

Algorithmen und Datenstrukturen

Übungsgruppe 14

Utz Pöhlmann

4pohlma@informatik.uni-hamburg.de
6663579

Louis Kobras

4pohlma@informatik.uni-hamburg.de
6658699

Rene Ogniwek

reneogniw@gm@.net
6425103

Steffi Kausow

s.kausow@gmail.com
6414862

8. Januar 2016

Punkte für den Hausaufgabenteil:

7.1	Σ
-----	---

Inhaltsverzeichnis

1	Zettel vom 14.-16. Oktober – Abgabe: N/A	1
1.1	Präsenzaufgabe 1.1	1
1.2	Präsenzaufgabe 1.2	1
1.3	Präsenzaufgabe 1.3	2
2	Zettel vom 15.10. – Abgabe: 26.10.	3
2.1	Übungsaufgabe 2.1	3
2.1.1	3
2.1.2	3
2.2	Übungsaufgabe 2.2	3
2.3	Übungsaufgabe 2.3	6
2.4	Übungsaufgabe 2.4	7
2.4.1	2.4.1	7
2.4.2	2.4.2	8
3	Zettel vom 29. 10. – Abgabe: 09. 11.	10
3.1	Übungsaufgabe 3.1	10
3.1.1	Aufgabe 3.1.1	10
3.1.2	Aufgabe 3.1.2	10
3.1.3	Aufgabe 3.1.3	10
3.2	Übungsaufgabe 3.2	10
3.3	Übungsaufgabe 3.3	11
3.3.1	Aufgabe 3.3.1	12
3.3.2	Aufgabe 3.3.2	13
3.3.3	Aufgabe 3.3.3	14
3.4	Übungsaufgabe 3.4	14
3.4.1	Aufgabe 3.4.1	15
3.4.2	Aufgabe 3.4.2	16
4	Zettel vom 09. 10. – Abgabe: 23. 11.	18
4.1	Übungsaufgabe 4.1	18
4.1.1	Aufgabe 4.1.1	18
4.1.2	Aufgabe 4.1.2	18
4.2	Übungsaufgabe 4.2	18
4.2.1	Aufgabe 4.2.1	19
4.2.2	Aufgabe 4.2.2	19
4.2.3	Aufgabe 4.2.3	20
4.3	Übungsaufgabe 4.3	20
4.3.1	Aufgabe 4.3.1	20
4.3.2	Aufgabe 4.3.2	20
	Zettel vom 26. 10. – Abgabe: 07. 12.	22
	Übungsaufgabe 5.1	22
	Aufgabe 5.1.1	22
	Aufgabe 5.1.2	22
	Übungsaufgabe 5.2	23
	Übungsaufgabe 5.3	24
	Übungsaufgabe 5.4	27
	Aufgabe 5.4.1	27
	Aufgabe 5.4.2	28

Zettel vom 10. 12. – Abgabe: 04. 01. 2016	29
Übungsaufgabe 6.1	29
Aufgabe 6.1.1	29
Aufgabe 6.1.2	29
Aufgabe 6.1.3	29
Aufgabe 6.1.4	30
Aufgabe 6.1.5	30
Aufgabe 6.1.6	30
Aufgabe 6.1.7	30
Aufgabe 6.1.8	31
Aufgabe 6.1.9	33
Übungsaufgabe 6.2	33
Aufgabe 6.2.1	34
Aufgabe 6.2.2	34
 Zettel vom 06. 01. – Abgabe: 18. 01. 2016	 36
Übungsaufgabe 7.1	36
Aufgabe 7.1.1	36
Aufgabe 7.1.2	36
Aufgabe 7.1.3	36
Aufgabe 7.1.4	36
Aufgabe 7.1.5	37

1 Zettel vom 14.-16. Oktober – Abgabe: N/A

1.1 Präsenzaufgabe 1.1

Wiederholen Sie die O -Notation und die verwandten Notationen. Wie sind die einzelnen Mengen definiert? Was bedeutet es, wenn $f \in O(g)$ gilt, was wenn $f \in \Theta(g)$ gilt und so weiter?

$$\begin{aligned} O(g(n)) : f(n) \in O(g(n)) &\Leftrightarrow \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \|f(n)\| \leq c \cdot \|g(n)\| \\ o(g(n)) : f(n) \in o(g(n)) &\Leftrightarrow \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \|f(n)\| < c \cdot \|g(n)\| \\ \Omega(g(n)) : f(n) \in \Omega(g(n)) &\Leftrightarrow \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \|f(n)\| \geq c \cdot \|g(n)\| \\ \omega(g(n)) : f(n) \in \omega(g(n)) &\Leftrightarrow \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : \|f(n)\| > c \cdot \|g(n)\| \\ \Theta(g(n)) : f(n) \in \Theta(g(n)) &\Leftrightarrow \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot \|g(n)\| \leq \|f(n)\| \leq c_2 \cdot \|g(n)\| \end{aligned}$$

1.2 Präsenzaufgabe 1.2

Beweisen Sie:

- $n^2 + 3n - 5 \in O(n^2)$
- $n^2 - 2n \in \Theta(n^2)$
- $n! \in O((n+1)!)$

Gilt im letzten Fall auch $n! \in o((n+1)!)$?

$$\begin{aligned} f(n) \in O(g(n)) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \\ f(n) &= n^2 + 3n - 5 \\ g(n) &= n^2 \\ \frac{f(n)}{g(n)} &= \frac{n^2 + 3n - 5}{n^2} \end{aligned}$$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n^2 + 3n - 5}{n^2} &= \lim_{n \rightarrow \infty} 1 + \frac{3}{n} - \frac{5}{n^2} \\ &= 1 + \frac{3}{\infty} - \frac{5}{\infty^2} \\ &= 1 + 0 + 0 \\ &= 1 < \infty \Rightarrow f(n) \in O(g(n)) \end{aligned}$$

□

$$\begin{aligned} c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot n^2 \leq n^2 - 2n \leq c_2 \cdot n^2 \\ \Leftrightarrow c_1 \leq 1 - \frac{2}{n} \leq c_2 \end{aligned}$$

Dies ist erfüllbar ab $n_0 \geq 2$, da für $n = 1$ im mittleren Ausdruck 0 herauskommt und c_1 größer als 0, aber kleiner als der mittlere Ausdruck sein muss. Ist $n \geq 2$, so kommt im mittleren Ausdruck 0,5 heraus, für c_1 lässt sich ein beliebiger Wert aus $]0; 0.5[$ wählen, sei es an dieser Stelle $\frac{1}{4}$. Als Obergrenze für c_2 lässt sich jeder Wert größer oder gleich 1 wählen, da der mittlere Ausdruck nicht größer als 1 werden kann und somit die Bedingung des "kleiner gleichsofort" erfüllt ist.

Somit wird als Ergebnis für die Belegung gewählt: $c_1 = \frac{1}{4}; c_2 = 1; n_0 = 2$. Mit dieser Belegung gilt $n^2 - 2n \in \Theta(n^2)$

□

$$\begin{aligned} f(n) \in O(g(n)) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \\ f(n) &= n! \\ g(n) &= (n+1)! = (n+1) \cdot n! \end{aligned}$$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n!}{(n+1) \cdot n!} &= \lim_{n \rightarrow \infty} \frac{1}{n+1} \\ &= \frac{1}{\infty} \\ &= 0 < \infty \Rightarrow f(n) \in O(g(n)) \end{aligned}$$

Da die Bedingung für $o(g(n))$ ist, dass der Quotient nicht nur kleiner unendlich, sondern gleich null ist, was hier wie oben gezeigt gegeben ist, gilt auch $n! \in o((n+1)!)$.

□

1.3 Präsenzaufgabe 1.3

Beweisen oder widerlegen Sie:

1. $f(n), g(n) \in O(h(n)) \Rightarrow f(n) + g(n) \in O(h(n))$
2. $f(n), g(n) \in O(h(n)) \Rightarrow f(n) \cdot g(n) \in O(h(n))$

$$\exists c_1 \in \mathbb{R}^+ \exists n_{0_1} \in \mathbb{N} \forall n \geq n_{0_1} : \|f(n)\| \leq c_1 \cdot \|h(n)\|$$

$$\exists c_2 \in \mathbb{R}^+ \exists n_{0_2} \in \mathbb{N} \forall n \geq n_{0_2} : \|g(n)\| \leq c_2 \cdot \|h(n)\|$$

$$n_0 = \max(n_{0_1}, n_{0_2})$$

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

Seien $f(n)$ und $g(n)$ Polynome zweiten Grades sowie $h(n)$ ein Polynom dritten Grades. Dann sind sowohl $f(n)$ als auch $g(n)$ durch die *limes*-Bedingung in $O(h(n))$. Das Produkt zweier Polynome zweiten Grades ist allerdings ein Polynom vierten Grades, sodass gilt:

$$\lim_{n \rightarrow \infty} \frac{n^2 \cdot n^2}{n^3} = \lim_{n \rightarrow \infty} \frac{n^4}{n^3} = \lim_{n \rightarrow \infty} n = \infty$$

Damit ist das Produkt der Polynome nicht mehr in $O(h(n))$, da die *limes*-Bedingung, nach der der Quotient der Polynome für n gegen Unendlich kleiner als Unendlich sein zu hat, nicht erfüllt ist. Damit ist (2) widerlegt.

□

2 Zettel vom 15.10. – Abgabe: 26.10.

2.1 Übungsaufgabe 2.1

[| 2]

Begründen Sie formal, warum folgende Größenabschätzungen gelten bzw. nicht gelten:

1. $3n^3 - 6n + 20 \in O(n^3)$
2. $n^2 \cdot \log n \in O(n^3) \cap \Omega(n^2)$

2.1.1

$$\begin{aligned} 3n^3 - 6n + 20 \in O(n^3) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{3n^3 - 6n + 20}{n^3} < \infty \\ \lim_{n \rightarrow \infty} \frac{3n^3 - 6n + 20}{n^3} &= \lim_{n \rightarrow \infty} \frac{3n^3}{n^3} - \frac{6n}{n^3} + \frac{20}{n^3} = \lim_{n \rightarrow \infty} 3 - \frac{6}{n^2} + \frac{20}{n^3} = 3 - 0 + 0 < \infty \\ &\Rightarrow 3n^3 - 6n + 20 \in O(n^3) \quad \square \end{aligned}$$

2.1.2

$$\begin{aligned} n^2 \cdot \log n \in O(n^3) \cap \Omega(n^2) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{n^2 \cdot \log n}{n^3} < \infty \wedge \lim_{n \rightarrow \infty} \frac{n^2 \cdot \log n}{n^2} > 0 \\ \frac{n^2 \cdot \log n}{n^2} &= \frac{1 \cdot \log n}{1} = \log n > 0 \quad \forall n > 1 \Rightarrow n^2 \cdot \log n \in \Omega(n^2) \\ \lim_{n \rightarrow \infty} \frac{n^2 \cdot \log n}{n^3} &= \lim_{n \rightarrow \infty} \frac{\log n}{n} \stackrel{\text{rH}}{=} \lim_{n \rightarrow \infty} \frac{1}{n} \cdot \frac{1}{1} = \lim_{n \rightarrow \infty} \frac{1}{n} = \frac{1}{\infty} = 0 \Rightarrow n^2 \cdot \log n \in O(n^3) \\ &\Rightarrow n^2 \cdot \log n \in O(n^3) \cap \Omega(n^2) \quad \square \end{aligned}$$

2.2 Übungsaufgabe 2.2

[| 4]

Ordnen Sie die folgenden Funktionen nach ihrem Wachstumsgrad in aufsteigender Reihenfolge, d.h. folgt eine Funktion $g(n)$ einer Funktion $f(n)$, so soll $f(n) \in O(g(n))$ gelten.

$$n, \log n, n^2, n^{\frac{1}{2}}, \sqrt{n^3}, 2^n, \ln n, 1000$$

Mit \log ist hier der Logarithmus zur Basis 2, mit \ln der natürliche Logarithmus (Basis e) gemeint. Begründen Sie stets Ihre Aussage. Zwei Funktionen $f(n)$ und $g(n)$ befinden sich ferner in der selben Äquivalenzklasse, wenn $f(n) \in \Theta(g(n))$ gilt. Geben Sie an, welche Funktionen sich in derselben Äquivalenzklasse befinden und begründen Sie auch hier ihre Aussage.

Die bearbeitete Menge wird i.F. als M_F bezeichnet. Die Menge, die gerade alle Elemente von M_F in aufsteigend sortierter Reihenfolge enthält, wird als M'_F bezeichnet.

M_F wird mit INSERTSORT in M'_F hineinsortiert.

