Skript zur Vorlesung

Algorithmische Mathematik I

Prof. Dr. Stefan Hougardy

WS 2008/2009

5. Februar 2009

Inhaltsverzeichnis

Vorwort

Dieses Skript gibt die wesentlichen Inhalte meiner im Wintersemester 2008/2009 an der Universität Bonn gehaltenen Vorlesung *Algorithmische Mathematik 1* wieder. Es wurde basierend auf meinen handschriftlichen Aufzeichnungen von Frau Christina Mohr erstellt. Die Auswahl der Inhalte orientiert sich stark an dem Skript *Algorithmische Mathematik* von Herrn Prof. Harbrecht aus dem Wintersemester 2007/2008. Die Darstellung und Schwerpunktsetzung ist jedoch vielfach unterschiedlich, so dass es mir sinnvoll erschien, ein eigenes Skript zu dieser Vorlesung zu verfassen. Hinweise auf Fehler jedweder Art nehme ich gerne entgegen.

0 Einleitung

Algorithmische Mathematik: Teilgebiet der Mathematik, das sich mit dem Erstellen und Analysieren von Algorithmen (Algorithmus = „Rechenvorschrift“) beschäftigt.

Mohammed ibn Musa Alchwarizmi ($\approx$ 780-840)
(Mohammed, Sohn des Musa aus Chwarizm (heute Chiwa), Stadt in Usbekistan)

- revolutionierte die Mathematik in der westlichen Welt durch ein Buch über das indische Zahlensystem und Rechenvorschriften (Addition, Subtraktion, Multiplikation, Division, Bruchrechnen, Wurzelziehen). Dieses Buch wird als erstes Algebra-Buch betrachtet. Bis zu diesem Zeitpunkt wurden römische Zahlzeichen verwendet.
- Der arabische (daher fälschlicherweise „arabische“ Zahlen genannt) Originaltext ist nicht erhalten.
- Eine erweiterte, lateinische Übersetzung aus dem 12. Jahrhundert ist erhalten. Sie beginnt mit den Worten „Dixit algorizmi laudes deo ...“.
- Für viele Jahrhunderte ist dieses Buch Grundlage für weitere Standardwerke über das Rechnen.
- Im Lateinischen wird „Algorismus“, abgeleitet aus „algorizmi“ für Rechenvorschrift benutzt.

Inhalt der Vorlesung:

Grundlegende numerische und diskrete Algorithmen sowie deren Effizienz und Korrektheit.

Beispiel 0.1 (Berechnung der Quadratwurzel einer natürlichen Zahl)
Betrachte zum Beispiel $\sqrt{14} = 3.74165\ldots$ Wir nutzen die Abschätzung $1 \leq \sqrt{n} \leq n$, die für jede natürliche Zahl gilt.
Zur Berechnung der ersten 5 Nachkommastellen von $\sqrt{14}$ berechne

$$\begin{aligned} &1.00000^2 \\ &1.00001^2 \\ &1.00002^2 \\ &\quad\vdots \\ 3.74165^2 &= 13.9999447\ldots \\ 3.74166^2 &= 14.0000195\ldots \end{aligned}$$

und stoppe, sobald ein Wert größer als 14 erreicht wird. Folglich ist 274 167 mal eine 6-stellige Zahl zu quadrieren.

Sollen k Nachkommastellen von $\sqrt{14}$ berechnet werden, sind mehr als $2 \cdot 10^k$ Rechenschritte notwendig.
Eine Berechnung jeder einzelnen Stelle beschleunigt die Rechnung:

1. Stelle	**2. Stelle**	**3. Stelle**	
$1^2 = 1$	$3.1^2 = 9.61$	$3.71^2 = 13.7641$	
$2^2 = 4$	$\vdots$	$\vdots$	$\cdots$
$3^2 = 9$	$3.7^2 = 13.69$	$3.74^2 = 13.9876$	
$4^2 = 16$	$3.8^2 = 14.44$	$3.75^2 = 14.0625$	

Hierbei sind maximal 9 Rechenschritte pro Stelle nötig. Daher genügen $9 \cdot (k+1)$ Rechenschritte, um k Nachkommastellen von $\sqrt{14}$ zu berechnen. Zum Beispiel $k = 100$:

$$\begin{array}{lcl} 2 \cdot 10^{100} \text{ Rechenschritte} & \Rightarrow & > 10^{80} \text{ Jahre Rechenzeit} \\ 9 \cdot 101 \text{ Rechenschritte} & \Rightarrow & < 1 \text{ ms Rechenzeit} \end{array}$$

Beispiel 0.2 Betrachte die folgende Rekursionsformel:

$$J_{k+1} = \frac{2k}{x} J_k - J_{k-1},$$

wobei $x = 2.13$, $J_0 = 0.149606770448844\ldots$ und $J_1 = 0.564996980564127\ldots$ gegeben sind. Bestimme den Wert von J_{23} zum Beispiel mit dem folgenden Programm:

Algorithmus 0.3

```
#include <iostream>

using namespace std;

main()
{   float x=2.13, J[24];

    J[0] = 0.14960677044884424099315232755094352695277323673223 6;
    J[1] = 0.56499698056412734679532159079179520144053399201887 1;

    for (int k=1; k<=22; ++k)
        J[k+1] = 2*k/x * J[k] - J[k-1];

    cout << J[23];
}
```

Er liefert das Ergebnis $-9.15575 \cdot 10^{12}$. Das ist jedoch völlig falsch, richtig wäre $1.57037227\ldots \cdot 10^{-22}$.
Was ist das Problem? Computer (wie zum Beispiel Taschenrechner) rechnen mit einer gewissen Genauigkeit, zum Beispiel `float`, meist 6 Stellen. Dadurch entstehen

Rechenfehler, die sehr schnell sehr groß werden können.
Abhilfe: Benutze Datentypen mit höherer Rechengenauigkeit, zum Beispiel `double` (z.B. 14 Stellen) oder `long double` (z.B. 18 Stellen). Dies liefert die Ergebnisse 15160.2 beziehungsweise 11835.7, die aber noch immer völlig falsch sind.

Beispiel 0.4 (Multiplikation zweier n-stelliger Zahlen)
Betrachte das Beispiel

$$\begin{array}{r} \underline{8631 \cdot 4297} \\ 34524 \\ 17262 \\ 77679 \\ \underline{60417} \\ 37087407 \end{array}$$

Wieviele elementare Rechenschritte erfordert diese Schulmethode?
Wir nehmen an, dass uns folgende **elementare Rechenoperationen** zur Verfügung stehen:

1. Addition von drei Ziffern mit einem zweistelligen Ergebnis. Zum Beispiel $3+7+5=15$ oder $4+1+2=07$.
2. Multiplikation von zwei Ziffern mit einem zweistelligen Ergebnis, zum Beispiel $7 \cdot 9 = 63$ oder $4 \cdot 2 = 08$.

Betrachte zunächst die Addition zweier n-stelliger Zahlen:

$$\begin{array}{r} 8631 \\ \underline{+\ 4297} \\ 12928 \end{array} \qquad \text{bzw. allgemein} \qquad \begin{array}{rr} & a_1 a_2 \ldots a_n \\ + & b_1 b_2 \ldots b_n \\ \hline & c_1 c_2 c_3 \ldots c_{n+1} \\ \hline & s_1 s_2 s_3 \ldots s_{n+1} \end{array}$$

wobei $a_i + b_i + c_{i+1} = 10c_i + s_i$ für $i = n, n-1, \ldots, 1$ sowie $s_1 = c_1$ und $c_{n+1} = 0$ gilt.
$\Rightarrow$ Man benötigt also höchstens n elementare Rechenoperationen, um die $n+1$-stellige Summe zweier n-stelliger Zahlen zu berechnen.
Multiplikation einer n-stelligen Zahl mit einer Ziffer:

$$\begin{array}{r} \underline{8631 \cdot 7} \\ 5420 \\ \underline{6217} \\ 60417 \end{array}$$

$\Rightarrow n$ Multiplikationen und n Additionen werden gebraucht, da die letzte Stelle einfach übernommen wird. Folglich braucht man höchstens $2n$ elementare Rechenoperationen, um das $n+1$-stellige Produkt einer n-stelligen Zahl mit einer Ziffer zu berechnen.

Die Multiplikation zweier n-stelliger Zahlen setzt sich zusammen aus n Multiplikationen einer n-stelligen Zahl mit einer Ziffer und $n-1$ Additionen von zwei $n+1$-stelligen Zahlen.

$$
\begin{array}{r}
\underline{8631 \cdot 4297} \\
34524 \\
17262 \\
77679 \\
\underline{60417}
\end{array}
\quad \Rightarrow \quad
\begin{array}{r}
60417 \\
+\ 77679 \\
\hline
0837207
\end{array}
\quad \Rightarrow \quad
\begin{array}{r}
0837207 \\
+\ 17262 \\
\hline
02563407
\end{array}
\quad \Rightarrow \quad
\begin{array}{r}
02563407 \\
+\ 34524 \\
\hline
037087407
\end{array}
$$

$\Rightarrow$ In jedem Schritt sind zwei $n+1$-stellige Zahlen zu addieren.
Man benötigt höchstens $n \cdot 2n + (n-1) \cdot (n+1) = 3n^2 - 1$ elementare Rechenoperationen, um zwei n-stellige Zahlen zu multiplizieren.

Geht das auch mit weniger elementaren Rechenoperationen?
Ja! Nutze dazu Folgendes:
Es seien $a = a_1 a_2 \dots a_{2n}$ und $b = b_1 b_2 \dots b_{2n}$ zwei $2n$-stellige Zahlen. Setze

$$
\begin{array}{ll}
a' = a_1 a_2 \dots a_n, & a'' = a_{n+1} a_{n+2} \dots a_{2n} \\
b' = b_1 b_2 \dots b_n, & b'' = b_{n+1} b_{n+2} \dots b_{2n}.
\end{array}
$$

Dann gilt

$$
\begin{aligned}
a \cdot b &= (10^n \cdot a' + a'') \cdot (10^n \cdot b' + b'') \\
&= 10^{2n} \cdot (a' \cdot b') + 10^n \cdot (a' \cdot b'' + a'' \cdot b') + a'' \cdot b'' \\
&= 10^{2n} \cdot (a' \cdot b') + 10^n \cdot ((a' + a'') \cdot (b' + b'') - a' \cdot b' - a'' \cdot b'') + a'' \cdot b''.
\end{aligned}
$$

$\Rightarrow$ Um $a \cdot b$ zu berechnen, reicht es, die drei Produkt $a' \cdot b'$, $a'' \cdot b''$ und $(a' + a'') \cdot (b' + b'')$ zu berechnen. Ansonsten werden nur Additionen bzw. Subtraktionen benötigt.
Man kann zeigen (siehe Übung), dass dieser Ansatz lediglich $99 \cdot n^{1.59}$ elementare Rechenoperationen benötigt.

$$
\begin{array}{lll}
n = 10^6: & \Rightarrow & \text{Schulmethode:} \quad > 3 \cdot 10^{12} \approx 20 \text{ min auf dem PC} \\
 & & \text{obiger Ansatz:} \quad 3.4 \cdot 10^{11} \approx \ 2 \text{ min auf dem PC}
\end{array}
$$

Beispiel 0.5 Betrachte folgendes Gleichungssystem:

$$
\begin{array}{rr}
10^{-20} x + & 2y = 1 \\
10^{-20} x + & 10^{-20} y = 10^{-20}
\end{array}
$$

Subtrahiert man die zweite von der ersten Gleichung, erhält man

$$(2 - 10^{-20}) y = 1 - 10^{-20}.$$

Der Taschenrechner liefert $y = 0.5$. Einsetzen in die erste Gleichung liefert $x = 0$, Einsetzen in die zweite Gleichung $x = 0.5$. Welches Ergebnis ist richtig?
$x = \frac{1}{2 - 10^{-10}} = 0.5000\dots$ und $y = \frac{1 - 10^{-20}}{2 - 10^{-20}} = 0.4999\dots$.

Man kann mit einem Computer Funktionen wie zum Beispiel $\sin(x)$, $\sqrt{x}$, x^2, $\ln(x)$, $\frac{1}{x}$, $n!$ usw. berechnen.
Gibt es auch Funktionen, die ein Computer **nicht** berechnen kann?

Betrachte nur Funktionen $f : \mathbb{N} \to \{0,1\}$. Davon gibt es überabzählbar viele, zum Beispiel für jede reelle Zahl $x \in (0,1)$ sei $f_x(i)$ die i-te Nachkommastelle der Binärdarstellung von x. Es gibt aber nur abzählbar viele C++-Programme, denn jedes entspricht einer endlichen Bitfolge im Speicher eines Computers.

$\Rightarrow$ Es gibt Funktionen $f : \mathbb{N} \to \{0,1\}$, die kein Computer jemals berechnen können wird.

Wie sieht eine solche nicht berechenbare Funktion aus?
Sei C die Menge aller möglichen C++-Programme, wobei jedes Programm eindeutig durch eine Zahl aus $\mathbb{N}$ repräsentiert wird. Das heißt $C \subset \mathbb{N}$. Definiere eine Funktion $f : C \times C \to \{0,1\}$ durch

$$f(x,y) = \begin{cases} 1 & \text{falls das Programm, das durch } x \text{ repräsentiert wird,} \\ & \text{bei Eingabe von } y \text{ nach endlich vielen Schritten hält} \\ 0 & \text{sonst.} \end{cases}$$

Gibt es ein C++-Programm, das als Eingabe ein Programm x und eine Zahl y erhält und nach endlicher Zeit ausgibt, ob Programm x bei Eingabe y nach endlich vielen Schritten anhält?

Angenommen, es gibt ein solches Programm, das f berechnet. Schreibe ein Programm P, das bei Eingabe von x die Funktion $f(x,x)$ berechnet und anhält, falls $f(x,x) = 0$ und in eine Endlosschleife läuft, falls $f(x,x) = 1$.
Was ist der Wert von $f(P,P)$?
Falls $f(P,P) = 1$ so gilt nach Definition von f: Programm P stoppt bei Eingabe P. Nach Konstruktion von P gilt dann aber $f(P,P) = 0$. ↯
Falls $f(P,P) = 0$ so gilt nach Definition von f: Programm P stoppt bei Eingabe P nicht. Nach Konstruktion von P gilt dann aber $f(P,P) = 1$. ↯

$\Rightarrow$ Das Programm P existiert nicht.

Literaturverzeichnis

[1] Vogel, Kurt: *Mohammed ibn Musa Alchwarizmi's Algorismus. Das früheste Lehrbuch zum Rechnen mit indischen Ziffern.* Zeller, Aalen, 1963

[2] Roson, Frederic: *Islamic Mathematics and Astronomy, Volume 1: The algebra of Mohammed Ben Musa.* 1977

[3] Deuflhard, Peter, Nowak, Ulrich und Lutz-Westphal, Brigitte: *Bessel'scher Irrgarten.* ZIB-Report 07-09, Juni 2007

[4] Stoer, Josef und Bulirsch, Roland: *Darstellung von Funktionen im Rechenautomaten.* In Sauer, Szabo (Hrsg): Mathematische Hilfsmittel des Ingenieurs, Teil III. Sprinegr, 1968

[5] Mehlhorn, Kurt und Sanders, Peter: *Algorithms and Data Structures.* Springer, 2008

[6] Schöning, Uwe: *Algorithmik.* Spektrum Akademischer Verlag, 2001

1 Zahlendarstellungen

1.1 Stellenwertsysteme

Zur Zahlendarstellung im Rechner benutzt man sogenannte **Stellenwertsysteme**. Am geläufigsten ist uns das Dezimalsystem, bei dem die Ziffern $0, 1, \ldots, 9$ benutzt werden. Die Position einer Ziffer innerhalb einer Dezimalzahl ergibt die Zehnerpotenz, mit der diese multipliziert werden muss, beispielsweise

$$273 = 2 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0.$$

(Vergleiche mit der Sprechweise!)

Wir wollen nun allgemeine Stellenwertsysteme betrachten. Ein **Alphabet** Σ ist eine Menge von Symbolen , z.B. $\Sigma = \{0, 1, \ldots, 9\}$. Ein **Wort** über einem Alphabet Σ ist eine Aneinanderreihung von Symbolen aus Σ. Die Anzahl der Symbole, die ein Wort w enthält, wird als die **Länge** von w bezeichnet. Für $\Sigma = \{0, 1, \ldots, 9\}$ sind zum Beispiel 7248 und 0042 Worte der Länge 4.

Zu einer natürlichen Zahl $b > 1$ definieren wir als Alphabet des ***b*-adischen Zahlensystems** die Menge $\Sigma_b = \{0, 1, \ldots, b-1\}$. Die Zahl b wird **Basis** des b-adischen Zahlensystems genannt.

Beispiel 1.1

- Das Dezimalsystem benutzt das Alphabet $\Sigma_{10} = \{0, 1, \ldots, 9\}$. Worte über diesem Alphabet sind zum Beispiel 5, 273 oder 7348. Eine feste Wortlänge, beispielsweise $n = 4$, erreicht man durch Hinzufügen von „führenden Nullen“: 0005, 0273, 7348. Im Rechner werden meist Worte fester Länge benutzt (16-, 32-, 64-Bit).
- $\Sigma_2 = \{0, 1\}$ ist das **Dual-** bzw. **Binäralphabet**.
  $\Sigma_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ ist das **Oktalalphabet**,
  $\Sigma_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ das **Hexadezimalalphabet**. Laut Definition gilt $\Sigma_{16} = \{0, 1, \ldots, 15\}$. Man benutzt jedoch $A, B, \ldots, F$ anstelle von $10, 11, \ldots, 15$, um die Notation zu vereinfachen.
  Die Basen 2, 8, 16 und 256 sind für die Zahlen- und Zeichendarstellung in Rechnern besonders wichtig.
- Darüber hinaus haben die Basen $b = 12$ (Dutzend), $b = 20$ (franz.: vingt) und $b = 60$ (Zeit- und Winkelrechnung) gewisse Bedeutung.

Natürliche Zahlen können in b-adischen Zahlensystemen eindeutig dargestellt werden:

Satz 1.2 (*b*-adische Darstellung natürlicher Zahlen)
Seien $b, n \in \mathbb{N}$ und $b > 1$. Dann ist jede Zahl $z \in \mathbb{N}_0 := \mathbb{N} \cup \{0\}$ mit $0 \leq z \leq b^n - 1$ eindeutig als Wort der Länge n über Σ_b darstellbar durch

$$z = \sum_{i=0}^{n-1} z_i b^i,$$

wobei $z_i \in \Sigma_b$ für $i = 0, 1, \ldots, n-1$ ist.

Beweis. Wir beweisen durch Induktion über z die Existenz einer solcher Darstellung:
Induktionsanfang: Für $0 \leq z < b$ hat z die Darstellung $z = \sum_{i=0}^{n-1} z_i b^i$ mit $z_0 = z$ und $z_i = 0$ sonst.
Induktionsannahme: Jede Zahl $\widetilde{z}$ mit $0 \leq \widetilde{z} < z$ besitzt eine Darstellung der Form $\widetilde{z} = \sum_{i=0}^{n-1} \widetilde{z}_i b^i$ mit $\widetilde{z}_i \in \Sigma_b$ für $i = 0, 1, \ldots, n-1$.
Induktionsschritt: Es gilt $z = \lfloor z/b \rfloor \cdot b + (z \bmod b)$, wobei für $x \in \mathbb{R}$ die untere bzw. obere Gaußklammer definiert ist durch

$$\begin{aligned} \lfloor x \rfloor &:= \max\{k \in \mathbb{Z} \mid k \leq x\}, \\ \lceil x \rceil &:= \min\{k \in \mathbb{Z} \mid k \geq x\}. \end{aligned}$$

Setze $\widetilde{z} := \lfloor z/b \rfloor$. Dann gilt $\widetilde{z} < z$, da $b > 1$. Nach der Induktionsannahme hat $\widetilde{z}$ daher eine Darstellung $\widetilde{z} = \sum_{i=0}^{n-1} \widetilde{z}_i b^i$ mit $\widetilde{z}_i \in \Sigma_b$ für $i = 0, 1, \ldots, n-1$. Wegen $\widetilde{z} \cdot b \leq z \leq b^n - 1$ muss $\widetilde{z}_{n-1} = 0$ gelten. Wir erhalten daher:

$$\begin{aligned} z = \widetilde{z} \cdot b + (z \bmod b) &= b \cdot \sum_{i=0}^{n-2} \widetilde{z}_i \cdot b^i + (z \bmod b) \\ &= \sum_{i=1}^{n-1} \widetilde{z}_{i-1} \cdot b^i + (z \bmod b) \\ &= \sum_{i=0}^{n-1} z_i \cdot b^i \end{aligned}$$

mit $z_0 := z \bmod b$ und $z_i := \widetilde{z}_{i-1}$ für $i = 1, 2, \ldots, n-1$.
Es bleibt zu zeigen, dass die Darstellung eindeutig ist. Angenommen ein $z_0 \in \mathbb{N}_0$ besitzt zwei unterschiedliche Darstellungen

$$z = \sum_{i=0}^{n-1} z_i b^i = \sum_{i=0}^{n-1} \widetilde{z}_i b^i.$$

Sei $m := \max\{i \mid 0 \leq i \leq n-1 \text{ mit } z_i \neq \widetilde{z}_i\}$. Für alle i mit $0 \leq i \leq n-1$ gilt $z_i \geq 0$ und $\widetilde{z}_i \leq b - 1$. Damit folgt $z_i - \widetilde{z}_i \geq 1 - b$. Ohne Beschränkung der Allgemeinheit sei

$z_m > \widetilde{z}_m$. Damit gilt

$$\begin{aligned} 0 &= z - z = \sum_{i=0}^{m} z_i b^i - \sum_{i=0}^{m} \widetilde{z}_i b^i \\ &= \sum_{i=0}^{m} (z_i - \widetilde{z}_i) b^i \\ &\geq b^m + \sum_{i=0}^{m-1} (1-b) b^i \\ &= b^m + \sum_{i=0}^{m-1} b^i - \sum_{i=0}^{m-1} b^{i+1} = b^m + b^0 - b^m \\ &= 1. \end{aligned}$$

↯

□

Für eine Zahl z mit $z = \sum_{i=0}^{n-1} z_i b^i$ wird häufig die einfachere **Ziffernschreibweise** $z = (z_{n-1} z_{n-2} \ldots z_0)_b$ benutzt, für $b = 10$ verwendet man die übliche Schreibweise $z_{n-1} z_{n-2} \ldots z_0$. Die Ziffern von Binärzahlen ($b = 2$) werden **Bits** (**b**inary dig**it**) genannt.
Satz 1.2 liefert unmittelbar einen Algorithmus, um für beliebiges b die Ziffernschreibweise einer Zahl $z \in \mathbb{N}_0$ zu erhalten:

Beispiel 1.3 Umwandlung von $z = 1622$ in eine Oktalzahl:

$$\left.\begin{aligned} 1622 &= 202 \cdot 8 + 6 \\ 202 &= 25 \cdot 8 + 2 \\ 25 &= 3 \cdot 8 + 1 \\ 3 &= 0 \cdot 8 + 3 \end{aligned}\right\} \Rightarrow 1622 = (3126)_8$$

Die Umwandlung einer b-adischen Zahl $(z_{n-1} z_{n-2} \ldots z_0)_b$ in eine Dezimalzahl z erfolgt gemäß $z = \sum_{i=0}^{n-1} z_i \cdot b^i$. Zur Verringerung des Rechenaufwands nutzt man die Darstellung

$$\sum_{i=0}^{n-1} z_i \cdot b^i = z_0 + b \cdot (z_1 + b \cdot (z_2 + \ldots + b \cdot (z_{n-2} + b \cdot z_{n-1}) \ldots)),$$

das sogenannte **Horner-Schema**.

1.2 Darstellung ganzer Zahlen

Wir sind gewohnt, negative Zahlen durch Vorausstellen eines „−“-Zeichens zu spezifizieren. In einem Rechner ließe sich dies zum Beispiel dadurch realisieren, dass man in einem Wort der Länge n das erste Symbol für das Vorzeichen reserviert und den Wert 0 mit „+“ und den Wert 1 mit „−“ identifiziert. Diese Darstellung nennt man **Vorzeichen-** oder **Betrags-Darstellung**.

Algorithmus 1.4

```
#include <iostream>

using namespace std;

void OutputBinary (int i)
{   if (i>0)
       { OutputBinary(i/2);
         cout << i % 2;
       }
}

main ()
{   int Zahl;

    cin >> Zahl;
    OutputBinary (Zahl);
    cout << endl << hex << showbase << Zahl << endl;
}
```

Beispiel 1.5 Wir betrachten Binärdarstellungen mit $n = 4$. Die Zahl $+5$ wird als $(0101)_2$ dargestellt, die Zahl -5 als $(1101)_2$. Mit dieser Darstellung lassen sich die Zahlen

$$\begin{array}{lll} 0 = (0000)_2 & \text{sowie} & -0 = (1000)_2 \\ 1 = (0001)_2 & & -1 = (1001)_2 \\ \vdots & & \vdots \\ 7 = (0111)_2 & & -7 = (1111)_2 \end{array}$$

darstellen. Insbesondere gibt es $+0$ und -0.

Ein Nachteil dieser Darstellung ist, dass Addition und Subtraktion unterschiedliche Algorithmen benötigen.

Definition 1.6 *Sei $z = (z_{n-1}z_{n-2}\dots z_0)_b$ eine n-stellige b-adische Zahl.*

(i) $K_{b-1}(z) := (x_{n-1}x_{n-2}\dots x_0)_b$ *mit* $x_i := b - 1 - z_i$ *für* $0 \le i \le n-1$ *ist das* ***$(b-1)$-Komplement*** *von z.*

(ii) Das ***b-Komplement*** *von z wird mit $K_b(z)$ bezeichnet und ist als die n-stellige b-adische Zahl $(K_{b-1}(z) + 1) \bmod b^n$ definiert.*

Beispiel 1.7

- $K_9((109)_{10}) = (890)_{10}$,
- $K_9((0000)_{10}) = (9999)_{10}$,
- $K_1((10110)_2) = (01001)_2$,
- $K_{10}((109)_{10}) = (891)_{10}$,
- $K_{10}((0000)_{10}) = (0000)_{10}$,
- $K_2((10110)_2) = (01010)_2$

Für $b = 2$ nennt man K_1 das **Einerkomplement** und K_2 das **Zweierkomplement**, für $b = 10$ heißt K_9 **Neunerkomplement** und K_{10} **Zehnerkomplement**.

Lemma 1.8 *Sei x eine n-stellige b-adische Zahl. Dann gelten:*

(i) $x + K_{b-1}(x) = b^n - 1$,

(ii) $x + K_b(x) = b^n$, *falls* $x \neq 0$,

(iii) $K_{b-1}(K_{b-1}(x)) = x$,

(iv) $K_b(K_b(x)) = x$.

Beweis.

(i) Sei $x = (x_{n-1}x_{n-2}\dots x_0)_b$. Dann gilt:

$$\begin{aligned} x + K_{b-1}(x) &= \sum_{i=0}^{n-1} x_i b^i + \sum_{i=0}^{n-1} (b-1-x_i)b^i \\ &= \sum_{i=0}^{n-1} (b-1)b^i \\ &= \sum_{i=0}^{n-1} b^{i+1} - \sum_{i=0}^{n-1} b^i = b^n - 1. \end{aligned}$$

(ii) Für $x \neq 0$ gilt: $x + K_b(x) = x + K_{b-1}(x) + 1 = b^n - 1 + 1 = b^n$.

(iii) $K_{b-1}(K_{b-1}(x))$ hat die i-te Ziffer $b - 1 - (b - 1 - x_i) = x_i$.

(iv) Für $x = 0$ gilt $K_b(0) = 0$. Somit gilt insbesondere $K_b(K_b(0)) = 0$. Für $x \neq 0$ gilt nach (ii) $K_b(x) + K_b(K_b(x)) = b^n = x + K_b(x) \Rightarrow K_b(K_b(x)) = x$.

□

Das b-Komplement nutzt man, um negative Zahlen darzustellen, indem man die negative Zahl x durch $K_b(-x)$ repräsentiert. Zum Beispiel wird -109 durch $K_{10}((109)_{10}) = (891)_{10}$ dargestellt.

Woher weiß man aber, ob $(891)_{10}$ die Zahl 891 oder die Zahl -109 darstellt?

Falls $b = 2$ ist, vereinbart man, dass die Zahl $(x_{n-1}x_{n-2}\dots x_0)_2$ genau dann eine negative Zahl repräsentiert, falls $x_{n-1} = 1$. Der Vorteil dieser Vereinbarung ist, dass man am ersten Bit erkennen kann, ob es sich um eine negative Zahl handelt.

Beispiel 1.9 Mit 3-stelligen Binärzahlen werden somit die folgenden Zahlen repräsentiert:

000	0
001	1
010	2
011	3
100	−4
101	−3
110	−2
111	−1

(1.1)

Durch n-stellige Binärzahlen werden die 2^n Werte $0, 1, 2, \ldots, 2^{n-1}-1$ und $-1, \ldots, -2^{n-1}$ repräsentiert. Allgemein vereinbart man, dass man mit n-stelligen b-adischen Zahlen die Werte $0, 1, \ldots, \lfloor b^n/2 \rfloor - 1$ sowie $-1, -2, \ldots, -\lfloor b^n/2 \rfloor$ repräsentiert.

Definition 1.10 *Die **n-stellige b-Komplement Darstellung** einer Zahl $z \in \mathbb{Z}$ mit $-\lfloor b^n/2 \rfloor \leq z \leq \lfloor b^n/2 \rfloor - 1$ ist definiert als*

$$(z)_{K_b} := \begin{cases} (z_{n-1}z_{n-2}\ldots z_0)_b & \text{falls } z \geq 0 \\ K_b((z_{n-1}z_{n-2}\ldots z_0)_b) & \text{sonst,} \end{cases}$$

wobei $(z_{n-1}z_{n-2}\ldots z_0)_b$ die n-stellige b-adische Darstellung von $|z|$ ist.

Beispiel 1.11 $n = 2$, $b = 10$:

$$\begin{aligned} (17)_{K_{10}} &= 17 \\ (-17)_{K_{10}} &= K_{10}((17)_{10}) = (83)_{10} \end{aligned}$$

Beachte: Es gibt Zahlen x, die eine n-stellige b-Komplement-Darstellung besitzen, aber $|x|$ bzw. $-x$ besitzen diese nicht. Betrachte zum Beispiel $n = 2$, $b = 10$. Dann gilt $(-50)_{K_{10}} = K_{10}((50)_{10}) = (50)_{10}$, aber zur Darstellung von $(50)_{K_{10}}$ werden 3 Stellen benötigt.

Satz 1.12 *Seien x, y und $x+y$ Zahlen, die eine n-stellige b-Komplement-Darstellung besitzen. Dann gilt:*

$$(x+y)_{K_b} = ((x)_{K_b} + (y)_{K_b}) \bmod b^n.$$

Beweis. Für $z > 0$ gilt $(z)_{K_b} = z$. Für $z < 0$ gilt $(z)_{K_b} = K_b(-z) = b^n + z$ (nach Definition 1.10 und Lemma 1.8 (ii)). Daraus folgt $z \equiv (z)_{K_b} \pmod{b^n}$.
Also gilt $(x+y)_{K_b} \equiv x + y \equiv (x)_{K_b} + (y)_{K_b} \pmod{b^n}$.
$\Rightarrow\ (x+y)_{K_b} = (x)_{K_b} + (y)_{K_b} + \alpha \cdot b^n$ mit $\alpha \in \mathbb{Z}$.
Da nach Voraussetzung x, y und $x+y$ zwischen $-\lfloor b^n/2 \rfloor$ und $\lfloor b^n/2 \rfloor - 1$ liegen, folgt die Behauptung und es gilt $\alpha = 0$ oder $\alpha = -1$. □

Beispiel 1.13 $n = 2$, $b = 10$:
Addition von 17 und -34:

$$(17)_{K_{10}} = 17, \quad (-34)_{K_{10}} = 66, \quad \Rightarrow 17 + 66 = 83 = (-17)_{K_{10}}.$$

Addition von -17 und 34:

$$(-17)_{K_{10}} = 83, \quad (34)_{K_{10}} = 34, \quad \Rightarrow 83 + 34 = 117 = 17 \pmod{100}.$$

Es kann also ein Übertrag entstehen, auch wenn x, y und $x+y$ jeweils eine n-stellige b-Komplement-Darstellung besitzen.

Der Vorteil der b-Komplement-Darstellung liegt darin, dass sich die Subtraktion auf die Addition zurückführen lässt:

Satz 1.14 *Seien x, y und $x-y$ Zahlen, die eine n-stellige b-Komplement-Darstellung besitzen. Dann gilt:*

$$(x-y)_{K_b} = (x)_{K_b} + K_b((y)_{K_b}) \ mod\ b^n.$$

Beweis. Es gilt $x - y = x + (-y)$. Nach Satz 1.12 gilt daher

$$(x-y)_{K_b} = (x)_{K_b} + (-y)_{K_b} \bmod b^n.$$

Es reicht also zu zeigen, dass $(-y)_{K_b} = K_b((y)_{K_b})$ gilt.

$$\begin{aligned} y > 0 \ &\Rightarrow \ (-y)_{K_b} \overset{\text{Def.}}{=} K_b(y) \overset{\text{Def.}}{=} K_b((y)_{K_b}) \\ y \leq 0 \ &\Rightarrow \ (-y)_{K_b} \overset{\text{Def.}}{=} -y \overset{\text{L. 1.8 (iv)}}{=} K_b(K_b(-y)) \overset{\text{Def.}}{=} K_b((y)_{K_b}). \end{aligned}$$

□

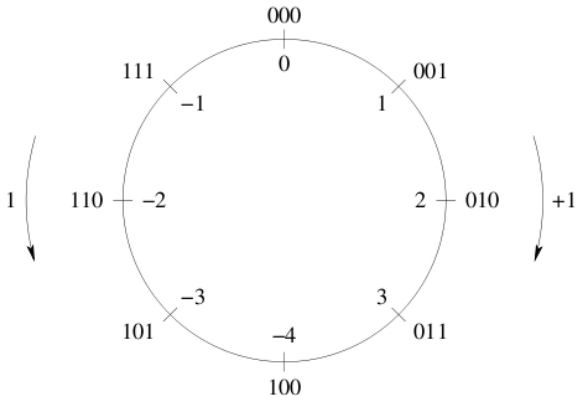
Beispiel 1.15 $n = 2$, $b = 10$:

$17 - 8$: $(17-8)_{K_{10}} = (9)_{K_{10}} = 09$
$(17)_{K_{10}} + K_{10}((8)_{K_{10}}) = 17 + K_{10}(8) = 17 + 92 = 109$

$17 - 34$: $(17-34)_{K_{10}} = (-17)_{K_{10}} = K_{10}(17) = 83$
$(17)_{K_{10}} + K_{10}((34)_{K_{10}}) = 17 + K_{10}(34) = 17 + 66 = 83$

$(-17) - (-34)$: $((-17)-(-34))_{K_{10}} = (17)_{K_{10}} = 17$
$(-17)_{K_{10}} + K_{10}((-34)_{K_{10}}) = K_{10}(17) + K_{10}(K_{10}(34))$
$= 83 + 34 = 117$

Bemerkung: Rechnen mit b-Komplement-Darstellung entspricht Rechnen modulo b^n:



Die folgende Tabelle gibt die wichtigsten C++-Datentypen für ganze Zahlen wieder. Die Darstellung der Zahlen erfolgt im Zweierkomplement, sodass sich die jeweils kleinste und größte darstellbare Zahl aus der Anzahl der Bits ergibt.

