Lösungen

1. Einleitung

1. Sei $k = 2l$, $l \in \mathbb{N}$. Die Zahl $2^k - 1 = 2^{2l} - 1$ ist durch 3 teilbar. Um dies einzusehen, rechnen wir $2^{2l} \bmod 3 = (2^2 \bmod 3)^l = 1$. Somit ist $2^k - 1$ durch 3 teilbar.
   Aus $n = \frac{2^k - 1}{3} \in \mathbb{N}$ für gerades $k \in \mathbb{N}$, folgt $2^k - 1 = 3n$. Somit ist n ungerade. Da $3n + 1 = 2^k$ gilt, terminiert $\mathrm{Col}(n)$.

2. Die Invariante der for-Schleife lautet: $a[1], \ldots, a[i]$ ist sortiert. Wir zeigen die Invariante der for-Schleife durch Induktion nach i. Der Induktionsanfang für $i = 1$ ist richtig. Sei $i \geq 2$. Wir schließen von $i - 1$ auf i und nehmen an, dass $a[1..i-1]$ sortiert ist. Falls $x \geq a[j] = a[i-1]$ gilt, ist $a[1..i]$ sortiert.
   Wir betrachten den Fall $x < a[j]$. Nach Terminierung der while-Schleife (d. h. unmittelbar vor Ausführung von Zeile 6) gilt:

   $$a[1] \leq a[2] \leq \ldots \leq a[j] \leq a[j+2] \leq a[j+3] \leq \ldots \leq a[i] \text{ und}$$
   $$a[j] \leq x < a[j+2].$$

   Wir zeigen dies durch Induktion nach der Anzahl k der Ausführungen der while-Schleife. Die Bedingung ist erfüllt, falls $k = 1$, d. h. $j = i - 2$. Sei also $k > 1$. Wir schließen von $k - 1$ auf k, d. h. von j auf $j - 1$.
   Durch absteigende Induktion nach j folgt:

   $$a[1] \leq a[2] \leq \ldots \leq a[j], a[j+2] \leq a[j+3] \leq \ldots \leq a[i] \text{ und } x < a[j+2].$$

   Nach Terminierung der while-Schleife gilt: $x \geq a[j]$. Insgesamt ergibt sich, dass $a[1..i]$ sortiert ist. Für $i = n$ folgt, dass $a[1..n]$ sortiert ist, d. h. der Algorithmus ist korrekt.

3.

$$\begin{aligned} T_1(m_1) &= c_1 m_1 \;=\; T \\ T_2(m_2) &= c_2 {m_2}^3 \;=\; T \\ T_3(m_3) &= c_3 2^{m_3} \;=\; T \end{aligned}$$

$$\begin{aligned} c_1 k m_1 &= kT \\ c_2 k {m_2}^3 = c_2(\sqrt[3]{k} m_2)^3 &= kT \\ c_3 k 2^{m_3} = c_3 2^{ld(k)+m_3} &= kT \end{aligned}$$

Für die neuen maximalen Größen der Eingaben $\tilde{m}_i$, $i = 1, 2, 3$, gilt: $\tilde{m}_1 = km_1$, $\tilde{m}_2 = \sqrt[3]{k} \cdot m_2$ und $\tilde{m}_3 = ld(k) + m_3$.

4. Bezüglich des asymptotischen Wachstums gilt:
$f_9 < f_5 < f_{11} < f_4 = f_3 < f_6 < f_{12} < f_2 < f_8 < f_7 < f_1 < f_{10}$

$f_9 < f_5$: $1/3^n \log(\log(n)) \longrightarrow 0, n \longrightarrow \infty$.

$f_5 < f_{11}$: $\log(\log(n))/\sqrt{\log(\log(n))\log(n)} =$
$\sqrt{\log(\log(n))^2/\log(\log(n))\log(n)} =$
$\sqrt{\log(\log(n))/\log(n)} \longrightarrow 0, n \longrightarrow \infty$.

$f_{11} < f_4$: $\sqrt{\log(\log(n))\log(n)}/\log(\sqrt{n}) =$
$\sqrt{\log(\log(n))\log(n)/(1/2\log(n))^2} =$
$\sqrt{4\log(\log(n)/\log(n)} \longrightarrow 0, n \longrightarrow \infty$.

$f_4 = f_3$: $\log(\sqrt{n})/\log(n) = 1/2$.

$f_3 < f_6$: $\log(n)/\log(n)^2 = 1/\log(n) \longrightarrow 0, n \longrightarrow \infty$.

$f_6 < f_{12}$: $\log(n)^2/2^{\sqrt{\log(\log(n))\log(n)}} =$
$2^{\log(\log(n)^2) - \sqrt{\log(\log(n))\log(n)}} \longrightarrow 0, n \longrightarrow \infty,$
denn $\log(\log(n)^2) - \sqrt{\log(\log(n))\log(n)} =$
$\sqrt{\log(\log(n))\log(n)} \left(2\sqrt{\frac{\log(\log(n))}{\log(n)}} - 1\right) \longrightarrow -\infty,$
$n \longrightarrow \infty$.

$f_{12} < f_2$: $2^{\sqrt{\log(\log(n))\log(n)}}/\sqrt{n} =$
$\left(2^{2\sqrt{\log(\log(n))\log(n)} - \log(n)}\right)^{1/2} \longrightarrow 0, n \longrightarrow \infty,$
denn $2\sqrt{\log(\log(n))\log(n)} - \log(n) =$
$\log(n)\left(2\sqrt{\frac{\log(\log(n))}{\log(n)}} - 1\right) \longrightarrow -\infty, n \longrightarrow \infty$.

$f_2 < f_8$: $\sqrt{n}/\sqrt{n}\log(n)^2 = 1/\log(n)^2 \longrightarrow 0, n \longrightarrow \infty$.

$f_8 < f_7$: $\sqrt{n}\log(n)^2\log(n)/n = \log(n)^3/\sqrt{n} \longrightarrow 0, n \longrightarrow \infty$.

$f_7 < f_1$: $n/n\log(n) = 1/\log(n) \longrightarrow 0, n \longrightarrow \infty$.

$f_1 < f_{10}$: $n/(3/2)^n = n/2^{n(\log(3)-1)} \longrightarrow 0, n \longrightarrow \infty$.

5. f_1 und f_2 sind von derselben Ordnung,
   $f_1, f_2, f_3 = O(f_4)$,
   $f_1, f_2, f_4 \neq O(f_3)$,
   $f_3 \neq O(f_1), f_3 \neq O(f_2)$.
6. Wir untersuchen zunächst das Verhalten von $n(\sqrt[n]{n} - 1)$ für $n \to \infty$.

$$\lim_{n\to\infty} n(\sqrt[n]{n} - 1) = \lim_{n\to\infty} \frac{n^{\frac{1}{n}} - 1}{\frac{1}{n}} = \lim_{x\to 0} \frac{\left(\frac{1}{x}\right)^x - 1}{x}$$
$$= \lim_{x\to 0} \frac{e^{x \ln\left(\frac{1}{x}\right)} - 1}{x}.$$

Wegen $\lim_{x\to 0} x \ln\left(\frac{1}{x}\right) = \lim_{y\to\infty} \frac{\ln(y)}{y} = 0$ ist der Limes dieses Quotienten von der Form „$\frac{0}{0}$“. Wir wenden die Regel von l'Hospital an und differenzieren Zähler und Nenner.
Aus $\frac{d}{dx}\left(x \ln\left(\frac{1}{x}\right)\right) = \ln\left(\frac{1}{x}\right) - 1$ folgt

$$\frac{d}{dx}\left(e^{x \ln\left(\frac{1}{x}\right)} - 1\right) = e^{x \ln\left(\frac{1}{x}\right)} \left(\ln\left(\frac{1}{x}\right) - 1\right).$$

Wegen

$$\lim_{x\to 0} \frac{e^{x \ln\left(\frac{1}{x}\right)} \left(\ln\left(\frac{1}{x}\right) - 1\right)}{1} = \infty$$

gilt auch $\lim_{n\to\infty} n(\sqrt[n]{n} - 1) = \infty$ (was ja nicht offensichtlich ist).

Wir zeigen jetzt, dass $n(\sqrt[n]{n} - 1) = O(\ln(n))$ gilt.

$$\lim_{n\to\infty} \frac{n(\sqrt[n]{n} - 1)}{\ln(n)} = \lim_{n\to\infty} \frac{n^{\frac{1}{n}} - 1}{\frac{\ln(n)}{n}} = \lim_{x\to 0} \frac{e^{x \ln\left(\frac{1}{x}\right)} - 1}{x \ln\left(\frac{1}{x}\right)} \quad \left(= \text{„}\tfrac{0}{0}\text{“}\right).$$

Wegen

$$\lim_{x\to 0} \frac{e^{x \ln\left(\frac{1}{x}\right)} \left(\ln\left(\frac{1}{x}\right) - 1\right)}{\ln\left(\frac{1}{x}\right) - 1} = \lim_{x\to 0} e^{x \ln\left(\frac{1}{x}\right)} = 1$$

gilt auch

$$\lim_{n\to\infty} \frac{n(\sqrt[n]{n} - 1)}{\ln(n)} = 1.$$

Hieraus folgt $n(\sqrt[n]{n} - 1) = O(\ln(n))$.

$f_2 = O(n^k)$, denn

$$\binom{n}{k} k! \frac{1}{n^k} = \frac{n(n-1)\dots(n-(k-1))}{n \cdot n \dots n} = \left(1 - \frac{1}{n}\right) \dots \left(1 - \frac{k-1}{n}\right)$$

konvergiert gegen 1 für $n \longrightarrow \infty$.

7. Die Differenzengleichung

$$k_n = \left(1 + \frac{p}{100}\right) k_{n-1} + c, k_0 = k,$$

besitzt die Lösung

$$\begin{aligned} k_n &= \left(1 + \frac{p}{100}\right)^n \left(k + \sum_{i=1}^{n} c \left(1 + \frac{p}{100}\right)^{-i}\right) \\ &= k \left(1 + \frac{p}{100}\right)^n + c \sum_{i=1}^{n} \left(1 + \frac{p}{100}\right)^{n-i} \\ &= k \left(1 + \frac{p}{100}\right)^n + c \sum_{i=0}^{n-1} \left(1 + \frac{p}{100}\right)^{i} \\ &= k \left(1 + \frac{p}{100}\right)^n + \frac{100c \left(\left(1 + \frac{p}{100}\right)^n - 1\right)}{p}. \end{aligned}$$

8. a. Lösung der Gleichung:

$$\pi_n = \prod_{i=2}^{n} 1 = 1, n \geq 1,\ x_n = 1 + \sum_{i=2}^{n} i = \frac{n(n+1)}{2}.$$

b. Lösung der homogenen Gleichung:

$$\pi_n = \prod_{i=2}^{n} \frac{i+1}{i} = \frac{n+1}{2}, n \geq 1.$$

Lösung der Gleichung:

$$\begin{aligned} x_n &= \frac{n+1}{2} \sum_{i=2}^{n} \frac{4(i-1)}{i(i+1)} \\ &= 2(n+1) \sum_{i=2}^{n} \left(\frac{2}{i+1} - \frac{1}{i}\right) \\ &= 2(n+1) \left(\mathrm{H}_n + \frac{2}{n+1} - 2\right) \\ &= 2(n+1)\mathrm{H}_n - 4n. \end{aligned}$$

9.

$$x_1 = 0,\ x_n = \sum_{i=1}^{n-1} x_i + 2(n-1).$$

$$x_n - x_{n-1} = x_{n-1} + 2(n-1) - 2(n-2).$$

$$x_n = 2x_{n-1} + 2.$$

$$\pi_n = \prod_{i=2}^{n} 2 \;=\; 2^{n-1}.$$

$$x_n = 2^{n-1} \sum_{i=2}^{n} \frac{2}{2^{i-1}} \;=\; 2^n \sum_{i=1}^{n-1} \frac{1}{2^i} \;=\; \sum_{i=1}^{n-1} 2^i \;=\; 2^n - 2.$$

10.

$$x_1 = 2,\ x_n = 2n \cdot x_{n-1} + (n+1)!.$$

$$\pi_n = \prod_{i=2}^{n} 2i \;=\; 2^{n-1} n!.$$

$$x_n = 2^{n-1} n! \left(2 + \sum_{i=2}^{n} \frac{(i+1)!}{2^{i-1} i!} \right)$$

$$\begin{aligned}
&\sum_{i=2}^{n} \frac{i}{2^{i-1}} + \sum_{i=2}^{n} \frac{1}{2^{i-1}} \\
&= \frac{(n+1)\left(1/2\right)^n \left(-1/2\right) - \left(\left(1/2\right)^n - 1\right)}{1/4} - 1 + \frac{\left(1/2\right)^n - 1}{-1/2} - 1 \\
&= 4(-(n+1)\left(1/2\right)^{n+1} - \left(1/2\right)^{n+1} + 1) - 1 - 2\left(\left(1/2\right)^n - 1\right) - 1 \\
&= 4 - (n+3)\left(1/2\right)^{n-1}.
\end{aligned}$$

$$\begin{aligned}
x_n &= 2^{n-1} n! (2 + 4 - (n+3)\left(1/2\right)^{n-1}) \\
&= n!(3 \cdot 2^n - (n+3))
\end{aligned}$$

11. Wir setzen $n = 2^{2^k}$ und $x_k = T(2^{2^k})$. Dann ist $k = \log_2(\log_2(n))$.

$$\begin{aligned}
T(n) &= T(\sqrt{n}) + \log_2(n)^l,\ l = 0, 1. \\
x_k &= T(2^{2^k}) \;=\; T(2^{2^{k-1}}) + (2^l)^k. \\
x_k &= x_{k-1} + (2^l)^k,\ x_1 = 2^l. \\
x_k &= 2^l + \sum_{i=2}^{k} (2^l)^i \;=\; \sum_{i=1}^{k} (2^l)^i \;=\; \begin{cases} k & \text{für } l = 0, \\ 2^{k+1} - 2 & \text{für } l = 1. \end{cases}
\end{aligned}$$

Die Anwendung der inversen Transformation $k = \log_2(\log_2(n))$ ergibt

$$\begin{aligned}
T(n) &= x_{\log_2(\log_2(n))} \\
&= \begin{cases} \log_2(\log_2(n)) & \text{für } l = 0, \\ 2^{\log_2(\log_2(n))+1} - 2 = 2(\log_2(n) - 1) & \text{für } l = 1. \end{cases}
\end{aligned}$$

12. Wir lösen die Gleichung mit Satz 1.15

$$x_k = a^{k-1}(ad + cb^l) + a^{k-1}c\sum_{i=2}^{k}\frac{b^{li}}{a^{i-1}} \;=\; a^k d + ca^k \sum_{i=1}^{k}\left(\frac{b^l}{a}\right)^i$$

$$= \begin{cases} a^k d + cn^l \frac{q^k-1}{q-1}, & \text{falls } q \neq 1, \\ a^k d + ckn^l, & \text{falls } q = 1. \end{cases}$$

Die Anwendung der inversen Transformation $k = \log_b(n)$ ergibt

$$T_{(\frac{a}{b})}(n) = x_{\log_b(n)} = \begin{cases} da^{\log_b(n)} + cn^l \frac{q^{\log_b(n)}-1}{q-1}, & \text{falls } b^l \neq a, \\ da^{\log_b(n)} + cn^l \log_b(n), & \text{falls } b^l = a. \end{cases}$$

Beachte $\log_b(n) = \lfloor \log_b(n) \rfloor$ für $n = b^k$.

13. a = 1, l = 0:

$$T(n) = 1 + \lfloor \log_2(n) \rfloor = O(\log_2(n)).$$

a = 1, l = 1:

$$1 + n\left(1 - \left(\frac{1}{2}\right)^{\lfloor \log_2(n) \rfloor}\right) \leq T(n) \leq 1 + 2n\left(1 - \left(\frac{1}{2}\right)^{\lfloor \log_2(n) \rfloor}\right) = O(n)$$

a = 2, l = 0:

$$T(n) = 2^{\lfloor \log_2(n) \rfloor + 1} - 1 = O(n)$$

a = 2, l = 1:

$$2^{\lfloor \log_2(n) \rfloor} + \frac{n}{2}\lfloor \log_2(n) \rfloor \leq T(n) \leq 2^{\lfloor \log_2(n) \rfloor} + n\lfloor \log_2(n) \rfloor = O(n \log_2(n)).$$

14. Für die Anzahl der aktiven rekursiven Aufrufe gilt:

$$a_n = a_{n-1} + 1, a_1 = 1.$$

Diese Gleichung besitzt die Lösung $a_n = n$, d. h. auf dem Stack wird Platz für n Aufrufe benötigt.

15. Wir stellen die Aufrufhierarchie zur Lösung des Puzzles bei n Scheiben dar: Der Aufruf TowersOfHanoi(n, A, B, C) bewirkt den Aufruf TowersOfHanoi$(n-1, A, C, B)$, die Bewegung der auf A verbliebenen Scheibe nach B und den Aufruf TowersOfHanoi$(n-1, C, B, A)$.

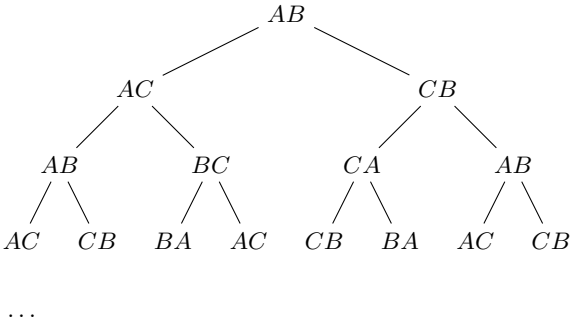


Fig. 2.1: Aufrufhierarchie von TowersOfHanoi.