Sei $e \in M_F$. Für e wird das Element 1000 gewählt. Da $|M'_F|$ leer ist, muss 1000 nicht weiter geprüft werden.

$$M'_F = \{1000\}$$

e wird nun über M_F iteriert, bis $M'_F = \text{Sorted}(M_F)$.

$$e = n$$

$$\begin{array}{l} f(n) = n \\ g(n) = 1000 \end{array} \quad \lim_{n \rightarrow \infty} \frac{n}{1000} = \infty \Rightarrow n \notin O(1000)$$

$$\begin{array}{l} f(n) = 1000 \\ g(n) = n \end{array} \quad \lim_{n \rightarrow \infty} \frac{1000}{n} = 0 \Rightarrow 1000 \in O(n) \Rightarrow n \text{ folgt } 1000$$

$$M'_F = \{1000, n\}$$

$$e = \log n$$

$$\begin{array}{l} f(n) = \log(n) \\ g(n) = n \end{array} \quad \lim_{n \rightarrow \infty} \frac{\log(n)}{n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{\ln(2) \cdot n}}{1} = \lim_{n \rightarrow \infty} \frac{1}{\ln(2) \cdot n} = 0 \Rightarrow \log(n) \in O(n) \Rightarrow n \text{ folgt } \log(n)$$

$$\begin{array}{l} f(n) = \log(n) \\ g(n) = 1000 \end{array} \quad \lim_{n \rightarrow \infty} \frac{\log(n)}{1000} = \infty \Rightarrow \log(n) \notin O(1000)$$

$$M'_F = \{1000, \log n, n\}$$

$$e = 4$$

$$\begin{array}{l} f(n) = 4 \\ g(n) = n \end{array} \quad \lim_{n \rightarrow \infty} \frac{4}{n} = 0 \Rightarrow 4 \in O(n) \Rightarrow n \text{ folgt } 4$$

$$\begin{array}{l} f(n) = 4 \\ g(n) = \log(n) \end{array} \quad \lim_{n \rightarrow \infty} \frac{4}{\log(n)} = 0 \Rightarrow 4 \in O(\log(n)) \Rightarrow \log(n) \text{ folgt } 4$$

$$\begin{array}{l} f(n) = 4 \\ g(n) = 1000 \end{array} \quad \lim_{n \rightarrow \infty} \frac{4}{1000} = 0,004 \Rightarrow 4 \in \Theta(1000) \Rightarrow 4 \text{ und } 1000 \text{ befinden sich in der selben } \ddot{\text{A}}\text{-klasse}$$

$$M'_F = \{4, 1000, \log n, n\}$$

$$e = n^2$$

$$\begin{array}{l} f(n) = n^2 \\ g(n) = n \end{array} \quad \lim_{n \rightarrow \infty} \frac{n^2}{n} = \infty \Rightarrow n^2 \notin O(n)$$

$$\begin{array}{l} f(n) = n \\ g(n) = n^2 \end{array} \quad \lim_{n \rightarrow \infty} \frac{n}{n^2} = 0 \Rightarrow n \in O(n^2) \Rightarrow n^2 \text{ folgt } n$$

$$M'_F = \{4, 1000, \log n, n, n^2\}$$

$$e = n^{\frac{1}{2}}$$

$$f(n) = n^{\frac{1}{2}}$$

$$g(n) = n^2 \quad \lim_{n \rightarrow \infty} \frac{n^{\frac{1}{2}}}{n^2} = \lim_{n \rightarrow \infty} \frac{1}{n^{\frac{3}{2}}} = 0 \quad \Rightarrow n^{\frac{1}{2}} \in O(n^2) \quad \Rightarrow n^2 \text{ folgt } n^{\frac{1}{2}}$$

$$f(n) = n^{\frac{1}{2}}$$

$$g(n) = n \quad \lim_{n \rightarrow \infty} \frac{n^{\frac{1}{2}}}{n} = \lim_{n \rightarrow \infty} \frac{1}{n^{\frac{1}{2}}} = 0 \quad \Rightarrow n^{\frac{1}{2}} \in O(n) \quad \Rightarrow n \text{ folgt } n^{\frac{1}{2}}$$

$$f(n) = n^{\frac{1}{2}}$$

$$g(n) = \log(n) \quad \lim_{n \rightarrow \infty} \frac{n^{\frac{1}{2}}}{\log(n)} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{2} \cdot n^{-\frac{1}{2}}}{\frac{1}{n(2) \cdot n}} = \lim_{n \rightarrow \infty} \frac{\ln(2) \cdot n^{\frac{1}{2}}}{2} = \infty \quad \Rightarrow n^{\frac{1}{2}} \notin O(\log(n))$$

$$M'_F = \{4, 1000, \log n, n^{\frac{1}{2}}, n, n^2\}$$

$$e = \sqrt{n}^3$$

$$f(n) = \sqrt{n}^3$$

$$g(n) = n^2 \quad \lim_{n \rightarrow \infty} \frac{\sqrt{n}^3}{n^2} = \lim_{n \rightarrow \infty} \frac{1}{n^{\frac{1}{2}}} = 0 \quad \Rightarrow \sqrt{n}^3 \in O(n^2) \quad \Rightarrow n^2 \text{ folgt } \sqrt{n}^3$$

$$f(n) = \sqrt{n}^3$$

$$g(n) = n \quad \lim_{n \rightarrow \infty} \frac{\sqrt{n}^3}{n} = \lim_{n \rightarrow \infty} n^{\frac{1}{2}} = \infty \quad \Rightarrow \sqrt{n}^3 \notin O(n)$$

$$M'_F = \{4, 1000, \log n, n^{\frac{1}{2}}, n, \sqrt{n}^3, n^2\}$$

$$e = 2^n$$

$$f(n) = 2^n$$

$$g(n) = n^2 \quad \lim_{n \rightarrow \infty} \frac{2^n}{n^2} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\ln(2) \cdot 2^n}{2 \cdot n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\ln^2(2) \cdot 2^n}{2} = \infty \quad \Rightarrow 2^n \notin O(n^2)$$

$$f(n) = n^2$$

$$g(n) = 2^n \quad \lim_{n \rightarrow \infty} \frac{n^2}{2^n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{2 \cdot n}{\ln(2) \cdot 2^n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{2}{\ln^2(2) \cdot 2^n} = 0 \quad \Rightarrow n^2 \in O(2^n) \quad \Rightarrow n^2 \text{ folgt } 2^n$$

$$M'_F = \{4, 1000, \log n, n^{\frac{1}{2}}, n, \sqrt{n}^3, n^2, 2^n\}$$

$$e = \ln n$$

$$f(n) = \ln(n)$$

$$g(n) = 2^n \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{2^n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\ln(2) \cdot 2^n} = \lim_{n \rightarrow \infty} \frac{1}{n \cdot \ln(n) \cdot 2^n} = 0 \quad \Rightarrow \ln(n) \in O(2^n) \quad \Rightarrow 2^n \text{ folgt } \ln(n)$$

$$f(n) = \ln(n)$$

$$g(n) = n^2 \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{n^2} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{2 \cdot n} = \lim_{n \rightarrow \infty} \frac{1}{2 \cdot n^2} = 0 \quad \Rightarrow \ln(n) \in O(n^2) \quad \Rightarrow n^2 \text{ folgt } \ln(n)$$

$$f(n) = \ln(n)$$

$$g(n) = \sqrt{n}^3 \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{\sqrt{n}^3} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{3}{2} \sqrt{n}} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{1}{6 \cdot n^{\frac{3}{2}}} = 0 \quad \Rightarrow \ln(n) \in O(\sqrt{n}^3) \quad \Rightarrow \sqrt{n}^3 \text{ folgt } \ln(n)$$

$$f(n) = \ln(n)$$

$$g(n) = n \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{n} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{1} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0 \quad \Rightarrow \ln(n) \in O(n) \quad \Rightarrow n \text{ folgt } \ln(n)$$

$$f(n) = \ln(n)$$

$$g(n) = n^{\frac{1}{2}} \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{n^{\frac{1}{2}}} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2} \cdot n^{\frac{1}{2}}} = \lim_{n \rightarrow \infty} \frac{2}{n^{\frac{3}{2}}} = 0 \quad \Rightarrow \ln(n) \in O(n^{\frac{1}{2}}) \quad \Rightarrow n^{\frac{1}{2}} \text{ folgt } \ln(n)$$

$$f(n) = \ln(n)$$

$$g(n) = \log(n) \quad \lim_{n \rightarrow \infty} \frac{\ln(n)}{\log(n)} \stackrel{\text{L'Hospital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{\ln(2) \cdot n}} = \ln(2) \quad \Rightarrow \ln(n) \in \Theta(\log(n)) \quad \Rightarrow \ln(n) \text{ und } \log(n) \text{ befinden sich in der selben } \ddot{\text{A}}\text{-klasse}$$

$$M'_F = \{4, 1000, \ln n, \log n, n^{\frac{1}{2}}, n, \sqrt{n}^3, n^2, 2^n\}$$

In der selben Äquivalenzklasse befinden sich zum einen 4 und 1000 und zum anderen $\log(n)$ und $\ln(n)$. Die restlichen Werte sind jeweils alleine in ihrer Äquivalenzklasse.

2.3 Übungsaufgabe 2.3

[| 2]

Beweisen oder widerlegen Sie:

$$f(n), g(n) \in O(h(n)) \Rightarrow f(n) \cdot g(n) \in O((h(n))^2)$$

Für diesen Beweis wird der Beweis des dritten Satzes der Summen- und Produkteigenschaften der O-Notation¹ zu Hilfe genommen:

Beweis. Sei $f \in O(h_1)$ und $g \in O(h_2)$, dann gibt es ein c, n_0 , so dass $f(n) \leq c \cdot h_1(n) \forall n \geq n_0$ und ebenso c', n'_0 , so dass $g(n') \leq c' \cdot h_2(n') \forall n' \geq n'_0$. Daraus folgt $f(n'') \cdot g(n'') \leq c \cdot c' \cdot h_1(n'') \cdot h_2(n'') \forall n'' \geq \max(n_0, n'_0)$, also $f \cdot g \in O(h_1 \cdot h_2)$. \square

Setzt man nun $h_1, h_2 = h$ folgt daraus für den letzten Ausdruck des Beweises $f(n) \cdot g(n) \in O(h(n) \cdot h(n)) \Rightarrow f(n) \cdot g(n) \in O((h(n))^2)$.

¹vgl. Vorlesung, Foliensatz 1 (14.10.), S.33

2.4 Übungsaufgabe 2.4

[| 8]

Seien

1.

$$T(n) := \begin{cases} 0, & \text{für } n = 0 \\ 3 \cdot T(n-1) + 2, & \text{sonst} \end{cases}$$

2.

$$S(n) := \begin{cases} c, & \text{für } n = 1 \\ 16 \cdot S(\frac{n}{4}) + n^2, & \text{sonst} \end{cases}$$

Rekurrenzgleichungen (c ist dabei eine Konstante).

Bestimmen Sie wie in der Vorlesung jeweils die Größenordnung der Funktion $T : \mathbb{N} \rightarrow \mathbb{N}$ einmal mittels der (a) Substitutionsmethode und einmal mittels des (b) Mastertheorems. Ihre Ergebnisse sollten zumindest hinsichtlich der O-Notation gleich sein, so dass Sie etwaige Rechenfehler entdecken können! Führen Sie bei (a) auch den Induktionsbeweis, der in der Vorlesung übersprungen wurde!

2.4.1 2.4.1

a)

$$\begin{aligned} T(n) &= 3 \cdot T(n-1) + 2 \\ &= 3 * (3 * T(n-2) + 2) + 2 = 3^2 * T(n-2) + 3^2 - 1 \\ &= 3^2 * (3 * T(n-3) + 2) + 8 = 3^3 * T(n-3) + 3^3 - 1 \\ &= \dots \\ &= 3^k * T(n-k) + 3^k - 1 \end{aligned}$$

Wir kommen auf eine sinnvolle Verallgemeinerung der Formel.

Beweis der Formel durch vollständige Induktion:

Induktionsanfang: $T(0)$ gilt nach Definition.

Induktionsschritt: Sei $n \in \mathbb{N}$ (s. Aufgabenstellung). Wir nehmen an, dass $T(n)$ gilt (Induktionsannahme) und zeigen $T(n+1)$. Es gilt

$$\begin{aligned} T(n) &= 3 * T(n-1) + 2 \\ T(n+1) &= 3 * T(n+1-1) + 2 \\ &= 3 * T(n) + 2 \end{aligned}$$

$$\begin{aligned} T(n) &= 3^k * T(n-k) + 3^k - 1 \\ T(n+1) &= 3^k * T(n+1-k) + 3^k - 1 \end{aligned}$$

Das zeigt $T(n+1)$.

Damit sind der Induktionsanfang und der Induktionsschritt bewiesen. Es folgt, dass $T(n)$ für alle $n \in \mathbb{N}$ gilt.

Da die Rekursion bei $T(0) = 0$, also $n-k = 0$ abbricht, wird mit $k = n$ weiter gerechnet.

$$\begin{aligned} T(n) &= 3^k * T(n-k) + 3^k - 1 \\ &= 3^n * T(n-n) + 3^n - 1 \\ &= 3^n * T(0) + 3^n - 1 \\ &= 3^n * 0 + 3^n - 1 \\ &= 3^n - 1 \in \Theta(3^n) \end{aligned}$$

b) Das Mastertheorem ist auf Aufgabe 1. nicht anwendbar, da die Form

$$T(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

bei

$$T(n) := \begin{cases} 0, & \text{für } n = 0 \\ 3 \cdot T(n-1) + 2, & \text{sonst} \end{cases}$$

nicht eingehalten wurde.

