Zusammenfassung*Algorithmische Mathematik I

gehalten von Prof. Dr. Jens Vygen, Universität Bonn
im Wintersemester 2012/2013

Robert Hemstedt
r@twopi.eu

9. Januar 2013

Hinweise zur Verwendung

Die stets aktuelle Version dieser Zusammenfassung lässt sich finden unter
`http://github.com/euklid/Zusf-AlMaI` .
Dort sind auch die `.tex`-Dateien zu finden, wenn man selbst Veränderungen vornehmen möchte.
Bitte beachtet die **Lizenzhinweise**.

Werter Kommilitone, diese Zusammenfassung basiert zum größten Teil auf meinen Mitschriften unserer AlMa-Vorlesungen bei Prof. Dr. Vygen sowie teilweise auf seinem Skript. Leider ist das Skript nur als PDF-Datei abrufbar, sodass ich mich während meiner Weihnachtsferien mal drangesetzt habe, selbst eine Zusammenfassung zu texen, da die Guttenberg-Tastatur bei LaTeX- Dokumenten leider nicht so effektiv ist.

Was die Nummerierung der Sätze und Kapitelabschnitte angeht, so ist sie nicht mit der aus dem offiziellen Skript identisch. Weiterhin sind manche Sätze, was die Laufzeit von Algorithmen an geht, weggelassen und die Laufzeit beim Algorithmus vermerkt. Ebenso wirst du nur Pseudo-Code finden, die Implementierung bitte im Originalskript beispielsweise nachlesen.

Gedacht ist diese Zusammenfassung explizit als Prüfungsvorbereitung und wird daher auch noch bis zur letzten Vorlesung weiterhin ergänzt. Wenn du Anmerkungen, Ergänzungen, Lob oder Kritik haben solltest, dann sprich mich einfach an, schick mir eine E-Mail oder, was das beste wohl ist, benutze `github.com`, um mir eine *pull*-Request zu schicken.

Ich hafte weder für „fehlende“ Inhalte noch für inhaltliche oder sprachliche Fehler. Weiterhin möchte ich hier an dieser Stelle noch einmal darauf hinweisen, dass das Vorlesungsskript von Herrn Prof. Dr. Vygen auf seiner Homepage mit diesem Link abrufbar ist:
`http://www.or.uni-bonn.de/~vygen/lectures/alma/alma.pdf`
und *er* der Verfasser des offiziellen Vorlesungsskript ist.

Ich hoffe, dass diese Zusammenfassung und der damit verbundene Aufwand sich nicht nur für mich als Verfasser, sondern auch noch für dich als Kommilitone lohnen wird und der Aufwand auch in Form einer möglichst guten Klausurnote entlohnt wird.
Viel Spaß beim Lernen!

*nach meinen persönlichen Aufzeichnungen

Lizenz

Ich veröffentliche dieses Dokument unter der Beerware-Lizenz:

Weitere Hinweise lassen sich finden unter: `http://de.wikipedia.org/wiki/Beerware`

1 Einleitung

Algorithmus (keine formale Definition): „ein Programm “

- Was wird als Eingabe akzeptiert?
- Welche Rechenschritte werden in welcher Reihenfolge (eventuell abhängig von der Eingabe oder Zwischenergebnissen) durchgeführt?
- Wann stoppt der Algorithmus und was wird dann ausgegeben?

Definition 1.1: *Seien A und B zwei Mengen. Dann ist deren* **Produkt** *durch*
$A \times B := \{(a, b) : a \in A, b \in B\}$
Eine **Relation** *auf (A,B) ist eine Teilmenge von $A \times B$*

Definition 1.2: *Seien A und B Mengen und $f \subseteq A \times B$, so dass es für jedes $a \in A$ genau ein b gibt mit $(a, b) \in f$. Dann heißt f* **Funktion** *von A nach B. Statt $(a, b) \in f$ schreiben wir $f(a) = b$ oder $a \mapsto f(a)$.*
Wir schreiben auch $f : A \to B$.
Eine Funktion $f : A \to B$ heißt

- **injektiv**, *wenn $f(a) \neq f(a')\forall\ a, a' \in A$ mit $a \neq a'$*
- **surjektiv**, *wenn $\forall b \in B\ \exists a \in A : f(a) = b$*
- **bijektiv**, *wenn f injektiv und surjektiv (*__Bijektion__*)*

Zwei Mengen A und B sind **gleichmächtig**, *wenn es eine Bijektion $f : A \to B$ gibt. Wir schreiben auch $|A| = |B|$. Eine Menge heißt* **endlich**, *wenn es eine injektive Funktion $f : A \to \{1, \ldots, n\}$ gibt für ein $n \in \mathbb{N}$, andernfalls* **unendlich**. *Eine Menge A heißt* **abzählbar**, *wenn es eine injektive Funktion $f : A \to \mathbb{N}$ gibt, andernfalls* **überabzählbar**.

Definition 1.3: *Sei A eine nichtleere endliche Menge. Für $k \in \mathbb{N} \cup \{0\}$ bezeichnen wir mit A^k die Menge der Funktionen $f : \{1, \ldots, k\} \to A$. Ein Element $f \in A^k$ schreiben wir als Folge $f(1), f(2), \ldots, f(k)$ und bezeichnen es als* **Wort** *(oder* **Zeichenkette***) der* **Länge** *k über ein* **Alphabet** *A. Das einzige Element $\emptyset$ von A^0 nennen wir das* **leere Wort**.
Wir setzen $A^* := \bigcup_{k \in \mathbb{N} \cup \{0\}} A^k$. Ein **Sprache** *über dem Alphabet A ist eine Teilmenge von A^*.*

Definition 1.4: *Ein* **Berechnungsproblem** *ist eine Relation $P \subseteq D \times E$, wobei zu jedem $d \in D$ ein $e \in E$ existiert mit $(d, e) \in P$.*
Wenn $(d, e) \in P$, so ist e eine **korrekte Ausgabe** *für das Problem P mit der Eingabe d.*
P heißt **eindeutig**, *wenn P eine Funktion ist.*
Sind D und E Sprachen über einem endlichen Alphabet, so heißt P **diskretes Berechnungsproblem**.
Sind D und E Teilmenge von $\mathbb{R}^m$ bzw. $\mathbb{R}^n$, so heißt P **numerisches Berechnungsproblem**.
Ein eindeutiges Berechnungsproblem (D, E) mit $|E| = 2$ heißt **Entscheidungsproblem**.

Definition 1.5: *Eine natürliche Zahl $n \in \mathbb{N}$ heißt* **prim**, *wenn $n \geq 2$ ist und es keine natürlichen Zahlen $a > 1$ und $b > 1$ gibt mit $n = a \cdot b$.*

Entscheidungsproblem 1.6 „prim?“:
formal ausgedrückt: $\{(n, e) \in \mathbb{N} \times \{0; 1\} : e = 1 \Leftrightarrow n \text{ prim}\}$
Eingabe: $n \in \mathbb{N}$
Frage: Ist n prim?
Naiver Algorithmus: Teste $n \geq 2$, und dann $\forall a, b \in \{2, \ldots, \lfloor \frac{n}{2} \rfloor\}$, teste ob $n = a \cdot b$.
Braucht bis zu $\left(\frac{n}{2} - 1\right)^2$ Multiplikationen $\Rightarrow$ Laufzeit $O(n^2)$.

Algorithmus 1.7 Einfacher Primzahltest:
Eingabe: $n \in \mathbb{N}$
Ausgabe: die Antwort auf die Frage, ob n prim ist.
if $n = 1$ **then** antwort $\leftarrow$ nein **else** antwort $\leftarrow$ ja
for $i \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**:
    if (i ist Teiler von n) **then** antwort $\leftarrow$ nein
output antwort
Hat Laufzeit $O(\sqrt{n})$.

Definition 1.8: *Für $x \in \mathbb{R}$ sei $\lfloor x \rfloor := \max\{k \in \mathbb{Z} : k \leq x\}$ und $\lceil x \rceil := \min\{k \in \mathbb{Z} : k \geq x\}$.*
Für $a, b \in \mathbb{N}$ sei $a \bmod b := a - b\lfloor \frac{a}{b} \rfloor$

Definition 1.9 Landau-Symbole: *Sei $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann definieren wir:*
$O(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} : \exists \alpha > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq \alpha g(n)\}$
$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} : \exists \alpha > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \geq \alpha g(n)\}$
$\Theta(g) := O(g) \cap \Omega(g)$
Statt $f \in O(g)$ sagen wir auch „f ist $O(g)$“oder sogar „$f = O(g)$“

Berechnungsproblem 1.10 Liste von Primzahlen:
Eingabe: $n \in \mathbb{N}$
Ausgabe: Berechne alle Primzahlen p mit $p \leq n$
Naiver Algorithmus: rufe für $i = 2, \ldots, n$ `primality_check(`i`)` auf
Laufzeit:$\sum_{i=2}^{n} \sqrt{i} = O(n\sqrt{n}) = \Theta(n\sqrt{n})$

Algorithmus 1.11 Sieb des Erathostenes:
Eingabe: $n \in \mathbb{N}$
Ausgabe: Alle Primzahlen p mit $p \leq n$
for $i \leftarrow 2$ **to** n **do**: $p(i) \leftarrow$ja
for $i \leftarrow 2$ **to** n **do:**
    if $p(i) =$ ja **then**:
        output i
        for $j \leftarrow i$ **to** $\lfloor \frac{n}{i} \rfloor$ **do**: $p(i \cdot j) \leftarrow$ nein
Hat Laufzeit $O(n \log n)$.