Beachte die Reihenfolge in jeder Ebene:
$AB, BC, CA, \ldots$ bzw. $AC, CB, BA, \ldots$.
Wir erhalten die zur Lösung notwendigen Arbeitsschritte, wenn wir diesen Baum inorder traversieren. Für $n = 4$ ergibt sich zum Beispiel die Folge
$AC, AB, CB, AC, BA, BC, AC, AB, CB, CA, BA, CB, AC, AB, CB$.
Der erste Arbeitsschritt entspricht dem Knoten, der in der untersten Ebene ganz links steht. Er ist AB, falls n ungerade ist, und AC, falls n gerade ist. Wir nummerieren die Knoten in der inorder Besuchsreihenfolge.
Nummern: $1, 3, 5, \ldots$ sind Blätter, d. h. die Knoten der Ebene $n-1$.
Nummern: $2, 6, 10, \ldots$ sind die Knoten der Ebene $n-2$.
Nummern: $4, 12, 20, \ldots$ sind die Knoten der Ebene $n-3$.

Die Knoten verteilen sich auf die einzelnen Ebenen wie folgt:

Ebene	Erster	Abstand	alg. Form	Anzahl
$n-1$	1	2	$1 + j * 2, j = 0, \ldots, 2^{n-1} - 1$	2^{n-1}
$n-2$	2	2^2	$2 + j * 2^2, j = 0, \ldots, 2^{n-2} - 1$	2^{n-2}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-i$	2^{i-1}	2^i	$2^{i-1} + j * 2^i, j = 0, \ldots, 2^{n-i} - 1$	2^{n-i}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	2^{n-2}	2^{n-1}	$2^{n-2} + j * 2^{n-1}, j = 0, 1 = 2^1 - 1$	2
0	2^{n-1}		2^{n-1}	1

Ein Knoten mit der Nummer k ist genau dann in der Ebene i, wenn

$$k \bmod 2^{n-i-1} = 0 \text{ und } k \bmod 2^{n-i} \neq 0.$$

Dies bedeutet, dass in der Binärentwicklung von k die erste 1 von links an der Stelle $n - i$ steht.

Für $n = 4$ erhalten wir

AC	AB	CB	AC	BA	BC	AC	AB	CB	CA	BA	CB	AC	AB	CB
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	2	3	1	3	2	3	0	3	2	3	1	3	2	3

.

Die zweite Zeile gibt die Nummer des Knotens an und die dritte Zeile gibt die Ebene an, in der sich der Knoten befindet.

Ausgehend von einem Blatt k (k ungerade) kann der Nachfolger $k+1$ und der Nachfolger $k+2$ des Nachfolgers einfach bestimmt werden: Befindet sich der Knoten $k+1$ in der Ebene l, die nach der Formel von oben berechnet werden kann, dann ist der Knoten k im Baum einmal linker Nachfolger und $n-l-2$ mal rechter Nachfolger von $k+1$. Ausgehend von k geht man im Baum $n-l-2$ mal nach links und einmal nach rechts, um zum Vorgänger $k+1$ zu gelangen. Wenn es sich bei einem Knoten um einen linken Nachfolger handelt, wird bei der Beschriftung das erste Symbol vom Vater übernommen, das zweite Symbol wird geändert. Für einen rechten Nachfolger ist es gerade umgekehrt. Geht man auf dem Weg zum Vorgänger $n-l-2$ mal nach links, so wird das zweite Symbol beibehalten. Das erste Symbol wird geändert, wenn $n-l-2$ ungerade ist. Geht man einmal nach rechts, so wird das erste Symbol beibehalten und das zweite geändert.
Der Nachfolger $k+2$ des Nachfolgers $k+1$ ist das Blatt rechts neben k. Wendet man diese Regeln an, so erhalten wir die folgende Tabelle, die den Nachfolger und den Nachfolger des Nachfolgers angibt.

k	$k+1, n-l-2$ gerade	$k+1, n-l-2$ ungerade	$k+2$
AB	AC	CA	BC
AC	AB	BA	CB
BA	BC	CB	AC
BC	BA	AB	CA
CA	CB	BC	AB
CB	CA	AC	BA

16. Bei Anwendung des generischen Algorithmus (Algorithmus 1.34) für Matroide, ist in jedem Schritt zu entscheiden, ob $O \cup \{s_i\}$ zulässig ist. Dazu halten wir die Liste O der gewählten Aufgaben aufsteigend nach Abschlusszeiten sortiert, d. h. wir halten bei Hinzunahme der nächsten Aufgabe s_i die Sortierung ein. Wir stellen s_i an das Ende der Liste und fügen s_i analog InsertionSort (Algorithmus 1.57) in die Liste ein. Jetzt steht s_i als letztes Element mit gleicher Abschlusszeit.
Sei s_i das l_i–te Element in der sortierten Liste. $O \cup \{s_i\}$ ist genau dann zulässig, wenn $l_i \leq$ der Abschlusszeit von s_i ist. Für die Implementierung von O mit einem Array ist l_i der Index von s_i. Wird O mit einer verket-

teten Liste implementiert, so muss die Nummerierung der Reihenfolge explizit erfolgen.

17. Seien $n_l > n_{l-1} > n_1$ die Werte der Münzen ($n_1 = 1$). Gesucht ist $\nu_1, \ldots, \nu_l \geq 0$ mit

$$\sum_{i=1}^{l} \nu_i n_i = n \text{ und } \sum_{i=1}^{l} \nu_i \text{ minimal.}$$

Greedy-Strategie: Wir setzen $r_l = n$ und berechnen nacheinander für $i = l \ldots 1$

$$\nu_i = \left\lfloor \frac{r_i}{n_i} \right\rfloor, \; r_{i-1} = r_i \bmod n_i \; = \; r_i - \nu_i n_i.$$

Verfügbare Münzen: 1 Cent, 2 Cent, 5 Cent, 10 Cent, 20 Cent, 50 Cent, 100 Cent, 200 Cent
Für eine Lösung, die nach obigem Algorithmus erzeugt wurde, gilt:

$$\begin{aligned} &\nu_1 \leq 1, \nu_2 \leq 2, \nu_3 \leq 1, \\ &\nu_4 \leq 1, \nu_5 \leq 2, \nu_6 \leq 1, \\ &\nu_7 \leq 1. \end{aligned}$$

Eine optimale Lösung erfüllt auch alle diese Bedingung, denn falls eine der Bedingungen verletzt ist, kann eine Lösung mit weniger Münzen gefunden werden.
Diese Bedingungen bestimmen eine Lösung eindeutig. $n \bmod 5 = \nu_1 + 2\nu_2$. Deshalb sind ν_1 und ν_2 eindeutig bestimmt ($n \bmod 5 = 0$: $\nu_1 = 0, \nu_2 = 0$, $n \bmod 5 = 1$: $\nu_1 = 1$, $n \bmod 5 = 2$: $\nu_2 = 1$, $n \bmod 5 = 3$: $\nu_1 = 1, \nu_2 = 1$, $n \bmod 5 = 4$: $\nu_2 = 2$). ν_3 ist genau dann 0, wenn $(n - \nu_1 - 2\nu_2) \bmod 10 = 0$. Deshalb ist auch ν_3 eindeutig bestimmt. ν_4 ist genau dann 0, wenn $(n - \nu_1 - 2\nu_2 - 5\nu_3) \bmod 20 = 0$. Deshalb ist auch ν_4 eindeutig bestimmt. $(n - \nu_1 - 2\nu_2 - 5\nu_3 - 10\nu_4) \bmod 50$ bestimmt ν_5 eindeutig. $(n - \nu_1 - 2\nu_2 - 5\nu_3 - 10\nu_4 - 20\nu_5) \bmod 100$ bestimmt ν_6 eindeutig. $(n - \nu_1 - 2\nu_2 - 5\nu_3 - 10\nu_4 - 20\nu_5 - 50\nu_6) \bmod 200$ bestimmt ν_7 eindeutig. Damit ist auch ν_8 eindeutig bestimmt. Der obige Algorithmus liefert eine optimale Lösung.
(Für $n_3 = 11, n_2 = 5, n_1 = 1$ führt die Greedy-Strategie nicht zum Erfolg ($n = 15$)).

18. Wir setzen die rekursive Formel zur Lösung der Editierdistanz in einen rekursiven Algorithmus P(i, j) um. Sei $T(i, j)$ die Anzahl der notwendigen Aufrufe, um mit dem Algorithmus P $d(i, j)$ zu berechnen. Dann gilt:

$$\begin{aligned} &T(0,0) = T(i,0) \; = \; T(0,j) \; = 1, \\ &T(i,j) = T(i,j-1) + T(i-1,j) + T(i-1,j-1) + 1. \end{aligned}$$

Es folgt

$$T(n,m) \geq 3T(n-1,m-1) \geq 3^k, \; k = \min\{n,m\}.$$

Obwohl es nur $(n+1)\cdot(m+1)$ viele Teilprobleme gibt, ist die Laufzeit exponentiell. Dies liegt daran, dass dieselben Distanzen immer wieder berechnet werden. Für große n und m kann der Algorithmus nicht eingesetzt werden.

19. Die Länge $l(n,m)$ berechnet sich rekursiv

$$\begin{aligned} &l(0,0)=0, l(i,0) = 0, l(0,j) = 0,\\ &l(i,j)=l(i-1,j-1)+1, \text{ falls } i,j>0 \text{ und } a_i=b_j,\\ &l(i,j)=\max\{l(i,j-1),l(i-1,j)\}, \text{ falls } i,j>0 \text{ und } a_i\neq b_j. \end{aligned}$$

Wende dynamisches Programmieren an, um $l(n,m)$ zu berechnen.

20. Sei $m_n=\max_{i,j\leq n} f(i,j)$. Wir definieren Folgen M_k, N_k durch:

$$\begin{aligned} M_1 &= N_1 = a_1,\\ N_k &= \max\{N_{k-1}+a_k, a_k\}, k\geq 2,\\ M_k &= \max\{M_{k-1}, N_k\}, k\geq 2. \end{aligned}$$

Durch Induktion nach k folgt:

$$N_k=\max_i f(i,k) \text{ und } M_k=m_k=\max_{i,j\leq k} f(i,j).$$

Die Aussage ist richtig für $k=1$. Schluss von $k-1$ auf k:

$$N_k=\max\{N_{k-1}+a_k,a_k\}=\max\{\max_i f(i,k-1)+a_k,a_k\}=\max_i f(i,k).$$

Wir unterscheiden die beiden Fälle:
a. a_k tritt bei der Berechnung von m_k nicht auf.
   Dann gilt $m_k=m_{k-1}=M_{k-1}=M_k$.
b. a_k tritt bei der Berechnung von m_k auf.
   Dann gilt $m_k=\max_i f(i,k)=N_k=M_k$.

Wir berechnen die Folgen $(N_k)_{k=1,\ldots,n}$ und $(M_k)_{k=1,\ldots,n}$ mit dynamischem Programmieren.

21. Wir folgen mit unserer Lösung [MotRag95, Seite 171]. Wir nehmen ohne Einschränkung an, dass $a,b\in\{0,1\}^*$. Wir betrachten $a_i\ldots a_{i+m-1}$ und $b_1\ldots b_m$ als Binärdarstellungen von Zahlen mit der höchstwertigen Stelle links. Die Zeichenkette b ist ein Teilstring von a, wenn $a_i\ldots a_{i+m-1}=b_1\ldots b_m$ für ein i, $1\leq i\leq n-m+1$. Das Problem durch vollständige Suche zu lösen besitzt eine Laufzeit in der Ordnung $O(mn)$. Eine effizientere Lösung kann durch die Verwendung von Fingerabdrücken entwickelt werden.
Wir wählen eine Familie von Hashfunktionen

$$h_p:\{0,1\}^l\longrightarrow\{0,\ldots,p-1\},\ x\longmapsto x \bmod p,$$

wie im Abschnitt 1.6.2. Die Kollisionswahrscheinlichkeit für eine zufällig gewählte Primzahl p ist für $x \neq y$

$$\mathrm{p}(h_p(x) = h_p(y)) \leq \frac{1}{t}.$$

(Satz 1.51). Wir vergleichen jetzt nicht mehr Teilstrings $a_i \dots a_{i+m-1}$ von a und $b_1 \dots b_m$, sondern deren Hashwerte

$$h_p(a_i \dots a_{i+m-1}) = h_p(b_1 \dots b_m).$$

Es gilt

$$a_{i+1} \dots a_{i+m} = 2(a_i \dots a_{i+m-1} - a_i 2^{m-1}) + a_{i+m}$$

und es folgt

$$h_p(a_{i+1} \dots a_{i+m}) = 2(h_p(a_i \dots a_{i+m-1}) - a_i 2^{m-1}) + a_{i+m} \bmod p.$$

$h_p(a_{i+1} \dots a_{i+m})$ kann aus $h_p(a_i \dots a_{i+m-1})$ in konstanter Zeit berechnet werden. Wir erhalten einen Algorithmus mit der Laufzeit $O(n+m)$. Für $h_p(a_i \dots a_{i+m-1}) = h_p(b_1 \dots b_m)$ kann in der Zeit $O(m)$ geprüft werden, ob eine Kollision vorliegt.

2. Sortieren

1. a. **Algorithmus 2.1.**

```
Sort2(item a[1..n])
 1  index l, r, boolean loop ← true
 2  l ← 1, r ← n, a[0] ← 1, a[n + 1] ← 2
 3  while loop do
 4       while a[l] = 1 do l ← l + 1
 5       while a[r] = 2 do r ← r − 1
 6       if l < r
 7         then exchange a[l] and a[r]
 8              l = l + 1, r = r − 1
 9         else loop ← false
```

Zunächst werden die Datensätze mit der 3 analog zu Sort2 an das Ende des Arrays gestellt. Anschließend wird Sort2 auf den Teil des Arrays, das nur 1 und 2 enthält angewendet.

b. **Algorithmus 2.2.**

```
Sort(item a[1..n])
1  index i = 1, j
2  while i ≤ n do
3       j ← a[i].key
4       if i = j
5         then i ← i + 1
6         else exchange a[i] and a[j]
```

2. Sei a_i die Anzahl der Ausführungen von Zeile i im schlechtesten Fall und $\tilde{a}_i$ die Anzahl der Ausführungen von Zeile i im Durchschnitt – jeweils in Abhängigkeit von n.

   a. Sortieren durch Einfügen (Algorithmus 1.57).
   Vergleiche in Zeile 4:

$$a_4 = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1.$$

$\tilde{a}_4$ = Anzahl der Inversionen + $(n-1)$:

$$\tilde{a}_4 = \frac{n(n-1)}{4} + (n-1) = \frac{n^2}{4} + \frac{3n}{4} - 1.$$

$$a_5 = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

$\tilde{a}_5$ = Anzahl der Inversionen im Mittel:

$$\tilde{a}_5 = \frac{n(n-1)}{4} = \frac{n^2}{4} - \frac{n}{4}.$$

Wir fassen die Ergebnisse zusammen:

Zeile i	a_i	$\tilde{a}_i$
$3, 6$	$n-1$	$n-1$
4	$n^2/2 + n/2 - 1$	$n^2/4 + 3n/4 - 1$
5	$n^2/2 - n/2$	$n^2/4 - n/4$

a_3 und a_6 hängen nicht von der Anordnung der Elemente in a ab. Deshalb gilt $a_i = \tilde{a}_i$, $i = 3, 6$. Der schlechteste Fall tritt für ein umgekehrt sortiertes Array ein. In diesem Fall wird Zeile 5 jedes Mal ausgeführt.

b. Bubble-Sort (Algorithmus 2.32).

$$a_4 = \tilde{a}_4 = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

$$a_5 = a_4 = \frac{n^2}{2} - \frac{n}{2}.$$

$\tilde{a}_5$ = Anzahl der Inversionen im Mittel:

$$\tilde{a}_5 = \frac{n(n-1)}{4} = \frac{n^2}{4} - \frac{n}{4}.$$

Wir fassen die Ergebnisse zusammen:

Zeile i	a_i	$\tilde{a}_i$
4	$n^2/2 - n/2$	$n^2/2 - n/2$
5	$n^2/2 - n/2$	$n^2/4 - n/4$

Der schlechteste Fall tritt für ein umgekehrt sortiertes Array ein. In diesem Fall wird Zeile 5 jedes Mal ausgeführt.

Wir erhalten für die Anzahl V_w, der Vergleiche im schlechtesten Fall, die Anzahl $V_\emptyset$, der Vergleiche im Durchschnitt, die Anzahl Z_w, der Zuweisungen im schlechtesten Fall und die Anzahl $Z_\emptyset$, der Zuweisungen im Durchschnitt:

	SelectionSort	InsertionSort	BubbleSort
V_w	$n^2/2 - n/2$	$n^2/2 + n/2 - 1$	$n^2/2 - n/2$
$V_\emptyset$	$n^2/2 - n/2$	$n^2/4 + 3n/4 - 1$	$n^2/2 - n/2$
Z_w	$n^2/2 - 7n/2 - 4$	$n^2/2 + 5n/2 - 3$	$3n^2/2 - 3n/2$
$Z_\emptyset$	$(n+1)\mathrm{H}_n + 2n - 4$	$n^2/4 + 11n/4 - 3$	$3n^2/4 - 3n/4$

3. InsertionSort und BubbleSort sind stabil.
   SelectionSort ist nicht stabil. $2_1, 2_2, 1$ wird zu $1, 2_2, 2_1$. SelectionSort kann stabil implementiert werden, falls alle Elemente von der Einfügestelle bis zur Position vor dem Minimum um eine Position nach hinten verschoben werden. Dadurch verschlechtert sich die Performance erheblich.
   Quicksort und Heapsort sind nicht stabil.