Datentyp	Mindestanzahl Bits nach C++ Standard	Anzahl Bits für g++ unter Windows XP	Anzahl Bits g++ für unter 64 Bit Linux
`short int`	16	16	16
`int`	16	32	32
`long int`	32	32	64

1.3 Darstellung reeller Zahlen

In Analogie zur b-adischen Darstellung natürlicher Zahlen besitzen auch reelle Zahlen eine b-adische Darstellung.

Beispiel 1.16
$\frac{1}{3} = 0.3333\ldots = 3 \cdot 10^{-1} + 3 \cdot 10^{-2} + 3 \cdot 10^{-3} + \ldots$

$b = 2$: $(0.1)_2 = \frac{1}{2}$, $(0.01)_2 = \frac{1}{4}$, $\ldots$
$\frac{1}{3} \cdot 4 = \frac{4}{3} = 1 + \frac{1}{3} \Rightarrow \frac{1}{3} = (0.010101\ldots)_2 = (0.\overline{01})_2$.

Was ist die Binärdarstellung von 0.3?

$$\left.\begin{array}{rcl} 2 \cdot 0.3 & = & 0 + 0.6 \\ 2 \cdot 0.6 & = & 1 + 0.2 \\ 2 \cdot 0.2 & = & 0 + 0.4 \\ 2 \cdot 0.4 & = & 0 + 0.8 \\ 2 \cdot 0.6 & = & 1 + 0.2 \\ 2 \cdot 0.2 & = & 0 + 0.4 \end{array}\right\} \Rightarrow 0.3 = (0.0\overline{1001})_2$$

Welche Zahl stellt $(0.0\overline{1})_2$ dar?

$$(0.0\overline{1})_2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = \frac{1}{2} \sum_{i=1}^{\infty} 2^{-i} = \frac{1}{2} = (0.1)_2.$$

Satz 1.17 (b-adische Darstellung reeller Zahlen)
Sei $b \in \mathbb{N}$ mit $b > 1$. Dann besitzt jede reelle Zahl $x \neq 0$ genau eine Darstellung der Form

$$x = \sigma \cdot b^N \cdot \sum_{i=1}^{\infty} x_i \cdot b^{-i}$$

mit $\sigma \in \{-1; +1\}$, $N \in \mathbb{Z}$ und $x_i \in \{0, 1, \ldots, b-1\}$, sofern man von den Zahlen x_i zusätzlich verlangt, dass $x_1 \neq 0$ und dass es zu jedem $n \in \mathbb{N}$ einen Index $i > n$ gibt mit $x_i \neq b - 1$. $(*)$

Beweis. Sei $x \in \mathbb{R}$, $x \neq 0$. Dann gilt $\sigma = -1$, falls $x < 0$, und $\sigma = +1$, falls $x > 0$. Es reicht daher, den Satz für $x > 0$ zu zeigen.

Für beliebige x_i mit $x_i \in \{0, 1, \ldots, b-1\}$, die $(*)$ erfüllen, gilt

$$\sum_{i=1}^{\infty} x_i \cdot b^{-i} < \sum_{i=1}^{\infty} (b-1) \cdot b^{-i} = (b-1) \sum_{i=1}^{\infty} b^{-i} = (b-1) \cdot \frac{1}{b-1} = 1.$$

Somit muss N so gewählt werden, dass $b^N > x$ gilt. Wegen $x_1 \neq 0$ gilt weiter

$$x \geq b^N \cdot b^{-1} = b^{N-1} \quad \Rightarrow \quad b^{N-1} \leq x < b^N.$$

$\Rightarrow N := \min\{k \in \mathbb{Z} \mid x < b^k\}$ ist eindeutig festgelegt.
Setze $a_1 := b^{-N} x$ und definiere rekursiv für $i \in \mathbb{N}$:

$$x_i := \lfloor a_i \cdot b \rfloor \quad \text{und} \quad a_{i+1} := a_i \cdot b - x_i.$$

Dann gilt $0 \leq a_i < 1$ für $i \in \mathbb{N}$, denn aus $b^{N-1} \leq x < b^N$ folgt $b^{-1} \leq a_1 < 1$ und $a_{i+1} = a_i \cdot b - \lfloor a_i \cdot b \rfloor < 1$. Zudem gilt $x_i \in \{0, 1, \ldots, b-1\}$, da $x_i = \lfloor a_i \cdot b \rfloor$.

Behauptung: Es gilt $a_1 = \sum_{i=1}^{n} x_i \cdot b^{-i} + b^{-n} \cdot a_{n+1}$ für alle $n \in \mathbb{N}_0$. $(**)$

Induktionsanfang: $n = 0$: $\quad a_1 = b^{-0} \cdot a_1 = a_1$.

Induktionsannahme: $a_1 = \sum_{i=1}^{n} x_i \cdot b^{-i} + b^{-n} \cdot a_{n+1}$

Induktionsschritt: $a_{n+2} = a_{n+1} \cdot b - x_{n+1} \Rightarrow b \cdot a_{n+1} = a_{n+2} + x_{n+1}$.

$$\begin{aligned} \Rightarrow \qquad a_1 &= \sum_{i=1}^{n} x_i \cdot b^{-i} + b^{-(n+1)} \cdot (b \cdot a_{n+1}) \\ &= \sum_{i=1}^{n} x_i \cdot b^{-i} + b^{-(n+1)} \cdot (a_{n+2} + x_{n+1}) \\ &= \sum_{i=1}^{n+1} x_i \cdot b^{-i} + b^{-(n+1)} \cdot a_{n+2}. \end{aligned}$$

Aus $a_1 - \sum_{i=1}^{n} x_i \cdot b^{-i} = b^{-n} \cdot a_{n+1}$ und $0 \leq a_{n+1} < 1$ folgt

$$0 \leq a_1 - \sum_{i=1}^{n} x_i \cdot b^{-i} < b^{-n}.$$

Für $n \to \infty$ folgt weiter $a_1 = \sum_{i=1}^{\infty} x_i \cdot b^{-i}$. Da $x = b^N \cdot a_1$ ist, folgt schließlich $x = b^N \sum_{i=1}^{\infty} x_i \cdot b^{-i}$.

Nachweis von $(*)$:
Angenommen $x_i = b - 1$ für alle $i > n$.

$$\Rightarrow \quad a_1 = \sum_{i=1}^{n} x_i \cdot b^{-i} + \sum_{i=n+1}^{\infty} (b-1) \cdot b^{-i} = \sum_{i=1}^{n} x_i \cdot b^{-i} + b^{-n}$$

Wegen $(**)$ gilt $a_{n+1} = 1$ $\lightning$ zu $0 \leq a_{n+1} < 1$.
Damit ist die Existenz bewiesen.

Beweis der Eindeutigkeit:
Die Eindeutigkeit von N wurde bereits gezeigt. Angenommen es gilt:

$$x = b^N \sum_{i=1}^{\infty} x_i \cdot b^{-i} = b^N \sum_{i=1}^{\infty} y_i \cdot b^{-i} \quad \Rightarrow \quad \sum_{i=1}^{\infty} (x_i - y_i) \cdot b^{-i} = 0.$$

Sei n der kleinste Index mit $x_n \neq y_n$. Ohne Beschränkung der Allgemeinheit sei $y_n - x_n \geq 1$.

$$\begin{aligned} \Rightarrow \qquad 0 &= b^n \sum_{i=n}^{\infty} (x_i - y_i) \cdot b^{-i} \\ &= x_n - y_n + \sum_{i=n+1}^{\infty} (x_i - y_i) \cdot b^{n-i} \\ &\leq -1 + \sum_{i=n+1}^{\infty} (b-1) \cdot b^{n-i} \\ &= -1 + 1 = 0. \end{aligned}$$

Daher muss überall Gleichheit gelten, also insbesondere $x_i - y_i = b - 1$.
$\Rightarrow x_i = b - 1$ und $y_i = 0$ für $i > n$. $\lightning$ zu $(*)$. □

Da in einem Rechner nur endlich viele Stellen einer reellen Zahl dargestellt werden können, nutzt man in Anlehnung an Satz 1.17 die folgende Darstellung:

Definition 1.18 *Eine b-adische m-stellige **normalisierte Gleitkommazahl** hat die Form*

$$x = \sigma \sum_{i=1}^{m} x_i \cdot b^{1-i} \cdot b^e$$

mit dem Vorzeichen $\sigma \in \{-1; +1\}$, der Mantisse $\sum_{i=1}^{m} x_i \cdot b^{1-i}$, der Basis b und dem Exponenten $e \in \mathbb{Z}$. Dabei wird $x_1 \neq 0$ vorausgesetzt (Normalisierung).

Beispiel 1.19 Für $b = 10$ hat die Zahl 136.58 die normalisierte 5-stellige Darstellung $136.58 = 1.3658 \cdot 10^2$.
Die Zahl $\frac{1}{3}$ besitzt für $b = 10$ keine Darstellung als m-stellige Gleitkommazahl. Die Zahl 0 besitzt keine Darstellung als normalisierte Gleitkommazahl.

Für die Darstellung im Rechner sind geeignete Werte für b, m und den zulässigen Bereich für e zu wählen. Dies definiert die Menge der **Maschinenzahlen**

$$F = F(b, m, e_{\min}, e_{\max}).$$

Im IEEE-Standard 754-1985 (IEEE - The Institute of Electrical and Electronics Engineers) wird beispielsweise ein Datentyp `double` mit $b = 2$, $m = 53$ und $e \in \{-1022, \ldots, 1023\}$ definiert. Zur Darstellung einer solchen Zahl werden 64 Bits benötigt: 1 Bit für das Vorzeichen, 52 Bit für die Mantisse sowie 11 Bit für den Exponenten.

Da wegen der Normalisierung das erste Bit immer den Wert 1 hat, kann dieses weggelassen werden. Es wird daher „hidden bit“ genannt. Für den Exponenten wird die sogenannte „Bias“-Darstellung benutzt:
Eine Konstante (Bias) wird hinzuaddiert, um alle Exponenten positiv zu machen. Im konkreten Fall wird 1023 hinzuaddiert.

Die Abspeicherung in Bits für die normalisierte Gleitkommadarstellung sieht wie folgt aus:

1 Bit	11 Bit	52 Bit
σ	$e + 1023$	$x_2\ x_3\ \ldots\ x_{53}$

Die „freien“ Exponenten -1023 und 1024 ($\hat{=}$ 0 und 2047 in Bias-Darstellung) werden benutzt, um ± 0 und $\pm\infty$ zu kodieren sowie für denormalisierte Zahlen und **NaN**s (**N**ot **a** **N**umber).
Die größte so darstellbare Zahl ist

$$\begin{aligned} F_{\max}(2, 53, -1022, 1023) &= 1.\underbrace{1\ldots1}_{52} \cdot 2^{1023} = (2 - 2^{-52}) \cdot 2^{1023} = 2^{1024} - 2^{971} \\ &\approx 1.7976931 \cdot 10^{308}. \end{aligned}$$

Die kleinste positive Maschinenzahl ist

$$F_{\min}(2, 53, -1022, 1023) = 1.0 \cdot 2^{-1022} \approx 2.2250739 \cdot 10^{-308}.$$

Da es nur endlich viele Maschinenzahlen gibt, muss man sich mit gerundeten Darstellungen von reellen Zahlen zufrieden geben:

Definition 1.20 *Die Abbildung* $\mathrm{rd} : \mathbb{R} \to F(b, m, e_{min}, e_{max})$ *ordnet jeder reellen Zahl* x *eine Maschinenzahl* $\mathrm{rd}(x)$ *zu, sodass gilt:*

$$|x - \mathrm{rd}(x)| = \min_{a \in F} |x - a|.$$

Beachte: rd ist nicht eindeutig festgelegt, da zum Beispiel in $F(10, 2, 0, 2)$ die Zahl 12.5 auf 12 oder 13 gerundet werden kann. Üblicherweise wird x zur betragsmäßig größeren Zahl gerundet, falls x genau zwischen zwei Maschinenzahlen liegt.

IEEE 754-1985 fordert: x wird so gerundet, dass die letzte Stelle gerade wird, also zum Beispiel $\mathrm{rd}(12.5) = 12$, $\mathrm{rd}(13.5) = 14$. Im Zweifelsfall wird genauso oft auf- wie abgerundet, um Effekte wie $((12 + 0.5) - 0.5) + 0.5 - \ldots \to \infty$ zu vermeiden.

Definition 1.21 *Es seien* $x, \widetilde{x} \in \mathbb{R}$*, wobei* $\widetilde{x}$ *eine Näherung von* x *sei. Dann heißt* $x - \widetilde{x}$ *der* ***absolute Fehler*** *und* $\frac{x - \widetilde{x}}{x}$ *der* ***relative Fehler****, falls* $x \neq 0$*.*

Definition 1.22 *Die* ***Maschinengenauigkeit*** **eps** *ist der betragsmäßig maximale relative Fehler, der auftritt, wenn* x *durch* $\mathrm{rd}(x)$ *ersetzt wird, für* $F_{\min} \leq |x| \leq F_{\max}$*.*

Satz 1.23 *Für Maschinenzahlen $F = F(b, m, e_{min}, e_{max})$ ist die Maschinengenauigkeit* $\text{eps} = \frac{1}{2} \cdot b^{1-m}$.

Beweis. Sei $x = b^e \cdot \sum_{i=1}^{\infty} x_i \cdot b^{1-i}$ mit $x_1 \neq 0$ und $F_{\min} < x < F_{\max}$. Dann sind

$$x' = b^e \cdot \sum_{i=1}^{m} x_i \cdot b^{1-i} \quad \text{und} \quad x'' = b^e \cdot (\sum_{i=1}^{m} x_i \cdot b^{1-i} + b^{1-m})$$

in F und es gilt $x' \leq x \leq x''$.
Nun gilt $|x - \text{rd}(x)| \leq \frac{1}{2}|x' - x''| = \frac{1}{2} \cdot b^e \cdot b^{1-m}$. Wegen $x_1 > 0$ gilt $x \geq b^e$.

$$\Rightarrow \quad \left| \frac{x - \text{rd}(x)}{x} \right| \leq \frac{1}{2} \cdot b^{1-m}.$$

□

Eine Maschinenzahl x hat s **signifikante Stellen**, falls für jede Zahl y mit $\text{rd}(y) = x$ gilt:

$$|x - y| \leq \frac{1}{2} \cdot b^{1-s}.$$

Beispiel 1.24 Für den durch IEEE 754-1985 definierten Typ `double` gilt:

$$\text{eps} = \frac{1}{2} \cdot 2^{-52} \approx 1.11 \cdot 10^{-16},$$

d.h. man hat 16 signifikante Stellen in der Dezimaldarstellung.

Bemerkung 1.25 Es gilt $\text{rd}(x) = x \cdot (1 + \varepsilon)$ mit $|\varepsilon| < \text{eps}$, da

$$\left| \frac{x - \text{rd}(x)}{x} \right| = \left| \frac{x - x \cdot (1 + \varepsilon)}{x} \right| = |\varepsilon| \leq \text{eps}\,.$$

Literaturverzeichnis

[1] Oberschlep, Walter und Vossen, Gottfried: *Rechneraufbau und Rechnerstrukturen*, 10. Auflage, Oldenbourg Verlag, 2006

[2] Hopcroft, John und Ullman, Jeffrey: *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979

[3] Hämmerlin, Günther und Hoffmann, Karl-Heinz: *Numerische Mathematik*. Springer, 1989

[4] Knuth, Donald: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd Edition, Addison Wesley,1997

[5] Goldberg, David: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, Volume 23, Number 1, 1991

[6] IEEE 754-1985

2 Fehleranalyse

2.1 Rechnerarithmetik

Wegen der nur endlich vielen Maschinenzahlen entstehen Fehler nicht nur bei der Eingabe von Daten (zum Beispiel ist 0.1 nicht exakt mit endlich vielen Stellen binär darstellbar), sondern auch bei den elementaren Rechenoperationen $+$, $-$, $\cdot$ und $/$.

Beispiel 2.1 Sei $F = F(10, 3, -5, 5)$.
Wähle $x = 4.518 \cdot 10^1 = 45.18$ und $y = 1.875 \cdot 10^0 = 1.875$. Dann sind

$$\begin{array}{lcllcl} x+y & = & 47.055 & = & 4.7055 \cdot 10^1 & \notin F \\ x-y & = & 43.305 & = & 4.3305 \cdot 10^1 & \notin F \\ x \cdot y & = & 84.7125 & = & 8.47125 \cdot 10^1 & \notin F \\ x/y & = & 24.096 & = & 2.4096 \cdot 10^1 & \notin F. \end{array}$$

Sei $\circ$ eine der vier Operationen $\{+, -, \cdot, /\}$. Wie Beispiel 2.1 verdeutlicht, gilt für $x, y \in F$ nicht notwendigerweise $x \circ y \in F$. Stattdessen wird auf dem Rechner eine Ersatzoperation $\circledcirc$ ausgeführt, sodass $x \circledcirc y \in F$ gilt. Wir nehmen im Folgenden an, dass

Annahme 2.2 $$x \circledcirc y = \mathrm{rd}(x \circ y).$$

Dies wird zum Beispiel vom IEEE-Standard 754 gefordert. Zudem wird dies dort auch für die Wurzelfunktion, nicht aber für zum Beispiel $\sin(x)$, $\exp(x)$ etc. vorausgesetzt. Das heißt, die Annahme ist, dass $x \circ y$ zunächst exakt berechnet und dann auf die nächste Maschinenzahl gerundet wird.
Beachte: Zur Berechnung von $x \circledcirc y$ muss $x \circ y$ nicht notwendigerweise berechnet werden.

Gemäß Annahme 2.2 und Bemerkung 1.25 gilt für den relativen Fehler

$$\left| \frac{x \circledcirc y - x \circ y}{x \circ y} \right| = \left| \frac{\mathrm{rd}(x \circ y) - x \circ y}{x \circ y} \right| \leq \mathrm{eps}\,.$$

Das Kommutativgesetz für Addition und Multiplikation gilt auch in der Rechnerarithmetik, das Distributiv- und Assoziativgesetz gelten jedoch im Allgemeinen nicht.

Beispiel 2.3 Sei $F = F(10, 1, -5, 5)$. Wähle $x = 4.1 \cdot 10^0$, $y = 8.2 \cdot 10^{-1}$ und $z = 1.4 \cdot 10^{-1}$. Dann sind

$$\begin{array}{lcccl} (x \oplus y) \oplus z & = & 4.9 \oplus 0.14 & = & 5.0 \\ x \oplus (y \oplus z) & = & 4.1 \oplus 0.96 & = & 5.1 \end{array}$$

$$\begin{array}{lcl} x \odot (y \oplus z) & = & 4.1 \oplus 0.96 = 3.9 \\ (x \odot y) \oplus (x \odot z) & = & 3.4 \oplus 0.57 = 4.0 \end{array}$$

$\Rightarrow$ Mathematisch äquivalente Ausdrücke können bei Ausführung in Rechnerarithmetik zu wesentlich unterschiedlichen Ergebnissen führen, auch wenn der Input als Maschinenzahlen darstellbar ist.

2.2 Fehlerfortpflanzung

Bei der Fehlerfortpflanzung untersucht man, wie sich Fehler in den Eingabedaten im Laufe der Berechnungen fortpflanzen. Man untersucht welcher Fehler bei **exakter** Auswertung der fehlerhaften Eingabedaten auftritt.
Das folgende Lemma gibt Auskunft darüber wie sich Fehler bei den Grundrechenarten fortpflanzen.

Lemma 2.4 *Es seien $\widetilde{x}$ und $\widetilde{y}$ die mit relativen Fehlern ε_x bzw. ε_y behafteten Werte für x und y. Dann gilt für den relativen Fehler $\varepsilon_\circ := \frac{x \circ y - (\widetilde{x} \circ \widetilde{y})}{x \circ y}$ mit $\circ \in \{+, -, \cdot, /\}$*

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

Beweis. Es gilt: $\varepsilon_x = \frac{x - \widetilde{x}}{x} \quad \Rightarrow \quad x \cdot \varepsilon_x = x - \widetilde{x} \quad \Rightarrow \quad \widetilde{x} = x \cdot (1 - \varepsilon_x)$. Analog folgt $\widetilde{y} = y \cdot (1 - \varepsilon_y)$.

$$\varepsilon_+ = \frac{x + y - (\widetilde{x} + \widetilde{y})}{x+y} = \frac{x + y - (1-\varepsilon_x) \cdot x - (1-\varepsilon_y) \cdot y}{x+y} = \frac{\varepsilon_x \cdot x}{x+y} + \frac{\varepsilon_y \cdot y}{x+y},$$

ε_- analog,

$$\begin{aligned} \varepsilon_\cdot &= \frac{x \cdot y - \widetilde{x} \cdot \widetilde{y}}{x \cdot y} = \frac{x \cdot y - (1-\varepsilon_x) \cdot x \cdot (1-\varepsilon_y) \cdot y}{x \cdot y} \\ &= 1 - (1-\varepsilon_x) \cdot (1-\varepsilon_y) = \varepsilon_x + \varepsilon_y - \varepsilon_x \cdot \varepsilon_y, \end{aligned}$$

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

□

Bei $\cdot$ und / addieren bzw. subtrahieren sich die relativen Fehler im Wesentlichen. Bei $+$ und $-$ kann der relative Fehler jedoch sehr verstärkt werden, falls $|x \pm y|$ sehr klein in Relation zu $|x|$ und $|y|$ ist. Dieser Effekt heißt **Auslöschung** und es gilt ihn so weit wie möglich zu vermeiden.

Beispiel 2.5 Betrachte die quadratische Gleichung

$$x^2 + px + q = 0.$$

Dann sind $x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$ die beiden Nullstellen. Berechne für $p = 200$, $q = 9$ und rechne mit drei signifikanten Stellen.

$$x_{1,2} = -100 \pm \sqrt{100^2 - 9} = -100 \pm \sqrt{9991} = -100 \pm 99.95498\ldots \approx -100 \pm 100.$$

$\Rightarrow \quad x_1 = 0$ und $x_2 = -200$.
Richtig wäre $x_1 = -0.0450$ und $x_2 = -200$, wenn das Ergebnis mit drei signifikanten Stellen angegeben werden soll.
Gemäß $(x - x_1) \cdot (x - x_2) = x^2 - (x_1 + x_2) \cdot x + x_1 \cdot x_2 = x^2 + p \cdot x + q$ gilt $x_1 = \frac{q}{x_2}$.
Besser wäre es daher gewesen, zunächst x_2 und dann $x_1 = \frac{q}{x_2} = \frac{9}{-200} = -0,045$ zu berechnen, um die Auslöschung bei der Berechnung von x_1 zu vermeiden.

Bei der Fehleranalyse unterscheidet man drei Arten von Fehlern:

1. **Datenfehler:** Um ein Rechenverfahren zu starten, benötigt man zunächst Eingabedaten. Diese sind im Allgemeinen ungenau (z.B. Messdaten) oder lassen sich nicht genau im Rechner darstellen. Anstelle von einem Eingabewert x hat man also einen gestörten Eingabewert $\widetilde{x}$. Selbst bei exakter Rechnung kann man also kein exaktes Ergebnis erwarten.
2. **Verfahrensfehler:** Viele Rechenverfahren liefern selbst bei exakter Rechnung nach endlich vielen Schritten keine exakten Lösungen sondern nähern sich einer solchen nur beliebig genau an. Der dadurch verursachte Fehler heißt Verfahrensfehler.
3. **Rundungsfehler:** Wegen der begrenzten Rechengenauigkeit sind alle Zwischenergebnisse gerundet. Dies kann zu extremen Verfälschungen des Endergebnisses führen (siehe Beispiel 0.2).

Definition 2.6 *Bei der **Vorwärtsanalyse** untersucht man, wie sich der relative Fehler im Laufe einer Rechnung akkumuliert.*
*Bei der **Rückwärtsanalyse** wird jeder Rechenschritt als exakter Rechenschritt auf gestörten Daten interpretiert und es wird abgeschätzt, wie groß die Eingangsdatenstörung ist.*

Beispiel 2.7 Betrachte die Addition zweier Zahlen:
Vorwärtsanalyse: $x \oplus y = \mathrm{rd}(x+y) = (x+y) \cdot (1+\varepsilon)$ mit $|\varepsilon| \leq \mathrm{eps}$ nach Bemerkung 1.25.
Rückwärtsanalyse: $x \oplus y = x \cdot (1+\varepsilon) + y \cdot (1+\varepsilon) = \widetilde{x} + \widetilde{y}$ mit $|\varepsilon| \leq \mathrm{eps}$ und $\widetilde{x} = x \cdot (1+\varepsilon)$ und $\widetilde{y} = y \cdot (1+\varepsilon)$.

Beispiel 2.8 Berechne $\sum_{i=1}^{n} a_i \cdot b_i$ für $a_i, b_i \in \mathbb{R}$.
Sei $t_i := a_i \odot b_i = a_i \cdot b_i \cdot (1+\varepsilon_i)$ mit $|\varepsilon_i| \leq \text{eps}$, $i = 1, \ldots, n$.
$s_i := s_{i-1} \oplus t_i = (s_{i-1} + t_i) \cdot (1+\gamma_i)$ mit $|\gamma_i| \leq \text{eps}$ für $i = 2, \ldots, n$ und $s_1 := t_1$.
Dann gilt:

$$\begin{aligned}
s_1 &= a_1 \cdot b_1 \cdot (1+\varepsilon_1), \\
s_2 &= (a_1 \cdot b_1 \cdot (1+\varepsilon_1) + a_2 \cdot b_2 \cdot (1+\varepsilon_2)) \cdot (1+\gamma_2) \\
&= a_1 \cdot b_1 \cdot (1+\varepsilon_1) \cdot (1+\gamma_2) + a_2 \cdot b_2 \cdot (1+\varepsilon_2) \cdot (1+\gamma_2), \\
s_3 &= a_1 \cdot b_1 \cdot (1+\varepsilon_1) \cdot (1+\gamma_2) \cdot (1+\gamma_3) + a_2 \cdot b_2 \cdot (1+\varepsilon_2) \cdot (1+\gamma_2) \cdot (1+\gamma_3) \\
&\quad + a_3 \cdot b_3 \cdot (1+\varepsilon_3) \cdot (1+\gamma_3), \\
&\vdots \\
s_n &= \sum_{i=1}^{n} a_i \cdot b_i \cdot (1+\varepsilon_i) \cdot \prod_{j=i}^{n} (1+\gamma_j),
\end{aligned}$$

wobei $\gamma_1 := 0$ ist.
Bei der Rückwärtsanalyse werden die Fehler als gestörte Eingabedaten interpretiert: $\widetilde{b_i} = b_i \cdot (1+\delta_i)$.
$\Rightarrow \quad 1+\delta_i = (1+\varepsilon_i) \cdot \prod_{j=i}^{n} (1+\gamma_i)$.
$\Rightarrow \quad (1-\text{eps})^{n-i+2} \leq 1+\delta_i \leq (1+\text{eps})^{n-i+2}$.
Nun gilt mit der Bernoulli'schen Ungleichung ($(1-x)^n \geq 1 - n \cdot x$):

$$\begin{aligned}
\delta_i &\geq (1-\text{eps})^{n-i+2} - 1 \\
&\geq 1 - (n-i+2) \cdot \text{eps} - 1 \\
&= -(n-i+2) \cdot \text{eps}
\end{aligned}$$

und

$$\begin{aligned}
\delta_i &\leq (1+\text{eps})^{n-i+2} - 1 \\
&= \frac{(1+\text{eps})^{n-i+2} \cdot (1-\text{eps})^{n-i+2}}{(1-\text{eps})^{n-i+2}} - 1 \\
&= \frac{(1-\text{eps}^2)^{n-i+2}}{(1-\text{eps})^{n-i+2}} - 1 \\
&\leq \frac{1}{1-(n-i+2) \cdot \text{eps}} - 1 \\
&= \frac{(n-i+2) \cdot \text{eps}}{1-(n-i+2) \cdot \text{eps}}.
\end{aligned}$$

$$\Rightarrow \quad |\delta_i| \leq \frac{(n-i+2) \cdot \text{eps}}{1-(n-i+2) \cdot \text{eps}} \quad \text{und} \quad s_n = \sum_{i=1}^{n} a_i \cdot b_i \cdot (1+\delta_i).$$

2.3 Kondition und Stabilität

Es sei $f : \mathbb{R} \to \mathbb{R}$ eine stetig differenzierbare Funktion. Wir wollen den relativen Fehler berechnen, der bei der Auswertung von f an der mit relativem Fehler ε behafteten

Stelle x auftritt:

$$\frac{f(x+\varepsilon)-f(x)}{f(x)} = \frac{f(x+\varepsilon)-f(x)}{\varepsilon \cdot x} \cdot \frac{\varepsilon \cdot x}{f(x)} \approx f'(x) \cdot \frac{\varepsilon \cdot x}{f(x)} = \varepsilon \cdot \frac{x \cdot f'(x)}{f(x)}.$$

Definition 2.9 *Es sei $f : \mathbb{R} \to \mathbb{R}$ eine stetig differenzierbare Funktion. Dann ist die* ***Konditionszahl*** *von f an der Stelle x definiert als $\frac{x \cdot f'(x)}{f(x)}$. Falls die Konditionszahl betragsmäßig groß ist, so heißt das Problem* ***schlecht konditioniert****, andernfalls* ***gut konditioniert****.*

Beispiel 2.10

- Sei $f(x) = x + a$. Dann ist die Konditionszahl $\frac{x}{x+a}$. Das Problem ist schlecht konditioniert, falls $|x + a|$ sehr klein im Verhältnis zu x ist (Auslöschung).
- Sei $f(x) = a \cdot x$. Dann ist die Konditionszahl $\frac{x \cdot a}{x \cdot a} = 1$. Folglich ist das Problem immer gut konditioniert.
- Sei $f(x) = \sqrt{x}$. Dann ist die Konditionszahl $\frac{x \cdot \frac{1}{2} \cdot \frac{1}{\sqrt{x}}}{\sqrt{x}} = \frac{1}{2}$. Somit ist auch dieses Problem stets gut konditioniert.

Definition 2.11 *Ein Algorithmus A heißt* ***numerisch stabiler*** *als ein Algorithmus B zur Berechnung von $f(x)$, falls der Gesamteinfluss der Rundungsfehler bei A geringer als bei B ist.*

Es gilt:

- **Kondition:** Eigenschaft des Problems
- **Stabilität:** Eigenschaft des Algorithmus

Ist ein Problem schlecht konditioniert, so kann kein Verfahren verhindern, dass sich Eingabefehler vergrößern.

3 Dreitermrekursion

Definition 3.1 *Eine* ***Dreitermrekursion*** *ist eine Rekursion der Form*

$$x_{k+1} + a_k \cdot x_k + b_k \cdot x_{k-1} + c_k = 0, \qquad k = 1, 2, \ldots$$

für Koeffizienten $a_k, b_k, c_k \in \mathbb{R}$. Gilt $c_k = 0$ für alle k, so heißt die Dreitermrekursion ***homogen****.*

Bei angegebenen Anfangswerten x_0 und x_1 lassen sich alle weiteren x_k aus der Rekursionsgleichung bestimmen.
In Beispiel 0.2 haben wir die Rekursionsgleichung $J_{k+1} = \frac{2k}{x} \cdot J_k - J_{k-1}$ betrachtet. In obige Form gebracht, schreiben wir diese als $x_{k+1} - \frac{2k}{x} \cdot x_k + x_{k-1} = 0$, das heißt, es gilt $a_k = -\frac{2k}{x}$, $b_k = 1$ und $c_k = 0$.
Wir haben bereits gesehen, dass die Auswertung der Rekursionsgleichung sehr instabil ist, und wollen nun untersuchen, woran das liegt. Dazu betrachten wir die homogene Dreitermrekursion

$$x_{k+1} + a \cdot x_k + b \cdot x_{k-1} = 0, \qquad k = 1, 2, \ldots$$

mit konstanten Koeffizienten $a, b \in \mathbb{R}$.
Wir gehen davon aus, dass x_0 fest gegeben ist und x_1 variiert, das heißt, wir betrachten die Funktionen $f_k(x_1) = x_k$ für $k = 2, 3, \ldots$ und berechnen die Kondition von f_k an der Stelle x_0 in Abhängigkeit von a, b, x_0 und k.

Beispiel 3.2 Betrachte die Dreitermrekursion

$$x_{k+1} - 2 \cdot x_k - 2 \cdot x_{k-1} = 0$$

mit $x_0 = \sqrt{3}$, $x_1 = \sqrt{3} - 3$. Welchen Wert hat x_{100}?

Algorithmus 3.3

```
#include <iostream>
#include <cmath>

using namespace std;

main ()
{   double x[101];

    x[0] = sqrt(3);
    x[1] = sqrt(3)-3;

    for (int k=1; k<=99; ++k)   x[k+1] = 2*x[k] + 2*x[k-1];

    cout << x[100] << endl;
}
```

Das Programm liefert das Ergebnis $-2.23547 \cdot 10^{27}$. Der folgende Satz hilft uns, das richtige Ergebnis zu ermitteln:

Satz 3.4 *Es seien $\lambda_1 \neq \lambda_2$ die Nullstellen des Polynoms $\lambda^2 + a \cdot \lambda + b = 0$. Dann ist die Lösung der Dreitermrekursion*

$$x_{k+1} + a \cdot x_k + b \cdot x_{k-1} = 0, \qquad k = 1, 2, \ldots$$

gegeben durch

$$x_k = \alpha \cdot \lambda_1^k + \beta \cdot \lambda_2^k,$$

wobei sich α und β durch die Gleichungen $\alpha + \beta = x_0$ und $\alpha \cdot \lambda_1 + \beta \cdot \lambda_2 = x_1$ definieren.

Beweis. Wir beweisen die Aussage durch Induktion über k:

Induktionsanfang: $k = 0$: $x_0 = \alpha \cdot \lambda_1^0 + \beta\lambda_2^0 = \alpha + \beta$
$k = 1$: $x_1 = \alpha \cdot \lambda_1^1 + \beta\lambda_2^1$

Induktionsannahme: Es gelte $x_k = \alpha \cdot \lambda_1^k + \beta \cdot \lambda_2^k$

Induktionsschritt:

$$\begin{aligned} 0 &= x_{k+1} + a \cdot (\alpha \cdot \lambda_1^k + \beta \cdot \lambda_2^k) + b \cdot (\alpha \cdot \lambda_1^{k-1} + \beta \cdot \lambda_2^{k-1}) \\ &= x_{k+1} + \alpha \cdot \lambda_1^{k-1} \cdot (a \cdot \lambda_1 + b) + \beta \cdot \lambda_2^{k-1} \cdot (a \cdot \lambda_2 + b) \\ &= x_{k+1} + \alpha \cdot \lambda_1^{k-1} \cdot (-\lambda_1^2) + \beta \cdot \lambda_2^{k-1} \cdot (-\lambda_2^2) \\ &= x_{k+1} - \alpha \cdot \lambda_1^{k+1} - \beta \cdot \lambda_2^{k+1} \end{aligned}$$

$\Rightarrow \quad x_{k+1} = \alpha \cdot \lambda_1^{k+1} + \beta \cdot \lambda_2^{k+1}.$ □

Bemerkung 3.5 Aus $\alpha + \beta = x_0$ und $\alpha \cdot \lambda_1 + \beta\lambda_2 = x_1$ berechnen sich α und β zu

$$\beta = \frac{x_1 - \lambda_1 \cdot x_0}{\lambda_2 - \lambda_1} \quad \text{und} \quad \alpha = -\frac{x_1 - \lambda_2 \cdot x_0}{\lambda_2 - \lambda_1}.$$

Beispiel 3.6 Wie in Beispiel 3.2 betrachten wir die Dreitermrekursion

$$x_{k+1} - 2 \cdot x_k - 2 \cdot x_{k-1} = 0$$

mit $x_0 = \sqrt{3}$, $x_1 = \sqrt{3} - 3$. Es gilt also $a = b = -2$. Das Polynom $\lambda^2 - 2 \cdot \lambda - 2$ hat die Nullstellen $\lambda_1 = 1 + \sqrt{3}$ und $\lambda_2 = 1 - \sqrt{3}$. Wir erhalten somit:

$$\alpha = -\frac{\sqrt{3} - 3 - (1 - \sqrt{3}) \cdot \sqrt{3}}{1 - \sqrt{3} - (1 + \sqrt{3})} = 0$$

und

$$\beta = \frac{\sqrt{3} - 3 - (1 + \sqrt{3}) \cdot \sqrt{3}}{1 - \sqrt{3} - (1 + \sqrt{3})} = \frac{\sqrt{3} - 3 - \sqrt{3} - 3}{1 - \sqrt{3} - 1 - \sqrt{3}} = \frac{-6}{-2 \cdot \sqrt{3}} = \sqrt{3}.$$

$\Rightarrow \quad x_k = \sqrt{3} \cdot (1 - \sqrt{3})^k$.
$\Rightarrow \quad x_{100} = 4.9281\ldots \cdot 10^{-14}$ statt $-2.23547 \cdot 10^{27}$.