2.4.2 2.4.2

a)

$$\begin{aligned} S(n) &= 16 \cdot S(\frac{n}{4}) + n^2 \\ &= 16 \cdot S(\frac{16 \cdot S(\frac{n}{4}) + n^2}{4}) + n^2 \\ &= 16 \cdot S(\frac{16 \cdot S(\frac{16 \cdot S(\frac{n}{4}) + n^2}{4}) + n^2}{4}) + n^2 \\ &= 16 \cdot S(\frac{16 \cdot S(\frac{16 \cdot S(\frac{n}{4}) + n^2}{4}) + n^2}{4}) + n^2 \\ &= 16 \cdot S(\frac{16 \cdot S(\frac{16 \cdot S(\frac{16 \cdot S(\frac{n}{4}) + n^2}{4}) + n^2}{4}) + n^2}{4}) + n^2 \end{aligned}$$

Keine sinnvolle Vereinfachung erkennbar. => Substitutionsmethode nicht anwendbar.

b) Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$S(n) := \begin{cases} c, & \text{für } n = 1 \\ 16 \cdot S(\frac{n}{4}) + n^2, & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $S(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)-\epsilon}) \\ n^2 &\in O(n^{\log_4(16)-\epsilon}) \\ n^2 &\in O(n^{2-\epsilon}) \end{aligned}$$

Hierfür kann kein ϵ gefunden werden. Daher gilt diese Aussage nicht

II. $S(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)}) \\ n^2 &\in O(n^{\log_4(16)}) \\ n^2 &\in O(n^2) \end{aligned}$$

Dies stimmt, daher gilt diese Aussage.

III. $S(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n .

$$\begin{array}{ll} f(n) & \in \Omega(n^{\log_b(a)+\epsilon}) \\ n^2 & \in \Omega(n^{\log_4(16)+\epsilon}) \\ n^2 & \in \Omega(n^{2+\epsilon}) \end{array}$$

Dies stimmt für alle $\epsilon \geq 0$, also auch für mindestens ein $\epsilon > 0$.

$$\begin{array}{ll} a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n) & \backslash \text{einsetzen} \\ 16 \cdot (\frac{n}{4})^2 \leq \delta \cdot n^2 & \backslash \sqrt{()} \\ 4 \cdot \frac{n}{4} \leq \sqrt{\delta} \cdot n & \\ n \leq \sqrt{\delta} \cdot n & \backslash :n \text{ (n ist immer positiv, da } n \in \mathbb{N}, \text{ s. Aufgabenstellung)} \\ 1 \leq \sqrt{\delta} & \backslash ()^2 \\ 1 \leq \delta & \end{array}$$

Damit ist $\delta \geq 1$ und nicht, wie benötigt, $\delta < 1$. Daher gilt diese Aussage nicht.

Da nur II. gilt, gilt $S(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$, also $S(n) \in \Theta(n^2 \cdot \log_2(n))$.

3 Zettel vom 29. 10. – Abgabe: 09. 11.

3.1 Übungsaufgabe 3.1

[| 3]

Gegeben seien die folgenden Code-Fragmente. Geben Sie eine möglichst dichte asymptotische obere Schranke für die Laufzeit der einzelnen Code-Fragmente jeweils in Abhängigkeit von n an. Begründen Sie Ihre Behauptung. (Es geht hier nicht um die Bedeutung des Codes, nur um die Laufzeit. An der Stelle von $sum = sum + j$ könnte sinnvoller(er) Code stehen. Die Einrückung gibt den Skopus der Schleifenkonstrukte an.)

ALGO1()	ALGO2()	ALGO3()
1 for $i = 0$ to n	1 $i = 1$	1 $i = 1$
2 for $j = n$ downto 1	2 while $i < 2 \cdot n$	2 while $i \cdot i < n$
3 $sum = sum + j$	3 for $j = 1$ to i	3 $i = i + 1$
4 for $j = 1$ to n	4 $sum = sum + j$	4 $j = n$
5 $sum = sum + j$	5 $i = i + 2$	5 while $j > 1$
		6 $sum = sum + j$
		7 $j = j/2$

3.1.1 Aufgabe 3.1.1

Der Algorithmus läuft in $O(n^2)$.

Bei beiden inneren **for**-Blöcke laufen jeweils $n - 1$ mal durch, sodass der innere Block insgesamt $2 \cdot (n - 1) = 2n - 2$ mal ausgeführt wird. Die äußere **for**-Schleife läuft gerade n mal; also wird der Block der Länge $2n - 2$ n mal ausgeführt. Dies führt zu einer Laufzeit von $n \cdot (2n - 2) = 2n^2 - 2n \in \mathcal{O}(n^2)$.

3.1.2 Aufgabe 3.1.2

Die äußere Schleife läuft zwar bis $2 \cdot n$, da die Zählvariable allerdings in Zweierschritten inkrementiert wird, wird die Schleife n mal durchlaufen. Die innere Schleife wird $n \cdot (n - 1)$ mal durchlaufen ($1 + 3 + 5 + 7 + \dots + 2 \cdot n - 3 + 2 \cdot n - 1 = n \cdot (n - 1)$). Dies ergibt insgesamt eine Laufzeit von $n \cdot n \cdot (n - 1) = n^3 - n^2 \in \mathcal{O}(n^3)$.

3.1.3 Aufgabe 3.1.3

Die äußere Schleife wird \sqrt{n} mal durchlaufen, da $i^2 = n \Rightarrow i = \sqrt{n}$. Zeilen 5-7 laufen in $\log n$. Das ergibt eine Gesamtlaufzeit in $\mathcal{O}(\sqrt{n} \cdot \log n)$.

3.2 Übungsaufgabe 3.2

[| 2]

Gegeben sei folgender Algorithmus, der ein Array A als Eingabe erwartet (dessen genaue Funktionsweise nachfolgend aber nicht wichtig ist):

Algorithm 1 FUNC(A)

```
1: if  $A.lnge < 4$  then
2:   return 4
3: else
4:    $sum = 0$ 
5:   for  $i = 1$  to  $A.laenge$  do
6:      $sum = sum + A[i]$ 
7:   end for
8:    $x = A.laenge/4$ 
9:    $y = \text{FUNC}(A[x + 1..2 \cdot x])$ 
10:   $z = \text{FUNC}(A[3 \cdot x + 1..A.laenge])$ 
11:   $r = y$ 
12:  for  $i = 1$  to  $A.laenge$  do
13:     $r = r + z \cdot A[i]$ 
14:  end for
15:  return  $sum + r$ 
16: end if
```

Leiten Sie eine Rekurrenzgleichung für die Laufzeit der Methode FUNC in Abhängigkeit von der Arraygröße n von A ab. Begründen Sie ihre Gleichung.

$$A(n) := \begin{cases} 5, & \text{falls } n < 4 \\ A(\frac{n}{2}) + A(\frac{n}{4}) + 2n + 4, & \text{sonst} \end{cases}$$

Die 5 im ersten Teil der Rekurrenzgleichung erklärt sich durch die ersten beiden Zeilen im Pseudocode.

Der Aufruf in Zeile 8 dauert $\frac{n}{2}$, der in Zeile 9 dauert $\frac{n}{4}$. Dadurch erklärt sich das $A(\frac{n}{2}) + A(\frac{n}{4})$ im zweiten Teil der Rekurrenzgleichung.

Zeile 5 und 6 laufen in n ab.

Zeile 11 und 12 laufen auch in n ab.

Zeile 4, 7, 10 und 13 laufen jeweils mit konstantem Zeitaufwand.

Deshalb ist das, was hinter den $A(x)$ -Aufrufen steht, $n + n + c$, wobei c in diesem Fall den Wert 4 hat, da wir vier Zeilen mit konstantem Zeitaufwand im Pseudocode haben.

3.3 Übungsaufgabe 3.3

[| 3]

Gegeben seien die folgenden Rekurrenzgleichungen T_1 , T_2 und T_3 , wobei die c_i und d_i Konstanten seien.

$$\begin{aligned} T_1(n) &:= \begin{cases} c_1 & \text{für } n = 1 \\ 8 \cdot T_1(\frac{n}{2}) + d_1 \cdot n^3, & \text{sonst} \end{cases} \\ T_2(n) &:= \begin{cases} c_2 & \text{für } n = 1 \\ 5 \cdot T_2(\frac{n}{4}) + d_2 \cdot n^2, & \text{sonst} \end{cases} \\ T_3(n) &:= \begin{cases} c_3 & \text{für } n = 1 \\ 6 \cdot T_3(\frac{n}{3}) + d_3 \cdot n \cdot \log n, & \text{sonst} \end{cases} \end{aligned}$$

Bestimmen Sie die Größenordnungen der Funktionen $T_i : \mathbb{N} \rightarrow \mathbb{N}$ mittels des Mastertheorems. Ist dies nicht möglich, geben Sie an warum. Geben Sie Ihre Ergebnisse nachvollziehbar an.

3.3.1 Aufgabe 3.3.1

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_1(n) := \begin{cases} c_1, & \text{für } n = 1 \\ 8 \cdot T_1(\frac{n}{2}) + d_1 \cdot n^3, & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_1(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)-\epsilon}) \\ d_1 \cdot n^3 &\in O(n^{\log_2(8)-\epsilon}) \\ d_1 \cdot n^3 &\in O(n^{3-\epsilon}) \end{aligned}$$

Hierfür kann kein ϵ gefunden werden. Daher gilt diese Aussage nicht.

II. $T_1(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)}) \\ d_1 \cdot n^3 &\in O(n^{\log_2(8)}) \\ d_1 \cdot n^3 &\in O(n^3) \end{aligned}$$

Dies stimmt, daher gilt diese Aussage.

III. $T_1(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n .

$$\begin{aligned} f(n) &\in \Omega(n^{\log_b(a)+\epsilon}) \\ d_1 \cdot n^3 &\in \Omega(n^{\log_2(8)+\epsilon}) \\ d_1 \cdot n^3 &\in \Omega(n^{3+\epsilon}) \end{aligned}$$

Dies stimmt für alle $\epsilon \geq 0$, also auch für mindestens ein $\epsilon > 0$.

$$\begin{aligned} a \cdot f(\frac{n}{b}) &\leq \delta \cdot f(n) && \backslash \text{einsetzen} \\ 8 \cdot d_1 \cdot (\frac{n}{2})^3 &\leq \delta \cdot d_1 \cdot n^3 && \backslash \sqrt[3]{()} \\ \sqrt[3]{d_1} \cdot 2 \cdot \frac{n}{2} &\leq \sqrt[3]{\delta} \cdot \sqrt[3]{d_1} \cdot n && \backslash : \sqrt[3]{d_1} \\ n &\leq \sqrt[3]{\delta} \cdot n && \backslash : n \text{ (n ist immer positiv, da } n \in \mathbb{N}, \text{ s. Aufgabenstellung)} \\ 1 &\leq \sqrt[3]{\delta} && \backslash ()^3 \\ 1 &\leq \delta \end{aligned}$$

Damit ist $\delta \geq 1$ und nicht, wie benötigt, $\delta < 1$. Daher gilt diese Aussage nicht.

Da nur II. gilt, gilt $T_1(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$, also $T_1(n) \in \Theta(n^3 \cdot \log_2(n))$.

3.3.2 Aufgabe 3.3.2

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_2(n) := \begin{cases} c_2, & \text{für } n = 1 \\ 5 \cdot T_2(\frac{n}{4}) + d_2 \cdot n^2, & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_2(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)-\epsilon}) \\ d_2 \cdot n^2 &\in O(n^{\log_4(5)-\epsilon}) \\ d_2 \cdot n^2 &\in O(n^{1.160964047443681-\epsilon}) \end{aligned}$$

Hierfür kann kein ϵ gefunden werden. Daher gilt diese Aussage nicht.

II. $T_2(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$, falls $f(n) \in \Theta(n^{\log_b(a)})$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)}) \\ d_2 \cdot n^2 &\in O(n^{\log_4(5)}) \\ d_2 \cdot n^2 &\in O(n^{1.160964047443681}) \end{aligned}$$

Dies stimmt nicht, daher gilt diese Aussage nicht.

III. $T_2(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n .

$$\begin{aligned} f(n) &\in \Omega(n^{\log_b(a)+\epsilon}) \\ d_2 \cdot n^2 &\in \Omega(n^{\log_4(5)+\epsilon}) \\ d_2 \cdot n^2 &\in \Omega(n^{1.160964047443681+\epsilon}) \end{aligned}$$

Dies stimmt für alle $\epsilon \geq 2 - \log_4(5) \approx 0.839035952556319$, also auch für mindestens ein $\epsilon > 0$.

$$\begin{aligned} a \cdot f(\frac{n}{b}) &\leq \delta \cdot f(n) && \backslash \text{einsetzen} \\ 4 \cdot d_2 \cdot (\frac{n}{5})^2 &\leq \delta \cdot d_2 \cdot n^2 && \backslash \sqrt{(\cdot)} \\ \sqrt{d_2} \cdot 2 \cdot \frac{n}{5} &\leq \sqrt{\delta} \cdot \sqrt{d_2} \cdot n && \backslash : \sqrt{d_2} \\ \frac{2}{5} \cdot n &\leq \sqrt{\delta} \cdot n && \backslash : n \text{ (n ist immer positiv, da } n \in \mathbb{N}, \text{ s. Aufgabenstellung)} \\ \frac{2}{5} &\leq \sqrt{\delta} && \backslash ()^2 \\ \frac{4}{25} &\leq \delta \\ 0.16 &\leq \delta \end{aligned}$$

Damit gilt $\delta \geq 0.16$. Somit wurde, wie benötigt, mindestens ein $\delta < 1$ gefunden ($0.16 \leq \delta < 1$). Daher gilt diese Aussage.

Da nur III. gilt, gilt $T_2(n) \in \Theta(f(n))$, also $T_2(n) \in \Theta(d_2 \cdot n^2)$, also $T_2 \in \Theta(n^2)$.