Lemma 1.12: *Sei $k \in \mathbb{N}, l \in \mathbb{N} \cup \{0\}$. Definiere $f^l : \{0, \ldots, k-1\}^l \to \{0, \ldots, k^l - 1\}$ durch $f^l(w) := \sum_{i=1}^{l} a_i k^{l-i}$ für $w = a_1 \ldots a_l$. Dann ist f^l wohldefiniert und bijektiv.*

Satz 1.13: *Sei A eine nichtleere endliche Menge, Dann ist A^* abzählbar.*

Korollar 1.14: *Die Menge aller C++-Programme ist abzählbar.*

Satz 1.15: *Es gibt Funktionen $f : \mathbb{N} \to \{0; 1\}$, die von keinem C++-Programm berechnet werden.*

Sei Q die abzählbare Menge der C++-Programme, die eine natürliche Zahl als Eingabe akzeptieren. Sei $g : \mathbb{N} \to Q$ eine surjektive Funktion. Definiere $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ durch

$$f(x, y) := \begin{cases} 1, & \text{falls } g(x) \text{ bei Eingabe } y \text{ nach endlich vielen Schritten hält.} \\ 0, & \text{sonst.} \end{cases}$$

Diese Entscheidungsproblem heißt **Halteproblem**.

Satz 1.16: *f wird von keinem C++-Programm berechnet.*

2 Darstellung ganzer Zahlen

Satz 2.1: *Seien $b \in \mathbb{N}, b \geq 2, n \in \mathbb{N}$. Dann existieren eindeutig bestimmte Zahlen $l \in \mathbb{N}$ und $z_i \in \{0, \ldots, b-1\}$ für $i = 0, \ldots, l-1$ mit $z_{l-1} \neq 0$ und*

$$n = \sum_{i=0}^{l-1} z_i b^i.$$

*Das Wort $z_{l-1} \ldots z_0$ heißt auch die b-**adische Darstellung** von n. Es gilt stets $l - 1 = \lfloor \log_b n \rfloor$.*

Definition 2.2: *Seien l und $b \geq 2$ natürliche Zahlen. Sei $n \in \{0, \ldots, b^l - 1\}$. Dann ist*

(a) $K_{b-1}^{b,l}(n) := b^l - 1 - n$ *das (l-stellige) (b-1)-Komplement von n (zur Basis b)*

(b) $K_b^{b,l}(n) := \left(b^l - n\right) \mod b^l$ *das (l-stellige) b-Komplement von n (zur Basis b)*

Lemma 2.3: *Sei $b, l \in \mathbb{N}, b \geq 2$. Sei $n = \sum_{i=0}^{l-1} z_i b^i$ mit $z_i \in \{0, \ldots, b-1\}$ für $i = 0, \ldots, l-1$. Dann gilt:*

(i) $K_{b-1}^{b,l}(n) = \sum_{i=0}^{l-1} (b - 1 - z_i)\, b^i$;

(ii) $K_b^{b,l}(n+1) = \sum_{i=0}^{l-1} (b - 1 - z_i)\, b^i$ *wobei $n \neq b^l - 1$; außerdem $K_b^{b,l}(0) = 0$;*

(iii) $K_{b-1}^{b,l}\left(K_{b-1}^{b,l}(n)\right) = n$;

(iv) $K_b^{b,l}\left(K_b^{b,l}(n)\right) = n$

Satz 2.4: *Seien l und $b \geq 2$ natürliche Zahlen. Sei $Z := \left\{-\left\lceil \frac{b^l-1}{2} \right\rceil, \ldots, \left\lfloor \frac{b^l-1}{2} \right\rfloor\right\}$.*
Sei $f : Z \to \left\{0, \ldots, b^l - 1\right\}$ die durch $z \mapsto \left(z + b^l\right) \mod b^l$ definierte Bijektion. Seien $x, y \in Z$, dann gilt:

(a) *Ist $x + y \in Z$, dann ist $f(x + y) = (f(x) + f(y)) \mod b^l$*

(b) *Ist $x \cdot y \in Z$, dann ist $f(x \cdot y) = (f(x) \cdot f(y)) \mod b^l$*

(c) *Ist $-x \in Z$, so gilt $f(-x) = K_b^{b,l}(x)$*

(d) *Sei b=2, dann ist $x \geq 0$ genau dann, wenn die erste Stelle der l-stelligen Binärdarstellung von x 0 ist.*

3 Rechnen mit ganzen Zahlen

3.1 Addition und Subtraktion

Proposition 3.1: *Für zwei ganze Zahlen x und y können die Summe x+y und Differenz x-y in Laufzeit O(l) berechnet werden, wobei* $l = 1 + \lfloor \log_2(max\{|x|, |y|, 1\}) \rfloor$ *ist.*

3.2 Multiplikation

Zwei Zahlen mit bis zu l Stellen nach der Schulmethode zu multiplizieren braucht $O(l^2)$ Zeit.

Algorithmus 3.2 Karatsuba:
Eingabe: $x, y \in \mathbb{N}$, in Binärdarstellung
Ausgabe: $x \cdot y$ in Binärdarstellung
if $x < 8$ **and** $y < 8$ **then output** $x \cdot y$ (direkte Multiplikation).
else:
$l \leftarrow 1 + \lfloor \log_2(\max\{x, y\}) \rfloor$
$K \leftarrow \lfloor \frac{l}{2} \rfloor$
$B \leftarrow 2^k$
$x' \leftarrow \lfloor \frac{x}{B} \rfloor$
$x'' \leftarrow x \mod B$
$y' \leftarrow \lfloor \frac{y}{B} \rfloor$
$y'' \leftarrow y \mod B$
$p \leftarrow x' \cdot y'$ (rekursiv)
$q \leftarrow x'' \cdot y''$ (rekursiv)
$r \leftarrow (x' + x'') \cdot (y' + y'')$ (rekursiv)
output $p \cdot B^2 + (r - p - q) \cdot B + q$

Satz 3.3: *Die Multiplikation zweier in Binärdarstellung gegebener Zahlen x und y ist mit Karatsubas Algorithmus in Laufzeit* $O(l^{\log_2 3})$ *möglich, wobei* $l = 1 + \lfloor \log_2(\max\{x, y\}) \rfloor$ *ist.*

3.3 Euklidischer Algorithmus

Definition 3.4: *Für* $a, b \in \mathbb{N}$ *ist* $\mathrm{ggT}(a, b)$ *der größte gemeinsame Teiler von a und b. Gilt* $\mathrm{ggT}(a, b) = 1$*, so heißen a und b* **teilerfremd**.
Ferner setzen wir $\mathrm{ggT}(a, 0) := a \ \forall a \in \mathbb{N}, \mathrm{ggT}(0, 0) := 0$. *Für* $a, b \in \mathbb{N}$ *ist das* $\mathrm{kgV}(a, b)$ *das kleinste gemeinsame Vielfache.*

Lemma 3.5: *Für alle* $a, b \in \mathbb{N}$ *gilt:*

(a) $a \cdot b = \mathrm{ggT}(a, b) \cdot \mathrm{kgV}(a, b)$

(b) $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \mod b, b)$

Algorithmus 3.6 Euklidischer Algorithmus:
Eingabe: $a, b \in \mathbb{N}$
Ausgabe: $\mathrm{ggT}(a, b)$
while $a > 0$ **and** $b > 0$ **do:**
if $a < b$ **then** $b \leftarrow b \mod a$ **else** $a \leftarrow a \mod b$
output $\max\{a, b\}$

Fibonacci-Zahlen: $F_0 := 1, F_1 := 1, F_{n+1} := F_n + F_{n-1}, n \in \mathbb{N}$

Lemma 3.7: $\forall n \in N \cup \{0\}$:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Lemma 3.8: *Falls $a > b > 0$ und Euklidischer Algorithmus $k \geq 1$ Iterationen der while-Schleife durchläuft, so gilt $a \geq F_{k+2}$ und $b \geq F_{k+1}$.*

Satz 3.9 Lamé: *Falls $a \geq b$ und $b < F_{k+1}$ für ein $k \in \mathbb{N}$, so durchläuft der euklidische Algorithmus weniger als k Iterationen der while-Schleife.*

Bemerkung 3.10: $k \in \mathbb{N} : \mathrm{ggT}(F_{k+2}, F_{k+1})$ braucht k Iterationen.
$F_n > \Phi^{n-2}$ mit $\Phi = \frac{1+\sqrt{5}}{2}$

Korollar 3.11: *Falls $a \geq b \geq 0$, so durchläuft euklidischer Algorithmus höchstens $\lceil \log_\Phi b \rceil$ Iterationen.*

4 Approximative Darstellung reeller Zahlen

4.1 b-adische Darstellung reeller Zahlen

Satz 4.1: *Sei $b \in \mathbb{N}, b \geq 2$. Sei $x \in \mathbb{R} \setminus \{0\}$. Dann existieren eindeutig bestimmte Zahlen $E \in \mathbb{Z}, \sigma \in \{-1; 1\}, z_i \in \{0, 1, \ldots, b-1\}$ für $i \in \mathbb{N} \cup \{0\}$ mit $\{i \in \mathbb{N} : z_i \neq b-1\}$ unendlich, $z_0 \neq 0$ und*

$$x = \sigma \cdot b^E \cdot \left(\sum_{i=0}^{\infty} z_i b^{-i} \right) \tag{4.1}$$

(4.1) *heißt die (normalisierte)* **b-adische Darstellung** *von x.*

4.2 Maschinenzahlen

Definition 4.2: *Seien $b, m \in \mathbb{N}, b \geq 2$. Eine b-adische m-stellige* **normalisierte Gleitkommazahl** *hat die Form*

$$x = \sigma \cdot b^E \cdot \left(\sum_{i=0}^{m} z_i b^{-i} \right)$$

mit dem Vorzeichen $\sigma \in \{-1; 1\}$, der Mantisse $\sum_{i=0}^{m} z_i b^{-i}$ mit $z_0 \neq 0, z_i \in \{0, \ldots, b-1\}$, für $i = 0, \ldots, m$ sowie dem Exponenten $E \in \mathbb{Z}$.