Das folgende Programm berechnet x_{100} gemäß Satz 3.4.

Algorithmus 3.7

```
#include <iostream>
#include <cmath>

using namespace std;

main ()
{   double a = -2, b = -2, x[101], alpha, beta, lambda1, lambda2;

    x[0] = sqrt(3);
    x[1] = sqrt(3)-3;

    lambda1 = -a/2 + sqrt(a*a/4 - b);
    lambda2 = -a/2 - sqrt(a*a/4 - b);
    alpha   = (x[1] -lambda2*x[0]) / (lambda1 - lambda2);
    beta    = x[0] -alpha;

    for (int k=2; k<=100; ++k)
        x[k] = alpha * pow(lambda1, k) + beta * pow(lambda2, k);

    cout << x[100] << endl;
}
```

Das Ergebnis ist $-4.47054 \cdot 10^{27}$, es ist also immer noch falsch.
Was ist das Problem?
Es gilt $\alpha = 0$, aber das Programm berechnet α zu $-1.00342 \cdot 10^{-16}$. Wegen $\lambda_1 = 2.73205\ldots$ ergibt sich damit das falsche Ergebnis.
Wir untersuchen nun die Kondition der Dreitermrekursion:

Satz 3.8 *Gegeben sei die Dreitermrekursion* $x_{k+1}+a\cdot x_k+b\cdot x_{k-1}=0$ *für* $k=1,2,\ldots$ *mit* $a,\ b\in\mathbb{R}$ *und gegebenem* $x_0\in\mathbb{R}$. *Die Funktion* $f_k(x_1)$ *gebe den Wert von* x_k *in Abhängigkeit von* x_1 *an. Damit gilt*

$$\lim_{k\to\infty}\left(x_1\cdot\frac{f_k'(x_1)}{f_k(x_1)}\right)=\begin{cases}\frac{x_1}{x_1-\lambda_1\cdot x_0} & \text{falls } x_1\neq\lambda_1\cdot x_0\\ \pm\infty & \text{sonst,}\end{cases}$$

wobei λ_1 *und* λ_2 *die Nullstellen von* $\lambda^2+a\cdot\lambda+b$ *mit* $|\lambda_2|>|\lambda_1|$ *sind.*

Beweis. Nach Satz 3.4 und Bemerkung 3.5 gilt:

$$\begin{aligned}
f_k(x_1) &= \alpha\cdot\lambda_1^k+\beta\cdot\lambda_2^k\\
&= \frac{x_1-\lambda_2\cdot x_0}{\lambda_1-\lambda_2}\cdot\lambda_1^k+\frac{x_1-\lambda_1\cdot x_0}{\lambda_2-\lambda_1}\cdot\lambda_2^k\\
&= \frac{1}{\lambda_1-\lambda_2}\cdot\left(\lambda_1^k\cdot(x_1-\lambda_2\cdot x_0)+\lambda_2^k\cdot(\lambda_1\cdot x_0-x_1)\right)\\
f_k'(x_1) &= \frac{1}{\lambda_1-\lambda_2}\cdot(\lambda_1^k-\lambda_2^k)\\
x_1\cdot\frac{f_k'(x_1)}{f_k(x_1)} &= x_1\cdot\frac{\lambda_1^k-\lambda_2^k}{\lambda_1^k\cdot(x_1-\lambda_2\cdot x_0)+\lambda_2^k\cdot(\lambda_1\cdot x_0-x_1)}\\
&= x_1\cdot\frac{\lambda_1^k-\lambda_2^k}{(\lambda_1^k-\lambda_2^k)\cdot x_1+x_0\cdot(\lambda_1\cdot\lambda_2^k-\lambda_1^k\cdot\lambda_2)}\\
&= \frac{x_1}{x_1+x_0\cdot\frac{\lambda_2^k\cdot\lambda_1-\lambda_1^k\cdot\lambda_2}{\lambda_1^k-\lambda_2^k}}\\
&= \frac{x_1}{x_1-\lambda_1\cdot x_0\cdot\frac{\lambda_2^k-\lambda_1^{k-1}\cdot\lambda_2}{\lambda_2^k-\lambda_1^k}}.
\end{aligned}$$

Nun gilt:

$$\frac{\lambda_2^k-\lambda_2\lambda_1^{k-1}}{\lambda_2^k-\lambda_1^k}=\frac{1-\left(\frac{\lambda_1}{\lambda_2}\right)^{k-1}}{1-\left(\frac{\lambda_1}{\lambda_2}\right)^k}.$$

Für $|\lambda_2|>|\lambda_1|$ gilt $\lim_{k\to\infty}\left(\frac{1-\left(\frac{\lambda_1}{\lambda_2}\right)^{k-1}}{1-\left(\frac{\lambda_1}{\lambda_2}\right)^k}\right)=1.$

$$\Rightarrow\quad\lim_{k\to\infty}\left(x_1\cdot\frac{f_k'(x_1)}{f_k(x_1)}\right)=\begin{cases}\frac{x_1}{x_1-\lambda_1\cdot x_0} & \text{falls } x_1\neq\lambda_1\cdot x_0\\ \pm\infty & \text{sonst.}\end{cases}$$

□

Beispiel 3.9 Wir betrachten nochmals die Dreitermrekursion aus den Beispielen 3.2 und 3.6:

$$x_{k+1}-2\cdot x_k-2\cdot x_{k-1}=0$$

mit $x_0 = \sqrt{3}$, $x_1 = \sqrt{3} - 3$. Es gilt $\lambda_2 = 1 + \sqrt{3}$, $\lambda_1 = 1 - \sqrt{3}$, $|\lambda_2| > |\lambda_1|$. Wegen $x_1 = \sqrt{3} - 3 = \sqrt{3} \cdot (1 - \sqrt{3}) = x_0 \cdot \lambda_1$ ist die Kondition dieses Problems für $k \to \infty$ unbeschränkt.

Bemerkung 3.10 Falls $x_1 = \lambda_1 \cdot x_0$, so erhält man

$$\beta = 0 \quad \text{und} \quad \alpha = \frac{x_1 - \lambda_2 \cdot x_0}{\lambda_1 - \lambda_2} = \frac{\lambda_1 \cdot x_0 - \lambda_2 \cdot x_0}{\lambda_1 - \lambda_2} = x_0.$$

Damit ist die Lösung der Dreitermrekursion in diesem Fall $x_k = \alpha \cdot \lambda_1^k + \beta \cdot \lambda_2^k = x_0 \cdot \lambda_1^k$. Wegen $|\lambda_2| > |\lambda_1|$ gilt $|\lambda_2|^k \gg |\lambda_1|^k$. Falls also $\widetilde{x}_1$ Näherung von $x_1 = \lambda_1 \cdot x_0$ ist, so ist $\beta \neq 0$ und $\beta \cdot \lambda_2^k$ dominiert die Lösung und führt daher zu einem beliebig falschen Ergebnis.

4 Der euklidische Algorithmus

Der euklidische Algorithmus ($\sim$ 300 vor Christus, Buch VII der „Elemente“) erlaubt das effiziente Berechnen von kgV und ggT zweier Zahlen.

Beispiel 4.1

$$\frac{11}{6} - \frac{8}{15} = \frac{165}{90} - \frac{48}{90} = \frac{117}{90} = \frac{13}{10},$$

beziehungsweise

$$\frac{11}{6} - \frac{8}{15} = \frac{55}{30} - \frac{16}{30} = \frac{39}{30} = \frac{13}{10}.$$

$\Rightarrow$ Beim Bruchrechnen bestimmt man gekürzte Darstellungen mittels ggT- und kgV-Berechnungen. Ähnliches gilt für den Vergleich zweier Brüche, zum Beispiel

$$\frac{861}{1107} = \frac{1029}{1323} = \frac{7}{9}.$$

Definition 4.2 *Es seien $a, b \in \mathbb{Z}$. Dann ist* ***a Teiler von b*** *(Schreibweise: $a|b$), falls es ein $k \in \mathbb{Z}$ gibt mit $b = k \cdot a$. Mit* $\mathrm{ggT}(a, b)$ *bezeichnen wir die größte Zahl $x \in \mathbb{N}_0$ mit $x|a$ und $x|b$, wobei* $\mathrm{ggT}(0, 0)$ *als 0 definiert ist.*

Algorithmus 4.3

```
#include <iostream>

using namespace std;

main ()
{   int a, b, ggT;

    cin >> a >> b;

    for (ggT = min(a,b); (a % ggT != 0)||(b % ggT != 0); --ggT);

    cout << "ggT = " << ggT << endl;
}
```

Algorithmus 4.3 kann sehr lange brauchen. Berechnet man zum Beispiel $\mathrm{ggT}(10^{12}, 10^{12}-1)$, so wird die `for`-Schleife $10^{12} - 1$ mal durchlaufen.
Geht das auch schneller?

Lemma 4.4 *Sind $a, b \in \mathbb{N}_0$ mit $a \geq b$, so gilt* $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - b, b)$.

Beweis. Für Zahlen $r, s, t \in \mathbb{Z}$ gilt:

$$r|s \quad \text{und} \quad r|t \quad \text{folgt} \quad r|s - t, \quad (*)$$

denn $s = \alpha \cdot r$ und $t = \beta \cdot r$ mit $\alpha, \beta \in \mathbb{Z}$.
$\Rightarrow$ $s - t = \alpha \cdot r - \beta \cdot r = (\alpha - \beta) \cdot r$ und $\alpha - \beta \in \mathbb{Z}$.
Nach Definition gilt: $\mathrm{ggT}(a, b)|a$ und $\mathrm{ggT}(a, b)|b$. Mit $(*)$ folgt $\mathrm{ggT}(a, b)|a - b$.
$\Rightarrow$ $\mathrm{ggT}(a, b)$ ist Teiler von b und $a - b$ und somit $\mathrm{ggT}(a, b) \leq \mathrm{ggT}(a - b, b)$. Nach Definition gilt: $\mathrm{ggT}(a - b, b)|a - b$ und $\mathrm{ggT}(a - b, b)|b$
$\Rightarrow$ wegen $(*)$ und $\mathrm{ggT}(a - b, b)| - b$ gilt $\mathrm{ggT}(a - b, b)$ teilt $a - b - (-b) = a$.
$\Rightarrow$ $\mathrm{ggT}(a - b, b)|a$ und $\mathrm{ggT}(a - b, b)|b$ und somit $\mathrm{ggT}(a - b, b) \leq \mathrm{ggT}(a, b)$.
$\Rightarrow$ $\mathrm{ggT}(a - b, b) = \mathrm{ggT}(a, b)$. □

Beispiel 4.5 $\mathrm{ggT}(10^{12}, 10^{12} - 1) = \mathrm{ggT}(1, 10^{12} - 1) = 1$. Die Anwendung von Lemma 4.4 ist nicht notwendigerweise schneller als Algorithmus 4.3.

Korollar 4.6 *Sind $a, b \in \mathbb{N}_0$ mit $a \geq b$, so gilt* $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \bmod b, b)$.

Beweis. Es gilt: $a \bmod b = a - \lfloor a/b \rfloor \cdot b$.
Induktion über $\lfloor a/b \rfloor$:
Induktionsanfang: $\lfloor a/b \rfloor = 1$: $\Rightarrow$ $a \bmod b = a - b$
Induktionsschritt: Gelte $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - k \cdot b, b)$ mit $k < \lfloor a/b \rfloor$.
$\Rightarrow$ $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - k \cdot b - b, b) = \mathrm{ggT}(a - (k + 1) \cdot b, b)$. □

Beispiel 4.7 $\mathrm{ggT}(610, 407) = \mathrm{ggT}(407, 203) = \mathrm{ggT}(1, 203) = \mathrm{ggT}(1, 0) = 1$,
$\mathrm{ggT}(610, 377) = \mathrm{ggT}(377, 233) = \mathrm{ggT}(233, 144) = \mathrm{ggT}(144, 89) = \mathrm{ggT}(89, 55) = \mathrm{ggT}(55, 34) = \mathrm{ggT}(34, 21) = \mathrm{ggT}(21, 13) = \mathrm{ggT}(13, 8) = \mathrm{ggT}(8, 5) = \mathrm{ggT}(5, 3) = \mathrm{ggT}(3, 2) = \mathrm{ggT}(2, 1) = \mathrm{ggT}(1, 0) = 1$.

Für eine genaue Laufzeitanalyse von Algorithmus 4.8 benötigen wir zunächst die **Fibonacci-Zahlen** F_i für $i = 0, 1, 2, \ldots$, gegeben durch $F_0 := 0$, $F_1 := 1$ und

$$F_{n+1} := F_n + F_{n-1} \quad \text{für } n \geq 1.$$

Die ersten 11 Fibonacci-Zahlen sind: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

Lemma 4.9 *Es gilt:*

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

Beweis. Induktion über n: (oder Satz 3.4 anwenden)
Induktionsanfang: $n = 0$ und $n = 1$: klar.

Algorithmus 4.8

```
#include <iostream>

using namespace std;

int ggT(int a, int b)
{   if (b==0) return a;
    else return (ggT(b, a % b))
}

main ()
{   int a, b;

    cin >> a >> b;
    cout << "ggT = " << ggT(a,b) << endl;
}
```

Induktionsschritt:

$$\begin{aligned}
F_{n+1} &= F_n + F_{n-1} \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\cdot\left(\frac{1+\sqrt{5}}{2}+1\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\cdot\left(\frac{1-\sqrt{5}}{2}+1\right)\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right),
\end{aligned}$$

da $\frac{1+\sqrt{5}}{2}+1 = \frac{3+\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^2$ und $\frac{1-\sqrt{5}}{2}+1 = \frac{3-\sqrt{5}}{2} = \left(\frac{1-\sqrt{5}}{2}\right)^2$ □

Lemma 4.10 *Falls $a > b \geq 1$ und Algorithmus 4.8 $k \geq 1$ rekursive Aufrufe macht, so gilt $a \geq F_{k+2}$ und $b \geq F_{k+1}$.*

Beweis. Induktion über k:

Induktionsanfang: $k = 1$: Es gilt $b \geq 1 = F_2$ und $a > b$. $\Rightarrow a \geq 2 = F_3$.

Induktionsschritt: Sei die Aussage wahr, falls höchstens $k-1$ rekursive Aufrufe erfolgen. Wegen $a > b \geq 1$ und $k > 0$ ruft $\mathrm{ggT}(a, b)$ rekursiv $\mathrm{ggT}(b, a \bmod b)$ auf. Die Berechnung von $\mathrm{ggT}(b, a \bmod b)$ erfordert $k-1$ rekursive Aufrufe.
$\Rightarrow b \geq F_{k+1}$ und $a \bmod b \geq F_k$.
Es gilt $a = \lfloor a/b \rfloor \cdot b + (a \bmod b) \geq b + (a \bmod b) \geq F_{k+1} + F_k = F_{k+2}$ □

Satz 4.11 (Satz von Lamé) *Falls $a > b \geq 1$ und $b < F_{k+1}$, so macht Algorithmus 4.8 weniger als k rekursive Aufrufe.*

Beweis. Folgt sofort aus Lemma 4.10. □

Bemerkung 4.12 Es gilt $\frac{1+\sqrt{5}}{2} \approx 1.618\ldots$ und $\frac{1-\sqrt{5}}{2} \approx -0.618\ldots$.

$$\Rightarrow \quad F_n \approx \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n .$$

Falls $\mathrm{ggT}(F_{k+1}, F_k)$ berechnet wird, so sind genau $k-1$ rekursive Aufrufe notwendig, denn

$$\mathrm{ggT}(F_{k+1}, F_k) = \mathrm{ggT}(F_k, F_{k+1} - F_k) = \mathrm{ggT}(F_k, F_{k-1}).$$

$\Rightarrow$ Satz 4.11 ist bestmöglich.

Korollar 4.13 *Falls $a \geq b > 1$, so macht Algorithmus 4.8 höchstens $\lceil \log_\phi b \rceil$ viele rekursive Aufrufe mit $\phi = \frac{1+\sqrt{5}}{2}$.*

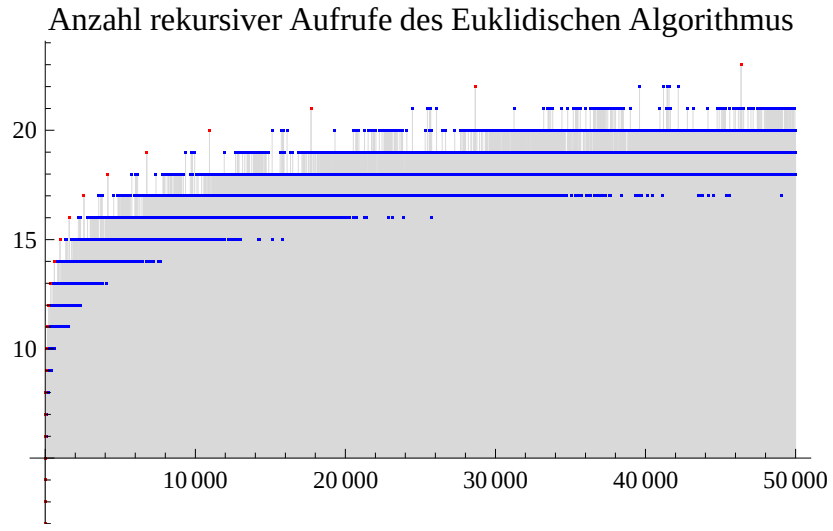
Beweis. Es gilt $F_n > \left(\frac{1+\sqrt{5}}{2}\right)^{n-2}$ für $n \geq 3$ (Beweis per Induktion). Falls $b < F_{k+1}$ so macht Algorithmus 4.8 gemäß Satz 4.11 höchstens $k-1$ rekursive Aufrufe.
Falls $b < \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} < F_n$, so macht Algorithmus 4.8 maximal $n-2$ rekursive Aufrufe.
Falls $n - 2 \geq \lceil \log_\phi b \rceil$, so gilt $\phi^{n-2} \geq b$. Damit folgt die Behauptung. □

Korollar 4.13 gibt eine obere Schranke für die **Worst-Case-Laufzeit** (Laufzeit = Anzahl elementarer Rechenoperationen) in Abhängigkeit von b für Algorithmus 4.8 an, das heißt, die maximale Laufzeit, die bei gegebenem b auftreten kann, wenn a beliebig variiert wird. Gemäß Bemerkung 4.12 wird die Worse-Case-Laufzeit aus Korollar 4.13 für einzelne Werte von b auch angenommen, nämlich für die Fibonacci-Zahlen. Im Allgemeinen werden aber weniger rekursive Aufrufe notwendig sein. Folgender Algorithmus bestimmt diese Anzahl:

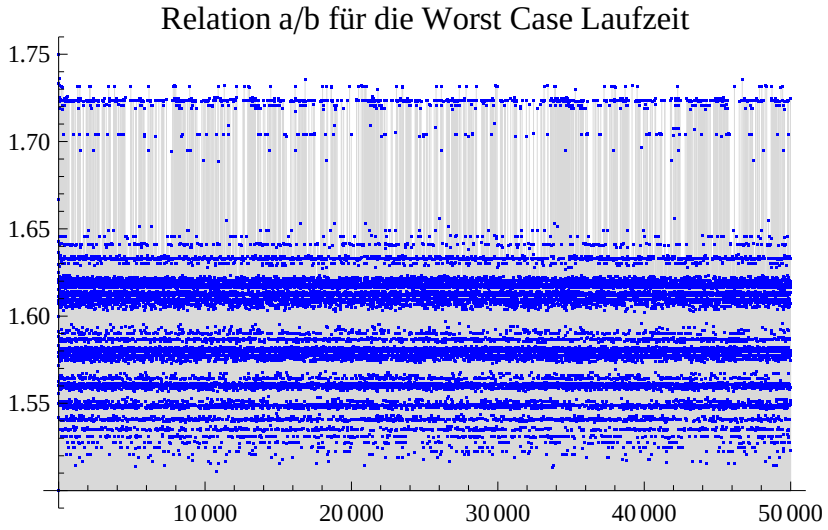
Algorithmus 4.14

```
#include <iostream>

using namespace std;

int NumCalls = 0;

int ggT(int a, int b)
{   if (b!=0)
    {  ++NumCalls;
       ggT(b, a % b);
    }
}

main ()
{   for (int b=1; b<=50000; ++b)
    {  int MaxNumCalls = 0;
       for (int a=b; a<=2*b; ++a)
       {  NumCalls = 0;
          ggT(a,b);
          if (NumCalls > MaxNumCalls) MaxNumCalls = NumCalls;
       }
       cout << MaxNumCalls << ", " ;
    }
}
```

Das Ergebnis dieses Algorithmus zeigt die folgende Abbildung:



Die roten Punkte sind die Anzahl der rekursiven Aufrufe für die Fibonacci-Zahlen. Merkt man sich in Algorithmus 4.14 für jedes b das kleinste a, das zur Worst-Case-Laufzeit für dieses b führt, so ergibt sich folgendes Bild für den Quotienten a/b.



Diese Bilder suggerieren folgende Fragen:

1. Ist die Worst-Case-Laufzeit stets größer $\lceil \log_2 b \rceil$?
2. Wird die Worst-Case-Laufzeit stets für ein a mit $1.5 \geq \frac{a}{b} \geq 1.75$ angenommen, falls $b \geq 5$? (Für $a = 4\,184\,066$ gilt zum Beispiel $b = 6\,299\,283$ und $\frac{a}{b} = 1.5055\ldots$. Für $a = 3\,728\,515$ gilt $b = 6\,500\,026$ und $\frac{a}{b} = 1.7433\ldots$.)

5 Sortieralgorithmen

5.1 Sortieren durch Einfügen

Das Sortierproblem:
Gegeben seien n Zahlen $a_1, a_2, \ldots, a_n \in \mathbb{R}$. Wir wollen diese Zahlen in aufsteigender Reihenfolge sortieren, das heißt, wir suchen eine bijektive Abbildung $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, sodass gilt:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n-1)} \leq a_{\pi(n)}.$$

Dies ist zum Beispiel nützlich, falls $\sum_{i=1}^{n} a_i$ berechnet werden soll.

Definition 5.1 *Eine bijektive Abbildung $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ heißt **Permutation**.*

Satz 5.2 *Es gibt $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ viele Permutationen der Menge $\{1, \ldots n\}$.*

Beweis. Induktion über n:

Induktionsanfang: $n = 1$: klar.

Induktionsschritt: Jede Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ lässt sich durch die Folge der Werte $\pi(1), \pi(2), \ldots, \pi(n)$ repräsentieren. Das Einfügen der Zahl $n+1$ in die Folge an den möglichen $n+1$ Stellen liefert genau alle Permutationen der Menge $\{1, \ldots, n+1\}$.
$\Rightarrow$ es gibt $(n+1) \cdot n! = (n+1)!$ davon. □

Damit lässt sich das Sortierproblem lösen, indem man alle $n!$ Permutationen bestimmt und für jede dieser Permutationen testet, ob sie Lösung des Sortierproblems ist. Im Worst-Case ist der Aufwand dafür mindestens $(n-1) \cdot n!$.

Der Beweis von Satz 5.2 und die Tatsache, dass die „$\leq$“-Relation transitiv ist ($a \leq b$, $b \leq c \Rightarrow a \leq c$) suggeriert die folgende effiziente Vorgehensweise:

Falls bereits $i-1$ Zahlen sortiert sind, so muss die i-te Zahl an einer der möglichen i Stellen geeignet eingefügt werden, um i sortierte Zahlen zu erhalten. Damit sind für die i-te Zahl maximal i Vergleiche erforderlich. Lässt man nun i von 1 bis n laufen, so erhält man eine Sortierung der n Zahlen und hat maximal $\sum_{i=1}^{n} = \frac{n(n+1)}{2}$ Vergleiche benötigt.

Beispiel 5.3 $a_1 = 5$, $a_2 = 2$, $a_3 = 7$, $a_4 = 3$, $a_5 = 6$.

Starte mit der ersten Zahl:	5				
Füge die zweite Zahl ein :	2	5			
Füge die dritte Zahl ein :	2	5	7		
Füge die vierte Zahl ein :	2	3	5	7	
Füge die fünfte Zahl ein :	2	3	5	6	7

Algorithmus 5.4

```
 1: #include <cstdlib>
 2:
 3: using namespace std;
 4:
 5: void InsertionSort (int *a, int n)
 6: { int i, j, tmp;
 7:   for (i = 1; i<n; ++i)
 8:   { tmp = a[i];
 9:     for (j=i-1; (j>=0) && (tmp < a[j]); --j) a[j+1]=a[j];
10:     a[j+1] = tmp;
11:   }
12: }
13:
14:
15: main()
16: { const int n=20;
17:   int a[n];
18:   for (int i = 0; i<n; ++i) a[i] = rand();
19:
20:   InsertionSort (a, n);
21: }
```

Satz 5.5 *Algorithmus 5.4 sortiert das Array $a[0 \ldots n-1]$ aufsteigend.*

Beweis. Behauptung: In jedem Durchlauf der ersten `for`-Schleife der Funktion `InsertionSort` enthält $a[0 \ldots i-1]$ die ersten i Elemente von a aufsteigend sortiert. Beweis durch Induktion nach i:

Induktionsanfang: $i = 1$: klar.

Induktionsschritt: Bei Beendigung der zweiten `for`-Schleife der Funktion `InsertionSort` gilt $j = -1$ oder $tmp > a[j]$. Somit wird durch $a[j+1] = tmp$ der Wert tmp korrekt einsortiert. □

5.2 $\mathcal{O}$-Notation

Für die Laufzeitanalyse ist folgende Notation nützlich.

Definition 5.6 *Seien $f : \mathbb{R}_+ \to \mathbb{R}_+$ und $g : \mathbb{R}_+ \to \mathbb{R}_+$ zwei Funktionen. Dann schreibt man:*

(a) $f(x) = \mathcal{O}(g(x))$ *(„$f(x)$ ist gleich groß O von $g(x)$“), falls es Konstanten c, $x_0 \in \mathbb{R}$ gibt mit $f(x) \leq c \cdot g(x) \quad \forall x \geq x_0$,*

(b) $f(x) = o(g(x))$ *(„$f(x)$ ist gleich klein o von $g(x)$“), falls gilt:* $\forall c > 0\ \exists x_0 \in \mathbb{R}$ *mit* $f(x) \leq c \cdot g(x) \quad \forall x \geq x_0$,

(c) $f(x) = \Omega(g(x))$, *falls* $g(x) = \mathcal{O}(f(x))$.

(d) $f(x) = \Theta(g(x))$, *falls* $f(x) = \mathcal{O}(g(x))$ *und* $g(x) = \mathcal{O}(f(x))$.

Beispiel 5.7 $f(x) = x + 7$, $g(x) = x^2 - 100$, $h(x) = 3x$.

1. $f(x) = \mathcal{O}(g(x))$, da für $c = 1$ und $x \geq x_0 = 108$ gilt:
   $1 \cdot g(x) = 1 \cdot (x^2 - 100) \geq 108x - 100 = x + 107x - 100 \geq x + 7 = f(x)$.
2. $f(x) = o(g(x))$, da für $x > \max\left(\frac{101}{c}, 1 + c\right)$ gilt:
   $c \cdot g(x) = cx^2 - 100c > 101x - 100c = x + 100x - 100c > x + 100(1 + c) - 100c = x + 100 > x + 7 = f(x)$.
3. $f(x) = \mathcal{O}(h(x))$, da für $c = 1$ und $x \geq x_0 = 4$ gilt:
   $1 \cdot h(x) = 3x \geq x + 8 \geq x + 7 = f(x)$
   $h(x) = \mathcal{O}(f(x))$, da für $c = 3$ und $x \geq x_0 = 1$ gilt:
   $3 \cdot f(x) = 3x + 21 > 3x = h(x)$
   $\Rightarrow f(x) = \Theta(h(x))$ und $h(x) = \Theta(f(x))$

Satz 5.8 *Seien* $f : \mathbb{R}_+ \to \mathbb{R}_+$ *und* $f : \mathbb{R}_+ \to \mathbb{R}_+$ *zwei Funktionen. Dann gilt:*

(a) $f(x) = o(g(x)) \quad \Leftrightarrow \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$.

(b) Falls es ein $c > 0$ *gibt mit* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$, *so gilt* $f(x) = \Theta(g(x))$.

Beweis.

(a)

$$\begin{aligned} f(x) = o(g(x)) &\Leftrightarrow \forall c > 0\ \exists x_0 \in \mathbb{R} \text{ mit } f(x) < c \cdot g(x)\ \forall x > x_0 \\ &\Leftrightarrow \forall c > 0\ \exists x_0 \in \mathbb{R} \text{ mit } \frac{f(x)}{g(x)} < c\ \forall x > x_0 \\ &\Leftrightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0. \end{aligned}$$

(b)

$$\begin{aligned} \lim_{x\to\infty} \frac{f(x)}{g(x)} = c &\Leftrightarrow \forall \varepsilon > 0\ \exists x_0 \text{ mit } \left|\frac{f(x)}{g(x)} - c\right| \leq \varepsilon\ \forall x > x_0 \\ &\Leftrightarrow \forall \varepsilon > 0\ \exists x_0 \text{ mit } c - \varepsilon \leq \frac{f(x)}{g(x)} \leq c + \varepsilon\ \forall x > x_0. \end{aligned}$$

Setze $\varepsilon := \frac{c}{2}$. $\Rightarrow \exists x_0$ mit $\frac{c}{2} \leq \frac{f(x)}{g(x)} \leq \frac{3}{2}c\ \forall x > x_0$.
$\Rightarrow \quad f(x) \leq \frac{3}{2}c \cdot g(x) \quad \forall x \geq x_0$
$\Rightarrow \quad f(x) = \mathcal{O}(g(x))$.
$\Rightarrow \quad g(x) \leq \frac{2}{c} \cdot f(x) \quad \forall x \geq x_0$
$\Rightarrow \quad g(x) = \mathcal{O}(f(x))$.
$\Rightarrow \quad f(x) = \Theta(g(x))$.

□

Satz 5.9 *Algorithmus 5.4 benötigt eine Laufzeit von $\mathcal{O}(n^2)$, um n Zahlen zu sortieren.*

Beweis. Die erste `for`-Schleife der Funktion `InsertionSort` wird $n-1$-mal durchlaufen. In jedem Durchlauf werden konstant viele elementare Operationen sowie die zweite `for`-Schleife der Funktion `InsertionSort` durchlaufen. Diese `for`-Schleife wird maximal i-mal durchlaufen und führt konstant viele elementare Operationen pro Durchlauf aus.
Also gilt, dass die Gesamtzahl der von Algorithmus 5.4 durchgeführten elementaren Rechenoperationen beschränkt ist durch

$$c_1 + \sum_{i=1}^{n-1}(c_2 + i \cdot c_3) \leq c_1 + n \cdot c_2 + \frac{n(n+1)}{2} \cdot c_3 = \mathcal{O}(n^2),$$

wobei $c_1, c_2, c_3 \in \mathbb{N}$ geeignete Konstanten sind. □

Bemerkung 5.10 Falls das Array a bereits aufsteigend sortiert ist, benötigt Algorithmus 5.4 lediglich eine Laufzeit von $\mathcal{O}(n)$. Falls a jedoch absteigend sortiert ist, so ist die Laufzeit von Algorithmus 5.4 $\Theta(n^2)$.

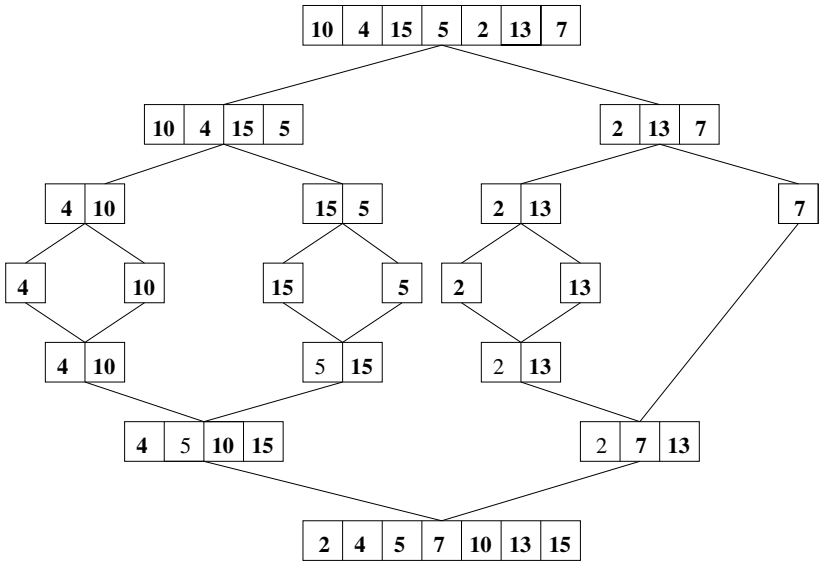
5.3 Merge-Sort

Merge-Sort ist ein weiteres Sortierverfahren, das gegenüber Algorithmus 5.4 den Vorteil hat, dass die Laufzeit durch $\mathcal{O}(n \log n)$ beschränkt ist, um n Zahlen zu sortieren.
Merge-Sort beruht auf dem „Divide-and-Conquer"-Prinzip: Teile das Ausgangsproblem in kleinere Teil, löse die kleinen Teile separat und füge die Lösung der kleinen Teile zur Gesamtlösung zusammen.
Merge-Sort geht dabei wie folgt vor:

1. Teile die zu sortierende Menge M in zwei möglichst gleich große Teile M_1 und M_2.
2. Sortiere M_1 und M_2 aufsteigend.
3. Verschmelze („merge") die Sortierungen von M_1 und M_2 zu einer Sortierung von M.

Schritt 1 ist offensichtlich einfach zu realisieren. Für Schritt 2 wird Merge-Sort rekursiv aufgerufen. Schritt 3 lässt sich leicht in $\mathcal{O}(n)$ bewerkstelligen, falls $|M_1| \leq |M_2| = n$, indem man ausgehend vom ersten Element in M_1 und M_2 jeweils das kleinere der beiden Elemente entfernt.