3.3.3 Aufgabe 3.3.3

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_3(n) := \begin{cases} c_3, & \text{für } n = 1 \\ 6 \cdot T_3(\frac{n}{3}) + d_3 \cdot n \cdot \log(n), & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_3(n) \in \Theta(n^{\log_b(a)})$, falls $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{aligned} f(n) &\in O(n^{\log_b(a)-\epsilon}) \\ d_3 \cdot n \cdot \log(n) &\in O(n^{\log_3(6)-\epsilon}) \\ d_3 \cdot n \cdot \log(n) &\in O(n^{1.6309297535714573-\epsilon}) \end{aligned}$$

Es gilt: $d_3 \cdot n \cdot \log(n) < n^{\log_3(6)}$ für alle $n > 0$, für $d_3 = 1$.

Da aber nur $d_3 \cdot n \cdot \log(n) \leq n^{\log_3(6)}$ für alle $n > 1$ (s. Definition T_3) gefordert ist, kann hierfür mindestens ein ϵ gefunden werden.

Für größere d_3 gilt allerdings, dass die n nicht mehr beliebig größer 1 sein dürfen, sondern eine obere Schranke bekommen.

Somit hängt die Lösung stark von d_3 ab und das Mastertheorem ist ungeeignet für die Lösung dieser Aufgabe.

3.4 Übungsaufgabe 3.4

[| 8]

1. Geben Sie für die Funktion *merge* in *mergesort* (siehe Folie 9, Kapitel 3) sinnvollen Pseudocode an. Orientieren Sie sich dabei an der natürlichsprachlichen Formulierung auf Folie 10, Kapitel 3. Ihr Pseudocode sollte diese Formulierung möglichst nah umsetzen. Weisen Sie dann Ihren Pseudocode von *merge* als korrekt nach, indem Sie eine sinnvolle Schleifeninvariante formulieren und nutzen. Nutzen Sie dies dann, um auch *mergesort* selbst als korrekt nachzuweisen. (Siehe auch Folie 11, Kapitel 3.)
2. Gegeben sei eine Sequenz $A = \langle a_1, a_2, \dots, a_n \rangle$ von Zahlen (als Array). Ein Paar (i, j) mit $i < j$, aber $a_i > a_j$ nennen wir einen *Konflikt* (in A). Gesucht ist nun, gegebend das Array A , die Anzahl der Konflikte in A . Ihre Aufgabe ist es, einen Divide&Conquer Algorithmus zu entwerfen, der das Problem in $\mathcal{O}(n \cdot \log n)$ löst. Beschreiben Sie zunächst Ihre Idee. Geben Sie dann den Algorithmus konkreter an und begründen Sie dann überzeugend, warum Ihr Algorithmus das Problem korrekt löst (sie müssen dazu nicht unbedingt auf die Schleifeninvarianten zurückgreifen). Zuletzt begründen Sie auch, warum die Laufzeit ihres Algorithmus tatsächlich in $\mathcal{O}(n \cdot \log n)$ liegt.

3.4.1 Aufgabe 3.4.1

MERGE(A, B)

```
1   $cur_A = 0$ 
2   $cur_B = 0$ 
3   $C = newlist$ 
4  while  $cur_A < A.length \ \& \ cur_B < B.length$ 
5      if  $cur_A < A.length$ 
6           $a_i = A[cur_A]$ 
7      if  $cur_B < B.length$ 
8           $b_j = B[cur_B]$ 
9      if  $a_i < b_j$ 
10          $C.append \ a_i$ 
11          $cur_A = cur_A + 1$ 
12     else
13          $C.append \ b_j$ 
14          $cur_B = cur_B + 1$ 
15 if  $cur_B = B.length$ 
16     for  $i = cur_A$  to  $A.length$ 
17          $C.append \ A[i]$ 
18 else
19     for  $j = cur_B$  to  $B.length$ 
20          $C.append \ B[j]$ 
21 return  $C$ 
```

Korrektheit von MERGE(A, B):

Schleifeninvariante:

Zu Beginn jeder Iteration sind die Listen A, B und C in sich sortiert.

Initialisierung:

Vor der ersten Iteration der **while**-Schleife sind die Listen A und B geordnet, weil sie geordnet „angeliefert“ werden. Die Liste C ist geordnet, weil sie zu diesem Zeitpunkt noch leer ist.

Fortsetzung:

Bei jeder Iteration wird jeweils das kleinste Element aus beiden Listen der Liste C hinzugefügt. Die Listen A und B verlieren kein Element, also sind sie immer noch sortiert. Die Liste C bekommt immer ein neues Element, welches größer oder gleich dem letzten ist. Somit bleibt C auch bei jeder Iteration sortiert.

Terminierung:

Die **while**-Schleife terminiert stets, da bei jeder Iteration mindestens einer der beiden Zeiger um eine Stelle in Richtung Listenende verrückt wird. Damit kommt ein Zeiger - sofern die Listen endlich sind - irgendwann am Ende an.

Neue Initialisierung:

Die **for**-Schleife in Zeile 16 oder 19 verletzt auch nicht die Invariante. Zu Beginn sind alle drei Listen sortiert, aber B - oder A - wurde schon vollständig an C angefügt. Somit sind alle Elemente der verbleibenden Liste A - oder B - größer oder gleich dem letzten - und größten - Element in C.

Neue Fortsetzung:

Da alle Elemente der verbleibenden Liste A - oder B - größer oder gleich jedem Element in C sind, können wir einfach das nächste - und damit kleinste - verbleibende Element aus A - oder B - an C anfügen. Bei

diesem Schritt bleibt A - oder B - sortiert, weil sie nicht verändert wird, und C auch, da sie nur ein weiteres größeres Element angesetzt bekommt.

Neue Terminierung:

Die **for**-Schleife terminiert auch, da der Zähler bei jeder Iteration um 1 erhöht wird. Solange die Liste A - oder B - also nicht unendlich ist, erreicht der Zähler irgendwann ihr Ende.

Wenn die Schleifen abgebrochen sind, gilt, dass die Listen A, B und C sortiert sind. Die Ausgabeliste C also auch. Dies wollten wir zeigen, damit ist dieser Algorithmus korrekt.

Korrektheit von MERGESORT(A, L, R):

Induktion:

Induktion über die Größe der Eingabeliste:

Induktionsanfang:

Hat die Liste die Länge 1, so ist $l = r$. Die Ursprungsliste wird nicht verändert, womit MERGESORT korrekt ist. Denn einelementige Listen sind immer sortiert.

Induktionsannahme:

Ist MERGESORT für die Länge $n \in \mathbb{N}$ wahr, so ist es auch für die Länge $n + 1$ wahr.

Induktionsschritt:

Durch den rekursiven Aufruf, wird MERGESORT(n) für jedes $n > 1$ zu zwei Aufrufen von MERGESORT($\frac{n}{2}$). Somit wird es bei jedem Rekursionsschritt halb so groß und kommt damit irgendwann bei x Aufrufen von MERGESORT(1) an. Dieses gilt nach Induktionsanfang.

3.4.2 Aufgabe 3.4.2

Idee:

Wir nehmen MERGESORT und jedes mal, wenn wir 2 Elemente tauschen müssen, inkrementieren wir den Zähler um 1. Wenn Die Liste sortiert wurde, dann sollte der Zähler die Anzahl der Konflikte anzeigen.

Pseudocode:

```
MERGECOUNT(A, L, R)
1  Zähler = 0
2  if  $l < r$ 
3       $q = (l + r) / 2$ 
4      MERGECOUNT(A, L, Q)
5      MERGECOUNT(A, Q + 1, R)
6       $B = \text{newArray}(A[l] \text{ to } A[q])$ 
7       $C = \text{newArray}(A[q + 1] \text{ to } A[r])$ 
8      MERGECOUNT-COUNTER(B, C, ZAEHLER)
9  return Zähler
```

```

MERGECOUNT-COUNTER(A, B, ZAEHLER)
1  C = newArray
2  while A.length > 0 & B.length > 0
3      if A[0] > B[0]
4          C.append(B[0])
5          B = cdr(B)
6          Zaehler = Zaehler + A.length
7      else
8          C.append(A[0])
9          A = cdr(A)
10 while A.length > 0
11     C.append(A[0])
12     A = cdr(A)
13 while B.length > 0
14     C.append(B[0])
15     B = cdr(B)
16 return C, Zaehler

```

Korrektheit:

Da die Schleifeninvariante nicht unbedingt nötig ist (s. Aufgabenstellung) folgt hier die rein logische Begründung:

Wir teilen die Liste in der Mitte in 2 gleiche Teile, ohne die Reihenfolge der Elemente zu verändern. Dies tun wir so lange, bis nur noch einelementige Listen da sind. Einelementige Listen sind immer sortiert. Diese Listen werden nun wie folgt zusammengefügt:

Sollte das kleinste - also erste - Element in der rechten Liste (B) kleiner sein, als das kleinste - also erste - Element in der linken Liste (A), dann steht das Element aus B mit allen Elementen aus A in Konflikt. Wir erhöhen also den Konfliktzähler um die Anzahl der Elemente in Liste A.

Wenn allerdings das erste Element von A kleiner ist, als das erste Element von B, existiert kein Konflikt und der Zähler bleibt unangetastet.

Ist eine Liste leer, so sind die restlichen Elemente der anderen Liste nach Vorgehensweise des Verfahrens allesamt größer als das letzte Element der, nun leeren, Liste und sortiert, weswegen sie keine Konflikte mehr vorweisen und direkt an das Ergebnis angehängt werden können. Somit bleibt auch hier der Zähler unangetastet.

Oben beschriebenes passiert bei jeder Rekursion aufs Neue, weshalb der Zähler kontinuierlich weiterwächst - sofern Konflikte vorhanden.

Der Algorithmus terminiert zudem immer, da MERGESORT ebenfalls immer terminiert und sich MERGECOUNT nur im Erhöhen des Zählers von MERGESORT unterscheidet.

Zeitaufwand:

MERGECOUNT und MERGECOUNT-COUNTER verhalten sich genau, wie MERGESORT und MERGE, bis auf eine Erhöhung eines Zählers, was allerdings mit konstantem Zeitaufwand zu bewältigen ist und daher bei der O-Notation nicht ins Gewicht fällt. Somit liegt MERGECOUNT - genauso wie MERGESORT - ebenfalls in $\mathcal{O}(n \cdot \log(n))$.

4 Zettel vom 09. 10. – Abgabe: 23. 11.

4.1 Übungsaufgabe 4.1

[| 3]

4.1.1 Aufgabe 4.1.1

Eingabe: Eine endliche Menge U und eine Größe $s(u) \in \mathbb{N}$, eine Kapazität $K \in \mathbb{N}$ und eine Zahl $n \in \mathbb{N}$.

Frage: Gibt es eine Partitionierung von U in paarweise disjunkten Teilmengen U_1, U_2, \dots, U_n (d.h. die U_i enthalten zusammen alle Elemente aus U und jedes Element aus U ist in genau einem der U_i) derart, dass $\sum_{u \in U_i} s(u) \leq K$ für jedes i gilt?

Es kann eine Variante des Binary Sort Algorithmus angewandt werden:

```
PARTITION(SET S, INT C)
1  if sum(s) < c
2      return true
3  elseif max(s) > c
4      return false
5  else
6      Partition(s[0; (s.size/2)], c)
7      Partition(s[(s.size/2); s.size], c)
```

Ist die gesamte Größe von s bereits kleiner gleich K , kann der Algorithmus sofort terminieren und **true** zurückgeben. Andernfalls wird U halbiert und für die Teilmengen wird erneut geprüft. Dies wird rekursiv fortgeführt, bis eine Ebene erreicht wurde, auf der die Größe aller Teilmengen kleiner gleich K ist.

Da bereits zu Beginn geprüft wird, ob ein einzelnes Element größer als die Kapazität ist, ist gewährleistet, dass bei rekursiven Aufruf der Algorithmus spätestens dann terminiert, wenn jede Teilmenge genau die Mächtigkeit 1 hat. Ist ein einzelnes Element von U bereits größer als K , so ist das Problem nicht lösbar und der Algorithmus terminiert, **false** zurückgebend.

Durch die jeweilige Halbierung wird die Funktion höchstens $\log n$ mal rekursiv aufgerufen. $\text{sum}(s)$ und $\text{max}(s)$ haben jeweils eine lineare Laufzeit aus $\mathcal{O}(n)$. Damit werden insgesamt $a \cdot b \cdot n \cdot \log n + f(n) + g(n)$ ¹ Schritte zur vollständigen Berechnung benötigt, die obere Schranke liegt also in $\mathcal{O}(n \cdot \log n)$. Dies ist nicht polynomiell und liegt somit in NP .

4.1.2 Aufgabe 4.1.2

Eingabe: Ein endliches Alphabet Σ , eine endliche Menge $S \subset \Sigma^*$ von Worten und eine Zahl $n \in \mathbb{N}$.

Frage: Gibt es ein Wort $w \in \Sigma^*$ mit $|w| \geq n$ derart, dass w ein Teilwort von jedem $x \in S$ ist?

Es wird das kürzeste Wort $x \in S$ gewählt (Laufzeit $\mathcal{O}(S.length)$). Nun wird ein Teilwort mit der Länge n aus x gewählt und geprüft, ob es in allen anderen Wörtern aus S auch existiert (Laufzeit $\mathcal{O}(S.length)$). Ist dies der Fall, wird **true** ausgegeben, sonst wird das nächste Teilwort aus x mit der Länge n gewählt (Laufzeit $\mathcal{O}(k)$; $k \in \mathbb{N}$). Wurde erfolglos über x iteriert, bricht der Algorithmus ab und gibt **false** zurück. Die Laufzeit ist konstant in $\mathcal{O}(a)$; $a \in \mathbb{N}$.