4. Die while-Schleife im Algorithmus 2.1 terminiert mit $l = r$ genau dann, wenn $a[r] \leq x$ und $a[l] \geq x$ gilt, d. h. $a[l] = a[r] = x$ $(= a[j])$. Falls in a alle Elemente paarweise verschieden sind, terminiert die while-Schleife im Algorithmus 2.1 mit $l = r + 1$. Es finden somit $n + 1$ viele Vergleiche statt. Wir erhalten

$$\tilde{V}(n, i) = V(i - 1) + V(n - i) + n + 1.$$

   $V(n)$ ergibt sich als Mittelwert über $\tilde{V}(n, i)$:

$$\begin{aligned} V(n) &= \frac{1}{n} \sum_{i=1}^{n} \tilde{V}(n, i) \\ &= \frac{1}{n} \sum_{i=1}^{n} (V(i - 1) + V(n - i) + n + 1) \\ &= \frac{2}{n} \sum_{i=0}^{n-1} V(i) + n + 1, n \geq 2, \end{aligned}$$

   Wir setzen

$$x_n = \sum_{i=0}^{n} V(i).$$

   Dann gilt

$$x_n - x_{n-1} = \frac{2}{n} x_{n-1} + n + 1.$$

   Wir erhalten die Differenzengleichung

$$\begin{aligned} x_1 &= V(0) + V(1) = 0, \\ x_n &= \frac{n + 2}{n} x_{n-1} + n + 1, n \geq 2. \end{aligned}$$

   Diese Gleichung besitzt die Lösung

$$x_n = (n + 1)(n + 2) \left(\mathrm{H}_{n+1} + \frac{1}{n + 2} - \frac{11}{6} \right)$$

   (Seite 17, Gleichung (D 1)). Wir erhalten

$$V(n) = \frac{2}{n} x_{n-1} + n + 1 = 2(n + 1)\mathrm{H}_n - \frac{8n + 2}{3}.$$

5. Der Aufruf von exchange in Zeile 11 ist überflüssig, falls das Pivotelement das größte Element ist. Dies ist mit Wahrscheinlichkeit $\frac{1}{n}$ der Fall. Die durchschnittliche Anzahl der Umstellungen (ohne Rekursion) gemittelt über alle Pivotelemente ist

$$\frac{1}{n}\sum_{i=1}^{n}\left(\frac{(i-1)(n-i)}{n-1}+1-\frac{1}{n}\right)=\frac{(n+4)}{6}-\frac{1}{n}.$$

Wir setzen $x_n=\sum_{i=0}^{n}U(i)$ und erhalten

$$\begin{aligned}x_1&=U(0)+U(1)=0,\\ x_n&=\frac{n+2}{n}x_{n-1}+\frac{n+4}{6}-\frac{1}{n}, n\geq 2.\end{aligned}$$

Diese Gleichung besitzt die Lösung

$$\begin{aligned}x_n&=\frac{(n+1)(n+2)}{6}\left(\sum_{i=2}^{n}\frac{i+4}{(i+1)(i+2)}-\sum_{i=2}^{n}\frac{6}{i(i+1)(i+2)}\right)\\ &=\frac{(n+1)(n+2)}{6}\left(\mathrm{H}_{n+1}-\frac{5}{n+2}+\frac{3}{n+1}-\frac{4}{3}\right).\end{aligned}$$

Partialbruchzerlegung:

$$\frac{6}{i(i+1)(i+2)}=\frac{3}{i}-\frac{6}{i+1}+\frac{3}{i+2}.$$

Wir erhalten

$$\begin{aligned}U(n)&=\frac{2}{n}x_{n-1}+\frac{n+4}{6}-\frac{1}{n}\\ &=\frac{n+1}{3}\left(\mathrm{H}_n-\frac{5}{n+1}+\frac{3}{n}-\frac{4}{3}\right)+\frac{n+4}{6}-\frac{1}{n}\\ &=\frac{1}{3}(n+1)\mathrm{H}_n-\frac{5n+8}{18}.\end{aligned}$$

6. Zu lösen ist

$$T(n)=cn+\frac{2}{n}\sum_{i=0}^{n-1}T(i), n\geq 2, T(0)=T(1)=b.$$

Ziel ist, diese Rekursion durch eine geeignete Substitution in eine Differenzengleichung zu transformieren. Bei Rekursionen, bei denen das n–te Glied von der Summe aus allen Vorgängern abhängt, gelingt dies mit der Substitution

$$x_n=\sum_{i=0}^{n}T(i).$$

Dann gilt

$$x_n - x_{n-1} = cn + \frac{2}{n}x_{n-1}.$$

Hieraus folgt

$$x_n = \frac{n+2}{n}x_{n-1} + cn, n \geq 2, x_1 = 2b.$$

Die Lösung der Gleichung ist

$$x_n = (n+1)(n+2)\left(c\mathrm{H}_{n+1} + \frac{2c}{n+2} + \frac{1}{6}(2b-13c)\right).$$

(Gleichung (D 1) auf Seite 17).
Aus der Lösung für x_n kann einfach eine Lösung für $T(n)$ abgeleitet werden.

$$\begin{aligned} T(n) &= \frac{2}{n}x_{n-1} + cn \\ &= \frac{2}{n}n(n+1)\left(c\mathrm{H}_n + \frac{2c}{n+1} + \frac{1}{6}(2b-13c)\right) + cn \\ &= 2c(n+1)\mathrm{H}_n + \frac{1}{3}(2b-10c)n + \frac{1}{3}(2b-c) \end{aligned}$$

ist eine geschlossene Lösung für $T(n)$.

7. a. Die Anzahl der Vergleiche ist $n-1$. Diese Optimierung erfolgt zulasten der Anzahl der Umstellungen. Der Code ist bestechend einfach.
   b. Invariante der for-Schleife:

$$a[i], \ldots, a[l-1] \leq x \text{ und } a[l], \ldots, a[k-1] > x.$$

Dies gilt für $k = i$, denn mit $k = i$ gilt $l = i$ und für leere Arrays ist die Aussage trivialerweise richtig.
Schluss von $k-1$ auf k:
Gilt $a[k] \leq x$, so werden $a[k]$ und $a[l]$ vertauscht. Nach Inkrementieren von l gilt $a[i], \ldots, a[l-1]$ sind $\leq x$ und nach Inkrementieren von k gilt $a[l], \ldots, a[k-1]$ sind $> x$.
Gilt $a[k] > x$, so bleibt l unverändert und nach Inkrementieren von k gilt $a[l], \ldots, a[k-1]$ sind $> x$.
Die Invariante gilt auch nach Terminierung der for-Schleife. Nach Ausführung von exchange in Zeile 8 befindet sich das Pivotelement an der Position l und es gilt $a[i], \ldots, a[l-1]$ sind $\leq x$ und $a[l+1], \ldots, a[j]$ sind $> x$.
QuickSortVariante sortiert das Array a in aufsteigender Reihenfolge. Die Teilarrays für die QuickSortVariante aufgerufen wird haben höchstens $n-1$ viele Elemente, wenn n die Anzahl der Elemente von a ist. Der Beweis folgt unmittelbar mit vollständiger Induktion.

8. Bei n Elementen muss das Pivotelement nur einmal mit jedem der $n-1$ übrigen Elemente verglichen werden. Dazu müssen die Indizes kontrolliert werden. Ein Vergleich ist nur notwendig, falls $l < r$ gilt und ein Vergleich für $l = r$. Wir erhalten

$$V(1) = 0, V(n) = V\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + V\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + n - 1.$$

Es gilt

$$V(n) \leq 2V\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1.$$

Wir ersetzen $\leq$ durch $=$ und lösen die Rekursion

$$V(n) = 2V\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1.$$

Sei $n = n_{k-1} \dots n_0$, $n_{l-1} = 1$, die Binärentwicklung von n. Dann ist $k = \lfloor \log_2(n) \rfloor + 1$ (Lemma B.3).

$$\begin{aligned} V(n) &= \sum_{i=0}^{k-2} 2^i \left(\left\lfloor \frac{n}{2^i} \right\rfloor - 1 \right) \leq (k-1)n - (2^{k-1} - 1) \\ &= n\lfloor \log_2(n) \rfloor - 2^{\lfloor \log_2(n) \rfloor} + 1 \end{aligned}$$

(Lemma 1.25 mit $g^i(n) = \lfloor \frac{n}{2^i} \rfloor$, $r(n) = n - 1$ und $d = 0$).

9. **Anmerkungen zum Algorithmus.**
   a. Pivot gibt einen Index $i \neq 0$ zurück, falls sich im Array mindestens zwei verschiedene Elemente befinden. In $a[i]$ ist das Pivotelement gespeichert und es befindet sich ein Element im Array, welches kleiner als $a[i]$ ist.
   b. Der Zerlegungsprozess ist nur notwendig, falls der Rückgabewert von Pivot > 0 ist.
   c. Der erste exchange (Zeile 5) ist nicht notwendig (ergibt aber einfachen Code).
   d. Nach Terminierung von repeat-until gilt:
   $a[i], \dots, a[l-1] < x$ und $a[l], \dots, a[j] \geq x$.

Das zu sortierende Array enthalte lauter verschiedene Elemente.
Setze $n := j - i + 1$. Aufwand für die Zeilen 2 - 8 (QuickSort): cn, c konstant. Das Pivotelement befindet sich an der ersten oder zweiten Stelle. Jede der $n-1$ Zerlegungen besitzt die Wahrscheinlichkeit $1/(n-1)$. Wird in Arrays der Länge r und $n-r$ zerlegt so ergibt sich für die Anzahl $T(n)$ der Operationen rekursiv:

$$T(n) = T(r) + T(n-r) + cn, c \text{ konstant.}$$

Durchschnittliche Laufzeit Für die durchschnittliche Laufzeit T(n) ergibt sich (gemittelt über alle Zerlegungen):

$$T(n) = cn + \frac{1}{n-1}\sum_{r=1}^{n-1}(T(r) + T(n-r)) = cn + \frac{2}{n-1}\sum_{r=1}^{n-1} T(r), n \geq 2$$

Zu lösen:

$$T(1) = b, T(n) = cn + \frac{2}{n-1}\sum_{i=1}^{n-1} T(i), n \geq 2$$

Setze:

$$x_n = \sum_{i=1}^{n} T(i).$$

Dann gilt

$$x_n - x_{n-1} = cn + \frac{2}{n-1}x_{n-1}.$$

Wir erhalten die Gleichung

$$x_1 = b,\ x_n = \frac{n+1}{n-1}x_{n-1} + cn, n \geq 2.$$

Für diese Gleichung gilt:

$$\pi_n = \prod_{i=2}^{n} \frac{i+1}{i-1} = \frac{n(n+1)}{2}, n \geq 2, \pi_1 = 1.$$

$$\begin{aligned} x_n &= \frac{n(n+1)}{2}\left(b + \sum_{i=2}^{n} ci\frac{2}{i(i+1)}\right) \\ &= \frac{n(n+1)}{2}\left(b + 2c\sum_{i=2}^{n}\frac{1}{i+1}\right) \\ &= \frac{n(n+1)}{2}(2c\mathrm{H}_{n+1} - 3c + b). \end{aligned}$$

Somit ergibt sich für $T(n)$:

$$\begin{aligned} T(n) &= \frac{2}{n-1}x_{n-1} + cn \\ &= \frac{2}{n-1}\frac{(n-1)n}{2}(2c\mathrm{H}_n - 3c + b) + cn \\ &= 2cn(\mathrm{H}_n - 1) + bn \end{aligned}$$

Laufzeit im schlechtesten Fall Der schlechteste Fall tritt ein, wenn in jedem Rekursionsschritt der Zerlegungsprozess eine einelementige und eine $(n-1)$–elementige Menge liefert.

$$T(n) = T(n-1) + T(1) + cn, T(1) = b.$$

besitzt als Lösung

$$T(n) = \left(b + \sum_{i=2}^{n} (c \cdot i + b) \right) = c \left(\frac{n(n+1)}{2} - 1 \right) + bn.$$

Der schlechteste Fall tritt bei der angegebenen Pivotprozedur für ein sortiertes Array ein.

10. **Algorithmus 2.3.**

```
QuickSort(item a[i..j])
 1  item x, index l, r
 2  while i < j do
 3      if p ← Pivot(a[i..j]) ≠ 0
 4        then x ← a[p], l ← i, r ← j
 5             repeat
 6                 exchange a[l] and a[r]
 7                 while a[l] < x do l ← l + 1
 8                 while a[r] ≥ x do r ← r − 1
 9             until l = r + 1
10             QuickSort(a[i..l − 1])
11             i ← l
12        else i ← j
```

Wir zeigen, dass die while-Schleife immer terminiert. Wir betrachten zwei Fälle:
1. Falls $\text{Pivot}(a, i, j) \neq 0$ ist, wird die Zerlegung durchgeführt. Nach der Zerlegung wird $i = l$ gesetzt. i wird somit um mindestens 1 erhöht.
2. Falls $\text{Pivot}(a, i, j) = 0$ ist, sind alle Elemente in $a[i..j]$ gleich und somit sortiert. Es wird $i = j$ gesetzt die while-Schleife und QuickSort terminieren. Dieser Fall tritt spätestens für $i = j - 1$ ein.
Die Betrachtung zeigt, dass die while-Schleife und QuickSort immer terminieren.
Es entsteht eine Folge von Anfangspunkten: $i_1 = i < i_2 < \ldots < i_n$. i_k bzw. $\tilde{i}_k$ bezeichne den Wert von i bei der k–ten Iteration der while-Schleife beim Schleifeneintritt bzw. beim Schleifenaustritt. Durch Induktion nach k folgt, dass $a[i, \tilde{i}_k]$ beim Schleifenaustritt sortiert ist. Nach Terminierung ist somit $a[i..j]$ sortiert.
Bei dieser Version von Quicksort werden dieselben Zerlegungen vorgenommen, wie in Algorithmus 2.34. Deshalb ergibt sich dieselbe Formel für die Laufzeit.
QuickSort **mit logarithmisch beschränkter Rekursionstiefe.** Die Idee besteht darin, zuerst den keineren Teil der Zerlegung rekursiv zu verarbeiten. Nach jedem rekursiven Aufruf nimmt die Anzahl der Elemente um mehr als die Hälfte ab. Dadurch ergibt sich logarithmisch beschränkte Rekursionstiefe.

11. Siehe [Ďurian86].

12. Verwende Heapsort und stoppe nach k–Schritten.

13. Die Anzahl der Zuweisungen ist bei allen Varianten gleich.
Zuweisungen in BuildHeap: Anzahl der Ausführungen von Zeile 2 + Anzahl der Ausführungen von Zeile 9 + Anzahl der Ausführungen von Zeile 10 in DownHeap:

$$2 \cdot \left\lfloor \frac{n}{2} \right\rfloor + I_1(n).$$

Zuweisungen in der Sortierphase: Anzahl der Ausführungen von Zeile 2 + Anzahl der Ausführungen von Zeile 9 + Anzahl der Ausführungen von Zeile 10 in DownHeap + 3 mal Anzahl der Ausführungen von Zeile 5 in Heapsort:

$$2(n-1) + I_2(n) + 3(n-1) = 5(n-1) + I_2(n).$$

14. Wir betrachten ein Array mit $n = 2^k - 1$ vielen Elementen das umgekehrt sortiert ist. In diesem Fall wird die obere Schranke angenommen. Betrachte zum Beispiel 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1. Jedes Element landet beim Einsickern in einem Blatt. Somit wird die maximale Anzahl der Iterationen

$$I(n) = \sum_{l=1}^{\lfloor \frac{n}{2} \rfloor} \left\lfloor \log_2 \left(\frac{n}{l} \right) \right\rfloor.$$

erreicht.

15. Mit Mergesort kann ohne zusätzlichen Speicher sortiert werden. Beim Zusammenfügen (mergen) der sortierten Teilarrays $a[i,l]$ und $a[l+1,j]$ sind, falls ein Element aus dem zweiten Teil einzufügen ist, die Elemente zwischen Einfügestelle und Entnahmestelle um eine Position nach hinten zu verschieben.
Mit $temp\,[1..\lfloor n/2 \rfloor]$ als Hilfsspeicher ist dies nicht notwendig. Kopiere den ersten Teil zunächst in den Hilfsspeicher und füge die beiden Teile in a zusammen.

In Merge finden höchstens $n-1$ Vergleiche statt. Sei $V(n)$ die Anzahl der Vergleiche. Dann gilt

$$V(n) \leq V\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + V\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1,\ V(1) = 0.$$

Wir ersetzen in der Rekursion $\leq$ durch $=$ und erhalten eine obere Schranke für die Anzahl der Vergleiche.
Setze $n = 2^k$ und $x_k = V(2^k)$. Dann gilt $V(2^k) = 2V(2^{k-1}) + 2^k - 1$ und $V(2) = 1$. Wir erhalten die Differenzengleichung

$$x_k = 2x_{k-1} + 2^k - 1,\ x_1 = 1.$$

Diese besitzt die Lösung

$$x_k = 2^{k-1}\left(1 + \sum_{i=2}^{k} \frac{2^i - 1}{2^{i-1}}\right) = (k-1)2^k + 1$$

(Satz 1.15). Mit $k = \log_2(n)$ folgt

$$V(n) = V(2^{\log_2(n)}) = x_{\log_2(n)} = (\log_2(n) - 1)n + 1 = n\log_2(n) - (n-1).$$

Wegen $\log_2(n) = \lfloor \log_2(n) \rfloor$ für $n = 2^k$ ist

$$V(n) = n\lfloor \log_2(n) \rfloor - (2^{\lfloor \log_2(n) \rfloor} - 1)$$

eine ganzzahlige Lösung (Lemma B.24).

16. Diese Version findet das erste Element unter gleichen, der Algorithmus 2.28 findet ein beliebiges unter gleichen. Es gilt
    a.
    $$l \leq \left\lfloor \frac{l+r}{2} \right\rfloor \leq r.$$
    b. Da $r_{i+1} - l_{i+1} < r_i - l_i$ gilt, terminiert while mit $l = r$.
    c. Nach einer Teilung hat der größere Teil $\left\lfloor \frac{n-1}{2} + 1 \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$ viele Elemente. $I(n)$ genügt daher Rekursion:
    $$I(n) \leq I\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1, I(1) = 0.$$
    Mit Satz 1.28 folgt, dass $I(n) \leq \lfloor \log_2(n) \rfloor + 1$ gilt.
17. Bestimme ein Element x der Ordnung k mit Quickselect. Führe anschließend einen Partitionierungsschritt von Quicksort mit Pivotelement x durch. Zerlege so, dass links von x Elemente $\leq x$ und rechts von x Elemente $> x$ stehen (analog zu Algorithmus 2.34). Die k kleinsten Elemente liegen links von x.