Beispiel 5.11



Algorithmus 5.12

```
 1: #include <cstdlib>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: void Merge (int *a, int Anfang, int Mitte, int Ende)
 7: // vereinigt a[Anfang..Mitte] und a[Mitte+1..Ende]
 8: { int tmp[Ende-Anfang+1], i = Anfang, j = Mitte+1;
 9:   for (int k=0; k <= Ende-Anfang; k++)
10:     if ( (i <= Mitte) && ((j > Ende) || (a[i] < a[j])) )
11:       {tmp[k] = a[i]; i++;}
12:     else
13:       {tmp[k] = a[j]; j++;}
14:   for (int k=0; k<= Ende-Anfang; k++)
15:     a[Anfang+k] = tmp[k];
16: }
17:
18:
19: void MergeSort (int* a, int Anfang, int Ende)
20: // sortiert a[Anfang..Ende] aufsteigend
21: { if (Anfang < Ende)
22:   { int Mitte = (Anfang + Ende) / 2;
23:     MergeSort (a, Anfang, Mitte);
24:     MergeSort (a, Mitte + 1, Ende);
25:     Merge (a, Anfang, Mitte, Ende);
26:   }
27: }
28:
29: main()
30: { const int n=20;
31:   int a[n];
32:   for (int i = 0; i<n; ++i) a[i] = rand();
33:
34:   MergeSort (a, 0, n-1);
35: }
```

Satz 5.13 *Die Laufzeit von Merge-Sort ist* $\mathcal{O}(n \log n)$.

Beweis. Bezeichne $T(n)$ die Laufzeit von Merge-Sort für n Elemente. Für $k \in \mathbb{N}$ und $n = 2^k$ gilt:

$$T(n) \leq 2 \cdot T(n/2) + c \cdot n \quad \text{mit } c \in \mathbb{N}.$$

Behauptung: $T(2^k) \leq 2^k \cdot T(1) + c \cdot k \cdot 2^k$.
Beweis per Induktion über k:
Induktionsanfang: $k = 1$: $T(2) \leq 2 \cdot T(1) + 2 \cdot c$.
Induktionsschritt:

$$\begin{aligned} T(2^{k+1}) &\leq 2 \cdot T(2^k) + c \cdot 2^{k+1} \\ &\leq 2 \cdot (2^k \cdot T(1) + c \cdot k \cdot 2^k) + c \cdot 2^{k+1} \\ &\leq 2^{k+1} \cdot T(1) + c \cdot 2^{k+1} \cdot (k+1) \end{aligned}$$

Wegen $k = \log_2 n$ gilt somit $T(n) = \mathcal{O}(n \log n)$. Beachte, dass gilt: $2 \cdot n \log_2(2 \cdot n) = \mathcal{O}(n \log_2 n)$ und $\log_a n = \mathcal{O}(\log_b n)$, falls $a, b \in \mathbb{N}$ mit $a, b \geq 2$. □

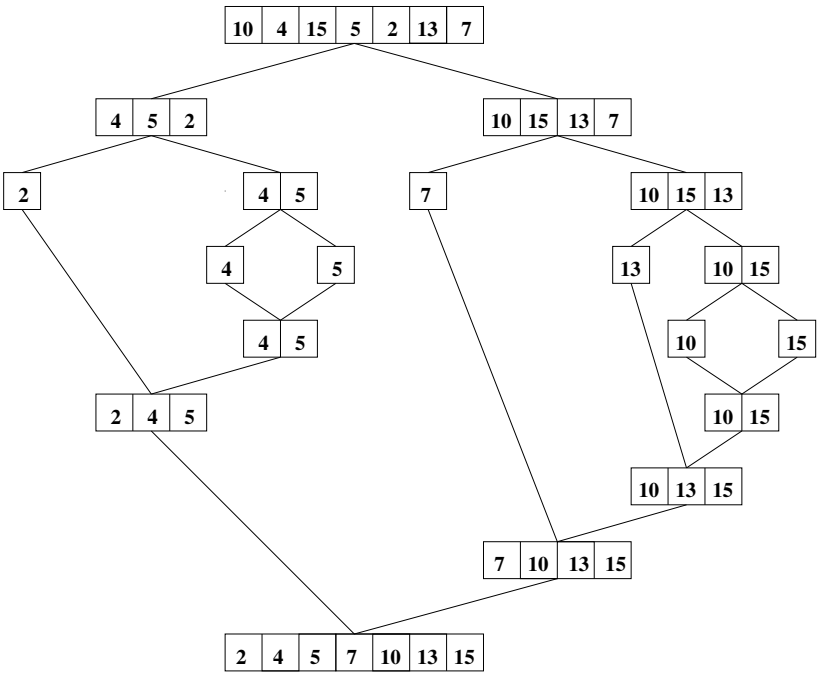
Bemerkung 5.14 Die Laufzeit von Merge-Sort ist $\Theta(n \log n)$, das heißt, auf einer schon sortierten Menge ist Merge-Sort langsamer als Algorithmus 5.4.

5.4 Quicksort

Quicksort basiert ebenfalls auf dem „Divide-and-Conquer"-Prinzip, spart jedoch den Aufwand des Merge-Schrittes, den Merge-Sort benötigt:

1. Teile die zu sortierende Menge M in zwei Teile M_1 und M_2, sodass es $x \in M$ gibt mit $y \leq x$, falls $y \in M_1$, und $y \geq x$, falls $y \in M_2$ und $|M_1|, |M_2| \geq 1$.
2. Sortiere M_1 und M_2 rekursiv mit Quicksort.
3. Füge die sortierten Mengen M_1 und M_2 aneinander.

Beispiel 5.15



Algorithmus 5.16

```
 1: #include <cstdlib>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: void QuickSort (int* a, int links, int rechts)
 7: // Sortiert a[links..rechts]
 8: { if (links < rechts)
 9:   { int pivot = a[rechts], l = links, r = rechts;
10:
11:     do { while (a[l] < pivot) l++;
12:          while (a[r] > pivot) r--;
13:          if (l <= r)
14:          { int tmp = a[l];
15:            a[l] = a[r];
16:            a[r] = tmp;
17:            l++;
18:            r--;
19:          }
20:        }
21:     while (l <= r);
22:
23:     QuickSort (a, links, r);
24:     QuickSort (a, l, rechts);
25:   }
26: }
27:
28: main()
29: { const int n=20;
30:   int a[n];
31:   for (int i = 0; i<n; ++i) a[i] = rand();
32:
33:   QuickSort (a, 0, n-1);
34: }
```

Ein weiterer Vorteil von Quicksort gegenüber Merge-Sort ist, dass kein zusätzlicher temporärer Speicherplatz benötigt wird, der nochmals genauso groß ist, wie die Inputdaten.

Satz 5.17 *Der Quicksort-Algorithmus hat eine Laufzeit von* $\mathcal{O}(n^2)$.

Beweis. Sei $T(n)$ die Laufzeit von Quicksort für n Elemente. Dann gilt

$$T(n) \leq c \cdot n + \max_{1 \leq k < n} (T(k) + T(n-k)).$$

Behauptung: $T(n) \leq c \cdot n^2$.
Beweis per Induktion über n.
Induktionsanfang: $n = 1$: klar.
Induktionsschritt:

$$\begin{aligned} T(n) &\leq c \cdot n + \max_{1 \leq k < n} (T(k) + T(n-k)) \\ &\leq c \cdot n + \max_{1 \leq k < n} (c \cdot k^2 + c \cdot (n-k)^2) \\ &\leq c \cdot n + c + c \cdot (n-1)^2 \\ &= c \cdot n^2 - c \cdot n + 2 \cdot c \leq c \cdot n^2. \end{aligned}$$

□

Bemerkung Die Worst-Case-Laufzeit wird angenommen, falls die Menge bereits sortiert ist.

Wir interessiern uns nun dafür, was die durchschnittliche Laufzeit $\overline{T}(n)$ des Quicksort-Algorithmus auf einer Menge von n Elementen ist. Sei Π die Menge aller $n!$ Permutationen der Menge $\{1, \ldots, n\}$. Mit $T(\pi)$ bezeichnen wir die Laufzeit des Quicksort-Algorithmus, um eine Permutation π zu sortieren. Dann definieren wir:

$$\overline{T}(n) := \frac{1}{n!} \cdot \sum_{\pi \in \Pi} T(\pi).$$

Satz 5.18 *Der Quicksort-Algorithmus hat durchschnittliche Laufzeit* $\mathcal{O}(n \log n)$.

Beweis. Für $n \geq 2$ gilt:

$$\begin{aligned} \overline{T}(n) &= n + 1 + \frac{1}{n} \cdot \left(\sum_{i=2}^{n-1} \overline{T}(i) + \overline{T}(n-i) \right) \\ &= n + 1 + \frac{2}{n} \cdot \sum_{i=1}^{n-1} \overline{T}(i) \end{aligned}$$

und $\overline{T}(0) = \overline{T}(1) = 0$.

$$\Rightarrow \quad \overline{T}(n+1) = n + 2 + \frac{2}{n+1} \sum_{i=1}^{n} \overline{T}(i).$$

$$\Rightarrow \quad n \cdot \overline{T}(n) = n^2 + n + 2 \cdot \sum_{i=1}^{n-1} \overline{T}(i)$$

$$\text{und } (n+1) \cdot \overline{T}(n+1) = n^2 + 3 \cdot n + 2 + 2 \cdot \sum_{i=1}^{n} \overline{T}(i)$$

$$\Rightarrow \quad (n+1) \cdot \overline{T}(n+1) - n \cdot \overline{T}(n) = 2 \cdot n + 2 + 2 \cdot \overline{T}(n)$$

$$\Rightarrow \quad \overline{T}(n+1) = 2 + \frac{n+2}{n+1} \cdot \overline{T}(n) \quad \text{für } n \geq 2.$$

Behauptung: $\overline{T}(n) = (n+1) \cdot \left(2 \cdot \sum_{i=1}^{n+1} \frac{1}{i} - \frac{8}{3} \right)$ für $n \geq 2$.

Induktionsanfang: $n = 2 : \overline{T}(n) = 3 \left(2 \cdot \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} \right) - \frac{8}{3} \right) = 3 \cdot \left(\frac{11}{3} - \frac{8}{3} \right) = 3.$

Induktionsschritt:

$$\begin{aligned} \overline{T}(n+1) &= 2 + \frac{n+2}{n+1} \cdot (n+1) \cdot \left(2 \cdot \sum_{i=1}^{n+1} \frac{1}{i} - \frac{8}{3} \right) \\ &= (n+2) \cdot \left(2 \cdot \sum_{i=1}^{n+1} \frac{1}{i} - \frac{8}{3} \right). \end{aligned}$$

Nun gilt $\sum_{i=1}^{n+1} \frac{1}{i} = 1 + \sum_{i=2} n+1 \frac{1}{i} \leq 1 + \int_{1}^{n+1} \frac{1}{x}\, dx = 1 + \ln(n+1) = \mathcal{O}(\log(n))$ und somit $\overline{T}(n) = (n+1) \cdot \left(2 \cdot \sum_{i=1}^{n+1} \frac{1}{i} - \frac{8}{3} \right) = \mathcal{O}(n \cdot \log n)$. □

5.5 Eine untere Schranke für das Sortieren

In diesem Abschnitt wollen wir untersuchen, welche Laufzeit ein Sortierverfahren, das n unterschiedliche Elemente sortiert, mindestens haben muss. Dazu nehmen wir an, dass die einzige Information, die wir über die n Elemente erhalten können, durch einen Vergleich „$<$“ zweier Elemente zu erzielen ist.

Satz 5.19 *Jeder Sortieralgorithmus, der n Elemente sortiert und Informationen über die Elemente nur durch paarweise Vergleiche von zwei Elementen gewinnt, benötigt mindestens* $\log_2(n!)$ *viele Vergleiche.*

Beweis. Betrachte alle $n!$ möglichen Permutationen auf n Elementen. Ein Vergleich „$a < b$“ von zwei Elementen a und b liefert für einen Teil der Permutationen „wahr“, für einen anderen Teil „falsch“. Der größere dieser beiden Teile enthält mindestens $\frac{n!}{2}$

viele Permutationen. Jeder weitere Vergleich halbiert den größeren der beiden Teile höchstens. Nach k Vergleichen gibt es also noch mindestens $\frac{n!}{2^k}$ viele Permutationen, die vom Algorithmus nicht unterschieden werden können. Jeder Sortieralgorithmus benötigt also mindestens k Vergleiche mit $2^k \geq n!$. Es folgt $k \geq \log(n!)$. □

Bemerkung 5.20 Sei n gerade. Dann gilt: $n! \geq \frac{n!}{(n/2)!} \geq \left(\frac{n}{2}\right)^{n/2}$.

$\Rightarrow \quad \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \Omega(n \log(n))$.

Somit ist Merge-Sort asymptotisch ein optimales Sortierverfahren. Allerdings benötigt Merge-Sort im Allgemeinen mehr als $\lceil \log(n!) \rceil$ viele Vergleiche. Man kann beweisen, dass es keinen Sortieralgorithmus gibt, der stets mit $\lceil \log(n!) \rceil$ vielen Vergleichen auskommt. Beispielsweise gibt es keinen Sortieralgorithmus, der 12 Elemente mit nur 29 Vergleichen sortiert. Man benötigt mindestens 30 Vergleiche.

5.6 Binäre Heaps und Heap-Sort

Eine **Prioritätswarteschlange** ist eine Datenstruktur, die die folgenden beiden Strukturen unterstützt:

- `Insert`: Einfügen eines Elementes in die Warteschlange,
- `ExtractMin`: Entfernen des kleinsten Elementes.

Nutzt man als Datenstruktur ein Array, so benötigt `Insert` $\mathcal{O}(1)$ Laufzeit und `ExtractMin` $\mathcal{O}(n)$ Laufzeit, falls n Elemente in der Prioritätswarteschlange sind. Nutzt man ein sortiertes Array, so benötigt `Insert` $\mathcal{O}(n)$ und `ExtractMin` $\mathcal{O}(1)$ Laufzeit. Binäre Heaps erlauben `Insert` und `ExtractMin` in $\mathcal{O}(\log n)$ durchzuführen.

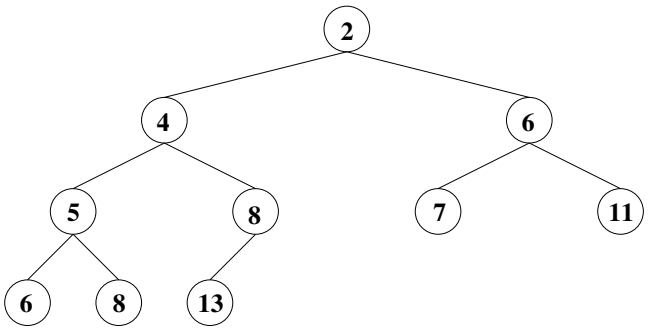
Definition 5.21 *Eine Zahlenfolge $a_1, a_2, \ldots, a_n$ hat die **Heapeigenschaft**, falls gilt:*

$$a_i \leq a_{2i} \quad \text{für } i = 1, \ldots, \lfloor n/2 \rfloor$$

und

$$a_i \leq a_{2i+1} \quad \text{für } i = 1, \ldots, \lfloor (n-1)/2 \rfloor.$$

Beispiel 5.22 Die Folge 2, 4, 6, 5, 8, 7, 11, 6, 8, 13 hat die Heapeigenschaft. Indem man jeweils die Elemente a_{2i} und a_{2i+1} unter a_i anordnet, lässt sich die Heapeigenschaft leicht ablesen, da zu jedem Element die beiden darunterstehenden mindestens genauso groß sein müssen.



Satz 5.23 *Falls $a_1, a_2, \ldots, a_n$ die Heapeigenschaft hat, so gilt $a_1 \leq a_i$ für alle $i = 1, \ldots, n$.*

Beweis. Sei j der kleinste Index mit $a_j \leq a_i$, falls $1 \leq i \leq n$. Falls $j = 1$, so gilt die Behauptung. Ansonsten muss wegen der Heapeigenschaft $a_{\lfloor j/2 \rfloor} \leq a_j$ gelten. Dies ist ein Widerspruch zur Wahl von j. □

Ein kleinstes Element lässt sich daher in einer Folge mit Heapeigenschaft in $\mathcal{O}(1)$ finden. Es bleibt zu zeigen, wie man ein Element hinzufügen oder a_1 entfernen kann und wieder eine Folge mit Heapeigenschaft erhält.

Idee von `Insert` und `ExtractMin`:
Füge neues Element ans Ende an beziehungsweise setze $a_1 := a_n$ und entferne a_n. Falls die Heapeigenschaft erfüllt ist, so bleibt nicht zu tun. Ansonsten liefern die folgenden Lemmata die Lösung.

Lemma 5.24 *Sei $a_1, a_2, \ldots, a_n$ eine Zahlenfolge, die nicht die Heapeigenschaft hat, aber diese erhält, falls a_j durch $a_{\lfloor j/2 \rfloor}$ ersetzt wird. Durch Vertauschen von a_j mit $a_{\lfloor j/2 \rfloor}$ erhält man entweder eine Folge mit Heapeigenschaft oder es gilt $a_{\lfloor j/2 \rfloor} < a_{\lfloor \lfloor j/2 \rfloor /2 \rfloor}$ und man erhält eine Folge mit Heapeigenschaft, falls man $a_{\lfloor j/2 \rfloor}$ durch $a_{\lfloor \lfloor j/2 \rfloor /2 \rfloor}$ ersetzt.*

Beweis. Falls Vertauschen von a_j mit $a_{\lfloor j/2 \rfloor}$ Heap liefert, so ist nichts zu zeigen. Falls nicht, so kann bei Vertauschen von a_j mit $a_{\lfloor j/2 \rfloor}$ lediglich die Relation $a_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} < a_{\lfloor j/2 \rfloor}$ verletzt sein, da $a_{\lfloor j/2 \rfloor}$ durch ein kleineres Element ersetzt wird und nach Voraussetzung das Ersetzen von a_j durch $a_{\lfloor j/2 \rfloor}$ eine Folge mit Heapeigenschaft liefert. Ersetzt man $a_{\lfloor j/2 \rfloor}$ durch $a_{\lfloor \lfloor j/2 \rfloor /2 \rfloor}$, so gilt offensichtlich die Heapeigenschaft. □

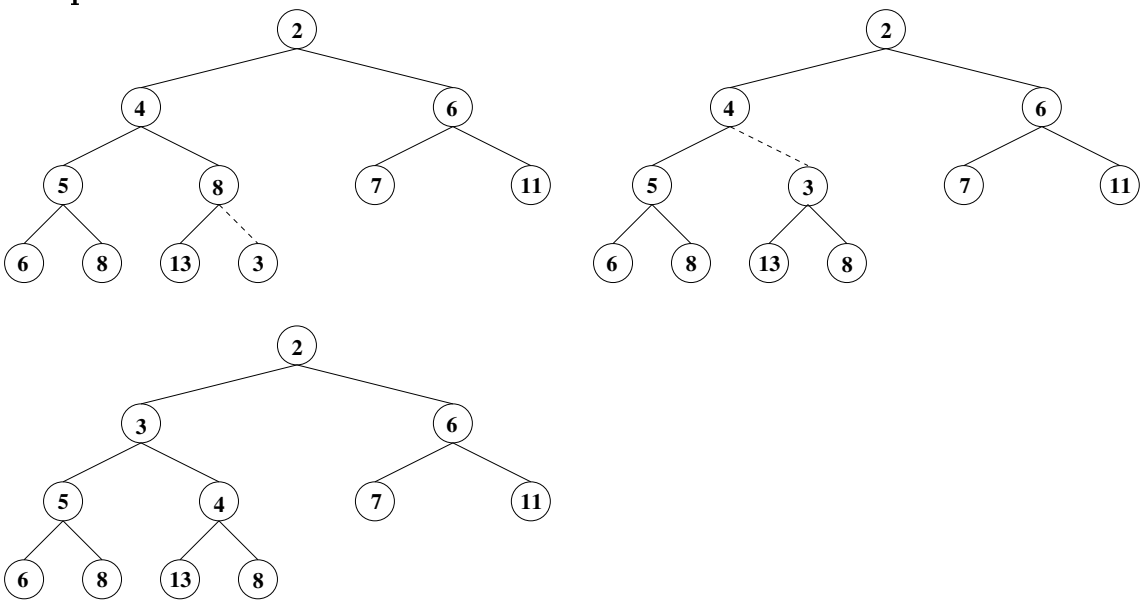
Algorithmus 5.25

```
 1: #include <cstdlib>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6: const int n=20;
 7: int HeapSize, a[n];
 8:
 9: void Insert (int value)
10: { HeapSize++;
11:   int i = HeapSize;
12:   while (a[i/2] > value)
13:   {  a[i] = a[i/2];
14:      i = i /2;
15:   }
16:   a[i] = value;
17: }
18:
19:
20: int ExtractMin ()
21: { int MinElement = a[1], i=1, LastElement = a[HeapSize];
22:   HeapSize--;
23:   while (2*i <= HeapSize)
24:   { int Child = 2*i;
25:     if ((Child < HeapSize) && (a[Child] > a[Child+1])) Child++;
26:     if (a[Child] >= LastElement) break;
27:     a[i] = a[Child];
28:     i = Child;
29:   }
30:   a[i] = LastElement;
31:   return (MinElement);
32: }
33:
34:
35: main()
36: { a[0]     = -1;
37:   HeapSize = 0;
38:   for (int i = 0; i<n; ++i) Insert(rand());
39:   for (int i = 0; i<n; ++i) cout << ExtractMin() << " ";
40: }
```

Lemma 5.26 *Die Funktion* `Insert` *fügt ein Element in die Folge* $a[1], \ldots, a[n+1]$ *in* $\mathcal{O}(\log n)$ *so ein, dass die Heapeigenschaft erhalten bleibt.*

Beweis. `Insert` wendet iterativ Lemma 5.24 auf die Folge $a[1], \ldots, a[n+1]$ an, wobei $a[n+1]$ das einzufügende Element ist. Korrektheit ergibt sich daher aus Lemma 5.24. Die Laufzeit ist $\mathcal{O}(\log n)$, da die Folge $n+1, \lfloor (n+1)/2 \rfloor, \lfloor \lfloor (n+1)/2 \rfloor /2 \rfloor, \ldots$ nach spätestens $\mathcal{O}(\log n)$ vielen Schritten den Wert 1 erreicht. □

Beispiel 5.27

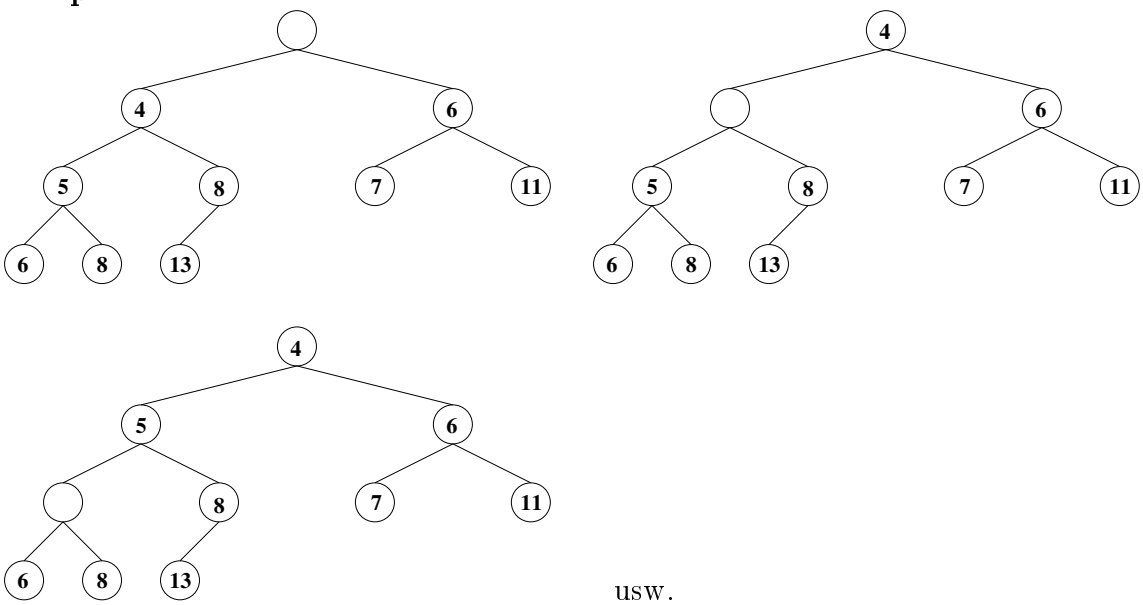


Die Funktion `ExtractMin` verläuft analog zu `Insert`, indem die Lücke, die das Entfernen von $a[1]$ verusacht, durch das kleinere der beiden Elemente $a[2]$ und $a[3]$ gefüllt wird. Dies wird iterativ fortgesetzt: Die Lücke zwischen $a[j]$ wird durch das kleinere der beiden Elemente $a[2j]$ und $a[2j+1]$ ersetzt, bis das letzte Element in die Lücke passt.

Lemma 5.28 *Falls $a[1], \ldots, a[n]$ die Heapeigenschaft hat, so liefert* `ExtractMin` *in $\mathcal{O}(\log n)$ eine Folge mit Heapeigenschaft, die das Element $a[1]$ nicht enthält.*

Beweis. Analog zu Lemma 5.26. □

Beispiel 5.29



usw.

Mit Hilfe binärer Heaps lassen sich nun leicht n Zahlen in $\mathcal{O}(n \log n)$ sortieren. Zunächst werden die Zahlen in $\mathcal{O}(n \log n)$ in den Heap einsortiert, sodann in $\mathcal{O}(n \log n)$ durch n Aufrufe von `ExtractMin` aus dem Heap wieder herausgeholt. Der resultierende Algorithmus wird **Heap-Sort** genannt.

Algorithmus 5.30

```
 1: #include <cstdlib>
 2: #include <iostream>
 3:
 4: using namespace std;
 5:
 6:
 7: class BinaryHeap
 8: { public:
 9:     BinaryHeap (int n)
10:       { size = 0;
11:         a = new int [n+1];
12:         a[0] = -1;
13:       }
14:     void Insert (int value);
15:     int ExtractMin ();
16:   private:
17:     int *a;
18:     int size;
19: };
20:
21:
22: void BinaryHeap::Insert (int value)
23: { size++;
24:   int i = size;
25:   while (a[i/2] > value)
26:   {  a[i] = a[i/2];
27:      i = i /2;
28:   }
29:   a[i] = value;
30: }
31:
32:
33: int BinaryHeap::ExtractMin ()
34: { int MinElement = a[1], i=1, LastElement = a[size];
35:   size--;
36:   while (2*i <= size)
37:   { int child = 2*i;
38:     if ((child < size) && (a[child] > a[child+1])) child = child+1;
39:     if (a[child] >= LastElement) break;
40:     a[i] = a[child];
41:     i = child;
42:   }
43:   a[i] = LastElement;
44:   return (MinElement);
45: }
46:
47:
48: main()
49: { const int n=20;
50:   BinaryHeap H(n);
51:   for (int i = 0; i<n; ++i) H.Insert(rand());
52:   for (int i = 0; i<n; ++i) cout << H.ExtractMin() << " ";
53: }
```

Satz 5.31 *Heap-Sort sortiert n Elemente in $\mathcal{O}(n \log n)$.*

Beweis. Folgt aus Lemma 5.26 und 5.28. □

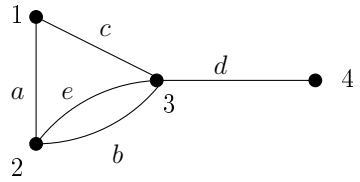
Bemerkung: Heap-Sort benötigt im Gegensatz zu Merge-Sort keinen zusätzlichen Speicher.

6 Algorithmen auf Graphen

6.1 Grundlagen

Ein **ungerichteter Graph** ist ein Tripel (V, E, Ψ), wobei V und E endliche Mengen sind und $\Psi : E \to \{X \subset V \mid |X| = 2\}$. Ein **gerichteter Graph** (Digraph) ist ein Tripel (V, E, Ψ), wobei V und E endliche Mengen sind und $\Psi : E \to \{(v, w) \in V \times V \mid v \neq w\}$. Ein **Graph** ist ein gerichteter oder ungerichteter Graph. Die Elemente aus V heißen **Knoten**, die Elemente aus E **Kanten**.

Beispiel 6.1 $(\{1, 2, 3, 4\}, \{a, b, c, d, e\}, \{a \mapsto \{1, 2\}, b \mapsto \{2, 3\}, c \mapsto \{1, 3\}, d \mapsto \{3, 4\}, e \mapsto \{3, 2\}\})$.
Veranschaulichung durch Zeichnung im $\mathbb{R}^2$:
Knoten $\hat{=}$ Punkte, Kanten $\hat{=}$ Linien zwischen Knoten. Im gerichteten Fall: Ein Pfeil auf einer Kante gibt die Richtung vom ersten zum zweiten Knoten an:



Zwei Kanten $e \neq e'$ mit $\Psi(e) = \Psi(e')$ heißen **parallel**. Ein Graph ohne parallele Kanten wird **einfacher Graph** genannt. Für einfache Graphen identifiziert man e mit $\Psi(e)$, das heißt, man schreibt $G = (V, E)$ mit $E \subseteq \{\{v, w\} \mid v, w \in V,\ v \neq w\}$ beziehungsweise $E \subseteq \{(v, w) \mid v, w \in V,\ v \neq w\}$. Wir brauchen diese vereinfachte Notation vielfach im Zusammenhang mit nicht einfachen Graphen, die Menge E ist dann eine Multimenge.

Für eine Kante $e = \{x, y\}$ beziehungsweise $e = (x, y)$ sagt man, dass e die beiden Knoten x und y **verbindet**. Die Knoten x und y heißen in diesem Fall **benachbart** beziehungsweise **adjazent**. Die Knoten x und y werden die **Endknoten** von e genannt; x ist **Nachbar** von y und y Nachbar von x. Die Knoten x und y sind mit der Kante e **inzident**. Falls $e = (x, y)$, so sagen wir: e beginnt in x und endet in y oder e geht von x nach y.

Die Anzahl der Elemente in V beziehungsweise in E, das heißt $|V|$ und $|E|$, wird mit $n(G)$ beziehungsweise mit $m(G)$ oder aber einfach mit n und m bezeichnet.

Für einen Graphen G und $X, Y \subseteq V(G)$ definieren wir:
$E(X, Y) := \{\{x, y\} \in E(G) \mid x \in X \setminus Y \text{ und } y \in Y \setminus X\}$, falls G ungerichtet,

$E^+(X,Y) := \{(x,y) \in E(G) \mid x \in X \setminus Y \text{ und } y \in Y \setminus X\}$, falls G gerichtet ist.

- Sei G ein ungerichteter Graph und $X \subseteq V(G)$.
  Dann definiere $\delta(X) := E(X, V(G) \setminus X)$. Die **Nachbarschaft** von X ist die Menge $N(X) := \{v \in V(G) \setminus X \mid E(X,\{v\}) \neq \emptyset\}$.
- Sei G Digraph und $X \subseteq V(G)$. Dann definiere $\delta^+(X) := E^+(X, V(G) \setminus X)$, $\delta^-(X) := E^+(V(G) \setminus X, X)$ und $\delta(X) := \delta^+(X) \cup \delta^-(X)$.

Für $x \in V$ setze $\delta(x) := \delta(\{x\})$, $N(x) := N(\{x\})$, $\delta^+(x) := \delta^+(\{x\})$ und $\delta^-(x) := \delta^-(\{x\})$. Der **Grad** eines Knoten ist als $|\delta(x)|$ definiert, das heißt, der Grad von x ist die Anzahl der mit x inzidenten Kanten. Im gerichteten Fall bezeichnen $|\delta^-(x)|$ beziehungsweise $|\delta^+(x)|$ den **Eingangs-** beziehungsweise **Ausgangsgrad** von x und es gilt, $|\delta(x)| = |\delta^+(x)| + |\delta^-(x)|$.

Ein Graph, in dem alle Knoten den Grad k haben heißt k-**regulär**. Ein Knoten vom Grad 0 heißt **isoliert**.

Satz 6.2 *Für jeden Graphen $G = (V,E)$ gilt: $\sum_{x \in V} |\delta(x)| = 2|E|$.*

Beweis. Jede Kante ist mit genau zwei Knoten inzident. Auf der linken Seite der Gleichung zählt man daher jeder Kante genau zweimal. □

Satz 6.3 *Für jeden Digraphen $G = (V,E)$ gilt: $\sum_{x \in V} |\delta^+(x)| = \sum_{x \in V} |\delta^-(x)|$.*

Beweis. Auf beiden Seiten wird jede Kante genau einmal gezählt. □

Aus Satz 6.2 folgt sofort:

Korollar 6.4 *Die Anzahl der Knoten in einem Graphen mit ungeradem Grad ist gerade.*

Ein Graph $H = (V(H), E(H))$ ist **Teilgraph** (Subgraph, Untergraph) des Graphen $G = (V(G), E(G))$, falls $V(H) \subseteq V(G)$ und $E(H) \subseteq E(G)$. Man sagt auch, dass G den Graphen H enthält. Falls $V(G) = V(H)$, so heißt H ein **(auf-)spannender Teilgraph**.

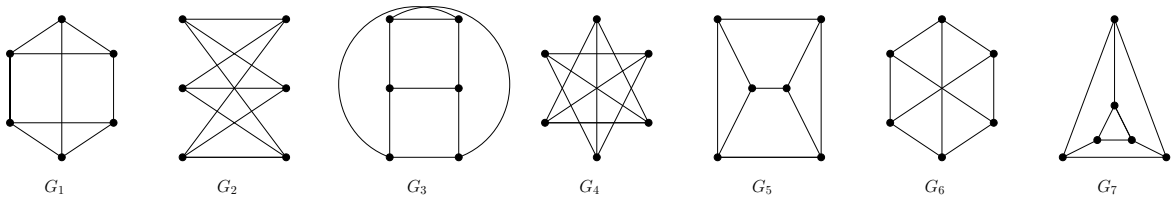
H heißt **induzierter Teilgraph**, falls $E(H) = \{\{x,y\} \in E(G) \mid x,y \in V(H)\}$ beziehungsweise $E(H) = \{(x,y) \in E(G) \mid x,y \in V(H)\}$. Ein induzierter Teilgraph von G ist bereits vollständig durch seine Knotenmenge $V(H)$ spezifiziert. Man sagt daher auch, dass H der **von** $V(H)$ **induzierte Teilgraph von** G ist und notiert diesen als $H := G[V(H)]$.

Für $v \in V(G)$ setze $G - v := G[V(G) \setminus \{v\}]$. Entsprechend sei für $e \in E(G)$ der Graph $G - e$ durch $G - e := (V(G), E(G) \setminus \{e\})$ definiert.

Ein einfacher ungerichteter Graph heißt **vollständig**, falls je zwei Knoten adjazent sind. Einen vollständigen Graphen auf n Knoten bezeichnen wir mit K_n. Das **Komplement** $\overline{G}$ eines einfachen ungerichteten Graphen G ist der Graph $\overline{G} = (V(G), E(\overline{G}))$ mit $E(\overline{G}) := \{\{x,y\} \mid x,y \in V(G), \{x,y\} \notin E(G)\}$.

Zwei Graphen heißen **isomorph**, falls es eine Bijektion $\varphi : V(G) \to V(H)$ gibt, die eine Bijektion zwischen $E(G)$ und $E(H)$ durch $\varphi(\{x,y\}) := \{\varphi(x), \varphi(y)\}$ beziehungsweise $\varphi((x,y)) := (\varphi(x), \varphi(y))$ impliziert. Wir schreiben $G \cong H$ oder sogar $G = H$, da wir zwischen isomorphen Graphen nicht unterscheiden. Ebenso sagen wir „H ist Teilgraph von G“, falls H isomorph zu einem Teilgraphen von G ist.

Beispiel 6.5



Es gilt: $G_1 \cong G_4 \cong G_5 \cong G_7$ und $G_2 \cong G_3 \cong G_6$.