Für die Darstellung am Computer müssen wir b, m und den zulässigen Bereich für den Exponenten $E \in \{E_{\min}, \ldots, E_{\max}\}$ festlegen.
Das bestimmt den **Maschinenzahlbereich** $F(b, m, E_{\min}, E_{\max})$, das ist die Menge der b-adischen m-stelligen normalisierten Gleitkommazahlen mit Exponent $E \in \{E_{\min}, \ldots, E_{\max}\}$ zuzüglich der Zahl 0.
IEEE-Standard 754: `double`: $b = 2, m = 52, E \in \{-1022, \ldots, 1023\}$. $F_{\texttt{double}} := F(2, 52, -1022, 1023)$. $z_0 = 1$ muss nicht gespeichert werden (hidden Bit). Die freien Exponenten werden genutzt, um z.B. $0, \pm\infty, NaN$ zu speichern. Für den Exponenten E wird die sogenannte **Bias-Darstellung** verwendet, wodurch zum Exponenten der feste Betrag 1023 hinzuaddiert wird,

um alle abgespeicherten Exponenten positiv zu machen.
$\max F_{\texttt{double}}$ =`std::numeric_limits<double>::max()`$=(2-2^{-52})\cdot 2^{1023} \approx 1,79 \cdot 10^{308}$
$\min\{f \in F_{\texttt{double}}, f > 0\}$ =`std::numeric_limits<double>::min()`$=2^{-1022} \approx 2,23 \cdot 10^{-308}$
$\text{range}(F)=[\min\{f \in F : f > 0\}, \max F]$

4.3 Rundung

Definition 4.3: *Sei $F = F(b, m, E_{min}, E_{max})$ ein Maschinenzahlbereich. Eine* **Rundung** *(zu F) ist eine Funktion $rd : \mathbb{R} \to F$, wenn für alle $x \in \mathbb{R}$ gilt: $|x - rd(x)| = \min_{a \in F} |x - a|$.*

Definition 4.4: *Seien $x, \tilde{x} \in \mathbb{R}$ (wobei $\tilde{x}$ eine Näherung von x sein soll). Dann heißt $|x - \tilde{x}|$ der* **absolute Fehler** *und, falls $x \neq 0$, $\left|\frac{x-\tilde{x}}{x}\right|$ der* **relative Fehler**.

Definition 4.5: *Sei F ein Maschinenzahlbereich. Die* **Maschinengenauigkeit** *$eps(F)$ von F ist definiert durch $eps(F) := sup\left\{\left|\frac{x-rd(x)}{x}\right| : |x| \in range(F), rd \text{ Rundung zu } F\right\}$.*

Satz 4.6: *Für jeden Maschinenzahlbereich $F = F(b, m, E_{min}, E_{max})$ gilt: $eps(F) = \frac{1}{1+2 \cdot b^m}$.*

$eps(F_{\texttt{double}}) = \frac{1}{1+2^{53}} \approx 1,11 \cdot 10^{-16}$
`std::numeric_limits<double>::epsilon()` liefert die kleinste darstellbare Zahl, die zu 1 ohne Rundungsfehler addiert werden kann.

Definition 4.7: *Sei F ein Maschinenzahlbereich und $s \in \mathbb{N}$. Eine Maschinenzahl $f \in F$ hat (mindestens) s* **signifikante Stellen** *in der b-adischen Gleitkommadarstellung, falls $f \neq 0$ und für jede Rundung rd und jede Zahl $x \in \mathbb{R}$ mit $rd(x) = f$ gilt: $|x - f| \leq \frac{1}{2} b^{\lfloor \log_b |f| \rfloor + 1 - s}$*

Jedes $f \in F_{\texttt{double}}$ hat mindestens 15 signifikante Stellen.
`#include<iomanip>`
`std::cout<<std::setprecision(15);` für 15 signifikante Stellen bei der Ausgabe in C++

4.4 Maschinenzahlarithmetik

Statt $+, -, \cdot, /$ werden Ersatzoperationen $\oplus, \ominus, \odot, \oslash$ durchgeführt, mit der **Annahme**
$x \circledcirc y = rd(x \circ y)$ für $\circ \in \{+, -, \cdot, /\}$ und $x, y \in F$, für eine Rundung rd zu F.
Für $\oplus, \odot$ gilt das Kommutativgesetz. Assoziativität und Distributivität gelten nicht.

5 Rechnen mit Fehlern

Fehlerarten:

- Datenfehler
  - nicht darstellbare Eingaben
  - Messfehler
- Rundungsfehler
- „Verfahrensfehler“: bspw. Ausgabe wird nur approximiert, eventuell nicht darstellbar.

5.1 Binäre Suche

Beispiel 5.1 Binäre Suche für $\sqrt{3}$: Bezeichne $[l_i; u_i]$ das Intervall der Iteration i und m_i der Mittelpunkt, so wissen wir **a priori**, dass $\left|m_i - \sqrt{3}\right| \leq \frac{1}{2}\left|u_i - l_i\right| \leq 2^{-i|U-L|}$. Im Nachhinein (**a posteriori**) kann man oft bessere Schranken angeben:

$$|m_i - \sqrt{3}| = \frac{|m_i^2 - 3|}{m_i + \sqrt{3}} \leq \frac{|m_i^2 - 3|}{m_i + l_i}$$

.

Algorithmus 5.2 Binäre Suche, diskret:
Eingabe: Ein **Orakel** zur Berechnung einer monoton wachsenden Funktion $f : \mathbb{Z} \to \mathbb{R}$.
$L, U \in \mathbb{Z}$ mit $L \leq U$ und $y \in \mathbb{R}$ mit $y \geq f(L)$
Ausgabe: Das maximale $n \in \{L, \ldots, U\}$ mit $f(n) \leq y$
$l \leftarrow L$
$u \leftarrow U + 1$
while $u > l + 1$ **do:**
    $m \leftarrow \left\lfloor \frac{l+u}{2} \right\rfloor$
    if $f(m) > y$ **then** $u \leftarrow m$ **else** $l \leftarrow m$
output l

Satz 5.3: *Der angegebene Algorithmus für die diskrete Binäre Suche arbeitet korrekt und terminiert nach $O(\log(U - L + 2)$ Iterationen.*

5.2 Fehlerfortpflanzung

Lemma 5.4: *Seien $x, y \in \mathbb{R} \setminus \{0\}$ und $\tilde{x}, \tilde{y}$ Näherungen mit $\varepsilon_x := \frac{x-\tilde{x}}{x}; \varepsilon_y := \frac{y-\tilde{y}}{y}$. Dann gilt für $\varepsilon_\circ := \frac{x \circ y - (\tilde{x} \circ \tilde{y})}{x \circ y}$ mit $\circ \in \{+, -, \cdot, /\}$:*

$$\begin{aligned}
\varepsilon_+ &= \varepsilon_x \cdot \frac{x}{x+y} + \varepsilon_y \cdot \frac{y}{x+y} \\
\varepsilon_- &= \varepsilon_x \cdot \frac{x}{x-y} - \varepsilon_y \cdot \frac{y}{x-y} \\
\varepsilon_\cdot &= \varepsilon_x + \varepsilon_y - \varepsilon_x \cdot \varepsilon_y \\
\varepsilon_/ &= \frac{\varepsilon_x - \varepsilon_y}{1 - \varepsilon_y}
\end{aligned}$$

5.3 Kondition

Definition 5.5: *Sei $P \subseteq D \times E$ ein numerisches Berechungsproblem mit $D, E \subseteq \mathbb{R}$. Sei $d \in D$ (eine Instanz von P) mit $d \neq 0$. Dann ist die* **(relative) Kondition** *von d, oft $\kappa(d)$ genannt, definiert als*

$$\lim_{\varepsilon \to 0} \sup \left\{ \inf \left\{ \frac{\left|\frac{e-e'}{e}\right|}{\left|\frac{d-d'}{d}\right|} : e \in E : (d, e) \in P \right\} : d' \in D, e' \in E, (d', e') \in P, 0 < |d - d'| < \varepsilon \right\},$$

wobei $\frac{0}{0} := 0$.
Bezeichnet $\kappa(d)$ die relative Kondition von d, so ist $\sup\{\kappa(d) : d \in D, d \neq 0\}$ die (rel.) Kondition von P.
Analog: **absolute Kondition** *mit Betrachtung der absoluten Fehler.*

Satz 5.6: *Sei $f : D \to E$ ein eindeutiges numerisches Berechnungsproblem mit $D, E \subseteq \mathbb{R}$, und sei $d \in D, d \neq 0$ und $f(d) \neq 0$. Dann ist ihre Kondition gleich:*

$$\frac{|d|}{|f(d)|} \cdot \lim_{\varepsilon \to 0} \sup \left\{ \frac{|f(d) - f(d')|}{|d - d'|}, d' \in D, |d - d'| < \varepsilon \right\}$$

Korollar 5.7: *Sei $f : D \to E$ ein eindeutiges numerisches Berechnungsproblem, wobei $D, E \subseteq \mathbb{R}$ und f differenzierbar ist. Dann ist die Kondition einer Instanz $d \in D$ mit $d \neq 0$ und $f(d) \neq 0$ gleich*

$$\frac{|f'(d)| \cdot |d|}{|f(d)|}$$

Ideal ist es, wenn die Kondition ≤ 1 ist. Dann spricht man von einem „gut konditioniertem Problem“.