3. Hashverfahren

1. Eine Abbildung f ist durch ihren Wertevektor $(f(0), \ldots, f(n-1))$ gegeben. Somit gilt $\mathcal{F}(M,N) \cong N^m$ und $|\mathcal{F}(M,N)| = |N^m| = n^m$. Eine injektive Abbildung f ist durch ihren Wertevektor $(f(0), \ldots, f(m-1))$, $f(i) \neq f(j)$ für $i \neq j$ gegeben. Die Anzahl der injektiven Abbildungen ist

$$\binom{n}{m} \cdot m! = \frac{n!}{(n-m)!}$$

   Der Prozentsatz der injektiven Abbildungen ist somit

$$\frac{n!}{(n-m)! \cdot n^m} \cdot 100.$$

   Für $n = m$ ist der Prozentsatz der injektiven Abbildungen $\frac{n!}{n^n} \cdot 100 \approx \frac{\sqrt{2\pi n}}{e^n} \cdot 100$.[11] Für $n = 100$ ist der Prozentsatz $\approx 10^{-40}$. Injektive Abbildungen treten selten auf.

2. Wir berechnen die Werte für einen 16-Bit Rechner und 11-Bit Hashwerte. Dazu entwickeln wir 16 Stellen von $(0.618)_d$ binär: $(0.1001111000110101)_b$. Die Hashwerte sind:

k	binär	dezimal
0	10011110001	1265
1	00111100011	483
2	01111000110	966
3	11110001101	1933
4	11100011010	1818
5	11000110101	1589
6	10001101010	1130
7	00011010100	212
8	00110101000	428
9	01101010000	848
10	11010100110	1696

3. Seien $(x_1, y_1), (x_2, y_2) \in \mathbb{Z}_p \times \mathbb{Z}_p, (x_1, y_1) \neq (x_2, y_2)$ gegeben.
   Aus $h_a = h_{\tilde{a}}$ folgt $ax + y = \tilde{a}x + y$ für alle $x, y \in \mathbb{Z}_p$. Somit gilt $ax = \tilde{a}x$ $(y = 0)$ und $a = \tilde{a}$ $(y = 0, x = 1)$. Also : $|\mathcal{H}| = p$.
   Aus $ax_1 + y_1 = ax_2 + y_2$ folgt $a(x_1 - x_2) = y_2 - y_1$ und für $x_1 \neq x_2$ folgt $a = (y_2 - y_1)(x_1 - x_2)^{-1}$, d. h. $|\{h_a \in \mathcal{H} \mid h_a(x_1, y_1) = h_a(x_2, y_2)\}| = 1$.
   Für $x_1 = x_2$ folgt $y_1 = y_2$, ein Widerspruch zur Annahme $(x_1, y_1) \neq (x_2, y_2)$. Also gilt $\mathrm{p}(h_a(x_1, y_1) = h_a(x_2, y_2)) = 1/p$.

[11] Hier geht die Stirlingsche Formel $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ ein.

4. Sei $\mathrm{Lin}(\mathbb{Z}_p^k, \mathbb{Z}_p^l)$ die Menge der linearen Abbildungen $A : \mathbb{Z}_p^k \longrightarrow \mathbb{Z}_p^l$. Die linearen Abbildungen werden durch $l \times k$–Matrizen dargestellt. Somit gilt $\mathrm{Lin}(\mathbb{Z}_p^k, \mathbb{Z}_p^l) \cong M(l \times k, \mathbb{Z}_p)$ und $|\mathrm{Lin}(\mathbb{Z}_p^k, \mathbb{Z}_p^l)| = |M(l \times k, \mathbb{Z}_p)| = p^{kl}$.
   Sei $A \in M(l \times k, \mathbb{Z}_p)$ und seien $x_1, x_2 \in \mathbb{Z}_p^k$, $x_1 \neq x_2$, mit $A(x_1) = A(x_2)$. Dann gilt $A(x_1 - x_2) = 0$. Sei $y \in \mathbb{Z}_p^k \setminus \{0\}$. Wir berechnen die Anzahl der Matrizen $A \in M(l \times k, \mathbb{Z}_p)$ mit $A(y) = 0$. Da $(y_1, \ldots, y_k) \neq 0$ gilt, gibt es ein $y_i \neq 0$. Für die Spalten $A_1, \ldots, A_k$ von A gilt $A_i = -\frac{1}{y_i} \sum_{k \neq i} y_k A_k$. Es folgt
   $$|\{A \in M(l \times k, \mathbb{Z}_p) \mid A(x_1) = A(x_2)\}| = p^{(k-1)l}$$
   Insgesamt folgt, dass die Familie $\mathrm{Lin}(\mathbb{Z}_p^k, \mathbb{Z}_p^l)$ eine universelle Familie von Hashfunktionen ist.

5. Behauptung: Die Elemente aus $\{j \cdot c \bmod m \mid 0 \leq j \leq m-1\}$ sind paarweise verschieden.
   Angenommen, $j \cdot c \equiv \tilde{j} \cdot c \bmod m \Longrightarrow m$ teilt $(i - j) \cdot c$.
   $\gcd(m, c) = 1 \Longrightarrow m$ teilt $j - \tilde{j}$, ein Widerspruch, da $|j - \tilde{j}| < m$.

6. $n = 10.000$, $B = {}^{n}/_{m}$.
   $1 + {}^{1}/_{2}B = 2 \Longrightarrow B = 2 \Longrightarrow m = 5000$.
   $\mathrm{E}(kol) = n - m(1 - p_0), p_0 = e^{-2} = 0.0006179$.
   ${}^{\mathrm{E}(kol)}/_{n} = 1 - {}^{1}/_{B}(1 - p_0) = 0.5676$.
   56.76 % der Datensätze verursachen Kollisionen.
   Überlaufbereich: 5676 Plätze, Primärbereich: 5000 Plätze.

7. Wir verwenden die Formel $\frac{1}{B} \ln\left(\frac{1}{1-B}\right)$ für die mittlere Länge einer Sondierfolge beim Suchen eines Elementes und erhalten für $B = 0.8$ zwei Zugriffe im Mittel. Mit $B = \frac{n}{m}$ erhalten wir $l = 1250$. Wir wählen für $m = 1259$, die kleinste Primzahl > 1250.

8. Verwende ein Hashverfahren mit Hashfunktion $h : \{0,1\}^k \longrightarrow t[0..m]$, um die Suche nach übereinstimmenden Teilketten zu beschleunigen. In der Tabelle t werden Indizes in den Text-Puffer $tp[i - w..i - 1]$ relativ zum Anfang des Vorschau-Puffers $vp[i..i + v - 1]$ gespeichert. Verwende zum Beispiel die ersten 3 Zeichen $vp[1..3]$ als Input für die Hashfunktion ($k = 24$). Ein Schritt bei der Komprimierung besteht aus
   a. Suche die längste übereinstimmende Teilfolge l mit den Startpunkten, welche durch die mit $h(vp[1], vp[2], vp[3])$ kollidierenden Einträge in t definiert sind.
   b. Speichere den Index für $v[1..3]$ in t ab.
   c. Codiere l und verschiebe Text- und Vorschau-Puffer.
   d. Führe ein Update aller Indizes in t durch, das durch die Verschiebung von Text- und Vorschau-Puffer notwendig ist.

   Für Details siehe den Anhang B in [HanHarJoh98], der auch eine Implementierung in C für die Komprimierung und Dekomprimierung enthält.

9. a. Eindeutigkeit: Verwende ein Hashverfahren und überprüfe die Kollisionen. Dazu sind n Hashwerte zu berechnen. Sei $L = \{l_1, \ldots, l_n\}$. Verwende eine Hashfunktion h und berechne nacheinander $h(l_i)$, $i = 1, \ldots, n$. Für $h(l_i) = h(l_j)$ vergleiche l_i und l_j.
   b. Verwende eine Hashfunktion h und berechne $h(z_i)$, $i = 1, \ldots, n$. Überprüfe dann, ob $h(s - z_i)$, $i = 1, \ldots, n$, mit $h(z_j)$, $j = 1, \ldots, n$ kollidiert. Bei Kollisionen prüfe ob $s - z_i = z_j$ gilt.

10. Für die Wahrscheinlichkeit p_i, die angibt, dass auf einen Wert j i Schlüssel abgebildet werden gilt:

$$p_i := p_{ij} := p(n_j = i) = \frac{\binom{\frac{|S|}{m}}{i}\binom{|S| - \frac{|S|}{m}}{n-i}}{\binom{|S|}{n}}.$$

Anzahl der günstigen Fälle:
Wähle in $h^{-1}(j)$ i Elemente. Dafür gibt es $\binom{\frac{|S|}{m}}{i}$ viele Möglichkeiten.
Wähle aus den restlichen Fasern $n - i$ Elemente. Dafür gibt es $\binom{|S| - \frac{|S|}{m}}{n-i}$ viele Möglichkeiten.
Anzahl der möglichen Fälle: $\binom{|S|}{n}$.

Die Verteilung $\mathrm{p}(n_j = i)$ ist die hypergeometrische Verteilung mit den Parametern $\left(n, M = \frac{|S|}{m}, N = |S| - \frac{|S|}{m}\right)$ Es gilt $\mathrm{E}(n_j) = n\frac{M}{N} = \frac{n}{m-1}$ (Satz A.24).

11. a. bm gibt den verfügbaren Speicher an.
   b. β ist proportional zu B ($\beta = bB$).

Die Anzahl der Schlüssel, die auf j abgebildet werden, ist

$$n_j = |\{s \in S \mid H(s) = j\}|, j = 0, \ldots, m-1.$$

Die Anzahl der Werte mit i Urbildern ist $w_i = \sum_{j=0}^{m-1} \delta_{n_j,i}, i = 0, \ldots, n.$

$$\mathrm{E}(w_i) = mp_i, p_i = \binom{n}{i}\left(\frac{1}{m}\right)^i\left(1 - \frac{1}{m}\right)^{n-i}.$$

Werden auf einen Wert i Schlüssel abgebildet, so führen $i - b$ Schlüssel zu Kollisionen. Die Anzahl der Kollisionen ist

$$kol = \sum_{i=b+1}^{n} (i - b)w_i.$$

n_j, w_i, kol sind Zufallsvariable.

Satz. *Sei H eine Hashtabelle mit m Blöcken, Blockgröße b und Belegungsfaktor β. In H seien n Elemente gespeichert.*
Die Anzahl der stattgefundenen Kollisionen beträgt im Mittel

$$m\left(\frac{n}{m} - \sum_{i=1}^{b} ip_i - b\left(1 - \sum_{i=0}^{b} p_i\right)\right).$$

Beweis.

$$\begin{aligned}
\mathrm{E}(Kol) = E\left(\sum_{i=b+1}^{n} (i-b)w_i\right) &= \sum_{i=b+1}^{n} (i-b)\mathrm{E}(w_i) \\
&= \sum_{i=b+1}^{n} (i-b)mp_i. = m\sum_{i=b+1}^{n} (i-b)p_i. \\
&= m\left(\sum_{i=b+1}^{n} ip_i - b\sum_{i=b+1}^{n} p_i\right) \\
&= m\left(\frac{n}{m} - \sum_{i=1}^{b} ip_i - b\left(1 - \sum_{i=0}^{b} p_i\right)\right).
\end{aligned}$$

□

Corollar. *Sei H eine Hashtabelle mit m Blöcken, Blockgröße b und Belegungsfaktor β. In H seien n Elemente gespeichert.*
Der Prozentsatz der eingefügten Elemente, die Kollisionen verursachen, beträgt:

$$f(b,\beta) = \frac{100}{\beta}\left(\beta - \sum_{i=1}^{b} ip_i - b\left(1 - \sum_{i=0}^{b} p_i\right)\right).$$

Beweis.

$$\begin{aligned}
\frac{\mathrm{E}(Kol)}{n} &= \frac{m}{n}\left(\frac{n}{m} - \sum_{i=1}^{b} ip_i - b\left(1 - \sum_{i=0}^{b} p_i\right)\right) \\
&= \frac{1}{\beta}\left(\beta - \sum_{i=1}^{b} ip_i - b\left(1 - \sum_{i=0}^{b} p_i\right)\right).
\end{aligned}$$

□

Wir berechnen jetzt Werte für $f(b,\beta)$. Dabei verwenden wir $p_i \approx \frac{\beta}{i!}e^{-\beta}$.

	$\beta=0.1$	$\beta=0.5$	$\beta=1$	$\beta=1.5$	$\beta=2$	$\beta=2.5$	$\beta=3$
$b=1$	4.8	21.3	36.8	48.2	56.8	63.3	68.2
$b=2$	0.2	3.3	10.4	18.7	27.1	34.8	41.5
$b=3$	0.0	0.4	2.3	6.0	10.9	16.5	22.3
$b=4$	0.0	0.0	0.4	1.6	3.8	6.8	10.6
$b=5$	0.0	0.0	0.1	0.4	1.1	2.5	4.4

Beispiel. Speicherplatz für 10000 Elemente, Anzahl der Elemente 5000:

b	1	2	3	4	5
m	10000	5000	3333	2500	2000
β	0.5	1	1.5	2.0	2.5
kol	1065	520	300	190	125
freie Blöcke (in %)	60.7	36.8	22.3	13.5	8.2

12. ST und BT sind Stapel mit den Operationen PUSH und POP.
Folgende Operationen werden ausgeführt:

Am Blockanfang:	PushBT($topST$).
Am Blockende:	$topST \leftarrow$ PopBT.
Zum Einfügen:	PushST($entry$).
Zum Suchen:	sequenzielle Suche in ST.
Zum Prüfen eindeutiger Namen:	sequenzielle Suche in ST, die Abbruchbedingung gibt das oberste Element in BT.

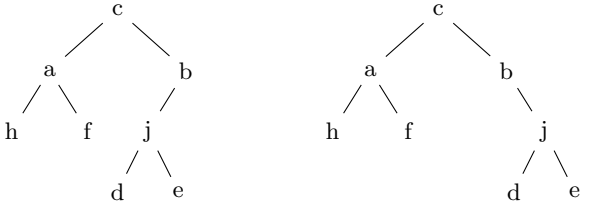
Dabei entsteht folgendes Problem: Die sequenzielle Suche ist wenig effizient.
Die Effizienz kann jedoch durch die Verwendung von Hashverfahren verbessert werden.

Stapel-Symboltabellen und Hashverfahren. Wie bisher sind ST und BT Stapel mit den Operationen PUSH und POP. Um verkettete Listen von Einträgen der Symboltabelle zu organisieren, erweitern wir die Symboltabelle um eine Spalte für Indizes. Die Hashtabelle HT speichert Zeiger (Indizes) auf Einträge in der Symboltabelle. Die Kollisionsauflösung bei gleichen Hashwerten erfolgt durch Verkettungen. Dazu dient die Indexspalte in ST.
Folgende Operationen werden ausgeführt:

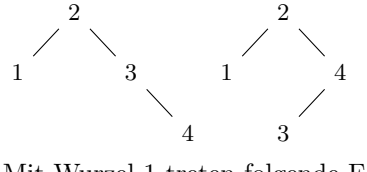
Am Blockanfang:	PushBT($topST$).
Am Blockende:	lösche alle Namen des Blocks aus der Hashorganisation, $topST \leftarrow$ PopBT.
Zum Einfügen:	$entry.next := HT[H(entry.name)]$, $HT[H(entry.name)] \leftarrow topST$, PushST($entry$),
Zum Suchen:	Suche über H in ST.
Zum Prüfen eindeutiger Namen:	Suche über H in ST, die Abbruchbedingung gibt das oberste Element in BT.

4. Bäume

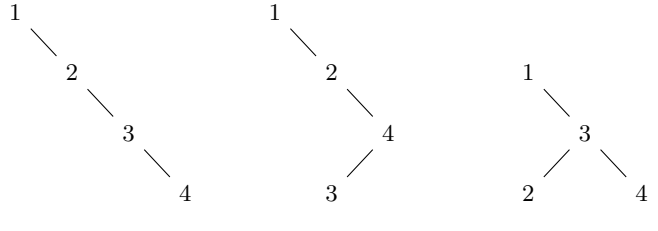
1. Wir bezeichnen mit L_1 die Preorder-Liste und mit L_2 die Postorder-Liste. Die Wurzel r ist erstes Element in L_1 und letztes Element in L_2. Sei x das zweite Element in L_1 und y vorletzte Element in L_2.
   Für $x \neq y$ gilt:
   (1) x ist linker Nachfolger der Wurzel. y ist rechter Nachfolger der Wurzel.
   (2) Alle Elemente in L_1, die zwischen x und y liegen gehören zum linken Teilbaum.
   (3) Alle Elemente in L_2, die zwischen x und y liegen gehören zum rechten Teilbaum.
   (4) Setze das Verfahren rekursiv mit den Teilbäumen mit Wurzel x und y fort.
   Für $x = y$ besitzt r nur einen Nachfolger x. Wir können nicht entscheiden, ob es sich um einen linken oder rechten Nachfolger handelt. Es gilt:
   (1) Der Teilbaum L mit Wurzel x enthält alle Elemente aus L_1, die rechts von x liegen.
   (2) Dies sind auch die Elemente aus L_2, die links von x liegen.
   (3) Setze das Verfahren rekursiv mit jeweils einer der Alternativen fort.

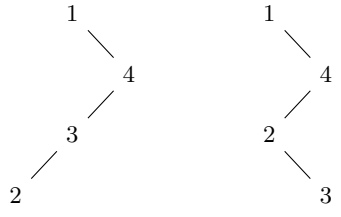


2. Mit Wurzel 2 treten folgende Fälle auf:



   Mit Wurzel 1 treten folgende Fälle auf:





Die Fälle Wurzel 3 und Wurzel 4 sind symmetrisch zu Wurzel 2 und Wurzel 1.
Insgesamt gibt es 14 verschiedene Anordnungen bei 24 verschiedenen Eingaben.