Für $n = 10$ gibt es 12 005 168 nicht isomorphe einfache ungerichtete Graphen. Man kann zeigen, dass es $(1+o(1))\frac{2^{\binom{n}{2}}}{n!}$ nicht isomorphe einfache ungerichtete Graphen und $(1+o(1))\frac{4^{\binom{n}{2}}}{n!}$ nicht isomorphe einfache gerichtete Graphen auf n Knoten gibt.
Falls mehrere Graphen auf der gleichen Knotenmenge betrachtet werden, ist es häufig erforderlich anzugeben, auf welchen Graphen man sich bezieht. Man bezeichnet daher in solchen Fällen die entsprechenden Parameter mit dem Graphen als Index, zum Beispiel $\delta_G(x)$ oder $N_G(x)$ etc.
Ein **Kantenzug** in einem Graphen G ist eine Folge $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_{k+1}$ mit $k \geq 0$ und $e_i = (x_i, x_{i+1}) \in E(G)$ beziehungsweise $e_i = \{x_i, x_{i+1}\} \in E(G)$ für $i = 1, \ldots, k$. Falls $x_1 = x_{k+1}$, so handelt es sich um einen **geschlossenen Kantenzug**. Ein **Weg** ist ein Graph $P = (\{x_1, \ldots, x_{k+1}\}, \{e_1, \ldots, e_k\})$, sodass $x_i \neq x_j$ für $1 \leq i < j \leq k+1$ gilt und $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_{k+1}$ ein Kantenzug ist. Man sagt auch, dass P ein x_1-x_{k+1}-Weg ist oder dass P x_1 und x_{k+1} verbindet oder dass x_{k+1} von x_1 aus erreichbar ist. Die Knoten x_1 und x_{k+1} werden die **Endknoten** des Weges P genannt. Mit $P_{[x,y]}$ mit $x, y \in V(P)$ bezeichnen wir den eindeutigen Teilgraphen von P, der x-y-Weg ist. Die Knoten $V(P) \setminus \{x, y\}$ sind die **inneren Knoten** von P.

Lemma 6.6 *Es gibt genau dann einen x-y-Weg in einem Graphen, falls es einen Kantenzug von x nach y gibt.*

Beweis.

„$\Rightarrow$“: Nach Definition ist ein x-y-Weg ein Kantenzug von x nach y.

„$\Leftarrow$“: Angenommen, im Kantenzug kommt ein Knoten mehrfach vor. Dann eliminiere alle Knoten zwischen dem ersten und einschließlich letzten Auftreten des Knotens. □

Ein **Kreis** ist ein Graph $C = (\{x_1, \ldots, x_k\}, \{e_1, \ldots, e_k\})$, sodass $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_1$ ein geschlossener Kantenzug ist und $x_i \neq x_j$ sowie $e_i \neq e_j$ für $1 \leq i < j \leq k$. Die

Länge eines Weges oder Kreises ist die Anzahl seiner Kanten. Ist ein Weg oder Kreis Teilgraph von G, so spricht man von einem Weg oder Kreis in G.

Lemma 6.7 *Sei G ein ungerichteter einfacher Graph, in dem jeder Knoten Grad mindestens k hat. Dann enthält G einen Weg der Länge mindestens k. Falls $k \geq 2$, so enthält G einen Kreis der Länge mindestens $k+1$.*

Beweis. Es sei $P = (\{x_1, \ldots, x_{l+1}\}, \{e_1, \ldots, e_l\})$ ein längster Weg in G. Dann liegen alle Nachbarn von x_{l+1} in der Menge $\{x_1, \ldots, x_l\}$, da sonst P nicht längster Weg wäre. Es gilt somit $k \leq |\delta(x_{l+1})| \leq l$, das heißt, der Weg P hat Länge mindestens k.
Ist i ein minimaler Index mit $\{x_i, x_{l+1}\} \in E(G)$, so sind die Knoten x_i und x_l wegen $k \geq 2$ verschieden und $(\{x_i, x_{i+1}, \ldots, x_{l+1}\}, \{e_i, e_{i+1}, \ldots, e_l\} \cup \{x_i, x_{l+1}\})$ ist ein Kreis der Länge mindestens $k+1$ in G. □

Sei G ein ungerichteter Graph. G heißt **zusammenhängend**, falls es für je zwei Knoten $x, y \in V(G)$ einen x-y-Weg in G gibt. Ansonsten heißt G **unzusammenhängend**. Die (inklusions-)maximalen zusammenhängenden Teilgraphen von G sind die **Zusammenhangskomponenten** von G. Ein Knoten v heißt **Artikulationsknoten**, falls $G - v$ mehr Zusammenhangskomponenten hat als G. Eine Kante e heißt **Brücke**, falls $G - e$ mehr Zusammenhangskomponenten hat als G.

Satz 6.8 *Ein ungerichteter Graph G ist genau dann zusammenhängend, falls $\delta(X) \neq \emptyset$ für alle $\emptyset \subset X \subset V(G)$.*

Beweis.

„⇒“: Sei $\emptyset \subset X \subset V(G)$ und $x \in X$, $v \in V(G) \setminus X$. Da G zusammenhängend ist, gibt es einen x-v-Weg in G. Da $x \in X$ aber $v \notin X$, muss der Weg eine Kante $\{a, b\}$ enthalten mit $a \in X$ und $b \notin X$. $\Rightarrow \delta(X) \neq \emptyset$.

„⇐“: Angenommen G ist unzusammenhängend. Seien a und b Knoten in $V(G)$, die nicht durch einen a-b-Weg verbunden sind.
Setze $X := \{x \in V(G) \mid \exists a\text{-}x\text{-Weg in } G\}$. Nach Definition von X gilt dann $\delta(X) = \emptyset$. Aber $a \in X$ und $b \notin X$ nach Annahme. $\Rightarrow \emptyset \subset X \subset V(G)$. $\Rightarrow$ $\delta(X) \neq \emptyset$ nach Voraussetzung. ↯
□

Bemerkung 6.9 Nach Definition kann man testen, ob ein Graph zusammenhängend ist, indem man für jedes der $\binom{n}{2}$ vielen Knotenpaare testet, ob diese durch einen Weg verbunden sind. Satz 6.8 liefert nun eine weitere Möglichkeit: Teste für jede Menge X mit $\emptyset \subset X \subset V(G)$, ob $\delta(X) \neq \emptyset$ ist. Da es $2^n - 2$ viele solcher Mengen X gibt, erfordert dieser Ansatz exponentiellen Aufwand.

Ein ungerichteter Graph heißt **Wald**, falls er keine Kreise enthält. Ein **Baum** ist ein zusammenhängender Wald, das heißt, ein Wald ist ein Graph, dessen Zusammenhangskomponenten Bäume sind. Ein Knoten vom Grad 1 in einem Baum heißt **Blatt**.

Sei $\mathcal{F}$ eine Familie von Mengen oder Graphen. Dann ist $F \in \mathcal{F}$ **minimal**, falls keine echte Teilmenge beziehungsweise kein echter Teilgraph von F in $\mathcal{F}$ ist. Entsprechend ist $F \in \mathcal{F}$ **maximal**, falls F keine echte Teilmenge beziehungsweise kein echter Teilgraph eines Elements in F ist.
Beachte, dass minimale beziehungsweise maximale Elemente nicht notwendigerweise kardinalitätsminimal beziehungsweise -maximal sind.

Satz 6.10 *Jeder Baum mit mindestens zwei Knoten enthält mindestens zwei Blätter.*

Beweis. Betrachte einen maximalen Weg im Baum. Dieser hat eine Länge ≥ 1 und seine Endknoten müssen wegen der Kreisfreiheit Blätter sein. □

Satz 6.11 *Sei G ein ungerichteter Graph auf n Knoten. Dann sind die folgenden Aussagen äquivalent:*

(a) G ist ein Baum.

(b) G hat $n-1$ Kanten und enthält keine Kreise.

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

(d) G ist ein minimal zusammenhängender Graph (das heißt, jede Kante von G ist eine Brücke).

(e) G ist minimaler Graph mit $\delta(X) \neq \emptyset$ für alle $\emptyset \subset X \subset V(G)$.

(f) G ist maximaler kreisfreier Graph (das heißt, das Hinzufügen einer beliebigen Kante zwischen in G nicht benachbarten Knoten erzeugt einen Kreis).

(g) Zwischen je zwei Knoten in G gibt es einen eindeutigen Weg.

Beweis.
(a)⇒(g): Angenommen es gibt zwei verschiedene x-y-Wege. Die Vereinigung dieser beiden x-y-Wege enthält einen Kreis. Widerspruch zur Definition eines Baumes.

(g)⇒(e): Nach Definition ist G zusammenhängend und nach Satz 6.8 gilt $\delta(X) \neq \emptyset$ für alle $\emptyset \subset X \subset V(G)$. Angenommen es gibt eine Kante e, sodass weiterhin $\delta(X) \neq \emptyset$ für alle $\emptyset \subset X \subset V(G-e)$ gilt. Dann ist $G-e$ nach Satz 6.8 zusammenhängend. ⇒ In G sind die Endknoten von e durch zwei verschiedene Wege verbunden. Widerspruch.

(e)⇒(d): Folgt unmittelbar aus Satz 6.8.

(d)⇒(f): Jede Kante ist eine Brücke ⇒ G enthält keine Kreise. ⇒ G ist zusammenhängend. ⇒ Das Hinzufügen einer Kante erzeugt einen Kreis.

(f)⇒(b): Induktion über n:
$n = 1$: klar.
$n > 1$: G enthält Knoten vom Grad 1, da G kreisfrei ist. Entferne diesen. Dann entsteht ein neuer Graph, der maximal kreisfrei ist. Nach Induktionsannahme hat dieser $n-1$ Knoten und $n-2$ Kanten. ⇒ G hat $n-1$ Kanten und ist kreisfrei.

(b)⇒(c): Induktion über n:
$n = 1$: klar.
$n > 1$: G hat keinen Kreis. Es gibt also Knoten vom Grad 1. Entferne diesen. ⇒ Der

Rest ist nach Induktionsannahme zusammenhängend. Damit ist auch G zusammenhängend.
(c)⇒(a): Induktion über n:
$n = 1$: klar.
$n > 1$: Entferne Knoten vom Grad 1. (Es existiert ein solcher Knoten, da nur $n-1$ Kanten im Graphen sind und $\sum_{x \in V} |\delta(x)| = 2n - 2$.) G ist ein Baum. ⇒ Nach Hinzufügen von Knoten vom Grad 1 bleibt der Graph zusammenhängend und kreisfrei. □

Korollar 6.12 *Ein Wald auf n Knoten mit k Zusammenhangskomponenten hat $n-k$ Kanten.*

Beweis. Jede Komponente ist ein Baum mit $n_i - 1$ Kanten für $i = 1, \ldots, k$ und $\sum_{i=1}^{k} n_i = n$. □

Korollar 6.13 *Ein ungerichteter Graph ist genau dann zusammenhängend, wenn er einen spannenden Baum enthält.*

Beweis. Dies gilt wegen (d) ⇒ (a) aus Satz 6.11. □

6.2 Implementierung von Graphen

Sei $G = (V, E)$ ein Graph auf n Knoten.
Die Matrix $A \in \mathbb{R}^{n \times n}$ heißt **Adjazenzmatrix** von G, falls gilt:

$$A = (a_{x,y})_{x,y \in V(G)} = \begin{cases} 1, & \text{falls } \{x, y\} \in E(G) \text{ bzw. } (x, y) \in E(G) \\ 0, & \text{sonst.} \end{cases}$$

Die Matrix $A \in \mathbb{R}^{n \times m}$ heißt **Inzidenzmatrix** von G, falls gilt: $A = (a_{x,e})_{x \in V(G), e \in E(G)}$ mit

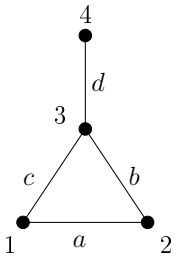
$$a_{x,e} = \begin{cases} 1, & \text{falls } x \in e \\ 0, & \text{sonst,} \end{cases}$$

falls G ungerichtet, und

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

falls G gerichtet.

Beispiel 6.14



Adjazenzmatrix

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	1	0
4	0	0	1	0

Inzidenzmatrix

	a	b	c	d
1	1	0	1	0
2	1	1	0	0
3	0	1	1	1
4	0	0	0	1

Der wesentliche Nachteil von Adjazenzmatrizen zur Darstellung von Graphen ist deren hoher Speicherbedarf. Eine Adjazenzmatrix benötigt $\Omega(n^2)$ Speicherplatz auch für dünne Graphen, das heißt für Graphen mit $o(n^2)$ vielen Kanten.
Bei der **Adjazenzlistendarstellung** eines Graphen merkt man sich für jeden Knoten eine Liste aller mit ihm adjazenten Knoten.

Beispiel 6.15 Die Adjazenzlistendarstellung des Graphen G aus Beispiel 6.14 hat die Form

$\boxed{1} \to \boxed{2} \to \boxed{3}$
$\boxed{2} \to \boxed{1} \to \boxed{3}$
$\boxed{3} \to \boxed{1} \to \boxed{2} \to \boxed{4}$
$\boxed{4} \to \boxed{3}$

Die Adjazenzlisten können je nach Bedarf einfach verkettete oder doppelt verkettete Listen sein. Auch kann man statt der adjazenten Knoten diese indirekt speichern, indem man sich die inzidenten Kanten merkt. Im Allgemeinen muss man davon ausgehen, dass die Knoten in den Adjazenzlisten in beliebiger Reihenfolge stehen.

Folgende Tabelle gibt Laufzeiten und Speicherbedarf von Adjazenzmatrix und Adjazenzlisten für einen Graphen mit n Knoten und m Kanten an.

	Adjazenzmatrix	Adjazenzliste
Speicherbedarf	$\mathcal{O}(n^2)$	$\mathcal{O}(n+m)$
Knoten einfügen	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Knoten entfernen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante entfernen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Test, ob $\{x,y\} \in E(G)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Für einen Digraphen G ist der **zugrundeliegende ungerichtete Graph** derjenige Graph G' mit gleicher Knotenmenge wie G, der für jede Kante $(x,y) \in E(G)$ die Kante $\{x,y\}$ enthält. G heißt auch **Orientierung** von G'.
Ein Digraph heißt **zusammenhängend**, falls der zugrundeliegende ungerichtete Graph zusammenhängend ist. Ein Digraph ist ein **Branching**, falls er keine Kreise enthält und $|\delta^-(x)| \leq 1$ für jeden Knoten x gilt. (Der einem Branching zugrundeliegende Graph ist ein Wald.)

Ein zusammenhängendes Branching heißt **Arboreszenz**. (Der einer Arboreszenz zugrundeliegende Graph ist ein Baum.) Eine Arboreszenz mit n Knoten hat somit $n-1$ Kanten. Wegen $|\delta^-(x)| \leq 1$ für alle Knoten x, muss es also genau einen Knoten r mit $\delta^-(r) = \emptyset$ geben. Dieser Knoten r heißt **Wurzel**. Knoten v mit $\delta^+(v) = \emptyset$ in einer Arboreszenz heißen **Blätter**.
Ein Digraph heißt **stark zusammenhängend**, falls es für alle $s, t \in V(G)$ einen Weg s nach t und einen Weg von t nach s gibt. Die **starken Zusammenhangskomponenten** eines Digraphen sind die maximal stark zusammenhängenden Teilgraphen.

Graph-Scanning-Algorithmus

Input: Ein Graph G, ein Knoten $s \in V(G)$

Output: Die Menge $R \subseteq V(G)$ der von s aus erreichbaren Knoten und einen Menge $T \subseteq E(G)$, sodass (R, T) eine Arboreszenz mit Wurzel s beziehungsweise ein Baum ist.

```
1  R := {s}, Q := {s}, T := ∅;
2  while Q ≠ ∅ do begin
3     wähle v ∈ Q beliebig;
4     if ∃e = (v, w) bzw. e = {v, w} mit w ∈ V(G) \ R
5        then R := R ∪ {w}, Q := Q ∪ {w}, T := T ∪ {e};
6        else Q := Q \ {v}
7  end while
```

Satz 6.16 *Der Graph-Scanning-Algorithmus arbeitet korrekt und kann in Laufzeit $\mathcal{O}(n+m)$ implementiert werden.*

Beweis. Behauptung: Zu jedem Zeitpunkt ist (R, T) ein Baum beziehungsweise eine Arboreszenz mit Wurzel s.
Dies gilt sicherlich zu Beginn des Algorithmus. Die einzige Veränderung erfährt (R, T) in Zeile 5. Dort wird ein Knoten und eine Kante hinzugefügt, es handelt sich weiterhin um einen Baum beziehungsweise eine Arboreszenz mit Wurzel s. Angenommen es gibt am Ende des Algorithmus einen von s aus erreichbaren Knoten w, der nicht in R ist. Sei P ein s-w-Weg und sei $\{x, y\}$ beziehungsweise (x, y) eine Kante in P mit $x \in R$ und $y \notin R$. Da $x \in R$ muss x auch zu einem bestimmten Zeitpunkt in Q gelegen haben (vergleiche Zeile 1 und Zeile 5). Der Algorithmus endet nicht, solange x nicht in Zeile 6 aus Q entfernt wurde. Zeile 6 wird aber nur erreicht, falls es keine Kante (x, y) beziehungsweise $\{x, y\}$ gibt mit $y \notin R$. Widerspruch.
Zur Laufzeit: Merke für jeden Knoten x innerhalb seiner Liste $\delta(x)$ beziehungsweise $\delta^+(x)$ die aktuelle Position, bis zu der schon alle inzidenten Kanten betrachtet wurden. Zu Beginn ist dies die Anfangsposition der Liste. Damit wird jede Kante des Graphen maximal zweimal betrachtet, jeder Knoten einmal aus Q entfernt. Die Gesamtlaufzeit ist also $\mathcal{O}(n+m)$. (Dabei wird angenommen, dass Zeile 3 in $\mathcal{O}(n)$ Gesamtlaufzeit durchgeführt wird, zum Beispiel indem man immer den ersten Knoten aus Q wählt.) □

Satz 6.17 *Die Zusammenhangskomponenten eines ungerichteten Graphen können in*

$\mathcal{O}(n+m)$ bestimmt werden.

Beweis. Starte den Graph-Scanning-Algorithmus in einem beliebigen Knoten. Falls $R = V(G)$, so ist der Graph zusammenhängend. Andernfalls ist $G[R]$ eine Zusammenhangskomponente von G und man iteriere mit $G[V \setminus R]$. □

Für zwei Knoten v und w in einem Graphen G bezeichne $\text{dist}(v,w)$ die Länge eines kürzesten v-w-Weges in G. Wir nennen $\text{dist}(v,w)$ auch den **Abstand** von v und w in G. Falls es keinen v-w-Weg in G gibt, so setzen wir $\text{dist}(v,w) := \infty$. In ungerichteten Graphen gilt $\text{dist}(v,w) = \text{dist}(w,v)$ für alle $v, w \in V(G)$.
Zeile 3 des Graph-Scanning-Algorithmus kann auf verschiedene Weisen realisiert werden. Wählt man jeweils den Knoten $v \in Q$, der als letzter zu Q hinzugefügt wurde (das heißt, Q wird als LIFO-Stack realisiert), so wird der Algorithmus **Tiefensuche** beziehungsweise **DFS** (für depth-first-search) genannt. Dieser wird meist wie folgt rekursiv implementiert:

```
1    R := ∅; DFS-visit(s);

DFS-visit(v)
1    R := R ∪ {v};
2    for w mit (v,w) ∈ E(G) do
3       if w ∉ R then DFS-visit(w)
```

Falls man jeweils den Knoten aus Q wählt, der als erstes zu Q hinzugefügt wurde (FIFO-queue), so handelt es sich um **Breitensuche** oder **BFS** (für breadth first search). Die Bäume beziehungsweise Arboreszenzen (R,T), die von DFS beziehungsweise BFS berechnet werden, heißen DFS-Baum beziehungsweise BFS-Baum.

Satz 6.18 *Ein BFS-Baum enthält einen kürzesten Weg von einem Knoten s zu allen anderen, von s aus erreichbaren Knoten. Die Werte* $\text{dist}(s,v)$ *für alle $v \in V(G)$ können in $\mathcal{O}(n+m)$ bestimmt werden.*

Beweis. Wir modifizieren den Algorithmus BFS wie folgt:

```
①  R := {s}, Q := {s}, T := ∅, l(s) := 0;
②  while Q ≠ ∅ do begin
③     v := erster Knoten in Q; entferne v aus Q;
④     for w ∈ N(v) \ R do
⑤        R := R∪{w}, Q := Q∪{w}, T := T∪{v,w} bzw. (v,w), l(w) := l(v)+1;
⑥  end while
```

Es gilt :
$(*)$ Wird v in ③ gewählt $\Rightarrow l(w) \leq l(v)+1$ für alle $w \in Q$ und $l(x) \geq l(y)$, falls x nach y in Q aufgenommen wird.
Beweis: Die Behauptung gilt zu Beginn und in ⑤ wird ein Knoten hinzugefügt, dessen

l-Wert genau um 1 größer ist als der von x.

Behauptung: Zu jedem Zeitpunkt des Algorithmus gilt: $l(v) = \text{dist}_{(R,T)}(s,v) = \text{dist}_G(s,v)$ für alle $v \in R$.
Angenommen es gibt einen Knoten w mit $\text{dist}_G(s,w) \neq \text{dist}_{(R,T)}(s,w)$, also $\text{dist}_G(s,w) < \text{dist}_{(R,T)}(s,w)$. Sei w so gewählt, dass es der Knoten mit geringstem Abstand zu s ist der diese Eigenschaft hat. Sei P kürzester s-w-Weg in G und sei $e = (v,w)$ beziehungsweise $e = \{v,w\}$ die letzte Kante auf P. Dann gilt $\text{dist}_G(s,v) = \text{dist}_{(R,T)}(s,v)$ und $\text{dist}_G(s,w) = \text{dist}_G(s,v) + 1 < \text{dist}_{(R,T)}(s,w)$. Also gilt

$$l(v) + 1 = \text{dist}_{(R,T)}(s,v) + 1 < \text{dist}_{(R,T)}(s,w) = l(w),$$

also wird v vor w aus Q entfernt. Nach obigen Überlegungen gilt: Falls w zu diesem Zeitpunkt in Q, so gilt $l(w) \leq l(v) + 1$, sonst wegen der Kante e und Zeile ⑤ $l(w) = l(v) + 1$. ↯ □

Ein **leerer Graph** ist ein Graph, der keine Kanten enthält. Eine **Bipartition** eines ungerichteten Graphen ist eine Partition der Knotenmenge $V(G) = A \dot\cup B$, sodass $G[A]$ und $G[B]$ leere Graphen sind. Ein Graph heißt **bipartit**, falls er eine Bipartition besitzt. Der **vollständig bipartite Graph** G mit $V(G) = A \dot\cup B$ und $E(G) = \{\{a,b\} \mid a \in A,\ b \in B\}$ mit $|A| = n$ und $|B| = m$ wird mit $K_{n,m}$ bezeichnet. Wenn wir die Notation $G = (A \dot\cup B, E)$ benutzen, so meinen wir damit, dass $G[A]$ und $G[B]$ leere Graphen sind. Ein **ungerader Kreis** ist ein Kreis ungerader Länge.

Satz 6.19 *Ein ungerichteter Graph ist genau dann bipartit, wenn er keinen ungeraden Kreis enthält. In linearer Zeit kann man für einen ungerichteten Graphen entweder eine Bipartition oder einen ungeraden Kreis finden.*

Beweis.

„⇒“ Sei G bipartit mit Bipartition $V(G) = A \dot\cup B$ und sei $((v_1,\ldots,v_k),(e_1,\ldots,e_k))$ ein Kreis in G. Ohne Beschränkung der Allgemeinheit sei $v_1 \in A$. Folglich gilt $v_2 \in B$, $v_3 \in A$ usw., das heißt $v_i \in A$ genau dann, wenn i ungerade. Wegen $e_k = (v_k, v_1)$ gilt: k ist gerade.

„⇐“ G ist ohne Beschränkung der Allgemeinheit zusammenhängend. (Argumentiere andernfalls für jede Zusammenhangskomponente.) Berechne einen beliebigen spannenden Baum T von G und wähle einen Knoten $s \in V(G)$. Sei $A := \{v \in V(G) \mid \text{dist}_T(s,v) \text{ ist gerade}\}$. Definiere $B := V \setminus A$.
Behauptung: $V(G) = A \dot\cup B$ ist Bipartition.
Falls nicht, so gibt es ohne Beschränkung der Allgemeinheit eine Kante $e = \{x,y\}$ in $G[A]$ oder $G[B]$. Aber dann bildet e zusammen mit dem x-y-Weg in T einen ungeraden Kreis. (Die Länge des x-y-Weges ist gerade, da x-s-Weg und y-s-Weg gleiche Parität haben und, falls sie eine Kante gemeinsam haben, folgt die Parität von beiden Teilen ändert sich.)

□

6.3 Minimale spannende Bäume

Sei $G = (V, E)$ ein ungerichteter Graph. Gegeben sei im Folgenden eine Gewichts-/ Kostenfunktion für die Kanten des Graphen $c : E(G) \to \mathbb{R}$. Für $F \subseteq E(G)$ sei $c(F) := \sum_{e \in F} c(e)$ und $c(\emptyset) = 0$.
Für einen Teilgraphen T von G seien $c(E(T)) = \sum_{e \in E(T)} c(e)$ die Kosten/das Gewicht von T.

Minimum-Spanning-Tree-Problem

Input: Ein zusammenhängender ungerichteter Graph $G = (V, E)$, Gewichte $c : E(G) \to \mathbb{R}$.

Output: Ein spannender Baum in G mit minimalem Gewicht (auch minimaler spannender Baum genannt).

Kruskals Algorithmus

Input: Ein zusammenhängender ungerichteter Graph $G = (V, E)$, Gewichte $c : E(G) \to \mathbb{R}$

Output: Ein minimaler spannender Baum T

① Sortiere $E(G)$, sodass $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$;
② $T := (V(G), \emptyset)$;
③ `for` $i = 1$ `to` m `do`
④ `if` $T + e_i$ ist kreisfrei `then` $T := T + e_i$;

Satz 6.20 *Der Algorithmus von Kruskal bestimmt einen minimalen spannenden Baum und kann in $\mathcal{O}(m \log(n))$ implementiert werden.*

Beweis. Der Graph T, der vom Algorithmus von Kruskal zurückgeliefert wird, ist offensichtlich kreisfrei. Angenommen T ist nicht zusammenhängend, dann gibt es eine Partition $V(G) = A \dot\cup B$ mit $|A|, |B| \geq 1$, sodass keine Kante von T zwischen A und B verläuft. Da G zusammenhängend ist, muss es aber in G eine Kante zwischen A und B geben. Diese wäre in Zeile ④ ausgwählt worden. Widerspruch.
Wir zeigen nun, dass T tatsächlich minimaler spannender Baum ist. Sei T^* ein minimaler spannender Baum, sodass der kleinste Index i, für den $e_i \in T$ aber $e_i \notin T^*$ größtmöglich ist. Falls dieser Index nicht existiert, so gilt $T = T^*$ und wir sind fertig. Ansonsten betrachte den Graphen $T^* + e_i$. Dieser enthält einen Kreis. Da T kreisfrei ist, muss dieser Kreis eine Kante $e_j \notin T$ enthalten. Die Kanten, die vor e_i in T eingefügt wurden, sind gemäß Annahme auch alle in T^* enthalten. Zu dem Zeitpunkt, zu dem Kruskals Algorithmus die Kante e_i zu T hinzugefügt, hätte auch e_j zu T hinzugefügt werden können, ohne einen Kreis zu erzeugen. Da $e_j \notin T$ muss also $j > i$ und damit $c(e_j) \geq c(e_i)$ gelten. Dann ist aber $(T^* + e_i) - e_j$ ein minimaler spannender Baum, der der Wahl von T^* widerspricht.

Laufzeit des Algorithmus:
Sortieren der m Kanten geht in $\mathcal{O}(m \log(m)) = \mathcal{O}(m \log(n))$. Die `for`-Schleife wird m-mal durchlaufen. Der Test, ob $T + e_i$ kreisfrei ist, lässt sich in $\mathcal{O}(1)$ durchführen,

wenn man sich für jeden Knoten die Komponente merkt, in der er aktuell liegt. Zu Beginn liegen alle Knoten in unterschiedlichen Komponenten. Wird eine Kante vom Algorithmus zu T hinzugefügt, so werden zwei Komponenten vereinigt und die entsprechenden Werte aller Knoten der einen Komponente auf die der anderen gesetzt. Damit lässt sich Kreisfreiheit in $\mathcal{O}(1)$ testen, da lediglich geprüft werden muss, ob die Endknoten der einzufügenden Kanten in unterschiedlichen Komponenten liegen.
Es bleibt zu zeigen, dass die Aktualisierung der Komponentennummern hinreichend schnell durchgeführt werden kann. Die Idee hierzu ist, dass man beim Einfügen einer Kante den Knoten der kleineren Komponente als neuen Komponentenwert den der größeren Komponente gibt. Damit ist sichergestellt, dass jeder Knoten maximal $\mathcal{O}(\log(n))$-mal eine Komponentennummer bekommt. Die Gesamtlaufzeit beträgt somit $\mathcal{O}(m + n\log(n))$ für die Zeilen ③ und ④.
Die Laufzeit des Algorithmus von Kruskal ist daher $\mathcal{O}(m\log(n))$. □

Eine **Eulertour** ist ein geschlossener Kantenzug, der jede Kante des Graphen genau einmal enthält. Ein ungerichteter Graph heißt **eulersch**, falls der Grad eines jeden Knotens gerade ist. Ein gerichteter Graph heißt eulersch, falls $|\delta^-(v)| = |\delta^+(v)|$ für alle $v \in V(G)$.

Satz 6.21 *Ein zusammenhängender Graph besitzt genau dann eine Eulertour, wenn er eulersch ist.*

Beweis.
„⇒“ trivial.
„⇐“ Folgt aus dem nachfolgendem Algorithmus.

□

Eulers Algorithmus
Input: Ein zusammenhängender eulerscher Graph G
Output: Eine Eulertour in G
① Wähle $v_1 \in V(G)$ beliebig. `Return`(Euler(G, v_1))

Euler(G, v_1)
① $W := v_1$, $x = v_1$;
② `while` $\delta(x) \neq \emptyset$ `do`
③ wähle $e(x, y) \in \delta(x)$;
④ $W := W, e, y$; $x := y$; $E(G) := E(G) \setminus \{e\}$;
⑤ Sei $W := v_1, e_1, \ldots, v_k, e_k, v_{k+1}$;
⑥ `for` $i = 1$ `to` k `do` $W_i :=$Euler(G, v_i);
⑦ `return` $(W_1, e_1, W_2, e_2, \ldots, W_k, e_k, v_{k+1})$;

Satz 6.22 *Eulers Algorithmus ist korrekt und kann in $\mathcal{O}(m+n)$ implementiert werden.*

Beweis. Wir benutzen Induktion über $|E(G)|$. $E(G) = \emptyset$. klar.
Aufgrund der Gradbedingung gilt $v_1 = v_{k+1}$ in Zeile ⑤, das heißt, W ist ein geschlossener Kantenzug, in dem jede Kante nur einmal vorkommt. Sei G' der Graph nach Entfernen von $e_1, \ldots, e_k$. Dann erfüllt auch G' die Gradbedingung. Für jedes $e \in E(G')$ gibt es ein kleinstes $i \in \{1, \ldots k\}$, sodass e in der gleichen Zusammenhangskomponente wie v_i liegt. Dann gehört e nach Induktionsannahme zu W_i und W, wie in Zeile ⑦ zurückgeliefert, enthält allen Kanten von G genau einmal.
Die Laufzeit ist linear, da jede einmal besuchte Kante sofort gelöscht wird. □

6.4 Kürzeste Wege

Gegeben sei ein Graph $G = (V, E)$ und $c : E \to \mathbb{R}$. Wir haben schon gesehen, wie man $\text{dist}_G(x, y)$ für $x, y \in V(G)$ mittels Breitensuche in $\mathcal{O}(n + m)$ ausrechnen kann. Definiere $\text{dist}_{(G,c)}(x, y) := \min\{c(E(P)) \mid P \text{ ist ein } x\text{-}y\text{-Weg}\}$.
Wie berechnet man $\text{dist}_{(G,c)}(x, y)$?
Kürzeste Wege sind im Folgenden immer gewichtet.

Kürzeste-Wege-Problem
Input: Ein Digraph G, $c : E(G) \to \mathbb{R}$, $s, t \in V(G)$.
Output: Ein kürzester s-t-Weg in G beziehungsweise die Angabe, dass ein solcher nicht existiert.

Es zeigt sich, dass das Problem schwieriger ist, falls negative Kantengewichte vorkommen. Wir setzen daher voraus, dass alle Kantengewichte nicht-negativ sind.

Dijkstras Algorithmus
Input: Ein Digraph G, $c : E(G) \to \mathbb{R}_+$, $s \in V(G)$.
Output: $l(x) = \text{dist}_{(G,c)}(s, x)$ für alle $x \in V(G)$.

① $l(s) := 0$, $l(x) = \infty$ für $x \in V(G) \setminus \{s\}$, $Q := V$;
② `while` $Q \neq \emptyset$ `do`
③ $x :=$ Knoten in Q mit $l(x)$ minimal;
④ $Q := Q \setminus \{x\}$;
⑤ `for` $y \in Q$ mit $(x, y) \in E(G)$ `do`
⑥ $l(y) := \min(l(y), l(x) + c((x, y)))$;

Satz 6.23 *Dijkstras Algorithmus bestimmt* $\text{dist}_{(G,c)}(s, x)$ *für alle* $x \in V(G)$ *und kann in* $\mathcal{O}(n^2)$ *implementiert werden.*

Beweis. Angenommen es gibt bei Beendigung des Algorithmus einen Knoten x mit $l(x) \neq \text{dist}_{(G,c)}(s, x)$. Betrachte den Zeitpunkt (Zeile ③), bevor x aus Q entfernt wird,

und sei x der erste Knoten, der aus Q entfernt wird mit $l(x) \neq \text{dist}_{(G,c)}(s,x)$. Dann gilt $x \neq s$ und $l(x) > \text{dist}_{(G,c)}(s,x)$, da Zeile ⑥ die Existenz eines s-x-Weges der Länge höchstens $l(x)$ impliziert.

Seien $s = v_1, v_2, \ldots, v_k = x$ die Knoten eines kürzesten s-x-Weges in G. Sei i der größte Index eines Knotens auf diesem Weg, der bereits aus Q entfernt wurde. Dieser Index existiert, da $s \notin Q$. Da $v_{i+1} \in Q$, gilt $l(v_{i+1}) \leq l(v_i) + c((v_i, v_{i+1}))$ nach Zeile ⑥. Nach Zeile ③ gilt:

$$l(x) \leq l(v_{i+1}) \leq l(v_i) + c((v_i, v_{i+1})) \leq \text{dist}_{(G,c)}(s,x).$$

Laufzeit: Die `while`-Schleife wird n-mal durchlaufen. Zeile ③ benötigt $\mathcal{O}(n) \Rightarrow \mathcal{O}(n^2)$. Die Zeilen ⑤ und ⑥ benötigen insgesamt $\mathcal{O}(m)$.
$\Rightarrow$ Die Gesamtlaufzeit beträgt $\mathcal{O}(n^2)$. □

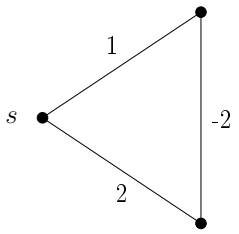
Bemerkung 6.24 Indem man sich in Zeile ⑥ für den Knoten y im Falle, dass $l(x) + c((x,y)) < l(y)$ gilt, den Knoten x als Vorgänger von y merkt, kann man sich auch leicht zusätzlich die kürzesten x-y-Wege für alle $x \in V(G)$ merken. Diese bilden alle zusammen eine Arboreszenz mit Wurzel s.

