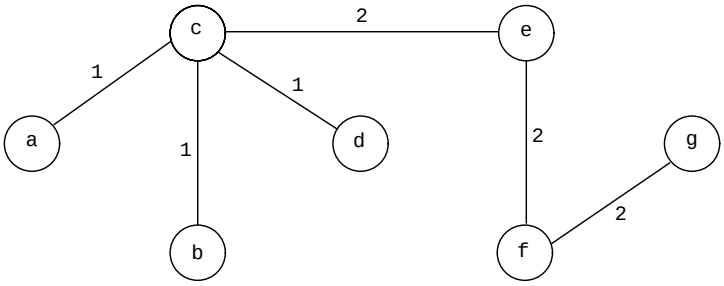
Der Beispielgraph aus der Aufgabenstellung ist nicht zusammenhängend, da die Adjazenzliste der transitiven Hülle mindestens eine 0 aufweist. Nur wenn die Adjazenzmatrix der transitiven Hülle ausschließlich aus Einsen besteht, ist der Graph nach Definition im Skript zusammenhängend.

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Aufgabe 51:

Der minimale Spannbaum des gegebenen Graphen sieht wie folgt aus:



Aufgabe 52:

Nach dem Dijkstra-Algorithmus ergibt sich die folgende Tabelle:

k	w	S_k	b	c	d	e	f	g
0	a	{a}	3	∞	∞	∞	∞	∞
1	b	{a,b}	-	7	5	∞	10	∞
2	d	{a,b,d}	-	6	-	14	10	∞
3	c	{a,b,d,c}	-	-	-	8	10	∞
4	e	{a,b,d,c,e}	-	-	-	-	10	14
5	f	{a,b,d,c,e,f}	-	-	-	-	-	12
6	g	{a,b,d,c,e,f,g}	-	-	-	-	-	-

Damit ist der günstigste Weg von a nach g der Weg:

a, b, f, g

Der Weg hat die Kosten 12.

Aufgabe 53: *(Bonusaufgabe)*

Der Source-Code in Modula-3 wird auf den Seiten des Lehrstuhls zum Download angeboten.

`http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/Datenstrukturen`

Viel Spaß beim Nachrechnen und Nachvollziehen...

Fehler und Korrekturen bitte an mich per E-Mail: Achim.Luecking@Post.rwth-aachen.de