Die Gesamtlaufzeit liegt damit bei $\mathcal{O}(a + k + (2 \cdot S.length))$.

4.2 Übungsaufgabe 4.2

[| 7]

¹ $a, b, f(n), g(n)$ sind Faktoren bzw. Konstanten, die in $\text{sum}(s)$ und $\text{max}(s)$ entstehen können, allerdings vernachlässigbar sind

4.2.1 Aufgabe 4.2.1

Zegen Sie, dass P und NP jeweils gegenüber Konkatenation abgeschlossen sind (d.h. mit $L_1, L_2 \in P$ ist auch $L_1 \cdot L_2 \in P$ und entsprechend für NP).

Für P : Seien $L_1, L_2 \in P, w_1 \in L_1, w_2 \in L_2, w_3 = w_1 \cdot w_2$. Sei außerdem $w_3 = b_1 b_2 b_3 b_4 \dots b_n; w_3 \in L_3; L_3 = L_1 \cdot L_2$.

Wähle $w_1 = b_1 b_2 b_3 \dots b_i$ und $w_2 = b_{i+1} b_{i+2} \dots b_n$.

Solange $w_1 \notin L_1 \wedge w_2 \notin L_2$ gilt, erhöhe i um 1 (als Startwert für i gilt 0.) Dies teilt das Wort in zwei Teilworte und wir wissen nun, wo wir w_3 teilen müssen, um w_1 und w_2 zu erhalten. Bei $i = 0$ gilt $w_1 = \lambda$, bei $i = n$ gilt $w_2 = \lambda$. Nun wird w_1 wie vorher in L_1 in Polynomialzeit gelöst. Ebenso w_2 in L_2 .

Die Zerlegung benötigt im schlimmsten Fall $n + 1$ Ausführungen, da es $n + 1$ mögliche Zerteilungen von w_3 gibt. Somit beträgt die Gesamtlaufzeit die Zeit, die w_1 in L_1 benötigt (ein Polynom) plus die Zeit, die w_2 in L_2 benötigt (auch ein Polynom) mal die Zeit, die das Zerschneiden benötigt ($n + 1$), also: $(Polynom(L_1) + Polynom(L_2)) * (n + 1)$. Dies ist allerdings wieder ein Polynom.

Für NP : Seien $L_1, L_w \in NP, w_1 \in L_1, w_2 \in L_2$ und $w_3 = w_1 \cdot w_2$ und $w_3 = b_1 b_2 b_3 b_4 \dots b_n$ und $w_3 \in L_3$ und $L_3 = L_1 \cdot L_2$.

Wähle $w_1 = b_1 b_2 b_3 \dots b_i$ und $w_2 = b_{i+1} b_{i+2} \dots b_n$.

Solange $w_1 \notin L_1$ und $w_2 \notin L_2$ gilt, erhöhe i um 1 (als Startwert für i gilt 0.)

Dies teilt das Wort in zwei Teilworte und wir wissen nun, wo wir w_3 teilen müssen, um w_1 und w_2 zu erhalten. Bei $i = 0$ gilt $w_1 = \lambda$, bei $i = n$ gilt $w_2 = \lambda$. Nun wird w_1 wie vorher in L_1 in Nichtpolynomialzeit gelöst. Ebenso w_2 in L_2 .

Die Zerlegung benötigt im schlimmsten Fall $n + 1$ Ausführungen, da es $n + 1$ mögliche Zerteilungen von w_3 gibt. Somit beträgt die Gesamtlaufzeit die Zeit, die w_1 in L_1 benötigt (ein Nichtpolynom) + die Zeit, die w_2 in L_2 benötigt (auch ein Nichtpolynom) * die Zeit, die das Zerschneiden benötigt ($n + 1$), also: $(Nichtpolynom(L_1) + Nichtpolynom(L_2)) * (n + 1)$. Dies ist allerdings wieder ein Nichtpolynom.

4.2.2 Aufgabe 4.2.2

Geben Sie unter der Annahme $P = NP$ einen Algorithmus an, der in polynomieller Zeit zu einer aussagenlogischen Formel eine erfüllende Belegung bestimmt, sofern eine existiert. Begründen Sie kurz, warum Ihr Algorithmus das Gewünschte leistet.

Brute Force:

Wir setzen für jedes Literal 1 ein und testen.

Erfüllt die Belegung nicht, ersetzen wir eine 1 durch eine 0 und prüfen dann alle Belegungen, die aus $(n - 1)$ 1 und einer 0 bestehen. Wenn die Belegung wieder nicht erfüllt, wird eine weitere 1 durch eine 0 ersetzt, solange, bis die Belegung aus 0^n besteht oder eine erfüllende Belegung gefunden wurde. Handelt es sich bei 0^n auch nicht um eine erfüllende Belegung, so existiert keine.

Dieser Algorithmus benötigt im schlimmsten Fall $\sum_{k=0}^n \binom{n}{k} = 2^n$ viele Schritte. Dies liegt in NP und damit unter der Annahme $P = NP$ auch in P . Damit ist es in polynomieller Zeit lösbar.

4.2.3 Aufgabe 4.2.3

Zeigen Sie, dass unter der Annahme $P = NP$ jedes $J \in P$ außer $L = \emptyset$ und $L = \Sigma^*$ NP -vollständig ist. Warum gilt diese Aussage nicht für die beiden ausgeschlossenen Fälle?

Sei $P = NP$. Damit sind alle Probleme aus NP auch in Polynomialzeit lösbar. Zudem ist jedes Problem aus P in Polynomialzeit auf jedes andere Problem aus P reduzierbar (, da es sich bei allen Problemen in P um Polynome handelt,) was genau der Definition von NP entspricht. Allerdings sind in P nur endliche Teilmengen von Σ^+ , denn \emptyset ist nicht entscheidbar, ebenso wie unendliche Sprachen (Σ^*). Somit sind sie auch nicht in P . Da jedoch $P = NP$ gilt, liegt der Rest aus P ebenfalls in NPC . Somit ist, wenn $P = NP$ gilt, jedes L aus P auch in NP , außer der Leeren Menge und dem unendlichen Σ^* .

4.3 Übungsaufgabe 4.3

[| 6]

4.3.1 Aufgabe 4.3.1

Sei 2-SAT die Menge aller (sinnvoll codierten) aussagenlogischen Formeln, die mindestens zwei erfüllende Belegungen haben. Zeigen Sie, dass 2-SAT NP -vollständig ist.

Ein Algorithmus für 2-SAT:

Als Algorithmus für 2-SAT kann eine modifizierte Version des Brute Force Algorithmus' aus 4.2.2 dienen:

Es wird jede mögliche Belegung für eine aussagenlogische Formel durchprobiert und geschaut, ob sie erfüllend ist. Wenn sie erfüllend ist, wird ein Zähler um 1 inkrementiert. Sollte dieser Zähler die Zahl 2 erreichen, wird "true" ausgegeben, sonst wird die nächste Belegung probiert. Hat der Zähler, nachdem alle Belegungen durchprobiert wurden, noch nicht die 2 erreicht, wird "false" ausgegeben.

Dieser Algorithmus läuft im schlimmsten Fall in $\sum_{k=0}^n \binom{n}{k} = 2^n$ vielen Schritten.

Dies liegt in NP und zeigt daher $2\text{-SAT} \in NP$.

Wir wissen aus der Vorlesung: $\text{SAT} \in NPC$

Reduktion von SAT auf 2-SAT:

Angenommen Automat A löst 2-SAT, dann hat A einen Unterautomaten B, der ausgibt, ob eine beliebige aussagenlogische Formel erfüllbar ist. A gibt nun "true" aus, wenn B für eine Formel mindestens 2 mal "true" ausgegeben hat.

Dieser Automat B wäre allerdings genau die Lösung zu unserem SAT-Problem, welches laut der Vorlesung ja sogar NPC ist. Daher kann dieser Automat B nicht existieren, woraus folgt, dass A auch nicht existieren kann.

Dies zeigt $2\text{-SAT} \in NPH$.

Somit muss 2-SAT in der Schnittmenge von NP und NPH sein, also gilt $2\text{-SAT} \in NPC$.

4.3.2 Aufgabe 4.3.2

Die folgende Beschreibung verallgemeinert das Spiel *Minesweeper* auf einen ungerichteten Graphen: Sei G ein ungerichteter Graph. Jeder Knoten von G enthält entweder eine einzelne *Mine* oder ist leer. Der Spieler kann einen Knoten wählen. Ist es eine Mine, hat er verloren. Ist der Knoten leer, dann wird dieser mit der Anzahl der direkt benachbarten Knoten beschriftet, die eine Mine enthalten. Der Spieler gewinnt, wenn alle leeren Knoten gewählt wurden. Das Problem ist nun folgendes: Gegeben ein Graph zuzüglich einiger beschrifteter Knoten, ist es möglich Minen so auf die verbleibenden Knoten abzulegen, dass jeder Knoten v , der mit k beschriftet ist, genau k direkt benachbarte Knoten besitzt, die eine Mine enthalten? Formulieren Sie dieses Problem sinnvoll als formale Sprache (und füllen Sie dadurch die Lücken in obiger Beschreibung) und zeigen

Sie dann, dass dieses Problem *NP*-vollständig ist.

Eingabe: Ein ungerichteter Graph $G = (V, E)$ und eine Funktion f , die jedem Knoten v aus V einen Wert $k \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ zuweist. Dieser Wert kann sich von Knoten zu Knoten unterscheiden.

Frage: Gibt es eine Verteilung von Minen U , sodass jeder Knoten v aus V genau k Nachbarknoten hat, die eine Mine besitzen?

Algorithmus: Wir gehen jeden Knoten durch und belegen - sofern die Zahl auf dem Knoten noch nicht der Anzahl an Nachbarknoten mit einer Mine entspricht - einen beliebigen Nachbarknoten mit einer Mine, bis die Zahl stimmt. Sobald wir an einem Knoten ankommen, der bereits mehr Nachbarknoten mit Minen besitzt, als seine Zahl groß ist, gehen wir einen Knoten zurück und merken uns die bisher gewählte Konfiguration als *falsch* und wählen eine andere. Sollten alle möglichen Konfigurationen - an der Anzahl $\binom{8}{k}$, wobei k der Wert des Knotens ist - als *falsch* markiert worden sein, wird ein Knoten zurück gegangen, die dortige aktuelle Konfiguration als *falsch* markiert und mit dieser neuen Kombination fortgefahren. Wobei jetzt sämtliche Konfigurationen der Folgeknoten wieder möglich sind.

Wird auf diese Weise bis zum ersten Knoten zurückgekehrt und werden dort sämtliche Konfigurationen als *falsch* markiert, gibt der Algorithmus "**false**" aus. Sollte allerdings jede Mine verteilt sein, ohne dass es zu einem Konflikt mit der Zahl auf einem der Knoten kommt, wird "**true**" ausgegeben.

Dies hat eine maximale Laufzeit von $\prod_{i=1}^n \binom{8}{k_i} \approx \binom{8}{k}^n \leq \binom{8}{4}^n = 70^n$, wenn für jeden Knoten alle Kombinationen durchlaufen werden müssen, wobei $\binom{8}{k}$ konstant und höchstens $70 = \binom{8}{4}$ ist.

Dies liegt in *NP*.

Reduktion: Wir können es nicht sinnvoll auf ein anderes Problem überführen, damit ist es offensichtlich schwerer als alle Probleme in *NP* und somit in *NPH*. Damit ist es auch in *NPC*.

Zettel vom 26. 10. – Abgabe: 07. 12.

Übungsaufgabe 5.1

[| 4]

Aufgabe 5.1.1

Führen Sie die in der Vorlesung behandelte Einfügeoperation **TreeInsert** für binäre Suchbäume nacheinander mit den Elementen 5, 8, 9, 6, 7 aus. Zu Anfang sei der Baum leer. Geben Sie den nach jeder Einfügeoperation erhaltenen Suchbaum an. (1 Pkt.)

Im Folgenden wurden die **nil**-Blätter der Übersicht halber nicht mitgeführt.