Durch die Benutzung binärer Heaps lässt sich Dijkstras Algorithmus effizienter implementieren.

Satz 6.25 *Dijkstras Algorithmus kann in $\mathcal{O}(m \log n)$ implementiert werden.*

Beweis. Benutze für die Menge Q binäre Heaps. Dann lassen sich die Zeilen ③ und ④ in $\mathcal{O}(\log n)$ realisieren. Zeile ⑥ benötigt ebenfalls $\mathcal{O}(\log n)$. Somit ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(n \log n + m \log n) = \mathcal{O}(m \log n)$. □

Bemerkung 6.26 Falls man auch negative Kantengewichte erlaubt, so funktioniert Dijkstras Algorithmus nicht mehr:



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

Lemma 6.27 *Sei G Digraph und $c : E(G) \to \mathbb{R}$ konservativ. Sei $k \in \mathbb{N}$ und $s, w \in V(G)$. Sei P ein kürzester s-w-Weg mit höchstens k Kanten und sei $e = (v,w)$ die letzte Kante in P. Dann ist $P_{[s,v]}$ ein kürzester s-v-Weg unter allen s-v-Wegen mit höchstens $k-1$ Kanten.*

Beweis. Angenommen Q ist kürzester s-v-Weg mit $|E(Q)| \leq k-1$ und $c(E(Q)) < c(E(P-e)) = c(E(P)) - c(e)$. Falls $w \notin V(Q)$, so ist $Q+e$ ein kürzerer s-w-Weg als P mit höchstens k Kanten. Ansonsten gilt:

$$\begin{aligned} c(E(Q_{[s,w]})) &= c(E(Q)) + c(e) - c(E(Q_{[w,v]} + e)) \\ &< c(E(P)) - c(E(Q_{[w,v]} + e)) \\ &\leq c(E(P)), \end{aligned}$$

da c konservativ ist und $Q_{[w,v]} + e$ Kreis ist. Widerspruch zur Wahl von P. □

Bellman-Moore-Algorithmus

Input: Ein Digraph G, konservatives $c : E(G) \to \mathbb{R}$, $s \in V(G)$.

Output: $l(x) = \text{dist}_{(G,c)}(s,x)$ für alle $x \in V(G)$.

① $l(s) := 0$, $l(x) = \infty$ für $x \in V(G) \setminus \{s\}$;
② `for` $i := 1$ `to` $n-1$ `do`
③ `for` $(x,y) \in E(G)$ `do`
④ $l(y) := \min\{l(y), l(x) + c((x,y))\}$;

Satz 6.28 *Der Bellman-Moore-Algorithmus arbeitet korrekt und kann in $\mathcal{O}(m \cdot n)$ implementiert werden.*

Beweis. Laufzeit: Zeile ④ wird $\mathcal{O}(m \cdot n)$ mal durchlaufen und kann in $\mathcal{O}(1)$ realisiert werden. Rest: $\mathcal{O}(n)$.

Korrektheit: Wir zeigen induktiv über die Anzahl der Kanten k eines kürzesten s-x-Weges für alle $x \in V(G)$:
$(*)$ Nach spätestens k Iterationen der `for`-Schleife in Zeile ② gilt $l(x) = \text{dist}_{(G,c)}(s,x)$.
Beweis: $k = 1$: klar.
Sei x ein Knoten, sodass ein kürzester s-x Weg P mit k Kanten existiert, und sei y vorletzter Knoten dieses Weges. Dann ist $P_{[s,y]}$ ein kürzester s-y-Weg, denn gäbe es einen kürzeren Weg Q, so enthielte $P \cup Q$ einen kürzeren s-x-Weg als P (falls $x \notin Q$, so ist $Q \cup \{x,y\}$ ein solcher Weg; sonst muss $Q_{[x,y]}$ kürzer als $\{x,y\}$ sein $\Rightarrow P_{[s,x]} + Q_{[x,y]}$ ist kürzer als P).
$\Rightarrow l(y) = \text{dist}_{(G,c)}(s,y)$ nach $k-1$ Iterationen.
$\Rightarrow l(x) \leq l(y) + c((y,x)) = \text{dist}_{(G,c)}(s,y) + c((y,x)) = \text{dist}_{(G,c)}(s,x)$. □

Bemerkung 6.29 Was passiert, wenn man den Bellman-Moore-Algorithmus auf einem Graphen mit nicht-konservativen Kantengewichten laufen lässt?
Falls sich in der n-ten Iteration eine Änderung ergibt, folgt, dass c nicht konservativ war. Ansonsten ändert sich auch in den nachfolgenden Iterationen nichts. Somit sind keine negativen Kreise von s aus erreichbar.

Wir wollen nun noch einen einfachen Algorithmus kennen lernen, mit dem man für konservatives c den Abstand zwischen allen Paaren von Knoten berechnen kann. Dies

geht, indem man n-mal den Bellman-Moore-Algorithmus aufruft, jedoch benötigt man dann eine Laufzeit von $\mathcal{O}(n^2 \cdot m)$. Nachfolgender Ansatz benötigt lediglich eine Laufzeit von $\mathcal{O}(n^3)$.

Sei G ein gerichteter Graph, c sei konservativ und die Knoten von G seien mit $1, \ldots, n$ bezeichnet. Sei $d_{i,j}(k)$ die Länge eines kürzesten i-j-Weges, bei dem alle inneren Knoten in der Menge $\{1, \ldots, k\}$ liegen. Dann ist $d_{i,j}(n)$ der gesuchte Abstand. Dieser lässt sich rekursiv wie folgt berechnen:

$$d_{i,j}(k) = \begin{cases} 0, & \text{falls } i = j, \\ c((i,j)), & \text{falls } i \neq j,\ (i,j) \in E(G),\ k = 0, \\ \infty, & \text{falls } (i,j) \notin E(G),\ k = 0, \\ \min\{d_{i,j}(k-1), d_{i,k}(k-1) + d_{k,j}(k-1)\} & \text{sonst.} \end{cases}$$

Floyd-Warshall-Algorithmus

Input: Ein Digraph G mit $V(G) = \{1, \ldots, n\}$, konservatives $c : E(G) \to \mathbb{R}$.

Output: $\text{dist}_{(G,c)}(x, y)$ für alle $x, y \in V(G)$.

① $d_{i,j}(0) := \infty$ für alle $(i,j) \notin E(G)$;
$d_{i,j}(0) := c((i,j))$ für alle $(i,j) \in E(G)$;
$d_{i,i}(0) := 0$ für alle i;
② `for` $k := 1$ `to` n `do`
③ `for` $i := 1$ `to` n `do`
④ `for` $j := 1$ `to` n `do`
⑤ $d_{i,j}(k) = \min\{d_{i,j}(k-1), d_{i,k}(k-1) + d_{k,j}(k-1)\}$

Satz 6.30 *Der Floyd-Warshall-Algorithmus ist korrekt und kann in $\mathcal{O}(n^3)$ implementiert werden.*

Beweis. Laufzeit: klar.
Korrektheit: Dazu reicht der Nachweis der Korrektheit der Rekursionsformel. Offensichtlich ist $d_{i,j}(0)$ korrekt initialisiert. Betrachte $d_{i,j}(k)$, also die Länge eines kürzesten i-j-Weges, dessen innere Knoten alle in der Menge $\{1, \ldots, k\}$ liegen. Falls der Knoten k in einem solchen Weg nicht vorkommt, so gilt $d_{i,j}(k) = d_{i,j}(k-1)$. Falls der Knoten k vorkommt, so gilt $d_{i,j}(k) = d_{i,k}(k-1) + d_{k,j}(k-1)$, denn, falls sich kürzester i-k-Weg P und kürzester k-j-Weg Q, die jeweils nur die Knoten $\{1, \ldots, k-1\}$ im Inneren haben, in einem dieser Knoten schneiden, so enthielte $P \cup Q$ einen Kreis, der k enthält. Da der Kreis die Länge ≥ 0 hat, gäbe es einen i-j-Weg, der Knoten k nicht enthält und höchstens so lang ist. □

Bemerkung 6.31 Der Floyd-Warshall-Algorithmus lässt sich vereinfachen, indem man statt $d_{i,j}(k)$ einfach nur $d_{i,j}$ in Zeile ① und ⑥ des Algorithmus benutzt.

6.5 Netzwerkflüsse

Gegeben sei ein Digraph $G = (V, E)$, Kantenkapazitäten $u : E(G) \to \mathbb{R}$ und zwei ausgezeichnete Knoten $s \in V(G)$, die **Quelle**, und $t \in V(G)$, die **Senke**. Das Tupel (G, u, s, t) bezeichnet man als **Netzwerk**.

Ein **Fluss** in einem Netzwerk (G, u, s, t) ist eine Funktion $f : E(G) \to \mathbb{R}_+$ mit

$$f(e) \leq u(e) \quad \forall e \in E(G) \qquad \text{(Zulässigkeit)}.$$

f ist ein ***s*-*t*-Fluss**, falls

$$\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e) \quad \forall v \in V(G) \setminus \{s, t\} \qquad \text{(Flusserhaltung)}.$$

Der **Wert** eines s-t-Flusses ist definiert als

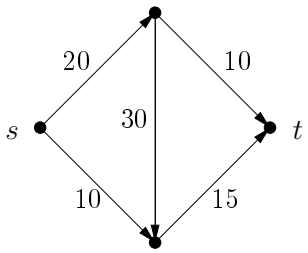
$$\text{val}(f) := \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e).$$

Maximum-Flow-Problem

Input: Ein Netzwerk (G, u, s, t) (ohne parallele Kanten).

Output: Ein s-t-Fluss maximalen Werts.

Beispiel 6.32



$\Rightarrow$ maximaler Wert eines s-t-Flusses ist 25.

Lemma 6.33 *Für jedes $A \subseteq V(G)$ mit $s \in A$, $t \notin A$ und jeden s-t-Fluss gilt:*

(a) $\text{val}(f) = \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e)$

(b) $\text{val}(f) \leq \sum_{e \in \delta^+(A)} u(e)$.

Beweis.

(a) Wegen der Flusserhaltung für $v \neq s$ gilt

$$\begin{aligned} \text{val}(f) &= \textstyle\sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \\ &= \textstyle\sum_{v \in A} \left(\sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \right) \\ &= \textstyle\sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e). \end{aligned}$$

(b) $f(e) \leq u(e)$ und $f(e) \geq 0$ für alle $e \in E(G)$.
$\Rightarrow \text{val}(f) = \sum_{e\in\delta^+(A)} f(e) - \sum_{e\in\delta^-(A)} f(e) \leq \sum_{e\in\delta^+(A)} u(e)$.

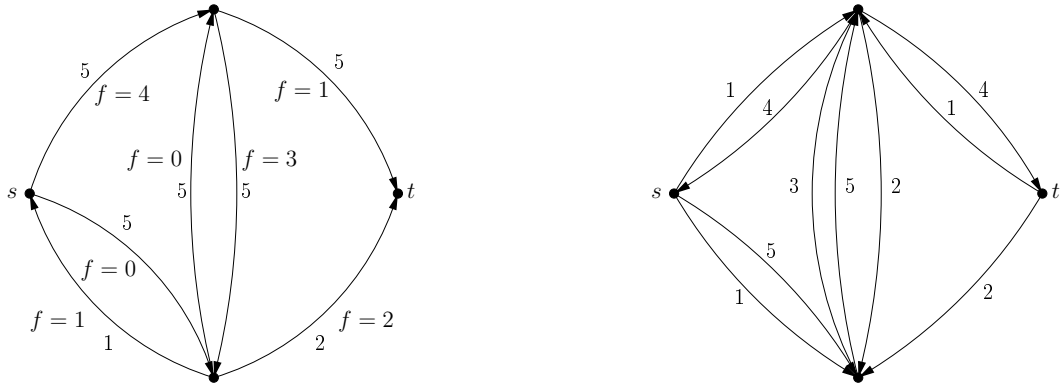
□

Sei $X \subset V(G)$ mit $s \in X$, $t \notin X$. Dann heißt $\delta^+(X)$ ein ***s-t*-Schnitt**. Die **Kapazität** eines s-t-Schnittes $\delta^+(X)$ ist als $\sum_{e\in\delta^+(X)} u(e)$ definiert. Ein **minimaler *s-t*-Schnitt** ist ein s-t-Schnitt mit minimaler Kapazität.

Sei $G = (V, E)$ ein Digraph. Für $e = (x, y) \in E(G)$ bezeichne $\overleftarrow{e}$ die **gegenläufige Kante** (y, x).

Gegeben sei ein Digraph $G = (V, E)$ und Kantenkapazitäten $u : E(G) \to \mathbb{R}_+$ und ein Fluss $f : E(G) \to \mathbb{R}_+$. Dann sind die **Restkapazitäten** u_f definiert durch $u_f(e) := u(e) - f(e)$ und $u_f(\overleftarrow{e}) := f(e)$. Der **Restgraph** G_f hat die Knotenmenge $V(G)$ und Kantenmenge $E(G_f) := \{e \in E(G) \mid u_f(e) > 0\} \dot\cup \{\overleftarrow{e} \mid e \in E(G) \text{ und } u_f(\overleftarrow{e}) > 0\}$.

Beispiel 6.34



Beachte: G_f kann parallele Kanten enthalten, die nicht vereinigt werden.

Gegeben sei ein Netzwerk (G, u, s, t) und ein s-t-Fluss f, so ist ein ***f*-augmentierender Weg** ein s-t-Weg im Restgraphen G_f. Sei P ein f-augmentierender Weg und $0 < \gamma \leq \min_{e\in E(P)} u_f(e)$. Dann verstehen wir unter der **Augmentierung von *f* entlang P um γ** das Folgende: Für jedes $e \in E(P)$ erhöhe $f(e)$ um γ, falls $e \in E(G)$ und erniedrige $f(\overleftarrow{e})$ um γ, falls $\overleftarrow{e} \in E(G)$. Beachte, dass nach Konstruktion von G_f auch bei parallelen Kanten eindeutig feststeht, ob $e \in E(G)$ oder $\overleftarrow{e} \in E(G)$. Die Augmentierung von f entlang P um γ liefert einen neuen Fluss f' mit $\text{val}(f') = \text{val}(f) + \gamma$.

Ford-Fulkerson-Algorithmus

Input: Ein Netzwerk (G, u, s, t) mit $u : E(G) \to \mathbb{Z}_+$.

Output: Ein s-t-Fluss maximalen Wertes.

① $f(e) := 0$ für alle $e \in E(G)$;
② `while` $\exists$ f-augmentierenden Weg P `do`

③ $\gamma := \min_{e \in E(P)} u_f(e)$;
④ augmentiere f entlang P um γ;

Die Korrektheit des Algorithmus ergibt sich aus dem folgenden Satz:

Satz 6.35 *Ein s-t-Fluss f hat genau dann maximalen Wert, wenn es keinen f-augmentierenden Weg gibt.*

Beweis. Falls es einen f-augmentierenden Weg gibt, so kann f entlang P zu einem Fluss mit größerem Wert augmentiert werden.
Falls es keinen f-augmentierenden Weg gibt, so ist t von s aus in G_f nicht erreichbar. Sei R die Menge der von s aus in G_f erreichbaren Knoten. Dann gilt $f(e) = u(e)$ für alle $e \in \delta_G^+(R)$ und $f(e) = 0$ für alle $e \in \delta_G^-(R)$, da sonst nach Definition von G_f eine Kante aus R herausläuft in G_f. Nach Lemma 6.33 (a) gilt:

$$\mathrm{val}(f) = \sum_{e \in \delta_+(R)} f(e) - \sum_{e \in \delta_-(R)} f(e) = \sum_{e \in \delta_+(R)} u(e).$$

$\Rightarrow$ Nach Lemma 6.33 (b) hat f maximalen Wert. □

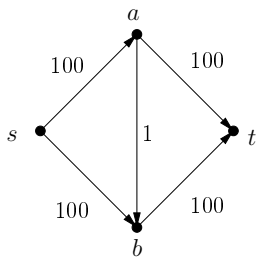
Satz 6.36 (Max-Flow-Min-Cut-Theorem) *In einem Netzwerk (G, u, s, t) stimmen der Wert eines maximalen s-t-Flusses und die Kapazität eines minimalen s-t-Schnitts überein.*

Beweis. Nach Lemma 6.33 (b) ist der Wert eines s-t-Flusses höchstens so groß wie die Kapazität eines s-t-Schnittes. Im Beweis von Satz 6.35 wurde ein Schnitt $\delta^+(R)$ definiert mit $\mathrm{val}(f) = \sum_{e \in \delta^+(R)} u(e)$. □

Satz 6.37 *Der Algorithmus von Ford-Fulkerson ist korrekt und benötigt höchstens $\mathcal{O}(m \cdot \mathrm{val}(f))$ Laufzeit.*

Beweis. Korrektheit: folgt aus Satz 6.35.
Laufzeit: In jedem Durchlauf der `while`-Schleife erhöht sich der Wert des Flusses mindestens um 1. Ein Schleifendurchlauf benötigt $\mathcal{O}(m)$. □

Beispiel 6.38



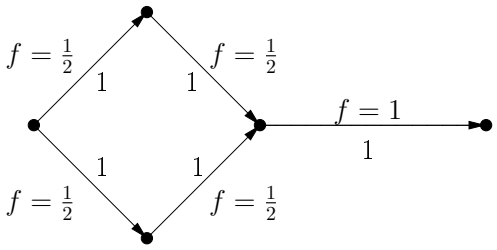
Falls augmentierende Wege s, a, b, t und s, b, a, t immer abwechselnd gewählt werden, so benötigt Ford-Fulkerson $\mathcal{O}(\mathrm{val}(f))$ Iterationen.

Was passiert bei nicht-ganzzahligen Kapazitäten beim Ford-Fulkerson-Algorithmus? Es gibt Beispiele, die zeigen, dass Ford-Fulkerson weder terminiert noch gegen das Optimum konvergiert.

Satz 6.39 *Falls alle Kapazitäten ganzzahlig sind, so existiert stets ein ganzzahliger Fluss maximalen Wertes.*

Beweis. Dies folgt aus der Korrektheit des Ford-Fulkerson-Algorithmus. □

Bemerkung 6.40 Satz 6.39 zeigt, dass es bei ganzzahligen Kapazitäten stets einen ganzzahligen Fluss mit maximalem Wert gibt. Aber nicht jeder Fluss mit maximalem Wert ist ganzzahlig:



Probleme mit dem Ford-Fulkerson-Algorithmus:
Der Algorithmus kann exponentielle Laufzeit benötigen und terminiert bei nicht-ganzzahligen Kapazitäten nicht notwendigerweise.

Edmonds-Karp-Algorithmus(Edmonds & Karp 1972)
Input: Ein Netzwerk (G, u, s, t) mit $u : E(G) \to \mathbb{R}_+$.
Output: Ein s-t-Fluss maximalen Wertes.

① $f(e) := 0$ für alle $e \in E(G)$;
② `while` $\exists$ f-augmentierenden Weg `do`
③ wähle kürzesten f-augmentierenden Weg P;
④ $\gamma := \min_{e \in E(P)} u_f(e)$;
⑤ augmentiere f entlang P um γ;

Für die Laufzeit des Ford-Fulkerson-Algorithmus benötigen wir zunächst folgendes Lemma:

Lemma 6.41 *Der Abstand* $\text{dist}_{G_f}(s, x)$ *wächst monoton mit jedem Schleifendurchlauf des Edmonds-Karps-Algorithmus für alle* $x \in V$.

Beweis. Für $x = s$ gilt dies sicherlich, sei also $x \neq s$. Angenommen es gibt einen Knoten x, sodass $\text{dist}_{G_f}(s, x)$ in einer Iteration des Algorithmus kleiner wird. Sei f der Fluss, bevor $\text{dist}_{G_f}(s, x)$ für ein $x \in V$ zum ersten Mal kleiner wird, und sei f' der Fluss eine Iteration später.
Sei x so gewählt, dass $\text{dist}_{G_{f'}}(s, x) < \text{dist}_{G_f}(s, x)$, und sei zudem $\text{dist}_{G_{f'}}(s, x)$ minimal

unter allen möglichen Wahlen von x. Sei P ein kürzester s-x-Weg in $G_{f'}$ und sei y der Vorgänger von x in P.
Dann gilt: $\text{dist}_{G_{f'}}(s,x) = \text{dist}_{G_{f'}}(s,y) + 1$. Zudem gilt nach Wahl von x:

$$\text{dist}_{G_{f'}}(s,y) \geq \text{dist}_{G_f}(s,y).$$

Wir behaupten nun, dass $(y,x) \notin E(G_f)$ gilt. Falls doch, so gälte:

$$\text{dist}_{G_f}(s,x) \leq \text{dist}_{G_f}(s,y) + 1 \leq \text{dist}_{G_{f'}}(s,y) + 1 = \text{dist}_{G_{f'}}(s,x) \quad \lightning$$

Es gilt also $(y,x) \in E(G_{f'})$ und $(y,x) \notin E(G_f)$.
Es folgt: Der Edmonds-Karp-Algorithmus muss den Fluss entlang der Kante (x,y) erhöht haben, um von f nach f' zu kommen. Edmonds-Karp wählt stets den kürzesten Weg. $\Rightarrow$ kürzester s-y-Weg in G_f enthält die Kante (x,y).

$$\begin{aligned} \Rightarrow \quad \text{dist}_{G_f}(s,x) &= \text{dist}_{G_f}(s,y) - 1 \\ &\leq \text{dist}_{G_{f'}}(s,y) - 1 \\ &= \text{dist}_{G_{f'}}(s,y) - 2 \end{aligned}$$

Dies ist ein Widerspruch zu $\text{dist}_{G_{f'}}(s,x) < \text{dist}_{G_f}(s,x)$. $\Rightarrow x$ existiert nicht. □

Satz 6.42 *Der Edmonds-Karp-Algorithmus ist korrekt und kann in $\mathcal{O}(n \cdot m^2)$ implementiert werden.*

Beweis. Laufzeit: Jeder Durchlauf der `while`-Schleife kann in $\mathcal{O}(m)$ mittels Breitensuche implementiert werden. Es reicht daher zu zeigen, dass es höchstens $\mathcal{O}(n \cdot m)$ Augmentierungsschritte gibt.

Eine Kante $(x,y) \in E(P)$ eines s-t-Weges in G_f heißt **kritisch**, falls $u_f((x,y)) = \min_{e \in E(P)} u_f(e)$.

Beobachtung: Nach Augmentierung von P wird jede kritische Kante in P aus G_f entfernt. Außerdem wird in jedem Augmentierungsschritt mindestens eine kritische Kante entfernt.

Sei (x,y) eine Kante, die kritisch ist in G_f. Da (x,y) auf einem kürzesten s-t-Weg in G_f liegt, gilt $\text{dist}_{G_f}(s,y) = \text{dist}_{G_f}(s,x) + 1$. Die Kante (x,y) wird aus G_f entfernt. Wann kann sie in einem späteren Schritt wieder auftreten? Nur falls (y,x) in einem augmentierenden Weg auftritt. Sei f' der Fluss zu diesem Zeitpunkt. Dann gilt

$$\text{dist}_{G_{f'}}(s,x) = \text{dist}_{G_{f'}}(s,y) + 1.$$

Da $\text{dist}_{G_f}(s,y) \leq \text{dist}_{G_{f'}}(s,y)$ nach Lemma 6.41, folgt:

$$\text{dist}_{G_{f'}}(s,x) = \text{dist}_{G_{f'}}(s,y) + 1 \geq \text{dist}_{G_f}(s,y) + 1 = \text{dist}_{G_f}(s,x) + 2.$$

$\Rightarrow$ Falls eine Kante (x, y) mehrmals kritisch ist, so erhöht sich der Abstand von s nach x jedesmal. Da der Abstand maximal $n-1$ sein kann, kann eine Kante maximal $\mathcal{O}(n)$ mal kritisch sein. Es gibt $\mathcal{O}(m)$ Kanten, die kritisch werden können. In jedem Schritt ist mindestens eine Kante kritisch. $\Rightarrow \mathcal{O}(n \cdot m)$ Schritte maximal.

Die **Korrektheit** folgt, da maximal $\mathcal{O}(n \cdot m)$ Augmentierungsschritte erfolgen und bei Terminierung des Algorithmus mit Fluss f kein s-t-Weg im Restgraphen G_f existiert und f nach Satz 6.35 somit Fluss maximalen Werts ist. □

6.6 Matchings in bipartiten Graphen

Sei $G = (V, E)$ ein ungerichteter Graph. Ein Teilmenge $M \subseteq E$ heißt **Matching**, falls für alle $e \neq f$, $e, f \in M$ gilt: $e \cap f = \emptyset$. M ist **maximales Matching**, falls $M \cup \{e\}$ kein Matching für alle $e \in E \setminus M$ ist. M ist **größtes Matching**, falls $|M| \geq |M'|$ für alle Matchings M'.

Größte-Matching-Problem
Input: $G = (V, E)$.
Output: Größtes Matching M.

Satz 6.43 *Ein maximales Matching kann in $\mathcal{O}(m)$ berechnet werden.*

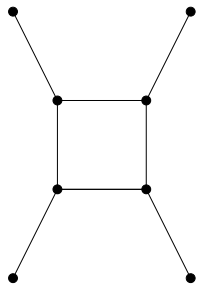
Beweis. Folgender Algorithmus liefert das Gewünschte:
$M := \emptyset$
`while` $\exists e \in E \setminus M$ mit $M \cup \{e\}$ ist Matching
    $M := M \cup \{e\}$ □

Satz 6.44 *Ist M ein maximales Matching und M^* ein größtes Matching, so gilt*

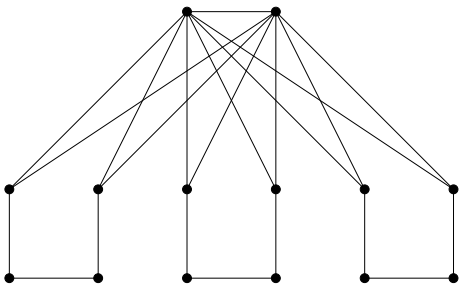
$$|M| \geq \frac{1}{2}|M^*|.$$

Beweis. Sei $S := \{x \in V \mid \exists e \in M \text{ mit } x \in e\}$. Dann gilt $|S| = 2|M|$. Für $e \in M^*$ gilt $e \cap S \neq \emptyset$ und $e \neq f$, $e, f \in M^*$. Somit ist $e \cap f = \emptyset$.
$\Rightarrow |M^*| \leq |S| = 2|M|$. □

Die Schranke in Satz 6.44 ist bestmöglich, wie folgende Beispiele zeigen:



$|M| = 2$, $|M^*| = 4$



$|M| = 4$, $|M^*| = 6$

Satz 6.45 *In bipartiten Graphen kann ein größtes Matching in $\mathcal{O}(n \cdot m)$ bestimmt werden.*

Beweis. Sei $G = (A \dot\cup B, E)$ ein bipartiter Graph. Dann konstruieren wir ein Netzwerk (G', u, s, t) aus G wie folgt:
$V(G') := V(G) \dot\cup \{s, t\}$,
$E(G') := \{(x, y) \mid x \in A, y \in B, \{x, y\} \in E(G)\} \cup \{(s, x) \mid x \in A\} \cup \{(x, t) \mid t \in B\}$,
$u(e) := 1$ für alle $e \in E(G')$.
Sei f ein s-t-Fluss maximalen Werts in (G', u, s, t) und M ein größtes Matching in G.
Behauptung: Es gilt $|M| = \text{val}(f)$.

- $|M| \geq \text{val}(f)$:
  Wir wissen nach Satz 6.39, dass f als ganzzahlig angenommen werden kann, das heißt also $f(e)$ ist entweder 0 oder 1 für jede Kante.
  Betrachte alle Kanten $(x, y) \in E(G')$ mit $f((x, y)) = 1$. Dann bilden die entsprechenden Kanten $\{\{x, y\} \mid f(x, y) = 1\}$ ein Matching in G, das $\text{val}(f)$ viele Kanten enthält.
- $|M| \leq \text{val}(f)$:
  Für $\{x, y\} \in M$ mit $x \in A$ und $y \in B$ definiere Fluss $\tilde f$ durch $\tilde f((s, x)) = \tilde f((x, y)) = \tilde f((y, t)) = 1$.
  $\Rightarrow \text{val}(\tilde f) = |M|$. $\Rightarrow \text{val}(f) \leq |M|$.

Es reicht also, einen s-t-Fluss maximalen Wertes in (G', u, s, t) auszurechnen. Dieser kann mit dem Algorithmus von Ford-Fulkerson in $\mathcal{O}(m \cdot \text{val}(f)) = \mathcal{O}(n \cdot m)$ berechnet werden. □

Ein Matching M in einem Graphen $G = (V, E)$ heißt **perfekt**, falls $|M| = \frac{1}{2}|V|$.

Satz 6.46 *Ein bipartiter Graph $G = (A \dot\cup B, E)$ mit $|A| = |B|$ besitzt genau dann ein perfektes Matching, wenn $|N(S)| \geq |S|$ gilt für alle $S \subseteq A$.*

Beweis. Falls G ein perfektes Matching besitzt, so zeigen die Matchingkanten, dass $|N(S)| \geq |S|$ für alle $S \subseteq A$ gilt.
Zum Nachweis der anderen Richtung führen wir eine Induktion über $|A|$. Falls $|A| = 1$, so gilt die Aussage offensichtlich. Sei also $|A| \geq 2$. Falls $|N(S)| \geq |S| + 1$ für alle $S \subseteq A$

mit $|S| < |A|$ gilt, so sei $\{x, y\}$ eine beliebige Kante in G. Nach Induktionsvoraussetzung besitzt der Graph $G \setminus \{x, y\}$ ein Matching, das zusammen mit der Kante $\{x, y\}$ ein perfektes Matching in G bildet.
Wir können nun also davon ausgehen, dass es ein $S \subset A$ mit $|N(S)| = |S|$ und $|S| < |A|$ gibt. Nach Induktionsvoraussetzung gibt es dann in dem von $s \cup N(S)$ induzierten Graphen ein perfektes Matching. Betrachte nun den Graphen $G' := G \setminus (S \cup N(S))$. Dieser erfüllt die Voraussetzungen des Satzes, denn gäbe es eine Menge $T \subseteq A \setminus S$ mit $|N_{G'}(T)| < |T|$, so gälte $|N_G(T \cup S)| < |T| + |S|$. Somit gibt es nach Induktionsvoraussetzung in G' ein perfektes Matching und insgesamt damit auch eines in G. □

Korollar 6.47 *Ein k-regulärer bipartiter Graph besitzt ein perfektes Matching.*

Beweis. Ein k-regulärer bipartiter Graph erfüllt die Voraussetzungen von Satz 6.46. □

7 Lineare Gleichungssysteme

Wir beschäftigen uns in diesem Kapitel mit der numerischen Lösung linearer Gleichungssysteme, das heißt Gleichungssysteme der Form:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ \vdots & & & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n \end{array}$$

oder kompakt geschrieben: $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$ und $\boldsymbol{A} = (a_{ij}) \in \mathbb{R}^{n \times n}$, wobei $\boldsymbol{x} = (x_1, \dots, x_n)^T$ und $\boldsymbol{b} = (b_1, \dots, b_n)^T$.

7.1 Der Gauß'sche Algorithmus

Eine Matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ heißt **obere Dreiecksmatrix**, falls $a_{ij} = 0$ für alle $i > j$ gilt, das heißt, falls $\boldsymbol{A}$ die Form

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & & & & \\ & \ddots & & a_{ij} & \\ & & \ddots & & \\ & 0 & & \ddots & \\ & & & & a_{nn} \end{pmatrix}$$

hat. $\boldsymbol{A}$ heißt **untere Dreiecksmatrix**, falls $a_{ij} = 0$ für alle $i < j$ gilt. Eine Dreiecksmatrix heißt **normiert**, falls $a_{ii} = 1$ für alle i gilt.

Satz 7.1 *Sei $\boldsymbol{b} \in \mathbb{R}^n$ und $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ obere Dreiecksmatrix mit $a_{ii} \neq 0$ für alle i. Dann lässt sich die Lösung $\boldsymbol{x} \in \mathbb{R}^n$ des linearen Gleichungssystems $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ sukzessive berechnen durch*

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{k=i+1}^{n} a_{ik} x_k \right) \qquad i = n, \dots, 1.$$

Beweis. Durch Umformen erhält man

$$x_i a_{ii} + \sum_{k=i+1}^{n} a_{ik} x_k = b_1,$$

was genau der i-ten Zeile des linearen Gleichungssystems $A \cdot x = b$ entspricht. □

Dieses Vorgehen nennt man auch **Lösen durch Rückwärtseinsetzen**.

Die Matrix $\boldsymbol{I} \in \mathbb{R}^{n\times n}$ mit $\boldsymbol{I} = (\delta_{ij})_{i,j=1,\dots,n}$ heißt **Einheitsmatrix**. Eine Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ heißt **regulär**, falls es eine Matrix $\boldsymbol{A}^{-1} \in \mathbb{R}^{n\times n}$ gibt mit $\boldsymbol{A}\cdot\boldsymbol{A}^{-1} = \boldsymbol{I}$. Andernfalls heißt $\boldsymbol{A}$ **singulär**.

Es sei $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$ ein lineares Gleichungssystem. Dann ändern die folgenden Operationen die Lösungsmenge nicht:

(1) Vertauschen von Zeilen

(2) Multiplikation einer Zeile mit einer Zahl $\neq 0$

(3) Addition eines Vielfachen einer Zeile zu einer anderen

Der Gauß'sche Algorithmus nutzt diese drei Operationen, um ein gegebenes lineares Gleichungssysteme so umzuformen, dass Satz 7.1 angewendet werden kann.

Gauß'scher Algorithmus

Input: Reguläre Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{b} \in \mathbb{R}^n$.

Output: $\boldsymbol{x} \in \mathbb{R}^n$ mit $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$.

① for $k := 1$ `to` n `do`
② bestimme $l \geq k$, sodass $a_{lk} \neq 0$;
③ vertausche Zeilen l und k;
④ `for` $j := k+1$ `to` n `do`
⑤ $l_{jk} := \frac{a_{jk}}{a_{kk}}$;
⑥ $b_j := b_j - l_{jk}b_k$;
⑦ `for` $i := k$ `to` n `do`
⑧ $a_{ji} := a_{ji} - l_{jk}a_{ki}$;
⑨ for $i := 1$ `to` n `do`
⑩ $x_i := \frac{1}{a_{ii}}\left(b_i - \sum_{k=i+1}^{n} a_{ik}x_k\right)$;

Satz 7.2 *Der Gauß'sche Algorithmus ist korrekt und berechnet eine Lösung in* $\mathcal{O}(n^3)$.

Beweis. Laufzeit: Klar, da drei ineinander geschachtelte `for`-Schleifen durchlaufen werden.

Korrektheit: Wir zeigen zunächst, dass in den Zeilen ③-⑧ das lineare Gleichungssystem auf obere Dreiecksgestalt gebracht wird. In den Zeilen ③-⑧ werden nur die Operationen (1) und (3) angewandt. Das entstehende lineare Gleichungssystem hat also die gleiche Lösungsmenge wie $A \cdot x = b$. Wir zeigen nun, dass am Ende von Zeile ⑧ die Matrix $\boldsymbol{A}$ obere Dreiecksmatrix ist.