5.4 Fehleranalyse

Ein Algorithmus heißt **numerisch stabil**, wenn jeder seiner Rechenschritte gut konditioniert ist.

- Bei der **Vorwärtsanalyse** untersucht man, wie sich der relative Fehler im Laufe einer Rechnung akkumuliert.
  - $x \oplus y = rd(x + y) = (x + y)(1 + \varepsilon)$ mit $|\varepsilon| \leq eps(F)$
- Bei der **Rückwärtsanalyse** wird jedes Zwischenergebnis als exaktes Ergebnis für gestörte Daten interpretiert, und es wird abgeschätzt, wie groß die Eingangsdatenstörung sein müsste, damit das berechnete Ergebnis korrekt wäre.
  - $x \oplus y = (1 + \varepsilon)x + (1 + \varepsilon)y = \tilde{x} + \tilde{y}$ für $|\varepsilon| \leq eps(F)$

5.5 Intervallarithmetik

Statt mit Zahlen wird mit Intervallen gerechnet. Schon eine Eingabe x kann durch ein Intervall $[rd^-(\min\{y \in \mathbb{R} : rd(y) = x\}), rd^+(\max\{y \in \mathbb{R} : rd(y) = x\})]$ ersetzt werden, die die gewünschte Eingabe enthält. Dabei bezeichnen rd^- und rd^+ die Ab- bzw. Aufrundung zur nächsten kleinsten bzw. größten Maschinenzahl.
Damit kann man dann rechnen: $[a_1, b_1] \oplus [a_2, b_2] = [rd^-(a_1 + a_2), rd^+(b_1 + b_2)]$.

5.6 Newton-Verfahren

Gesucht: Nullstelle einer differenzierbaren Funktion $f : \mathbb{R} \to \mathbb{R}$.
„Rate“ Startwert x_0.
Setze iterativ $x_{n+1} \leftarrow x_n - \frac{f(x_n)}{f'(x_n)}$.
Konvergiert unter bestimmten Voraussetzungen.

Beispiel 5.8 Babylonisches Wurzelziehen: Wir suchen die positive Nullstelle der Funktion $f(x) = x^2 - a$.

$$x_{n+1} \leftarrow x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{5.1}$$

z.B. mit Startwert $x_0 = 1$

Definition 5.9: *Sei $(x_n)_{n\in\mathbb{N}}$ eine konvergente Folge und $x^* := \lim_{n\to\infty} x_n$.*

- *Existiert eine Konstante $c < 1$, so dass*
$$|x_{n+1} - x^*| \leq c|x_n - x^*| \quad \forall n \in \mathbb{N},$$
so hat die Folge (mindestens) **Konvergenzordnung** *1 (konvergiert linear). Ein Beispiel ist die Binäre Suche mit $c = \frac{1}{2}$.*
- *Sei $p > 1$. Existiert eine Konstante $c \in \mathbb{R}$, so dass*
$$|x_{n+1} - x^*| \leq c|x_n - x^*|^p \quad \forall n \in \mathbb{N},$$
so hat die Folge (mindestens) Konvergenzordnung p ($p = 2$: konvergiert quadratisch)

Satz 5.10: *Sei $a \geq 1, x_0 \geq 1$. Für die in (5.1) definierte Folge $(x_n)_{n\in\mathbb{N}}$ gilt:*

- $x_1 \geq x_2 \geq \ldots \geq \sqrt{a}$
- *Die Folge konvergiert quadratisch gegen $\lim_{n\to\infty} x_n = \sqrt{a}$, genauer gilt: $\forall n \geq 0$* $x_{n+1} - \sqrt{a} \leq \frac{1}{2}\left(x_n - \sqrt{a}\right)^2$
- *$\forall n \geq 1$ gilt:* $x_n - \sqrt{a} \leq x_n - \frac{a}{x_n} \leq 2\left(x_n - \sqrt{a}\right)$

6 Graphen

Definition 6.1: *Ein* **ungerichteter Graph** *ist ein Tripel (V, E, Ψ), wobei V und E endliche Mengen sind, $V \neq \emptyset$, und $\Psi : E \to \{X \subseteq V : |X| = 2\}$.*
Ein **gerichteter Graph** *ist ein Tripel (V, E, Ψ), wobei V und E endliche Mengen sind, $V \neq \emptyset$ und $\Psi : E \to \{(v, w) \in V \times V : v \neq w\}$.*
Ein Graph ist ein gerichteter Graph (oder auch **Digraph***) oder ein ungerichteter Graph.*
Die Elemente von V heißen **Knoten** *und die Elemente von E* **Kanten**.
Zwei Kanten $e \neq e'$ mit $\Psi(e) = \Psi(e')$ heißen parallel. Graphen ohne parallele Kanten heißen einfach. In einfachen zusammenhängenden Graphen kann man jede Kante e mit $\Psi(e)$ identifizieren. Man schreibt $G = (V(G), E(G))$, wobei
$E(G) \subseteq \{\{v, w\} : v, w \in V(G), v \neq w\}$ *bzw.* $E(G) \subseteq \{(v, w) : v, w \in V(G), v \neq w\}$
$\Psi(e) = \{v, w\}$ *oder* $\Psi(e) = (v, w)$. *Man sagt e* **verbindet** *v und w, e* **geht von** *v* **nach** *w, e ist mit v und w* **inzident**, *v und w sind* **Nachbarn (adjazent)**, *v und w sind die* **Endknoten von** *e.*

Definition 6.2: *Sei G ein gerichteter Graph und $X \subseteq V(G)$. Dann sei*
$$\delta(X) := \{e = \{x, y\} \in E(G) : x \in X, y \in V(G) \setminus X\}$$
Die **Nachbarschaft** *von X ist die Menge*
$$\Gamma(X) := \{v \in V(G) \setminus X : \delta(X) \cap \delta(\{v\}) \neq \emptyset\}$$
Sei G ein Digraph und $X \subseteq V(G)$. Dann sei
$$\delta^+(X) := \{e = (x, y) \in E(G) : x \in X, y \in V(G) \setminus X\},$$
$$\delta^-(X) := \delta^+(V(G) \setminus X) \text{ und } \delta(X) := \delta^+(X) \cup \delta^-(X).$$
Für $x \in V(G)$ schreiben wir $\delta(x) := \delta(\{x\})$ usw. Der **Grad** *eines Knotens v ist $|\delta(v)|$, d.h. die Anzahl der mit v inzidenten Knoten. Im gerichteten Fall unterscheidet man noch zwischen* **Eingangsgrad** *$|\delta^-(v)|$ und* **Ausgangsgrad** *$|\delta^+(v)|$.*

Satz 6.3: *Für jeden Graph G gilt $\sum_{v\in V(G)} |\delta(v)| = 2|E(G)|$.*

Satz 6.4: *Für jeden Digraphen G gilt $\sum_{v\in V(G)} |\delta^-(v)| = \sum_{v\in V(G)} |\delta^+(v)|$.*

Korollar 6.5: *In jedem Graphen ist die Anzahl der Knoten mit ungeradem Grad gerade.*

Definition 6.6: *Ein Graph H ist ein* **Teilgraph (=Subgraph=Untergraph)** *des Graphen G, wenn $V(H) \subseteq V(G)$ und $E(H) \subseteq E(G)$ (und auch Ψ auf den Kanten in H übereinstimmt). Falls $V(H) = V(G)$, so heißt H* **aufspannender Teilgraph**. *H heißt* **induzierter Teilgraph**, *wenn $E(H) = \{\{x, y\} \in E(G) : x, y \in V(H)\}$ bzw. $E(H) = \{(x, y) \in E(G) : x, y \in V(H)\}$. Man nennt H auch den von $V(H)$ induzierten Teilgraphen und schreibt $H{=}G[V(H)]$ Weiterhin definiere $G - v := G[V(G) \setminus \{v\}]$ und $G - e := G(V(G), E(G) \setminus \{e\})$.*

6.1 Wege und Kreise

Definition 6.7: *Ein Kantenzug (von x_1 nach x_{k+1}) in einem Graphen G ist eine Folge $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_{k+1}$ mit $k \in \mathbb{N} \cup \{0\}$ und $e_i = (x_i, x_{i+i}) \in E(G)$ bzw. $e_i = \{x_i, x_{i+i}\} \in E(G)$ für $i = 1, \ldots, k$. Falls $x_1 = x_{k+1}$, so handelt es sich um einen* **geschlossenen Kantenzug**. *Ein* **Weg** *ist ein Graph $P = (\{x_1, \ldots, x_{k+1}\}, \{e_1, \ldots, e_k\})$, so dass $x_i \neq x_j$ für $1 \leq i < j \leq k + 1$ und $x_1, e_1, x_2, e_2, \ldots, e_k, x_{k+1}$ ist ein Kantenzug. P heißt auch $x_1 - x_{k+1}-$***Weg*** *oder* **Weg von** *x_1* **nach** *x_{k+1}. x_1 und x_{k+1} heißen* **Endknoten** *von P, die anderen Knoten* **innere Knoten** *von P. Ein* **Kreis** *ist ein Graph $C = (\{x_1, \ldots, x_k\}, \{e_1, \ldots, e_k\})$ mit $k \geq 2, x_i \neq x_j$ und $e_i \neq e_j$ für $1 \leq i < j \leq k$ und $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_1$ ist ein geschlossener Kantenzug. Die* **Länge** *eines Weges oder Kreises ist die Anzahl seiner Kanten. Ist ein Weg oder Kreis Teilgraph von G, so sprechen wir auch von einem* **Weg** *oder* **Kreis in** *G. In einem Graph G heißt ein Knoten y von einem Knoten x aus* **erreichbar**, *wenn es einen x-y-Weg gibt.*

Lemma 6.8: *Sei G ein Graph, $x, y \in V(G)$. Es gibt genau dann einen x-y-Weg in G, wenn es einen Kantenzug von x nach y gibt.*

„Erreichbarkeit“ist eine transitive Relation auf $V(G)$. Im ungerichteten Fall ist dies eine Äquivalenzrelation.
Zwei Graphen heißen **knoten- bzw. kantendisjunkt**, wenn sie keine Knoten bzw. Kanten gemeinsam haben.