3. Die Postfix-Notation wird durch die Postorder-Ausgabe, die Prefix-Notation durch die Preorder-Ausgabe und die Infix-Notation wird durch die Inorder-Ausgabe erzeugt. Bei der Inorder-Ausgabe müssen Klammern gesetzt werden. Beim rekursiven Abstieg wird nach $*, +$ eine öffnende Klammer ausgegeben und auf dem Rückweg nach $+, *$ eine schließende Klammer.
Struktur eines arithmetischen Ausdrucks:

$$(a + b) * c * ((a + b) * c + (a + b + e) * (e + f)).$$

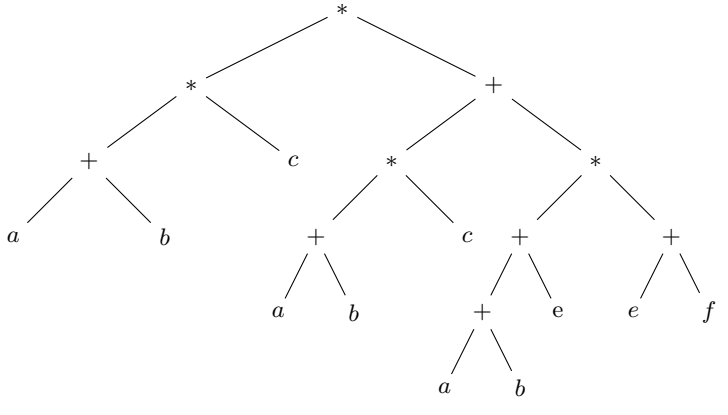


Fig. 2.2: Syntaxbaum eines arithmetischen Ausdrucks.

4. a. Wenn v einen linken Nachfolger hat, liegt vv im linken Teilbaum von v am weitesten rechts. Dieser Knoten hat keinen rechten Nachfolger.
   b. In einem binären Baum und einem gegebenen Knoten v bezeichnen wir mit nv denjenigen Knoten, der in der Inorder-Reihenfolge unmittelbar nach v auftritt (sofern existent). Zeigen Sie: Wenn v einen rechten Nachfolger nv hat, so hat nv keinen linken Nachfolger.

5. Wir definieren für jede Teilfolge $s_1, s_2, \ldots, s_k$, $k \geq 1$, ein Intervall I_k. Wir setzen $I_1 = [-\infty, \infty]$ und

$$I_2 = \begin{cases} [s_1, \infty], & \text{falls } s_2 \text{ rechter Nachfolger von } s_1 \text{ ist,} \\ [-\infty, s_1], & \text{falls } s_2 \text{ linker Nachfolger von } s_1 \text{ ist.} \end{cases}$$

Sei $k \geq 3$ und $I_k = [b_k, c_k]$ aufgrund von $s_1, s_2, \ldots, s_k$ definiert. Falls $s_{k+1} \notin I_k$ ist, dann ist die Folge nicht möglich. Sonst setzen wir

$$I_{k+1} = \begin{cases} I_k, & \text{falls } s_k \text{ und } s_{k+1} \text{ beide linke oder rechte Nachfolger sind,} \\ [s_k, c_k], & \text{falls } s_k \text{ linker und } s_{k+1} \text{ rechter Nachfolger ist,} \\ [b_k, s_k], & \text{falls } s_k \text{ rechter und } s_{k+1} \text{ linker Nachfolger ist.} \end{cases}$$

Falls $s \in I_{n-1}$ ist, dann ist die Folge möglich.

6. Entweder ist x rechter oder linker Nachfolger von y. Da x ein Blatt ist, wird beim Inorder-Traversieren x unmittelbar vor y oder unmittelbar nach y ausgegeben. Da Inorder-Traversierens die sortierte die Ausgabe erzeugt, folgt die Behauptung.

7. Wir konstruieren einen binären Suchbaum für $\{1, \ldots, n\}$ durch folgende Vorschrift: Wähle als Wurzel das mittlere Element x. Setze die Konstruktion für die Elemente kleiner x bzw. größer x rekursiv fort.
Für die Höhe $h(n)$ des Suchbaumes gilt

$$h(1) = 1,\ h(n) \leq h\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1 = h\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

Mit Satz 1.28 folgt $h(n) \leq \lfloor \log_2(n) \rfloor + 1$.

8. Füge die Elemente in der „Reihenfolge“ der Ebenen ein. Erst die Wurzel, dann die Elemente der Ebene 1 von links nach rechts und so weiter. Durch die binäre Suchbaumeigenschaft entsteht der ursprüngliche Baum. Die AVL-Bedingung ist erfüllt.

9. Es entsteht ein Baum, dessen Blätter in höchstens zwei Ebenen enthalten sind. Nachdem $2^k - 1$ Element eingefügt sind, ist die unterste Ebene voll besetzt. Für die Höhe h gilt $h = \lfloor \log_2(n) \rfloor$.

10. Sei u ein Knoten in $\overline{T}$, sei $u = u_0, u_1, \ldots, u_n$ ein Pfad, der u mit einem Blatt verbindet und sei $\overline{T}_u$ der Teilbaum von $\overline{T}$ mit Wurzel u. Mit hb_u bezeichnen wir die Anzahl der schwarzen Kanten in $u_0, u_1, \ldots, u_n$, und mit n_u die Anzahl der inneren Knoten von $\overline{T}_u$. Beachte, hb_u ist unabhängig von der Wahl des Blattes. Wir zeigen durch Induktion nach der Höhe h_u von u, dass

$$n_u \geq 2^{hb_u} - 1$$

gilt. Für $h_u = 0$ ist u ein Blatt und $2^{hb_u} - 1 = 2^0 - 1 = 0$ und $n_u = 0$. Sei $h_u \geq 1$. Der Knoten u besitzt zwei Nachfolger v und w. Es gilt

$n_u = n_v + n_w + 1$ und $h_v, h_w < h_u$. Es gilt $hb_v, hb_w \geq hb_u - 1$. Hieraus folgt

$$n_u = n_v + n_w + 1 \geq 2^{hb_u - 1} - 1 + 2^{hb_u - 1} - 1 + 1 = 2^{hb_u} - 1.$$

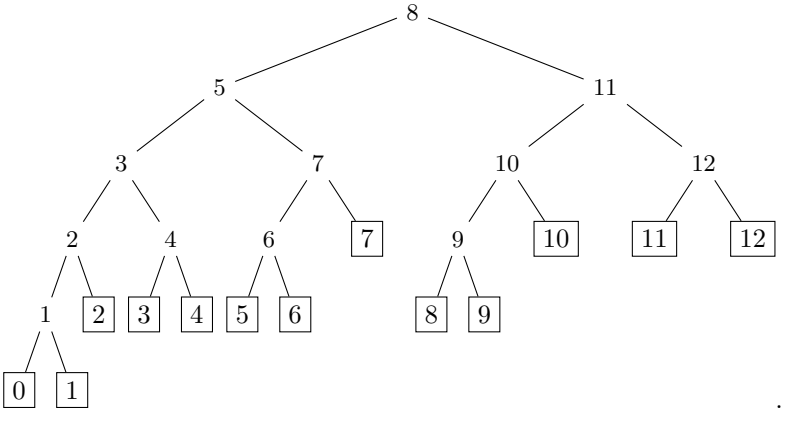
Dies zeigt die Behauptung.
Da auf eine rote Kante eine schwarze Kante folgt, gilt $hb_r \geq h/2$ für die Wurzel r und die Höhe h von $\overline{T}$. Es folgt

$$n \geq 2^{hb_r} - 1 \geq 2^{\frac{h}{2}} - 1,$$

was äquivalent zu $h \leq 2\log_2(n+1)$ ist.
Algorithmen zum Einfügen und Löschen für Rot-Schwarz-Bäume werden zum Beispiel in [CorLeiRivSte07, Chapter 13] behandelt.

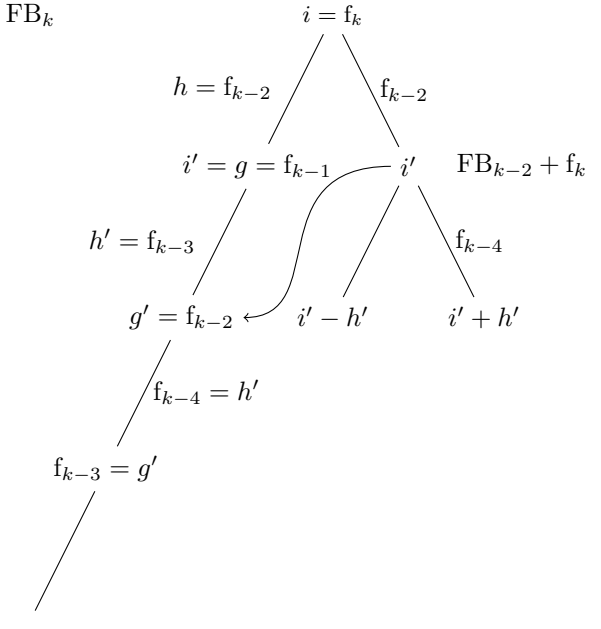
11. Der folgende Algorithmus 4.1 setzt voraus, das drei aufeinander folgende Fibonacci-Zahlen vorberechnet sind. Dies kann mit Algorithmus 1.20 erfolgen. Alternativ können die ersten κ Fibonacci-Zahlen vorberechnet und in dem Array $fib[0..\kappa]$ gespeichert werden. Aus $\mathrm{f}_\kappa \approx \frac{\mathrm{g}^k}{\sqrt{5}} \geq n$ folgt $k \approx \log_{\mathrm{g}}(n\sqrt{5}) = O(\log_2(n))$. Um fib zu initialisieren sind $O(\log_2(n))$ Additionen notwendig.

Im folgenden Fibonacci-Baum FB_6 finden wir die Suchpfade, die entstehen, wenn wir alle Elemente in einem Array $a[1..n]$, $8 \leq n \leq 12$, suchen.



.

Wir nehmen $n = \mathrm{f}_{k+1} - 1$ an. Dann verwenden wir die inneren Knoten des Fibonacci-Baums FB_k. Für einen inneren Knoten v, haben beide Nachfolger dieselbe Differenz mit v und diese Differenz ist eine Fibonacci-Zahl. Die Fibonacci-Zahlen finden wir in absteigender Reihenfolge ganz links in FB_k Wir leiten die Navigation mithilfe der folgenden Skizze ab.



Initialisierung: $i = \mathrm{f}_k$, $g = \mathrm{f}_{k-1}$ und $h = \mathrm{f}_{k-2}$.

$h > 0$ und $a[i] > x$:	$g > 1$ und $a[i] < x$:
$i' = i - h$ (links),	$i' = i + h$ (rechts),
$g' = h$,	$g' = g - h$,
$h' = g - h$.	$h' = h - g'$.

$(g, h) = (\mathrm{f}_j, \mathrm{f}_{j-1})$ sind zwei aufeinander folgende Fibonacci-Zahlen. Ist der Abstand zwischen Vorgänger v und Nachfolger w (im Fibonacci-Baum) gleich f_j, dann ist der nächste Abstand f_{j-1}, wenn w ein linker Nachfolger, und f_{j-2}, wenn w ein rechter Nachfolger ist.
Also ist der nächste Abstand nicht existent für „links“ und $h = \mathrm{f}_0 = 0$ und für „rechts“ und $g = \mathrm{f}_2 = 1$ (und $h = 1$ oder $h = 0$).
In diesen Fällen befindet sich das zu suchende Element nicht in a.
Falls $\mathrm{f}_k < n + 1 < \mathrm{f}_{k+1}$ gilt, kann der Fall $i' > n$ eintreten. Dann erfolgt ein Schritt nach links und kein Zugriff auf a.

Sei $k+1$ der Index der kleinsten Fibonacci-Zahl, für die $\mathrm{f}_{k+1} \geq n+1$ gilt. Beim Aufruf von FibSearch ist das Array $a[1..n]$, in dem gesucht werden soll, das zu suchende Element x und der Index k zu übergeben.

Algorithmus 4.1.

```
FibSearch(item a[1..n], x; index k)
 1  int i ← f_k, g ← f_{k−1}, h ← f_{k−2}
 2  while true do
 3      if i > n
 4        then i ← i − h, t ← g, g ← h, h = t − h
 5        else if a[i] < x
 6               then if g = 1 then return −1
 7                    i ← i + h, g ← g − h, h = h − g
 8        else if a[i] > x
 9               then if h = 0 then return −1
10                    i ← i − h, t ← g, g ← h, h = t − h
11        else if a[i] = x then return i
```

Wir zerlegen ein Array der Länge $f_k - 1$ in Arrays der Länge $f_{k-1} - 1$, 1 und $f_{k-2} - 1$. Wir stellen eine Rekursionsgleichung für die Vergleiche mit Array-Elementen auf.

$$V(f_k - 1) \leq V(f_{k-1} - 1) + 1$$

Wir setzen $n = f_k - 1$ und $x_k = V(f_k - 1)$ und erhalten $x_k = x_{k-1} + 1 = \sum_{i=2}^{k} 1 = k - 1$. $n + 1 = f_k \approx \frac{g^k}{\sqrt{5}}$.

Es folgt $k \approx \log_g(\sqrt{5}(n+1))$ und

$$V(n) \approx \log_g(\sqrt{5}(n+1)) - 1 = O(\log_2(n)).$$

Siehe auch Satz 4.14, der die Höhe eines ausgeglichenen Baumes mit n Knoten abschätzt.

12. Da ein Treap eindeutig durch die gespeicherten Elemente bestimmt ist, hängt der Treap nicht davon ab durch welche Einfüge- und Löschoperationen er entstanden ist.

13. Sei c_n die Anzahl der binären Bäume mit n Knoten. Es gibt genau einen binären Baum mit 0 Knoten. Sei B ein binärer Baum mit n Knoten $n \geq 0$. Der linke Teilbaum der Wurzel habe j Knoten, der rechte Teilbaum habe $n - j - 1$ Knoten. Es ergibt sich deshalb die Rekursion

$$c_0 = 1,$$
$$c_n = \sum_{j=0}^{n-1} c_j c_{n-j-1}.$$

Wir betrachten die erzeugende Funktion

$$G(z) = \sum_{i=0}^{\infty} c_i z^i = \sum_{i=0}^{\infty} \sum_{j=0}^{i-1} c_j c_{i-j-1} z^i$$
$$= zG(z)^2 - 1.$$

Die Gleichung $x^2 - \frac{1}{z}x - \frac{1}{z}$ besitzt die Lösung $x = 1/2z(1 - \sqrt{1-4z})$. Wir entwickeln $1/2z(1 - \sqrt{1-4z})$ mit der binomischen Reihe und erhalten

$$G(z) = \frac{1}{2z}(1 - 4\sqrt{1-4z}) = \frac{1}{2z}\left(1 - \sum_{n=0}^{\infty}\binom{\frac{1}{2}}{n}(-4z)^n\right)$$
$$= 2\sum_{n=0}^{\infty}\binom{\frac{1}{2}}{n+1}(-4z)^n = \sum_{n=0}^{\infty}\binom{-\frac{1}{2}}{n}\frac{(-4z)^n}{n+1}$$
$$= \sum_{n=0}^{\infty}\binom{2n}{n}\frac{z^n}{n+1}.$$

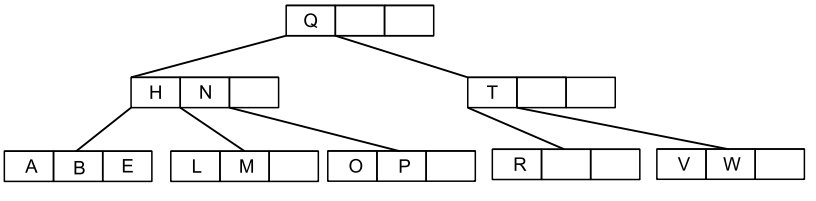
Hieraus folgt $c_n = \frac{1}{n+1}\binom{2n}{n}$.

14. Es gibt Seiten mit 4 und mit 2 Elementen. Daher gilt $d \geq 5$ und $\left\lfloor \frac{d-1}{2} \right\rfloor \leq 2$. Also folgt $d = 5$ oder $d = 6$.

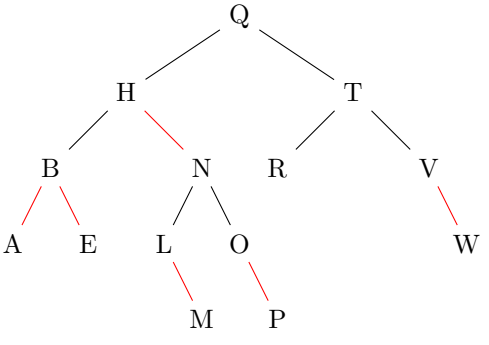
15. a. $3 \leq d - 1$, $\left\lfloor \frac{d-1}{2} \right\rfloor \leq 1$, also folgt $d = 4$.
    b. Vertausche h mit l, lösche h:
    Ebene 0: m, Ebene 1: e, t und x, Ebene 2: b, f, l, r, u, v, w und y.
    Lösche l, lösche b: Underflow in Ebene 2. Fasse zusammen:
    Ebene 0: m, Ebene 1: t und x, Ebene 2: e, f, r, u, v, w und y.
    Underflow in Ebene 1, gleiche aus:
    Ebene 0: t, Ebene 1: m und x, Ebene 2: e, f, r, u, v, w und y.

16. Es ist einfach ein Gegenbeispiel zu finden: Sei $d = 5$.
    a. Ebene 0: 11,50 und Ebene 1: 2,3; 12,17; 52,53.
    b. Ebene 0: 17 und Ebene 1: 2,3,11,12; 50,52,53.