Sei $j > i$. Der Wert von a_{ji} wird letztmalig geändert, wenn $k = i$ ist. Dann gilt:

$$a_{ji} - l_{jk}a_{ki} = a_{ji} - \frac{a_{jk}}{a_{kk}}a_{ki} = a_{ji} - \frac{a_{ji}}{a_{ii}}a_{ii} = 0.$$

Es bleibt zu zeigen, dass für reguläres $\boldsymbol{A}$ in Zeile ② stets ein $a_{lk} \neq 0$ gefunden werden kann. Es gilt: $\boldsymbol{A}$ ist regulär genau dann, wenn $\det(\boldsymbol{A}) \neq 0$ ist. Somit wird für $k = 1$ ein $a_{l1} \neq 0$ gefunden, da sonst die erste Spalte von $\boldsymbol{A}$ der Nullvektor wäre und somit $\det(\boldsymbol{A}) = 0$ gälte. Das Vertauschen zweier Zeilen ändert das Vorzeichen der Determinante, während Operation (3) den Wert der Determinante nicht ändert. Folglich hat $\boldsymbol{A}$ nach einem Schleifendurchlauf die Gestalt

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & & & \\ \vdots & & \boldsymbol{A}' & \\ 0 & & & \end{pmatrix}$$

und es gilt $\det(\boldsymbol{A}) = \pm a_{11} \det(\boldsymbol{A}')$. Per Induktion folgt die Aussage für $\boldsymbol{A}'$. □

Korollar 7.3 *Die Determinante einer Matrix lässt sich in $\mathcal{O}(n^3)$ berechnen.*

Beweis. Falls die Matrix regulär ist, so gilt, wie im Beweis von Satz 7.2 gesehen, dass $\det(\boldsymbol{A}) = \pm \prod_{i=1}^{n} a'_{ii}$ ist, wobei a'_{ii} die Einträge bei Beendigung des Gauß'schen Algorithmus sind. Ein Vorzeichenwechsel tritt bei jeder Ausführung von Zeile ③ mit $l \neq k$ auf.
Ist die Matrix nicht regulär, folgt $\det(\boldsymbol{A}) = 0$. □

Beispiel 7.4

$$\begin{array}{rcrcrcr} 3x_1 & + & x_2 & + & 8x_3 & = & 11 \\ 6x_1 & + & 2x_2 & + & 5x_3 & = & 0 \\ 9x_1 & + & 4x_2 & + & 7x_3 & = & 0 \end{array}$$

$$\left(\begin{array}{ccc|c} 3 & 1 & 8 & 11 \\ 6 & 2 & 5 & 0 \\ 9 & 4 & 7 & 0 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 3 & 1 & 8 & 11 \\ 0 & 0 & -11 & -22 \\ 0 & 1 & -17 & -33 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 3 & 1 & 8 & 11 \\ 0 & 1 & -17 & -33 \\ 0 & 0 & -11 & -22 \end{array}\right)$$

$\Rightarrow \boldsymbol{x} = (-2, 1, 2)^T$ und $\det(\boldsymbol{A}) = 33$.

Es bezeichne im Folgenden $a_{ji}^{(k)}$ den Wert von a_{ji} nach k Durchläufen der `for`-Schleife in Zeile ① des Gauß'schen Algorithmus. Dann gilt gemäß Zeile ⑧ $a_{ji}^{(k)} = a_{ji}^{(k-1)} - l_{jk} a_{ki}^{(k-1)}$. Zeile ⑧ wird nur für $j > k$ durchlaufen. Somit gilt $a_{ji}^{(n)} = a_{ji}^{(n-1)} = \ldots = a_{ji}^{(j-1)}$. Sei $r_{ji} := a_{ji}^{(n)}$, das heißt $(r_{ji})_{j,i=1,\ldots,n}$ sei die Matrix bei Beendigung des Gauß'schen Algorithmus. Dann gilt:

$$\begin{aligned} r_{ji} = a_{ji}^{(j-1)} &= a_{ji}^{(j-2)} - l_{jj-1} a_{j-1i}^{(j-2)} \\ &= a_{ji}^{(j-3)} - l_{jj-2} a_{j-2i}^{(j-3)} - l_{jj-1} a_{j-1i}^{(j-2)} \\ &\vdots \\ &= a_{ji} - \sum_{k=1}^{j-1} l_{jk} a_{ki}^{(k-1)} = a_{ji} - \sum_{k=1}^{j-1} l_{jk} r_{ki} \end{aligned}$$

Man erhält somit

$$a_{ji} = r_{ji} - \sum_{k=1}^{j-1} l_{jk} r_{ki} \tag{7.1}$$

$(r_{ji})_{j,i=1,\ldots,n}$ ist obere Dreiecksmatrix. $\Rightarrow r_{ji} = 0$, falls $j > i$.
$\Rightarrow$ für $j > i$ gilt

$$a_{ji} = \sum_{k=1}^{i} l_{jk} r_{li}. \tag{7.2}$$

Sei $\boldsymbol{L} = \begin{pmatrix} 1 & & & \\ & \ddots & & 0 \\ & & \ddots & \\ & l_{ij} & & \ddots \\ & & & 1 \end{pmatrix}$ und $\boldsymbol{R} = \begin{pmatrix} r_{11} & & & & \\ & \ddots & & r_{ij} & \\ & & \ddots & & \\ & 0 & & \ddots & \\ & & & & r_{nn} \end{pmatrix}$.

Satz 7.5 *Falls beim Gauß'schen Algorithmus keine Zeilenvertauschungen stattfinden, so liefert der Algorithmus eine Zerlegung $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{R}$.*

Beweis. Ausrechnen von $\boldsymbol{L} \cdot \boldsymbol{R}$ ergibt:

$$\begin{aligned} (\boldsymbol{L} \cdot \boldsymbol{R})_{ij} &= (l_{i1}, l_{i2}, \ldots, l_{ii-1}, 1, 0, \ldots, 0) \cdot \begin{pmatrix} r_{1j} \\ r_{2j} \\ \vdots \\ r_j \\ 0 \\ \vdots \\ 0 \end{pmatrix} \\ &= \begin{cases} \sum_{k=1}^{i-1} l_{ik} r_{kj} + r_{ij}, & \text{falls } j \geq i \geq 1 \\ \sum_{k=1}^{j} l_{ik} r_{kj}, & \text{falls } i > j \geq 1. \end{cases} \end{aligned}$$

Ein Vergleich mit (7.1) und (7.2) zeigt die Behauptung. □

Eine Zerlegung $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{R}$, wobei $\boldsymbol{L}$-normierte untere Dreiecksmatrix und $\boldsymbol{R}$ obere Dreiecksmatrix sind, nennt man **$\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung** (im Englischen **$\boldsymbol{L}$-$\boldsymbol{U}$**-Zerlegung).

Bemerkung Die $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung ist eindeutig.

Satz 7.6 *Ist eine $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{R}$ gegeben, so lässt sich das lineare Gleichungssystem $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ in $\mathcal{O}(n^2)$ lösen.*

Beweis. Sei $b_j^{(k)}$ der Wert von b_j nach k Iterationen. Setze $c_j := b_j^{(n)}$. Dann gilt

$$\begin{aligned} c_j = b_j^{(j-1)} &= b_j^{(j-2)} - l_{jj-1} b_{j-1}^{(j-2)} \\ &= b_j^{(j-3)} - l_{jj-2} b_{j-2}^{(j-3)} - l_{jj-1} b_{j-1}^{(j-2)} \\ &\vdots \\ &= b_j^{(0)} - \sum_{i=1}^{j-1} l_{ji} c_i \end{aligned}$$

$\Rightarrow b_j = c_j + \sum_{i=1}^{j-1} l_{ji} c_i$.
$\Rightarrow \boldsymbol{L} \cdot \boldsymbol{c} = \boldsymbol{b}$.
Zudem gilt $\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{c}$. $\Rightarrow$ Löse zunächst in $\mathcal{O}(n^2)$ $\boldsymbol{L} \cdot \boldsymbol{c} = \boldsymbol{b}$ nach $\boldsymbol{c}$ auf und löse danach in $\mathcal{O}(n^2)$ $\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{c}$. □

Nicht jede reguläre Matrix besitzt eine $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung, wie zum Beispiel $\boldsymbol{A} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ zeigt.
Eine **Permutationsmatrix** $\boldsymbol{P} \in \mathbb{R}^{n \times n}$ zu einer Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ hat die Form

$$\boldsymbol{P} = \begin{pmatrix} \boldsymbol{e}_{\pi(1)}^T \\ \vdots \\ \boldsymbol{e}_{\pi(n)}^T \end{pmatrix},$$

wobei $\boldsymbol{e}_{\pi(i)}^T$ die Transponierte des $\pi(i)$-ten Einheitsvektors ist.

Beispiel 7.7 $\pi = (3, 1, 2, 4)$. Dann ist $\boldsymbol{P} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Lemma 7.8 *Sei $\boldsymbol{P} \in \mathbb{R}^{n \times n}$ die zu einer Permutation π gehörige Permutationsmatrix. Ist $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, so entsteht $\boldsymbol{P} \cdot \boldsymbol{A}$ aus $\boldsymbol{A}$, indem man die Zeilen von $\boldsymbol{A}$ entsprechend π permutiert.*

Beweis. Sei $\boldsymbol{A} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)$, wobei $\boldsymbol{a}_i \in \mathbb{R}^n$ die i-te Spalte von $\boldsymbol{A}$ bezeichnet. Dann gilt für die k-te Zeile von $\boldsymbol{P} \cdot \boldsymbol{A}$:

$$\begin{aligned} (\boldsymbol{e}_{\pi(k)} \boldsymbol{a}_1, \ldots, \boldsymbol{e}_{\pi(k)} \boldsymbol{a}_n) &= (\boldsymbol{a}_{\pi(k),1}, ldots, \boldsymbol{a}_{\pi(k),n}) \\ &= \pi(k)\text{-te Zeile von } \boldsymbol{A} \end{aligned}$$

□

Der Gauß'sche Algorithmus liefert die Zerlegung $\boldsymbol{P} \cdot \boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{R}$. Löse $\boldsymbol{L} \cdot \boldsymbol{c} = \boldsymbol{P} \cdot \boldsymbol{b}$ und $\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{c}$.

Korollar 7.9 *Sei $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ regulär. Dann lässt sich $\boldsymbol{A}^{-1}$ in $\mathcal{O}(n^3)$ berechnen.*

Beweis. $\boldsymbol{A} \cdot \boldsymbol{A}^{-1} = \boldsymbol{I}$. Somit sind die Spalten von $\boldsymbol{A}^{-1}$ Lösungen der n linearen Gleichungssysteme $\boldsymbol{A} \cdot (\boldsymbol{a}_i^{-1}) = \boldsymbol{e}_i$. Bestimme die $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung $\boldsymbol{P} \cdot \boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{R}$ in $\mathcal{O}(n^3)$ und löse $\boldsymbol{L} \cdot \boldsymbol{c} = \boldsymbol{P} \cdot \boldsymbol{e}_i$ und $\boldsymbol{R} \cdot (\boldsymbol{a}_i^{-1}) = \boldsymbol{c}$ für $i = 1, \ldots, n$.
$\Rightarrow$ die Laufzeit beträgt $n \cdot \mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$. □

7.2 Vektor- und Matrixnormen

Eine numerisch berechnete Lösung $\widetilde{\boldsymbol{x}}$ eines linearen Gleichungssystems $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ weist einen Fehler $\boldsymbol{x} - \widetilde{\boldsymbol{x}}$ auf. Wir benötigen ein Maß für die Größe eines Vektors beziehungsweise einer Matrix, um Aussagen über den Fehler treffen zu können.

Sei V ein Vektorraum über $\mathbb{R}$ (zum Beispiel $V = \mathbb{R}^n$ oder $V = \mathbb{R}^{n \times n}$). Eine Abbildung $\|\cdot\| : V \to \mathbb{R}$ heißt **Norm** auf V, falls gilt:

1. $\|\boldsymbol{x}\| > 0$ für alle $\boldsymbol{x} \neq 0$,
2. $\|\alpha \boldsymbol{x}\| = |\alpha| \|\boldsymbol{x}\|$ für alle $\alpha \in \mathbb{R}$ und alle $\boldsymbol{x} \in V$,
3. $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ für alle $\boldsymbol{x}, \boldsymbol{y} \in V$ (Dreiecksungleichung).

Beispiele für Vektornormen ($\boldsymbol{x} \in \mathbb{R}^n$)

- $\|\boldsymbol{x}\|_1 := \sum_{i=1}^n |x_i|$ (Betragssummennorm, L_1-Norm, Manhattennorm)
- $\|\boldsymbol{x}\|_2 := \sqrt{\sum_{i=1}^n x_i^2}$ (euklidische Norm, L_2-Norm)
- $\|\boldsymbol{x}\|_p := (\sum_{i=1}^n x_i^p)^{1/p}$ (L_p-Norm, für $p \in \mathbb{R}_+$)
- $\|\boldsymbol{x}\|_\infty := \max_{1 \leq i \leq n} |x_i|$ (Maximumnorm)

Beispiele für Matrixnormen ($\boldsymbol{A} \in \mathbb{R}^{n \times n}$)

- $\|\boldsymbol{A}\|_1 := \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|$ (Spaltensummennorm)
- $\|\boldsymbol{A}\|_\infty := \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$ (Zeilensummennorm)
- $\|\boldsymbol{A}\|_F := \sqrt{\sum_{i,j=1}^n a_{ij}^2}$ (Frobeniusnorm)

Sei V ein Vektorraum über $\mathbb{R}$. Zwei Normen $\|\cdot\|_a : V \to \mathbb{R}$ und $\|\cdot\|_b : V \to \mathbb{R}$ heißen **äquivalent**, wenn es Konstanten $c, C \in \mathbb{R}_{>0}$ gibt mit

$$c\|\boldsymbol{x}\|_a \leq \|\boldsymbol{x}\|_b \leq C\|\boldsymbol{x}\|_a \qquad \forall \boldsymbol{x} \in V.$$

Satz 7.10 *Alle Normen des $\mathbb{R}^n$ sind äquivalent.*

Beweis. Wir zeigen, dass jede Norm $\|\cdot\|$ des $\mathbb{R}^n$ äquivalent zur L_1-Norm ist. Für $\boldsymbol{x} \in \mathbb{R}^n$ gilt

$$\|\boldsymbol{x}\| = \left\| \sum_{i=1}^n x_i \boldsymbol{e}_i \right\| \leq \sum_{i=1}^n \|x_i \boldsymbol{e}_i\| = \sum_{i=1}^n |x_i| \|\boldsymbol{e}_i\| \leq \max_{1 \leq i \leq n} \{\|\boldsymbol{e}_i\|\} \|\boldsymbol{x}\|_1.$$

Setze $M := \max_{1\le i\le n}\{\|\boldsymbol{e}_i\|\}$. Dann gilt

$$|\|\boldsymbol{x}\| - \|\boldsymbol{y}\|| \le \|\boldsymbol{x}-\boldsymbol{y}\| \le M\|\boldsymbol{x}-\boldsymbol{y}\|_1.$$

Somit ist $\|\cdot\|$ bezüglich der L_1-Norm eine stetige Funktion und nimmt daher nach dem Satz von Weierstraß auf der kompakten Menge $K := \{\boldsymbol{x}\in\mathbb{R}^n \mid \|\boldsymbol{x}\|_1 = 1\}$ ihr Minimum c und ihr Maximum C an.
Somit gilt für alle $\boldsymbol{x}\in\mathbb{R}^n\setminus\{0\}$ wegen $\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_1}\in K$:

$$c \le \left\|\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_1}\right\| \le C \qquad \Rightarrow \qquad c\|\boldsymbol{x}\|_1 \le \|\boldsymbol{x}\| \le C\|\boldsymbol{x}\|_1.$$

Folglich sind $\|\cdot\|$ und $\|\cdot\|_1$ äquivalente Normen. □

Beispiel 7.11 Es gilt

$$\|\boldsymbol{x}\|_2 \le \max_{1\le i\le n}\{\|\boldsymbol{e}_i\|\}\|x\|_1 = \|\boldsymbol{x}\|_1$$

und

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^n |x_i| \le n\max_{1\le i\le n}\{|x_i|\} \le n\sqrt{\sum_{i=1}^n x_i^2} = n\|\boldsymbol{x}\|_2.$$

$\Rightarrow \|\boldsymbol{x}\|_2 \le \|\boldsymbol{x}\|_1 \le n\|\boldsymbol{x}\|_2$.

Eine Matrixnorm heißt **submultiplikativ**, falls für alle $\boldsymbol{A},\boldsymbol{B}\in\mathbb{R}^{n\times n}$ gilt

$$\|\boldsymbol{A}\cdot\boldsymbol{B}\| \le \|\boldsymbol{A}\|\cdot\|\boldsymbol{B}\|.$$

Beispiel 7.12 Für die Spaltensummennorm $\|\cdot\|_1$ gilt

$$\begin{aligned}
\|\boldsymbol{A}\cdot\boldsymbol{B}\|_1 &= \max_{1\le j\le n}\sum_{i=1}^n |c_{ij}| \\
&= \max_{1\le j\le n}\sum_{i=1}^n \left|\sum_{k=1}^n a_{ik}b_{kj}\right| \\
&\le \max_{1\le j\le n}\sum_{k=1}^n\sum_{i=1}^n |a_{ik}||b_{kj}| \\
&= \max_{1\le j\le n}\sum_{k=1}^n |b_{kj}|\sum_{i=1}^n |a_{ik}| \\
&= \sum_{i=1}^n |a_{ik}|\|\boldsymbol{B}\|_1 \le \|\boldsymbol{A}\|_1\|\boldsymbol{B}\|_1.
\end{aligned}$$

Die Matrixnorm $\|\boldsymbol{A}\| := \max_{1\le i,j,\le n} |a_{ij}|$ ist nicht submultiplikativ, denn für $\boldsymbol{A} := \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \boldsymbol{B} := \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ ist $\|\boldsymbol{A}\| = \|\boldsymbol{B}\| = 1$, aber $\|\boldsymbol{A} \cdot \boldsymbol{B}\| = \left\| \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix} \right\| = 2 > 1$.
Eine Matrixnorm $\|\cdot\|_M$ auf $\mathbb{R}^{n\times n}$ heißt **verträglich** mit einer Vektornorm $\|\cdot\|_V$ auf $\mathbb{R}^n$, falls für alle $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ und alle $\boldsymbol{x} \in \mathbb{R}^n$

$$\|\boldsymbol{A} \cdot \boldsymbol{x}\|_V \le \|\boldsymbol{A}\|_M \|\boldsymbol{x}\|_V.$$

Beispiel 7.13 Die Frobeniusnorm ist mit der euklidische Norm verträglich, denn es gilt:

$$\begin{aligned}
\|\boldsymbol{A} \cdot \boldsymbol{x}\|_2 &= \left(\sum_{i=1}^{n} \left(\sum_{k=1}^{n} a_{ik} x_k \right)^2 \right)^{1/2} \\
&\le \left(\sum_{i=1}^{n} \left(\sum_{k=1}^{n} a_{ik}^2 \right) \left(\sum_{k=1}^{n} x_k^2 \right) \right)^{1/2} \qquad \text{(Cauchy-Schwarz)} \\
&= \left(\sum_{i=1}^{n} \sum_{k=1}^{n} a_{ik}^2 \right)^{1/2} \left(\sum_{k=1}^{n} x_k^2 \right)^{1/2} \\
&= \|\boldsymbol{A}\|_F \|\boldsymbol{x}\|_2.
\end{aligned}$$

Sei $\|\cdot\|$ eine Vektornorm auf $\mathbb{R}^n$. Dann definiert

$$|||\boldsymbol{A}||| := \max_{\boldsymbol{x}\neq 0} \frac{\|\boldsymbol{A} \cdot \boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \max_{\|\boldsymbol{x}\|=1} \|\boldsymbol{A} \cdot \boldsymbol{x}\|$$

die von $\|\cdot\|$ **induzierte Norm** auf $\mathbb{R}^{n\times n}$.
Durch Nachrechnen bestätigt man, dass $|||\cdot|||$ eine Matrixnorm ist

1. Falls $\boldsymbol{A} \neq 0$ folgt: Es gibt $a_{ij} \neq 0$. $\Rightarrow \boldsymbol{A} \cdot \boldsymbol{e}_j \neq 0 \Rightarrow \|\boldsymbol{A}\boldsymbol{e}_j\| > 0$, $\|\boldsymbol{e}_j\| > 0$
   $\Rightarrow \max_{\boldsymbol{x}\neq 0} \frac{\|\boldsymbol{A}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} > 0$.
2. $|||\alpha\boldsymbol{A}||| = \max_{\|\boldsymbol{x}\|=1} \|\alpha\boldsymbol{A} \cdot \boldsymbol{x}\| = |\alpha| \max_{\|\boldsymbol{x}\|=1} \|\boldsymbol{A} \cdot \boldsymbol{x}\| = |\alpha| \, |||\boldsymbol{A}|||$.
3. $$\begin{aligned}
   |||\boldsymbol{A} + \boldsymbol{B}||| &= \max_{\|\boldsymbol{x}\|=1} \|(\boldsymbol{A} + \boldsymbol{B}) \cdot \boldsymbol{x}\| \\
   &\le \max_{\|\boldsymbol{x}\|=1} \{\|\boldsymbol{A} \cdot \boldsymbol{x}\| + \|\boldsymbol{B} + \boldsymbol{x}\|\} \\
   &= \max_{\|\boldsymbol{x}\|=1} \|\boldsymbol{A} \cdot \boldsymbol{x}\| + \max_{\|\boldsymbol{x}\|=1} \|\boldsymbol{B} \cdot \boldsymbol{x}\| \\
   &= |||\boldsymbol{A}\,||| + |||\,\boldsymbol{B}|||
   \end{aligned}$$

Es gilt zudem die Submultiplikativität.

Beispiel 7.14 Was ist die von der Maximumnorm $\|\cdot\|_\infty$ induzierte Norm?
In diesem Fall gilt für $\boldsymbol{A} \in \mathbb{R}^{n\times n}$

$$\begin{aligned}
|||\boldsymbol{A}||| &= \max_{\|\boldsymbol{x}\|=1} \|\boldsymbol{A}\cdot\boldsymbol{x}\|_\infty \\
&= \max_{\|\boldsymbol{x}\|=1} \left\{ \max_i \left| \sum_{k=1}^{n} a_{ik}x_k \right| \right\} \\
&= \max_i \left\{ \max_{\|\boldsymbol{x}\|=1} \left| \sum_{k=1}^{n} a_{ik}x_k \right| \right\} \\
&= \max_i \sum_{k=1}^{n} |a_{ik}| \\
&= \|\boldsymbol{A}\|_\infty
\end{aligned}$$

Das heißt, die von der Maximumnorm induzierte Norm ist die Zeilensummennorm.

Ist $\|\cdot\|$ eine Vektornorm, $\boldsymbol{x} \in \mathbb{R}^n$ und $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, so gilt $\|\boldsymbol{A}\cdot\boldsymbol{x}\| \leq \max_{\boldsymbol{y}\neq 0} \frac{\|\boldsymbol{A}\cdot\boldsymbol{y}\|}{\|\boldsymbol{y}\|}\cdot\|\boldsymbol{x}\|$. Die induzierte Matrixnorm ist folglich verträglich mit der sie induzierenden Vektornorm. Wir halten dies im folgenden Satz fest:

Satz 7.15 *Sei $\|\cdot\|$ eine Vektornorm auf dem $\mathbb{R}^n$. Dann definiert $\max_{\boldsymbol{x}\neq 0} \frac{\|\boldsymbol{A}\cdot\boldsymbol{x}\|}{\|\boldsymbol{x}\|}$ eine Matrixnorm auf dem $\mathbb{R}^{n\times n}$, die mit der Vektornorm verträglich ist.*

Bemerkung Zu jedem $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ gibt es ein $\boldsymbol{x}$ mit $\max_{\boldsymbol{y}\neq 0} \frac{\|\boldsymbol{A}\cdot\boldsymbol{y}\|}{\|\boldsymbol{y}\|} = \frac{\|\boldsymbol{A}\cdot\boldsymbol{x}\|}{\|\boldsymbol{x}\|}$, das heißt, die von einer Vektornorm induzierte Matrixnorm ist die kleinstmögliche mit der Vektornorm verträgliche Matrixnorm. Die L_∞-Norm induziert die Zeilensummennorm (vergleiche Beispiel 7.14), die L_1-Norm induziert die Spaltensummennorm (analog zu Beispiel 7.14), die L_2-Norm induziert die Spektralnorm $\|\boldsymbol{A}\|_2 := \sqrt{\lambda_{\max}(\boldsymbol{A}^H\boldsymbol{A})}$.

7.2.1 Fehlerbetrachtungen

Sei ein Gleichungssystem $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$ gegeben und sei $\widetilde{\boldsymbol{x}}$ die numerisch berechnete Lösung. Dann ist $\boldsymbol{r} := \boldsymbol{A}\cdot\widetilde{\boldsymbol{x}} - \boldsymbol{b}$ der sogenannte **Residuenvektor**.

Beispiel 7.16

$$\begin{array}{rcrcr} 0.001x & + & 0.001y & = & 0.001 \\ 0.001x & + & 2y & = & 1 \end{array}$$

Bei Rechnung mit zwei signifikanten Stellen beim Gauß-Algorithmus erhalten wir $y = 0.5$ und $x = 0$.
$\Rightarrow \boldsymbol{r} = \begin{pmatrix} 0 \\ 0.0005 \end{pmatrix}$. $\Rightarrow \|r\|_\infty = 0.0005$ ist klein. Ist die Lösung daher gut?

Im Folgenden gehen wir davon aus, dass $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ und $\|\cdot\| : \mathbb{R}^{n\times n} \to \mathbb{R}$ miteinander verträgliche Vektor- und Matrixnormen sind. Dann gilt $\|\boldsymbol{b}\| = \|\boldsymbol{A}\cdot\boldsymbol{x}\| \le \|\boldsymbol{A}\|\|\boldsymbol{x}\|$ und $\boldsymbol{A}\cdot(\boldsymbol{x}-\widetilde{\boldsymbol{x}}) = \boldsymbol{A}\cdot\boldsymbol{x} - \boldsymbol{A}\cdot\widetilde{\boldsymbol{x}} = \boldsymbol{b} - (\boldsymbol{r}+\boldsymbol{b}) = -\boldsymbol{r}$. $\Rightarrow \boldsymbol{x}-\widetilde{\boldsymbol{x}} = -\boldsymbol{A}^{-1}\cdot\boldsymbol{r}$.
$\Rightarrow \|\boldsymbol{x}-\widetilde{\boldsymbol{x}}\| = \|-\boldsymbol{A}^{-1}\cdot\boldsymbol{r}\| \le \|\boldsymbol{A}^{-1}\|\|\boldsymbol{r}\|$.

$$\Rightarrow \qquad \frac{\|\boldsymbol{x}-\widetilde{\boldsymbol{x}}\|}{\|\boldsymbol{x}\|} \le \frac{\|\boldsymbol{A}^{-1}\|\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|}\|\boldsymbol{A}\| = \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|}.$$

Die Größe $\kappa(\boldsymbol{A}) := \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|$ nennt man die **Kondition** der Matrix $\boldsymbol{A}$. Diese ist abhängig von der verwendeten Matrixnorm. Falls der Residuenvektor klein ist in Bezug auf die rechte Seite $\boldsymbol{b}$ und die Konditionszahl von $\boldsymbol{A}$, so kann auf einen kleinen relativen Fehler geschlossen werden.

Weiter mit Beispiel 7.16: Es gilt $\|\boldsymbol{r}\|_\infty/\|\boldsymbol{b}\|_\infty = 0.0005$.

$$\boldsymbol{A} := \begin{pmatrix} 1/1000 & 2 \\ 1/1000 & 1/100 \end{pmatrix} \qquad \Rightarrow \qquad \boldsymbol{A}^{-1} \approx \begin{pmatrix} 1/2 & 1000 \\ -1/2 & -1/2 \end{pmatrix}$$

$\Rightarrow \|\boldsymbol{A}\|_\infty\|\boldsymbol{A}^{-1}\|_\infty \approx 2000$. $\Rightarrow \|\boldsymbol{A}\|_\infty\|\boldsymbol{A}^{-1}\|_\infty\frac{\|\boldsymbol{r}\|_\infty}{\|\boldsymbol{b}\|_\infty} \approx 1$, das heißt der relative Fehler kann sehr groß sein. Die korrekte Lösung ist $x \approx y \approx 0.5$ statt $y = 0.5$, $x = 0$.
Multipliziere die zweite Gleichung mit 1000:

$$\begin{array}{rcrcl} 0.001x & + & 2y & = & 1 \\ x & + & y & = & 1 \end{array}$$

Der Gauß-Algorithmus mit zwei signifikanten Stellen liefert immer noch $y = 0.5$ und $x = 0$. Jetzt gilt aber

$$\boldsymbol{A} := \begin{pmatrix} 1/1000 & 2 \\ 1 & 1 \end{pmatrix} \qquad \Rightarrow \qquad \boldsymbol{A}^{-1} \approx \begin{pmatrix} -1/2 & 1 \\ 1/2 & 0 \end{pmatrix}.$$

$\Rightarrow \|\boldsymbol{A}\|_\infty\|\boldsymbol{A}^{-1}\|_\infty \approx 2$ ist nun klein, aber $\|\boldsymbol{r}\|_\infty/\|\boldsymbol{b}\|_\infty = \left\|\begin{pmatrix} 0 \\ 1/2 \end{pmatrix}\right\|_\infty \Big/ \left\|\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right\|_\infty = 0.5$.
$\Rightarrow \|\boldsymbol{A}\|_\infty\|\boldsymbol{A}^{-1}\|_\infty\frac{\|\boldsymbol{r}\|_\infty}{\|\boldsymbol{b}\|_\infty} \approx 1$ wie oben, aber $\boldsymbol{A}$ hat nun kleine Kondition.

Betrachte nun die Störung $\Delta\boldsymbol{x}$ der Lösung $\boldsymbol{x}$ eines linearen Gleichungssystems $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$, falls die Eingabedaten eine Störung $\Delta\boldsymbol{A}$ beziehungsweise $\Delta\boldsymbol{b}$ aufweisen. Es gelte also $(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot(\boldsymbol{x}+\Delta\boldsymbol{x}) = \boldsymbol{b}+\Delta\boldsymbol{b}$. Wie groß ist $\Delta\boldsymbol{x}$?

Satz 7.17 *Sei $\boldsymbol{x}$ die Lösung des linearen Gleichungssystems $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{b} \neq 0$ und sei $\boldsymbol{A}$ regulär. $\Delta\boldsymbol{A}$ und $\Delta\boldsymbol{b}$ seien Störungen von $\boldsymbol{A}$ beziehungsweise $\boldsymbol{b}$ und $\boldsymbol{x}+\Delta\boldsymbol{x}$ sei Lösung des gestörten Systems. Falls $\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\| < 1$, so gilt*

$$\frac{\|\Delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \le \frac{\kappa(\boldsymbol{A})}{1-\kappa(\boldsymbol{A})\|\Delta\boldsymbol{A}\|/\|\boldsymbol{A}\|}\left(\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|} + \frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}\right).$$

Beweis. Wir zeigen zunächst, dass $\boldsymbol{A}+\Delta\boldsymbol{A}$ regulär ist und damit das gestörte System eine Lösung hat.
Es gilt

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{A}^{-1}\cdot(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot\boldsymbol{x}-\boldsymbol{A}^{-1}\cdot\Delta\boldsymbol{A}\cdot\boldsymbol{x} \\ \Rightarrow\quad \|\boldsymbol{x}\| &\leq \|\boldsymbol{A}^{-1}\|\|(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot\boldsymbol{x}\|-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|\|\boldsymbol{x}\| \\ \Rightarrow\quad \|\boldsymbol{x}\| &\leq \frac{\|\boldsymbol{A}^{-1}\|}{1-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|}\|(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot\boldsymbol{x}\| \end{aligned}$$

$\Rightarrow \boldsymbol{A}+\Delta\boldsymbol{A}$ ist injektiv $\Rightarrow$ bijektiv $\Rightarrow$ Behauptung.

$$\begin{aligned} &(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot(\boldsymbol{x}+\Delta\boldsymbol{x})=\boldsymbol{b}+\Delta\boldsymbol{b} \\ \Rightarrow\quad &\boldsymbol{A}\cdot\Delta\boldsymbol{x}+\Delta\boldsymbol{A}\cdot(\boldsymbol{x}+\Delta\boldsymbol{x})=\Delta\boldsymbol{b} \\ \Rightarrow\quad &\boldsymbol{x}=\boldsymbol{A}^{-1}\cdot(\Delta\boldsymbol{b}-\Delta\boldsymbol{A}\cdot(\boldsymbol{x}+\boldsymbol{\Delta x})) \\ \Rightarrow\quad &\|\boldsymbol{x}\|\leq\|\boldsymbol{A}^{-1}\|\,(\|\Delta\boldsymbol{b}\|+\|\Delta\boldsymbol{A}\|\|\boldsymbol{x}\|+\|\Delta\boldsymbol{A}\|\|\boldsymbol{\Delta x}\|) \\ \Rightarrow\quad &\|\Delta\boldsymbol{x}\|\left(1-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|\right)\leq\|\boldsymbol{A}^{-1}\|\,(\|\Delta\boldsymbol{b}\|+\|\Delta\boldsymbol{A}\|\|\boldsymbol{x}\|) \end{aligned}$$

wegen $\|\boldsymbol{b}\|=\|\boldsymbol{A}\cdot\boldsymbol{x}\|\leq\|\boldsymbol{A}\|\|\boldsymbol{x}\|$ folgt $\|\boldsymbol{x}\|=\|\boldsymbol{b}\|/\|\boldsymbol{A}\|$. Somit gilt

$$\begin{aligned} \frac{\|\Delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} &\leq \frac{1}{1-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|}\|\boldsymbol{A}^{-1}\|\left(\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{x}\|}+\|\Delta\boldsymbol{A}\|\right) \\ &\leq \frac{\|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\|}{1-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|}\left(\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|}+\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}\right) \\ &\leq \frac{\kappa(\boldsymbol{A})}{1-\kappa(\boldsymbol{A})\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}}\left(\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|}+\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}\right) \end{aligned}$$

□

Ähnlich zu Satz 7.17 gibt der folgende Satz Auskunft darüber, welche Auswirkung die Störung einer Matrix $\boldsymbol{A}$ auf die Inverse hat.

Satz 7.18 *Sei $\boldsymbol{A}$ regulär und $\Delta\boldsymbol{A}$ sei eine Störung von $\boldsymbol{A}$ mit $\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|<1$. Dann gilt*

$$\frac{\|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\|}{\|\boldsymbol{A}^{-1}\|}\leq\kappa(\boldsymbol{A})\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}(1+\mathcal{O}(\|\Delta\boldsymbol{A}\|)).$$

Beweis. Wie in Satz 7.17 gezeigt, existiert $(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}$. Es gilt

$$\begin{aligned} (\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1} &= (\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\cdot(\boldsymbol{A}+\Delta\boldsymbol{A})\cdot(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1} \\ &= (\boldsymbol{I}-\boldsymbol{A}^{-1}\cdot(\boldsymbol{A}+\Delta\boldsymbol{A}))\cdot(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1} \\ &= -\boldsymbol{A}^{-1}\cdot\Delta\boldsymbol{A}\cdot(\boldsymbol{A}+\Delta\boldsymbol{A}) \end{aligned}$$

$$\begin{aligned} \Rightarrow\quad \|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\| &\leq \|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|\|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}+\boldsymbol{A}^{-1}\| \\ &\leq \|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|(\|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\|+\|\boldsymbol{A}^{-1}\|) \\ &= \|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\|\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|+\|\boldsymbol{A}^{-1}\|^2\|\Delta\boldsymbol{A}\| \end{aligned}$$

$$\Rightarrow \quad \|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\| \leq \frac{\|\boldsymbol{A}^{-1}\|^2\|\Delta\boldsymbol{A}\|}{1-\|\boldsymbol{A}^{-1}\|\|\Delta\boldsymbol{A}\|} = \|\boldsymbol{A}^{-1}\|^2\|\Delta\boldsymbol{A}\|(1+\mathcal{O}(\|\Delta\boldsymbol{A}\|)),$$

da $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i = 1+\mathcal{O}(x)$ für $x<1$.

$$\begin{aligned} \Rightarrow \quad \frac{\|(\boldsymbol{A}+\Delta\boldsymbol{A})^{-1}-\boldsymbol{A}^{-1}\|}{\|\boldsymbol{A}^{-1}\|} &\leq \|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\|\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}(1+\mathcal{O}(\|\Delta\boldsymbol{A}\|)) \\ &= \kappa(\boldsymbol{A})\frac{\|\Delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|}(1+\mathcal{O}(\|\Delta\boldsymbol{A}\|)) \end{aligned}$$

□

In Beispiel 7.16 haben wir bereits gesehen, dass man die Kondition einer Matrix durch Multiplikation einer Zeile mit einer Zahl verbessern kann. Was ist die optimale Strategie dafür?