Lemma 6.9: *Sei G ein ungerichteter Graph mit $|\delta(V)|$ gerade $\forall v \in V(G)$ oder ein gerichteter Graph mit $|\delta^-(v)| = |\delta^+(v)|$ $\forall v \in V(G)$. Dann existiert ein $k \geq 0$ und paarweise kantendisjunkte Kreise $C_1, \ldots, C_k$, sodass $E(G) = E(C_1) \cup \ldots \cup E(C_k)$.*

Für zwei Mengen A und B sei $A \triangle B := (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$ die **symmetrische Differenz** von A und B. Mit $A \dot\cup B$ bezeichnen wir die **disjunkte Vereinigung**, in der Elemente aus $A \triangle B$ einfach und Elemente aus $A \cap B$ doppelt vorkommen.

Lemma 6.10: *Sei G ein Graph, und seien P ein s-t-Weg in G und Q ein t-s-Weg mit $P \neq Q$. Dann enthält $(V(P) \cup V(Q), E(P) \cup E(Q))$ einen Kreis.*

6.2 Zusammenhang und Bäume

Sei $\mathcal{F}$ eine Familie von Mengen bzw. Graphen. Dann heißt $F \in \mathcal{F}$ **inklusionsminimal** bzw. **minimal**, falls keine echte Teilmenge bzw. kein echter Teilgraph von F in $\mathcal{F}$ ist. $F \in \mathcal{F}$ heißt **inklusionsmaximal** bzw. **maximal** falls F keine echte Teilmenge bzw. kein echter Teilgraph eines Elementes von $\mathcal{F}$ ist.

Definition 6.11: *Sei G ein ungerichteter Graph. G heißt* **zusammenhängend***, falls es für je zwei Knoten $x, y \in V(G)$ einen x-y-Weg gibt, sonst heißt G* **unzusammenhängend***. Die maximalen zusammenhängenden Teilgraphen von G sind die* **Zusammenhangskomponenten** *von G. Ein Knoten heißt* **Artikulationsknoten** *von G, wenn er nicht der einzige Knoten von G ist und $G - v$ mehr Zusammenhangskomponenten hat als G. Eine Kante heißt* **Brücke***, wenn $G - e$ mehr Zusammenhangskomponenten hat als G.*

Satz 6.12: *Ein ungerichteter Graph G ist genau dann zusammenhängend, wenn $\delta(X) \neq \emptyset$ für alle $\emptyset \subset X \subset V(G)$.*

Bemerkung 6.13: Man kann also Zusammenhang testen, indem man für alle $\emptyset \subset X \subset V(G)$ testet, ob $\delta(X) \neq \emptyset$ ist. Dies sind aber $2^{|V(G)|} - 2$ Mengen. $\Rightarrow$ zu langsam!

Definition 6.14: *Ein ungerichteter Graph heißt* **Wald***, wenn er keine Kreise enthält. Ein* **Baum** *ist ein zusammenhängender Wald. Ein Knoten vom Grad 1 in einem Baum heißt* **Blatt***.*

Satz 6.15: *Jeder Baum mit mindestens zwei Knoten hat mindestens zwei Blätter.*

Lemma 6.16: *Sei G ein Wald mit n Knoten, m Kanten und p Zusammenhangskomponenten. Dann ist $n = m + p$.*

Satz 6.17: *Sei G ein ungerichteter Graph aus n Knoten. Dann sind folgende Aussagen äquivalent:*

(a) *G ist ein Baum.*

(b) *G hat n-1 Kanten und enthält keinen Kreis.*

(c) *G hat n-1 Kanten und ist zusammenhängend.*

(d) *G ist ein minimaler zusammenhängender Graph mit Knotenmenge $V(G)$. (Das heißt G ist zusammenhängend und jede Kante ist eine Brücke.)*

(e) *G ist ein minimaler Graph mit Knotenmenge $V(G)$ und $\delta(X) \neq \emptyset$ $\forall \emptyset \subset X \subset V(G)$.*

(f) *G ist ein maximaler Wald mit Knotenmenge $V(G)$.*

(g) *Je zwei Knoten in G sind durch genau einen Weg in G verbunden.*

Korollar 6.18: *Ein ungerichteter Graph ist genau dann zusammenhängend, wenn er einen aufspannenden Baum enthält.*

6.3 Starker Zusammenhang und Aboreszenzen

Definition 6.19: *Für einen Digraphen G ist der* **zugrunde liegende zusammenhängende Graph** *G', der Graph auf der Knotenmenge $V(G)$, der für jede Kante (v,w) eine Kante $\{v,w\}$ enthält. G heißt auch eine* **Orientierung** *von G'. Ein Digraph G heißt* **(schwach) zusammenhängend**, *wenn der zugrunde liegende ungerichtete Graph zusammenhängend ist. G heißt* **stark zusammenhängend**, *wenn es für alle $s,t \in V(G)$ einen s-t-Weg und einen t-s-Weg gibt. Die* **starken Zusammenhangskomponenten** *eines Digraphen sind seine maximalen stark zusammenhängenden Teilgraphen.*

Satz 6.20: *Sei G ein Digraph, $r \in V(G)$. Dann gibt es für jedes $v \in V(G)$ einen r-v-Weg genau dann, wenn $\delta^+(X) \neq \emptyset$ für alle $X \subset V(G)$ mit $r \in X$.*

Definition 6.21: *Ein Digraph G ist ein* **Branching**, *falls der zugrunde liegende ungerichtete Graph ein Wald ist und $|\delta^-(v)| \leq 1 \ \forall v \in V(G)$. Ein zusammenhängendes Branching ist eine* **Aboreszenz**. *Eine Aboreszenz auf n Knoten hat n-1 Kanten, somit gibt es genau einen Knoten mit $\delta^-(r) = \emptyset$, die* **Wurzel**. *Ist $(v,w) \in E(G)$, so heißt w* **Kind** *von v (und v* **Vorgänger** *von w).*

Satz 6.22: *Sei G ein Digraph mit n Kanten und $r \in V(G)$. Dann sind die folgenden Aussagen äquivalent:*

(a) *G ist eine Aboreszenz mit Wurzel r (d.h. zusammenhängendes Branching mit $\delta^-(r) = \emptyset$).*

(b) *G ist ein Branching mit n-1 Kanten und $\delta^-(r) = \emptyset$.*

(c) *G hat n-1 Kanten und jeder Knoten ist von r aus erreichbar.*

(d) *Jeder Knoten ist von r aus erreichbar und das Entfernen eines beliebigen Knotens zerstört diese Eigenschaft.*

(e) *$\delta^+(X) \neq \emptyset \ \forall X \subset V(G)$ mit $r \in X$ und das Entfernen einer beliebigen Kante zerstört diese Eigenschaft.*

(f) *$\delta^-(r) = \emptyset$ und für jedes $v \in V(G) \setminus \{r\}$ existiert ein eindeutiger Kantenzug von r nach v.*

(g) *$\delta^-(r) = \emptyset$, $|\delta^-(v)| = 1 \ \forall v \in V(G) \setminus \{r\}$ und G enthält keinen Kreis (ist kreisfrei).*

Korollar 6.23: *Ein Digraph ist genau dann stark zusammenhängend, wenn er für jedes $v \in V(G)$ eine Aboreszenz mit Wurzel v enthält.*

6.4 Darstellungen von Graphen

Definition 6.24: *Sei $G = (V,E,\Psi)$ ein Graph mit n Knoten und m Kanten. Die Matrix $A = (a_{x,y})_{x,y \in V} \in \mathbb{Z}^{n \times n}$ heißt* **Adjazenzmatrix** *von G, wenn*
$a_{x,y} = |\{e \in E(G) : \Psi(e) = \{x,y\}$ bzw. $\Psi(e) = (x,y)\}|$.
$A = (a_{x,e})_{x \in V, e \in E} \in \mathbb{Z}^{n \times m}$ heißt **Inzidenzmatrix** *von G , wenn*

$$a_{x,e} = \begin{cases} 1 & \text{wenn } x \text{ Endknoten von } e, \\ 0 & \text{sonst,} \end{cases}$$

wenn G ungerichtet,
bzw. wenn

$$a_{x,e} = \begin{cases} -1 & \textit{wenn } e \textit{ in } x \textit{ beginnt,} \\ 1 & \textit{wenn } e \textit{ in } x \textit{ endet} \\ 0 & \textit{sonst,} \end{cases}$$

wenn G gerichtet.

Sei $\mathcal{G}$ die Menge aller Graphen, $f : \mathcal{G} \to \mathbb{R}_{\geq 0}$ und $g : \mathcal{G} \to \mathbb{R}_{\geq 0}$. Wir sagen: $f = O(g)$, wenn $\alpha > 0$ und $n_0 \in \mathbb{N}$ existieren, sodass $\forall G \in \mathcal{G}$ mit $|V(G)| \geq n_0$ gilt $f(G) \leq \alpha \cdot g(G)$. Analog für Ω und Θ. f misst Laufzeit/Speicherbedarf. g hängt nur von der Anzahl der Knoten (n) und Kanten (m) ab.
Beispielsweise ist der Speicherbedarf der Adjazenzmatrix $\Omega(n^2)$ und der der Inzidenzmatrix ist $\Theta(m \cdot n)$. Für große Graphen mit $\Theta(n)$ Kanten ist das zuviel.
Adjazenzlistendarstellung: speichere für jeden Knoten eine Liste der inzidenten Kanten.