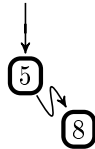
Füge 5 ein:

Nil



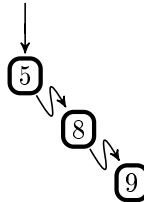
Füge 8 ein:

Nil



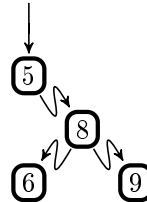
Füge 9 ein:

Nil



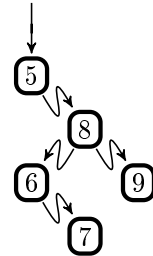
Füge 6 ein:

Nil



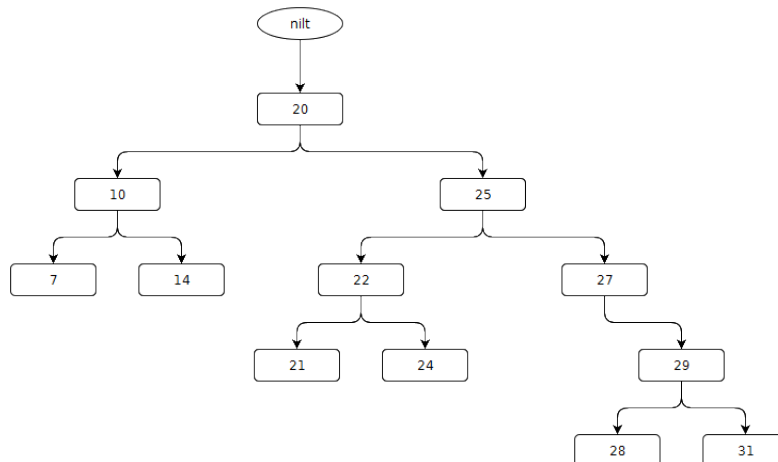
Füge 7 ein:

Nil



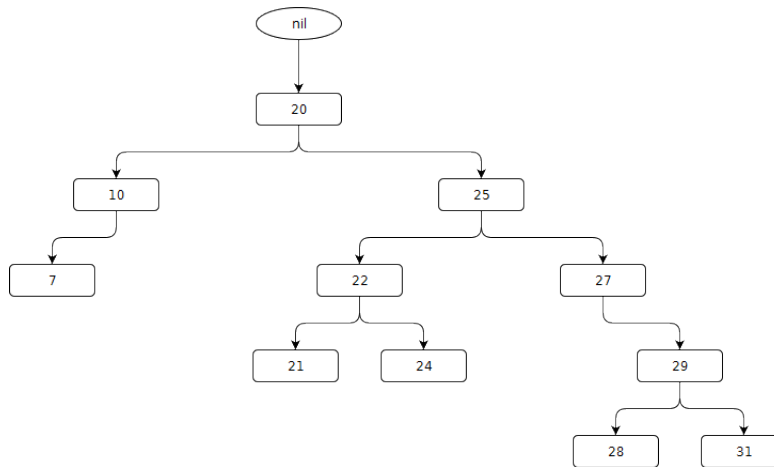
Aufgabe 5.1.2

Gegeben sei folgender binärer Suchbaum:

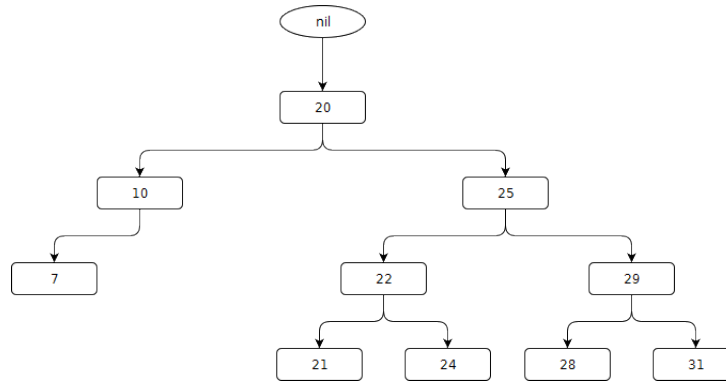


Führen Sie nacheinander die in der Vorlesung behandelte Löschoption für binäre Suchbäume mit den Elementen 14, 27, 25 aus. Geben Sie den nach jeder Löschoption erhaltenen Suchbaum an. (3 Pkt.)

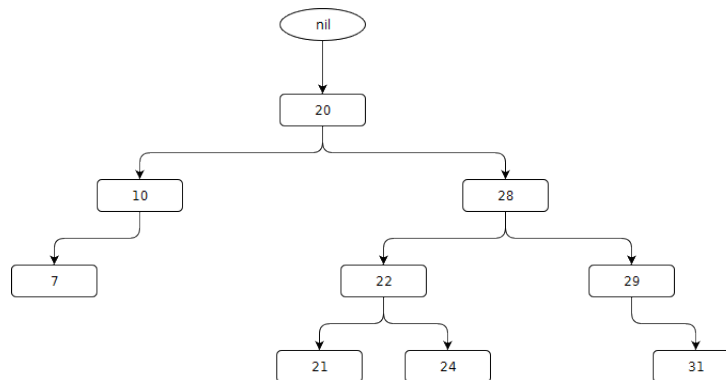
Nach Entfernen der 14:



Nach Entfernen der 27:



Nach Entfernen der 25:



Übungsaufgabe 5.2

[| 4]

Geben Sie für die Routine **InorderTreeWalk** einen nichtrekursiven Algorithmus (3 Pkt.) und für die Routine **TreeMinimum** einen rekursiven Algorithmus (1 Pkt.) an. Erläutern Sie in beiden Fällen unbedingt die Arbeitsweise ihres Algorithmus.

(Hinweis: Beide Routinen wurden in der Vorlesung behandelt. InorderTreeWalk allerdings in rekursiver, TreeMinimum in nichtrekursiver Darstellung. Für die erste Teilaufgabe hilft ein Vergleich mit der Breiten-

und Tiefensuche bei Graphen. Für die zweite Teilaufgabe betrachte man die Routine *TreeSearch* in ihren beiden Varianten.)

```
inorderTreeWalk_nichtRekursiv(x)
1  AusgabeArray = null
2  TreeMinimum(x)           ; s. Vorlesung 7 Folie 37
3  AusgabeArray.add(x)
4  while TreeSucessor(x) ≠ nil ; s. Vorlesung 7 Folie 47
5      x = TreeSucessor(x)
6      AusgabeArray.add(x)
7      return AusgabeArray
```

Dieser Algorithmus sucht sich das Minimum eines Baumes und packt es in ein Array. Danach wird sein Nachfolger gesucht und in das Array gepackt. Sollte ein Knoten irgendwann keinen Nachfolger mehr haben, terminiert der Algorithmus und wir geben das Array aus.

```
TreeMinimum_rekursiv(x)
1  if links[x] ≠ nil :
2      TreeMinimum_rekursiv(links[x])
3  else return x
```

Dieser Algorithmus prüft, ob der aktuelle Knoten ein linkes Kind hat. Ist dies der Fall, wird die Methode rekursiv am linken Kind aufgerufen (das linke Kind wird von seinem Vater gefragt, was denn sein Minimum sei). Sollte dies allerdings nicht so sein und der aktuelle Knoten besitzt kein linkes Kind, dann ist er selbst sein eigenes Minimum und damit auch das Minimum aller Knoten über ihm und wird ausgegeben.

Übungsaufgabe 5.3

[| 4]

Gegeben sei ein ungerichteter Graph $G = (V, E)$ auf n Knoten. Ferner seien $s, t \in V$ Knoten, deren Distanz (ihr kürzester Abstand) echt größer als $\frac{n}{2}$ sei, d.h. $d(s, t) > n/2$. Zeigen Sie zunächst, dass dann ein von s und t verschiedener Knoten v existieren muss, dessen Löschung alle s - t -Pfade zerstört. (Der Graph G' , der aus G entsteht, wenn man v löscht, enthält also keine Pfade von s zu t mehr.)

Geben Sie dann einen Algorithmus an, der den Knoten v findet. Können Sie einen Algorithmus angeben, der in Zeit $O(n + m)$ arbeitet? Erläutern Sie die Arbeitsweise Ihres Algorithmus und begründen Sie seine Korrektheit sowie seine Laufzeit.

(Anmerkung: Die obige Fragestellung tritt bei der Untersuchung von Netzwerken auf. Sie besagt, dass, wenn zwei Knoten zu weit auseinander sind (nur durch viele Zwischenknoten verbunden sind), diese eine anfälligere Verbindung haben als Knoten die dichter beieinander sind. Obiges Resultat zeigt nämlich gerade, dass der Ausfall eines Knotens v genügt, um s und t zu trennen. Allerdings ist die Aussage nicht, dass irgendein Knoten v ausfallen kann und dann sofort s und t getrennt wären, sondern nur, dass (mindestens) ein solcher Knoten existiert.)

Sei $G = (V, E)$ ein Graph. Dieser Graph besitzt die Knoten s, t und v und die Bedingung, dass $d(s, t) > |V|/2$ gilt. Damit enthält der kürzeste Pfad von s nach t mindestens die Hälfte aller Knoten (Genauer $|V|/2 + 2$ Knoten, wobei $|V|/2$ abgerundet wird und die $+ 2$ durch s und t zustandekommen). Um nun einen weiteren Pfad dieser Länge zu kreieren, muss ein Teil des bisherigen Pfades wiederverwertet werden, da sonst nicht

mehr genügend freie Knoten zur Verfügung stünden. Und für jeden weiteren Pfad werden jeweils Teile der alten Pfade wieder verwertet. Irgendwann gehen die nicht benutzten Knoten aus und auch die Möglichkeiten übrige Knoten zu verbinden sind ausgeschöpft dann können weitere Pfade nicht generiert werden. Alle Knoten, die nun in der Schnittmenge aller Pfade liegen, sind potentielle v Kandidaten, da jeder mögliche Pfad sie enthält - sie sozusagen die Schnittstellen aller möglichen kürzesten Pfade sind und so in jedem Pfad vorkommen müssen, damit er ein Pfad von t nach s ist.

Wir setzen an jeden Knoten eine hochlaufende Zahl an, so dass jeder Knoten eine eigene individuelle Zahl erhält ($O(n)$).

Wir führen eine Breitensuche aus, um den kleinsten Pfad zu bestimmen ($O(n + m)$).

Wir führen eine Tiefensuche aus, um irgendeinen anderen Pfad zu bestimmen ($O(n + m)$).

Da ein Pfad von s nach t existiert, werden beide Varianten dieser Suchen einen Pfad finden.

Wir drehen die Reihenfolge der Adjazenzlisten um (Für jeden Knoten n müssen alle seine Nachbarn bearbeitet werden. Da eine Kante durch 2 "Nachbargaarein der Adjazenzliste dargestellt wird, ergibt sich eine Laufzeit von $O(n + 2 * m) = O(n + m)$).

Wir führen eine zweite Tiefensuche durch, um einen Pfad maximal verschieden zum Zweiten zu erhalten ($O(n + m)$).

Nun werden die 3 Listen der Suchen jeweils nach der hochlaufenden Zahl sortiert ($O(\text{Stelligkeit} * (\text{Listenlänge} + \text{Anzahlverschiedener Stellen}))$).¹

Die Stelligkeit dieser Zahlen ist maximal die Stelligkeit von n. Das Herausfinden dieser Stelligkeit geht mit konstantem Zeitaufwand.

Jede Stelle hat von Natur aus 10 mögliche Werte (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Die Listenlänge ist im worst-case für jede Liste n.

⇒ Laufzeit: $3 * c * (n + 10)$. Das liegt in $O(n)$

Nun wird durch die 3 Listen gelaufen und jeweils verglichen, ob die Elemente in jeder dieser Liste drin sind. Da es sich bei diesen Elementen um sortierte Zahlen handelt, die sich innerhalb einer Liste nicht wiederholen, muss sich der Algorithmus jedes Element jeder Liste im worst-case genau einmal anschauen (Laufzeit: $3 * n$). Dies liegt in $O(n)$.

Das erste gefundene Element/jedes gefundene Element (im Fall "jedes Element" muss der obige Vergleichsalgorithmus sich immer jedes Element jeder Liste anschauen, verändert aber nicht seine Laufzeit) wird in einen Knoten umgewandelt und ausgegeben (Laufzeit: $c + c$. Dies liegt in $O(c)$).

Gesamtlaufzeit: $O(n) + O(n + m) + O(n + m) + O(n + m) + O(n + m) + O(n) + O(n) + O(c) = O(n + m)$

Idee vollständig?: Ja, da durch die 2 Tiefensuchen mit Umkehrung der Adjazenzliste 2 maximal voneinander unabhängige Pfade gefunden werden. Die Schnittmenge von diesen wird in beiden Pfaden benötigt, um von s nach t zu kommen. Durch die Breitensuche wird dann auch noch garantiert, dass es sich um einen kürzesten Pfad handelt und somit um die minimale Anzahl der Knoten. Dies führt dazu, dass die Schnittmenge von diesem Pfad und den obigen beiden minimal ist. Und in dieser minimalen Schnittmenge ist nun jeder Knoten ein v-Kandidat.