Es sei $\boldsymbol{D} \in \mathbb{R}^{n\times n}$ eine **Diagonalmatrix**, das heißt

$$\boldsymbol{D} = \begin{pmatrix} d_1 & & & \\ & \ddots & & 0 \\ & & \ddots & \\ & 0 & & \ddots \\ & & & & d_n \end{pmatrix} = \operatorname{diag}(d_1,\ldots,d_n)$$

und sei $\boldsymbol{A} \in \mathbb{R}^{n\times n}$. Dann erhält man $\boldsymbol{D}\cdot\boldsymbol{A}$ aus $\boldsymbol{A}$, indem man die i-te Zeile von $\boldsymbol{A}$ mit d_i multipliziert. Statt $\boldsymbol{A}\cdot\boldsymbol{x}=\boldsymbol{b}$ können wir auch $\boldsymbol{D}\cdot\boldsymbol{A}\cdot\boldsymbol{x}=\boldsymbol{D}\cdot\boldsymbol{b}$ lösen, falls $\boldsymbol{D}$ eine reguläre Diagonalmatrix ist. Wie muss man $\boldsymbol{D}$ wählen, damit $\boldsymbol{D}\cdot\boldsymbol{A}$ eine bessere Kondition hat als $\boldsymbol{A}$?

Satz 7.19 *Es sei $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ reguläre Matrix mit $\sum_{j=1}^{n}|a_{ij}| = 1$ für $1 \leq i \leq n$. Dann gilt für jede reguläre Diagonalmatrix $\boldsymbol{D}$*

$$\kappa_\infty(\boldsymbol{D}\cdot\boldsymbol{A}) \geq \kappa(\boldsymbol{A}).$$

Beweis. Es gilt $\|\boldsymbol{D}\cdot\boldsymbol{A}\|_\infty = \max_{1\leq i\leq n}|d_i|\sum_{j=1}^{n}|a_{ij}| = \max_{1\leq i\leq n}|d_i| = \|\boldsymbol{A}\|_\infty \max_{1\leq i\leq n}|d_i|$.
Es gilt weiter $\boldsymbol{D} = \operatorname{diag}(d_1,\ldots,d_n) \Rightarrow \boldsymbol{D}^{-1} = \operatorname{diag}\left(\frac{1}{d_1},\ldots,\frac{1}{d_n}\right)$.
Sei $\boldsymbol{A}^{-1} = (\widetilde{a}_{ij})_{i,j=1,\ldots,n}$. Dann erhalten wir:

$$\begin{aligned} \|(\boldsymbol{D}\cdot\boldsymbol{A})^{-1}\|_\infty &= \|\boldsymbol{A}^{-1}\cdot\boldsymbol{D}^{-1}\|_\infty = \max_{1\leq i\leq n}\sum_{j=1}^{n}\frac{|\widetilde{a}_{ij}|}{|d_j|} \\ &\geq \frac{\max_{1\leq i\leq n}\sum_{j=1}^{n}|\widetilde{a}_{ij}|}{\max_{1\leq k\leq n}|d_k|} = \frac{\|\boldsymbol{A}^{-1}\|_\infty}{\max_{1\leq k\leq n}|d_k|} \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad \kappa(\boldsymbol{D}\cdot\boldsymbol{A}) &= \|\boldsymbol{D}\cdot\boldsymbol{A}\|_\infty \|(\boldsymbol{D}\cdot\boldsymbol{A})^{-1}\|_\infty \geq \|\boldsymbol{A}\|_\infty \max_{1\leq i\leq n}|d_i| \frac{\|\boldsymbol{A}^{-1}\|_\infty}{\max_{1\leq k\leq n}|d_k|} \\ &= \|\boldsymbol{A}\|_\infty \|\boldsymbol{A}^{-1}\|_\infty = \kappa(\boldsymbol{A}) \end{aligned}$$

□

Die bestmögliche Kondition erhält man also, indem man die Zeilen von $\boldsymbol{A}$ so multipliziert, dass alle Zeilensummen gleich sind (zum Beispiel = 1). Dies nennt man **Äquilibrierung** von $\boldsymbol{A}$.

7.3 Pivotisierung und Stabilität des Gauß-Algorithmus

Zeile ② des Gauß-Algorithmus :
② bestimme $l \geq k$, sodass $a_{lk} \neq 0$.
Naheliegend ist es das kleinste $l \geq k$ mit $a_{lk} \neq 0$ zu wählen. Zur Erhöhung der Stabilität das Gauß-Algorithmus gibt es jedoch bessere Strategien.

Beispiel 7.20 Betrachte das lineare Gleichungssystem $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{A} = \begin{pmatrix} \frac{1}{1000} & 1 \\ 1 & 1 \end{pmatrix}$ und $\boldsymbol{b} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$. Dann lautet die exakte Lösung $x_1 = \frac{1000}{999} \approx 1$ und $x_2 = \frac{998}{999} \approx 1$.
Bei Anwendung des Gauß-Algorithmus mit zwei signifikanten Stellen erhält man:

$$\left(\begin{array}{cc|c} \frac{1}{1000} & 1 & 1 \\ 1 & 1 & 2 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} \frac{1}{1000} & 1 & 1 \\ 0 & 1-1000 & 2-1000 \end{array}\right) \approx \left(\begin{array}{cc|c} \frac{1}{1000} & 1 & 1 \\ 0 & -1000 & -1000 \end{array}\right)$$

$\Rightarrow x_2 = 1$, $x_1 = 0$.
Wählt man in Zeile ② des Gauß-Algorithmus hingegen das betragsmäßig größte Element, so erhält man:

$$\left(\begin{array}{cc|c} \frac{1}{1000} & 1 & 1 \\ 1 & 1 & 2 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 0 & 1-\frac{1}{1000} & 1\frac{2}{1000} \\ 0 & 1 & 2 \end{array}\right) \approx \left(\begin{array}{cc|c} 0 & 1 & 1 \\ 0 & 1 & 2 \end{array}\right)$$

$\Rightarrow x_2 = 2$, $x_1 = 1$.
Was ist passiert? Falls $l_{21} = \frac{a_{21}}{a_{11}}$ sehr groß ist, so gilt $a_{22} - l_{21}a_{12} \approx -l_{21}a_{12}$ und $b_2 - l_{21}b_1 \approx -l_{21}b_1$.
$\Rightarrow x_2 \frac{-l_{21}b_1}{a_{22}-l_{21}a_{12}} \approx \frac{b_1}{a_{12}}$. Da a_{11} klein ist, ist dies eine gute Näherung. $x_1 = (b_1 - a_{12}x_2)/a_{11}$ ist wegen $x_2 \approx \frac{b_1}{a_{12}}$ schlecht konditioniert (Auslöschung).

Die Elemente a_{kk} im k-ten Schritt des Gauß-Algorithmus werden **Pivotelemente** genannt. Zur Erhöhung der Stabilität wählt man in jedem Schritt das Element der k-ten Spalte, das betragsmäßig am größten ist (Spaltenpivotsuche). Damit ist garantiert, dass $|l_{jk}| = \left|\frac{a_{jk}}{a_{kk}}\right| \leq 1$.

Beispiel 7.21 ***L*-*R*-Zerlegung mit Spaltenpivotsuche**

$$\begin{pmatrix} 1 & -2 & 3 & 4 \\ 2 & -4 & 8 & 6 \\ -1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & -4 & 8 & 6 \\ \frac{1}{2} & 0 & -1 & 1 \\ -\frac{1}{2} & 0 & 7 & 7 \\ 0 & 1 & 2 & 3 \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} 2 & -4 & 8 & 6 \\ 0 & 1 & 2 & 3 \\ -\frac{1}{2} & 0 & 7 & 7 \\ \frac{1}{2} & 0 & -1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & -4 & 8 & 6 \\ 0 & 1 & 2 & 3 \\ -\frac{1}{2} & 0 & 7 & 7 \\ \frac{1}{2} & 0 & -\frac{1}{7} & 2 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & 1 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{7} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & -4 & 8 & 6 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 7 & 7 \\ 0 & 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 2 & -4 & 8 & 6 \\ 0 & 1 & 2 & 3 \\ -1 & 2 & 3 & 4 \\ 1 & -2 & 3 & 4 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & -2 & 3 & 4 \\ 2 & -4 & 8 & 6 \\ -1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \end{pmatrix}$$

Satz 7.22 *Seien $\widetilde{\boldsymbol{L}}$ und $\widetilde{\boldsymbol{R}}$ die numerisch mittels Gauß-Algorithmus berechnete $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung der Matrix $\boldsymbol{A}$ mit Spaltenpivotsuche, das heißt $|\widetilde{l}_{ij}| \leq 1$ für alle $i, j = 1, \ldots n$ und $\widetilde{\boldsymbol{A}} := \widetilde{\boldsymbol{L}} \cdot \widetilde{\boldsymbol{R}}$. Dann gilt:*

$$|a_{ij} - \widetilde{a}_{ij}| \leq 2a \min(i-1, j) \frac{\text{eps}}{1 - \text{eps}},$$

wobei $a = \max_{i,j,k} |\widetilde{a}_{ij}^{(k)}|$ und eps *die Maschinengenauigkeit.*

Beweis. Der k-te Schritt des Gauß-Algorithmus lautet unter Berücksichtigung von Rundungsfehlern:

$$\begin{aligned} \widetilde{a}_{ji}^{(k)} &= \left(\widetilde{a}_{ji}^{(k-1)} - \widetilde{l}_{ji}\widetilde{a}_{ki}^{(k-1)}(1+\varepsilon_{ijk})\right)(1+\gamma_{ijk}) \\ &= \widetilde{a}_{ji}^{(k-1)} - \widetilde{l}_{ji}\widetilde{a}_{ki}^{(k-1)} + \mu_{ijk} \\ \Rightarrow \quad \widetilde{a}_{ji}^{(k)} - \mu_{ijk} &= \widetilde{a}_{ji}^{(k-1)} - \widetilde{l}_{ji}\widetilde{a}_{ki}^{(k-1)}. \end{aligned} \tag{7.3}$$

Zudem gilt:

$$\frac{\widetilde{a}_{ij}^{(k)}}{1+\gamma_{ijk}} + \widetilde{l}_{jk}\widetilde{a}_{ki}^{(k-1)}\varepsilon_{ijk} = \widetilde{a}_{ji}^{(k-1)} - \widetilde{l}_{jk}\widetilde{a}_{ki}^{(k-1)} \tag{7.4}$$

(7.3) und (7.4) implizieren

$$\begin{aligned} \mu_{ijk} &= \widetilde{a}_{ji}^{(k)} - \frac{\widetilde{a}_{ij}^{(k)}}{1+\gamma_{ijk}} - \widetilde{l}_{jk}\widetilde{a}_{ki}^{(k-1)}\varepsilon_{ijk} \\ \Rightarrow \quad |\mu_{ijk}| &= \left|\widetilde{a}_{ji}^{(k)}\right| + \frac{|\gamma_{ijk}|}{1-|\gamma_{ijk}|} - \underbrace{|\widetilde{l}_{jk}|}_{\leq 1}\left|\widetilde{a}_{ki}^{(k-1)}\right||\varepsilon_{ijk}| \\ &\leq a\frac{\text{eps}}{1-\text{eps}} + a\,\text{eps} \leq 2a\frac{\text{eps}}{a-\text{eps}} \end{aligned}$$

$\widetilde{\boldsymbol{A}} = \widetilde{\boldsymbol{L}} \cdot \widetilde{\boldsymbol{R}} \Rightarrow \widetilde{a}_{ji} = \sum_{k=1}^{\min(i,j)} \widetilde{l}_{jk}\widetilde{a}_{ki}^{(k-1)}$
Falls $j > i$ so gilt

$$\begin{aligned} \widetilde{a}_{ji} &= \sum_{k=1}^{i} \widetilde{l}_{jk}\widetilde{a}_{ki}^{k-1} \\ &= \sum_{k=1}^{i} \left(\widetilde{a}_{ji}^{(k-1)} - \widetilde{a}_{ji}^{(k)} + \mu_{ijk}\right) \quad \text{wegen (7.3)} \\ &= \widetilde{a}_{ji}^{(0)} - \underbrace{\widetilde{a}_{ji}^{(i)}}_{=0} + \sum_{k=1}^{i} \mu_{ijk}. \end{aligned}$$

Falls $j \leq i$ so gilt

$$\begin{aligned} \widetilde{a}_{ji} &= \sum_{k=1}^{j-1} \left(\widetilde{a}_{ji}^{(k-1)} - \widetilde{a}_{ji}^{(k)} + \mu_{ijk}\right) + \underbrace{\widetilde{l}_{jj}}_{=1}\widetilde{a}_{ji}^{(j-1)} \\ &= \widetilde{a}_{ji}^{(0)} - \widetilde{a}_{ji}^{(j-1)} + \widetilde{a}_{ji}^{(j-1)} + \sum_{k=1}^{j-1} \mu_{ijk}. \end{aligned}$$

$\Rightarrow |a_{ji} - \widetilde{a}_{ji}| \leq \max_{ijk} |\mu_{ijk}| \min(j-1, i) \leq 2a\frac{\text{eps}}{1-\text{eps}} \min(j-1, i).$ □

In Matrixschreibweise lautet Satz 7.22

$$|\boldsymbol{A} - \widetilde{\boldsymbol{L}} \cdot \widetilde{\boldsymbol{R}}| \leq 2a\frac{\text{eps}}{1-\text{eps}} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 \\ 1 & 2 & 3 & 3 & 3 & 3 \\ & & & \vdots & & \\ 1 & 2 & 3 & \dots & n-1 & n-1 \end{pmatrix}.$$

7.4 Die Cholesky-Zerlegung

Eine Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ heißt **symmetrisch**, falls $\boldsymbol{A} = \boldsymbol{A}^T$. $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ heißt **positiv definit**, falls $\boldsymbol{x}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x} > 0$ für alle $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x} \neq 0$, gilt.

Bemerkung $\boldsymbol{A}$ positiv definit impliziert, dass $\boldsymbol{A}$ regulär ist, da sonst $\boldsymbol{x} \neq 0$ existiert mit $\boldsymbol{A} \cdot \boldsymbol{x} = 0$.

Satz 7.23 *Sei $\boldsymbol{A} \in \mathbb{R}^{n \times n}$. Dann ist $\boldsymbol{A}^T \cdot \boldsymbol{A}$ symmetrisch und, falls $\boldsymbol{A}$ regulär ist, so ist $\boldsymbol{A}^T \cdot \boldsymbol{A}$ positiv definit.*

Beweis. Für beliebige Matrizen $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$ gilt $(\boldsymbol{A} \cdot \boldsymbol{B})^T = \boldsymbol{B}^T \cdot \boldsymbol{A}^T$. Dies lässt sich leicht durch Nachrechnen zeigen. Somit gilt $(\boldsymbol{A}^T \cdot \boldsymbol{A})^T = \boldsymbol{A}^T \cdot (\boldsymbol{A}^T)^T$, das heißt $\boldsymbol{A}^T \cdot \boldsymbol{A}$ ist symmetrisch.
Sei nun $\boldsymbol{A}$ regulär. Dann gilt für $\boldsymbol{x} \in \mathbb{R}^n$: $\boldsymbol{A} \cdot \boldsymbol{x} = 0 \Leftrightarrow \boldsymbol{x} = 0$. Nun gilt $\boldsymbol{x}^T \cdot (\boldsymbol{A}^T \cdot \boldsymbol{A}) \cdot \boldsymbol{x} = (\boldsymbol{x}^T \cdot \boldsymbol{A}^T) \cdot (\boldsymbol{A} \cdot \boldsymbol{x}) = (\boldsymbol{A} \cdot \boldsymbol{x})^T \cdot (\boldsymbol{A} \cdot \boldsymbol{x})$. Sei $\boldsymbol{y} = \boldsymbol{A} \cdot \boldsymbol{x}$. Dann gilt $\boldsymbol{y}^T \cdot \boldsymbol{y} = 0 \Leftrightarrow \boldsymbol{y} = 0$. Aus $\boldsymbol{y} = 0$ folgt $\boldsymbol{A} \cdot \boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = 0$, da $\boldsymbol{A}$ regulär. □

Bemerkung Mit $\boldsymbol{A}$ ist auch $\boldsymbol{A}^T$ regulär. Folglich gilt Satz 7.23 gilt auch für $\boldsymbol{A} \cdot \boldsymbol{A}^T$.

Sei $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ und $\emptyset \neq I \subseteq \{1, \ldots, n\}$. Dann heißt die Matrix $(a_{ij})_{i,j \in I} \in \mathbb{R}^{|I| \times |I|}$ **Hauptuntermatrix** von $\boldsymbol{A}$.

Beispiel 7.24 Die Matrix $\begin{pmatrix} 3 & -1 & 0 \\ 4 & 5 & 8 \\ -2 & 1 & 1 \end{pmatrix}$ besitzt die Hauptuntermatrizen (3), (5), (1), $\begin{pmatrix} 3 & -1 \\ 4 & 5 \end{pmatrix}$, $\begin{pmatrix} 5 & 8 \\ 1 & 1 \end{pmatrix}$, $\begin{pmatrix} 3 & 0 \\ -2 & 1 \end{pmatrix}$ und die Matrix selbst.

Satz 7.25 *Sei $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ und $\boldsymbol{A}'$ Hauptuntermatrix von $\boldsymbol{A}$. Dann gilt*

(a) Ist $\boldsymbol{A}$ symmetrisch, so auch $\boldsymbol{A}'$.

(b) Ist $\boldsymbol{A}$ positiv definit, so auch $\boldsymbol{A}'$.

Beweis.

(a) Folgt unmittelbar aus der Definition.

(b) Angenommen $\boldsymbol{A}'$ ist nicht positiv definit. Sei $I \subseteq \{1, \ldots, n\}$ die Indexmenge, die $\boldsymbol{A}'$ definiert. Dann gibt es $\boldsymbol{x} \in \mathbb{R}^{|I|}$ mit $\boldsymbol{x}^T \cdot \boldsymbol{A}' \cdot \boldsymbol{x} = 0$ und $\boldsymbol{x} \neq 0$. Definiere nun $\boldsymbol{y} \in \mathbb{R}^n$ durch

$$y_i = \begin{cases} x_i, & \text{falls } i \in I \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt: $\boldsymbol{y}^T \cdot \boldsymbol{A} \cdot \boldsymbol{y} = \boldsymbol{x}^T \cdot \boldsymbol{A}' \cdot \boldsymbol{x} = 0$ und $\boldsymbol{y} \neq 0$. Dies widerspricht der positiven Definitheit von $\boldsymbol{A}$.

□

Satz 7.26 (Satz von der Cholesky-Zerlegung) *Es sei $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ symmetrisch und positiv definit. Dann existiert genau eine reguläre untere Dreiecksmatrix $\boldsymbol{L} \in \mathbb{R}^{n \times n}$, sodass $\boldsymbol{A} = \boldsymbol{L}^T \cdot \boldsymbol{L}$ und $l_{ii} > 0$ für $i = 1, \ldots, n$.*

Beweis. Durch Induktion nach n. Für $n = 1$ gilt $\boldsymbol{A} = (a_{11})$ mit $a_{11} > 0$ wegen der positiven Definitheit von $\boldsymbol{A}$. Somit ist $\boldsymbol{L} := (\sqrt{a_{11}})$ eindeutige Lösung. Gelte nun die Aussage für $n-1$. Wir schreiben $\boldsymbol{A}$ als

$$\boldsymbol{A} = \begin{pmatrix} \widetilde{\boldsymbol{A}} & \boldsymbol{b} \\ \boldsymbol{b}^T & a_{nn} \end{pmatrix}$$

mit $\boldsymbol{b} \in \mathbb{R}^{n-1}$ und $\widetilde{\boldsymbol{A}} \in \mathbb{R}^{(n-1)\times(n-1)}$. Dann gilt $a_{nn} = \boldsymbol{e}_n^T \cdot \boldsymbol{A} \cdot \boldsymbol{e}_n > 0$ wegen der positiven Definitheit von $\boldsymbol{A}$. Zudem ist $\widetilde{\boldsymbol{A}}$ Hauptuntermatrix von $\boldsymbol{A}$ und somit nach Satz 7.25 symmetrisch und positiv definit. Es gibt somit genau eine reguläre untere Dreiecksmatrix $\widetilde{\boldsymbol{L}}$ mit $\widetilde{\boldsymbol{A}} = \widetilde{\boldsymbol{L}} \cdot \widetilde{\boldsymbol{L}}^T$ und $\widetilde{l}_{ii} > 0$ für $i = 1, \ldots, n-1$. Die gesuchte Matrix $\boldsymbol{L}$ hat somit die Form

$$\boldsymbol{L} = \begin{pmatrix} \widetilde{\boldsymbol{L}} & 0 \\ \boldsymbol{c}^T & \alpha \end{pmatrix}$$

mit $\alpha > 0$ und $\boldsymbol{c} \in \mathbb{R}^{n-1}$. Es gilt

$$\boldsymbol{A} = \begin{pmatrix} \widetilde{\boldsymbol{A}} & \boldsymbol{b} \\ \boldsymbol{b}^T & a_{nn} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{L}} & 0 \\ \boldsymbol{c}^T & \alpha \end{pmatrix} \cdot \begin{pmatrix} \widetilde{\boldsymbol{L}}^T & \boldsymbol{c} \\ 0 & \alpha \end{pmatrix}$$

$\Rightarrow \boldsymbol{c}^T \cdot \widetilde{\boldsymbol{L}}^T = \boldsymbol{b}^T$ beziehungsweise $\widetilde{\boldsymbol{L}} \cdot \boldsymbol{c} = \boldsymbol{b}$ und $\boldsymbol{c}^T \cdot \boldsymbol{c} + \alpha^2 = a_{nn}$. Da $\widetilde{\boldsymbol{L}}$ regulär ist, gilt $\boldsymbol{c} = \widetilde{\boldsymbol{L}}^{-1} \cdot \boldsymbol{b}$. Zudem gilt für $\boldsymbol{x} \in \mathbb{R}^n$ mit $\boldsymbol{x} := \begin{pmatrix} \left(\widetilde{\boldsymbol{L}}^T\right)^{-1} \cdot \boldsymbol{c} \\ -1 \end{pmatrix}$ offensichtlich $\boldsymbol{x} \neq 0$. Somit gilt

$$\begin{aligned} 0 &< \boldsymbol{x}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x} \\ &= \boldsymbol{x}^T \cdot \begin{pmatrix} \widetilde{\boldsymbol{L}} \cdot \widetilde{\boldsymbol{L}}^T & \boldsymbol{b} \\ \boldsymbol{b}^T & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} \left(\widetilde{\boldsymbol{L}}^T\right)^{-1} \cdot \boldsymbol{c} \\ -1 \end{pmatrix} \\ &= \boldsymbol{x}^T \cdot \begin{pmatrix} \widetilde{\boldsymbol{L}} \cdot \boldsymbol{c} - \boldsymbol{b} \\ \boldsymbol{c}^T \cdot \boldsymbol{c} - a_{nn} \end{pmatrix} \\ &= (\boldsymbol{c}^T \cdot \widetilde{\boldsymbol{L}}^{-1}, -1) \cdot \begin{pmatrix} 0 \\ \boldsymbol{c}^T \cdot \boldsymbol{c} - a_{nn} \end{pmatrix} \\ &= a_{nn} - \boldsymbol{c}^T \cdot \boldsymbol{c}. \end{aligned}$$

$\Rightarrow \alpha > 0$ mit $\alpha^2 = a_{nn} - \boldsymbol{c}^T \cdot \boldsymbol{c}$ ist eindeutig bestimmt. Da $\boldsymbol{A}$ regulär ist muss auch $\boldsymbol{L}$ regulär sein. □

Die Darstellung $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T$, wobei $\boldsymbol{L}$ reguläre untere Dreiecksmatrix mit $l_{ii} > 0$ für $i = 1, \ldots, n$ ist, heißt **Cholesky-Zerlegung**.

Satz 7.27 *Eine Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ besitzt genau dann eine Cholesky-Zerlegung, wenn sie symmetrisch und positiv definit ist.*

Beweis. Falls $\boldsymbol{A}$ symmetrisch und positiv definit ist, so besitzt $\boldsymbol{A}$ nach 7.26 eine Cholesky-Zerlegung.
Besitze nun umgekehrt $\boldsymbol{A}$ eine Cholesky-Zerlegung. Dann ist $\boldsymbol{L}$ wegen $l_{ii} > 0$ regulär und nach Satz 7.23 ist $\boldsymbol{A}$ symmetrisch und positiv definit. □

Wie berechnet man die Cholesky-Zerlegung einer symmetrischen positiv definiten Matrix?

$\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T \Rightarrow a_{ij} = \sum_{k=1}^{n} l_{ik} l_{jk} = \sum_{k=1}^{\min(i,j)} l_{ik} l_{jk}$. Für $i \geq j$ gilt somit $a_{ij} = \sum_{k=1}^{j} l_{ik} l_{jk}$. Damit lassen sich die Einträge l_{ij} für $j = 1, \ldots, n$ wie folgt bestimmen

$$
\begin{aligned}
l_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}, \\
l_{ij} &= \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) \qquad \text{für } i > j.
\end{aligned}
$$

Cholesky-Zerlegung

Input: Symmetrische positiv definite Matrix $\boldsymbol{A}$.

Output: reguläre untere Dreiecksmatrix $\boldsymbol{L}$ mit $l_{ii} > 0$, sodass $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T$.

① $l_{ij} = 0$;
② `for` $j := 1$ `to` n `do`
③ $\quad l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$;
④ $\quad$ `for` $i := j + 1$ `to` n `do`
⑤ $\quad\quad l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right)$;

Bemerkungen

- Der Aufwand der Cholesky-Zerlegung ist wie bei der $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung $\mathcal{O}(n^3)$, allerdings spart man einen Faktor 2 wegen der Symmetrie.
- Da $\boldsymbol{A}$ symmetrisch ist, muss man lediglich die obere Dreiecksmatrix von $\boldsymbol{A}$ speichern. $\boldsymbol{L}$ kann bis auf die Diagonale im unteren Teil von $\boldsymbol{A}$ gespeichert werden. $\Rightarrow \mathcal{O}(n)$ Speicheraufwand statt $\mathcal{O}(n^2)$ bei der $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung, falls $\boldsymbol{A}$ erhalten bleiben soll.
- Falls $\boldsymbol{A}$ nicht symmetrisch und positiv definit ist, bricht der obige Algorithmus mit einer Division durch 0 oder der Wurzel aus einer negativen Zahl ab.

Sei $\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T$ eine Cholesky-Zerlegung. Dann gilt $a_{jj} = \sum_{k=1}^{j} (l_{jk}^2) \Rightarrow |l_{jk}| \leq \sqrt{\max_{1 \leq i \leq n} |a_{ii}|}$, das heißt die Einträge von $\boldsymbol{L}$ können nicht zu groß werden. Die Cholesky-Zerlegung ist numerisch stabiler als der Gauß-Algorithmus.

Beispiel 7.28

$$\boldsymbol{A} = \begin{pmatrix} 4 & -2 & 2 \\ -1 & 10 & 5 \\ 2 & 5 & 6 \end{pmatrix}$$

$$\begin{aligned} \Rightarrow \quad l_{11} &= \sqrt{a_{11}} = 2 \\ l_{21} &= \frac{1}{2} \cdot (-2) = -1 \\ l_{31} &= \frac{1}{2} \cdot 2 = 1 \\ l_{22} &= \sqrt{a_{22} - l_{21}^2} = 3 \\ l_{32} &= \frac{1}{3}(5 - 1 \cdot (-1)) = 2 \\ l_{33} &= \sqrt{a_{33} - l_{31}^2 - l_{32}^2} = \sqrt{6 - 1 - 4} = 1 \end{aligned}$$

$$\Rightarrow \quad \boldsymbol{L} = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Bemerkung Die Cholesky-Zerlegung erlaubt genau wie die $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung das Lösen von linearen Gleichungssystemen mittels Vorwärts- und Rückwärtseinsetzen:
$\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{L}^T$, $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$. $\Rightarrow \boldsymbol{L} \cdot \boldsymbol{y} = \boldsymbol{b}$ und $\boldsymbol{L}^T \cdot \boldsymbol{x} = \boldsymbol{y}$.

7.5 Die Komplexität von Matrixmultiplikation und Matrixinversion

Matrixmultiplikation und Matrixinversion lassen sich beide in $\mathcal{O}(n^3)$ durchführen, wobei dies für die Matrixmultiplikation trivial ist und für die Matrixinversion aus dem Gauß-Algorithmus beziehungsweise der $\boldsymbol{L}$-$\boldsymbol{R}$-Zerlegung folgt. Kann man diese Probleme auch in $o(n^3)$ lösen? Welches der beiden Probleme ist schwieriger?

Satz 7.29 *Sei $f(n) = \Omega(n^2)$ mit $f(3n) = \mathcal{O}(f(n))$ die Laufzeit, um eine $n \times n$-Matrix zu invertieren. Dann kann man in $\mathcal{O}(f(n))$ zwei $n \times n$-Matrizen multiplizieren.*

Beweis. Seien $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$. Definiere die Matrix $\boldsymbol{A} \in \mathbb{R}^{3n \times 3n}$ durch

$$\boldsymbol{D} = \begin{pmatrix} \boldsymbol{I}_n & \boldsymbol{A} & 0 \\ 0 & \boldsymbol{I}_n & \boldsymbol{B} \\ 0 & 0 & \boldsymbol{I}_n \end{pmatrix}$$

Dann gilt

$$\boldsymbol{D}^{-1} = \begin{pmatrix} \boldsymbol{I}_n & -\boldsymbol{A} & \boldsymbol{A} \cdot \boldsymbol{B} \\ 0 & \boldsymbol{I}_n & -B \\ 0 & 0 & \boldsymbol{I}_n \end{pmatrix}$$

$\Rightarrow$ Das Produkt $\boldsymbol{A}\cdot\boldsymbol{B}$ ist eine Untermatrix von $\boldsymbol{D}^{-1}$. $\boldsymbol{D}^{-1}$ kann in $\mathcal{O}(f(3n)) = \mathcal{O}(f(n))$ berechnet werden. $\square$

Bemerkung Jedes $f(n)$ der Form $f(n) = \Theta(n^\alpha \log^\beta n)$ erfüllt $f(3n) = \mathcal{O}(f(n))$.

Satz 7.30 *Sei $f(n) = \Omega(n^2)$ die Laufzeit um zwei $n \times n$-Matrizen zu multiplizieren mit $f(n+k) = \mathcal{O}(f(n))$ für $0 \leq k \leq n$ und $f\left(\frac{n}{2}\right) \leq cf(n)$ für ein $c < \frac{1}{2}$. Dann kann man die Inverse einer regulären Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ in $\mathcal{O}(f(n))$ berechnen.*

Beweis. Ohne Einschränkung sei n eine Zweierpotenz. Falls nicht, so beachte, dass $\begin{pmatrix} \boldsymbol{A} & 0 \\ 0 & \boldsymbol{I}_k \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{A}^{-1} & 0 \\ 0 & \boldsymbol{I}_k \end{pmatrix}$ gilt für jedes $k > 0$. Falls n keine Zweierpotenz ist, so gibt es $k < n$, sodass $n + k$ eine Zweierpotenz ist.
$\boldsymbol{A}^T\cdot\boldsymbol{A}$ ist nach Satz 7.23 symmetrisch und positiv definit. Es gilt $\boldsymbol{A}^{-1} = (\boldsymbol{A}^T\cdot\boldsymbol{A})^{-1}\cdot\boldsymbol{A}^T$. Es reicht daher, die symmetrische und positiv definite Matrix $\boldsymbol{A}^T \cdot \boldsymbol{A}$ zu invertieren. Dann benötigt man lediglich eine Matrixmultiplikation, um $\boldsymbol{A}^T$ zu berechnen.
Wir zeigen nun, wie man eine symmetrische und positiv definite Matrix $\boldsymbol{A}$ mithilfe von Matrixmultiplikationen invertieren kann. Sei $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ mit n Zweierpotenz. Partitioniere $\boldsymbol{A}$ in vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen wie folgt

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{B} & \boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{D} \end{pmatrix}.$$

Setze $\boldsymbol{S} := \boldsymbol{D} - \boldsymbol{C}\cdot\boldsymbol{B}^{-1}\cdot\boldsymbol{C}^T$. Dann gilt

$$\boldsymbol{A}^{-1} = \begin{pmatrix} \boldsymbol{B}^{-1} + \boldsymbol{B}^{-1}\cdot\boldsymbol{C}^T\cdot\boldsymbol{S}^{-1}\cdot\boldsymbol{C}\cdot\boldsymbol{B}^{-1} & -\boldsymbol{B}^{-1}\cdot\boldsymbol{C}^T\cdot\boldsymbol{S}^{-1} \\ -\boldsymbol{S}^{-1}\cdot\boldsymbol{C}\cdot\boldsymbol{B}^{-1} & \boldsymbol{S}^{-1} \end{pmatrix},$$

wie man durch Nachrechnen verifiziert.
$\Rightarrow \boldsymbol{A}^{-1}$ kann rekursiv berechnet werden, indem man $\boldsymbol{B}^{-1}$ berechnet, dann $\boldsymbol{C}\cdot\boldsymbol{B}^{-1}$, dann $(\boldsymbol{C}\cdot\boldsymbol{B}^{-1})\cdot\boldsymbol{C}^T$, dann $\boldsymbol{S} := \boldsymbol{D} - (\boldsymbol{C}\cdot\boldsymbol{B}^{-1})\cdot\boldsymbol{C}^T$, dann $\boldsymbol{S}^{-1}$, dann $\boldsymbol{S}^{-1}\cdot(\boldsymbol{C}\cdot\boldsymbol{B}^{-1})$, dann $(\boldsymbol{C}\cdot\boldsymbol{B}^{-1})^T\cdot(\boldsymbol{S}^{-1}\cdot(\boldsymbol{C}\cdot\boldsymbol{B}^{-1}))$.
$\Rightarrow$ Vier $\frac{n}{2} \times \frac{n}{2}$-Matrixmultiplikationen und zwei $\frac{n}{2} \times \frac{n}{2}$-Matrixinversionen. Sei $h(n)$ die Laufzeit, dann gilt

$$\begin{aligned} h(n) &\leq 2h\left(\frac{n}{2}\right) + 4f(n) + \mathcal{O}(n^2) \\ &= 2h\left(\frac{n}{2}\right) + \Theta(f(n)) \\ &= \mathcal{O}(f(n)). \end{aligned}$$

Dies lässt sich mit der geometrischen Reihe zeigen. $\square$

Index