$\boxed{\delta^+(1)} \to \boxed{c} \to \blacksquare$	$\boxed{\delta^-(1)} \to \boxed{a} \to \blacksquare$	$\boxed{a}=(2,1)$
$\boxed{\delta^+(2)} \to \boxed{a} \to \blacksquare$	$\boxed{\delta^-(2)} \to \boxed{e} \to \boxed{b} \to \blacksquare$	$\boxed{b}=(3,2)$
$\boxed{\delta^+(3)} \to \boxed{b} \to \boxed{d} \to \boxed{e} \to \blacksquare$	$\boxed{\delta^-(3)} \to \boxed{c} \to \blacksquare$	$\boxed{c}=(1,3)$
$\boxed{\delta^+(4)} \to \blacksquare$	$\boxed{\delta^-(4)} \to \boxed{d} \to \blacksquare$	$\boxed{d}=(3,4)$
		$\boxed{e}=(3,2)$

Speicherbedarf: $O(n+m)$ mit der **Annahme**, dass Speicherplatz für Knoten- und Kantenindizes in $O(1)$ und Zugriff bzw. Verfolgen eines next-Pointers in $O(1)$ Zeit.

7 Einfache Graphenalgorithmen

7.1 Graphendurchmusterung

Algorithmus 7.1 Graphendurchmusterung:
Eingabe: ein Graph G, $r \in V(G)$
Ausgabe: die Menge $R \subseteq V(G)$ der von r aus erreichbaren Knoten und eine Menge $F \subseteq E(G)$, sodass (R, F) eine Aboreszenz mit Wurzel r bzw. ein Baum ist.
$R \leftarrow \{r\}$
$Q \leftarrow \{r\}$
$F \leftarrow \emptyset$
while $Q \neq \emptyset$ **do:**
    Wähle $v \in Q$
    if $\exists e = (v, w) \in \delta_G^+(v)$ bzw. $e = \{v, w\} \in \delta_G(v)$ mit $w \notin R$
    then $R \leftarrow R \cup \{w\}$, $Q \leftarrow Q \cup \{w\}$, $F \leftarrow F \cup \{e\}$
    else $Q \leftarrow Q \setminus \{v\}$
output (R, F)
kann in linearer Laufzeit $O(m + n)$ implementiert werden, wobei $n = |V(G)|$ und $m = |E(G)|$.

Korollar 7.2: *Die Zusammenhangskomponenten eines ungerichteten Graphen G können in linearer Zeit $O(n + m)$ bestimmt werden.*

Wichtige Variaten:

- **DFS** = Depth-First-Search=Tiefensuche; Q=**Stack**, **LIFO**= Last In, First Out
- **BFS** = Breadth-First-Search=Breitensuche; Q=**Queue**, **FIFO**= First In, First Out

7.2 Breitensuche

Für zwei v und w in einem Graphen G bezeichnet $\text{dist}(v, w)$ die Länge eines kürzesten Weges von v nach w (v-w-Weg) in G (oder ∞, wenn kein v-w-Weg existiert); $\text{dist}(v, w)$ heißt auch **Abstand** von v nach w.

Satz 7.3: *Jeder BFS-Baum enthält einen kürzesten Weg von einem Knoten r zu allen von r aus erreichbaren Knoten. Die Zahlen* $\text{dist}(r, v)$ $\forall v \in V(G)$ *können in linearer Zeit $O(n + m)$ bestimmt werden.*

7.3 Bipartite Graphen

Definition 7.4: *Eine* **Partition** *einer Menge M ist eine Menge nichtleerer, paarweise disjunkter Teilmengen von M, deren Vereinigung M ist. Ein* **leerer Graph** *ist ein Graph ohne Kanten. Ein* **vollständiger Graph** *ist ein einfacher ungerichteter Graph, in dem je zwei Knoten durch eine Kante verbunden sind. Einen vollständigen Graphen auf n Knoten bezeichnet man oft als K_n.*
Die Knotenmengen der Zusammenhangskomponenten eines ungerichteten Graphen G bilden eine Partition von $V(G)$. Eine **Bipartition** *eines ungerichteten Graphen ist eine Partition $\{A, B\}$ der Knotenmenge $V(G) = A \dot\cup B$, so dass $G[A]$ und $G[B]$ leere Graphen sind. Ein Graph heißt* **bipartit**, *wenn er nur einen Knoten hat oder eine Bipartition. Oft schreiben wir für bipartite Graphen $G = (A \dot\cup B, E(G))$ und meinen, dass $\{A, B\}$ eine Bipartition von G ist. Ein* **vollständiger bipartiter Graph** *ist ein Graph G mit Bipartition $V(G) = A \dot\cup B$ und Kantenmenge $E(G) = \{\{a, b\} : a \in A, b \in B\}$. Wir bezeichnen ihn oft mit $K_{|A|,|B|}$.*
Ein **ungerader Kreis** *ist ein Kreis ungerader Länge.*

Satz 7.5 König: *Ein ungerichteter Graph ist genau dann bipartit, wenn er keinen ungeraden Kreis hat. In linearer Zeit kann man zu einem gegebenen ungerichteten Graphen mit mindestens zwei Knoten entweder eine Bipartition oder einen ungeraden Kreis finden.*

7.4 Azyklische Digraphen

Definition 7.6: *Ein Digraph ist* **azyklisch**, *wenn er keinen Kreis enthält.*
Sei G ein Digraph mit n Knoten. Eine **topologische Ordnung** *von G ist eine Ordnung der Knoten $V(G) = \{v_1, \ldots, v_n\}$, sodass $i < j$ für alle Kanten $(v_i, v_j) \in E(G)$ gilt.*

Lemma 7.7: *Ist G ein Digraph mit $\delta^+(v) \neq \emptyset$ $\forall v \in V(G)$, so kann man in ($O(|V(G)|)$ Zeit einen Kreis in G finden. (irgendwo starten und Kanten entlang laufen, bis sich ein Knoten wiederholt.)*

Satz 7.8: *Ein Digraph hat genau dann eine topologische Ordnung, wenn er azyklisch ist. Zu jedem Digraph kann man in linearer Zeit entweder eine topologische Ordnung oder einen Kreis finden.*

8 Sortieren

8.1 Das allgemeine Sortierproblem

Definition 8.1: *Sei S eine Menge. Eine Relation $R \subseteq S \times S$ heißt* **partielle Ordnung** *(von S), wenn für alle $a, b, c \in S$ gilt:*

- $(a, a) \in R$ *(Reflexivität)*
- $((a, b) \in R \wedge (b, a) \in R) \Rightarrow a = b$ *(Antisymmetrie)*
- $((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$ *(Transitivität)*

Statt $(a, b) \in R$ schreibt man oft aRb. Eine partielle Ordnung R ist eine **totale Ordnung** *(von S), wenn $(a, b) \in R \vee (b, a) \in R \; \forall a, b \in S$. Eine* **lineare Erweiterung** *einer partiellen Ordnung R von S ist eine totale Ordnung T von S mit $R \subseteq T$.*

Beispiele: „$\leq$“ ist auf $\mathbb{R}$ eine totale Ordnung. „$\subseteq$“ auf einer Mengenfamilie ist eine partielle, aber i.A. keine totale Ordnung. In einem gerichteten Graphen G ist $R := \{(v, w) :$ w ist von v aus erreichbar$\}$ genau dann eine partielle Ordnung, wenn G azyklisch ist. Für eine endliche Menge S kann man eine partielle Ordnung $\preceq$ durch Nummerierung der Elemente angegeben werden, d.h. durch eine Bijektion $f : S \to \{1, \ldots, n\}$ (wobei $f(s) = |\{x \in S : x \preceq s\}|$ für $s \in S$).

Berechnungsproblem 8.2 Allgemeines Sortierproblem:
Eingabe: eine endliche Menge S mit einer partiellen Ordnung $\preceq$ (durch Orakel gegeben)
Aufgabe: Berechne eine Bijektion $f : S \to \{1, \ldots, n\}$ mit $f(j) \not\preceq f(i)$ für alle $1 \leq i < j \leq n$.

Manchmal hat man mehr Informationen über $\preceq$ oder $\prec$ und kann diese für schnellere Algorithmen nutzen. Beispielsweise, wenn ein $g : S \to \{1, \ldots, k\}$ mit $a \prec b \Leftrightarrow g(a) < g(b)$ bekannt ist, kann man das Sortierproblem in $O(|S| + k)$ Zeit lösen.

8.2 Sortieren durch sukzessive Auswahl

Algorithmus 8.3 Sortieren durch sukzessive Auswahl:
Eingabe und Ausgabe: wie oben, $S = \{s_1, \ldots, s_n\}$
for $i \leftarrow 1$ **to** n **do** $f(i) \leftarrow s_i$
for $i \leftarrow 1$ **to** n **do**:
    for $j \leftarrow i$ **to** n **do:**
        if $f(j) \preceq f(i)$ **then swap**$(f(i), f(j))$
output f
Laufzeit: $O(n^2)$

Für $0 \leq k \leq n$ sei $\binom{n}{k} := \frac{n!}{k! \cdot (n-k)!}$ (wobei $0! := 1$). Für eine endliche Menge S und $0 \leq k \leq |S|$ sei $\binom{S}{k} := \{A \subseteq S : |A| = k\}$. Es ist $\left|\binom{S}{k}\right| = \binom{|S|}{k}$. Der obige Algorithmus ruft $\binom{n}{2}$ mal das Orakel auf (, wenn man die Aufrufe (i, i) weglässt).