17. Wir ordnen dem B-Baum



seinen Rot-Schwarz-Baum zu.



Sei B ein B-Baum und R_B der zugeordnete Rot-Schwarz-Baum. Auf ein Element in einer B-Baumseite folgen B-Baumkanten. Die einer roten Kante in R_B nachfolgenden Kanten sind schwarz, also folgt auf eine rote Kante in einem Pfad eine schwarze Kante. Da sich die Blätter im B-Baum auf einer Ebene befinden, sind alle Pfade, die von einem Knoten aus zu einem Blatt gehen, gleich lang. Deshalb sind für jeden Knoten $v \in R_B$ für alle Pfade, die in v beginnen und in einem Blatt enden, die Anzahl der schwarzen Kanten gleich.
Sei R ein Rot-Schwarz-Baum. Wir ordnen R einen B-Baum B_R zu. Da die Nachfolger einer roten Kante schwarz sind, sind höchstens drei Knoten durch (zwei) rote Kanten verbunden. Wir definieren eine B-Baumseite durch Knoten die durch eine rote Kante verbunden sind. Die schwarzen Kanten werden zu Kanten zwischen den B-Baumseiten. Da in einem Rot-Schwarz-Baum alle Pfade von der Wurzel zu einem Blatt die gleiche Anzahl von schwarzen Kanten besitzen, befinden sich im zugeordneten B-Baum alle Blätter auf einer Ebene. Die Konstruktion ergibt einen B-Baum der Ordnung 4.

18. a. C_1 ist nicht eindeutig decodierbar, da bbaa‖c‖dea‖bbd = bb‖aacde‖abbd.
    b. C_2 ist eindeutig decodierbar, weil der reverse Code {c, bb, dbb, aed, ebba, daab, aabb, edcaa} präfixfrei ist.

19. Sei $Y = \{y_1, \ldots, y_n\}$. Der Codebaum für $Y^* = \cup_{i=0}^{\infty} Y^i$ besitzt in der i–ten Ebene n^i viele Knoten. In der i–ten Ebene besitzen die belegten n_k Knoten aus der k–ten Ebene, $k = 1, \ldots, i-1$, $n_k n^{i-k}$ viele Nachfolger. Für einen Präfixcode C können diese Knoten nicht im Codebaum von C enthalten sein. Die Knoten der Codewörter der Länge i sind eine Teilmenge der verbleibenden $n^i - n_1 n^{i-1} - \ldots - n_{i-1} n$ Knoten. Deshalb gilt

$$
\begin{aligned}
n_1 &\leq n \\
n_2 &\leq n^2 - n_1 n \\
n_3 &\leq n^3 - n_1 n^2 - n_2 n \\
&\vdots \\
n_i &\leq n^i - n_1 n^{i-1} - \ldots - n_{i-1} n \\
&\vdots \\
n_l &\leq n^l - n_1 n^{l-1} - \ldots - n_{l-1} n
\end{aligned}
$$

20. a. Der angegeben Code ist kompakt, da er mit dem Algorithmus 4.43 erzeugt werden kann.
    b. Das Ergebnis von Algorithmus 4.43 ist nicht eindeutig. Durch Auswahl von zwei Knoten aus der Menge der Knoten mit niedrigster Wahrscheinlichkeit können sogar Codes mit unterschiedlichen Wortlängen entstehen.
    C = {00,010,011,100,101,110,111} ist auch ein Huffman-Code. Für beide Codes ist die mittlere Wortlänge $2\frac{5}{7}$.

21. Wird über einem Alphabet mit q Symbolen codiert, so können q viele Nachrichten mit einem Symbol codiert werden. Für ein Codewort können durch Anfügen eines Symbols q viele Verlängerungen erzeugt werden. Deshalb werden in einem Schritt die q Knoten mit den niedrigsten Wahrscheinlichkeiten zusammengefasst. Damit der Algorithmus mit q Nachrichten terminiert, muss unter Umständen X mit Nachrichten mit Wahrscheinlichkeit 0 ergänzt werden, sodass $|X| = q - k(q-1)$ gilt.

22. Es gilt $H(X) = l(C)$. Mit dem Quellencodierungssatz folgt, dass der Code kompakt ist.

23. Aus $\sum_{i=1}^{k} p_i = 1$ folgt $p_i = \frac{1}{2^i}$ für $i = 1, \ldots, k-1$ und $p_k = \frac{1}{2^{k-1}}$. Sei

$$c_1 = 0, c_2 = 10, c_3 = 110, \ldots, c_{k-1} = \underbrace{1 \ldots 1}_{k-2} 0, c_k = \underbrace{1 \ldots 1}_{k-1}.$$

    Es gilt

$$l(C) = \sum_{i=1}^{k-1} \frac{i}{2^i} + \frac{k-1}{2^{k-1}} = 2 - \frac{1}{2^{k-2}}.$$

    Aus $H(X) = l(C)$ folgt mit dem Quellencodierungssatz, dass der Code C kompakt ist.

24. Jeder kompakte Code ist ein Huffman-Code. Wir betrachten deshalb den Algorithmus 4.43 zur Konstruktion eines kompakten Codes. Die Summe

der Wortlängen $\sum_{i=1}^{n} l_i$ ist am größten, wenn in jedem Konstruktionsschritt ein Codewort maximaler Länge durch 0 bzw. 1 verlängert wird. Es ergeben sich die Codewortlängen $1, 2, 3, \ldots, n-1, n-1$. Für diese Wortlängen gilt

$$\sum_{i=1}^{n} l_i = \sum_{i=1}^{n-1} i + n - 1 = \frac{(n-1)(n+2)}{2} = \frac{1}{2}(n^2 + n - 2).$$

(Anhang B, Seite 362).

25. a. Der Code für die Nachricht $acfg$ ist 1709.
    b. Die Zahl 1688 codiert die Nachricht acfaeb.

26. Sei $X = \{x_1, \ldots, x_{2^l}\}$, $\mathrm{p}(x_i) = 1/2^l$.

$$I(x_{i_1} \ldots x_{i_n}) = \left[\frac{j}{2^{nl}}, \frac{j+1}{2^{nl}}\right[, j = 0, \ldots, 2^{nl} - 1.$$

Codiere j mit nl Bit: $j_{nl-1} \ldots j_k \ldots j_0$, $j_{nl-1} = \ldots = j_k = 0$, $j_{k-1} = 1$ und

$$x_{i_1} \ldots x_{i_n} \to j_{k-1} \ldots j_0.$$

27. $\mathrm{p}(a) = 1 - 1/2^k$, $\mathrm{p}(b) = 1/2^k$. Es gilt $I(b) = [.\underbrace{1 \ldots 1}_{k}, \overline{1}[$.

Per Induktion folgt $I(b^n) = [.\underbrace{1 \ldots 1}_{nk}, \overline{1}[$:

Wir zeigen den Schluss von n auf $n+1$:

$$\begin{aligned} I(b^{n+1}) &= .\underbrace{1 \ldots 1}_{nk} + .\underbrace{0 \ldots 0}_{nk}\,\overline{1}[.\underbrace{1 \ldots 1}_{k}, \overline{1}[\\ &= [.\underbrace{1 \ldots 1}_{(n+1)k}, \overline{1}[. \end{aligned}$$

$$\begin{aligned} I(b^n a) &= .\underbrace{1 \ldots 1}_{nk} + .\underbrace{0 \ldots 0}_{nk}\,\overline{1}[0, .\underbrace{1 \ldots 1}_{k}[\\ &= [.\underbrace{1 \ldots 1}_{nk}, .\underbrace{1 \ldots 1}_{(n+1)k}[\end{aligned}$$

Codiere

$$b^n a \to .\underbrace{1 \ldots 1}_{nk}.$$

28. Zum Beispiel erfordert $r = 5$ und

$$abcdefabcdefabcdefabcdef \ldots \in \{a, b, c, d, e, f\},$$

dass jedes Zeichen als Einzelzeichen codiert werden muss. Es liegt maximale Expansion vor.

5. Graphen

1. Wenn G einen Eulerkreis besitzt, ist G zusammenhängend. Ein Eulerkreis, der in einen Knoten hineinführt, muss auch wieder aus dem Knoten hinausführen. Deshalb ist der Grad eines jeden Knoten gerade.
   Für die umgekehrte Richtung betrachten wir einen Pfad

   $$P : v_0, \ldots, v_k$$

   maximaler Länge, der keine Kante mehrfach enthält. Da es sich um einen Pfad maximaler Länge handelt, liegen alle zu v_k inzidenten Kanten auf diesem Pfad. Da mit jedem Auftreten von v_k im Inneren von P zwei inzidente Kanten verbunden sind, folgt aus $v_0 \neq v_k$, dass $\deg(v_k)$ ungerade ist. Daher gilt $v_0 = v_k$. Wir zeigen, dass alle Kanten von G in P vorkommen.
   Angenommen, es gibt eine Kante, die nicht auf P liegt. Da G zusammenhängend ist, gibt es eine zu einem Knoten von P inzidente Kante e, die nicht auf P liegt. Sei $e = \{v_i, u\}$, $0 \leq i \leq k$. Dann ist

   $$u, v_i, v_{i+1}, \ldots, v_{k-1}, v_k = v_0, v_1 \ldots, v_{i-1}, v_i$$

   ein Pfad, der keine Kante mehrfach enthält und die Länge $k+1$ besitzt. Ein Widerspruch zur Wahl von P.

2. Wir führen den Beweis durch Induktion nach der Anzahl $|E|$ der Kanten. Für $|E| = 0$ ist $|V| = 1$ und $|F| = 1$. Die Formel ist richtig. Sei $|E| \geq 1$. Ist der Graph ein Baum, dann gilt $|E| = |V| - 1$ und $|F| = 1$. Die Formel ist auch richtig.
   Sei jetzt Z ein Zyklus und e eine Kante von Z. Dann gilt für den Graphen $G' = G \setminus \{e\}$, der aus E durch Wegnahme der Kante e entsteht, $|E'| = |E| - 1$ und da auf jeder Seite von e eine andere Fläche liegt, die sich bei Wegnahme von e vereinigen, gilt $|F'| = |F| - 1$. Nach Induktionsvoraussetzung gilt für G'

   $$|V'| - |E'| + |F'| = 2$$

   und damit gilt die Formel

   $$|V| - |E| + |F| = |V'| - (|E'| + 1) + (|F'| + 1) = 2$$

   auch für G.

3. Sei $G = (V, E)$ ein Graph, $V = \{1, 2, \ldots, n\}$ und seien $col = \{1, 2, \ldots, n\}$ Farbnummern paarweise verschiedener Farben.

Algorithmus 5.1.

col $col[1..n]$; node $adl[1..n]$

```
colourNodes()
 1  node no; set availableCol
 2  col[1] ← 1
 3  for k ← 2 to n do
 4      col[k] ← 0
 5  for k ← 2 to n do
 6      availableCol ← {1, ..., n}
 7      no ← adl[k]
 8      while no ≠ null do
 9          availableCol ← availableCol \ {col[no.v]}
10          no ← no.next
11      col[k] ← min availableCol
```

Wir färben die Knoten nacheinander. In jedem Schritt sind die Farben, die für die Nachbarn des Knotens k verwendet wurden, ausgeschlossen. Sei l die kleinste Farbnummer aus $avalableCol$. l ist am größten, wenn die Farben mit den Farbnummern $1, \ldots, \deg(k)$ ausgeschlossen sind. Daher gilt $l \leq \deg(k)+1 \leq \Delta+1$, $k = 1, \ldots, n$. Für einen vollständigen Graphen wird die Schranke angenommen. Die Laufzeit ist von der Ordnung $O(n+m)$.

4. Sei m die Anzahl der Kanten von $G = (V, E)$. Dann gilt

$$\sum_{v \in V} \deg(v) = 2m.$$

Da die Summe einer ungeraden Anzahl von ungeraden Summanden ungerade ist, folgt die Behauptung.

5. Seien $[x] = [\tilde{x}]$ und $[y] = [\tilde{y}]$ gegeben. 21 teilt $x - \tilde{x}$ und $\tilde{y} - y$. Also teilt 21 auch $x - \tilde{x} + \tilde{y} - y = x - y - (\tilde{x} - \tilde{y})$. Somit teilt auch 7 $x - y - (\tilde{x} - \tilde{y})$. Es gilt deshalb: 7 teilt $x - y$ genau dann, wenn 7 teilt $\tilde{x} - \tilde{y}$. Die Definition hängt somit nicht von der Wahl des Repräsentanten ab.
Die Äquivalenzklassen sind die starken Zusammenhangskomponenten.

6. a. Beweis durch Induktion nach $m = |E|$.
(1) Induktionsbeginn: $m = 0 \Longrightarrow n = 1$ (da G zusammenhängend ist). Die Formel ist somit richtig.
(2) „$< m \Longrightarrow m$“: Wir entfernen eine Kante in G. Entweder bleibt G zusammenhängend. Dann gilt nach Induktionsvoraussetzung $m - 1 \geq n - 1$ also gilt auch $m \geq n - 1$. Im anderen Fall zerfällt der Graph in zwei Komponenten $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$. Nach Induktionsvoraussetzung gilt:
$|E_1| \geq |V_1| - 1$ und $|E_2| \geq |V_2| - 1 \Longrightarrow |E| = |E_1| + |E_2| + 1 \geq |V_1| + |V_2| - 1$.

b. Beweis durch Induktion nach $n := |V|$.
(1) Induktionsbeginn: $n = 1 \Longrightarrow G$ zusammenhängend. Die Aussage ist somit richtig.
(2) „$n \Longrightarrow n+1$“: Sei $n \geq 2$ und sei $|E| \geq \binom{n}{2} + 1$ vorausgesetzt. Zu zeigen ist, dass G zusammenhängend ist. Wenn der Graph vollständig ist, ist er auch zusammenhängend. Ist der Graph nicht vollständig, so gibt es einen Knoten v mit weniger als $n-1$ inzidente Kanten. Wir entfernen v und alle inzidenten Kanten in G und erhalten $G' = (V', E')$. Für G' gilt:

$$|E'| \geq |E| - (n-1) \geq \binom{n}{2} + 1 - (n-1) \geq \binom{n-1}{2} + 1.$$

Nach Induktionsvoraussetzung ist G' zusammenhängend. Zwischen den Knoten aus $V \setminus \{v\}$ gibt es höchstens $\binom{n-1}{2}$ viele Kanten. Die restliche Kante besitzt deshalb einen Endpunkt in E' und der andere Endpunkt ist v. G ist somit zusammenhängend.

7. a $\Longleftrightarrow$b: Falls G ein Baum ist, ist G azyklisch und besitzt $|V| - 1$ viele Kanten (siehe Satz 5.7).
Sei G azyklisch mit $|V| - 1$ vielen Kanten. Angenommen, G wäre nicht zusammenhängend. Seien $G_1, \ldots, G_r$, $r \geq 2$, die Zusammenhangskomponenten von G. Durch Hinzunahme von $r-1$ vielen Kanten wird G zusammenhängend und bleibt dabei azyklisch. G ist ein Baum mit $|V| - 1 + r - 1 > |V| - 1$ vielen Kanten ein Widerspruch (siehe Satz 5.7).
a $\Longleftrightarrow$c: Sei $G = (V, E)$ ein Baum (insbesondere ist G zusammenhängend). Sei $e = \{v, w\}$ eine zusätzliche Kante. In G gibt es einen Pfad P von v nach w. Verlängern wir P um $\{w, v\}$ so erhalten wir einen Zyklus. Sind umgekehrt $v, w \in V$ gegeben. Entsteht durch Hinzunahme der Kante $\{w, v\}$ ein Zyklus, so muss es schon vorher einen Pfad von v nach w gegeben haben. G ist somit zusammenhängend.
a $\Longleftrightarrow$d: Sei $G = (V, E)$ ein Graph und $e = \{v, w\} \in E$. Für G gilt, $G \setminus \{e\} = (V, E \setminus \{e\})$ zerfällt in höchstens zwei Komponenten. $G \setminus \{e\}$ ist genau dann zusammenhängend, wenn es in G Zyklen gibt. $G \setminus \{e\}$ zerfällt in mindestens zwei Komponenten, wenn G azyklisch ist. Also gilt die Aussage in Punkt d für einen Baum. Zerfällt ein Graph nach Wegnahme von e in zwei Komponenten für alle Kanten e, so ist der Graph azyklisch. Somit ist G ein Baum.
a $\Longleftrightarrow$e: G ist genau dann zusammenhängend, wenn es zwischen je zwei Knoten mindestens einen Pfad gibt.
G ist genau dann azyklisch, wenn es zwischen zwei Knoten höchstens einen Pfad gibt.

8. Sei $v \in V$. Entweder gibt es drei weitere Knoten, die zu v adjazent sind oder es gibt drei weitere Knoten, die nicht zu v adjazent sind.
Betrachten wir zunächst den ersten Fall. Entweder gibt es zwei Knoten unter den dreien, die untereinander adjazent sind, dann bilden diese bei-

den mit v eine Dreiergruppe und je zwei Knoten aus dieser Gruppe sind adjazent. Oder je zwei Knoten dieser Dreiergruppe sind nicht adjazent. Dann ist dies eine Dreiergruppe und je zwei Knoten dieser Gruppe sind nicht adjazent.
Der zweite Fall - es gibt drei weitere Knoten, die nicht zu v adjazent sind - kann mit einer analogen Argumentation behandelt werden.