Pseudocode:

```
ein_v_knoten(Graph G, Knoten s, Knoten t)
```

```
    //gibt einen möglichen v-Knoten aus
```

```
1  List Vs = v_knoten(G, s, t)
```

```
2  index = random(Vs.length)
```

```
3  return Vs[index]
```

¹Quelle: s. Vorlesung 5 Folie 46 http://www.informatik.uni-hamburg.de/TGI/lehre/v1/WS1516/AuD/Folien/aud_v5.pdf

```

v_knoten(Graph G, Knoten s, Knoten t)
    //gibt eine Liste mit allen v-Knoten aus
1  inti = 0
2  for KnotenninG
3      n.zahlenwert = i
4      i = i + 1
5  List A = sucheKleinstenPfad(G, s, t)
6  List B = sucheAnderenPfad(G, s, t)
7  GraphG' = G
8  G'.Adjazenzliste = reverse(G.Adjazenzliste)
9  List C = sucheAnderenPfad(G', s, t)
10 List A' = RadixSort(A nach Knoten.zahlenwert)
    // s. Vorlesung 5 Folie 46; Link: http://www.informatik.uni-hamburg.de/TGI/lehre/v1/WS1516/AuD/Folien/aud\_v5.pdf
11 List B' = RadixSort(B nach Knoten.zahlenwert)
12 List C' = RadixSort(C nach Knoten.zahlenwert)
13 List ErgebnisListe = new List
14 aktuellerKnoten = new Knoten
15 indexA = 0
16 indexB = 0
17 indexC = 0
18 while A'.length > indexA and B'.length > indexB and C'.length > indexC :
    //echt größer, da der Index ab 0 beginnt
19     aktuellerKnoten = A'[indexA]
20     if aktuellerKnoten < B'[indexB]
21         indexA = indexA + 1
22     else if aktuellerKnoten > B'[indexB]
23         indexB = indexB + 1
24     else
        //falls aktuellerKnoten == B'[indexB] ist
25         if aktuellerKnoten < C'[indexC]
26             indexA = indexA + 1
27         else if aktuellerKnoten > C'[indexC]
28             indexC = indexC + 1
29         else
            //falls aktuellerKnoten == C'[indexC] ist
30             ErgebnisListe.add(aktuellerKnoten)
31             indexA = indexA + 1
32 return ErgebnisListe

sucheAnderenPfad(Graph G, Knoten s, Knoten t)
    //gibt einen Pfad von s nach t aus aus
1  Tiefensuche(G, s)
2  return bauePfad(G, s, t)

```

```

bauePfad(Graph G, Knoten s, Knoten t)
    //baut gegeben einen durchsuchten Graphen einen Pfad von s nach t
    //(s und t sind nicht in dem Pfad enthalten um später nicht in der Schnittmenge zu landen)
1  Knoten n = t.parent
2  List A = newList
3  while !(n == s)
4  A.add(n)
5  n = n.parent
6
7  Return A

```

```

sucheKleinstenPfad(Graph G, Knoten s, Knoten t)
    // gibt einen kürzesten Pfad von s nach t aus
1  Breitensuche(G, s)
2  return bauePfad(G, s, t)

```

```

Tiefensuche(Graph G, Knoten wurzel)
    // führt eine Tiefensuche auf dem Graphen durch
1  for Knoten n in G
2      n.entdeckt = false
3      n.parent = NIL
4  Stack S = new Stack
5  S.push(wurzel)
6  Knoten aktuellerKnoten = NULL
7  while !(S.isEmpty)
8      aktuellerKnoten = S.pop()
9      if aktuellerKnoten.entdeckt == false
10         aktuellerKnoten.entdeckt = true
11         for Knoten m in aktuellerKnoten.AdjazenListe
12             S.push(m)
13             m.parent = aktuellerKnoten

```

```

Breitensuche(Graph G, Knoten wurzel)
    // führt eine Breitensuche auf dem Graphen durch
1  for Knoten n in G
2      n.distanz = INFINITY
3      n.parent = NIL
4  Queue Q = new Queue
5  wurzel.distanz = 0
6  q.enqueue(wurzel)
7  Knoten aktuellerKnoten = NULL
8  while !(Q.isEmpty)
9      aktuellerKnoten = Q.dequeue()
10     for Knoten m in aktuellerKnoten.Adjazenliste
11         if m.distanz == INFINITY
12             m.distanz = aktuellerKnoten.distanz + 1
13             m.parent = aktuellerKnoten
14             q.enqueue(m)

```

Übungsaufgabe 5.4

[| 4]

Aufgabe 5.4.1

Im Problem **Big-Clique** ist ein ungerichteter Graph G gegeben. Die Frage ist, ob G eine Clique enthält, die aus mindestens $\frac{n}{2}$ Knoten besteht (wobei n die Anzahl der Knoten von G ist). Zeigen Sie, dass **Big-Clique** NP-vollständig ist. (2 Pkt.)

Big-Clique liegt in NP.

Verifikationsalgorithmus, der die Richtigkeit eines gegebenen Zertifikates in polynomialzeit überprüft:

Unser Zertifikat: Eine Menge an Knoten - unsere Clique - und der Graph

Wir schauen in der Adjazenzliste nach, ob zu jedem gegebenen Knoten, zu allen anderen gegebenen Knoten eine Kante existiert. Ist dies der Fall, ist das Zertifikat richtig, sonst nicht.

Big-Clique liegt in NPH.

Reduktion eines Algorithmusses aus NPH auf Big-Clique:

Wir nehmen $\text{Clique}(G, k)$: $G = (V, K)$ Graph, k = Größe der Clique

Wenn unser k größer oder gleich $|V|/2$ ist, geben wir das Problem einfach in Big-Clique.

Ist es allerdings kleiner, fügen wir für jeden Knoten, der uns auf $|V|/2$ fehlt, 2 Knoten an den Graphen an, die wir jeweils mit Kanten zu allen anderen Knoten versehen.

Diesen neuen Graphen fügen wir in den Automaten ein, der Big-Clique löst und geben die Lösung als unsere Lösung für unser Clique-Problem aus.

Da wir beides obere haben, folgt daraus: Big-Clique liegt in NPC

Aufgabe 5.4.2

Geben Sie unter der Annahme $P = NP$ einen deterministischen Algorithmus an, der in polynomialer Zeit zu einem ungerichteten Graphen G eine maximal große Clique bestimmt. (2 Pkt.)

Verifikationsalgorithmus, der die Richtigkeit eines gegebenen Zertifikates in polynomialzeit überprüft:

Unser Zertifikat: Eine Menge an Knoten - unsere größte mögliche Clique - und der Graph

Wir geben unseren Graph und die Zahl $k = |\text{Menge an Knoten}| + 1$ in Clique.

Wenn Clique wahr ausgibt, ist das Zertifikat falsch, da eine größere Clique existiert.

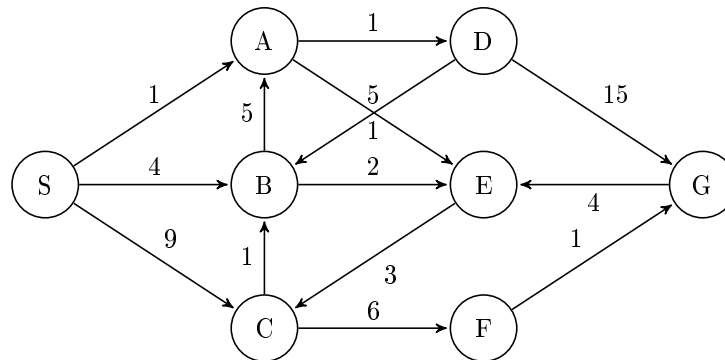
Dies läuft in polynomialzeit, da $P = NP$ angenommen wird.

Durch diesen Verifikationsalgorithmus ist unser Problem in NP.

Unter der Annahme $P = NP$ existiert allerdings auch ein P-Algorithmus und dieser ist deterministisch.

Übungsaufgabe 6.1

[| 10]



Betrachten Sie obigen Graphen G . Wenden Sie jeweils das verlangte Verfahren an bzw. beantworten Sie die Frage oder begründen Sie, warum dies nicht nicht. (Bei den ersten sechs Teilaufgaben spielen die Kantengewichte keine Rolle.)

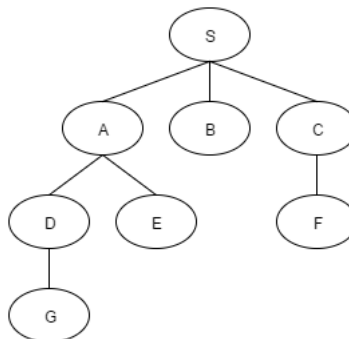
Aufgabe 6.1.1

Ermitteln Sie mit der Breitensuche einen Breitensuchbaum. Starten Sie den Algorithmus bei s . (1 Pkt.)

Ergebnis der Breitensuche:

S A B C D E F G

BFS-Baum:



Aufgabe 6.1.2

Ist das Ergebnis der Breitensuche eindeutig? (1 Pkt.)

Ja, ist es, da die Reihenfolge der Kanten, die gewählt werden, gegeben ist durch die Struktur der Adjazenzmatrix des Graphen. Zumindest ist dies der Fall bei den bisher im Modul gegebenen Algorithmen. Liegt keine Adjazenzmatrix vor oder arbeitet ein Algorithmus mit einer Modifikation der BFS, so kann es, gegeben es gibt mehrere gültige Breitensuchbäume, dazu kommen, dass unterschiedliche Ergebnisse zurückgegeben werden.

Aufgabe 6.1.3

Ermitteln Sie mit der Tiefensuche einen Tiefensuchwald und insb. die Zeiten $d[v]$ und $f[v]$ für jeden Knoten v . Starten Sie den Algorithmus wieder bei s . (1 Pkt.)

Ergebnis	Stack	$d[v]$	$f[v]$	Ergebnis	Stack	$d[v]$	$f[v]$	Knoten	$d[v]$	$f[v]$
/	S	–	–	SADBEFCFG	SADBEFCF	–	G:8	S	0	15
S	SA	S:0	–	SADBEFCFG	SADBEC	–	F:9	A	1	14
SA	SAD	A:1	–	SADBEFCFG	SADBE	–	C:10	B	3	12
SAD	SADB	D:2	–	SADBEFCFG	SADB	–	E:11	C	5	10
SADB	SADBE	B:3	–	SADBEFCFG	SAD	–	B:12	D	2	13
SADBE	SADBEC	E:4	–	SADBEFCFG	SA	–	D:13	E	4	11
SADBEC	SADBEFCF	C:5	–	SADBEFCFG	S	–	A:14	F	6	9
SADBEFCF	SADBEFCFG	F:6	–	SADBEFCFG	/	–	S:15	G	7	8
SADBEFCFG	SADBEFCFG	G:7	–							

Aufgabe 6.1.4

Geben Sie eine topologische Sortierung des Graphen G an. (1 Pkt.)

Es ist nicht möglich, eine topologische Sortierung für G anzugeben, da G zyklisch ist und somit unabhängig von der gewählten Reihenfolge keine Möglichkeit besteht, rückwärtige Kanten zu vermeiden.

Aufgabe 6.1.5

Bestimmen Sie mit dem Algorithmus aus der Vorlesung die starken Zusammenhangskomponenten von G . Geben Sie dazu das Ergebnis von $DFS(G)$ und $DFS(G^T)$ sowie die starken Zusammenhangskomponenten an. (1 Pkt.)

$DFS(G)$: s. 6.1.3.

$DFS(G^T)$: nicht möglich, da keine topologische Sortierung vorliegt.

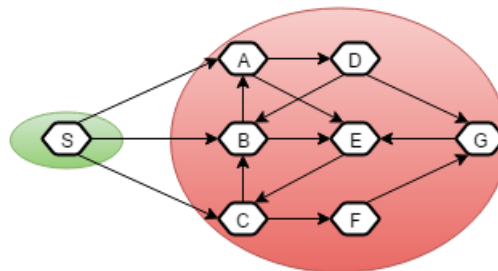
Starke Zusammenhangskomponenten:

[S], [A,B,C,D,E,F,G]

Aufgabe 6.1.6

Geben Sie den Komponentengraphen von G an. (1 Pkt.)

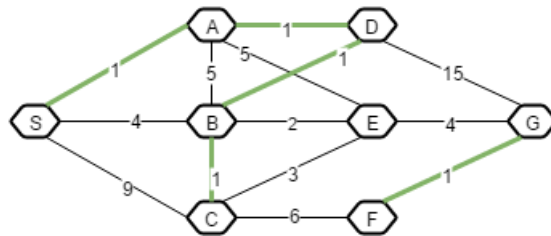
Der Übersicht halber wurden die Kantengewichte, da sie für diese Aufgabe ohnehin irrelevant sind, weggelassen.



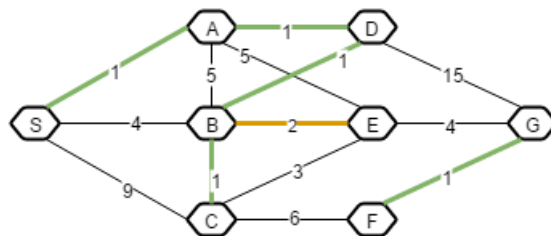
Aufgabe 6.1.7

Interpretieren Sie G nun als ungerichteten Graphen. Wenden Sie den Algorithmus von Kruskal an, um einen minimalen Spannbaum zu bestimmen. (Es genügt hier anzugeben welche Kanten in den minimalen Spannbaum aufgenommen werden und in welcher Reihenfolge dies geschieht!) (1 Pkt.)

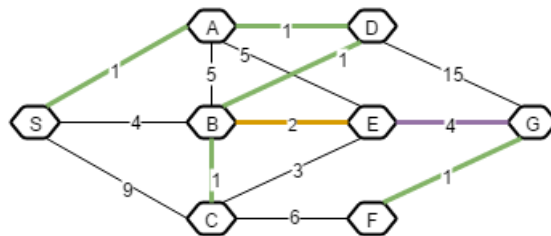
Schritt 1: Hinzufügen aller Kanten mit Gewicht 1.
 $\{(S, A), (A, D), (B, D), (B, C), (F, G)\}$



Schritt 2: Hinzufügen aller Kanten mit Gewicht 2.
 $\{(B, E)\}$



Schritt 3: Kanten mit Gewicht 3 sind redundant. Fahre fort mit Kanten mit Gewicht 4.
 $\{(E, G)\}$



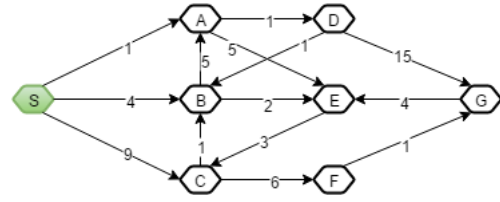
Aufgabe 6.1.8

Bestimmen Sie mit dem Algorithmus von Dijkstra kürzeste Pfade von s zu allen anderen Knoten. Geben Sie hierzu jeweils den Graph mit den Werten von $d[v]$ nach jeder Iteration der while-Schleife an und machen Sie $\pi[v]$ z.B. durch dickere Kanten deutlich. (2 Pkt.)