Satz 8.4: *Für jeden Algorithmus für das allgemeine Sortierproblem und jedes $n \in \mathbb{N}$ gibt es eine Eingabe $(S, \preceq)$ mit $|S| = n$, für die der Algorithmus mindestens $\binom{n}{2}$ Orakelaufrufe braucht.*

8.3 Sortieren nach Schlüsseln

In vielen Anwendungen sortiert man nach **Schlüsseln**. Für jedes $s \in S$ hat man einen Schlüssel $k(s) \in K$, wobei K eine Menge mit einer totalen Ordnung ist (oft $\mathbb{N}$ oder $\mathbb{R}$ mit der gewohnten Ordnung). Eine partielle Ordnung $\leq$ auf S ist dann durch $a \preceq b :\Leftrightarrow$
$(a = b \vee K(a) < K(b)), (a, b \in S)$ gegeben. Eine so entstandene partielle Ordnung heißt **durch Schlüssel induziert**.

Berechnungsproblem 8.5 Sortieren mit Schlüsseln:
Eingabe: endliche Menge S, $k : S \to K$ und eine totale Ordnung $\leq$ auf K
Aufgabe: Berechne eine Bijektion $f : \{1, \ldots, n\} \to S$ mit $K(i) \leq K(j)$ $\forall 1 \leq i < j \leq n$

Variante: Wir kennen $k, K, \leq$ nur über ein Orakel für $\leq$ (wie im allgemeinen Sortierproblem), wissen aber nun, dass $\leq$ durch Schlüssel induziert ist.
Bekannte Sortierverfahren für dieses Problem:

- Sortieren durch sukzessive Auswahl
- Insertion-Sort
- Bubble-Sort

Diese Verfahren brauchen $O(n^2)$ Vergleiche und diese Schranke ist scharf.

8.4 Merge-Sort

Algorithmus 8.6 Merge-Sort:
Eingabe: endliche Menge $S = \{s_1, \ldots, s_n\}$ eine durch Schlüssel induzierte partielle Ordnung $\leq$ (per Orakel)
Ausgabe: Bijektion $f : \{1, \ldots, n\} \to S$ mit $f(j) \not\preceq f(i)$ $\forall 1 \leq i < j \leq n$.

1. Wenn $|S|$=1, ist nichts zu tun.
2. Partitioniere S in S_1 und S_2 mit $\lfloor \frac{n}{2} \rfloor$ bzw. $\lceil \frac{n}{2} \rceil$ Elementen.
3. Sortiere S_1 und S_2 rekursiv mit Merge-Sort.
4. Verschmelze („merge“) die totalen Ordnungen von S_1 und S_2 zu einer totalen Ordnung von S. (Dies geht in $O(|S|)$ Zeit.)

Laufzeit: $O(n \log n)$.

Satz 8.7: *Selbst, wenn man die Eingabe auf total geordnete Mengen beschränkt, benötigt jeder Sortieralgorithmus, der Informationen über die Ordnung nur durch paarweises Vergleichen zweier Elemente (Orakel aufrufen) gewinnt, zum Sortieren einer n-elementigen Menge mindestens* $\log_2(n!)$ *Vergleiche.*

Bemerkung: $n! \geq \lfloor \frac{n}{2} \rfloor^{\lceil \frac{n}{2} \rceil} \Rightarrow \log_2(n!) \geq \lceil \frac{n}{2} \rceil \log_2 \lfloor \frac{n}{2} \rfloor = \Omega(n \log n)$

8.5 Quicksort

Algorithmus 8.8 Quicksort:
Eingabe: Eine Menge $S = \{s_1, \ldots, s_n\}$, eine durch Schlüssel induzierte partielle Ordnung $\preceq$ auf S (per Orakel)
Ausgabe: Eine Bijektion $f : \{1, \ldots, n\} \to S$ mit $f(j) \not\preceq f(i)$ $\forall 1 \leq i < j \leq n$.

1. Wenn $|S| = 1$ ist, dann ist nichts zu tun.
2. Wähle ein Element $x \in S$ und setze $S_1 := \{s \in S : s \preceq x\} \backslash \{x\}$, $S_2 := \{s \in S : x \preceq s\} \backslash \{x\}$ und $S_x := S \setminus (S_1 \cup S_2)$.

3. Sortiere S_1 und S_2 rekursiv (mit Quicksort).

4. Füge die sortierten Listen S_1, S_x, S_2 aneinander.

Laufzeit: $O(n^2)$

Random-Quicksort: Wähle x in Schritt 2 zufällig (unabhängig, gleichverteilt).

Satz 8.9: *Der Erwartungswert $\overline{T}(n)$ der Laufzeit von Random-Quicksort mit n Elementen ist $O(n \log n)$.*

8.6 Binäre Heaps und Heap-Sort

Eine **Prioritätswarteschlange** (priority queue) ist eine Datenstruktur, die mindestens folgende Operationen zur Verfügung stellt:

- `init`: Einrichten einer leeren Warteschlange (Konstruktur)
- `insert(`s`,`$k(s)$`)`: füge Element s mit Schlüssel $k(s)$ ein.
- s=`extract_min`: finde ein Element mit minimalem Schlüssel und entferne es
- `clear` Löschen der Warteschlange (Destruktor)

Liste/Array/vector: `insert` in $O(1)$; `extract_min` in $O(n)$ (bei n Elementen)
sortiertes Array/vector=sortierte Liste: `find` in $O(n \log n)$ mit binärer Suche; `extract_min`: $O(1)$; `insert`: $O(n)$
Ein **Heap** leistet alle Operationen in $O(\log n)$ Zeit, auch:

- s=`find_min`: Finden eines Elementes mit minimalem Schlüssel, ohne es zu entfernen
- `remove(`s`)`: Entfernen von s
- `decrease_key(`s`,`$k(s)$`)`: Verringern des Schlüssels von s auf $k(s)$

Definition 8.10: *Ein Heap für eine endliche Menge S mit Schlüsseln $k : S \to K$ und einer totalen Ordnung $\leq$ von K ist ein Branching B mit einer Bijektion $f : V(B) \to S$, so dass die* **Heapordnung**

$$k(f(v)) \leq k(f(w)) \ \forall (v, w) \in E(B)$$

gilt. Im Folgenden ist B eine Aboreszenz. Für `find_min` *braucht man dann nur auf das in der Wurzel r gespeicherte Element $f(r)$ zurückgreifen.*

Proposition 8.11: *Sei $n \in \mathbb{N}$. Dann ist der Graph B_n mit $V(B_n) = \{0, \ldots, n-1\}$ und $E(B_n) = \{(i, j) : i \in V(B_n), j \in V(B_n) \cap \{2i+1; 2i+2\}\}$ eine Aboreszenz mit Wurzel 0. Kein Weg in B_n ist länger als $\lfloor \log_2 n \rfloor$.*

Aboreszenzen dieses Typs heißen auch **vollständige Binärbäume**. Ein Heap (B, f) für S heißt **Binärheap**, wenn $B = B_{|S|}$.

Satz 8.12: *Heapsort sortiert n Elemente mit Schlüsseln in $O(n \log n)$ Zeit.*

9 Optimale Bäume und Wege

9.1 Optimale aufspannende Bäume

Berechnungsproblem 9.1 Minimum-Spanning-Tree-Problem:
Eingabe: ein ungerichteter zusammenhängender Graph G, Kantengewichte $c : E(G) \to \mathbb{R}$
Aufgabe: Berechne einen aufspannenden Baum $(V(G), T)$ in G mit $c(T) := \sum_{e \in T} c(e)$ minimal.

Algorithmus 9.2 Kruskals Algorithmus:
Eingabe und Ausgabe: wie oben
Sortiere $E(G) = \{e_1, \ldots, e_m\}$, so dass $c(e_1) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i \leftarrow 1$ **to** m **do:**
    if $(V(G), T \cup \{e_i\})$ ist Wald **then** $T \leftarrow T \cup \{e_i\}$
Mit Laufzeit $O(m \log n)$ implementierbar, wobei $n = |V(G)|$ und $m = |E(G)|$.

Lemma 9.3: *Sei (G, c) eine Instanz des MST-Problems. Dann heiße eine Kantenmenge $F \subseteq E(G)$* **gut***, wenn es einen optimalen aufspannenden Baum (MST) $(V(G), T)$ gibt mit $F \subseteq T$. Sei $F \subset E(G)$ gut, $e \in E(G) \setminus F$. Dann ist $F \cup \{e\}$ gut genau dann, wenn ein $X \subset V(G)$ existiert mit $\delta_G(X) \cap F = \emptyset, e \in \delta_G(X)$ und $c(e) \leq c(f)\ \forall f \in \delta_G(X)$.*

Algorithmus 9.4 Prims Algorithmus:
Eingabe und Ausgabe: wie bei Kruskal.
Wähle $v \in V(G)$ beliebig.
$X \leftarrow \{v\}$
$T \leftarrow \emptyset$
while $X \neq V(G)$ **do:**
    wähle ein $e \in \delta_G(X)$ mit minimalem Gewicht; sei $e = \{x, y\}, x \in X, y \notin X$
    $T \leftarrow T \cup \{e\}$
    $X \leftarrow X \cup \{y\}$
Mit Hilfe von Binärheaps so implementierbar, dass Laufzeit $O(m \log n)$, $n = |V(G)|$, $m = |E(G)|$.