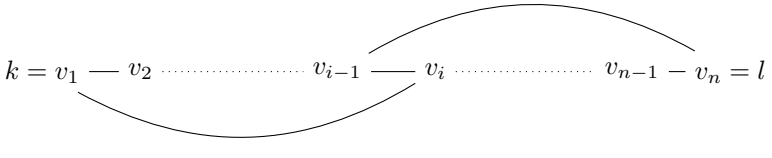
9. Angenommen, die Aussage wäre nicht richtig. Wir betrachten in der Menge der Graphen mit n Knoten einen Graphen G mit einer maximalen Anzahl von Kanten für den die Aussage nicht gilt, d. h. für nicht adjazente Knoten k und l gilt

$$(*) \qquad \deg(k) + \deg(l) \geq n,$$

aber G besitzt keinen Hamiltonkreis.
Sei $G' = G \cup \{k, l\}$. In G' gilt die Bedingung $(*)$ und für G' ist die Aussage richtig. Es gibt somit in G' einen Zyklus Z, der jeden Knoten genau einmal enthält. Dieser Zyklus muss, die Kante $\{k, l\}$ enthalten. Sei P der Pfad, den wir erhalten, wenn wir in Z die Kante $\{k, l\}$ entfernen.

$$P : k = v_1, \ldots, v_n = l.$$

Wir zeigen gleich, dass für jeden Knoten v_i, $i = 2, \ldots, n$, der zu k adjazent ist, v_{i-1} nicht zu l adjazent ist. Aus dieser Bedingung folgt $\deg(l) \leq n - 1 - \deg(k)$, d. h. $\deg(k) + \deg(l) \leq n - 1$, ein Widerspruch zur Bedingung $\deg(k) + \deg(l) \geq n$. Die Aussage der Aufgabenstellung ist somit richtig.
Wir zeigen jetzt die obige Bedingung. Dazu nehmen wir an, es gäbe ein i mit v_i ist zu k und v_{i-1} ist zu l adjazent.



Dann ist

$$v_1, v_i, \ldots, v_n, v_{i-1}, v_{i-2}, \ldots, v_1$$

ein Zyklus in G, der jeden Knoten von G genau einmal enthält, ein Widerspruch.

10. Der Algorithmus sortiert die Intervalle nach den Anfangspunkten und benutzt eine zusätzliche Liste von aktiven Knoten (Intervallen). Die Liste der aktiven Knoten wird nach den Endpunkten sortiert.
1. Sortiere die Liste V der Intervalle nach den Anfangspunkten.
2. Arbeite die sortierte Liste V ab. Jeder Schritt besteht aus: (1) Füge den Knoten I in die Liste der aktiven Knoten ein. (2) Entferne jeden

Knoten aus der aktiven Liste, dessen Endpunkt vor dem Anfangspunkt von I liegt. (3) Alle Knoten, die in der aktiven Liste verbleibenden, sind adjazent zu I.

11. Das Problem kann durch die Suche eines kürzesten Weges zwischen zwei Knoten in einem Graphen gelöst werden. Wir modellieren die Situation mit einem Graphen. Ein Knoten ist durch eine Teilmenge von {m,k,w,z} (m = Mann, k = Krautkopf, w= Wolf, z = Ziege) gegeben, bestehend aus den Elementen, die sich auf dieser Seite des Flusses befinden. Zunächst befindet sich alle Elemente von {m,k,w,z} auf dieser Seite dieses Flusses. Jede Abfahrt oder Ankunft des Bootes definiert eine neue Teilmenge. Zulässige Teilmengen sind jene Teilmengen, die nicht zu einer Katastrophe führen (Ziege frisst Krautkopf, Wolf frisst Ziege). Zwischen zulässigen Teilmengen z_1 und z_2 wird eine Kante gezeichnet, wenn ein Übergang von z_1 nach z_2 durch Abfahrt oder Ankunft des Bootes möglich ist. Gesucht ist ein Weg, der {m,k,w,z} und {} verbindet. Ein kürzester Weg kann zum Beispiel mit Breitensuche gefunden werden. Eine Lösung ist (einfach ohne Graphen) zu finden.
    a. Der Mann setzt mit der Ziege über (dies ist die einzige Möglichkeit).
    b. Er rudert alleine zurück.
    c. Der Mann setzt mit dem Wolf über.
    d. Er rudert mit der Ziege zurück.
    e. Der Mann setzt mit dem Krautkopf über.
    f. Er rudert alleine zurück.
    g. Der Mann setzt mit der Ziege über.

12. Wir ersetzen die Queue im Algorithmus 5.11 durch einen Stack. Die Besuchsreihenfolge stimmt nicht immer komplett mit der Besuchsreihenfolge der rekursiven Version überein.

Algorithmus 5.2.

```
DepthFirstSearch()
 1  vertex k, int nr
 2  for k ← 1 to p do
 3       where[k] ← 0; parent[k] ← 0
 4  nr ← 1
 5  for k ← 1 to p do
 6       if where[k] = 0
 7          then Visit(k); nr ← nr + 1
```

```
Visit(vertex k)
 1  node mo; no = adl[k]
 2  if no ≠ null
 3    then Push(k, no); where[k] ← −1
 4  repeat
 5      (k, no) = Top()
 6      while where[no.v] ≠ 0 and no ≠ null do
 7          no ← no.next
 8      (k, mo) = Pop()
 9      if no ≠ null
10        then parent[no.v] ← k; Push(k, no)
11             mo = adl[no.v]
12             if mo ≠ null
13               then Push(no.v, mo); where[no.v] ← −1
14        else  where[k] ← nr
15  until QueueEmty
```

13. Bei einem Graphen gibt es bei BFS nur Baumkanten und Querkanten (keine Rückwärts- und keine Vorwärtskanten). Eine mögliche Rückwärts- oder Vorwärtskante würde bedeuten, dass ein Knoten einer tieferen Ebene zu einer höheren Ebene (Höhenunterschied mindestens 2) adjazent ist. Dann erscheint der Knoten der tieferen Ebene aber als Nachfolger des Knotens der höheren Ebene.
    Bei DFS entstehen bei einem Graphen keine Querkanten. Eine mögliche Querkante reiht einen der beiden Endknoten der Kante als Nachfolger des anderen ein. Kanten zwischen Vorgänger und Nachfolger im DFS-Baum sind sowohl Vorwärts- als auch Rückwärtskanten.
    Bei einem gerichteten Graphen gibt es bei BFS neben Baumkanten Quer- und Rückwärtskanten (keine Vorwärtskanten). Eine mögliche Vorwärtskante würde bedeuten, dass ein Knoten einer tieferen Ebene zu einer höheren Ebene (Höhenunterschied mindestens 2) adjazent ist. Dann erscheint der Knoten der tieferen Ebene aber als Nachfolger des Knotens der höheren Ebene.
    Bei DFS entstehen bei einem gerichteten Graphen alle Arten von Kanten.

14. In den Anordnungen von (1) und (3) kommt G vor C, deshalb handelt es sich nicht um topologische Sortierungen.
    Die Anordnung (2) kann durch Aufrufe von Visit(H), Visit(C), Visit(J) und Visit(A) erzeugt werden. Es handelt sich somit um eine topologische Sortierung.
    Ist $v_{i_1}, \ldots, v_{i_n}$ eine topologische Sortierung, so ergeben die Aufrufe von Visit mit den Parametern $v_{i_n}, \ldots, v_{i_1}$ in der angegebenen Reihenfolge die ursprüngliche Sortierung.

15. Bei einer topologischen Sortierung steht ein Knoten an der ersten Stelle. In diesen Knoten führt keine Kante hinein. Dies ist genau dann der einzige Knoten mit dieser Eigenschaft, wenn alle übrigen Knoten von diesem erreichbar sind.

16. Angenommen, in G_{red} gäbe es gegenseitig erreichbare Knoten V_i und V_j. Dann gäbe es $v_1, v_2 \in V_i$ und $w_1, w_2 \in V_j$ und Kanten (v_1, w_1), (w_2, v_2) in G. Dann wären aber auch v_1 und w_1 gegenseitig erreichbar. Ein Widerspruch.

17. a. Wir beweisen die Aussage durch vollständige Induktion nach r:
Für $r = 1$ folgt die Aussage aus der Definition der Adjazenzmatrix. Wir zeigen jetzt den Schluss von r auf $r+1$. Ein Pfad der Länge $r+1$ von i nach j setzt sich zusammen aus einem Pfad der Länge r von i nach k und einer Kante von (k, j). Nach Induktionsvoraussetzung ist $A^r[i, k]$ die Anzahl der Wege der Länge r von i nach k. Deshalb ist $\sum_{k=1}^{n} A^r[i, k]A[k, j]$ die Anzahl der Wege der Länge $r + 1$ und

$$\sum_{k=1}^{n} A^r[i, k]A[k, j] = A^{r+1}[i, j]$$

nach der Definition des Matrizenprodukts.

b. Wir betrachten A als Adjazenzmatrix eines gerichteten Graphen. Durch Tiefensuche kann ein gerichteter Graph auf Zyklen getestet werden (Satz 5.15). Die Laufzeit ist in der Ordnung $O(n + m)$.

18. a. Der Graph der Basisrelationen:

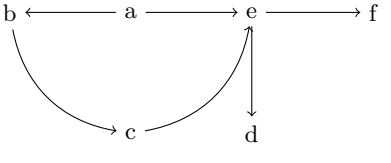
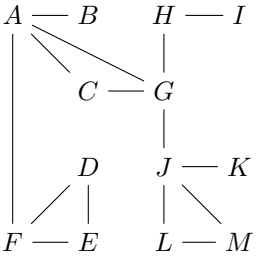


Fig. 2.3: Basisrelationen einer Anordnung.

b. Ein Zyklus $n_1 < n_2 < \ldots < n_1$ impliziert wegen der Transitivität $n_1 < n_1$, ein Widerspruch. Der Graph ist somit azyklisch. Jeder Knoten bildet eine starke Zusammenhangskomponente.

c. Alle Elemente $b \in M$ mit $b > a$ erhalten wir durch Tiefensuche mit Startpunkt a in G und alle Elemente $b \in M$ mit $b < a$ erhalten wir durch Tiefensuche mit Startpunkt a in $\mathrm{rev}(G)$, dem reversen Graphen von G.

19. a. Die Artikulationspunkte des folgenden Graphen



sind die Knoten A, F, H, G, J.
Ein Graph mit einem Artikulationspunkt besitzt mindestens drei Knoten.

b. Sei v ein Artikulationspunkt in G. Dann zerfällt $G\backslash\{v\}$ in mindestens zwei Zusammenhangskomponenten. Wähle u in der einen und w in der anderen Komponente, dann gilt, alle Wege von u nach w in G gehen durch v. Gibt es umgekehrt Knoten u und w in G, sodass alle Wege von u nach w in G durch v gehen, so ist der Knoten u nicht mehr von w erreichbar, nachdem wir v entfernen. $G \setminus \{v\}$ besitzt mindestens zwei Zusammenhangskomponenten.

c. Da im DFS-Baum von G keine Querkanten auftreten, ist die Wurzel genau dann Artikulationspunkt, wenn sie zwei Nachfolger besitzt.

d. Ist v Artikulationspunkt, so zerfällt $G \setminus \{v\}$ in mindestens zwei Zusammenhangskomponenten. Es gibt somit einen Sohn v' von v und für alle von v' erreichbaren Knoten u in T gibt es keine Rückwärtskante (u, w) zu einem Vorfahren w von v. Gibt es umgekehrt einen Sohn v' von v und für alle von v' erreichbaren Knoten u in T gibt es keine Rückwärtskante (u, w) zu einem Vorfahren w von v, so ist G ohne v nicht zusammenhängend. d. h. v ist ein Artikulationspunkt.

e. Ist v ein Artikulationspunkt und v' ein Sohn von v und für alle von v' erreichbaren Knoten u in T gibt es keine Rückwärtskante (u, w) zu einem Vorfahren w von v, dann folgt $low(v') \geq t_b(v)$. Gibt es einen Sohn v' von v mit $low(v') \geq t_b(v)$, so kann es keine Rückwärtskante von einem von v' erreichbaren Knoten zu einem Vorfahren von v geben. Insgesamt gilt: v ist genau dann ein Artikulationspunkt, wenn es einen Sohn v' von v mit $low(v') \geq t_b(v)$ gibt. Wir formulieren die Berechnung von $low(v)$ rekursiv:

$$low(v) = min(\{t_b(v)\} \cup \{t_b(w)|(v, w) \in R\} \cup \{low(v')|v' \text{ ist Sohn von } v\}).$$

Deshalb kann die Bedingung $low(v') \geq t_b(v)$ mit Tiefensuche, analog zum Algorithmus von Tarjan zur Berechnung der starken Zusammenhangskomponenten, geprüft werden. Dazu ist der Algorithmus 5.19 einfach anzupassen.

20. Seien $u, v \in V$. Wenn es zu Knoten einen Kreis gibt, der u und v, gibt es zwei disjunkte Pfade von u nach v. Entfernen wir $w \notin \{u, v\}$ so bleiben u und v gegenseitig erreichbar.
Die umgekehrte Richtung beweisen wir durch Induktion nach $d(u, v)$.
Induktionsbehauptung: Für jeden Knoten v gibt es einen Kreis mit Start- und Endknoten $u \neq v$ auf dem v liegt.
Wir führen DFS mit dem Startknoten u aus. Es gibt einen Pfad P von u nach v im DFS-Baum T von G.
Induktionsbegin $d(u, v) = 1$: v ist kein Artikulationspunk. Da in T keine Querkanten auftreten, gibt es eine Rückwärtskante von einem Nachfolger von v mit Endknoten u. Mit der Kante u, v erhalten wir einen Zyklus in G auf dem auch v liegt.
Sei $d(u, v) \geq 2$: Sei w der Vorgänger von v in T. Da $d(u, w) < d(u, v)$ ist, gibt es einen Zyklus Z mit Startknoten u der w enthält. Da w kein Artikulationspunkt ist, gibt es einen Pfad $u = v_0, \ldots, v_l = v$ von u nach v in $G \setminus \{w\}$. Sei i der größte Index mit $v_i \in Z$. Wir erhalten einen Zyklus Z' mit Startknoten u: Wir folgen Z von u bis v_i, dann $v_{i+1}, \ldots, v_l = v, w$ dann Z von w bis u. Z' ist ein Zyklus mit Start- und Endknoten u und enthält v.

6. Gewichtete Graphen

1. a. Induktionsanfang: $A(1,0) = A(0,1) = 2$.
   Induktionsvoraussetzung: $A(1,n) = n+2$.
   Induktionsbehauptung: $A(1,n+1) = (n+1)+2$.
   Induktionsschluss: $A(1,n+1) = A(0,A(1,n)) = A(0,n+2) = (n+1)+2$.

   b. Induktionsanfang:
   $A(2,0) = A(1,1) = 1+2 = 3$.
   Induktionsvoraussetzung: $A(2,n) = 2n+3$.
   Induktionsbehauptung: $A(2,n+1) = 2(n+1)+3$.
   Induktionsschluss:
   $A(2,n+1) = A(1,A(2,n)) = A(1,2n+3) = 2n+3+2 = 2(n+1)+3$.

   c. Induktionsanfang:
   $A(3,0) = A(2,1) = 2+3 = 5$.
   Induktionsvoraussetzung: $A(3,n) = 2^{n+3} - 3$.
   Induktionsbehauptung: $A(3,n+1) = 2^{(n+1)+3} - 3$.
   Induktionsschluss:
   $A(3,n+1) = A(2,A(3,n)) = A(2,2^{n+3}-3) = 2 \cdot (2^{n+3}-3) + 3 = 2^{(n+1)+3} - 3$.

   d. Induktionsanfang:

$$A(4,0) = A(3,1) = 2^4 - 3 = 13.$$

$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+3 \text{ mal}} - 3 = 2^{2^2} - 3 = 16 - 3 = 13.$$

   Induktionsvoraussetzung:

$$A(4,n) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+3 \text{ mal}} - 3$$

   Induktionsbehauptung:

$$A(4,n+1) \; \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+4 \text{ mal}} - 3.$$

   Induktionsschluss:

$$A(4,n+1) = A(3,A(4,n)) \;=\; A(3, \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+3 \text{ mal}} - 3)$$

$$= 2^{\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+3 \text{ mal}} - 3 + 3} - 3 \;=\; \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}}_{n+4 \text{ mal}} - 3.$$

2. a. Das Axiom „positiv definit“ und die Dreiecksungleichung sind verletzt.
   b. Dijkstras Algorithmus liefert den Pfad 4 - 3 - 1 - 2 der Länge 4. Der Pfad 4 - 5 - 2 besitzt die Länge 3.
   Dijkstras Algorithmus liefert somit kein korrektes Ergebnis bei negativen Gewichten.
   c. Kruskals Algorithmus liefert einen minimalen aufspannenden Baum vom Gewicht 2 mit den Kanten (1,2), (1,3), (2,5) und (3,4).
   Dies gilt ganz allgemein, weil der Korrektheitsbeweis auch gültig ist, falls Kanten mit negativem Gewicht vorhanden sind. Der Beweis benutzt nur Vergleichsargumente, die auch bei negativen Kanten gelten.

3. Wir setzen voraus, dass der Knoten 1 der Startknoten und der Knoten n der Endknoten ist. In den Startknoten führen keine Kanten hinein und aus dem Endknoten führen keine Kanten heraus. Weiter nehmen wir an, dass n von 1 aus erreichbar ist.
   Der folgende Algorithmus berechnet einen längsten Pfad von 1 nach n. Wir verwenden mit 0 initialisierte Arrays $succ[1..n]$ und $len[1..n]$. Das Array $succ[k]$ speichert den Nachfolger eines Knotens auf dem längsten Pfad von k nach n und $len[k]$ speichert die Länge eines längsten Pfades von k nach n.

   Algorithmus 6.1.
   vertex $parent[1..n]$, $succ[1..n]$; *node* $adl[1..n]$; *boolean* $visited[1..n]$

```
Visit(vertex k)
 1  node no
 2  visited[k] ← true
 3  no ← adl[k]
 4  while no ≠ null do
 5      if visited[no.v] = false
 6        then parent[no.v] ← k; Visit(no.v)
 7             if len[no.v] + no.w > len[k]
 8               then len[k] ← len[no.v] + no.w; succ[k] = no.v
 9      no ← no.next
```

   Nach Terminierung des Aufrufs Visit(1) enthält $len[1]$ die Länge eines kritischen Pfades und mithilfe des Arrays $succ$ kann ein kritischer Pfad dargestellt werden.