Sei $v \in V(G)$. Dann ordnet $d(v)$ jedem Knoten seine Distanz zum Startknoten zu. Sei desweiteren die Startdistanz jedes Knoten zum Startknoten zunächst ∞ . Außerdem werden fertig bearbeitete Knoten mit **roter** Schrift markiert, der als nächstes geprüfte Knoten in **grüner** Schrift.

1. Anfang

v	S	A	B	C	D	E	F	G
$d(v)$	0	∞	∞	∞	∞	∞	∞	∞

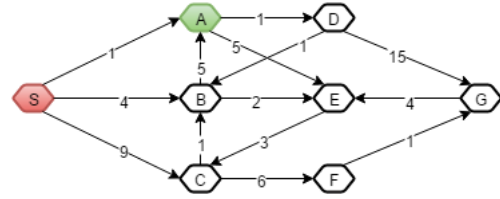


2. S

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	4	9	∞	∞	∞	∞

Kanten:

$(s, a), (s, b), (s, c)$

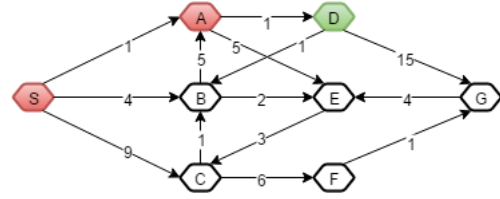


3. A

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	4	9	2	6	∞	∞

Kanten:

$(s, a), (s, b), (s, c), (a, d), (a, e)$

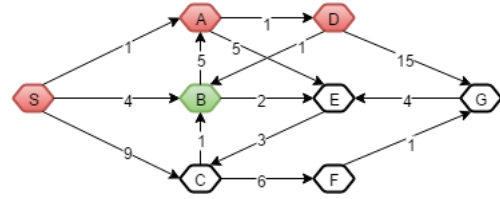


4. D

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	4	9	2	6	∞	17

Kanten:

$(s, a), (s, c), (a, d), (a, e), (d, b), (d, g)$

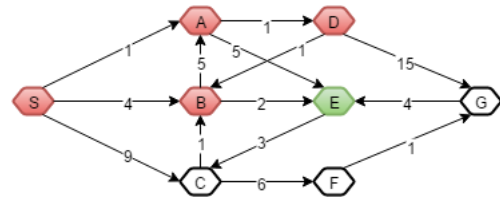


5. B

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	3	9	2	5	∞	17

Kanten:

$(s, a), (s, b), (s, c), (a, d), (a, e)$



6. E

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	3	8	2	5	∞	17

Kanten:

$(s, a), (a, d), (d, b), (d, g), (b, e), (e, c)$

7. C

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	3	8	2	5	14	17

Kanten:

$(s, a), (a, d), (d, b), (d, g), (b, e), (e, c), (c, f)$

8. F

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	3	8	2	5	14	15

Kanten:

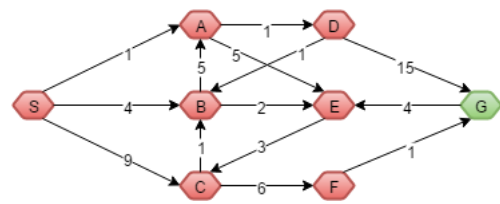
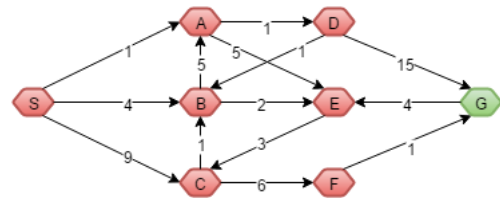
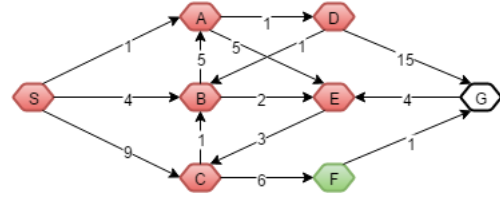
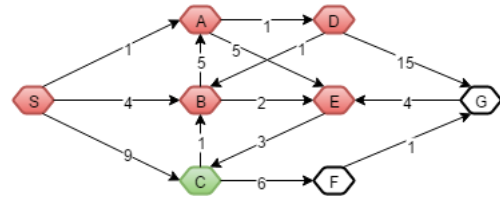
$(s, a), (a, d), (d, b), (b, e), (e, c), (c, f), (f, g)$

9. G

v	S	A	B	C	D	E	F	G
$d(v)$	0	1	3	8	2	5	14	15

Kanten:

$(s, a), (a, d), (d, b), (b, e), (e, c), (c, f), (f, g)$



Aufgabe 6.1.9

Betrachten Sie den Sourcecode von Dijkstras Algorithmus und vom Algorithmus von Prim. Was ist der hauptsächliche Unterschied der beiden Algorithmen bei der Ausführung? (1 Pkt.)

Während Dijkstra die Entfernung jedes Knotens zum Startknoten speichert, enthalten die Knoten nach Prim jeweils nur die Entfernung zum Vorgängerknoten. Dieser Unterschied wird durch Dijkstra's `Relax` in Zeile 8ff¹ hervorgerufen (Dieser `if`-Block ist äquivalent zur Funktion `Relax` von der Seite vorher).

¹Foliensatz 9, Version 03.01.2016, Seite 119

Übungsaufgabe 6.2

[| 6]

Aufgabe 6.2.1

Sei $G = (V, E)$ ein ungerichteter Graph. Bei einer k -Färbung von G wird jedem Knoten $v \in V$ eine Farbe $c \in [k] = \{1, 2, \dots, k\}$ zugewiesen. Die Färbung ist korrekt, wenn zwei benachbarte Knoten nie die gleiche Farbe haben. Formal ist eine k -Färbung dann also eine Abbildung $f : V \rightarrow [k]$ und eine *korrekte k -Färbung* erfüllt $f(x) \neq f(y)$ für jede Kante $\{x, y\} \in E$. Wir definieren folgendes Problem:

$$k\text{-col} = \{\langle G \rangle \mid G \text{ besitzt eine korrekte } k\text{-Färbung}\}$$

Angenommen Sie wissen, dass 3-col NP-vollständig ist. Zeigen Sie, dass auch 8-col NP-vollständig ist. (2 Pkt.)

Sei das Zertifikat eine Menge bestehend aus einem Graphen $G = (V, E)$ und einer Färbungsfunktion f . Die Prüfung des korrekten Ergebnisses der Funktion in Form von $f(x) \neq f(y)$ für zwei benachbarte Knoten $x, y \in V$ lässt sich in polynomieller Zeit erledigen, womit 8-col in NP liegt.

Reduktion

Es wird nach Aufgabenstellung 3-col auf 8-col reduziert.

Nach Aufgabenstellung ist bekannt, dass 3-col in NP liegt.

Sei die Gruppe H eine Menge von Knoten, die eine erfüllende Belegung von 3-col darstellt. Zu H wird ein Knoten hinzugefügt, der paarweise mit allen drei Knoten aus H verbunden wird. Da der neue Knoten mit allen anderen Knoten verbunden ist, muss er eine Farbe erhalten, die bisher nicht vergeben ist. H' ist nun eine erfüllende Belegung für 4-col und jede 3 Knoten aus H sind eine erfüllende Belegung für 3-col. Dies wird wiederholt, bis H'_n 8 Knoten vorweist. Für jede k Knoten gilt hierbei, dass es sich um eine erfüllende Belegung für k -col handelt, und jede Auswahl aus $k-1$ Knoten aus H eine erfüllende Belegung für $(k-1)$ -col ist. Diese Erweiterung liegt in $\mathcal{O}(k - |V_{H_0}|)$.

Aufgabe 6.2.2

Wir betrachten das folgende Spiel: Auf einem Feld mit n Spalten, n Zeilen und $n \times n$ Feldern können auf jeder der n^2 Positionen entweder ein weißer oder ein schwarzer Stein liegen oder das Feld kann leer sein. Eine Anfangsmarkierung besteht aus einer Zuweisung von Spielsteinen zu Feldern. Das Spiel wird dann (alleine) gespielt, indem beliebig viele Steine entfernt werden. Ziel ist es, dass jede Spalte nur noch Steine einer Farbe enthält (oder leer ist) und jede Zeile mindestens einen Stein enthält. Ob man dieses Ziel erreichen kann (und damit gewinnt), hängt von der Anfangsmarkierung ab. Die Eingabe ist ein Spiel S bestehend aus dem Feld und der Anfangsmarkierung (für das Feld reicht die Zahl n). Die Eingabe wird akzeptiert, wenn S gewonnen werden kann. Zeigen Sie, dass dieses Problem NP-vollständig ist. (4 Pkt.)

Reduktion von 3KNF auf Kreuzgo¹:

Eine Spalte wird als Klausel interpretiert, die dann wahr ist, wenn für jedes Feld gilt:

$$((\text{schwarz} \wedge \neg \text{weiss}) \vee (\neg \text{schwarz} \wedge \text{weiss}) \vee \text{null})$$

je nachdem, ob der erste gefundene Stein in der Spalte schwarz oder weiß ist.

Ebenso wird eine Zeile als Klausel interpretiert, die dann wahr ist, wenn für mindestens ein Feld gilt:

$$(\text{schwarz} \vee \text{weiss} \vee \neg \text{null})$$

¹Wir haben das Spiel in Ermangelung eines Namen so getauft

Diese Belegung muss für jede Zeile bzw. jede Spalte respektive gelten; die einzelnen Klauseln sind konjunkt. Gibt es eine erfüllende Belegung für die dadurch entstehende KNF, so ist diese Lösung auch eine Lösung für Kreuzgo.

Übungsaufgabe 7.1

Aufgabe 7.1.1

Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitätsfunktion $c, u, v \in V$ und f ein Fluss in G . Beweisen Sie die folgende Gleichung:

$$c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u),$$

wobei c_f wie in der Vorlesung definiert ist. Geben Sie bei jedem Schritt eine Begründung an (z.B. folgt aus der Definition o.ä.). Was ist die Aussage der obigen Gleichung? (2 Pkt.)

Aufgabe 7.1.2

Seien u und v Knoten in einem Flussnetzwerk $G = (V, E)$ derart, dass $(u, v) \notin E$ und $(v, u) \notin E$ gilt. Beweisen Sie, dass dann $f(u, v) = f(v, u) = 0$ gilt. Begründen Sie auch hier Ihre Schritte. (2 Pkt.)

Aufgabe 7.1.3

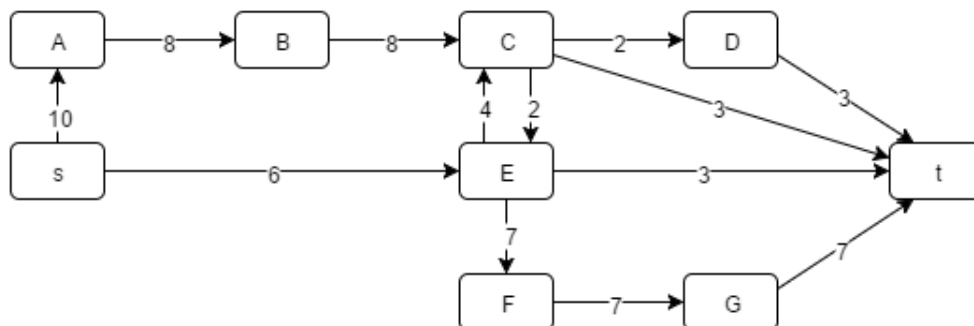
Sei $G = (V, E)$ ein Flussnetzwerk und f ein Fluss in G . Seien $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$. Beweisen Sie

$$\begin{aligned} f(X \cup Y, Z) &= f(X, Z) + f(Y, Z) \\ f(Z, X \cup Y) &= f(Z, X) + f(Z, Y) \end{aligned}$$

Begründen Sie auch hier ihre Schritte. Warum ist die Bedingung $X \cap Y = \emptyset$ nötig? (4 Pkt.)

Aufgabe 7.1.4

Betrachten Sie das folgende Flussnetzwerk:



Bestimmen Sie mit dem Algorithmus von Edmonds-Karp einen maximalen Fluss und diesem Netzwerk. Geben Sie dazu stets als erstes den Erweiterungspfad im Restnetzwerk an (beachten Sie, dass das erste Restnetzwerk dem gegebenen Flussnetzwerk entspricht!), indem Sie einfach die Knoten des Pfades auflisten (z.B. $SABC$ für den Pfad von S nach C über A und B) und den Wert des Flusses angeben. Zeichnen Sie dann das obige Flussnetzwerk mit dem ermittelten gesamten Fluss. Als drittes zeichnen Sie das neue Restnetzwerk. Dann geben Sie sofern möglich einen neuen Erweiterungspfad an und so weiter. (6 Pkt.)

Aufgabe 7.1.5

Sei $G = (V, E)$ mit der Kapazitätsfunktion c , Quelle s und Senke t ein Flussnetzwerk. Sei f ein Fluss in G . Sei (S, T) ein Schnitt. Wie hängen $|f|$, $f(S, T)$ und $c(S, T)$ zusammen? Begründen Sie ihre Antwort. (2 Pkt.)