9.2 Kürzeste Wege

Für einen Graphen G mit Kantengewichten $c : E(G) \to \mathbb{R}$ sei $\text{dist}_{(G,c)}(x, y) := \min\{c(E(P)) : P \text{ ein } x\text{-}y\text{-Weg in } G\}$ der **Abstand** von x und y in G (und ∞, wenn y nicht von x aus erreichbar ist). Für einen Weg P bezeichne $c(E(P))$ die **Länge** von P.

Algorithmus 9.5 Dijkstras Algorithmus:
Eingabe: ein Digraph G mit Kantengewichten $c : E(G) \to \mathbb{R}_{\geq 0}$, ein Knoten $s \in V(G)$.
Ausgabe: eine Aboreszenz $A = (R, T)$ in G (mit Wurzel s), sodass R alle von s aus erreichbaren Knoten enthält und $\text{dist}_{(G,c)}(s, v) = \text{dist}_{(A,c)}(s, v)$ für alle $v \in R$.
$R \leftarrow \emptyset$
$Q \leftarrow \{s\}$
$l(s) \leftarrow 0$
while $Q \neq \emptyset$ **do:**
    wähle ein $v \in Q$ mit $l(v)$ minimal.
    $Q \leftarrow Q \setminus \{v\}$

$R \leftarrow R \cup \{v\}$
for $e = (v, w) \in \delta^+(v)$ **do:**
if $w \notin R \cup Q$ **then** $l(w) \leftarrow l(v) + c(e), p(w) \leftarrow e, Q \leftarrow Q \cup \{w\}$
if $w \in Q \setminus R$ **and** $l(v) + l(e) < l(w)$ **then** $l(w) \leftarrow l(v) + c(e), p(w) \leftarrow e$
$T \leftarrow \{p(v) : v \in R \setminus \{s\}\}$
Laufzeit $O(n^2 + m)$, wobei $n = |V(G)|$ und $m = |E(G)|$.

Satz 9.6: *Mit Hilfe von Binärheaps kann man Dijkstras Algorithmus so implementieren, dass er Laufzeit $O(m \log n)$ hat.*

9.3 Konservative Kantengewichte

Definition 9.7: *Sei G ein Graph und $c : E(G) \to \mathbb{R}$. Dann heißt c* **konservativ***, falls es in (G, c) keinen Kreis mit negativem Gewicht gibt.*

Proposition 9.8: *Sei G ein Digraph, $c : E(G) \to \mathbb{R}$ konservativ. Seien $s, w \in V(G), s \neq w$. Sei P ein kürzester s-w-Weg und sei $e = (v, w)$ die letzte Kante von P. Dann ist $P_{[s,v]}$ (der s-v-Teilweg von P) ein kürzester s-v-Weg.*

Sei G ein Graph, $c : E(G) \to \mathbb{R}, s \in V(G)$. Ein **kürzeste-Wege-Baum** mit Wurzel s ist ein Teilgraph H von G, so dass H Baum bzw. Aboreszenz ist und H einen kürzesten s-v-Weg für alle von s aus erreichbaren Knoten v enthält.

Algorithmus 9.9 Moore-Bellman-Ford-Algorithmus:
Eingabe: ein Digraph mit konservativen Kantengewichten $c : E(G) \to \mathbb{R}$, ein Knoten $s \in V(G)$
Ausgabe: ein kürzeste-Wege-Baum, d.h. eine Aboreszenz $A := (R, T)$ in G, so dass R alle von s aus erreichbaren Knoten enthält und $\text{dist}_{(G,c)}(s, v) = \text{dist}_{(A,c)}(s, v)\ \forall v \in R$
$n \leftarrow |V(G)|$
$l(v) \leftarrow \infty\ \forall v \in V(G) \setminus \{s\}$
$l(s) \leftarrow 0$
for $i \leftarrow 1$ **to** $n - 1$ **do:**
for $e = (v, w) \in E(G)$ **do:**
if $l(v) + c(e) < l(w)$ **then:** $l(w) \leftarrow l(v) + c(e)$, $p(w) \leftarrow e$
$R \leftarrow \{v \in V(G) : l(v) < \infty\}$
$T \leftarrow \{p(v) : v \in R \setminus \{s\}\}$
Laufzeit: $O(n \cdot m)$, wobei $n = |V(G)|$ und $m = |E(G)|$.

9.4 Kürzeste Wege mit beliebigen Kantengewichten

Definition 9.10: *Ein Algorithmus, dessen Eingabe aus Nullen und Einsen besteht, heißt* **polynomiell** *(man sagt auch: hat* **polynomielle Laufzeit***), wenn es ein $k \in \mathbb{N}$ gibt, so dass seine Laufzeit $O(n^k)$ ist, wobei n die Länge (Anzahl Bits) der Eingabe ist. Ein Algorithmus, dessen Eingabe auch reelle Zahlen enthalten kann, heißt* **streng polynomiell***, wenn er für rationale Eingaben polynomiell ist und es ein k gibt, sodass die Anzahl seiner Rechenschritte (inklusive Vergleiche und Grundrechenarten mit reellen Zahlen) $O(n^k)$ ist, wobei n die Anzahl der Bits und reellen Zahlen ist ist.*

Satz 9.11: *Das allgemeine kürzeste-Wege-Problem kann in $O(m + n^2 \cdot 2^n)$ Zeit gelöst werden, wobei $n = |V(G)|$, $m = |E(G)|$.*

Ein polynomieller Algorithmus für das allgemeine kürzeste-Wege-Problem existiert genau dann, wenn $P = NP$ ist. Ob $P = NP$ ist, ist eine wichtigste offene Frage der Mathematik. P ist die Menge der Entscheidungsprobleme, für die es einen polynomiellen Algorithmus gibt. NP enthält alle Entscheidungsprobleme mit folgender Eigenschaft: Es gibt ein Polynom p, sodass für alle $n \in \mathbb{N}$ und alle Instanzen mit n Bits, für die die korrekte Antwort „ja“ lautet, eine Folge von höchstens $p(n)$ Bits (Zertifikat genannt) existiert, die „beweist“, dass es sich um eine Ja-Instanz handelt: es gibt einen polynomiellen Algorithmus, der Paare von Instanzen und Zertifikaten korrekt prüft.
Beispiel: Hamiltonkreis-Problem (liegt in NP. Liegt in P genau dann, wenn $P = NP$)
Eingabe: ein ungerichteter Graph
Ausgabe: Enthält G einen aufspannenden Kreis (=Hamiltonkreis)
Beispiel: Traveling-Salesman-Problem (liegt auch in NP. Liegt in P genau dann, wenn $P = NP$)
Eingabe: ein vollständiger ungerichteter Graph G, $c : E(G) \to \mathbb{R}_{\geq 0}$
Ausgabe: Finde einen Hamiltonkreis minimalen Gewichts.

10 Matchings und Netzwerkflüsse

Definition 10.1: *Sei G ein ungerichteter Graph. Eine Menge von Kanten $M \subseteq E(G)$ heißt* **Matching** *in G, falls $|\delta_G(v) \cap M| \leq 1$ für alle $v \in V(G)$ gilt. Ein* **perfektes Matching** *ist ein Matching mit $\frac{|V(G)|}{2}$ Kanten.*

Berechnungsproblem 10.2 Matching-Problem:
Eingabe: ein ungerichteter Graph G
Ausgabe: Berechne ein Matching in G mit maximaler Kardinalität.

Proposition 10.3: *Ein inklusionsmaximales Matching kann in $O(n + m)$ Zeit berechnet werden, wobei $n = |V(G)|$ und $m = |E(G)|$.*

Satz 10.4: *Ist M ein inklusionsmaximales Matching und M^* ein kardinalitätsmaximales, dann gilt $|M| \geq \frac{1}{2}|M^*|$.*

Definition 10.5: *Sei G ein ungerichteter Graph und M ein Matching in G. Ein* **M-augmentierender** *Weg in G ist ein Weg P in G mit $|E(G) \cap M| = |E(P) \setminus M| - 1$, dessen Endpunkte zu keiner Kante von M inzident sind.*

Satz 10.6: *Sei G ein ungerichteter Graph, M ein Matching in G. Dann gibt es genau dann ein Matching M' mit $|M'| > |M|$, wenn es einen M-augmentierenden Weg gibt.*

Algorithmus 10.7 Bipartiter-Matching-Algorithmus:
Eingabe: ein bipartiter Graph G
Ausgabe: ein Matching M in G mit maximaler Kardinalität
$M \leftarrow \emptyset$
$X \leftarrow \{v \in V(G) : \delta(v) \neq \emptyset\}$
Berechne eine Bipartition $V(G) = A \dot\cup B$ von $G[X]$.
while true do:
$V(H) \leftarrow A \dot\cup B \dot\cup \{s, t\}$,
$E(H) \leftarrow (\{s\} \times (A \cap X)) \cup \{(a, b) \in A \times B : \{a, b\} \in E(G) \setminus M\} \cup$
$\{(b, a) \in B \times A : \{a, b\} \in M\} \cup ((B \cap X) \times \{t\})$

if t nicht von s aus in H erreichbar **then stopp**.
Sei P ein $s-t-$Weg in H.
$M \leftarrow M \triangle \{\{v,w\} \in E(G) : (v,w) \in E(P)\}, X \leftarrow X \setminus V(P)$
Laufzeit: $O(m \cdot n)$, wobei $n = |V(G)|$ und $m = |E(G)|$

Satz 10.8 Heiratssatz: *Ein bipartiter Graph $G = (A \dot\cup B, E(G))$ mit $|A| = |B|$ besitzt genau dann ein perfektes Matching, wenn $\Gamma(S)| \geq |S|$ für alle $S \subseteq A$.*