4. Das Problem die Abstände von einem Knoten s zu allen Knoten in G zu berechnen, kann für einen azyklischen Graphen durch Tiefensuche in der Zeit $O(n + m)$ gelöst werden.
   Wir starten die Tiefensuche in s und erzeugen den DFS-Baum. In jedem Knoten k zeichnen wir den Abstand zwischen s und k im DFS-Baum auf. Ist k Endknoten einer Quer- oder Vorwärtskante, so vermerken wir für

k den neuen Abstand von s, der sich daraus ergibt. Da G azyklisch ist treten keine Rückwärtskanten auf.
Wir wenden nochmals Tiefensuche im DFS-Baum an und berechnen die Abstände unter Einbeziehung der Abstände, die von Quer- und Vorwärtskanten herrühren. Tiefensuche im DFS-Baum erfolgt mit der Laufzeit $O(n)$. Insgesamt erhalten wir die Laufzeit $O(n+m)$.

5. Besitzen alle Kanten verschiedenes Gewicht, dann ist der minimale aufspannende Baum und damit auch das Ergebnis von Kruskals Algorithmus eindeutig.
Bei gleichen Gewichten ergibt sich der Freiheitsgrad von Kruskals Algorithmus aus der Wahl der Reihenfolge der Kanten gleichen Gewichts. Wir sortieren die Kantenliste E von G aufsteigend nach Gewichten. Dann setzen wir die Kanten des gegebenen minimalen aufspannenden Baumes T in jeder Gruppe von Kanten gleichen Gewichts an die erste Stelle der Gruppe. Werden die Kanten in dieser Reihenfolge verarbeitet, so führen die Kanten aus T zu keinen Zyklen. Sie werden für den durch Kruskals Algorithmus konstruieren minimalen aufspannenden Baum aufgenommen.

6. Die Datenstruktur Priority-Queue verallgemeinert Queue und Stack. Bei der Vergabe von aufsteigenden Prioritäten simuliert die Priority-Queue eine Queue und bei Vergabe von absteigenden Prioritäten einen Stack.

7. a. Durch Hinzunahme einer Kante e zu T entsteht ein Zyklus Z. Wir ermitteln den letzten gemeinsamen Vorfahren der Endknoten von e und damit Z (erfolgt in der Zeit $O(|V_T|)$ (Satz 6.15)). Alternativ kann Z auch durch Tiefensuche ermittelt werden (erfolgt auch in der Zeit $O(|V_T|)$.
   Wir entfernen die Kante maximalen Gewichts e' aus $T \cup \{e\}$. Der Baum $(T \cup e) \setminus \{e'\}$ ist ein minimaler aufspannender Baum von $G \cup \{e\}$.
   b. Füge an G und an den MST T zunächst den Knoten mit der Kante geringsten Gewichts an. Das Ergebnis bezeichnen wir mit G' und T'. T' ist ein minimaler aufspannender Baum für G'. Die restlichen Kanten fügen wir mit der Methode aus Punkt a hinzu.

8. Sei e die Kante von G, deren Gewicht um x verändert wird. Wir unterscheiden:
   a. Die Kante e ist eine Baumkante.
      i. $x < 0$. Die Änderung hat keine Auswirkung auf T. Wir positionieren in der Kruskals Algorithmus zugrunde liegenden Sortierreihenfolge der Kanten nach Gewichten, die Kante e als erste Kante unter denen mit gleichem Gewicht. Kruskals Algorithmus, angewendet auf die neue Sortierung liefert T.
      ii. $x > 0$. Falls es eine Kante $e' \notin T$ mit $w(e) \leq w(e') < w(e) + x$ gibt, sodass durch Hinzunahme von $\{e'\}$ zu $T \setminus \{e\}$ kein Zyklus entsteht, ist T nicht mehr MST nach der Gewichtsänderung.

b. Die Kante e ist keine Baumkante. T ist ein minimaler aufspannender Baum von $G' = G \setminus \{e\}$. Jetzt können wir mit 7. a den minimalen aufspannenden Baum von $G = G' \cup \{e\}$ bestimmen.

9. Die Wahrscheinlichkeit dafür, dass die Kante (u, v) nicht ausfällt, ist $q_{(u,v)} = 1 - \mathrm{p}(u, v)$. Die Wahrscheinlichkeit, dass eine Verbindung $v_0, \ldots, v_n$ von v_0 nach v_n nicht ausfällt ist $\prod_{i=1}^{n} q_{(v_{i-1},v_i)}$.

$$\log(\prod_{i=1}^{n} q_{(v_{i-1},v_i)}) = \sum_{i=1}^{n} \log(q_{(v_{i-1},v_i)}) =: p.$$

p ist genau dann maximal, wenn $-p$ minimal ist. Wir gewichten eine Kante (u, v) mit $-\log(q_{(u,v)})(> 0)$. Dann besitzt ein Weg, der zwei Knoten verbindet, genau dann maximale Wahrscheinlichkeit dafür, dass eine Verbindung längs des Wegs nicht ausfällt, wenn der Weg minimale Länge in einem gewichteten Graphen besitzt, d. h. gleich dem Abstand von zwei Knoten in einem gewichteten Graphen ist. Wir berechnen den Abstand mit Dijkstras Algorithmus.

10. Wir modellieren das Problem mit einem Zustandsgraphen. Die Knoten sind durch die Personen, die den Fluss noch überqueren müssen und durch die Position der Fackel definiert. Zum Beispiel ist $\{P_1, P_2, P_3, P_4, F\}$ der Startzustand und der $\{\overline{F}\}$ der Endzustand.
Überqueren zum Beispiel P_1 und P_2 den Steg, so ist eine Kante von $\{P_1, P_2, P_3, P_4, F\}$ nach $\{P_3, P_4, \overline{F}\}$ mit dem Gewicht 10 zu zeichnen. Läuft P_1 zurück, dann erfolgt ein Übergang zu $\{P_1, P_3, P_4, F\}$.
Das Problem kann mit Dijkstras Algorithmus gelöst werden. Kürzester Pfad: $\{P_1, P_2, P_3, P_4, F\} \to \{P_3, P_4, \overline{F}\} \to \{P_2, P_3, P_4, F\} \to \{P_4, \overline{F}\} \to \{P_1, P_4, F\} \to \{\overline{F}\}$

11. a. $r(G) = \min_{i=1,\ldots,n} e[i] \leq \max_{i=1,\ldots,n} e[i] = d(G)$. Seien i und j Knoten mit $d(i, j) = d(G)$ und k ein Mittelpunkt. Dann gilt $d(i, j) \leq d(i, k) + d(k, j) \leq \max_{i=1,\ldots,n} d(i, k) + \max_{j=1,\ldots,n} d(k, j) = e(k) + e(k) = 2r(G)$.

b. i. Berechne die Abstandsmatrix $A[1..n, 1..n]$ für G mit dem Algorithmus von Floyd (Algorithmus 6.57).

ii. Berechne das Array $e[1..n]$, $r(G)$ und $d(G)$:

$$e[i] = \max_{j=1,\ldots,n} d(i, j), r(G) = \min_{i=1,\ldots,n} e[i] \text{ und } d(G) = \max_{i=1,\ldots,n} e[i]$$

und alle i mit $e[i] = r(G)$.

c. Berechne in jedem Schritt die Exzentrizität für einen Knoten mit dem Algorithmus von Dijkstra (Abschnitt 6.2). Berechne dabei das Minimum m der bereits berechneten Exzentrizitäten. Im nächsten Knoten kann Dijkstras Algorithmus abgebrochen werden, sobald ein Abstand $> m$ gefunden ist.

12. a. Starte mit einem vollständigen gewichteten Graphen G mit den Städten als Knoten und den Entfernungen als Gewichte.
    b. Berechne einen minimalen aufspannenden Baum T für G.
    c. Berechne die Abstandsmatrix A für T. Bilde das Array $d[1..n]$, definiert durch $d[i] = \sum_j A[j,i]$.
    d. Bestimme die Minima in d (Mittelpunkte). Diese erfüllen die Bedingung für Punkt c.

13. Sei $d(v)$ der Abstand von v von der Wurzel r in T. Falls für alle $u \in V$

$$d(u) + g(u,v) \geq d(v) \text{ für } v \in U(u)$$

gilt, ist jeder Pfad in T auch ein kürzester Pfad in G.
Angenommen, es gibt einen Knoten w und einen Pfad $s = v_0, \ldots, v_l = w$ in G mit $\sum_{i=1}^{l} g(v_{i-1}, v_i) < d(w)$. Sei j der kleinste Index mit $\sum_{i=1}^{j} g(v_{i-1}, v_i) < d(v_j)$. Die Kante (v_{j-1}, v_j) ist keine Baumkante. Für $u = v_{j-1}$ und $v = v_j$ gilt $d(u) + g(u,v) < d(v)$, ein Widerspruch.
Die Bedingung kann durch Traversieren von T getestet werden. In Jedem Knoten u sind $\deg(u)$ viele Vergleiche notwendig, insgesamt $\sum_{u \in V} \deg(u) = 2m$ viele Vergleiche.

14. Sei $G_T = (V_T, E_T)$ der transitive Abschluss von G. Folgende Bedingungen sind äquivalent:
    a. b ist erfüllbar.
    b. Für alle Knoten v gilt $(v, \overline{v}) \notin E_T$ oder $(\overline{v}, v) \notin E_T$.

    a. $\Longrightarrow$b: Angenommen es gibt einen Knoten v mit $(v, \overline{v}) \in E_T$ und $(\overline{v}, v) \in E_T$. Dann folgt $v \Longrightarrow \overline{v}$ und $\overline{v} \Longrightarrow v$. Für eine erfüllende Belegung für b gelte $v = 0$ (ohne Einschränkung). Dann gilt $\overline{v} = 1$ und $1 \Longrightarrow 0$ ein Widerspruch.
    Es gilt ganz allgemein: $(v,w) \in E_T$ genau dann, wenn $(\overline{w}, \overline{v}) \in E_T$.
    b. $\Longrightarrow$a: Wir definieren eine erfüllende Belegung der Variablen $x_1, \ldots, x_n$ iterativ. Sei v ein Knoten für den keine Belegung definiert ist und sei $(v, \overline{v}) \notin E_T$ (ohne Einschränkung der Allgemeinheit annehmbar, sonst betrachte $(\overline{v}, v)$), d. h. in G gibt es keinen Weg, der von v nach $\overline{v}$ führt. Sei Z_v die Menge der von v aus erreichbaren Knoten in G. Für $w \in Z_v$ gilt $\overline{w} \notin Z_v$. Denn angenommen, $w \in Z_v$ und $\overline{w} \in Z_v$, dann gilt wegen $(\overline{w}, \overline{v}) \in E_T$ auch $(v, \overline{v}) \in E_T$, ein Widerspruch. Wir setzen für $w \in Z_v$ $w = 1$ und $\overline{w} = 0$. Wir wiederholen die Konstruktion mit einem Knoten für den keine Belegung definiert ist, solange bis alle Knoten eine Belegung aufweisen. Diese Belegung ist eine erfüllende Belegung für b, da wir in jedem Schritt für alle Knoten, die von w aus erreichbar sind, die Belegung mit 1 vornehmen (die Implikation $1 \Longrightarrow 0$ kann nicht auftreten).
    Um zu entscheiden, ob b erfüllbar ist, ist zu prüfen, ob es kein v gibt, sodass v und $\overline{v}$ gegenseitig erreichbar sind. v und $\overline{v}$ sind genau dann gegenseitig erreichbar, wenn sie in derselben starken Zusammenhangskomponente liegen.

Die starken Zusammenhangskomponenten können für einen gerichteten Graphen mit n Knoten und m Kanten mittels Tiefensuche in der Zeit $O(n+m)$ berechnet werden (siehe Abschnitt 5.6).
Eine alternative Lösung erhalten wir, wenn wir den transitiven Abschluss G_T von G berechnen und prüfen, ob es keinen Knoten v mit $(v, \overline{v}) \in E_T$ and $(\overline{v}, v) \in E_T$ gibt. Trifft dies zu, so ist b erfüllbar.

15. Der berechnete Pfad mit Zunahme ist S - H - J - F - B - F. Seine Zunahme ist 5. Nach Durchführung der Flusserhöhung ist im Restgraphen von S aus nur noch D und H erreichbar. Der maximale totale Fluss ist 70 und ein Schnitt minimaler Kapazität ist {S,D,H}.

16. Seien $p_1, \ldots, p_n$ die Prozesse die fest P und $q_1, \ldots, q_m$, die Prozesse, die fest Q zugeordnet sind. Die noch zuzuordnenden Prozesse bezeichnen wir mit $pq_1, \ldots, pq_l$. Wir modellieren mit einem gewichteten Graphen. Die Knoten sind P, Q, $p_1, \ldots, p_n$, $q_1, \ldots, q_m$ und $pq_1, \ldots, pq_l$. Wir definieren Kanten $(P, p_1), \ldots, (P, p_n)$, $(Q, q_1), \ldots, (Q, q_m)$ mit der Kapazität ∞. Die Kapazität der Kanten zwischen den Prozessen ist durch den Kommunikationsaufwand zwischen den Prozessen gegeben.
Die Verteilung der Prozesse kann jetzt durch die Berechnung eines Schnitts minimaler Kapazität vorgenommen werden. Dieser wiederum kann auf die Berechnung eines maximalen Flusses und anschließende Breitensuche im Restgraphen zurückgeführt werden.

17. Wir reduzieren das Problem auf das Flussproblem mit einer Quelle S und einer Senke T. Wir erweitern den Graphen um eine Quelle S, eine Senke T, Kanten (S,S_i,c_i), $i = 1, \ldots, n$, und (T_i,T,d_i), $i = 1, \ldots, m$. Wir setzen c_i gleich der Summe der Kapazitäten der Kanten, die S_i als Anfangsknoten besitzen und d_i gleich der Summe der Kapazitäten der Kanten, die in T_i enden. Anschließend wenden wir den Algorithmus von Ford-Fulkerson in der Variante von Edmonds-Karp an.

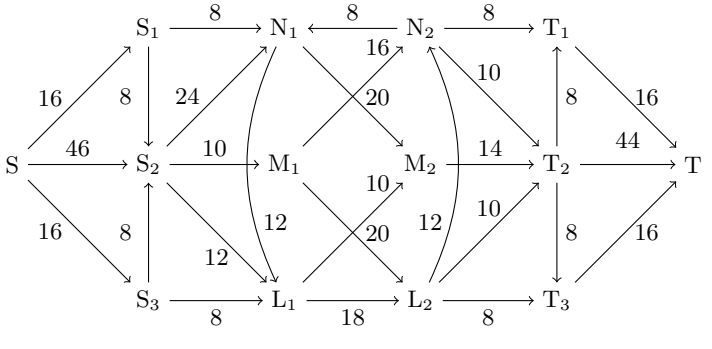


Fig. 2.4: Variante.

18. Wir ordnen dem bipartiten Graphen $G = (V \cup W, E)$ ein Flussnetzwerk $N = (V \cup W, E, s, t)$ mit Kapazität $c : E \longrightarrow \mathbb{Z}$ und Fluss $f : E \longrightarrow \mathbb{Z}$ zu. Wir nehmen zu den Knoten von G zwei zusätzliche Knoten s und t hinzu. Die Kanten aus E werden zu gerichteten Kanten mit der Richtung von V nach W. Weiter nehmen wir für jeden Knoten v aus V eine Kante (s, v) und für jeden Knoten w aus W eine Kante (w, t) hinzu. Alle diese Kanten erhalten die Kapazität 1.
    Sei Z eine Zuordnung in G. Für jede Kante $e = (v, w)$ aus Z definieren wir $f(e) = 1$, $f(s, v) = 1$ und $f(w, t) = 1$. Kapazitätsschranke und Flusserhaltung werden eingehalten. Die Anzahl der Kanten in Z ist gleich dem f zugeordneten totalen Fluss F.
    Sei f ein Fluss für N, der mit dem Algorithmus von Ford-Fulkerson mithilfe von zunehmenden Pfaden berechnet wurde. Die Auswertung eines Pfades mit Zunahme erzeugt einen Fluss f mit $f(e) = 0$ oder $f(e) = 1$. Da die Kanten von s nach V und von W nach t die Kapazität 1 besitzen, besitzen alle Kanten mit Fluss 1 verschiedene Endpunkte. Sie definieren eine Zuordnung Z. Die Anzahl der Kanten in Z ist gleich dem f zugeordneten totalen Fluss F.
    Insgesamt folgt, dass die Anzahl der Kanten einer maximalen Zuordnung in G gleich dem maximalen totalen Fluss F im G zugeordneten Netzwerk ist.
    Die Berechnung einer maximalen Zuordnung in G wird auf die Berechnung eines maximalen totalen Flusses in N reduziert.

19. a. Ein Pfad mit Zunahme startet in s und endet in t. Die Anzahl $n - 1$ der Kanten eines Pfades, der von einem Knoten aus V_1 startet und in einem Knoten aus V_2 endet, ist ungerade. Insgesamt besitzt der Pfad mit Zunahme $n + 1$ viele Kanten.
    b. Da die Kanten alle von V_1 nach V_2 gerichtet sind und der Fluss längs einer Kante nur die Werte 0 oder 1 annehmen kann, ist $e_1 \in Z$, $e_2 \notin Z$, $e_3 \in Z$, ...
    c. Wegen Punkt b beträgt die Flusserhöhung längs eines Pfades mit Zunahme 1. Deshalb nimmt auch die Anzahl der Kanten der dem Fluss zugeordneten Zuordnung in jedem Schritt um 1 zu.
    d. Die Anzahl k der Kanten in einer Zuordnung ist durch $\min\{|V_1|,|V_2|\} = O(|V|)$ beschränkt. Deshalb ist die Anzahl der Iterationen von Ford-Fulkerson in der Ordnung $O(m)$, wobei m gleich der Anzahl der Kanten von G ist.