# COMPUTERORIENTIERTE MATHEMATIK I


Christof Schütte
Wintersemester 2018/19

## Computer-orientierte Mathematik

9. Vorlesung - Christof Schuette

22.12.18

**Werkzeugkiste**

Definitionen und Berechnungsmethoden zu Kondition und Stabilität

**Auswertungsbäume zur systematischen Stabilitätsabschätzung**

Von der Wurzel zu den Blättern: Auswertungsbaum
Von den Blättern zur Wurzel: rekursive Stabilitätsabschätzung
Blätter: Eingabewerte (Stabilität $\sigma = 1$),
Beispiele.

**Polynom-Desaster revisited**

Niedrige Auflösung (einfaches Modell zur Stabilitätsabschätzung) versus höhere Auflösung (komplexeres Modell zur Stabilitätsabschätzung).

bisher: Teil I

Auswirkung von (Rundungs)-Fehlern auf das Ergebnis:

| | |
|---|---|
| Kondition eines Problems: | Eingabefehler |
| Stabilität eines Algorithmus: | Auswertungsfehler |

bisher: Teil I

Auswirkung von (Rundungs)-Fehlern auf das Ergebnis:

| | |
|---|---|
| Kondition eines Problems: | Eingabefehler |
| Stabilität eines Algorithmus: | Auswertungsfehler |

jetzt: Teil II

Rechenaufwand:

- Komplexität eines Problems
- Effizienz eines Algorithmus

Wie lange muß ich auf das Ergebnis warten?

a) Ist das Problem schwierig? (Komplexität)

b) Ist mein Algorithmus zu langsam? (Effizienz)

Theoretische Informatik/Diskrete Mathematik:

$\rightarrow$ Komplexitätstheorie, Berechenbare Funktionen

# Aufwand (Laufzeit) eines Algorithmus

hängt ab von: Algorithmus, Eingabedaten ("Größe" des Problems)

Option 1: Aufwandsmaß: reine Rechenzeit

Nachteil: Abhängigkeit von zusätzlichen Parametern:

Implementierung, Programmiersprache, Compiler, Prozessor,...

# Aufwand (Laufzeit) eines Algorithmus

hängt ab von: Algorithmus, Eingabedaten ("Größe" des Problems)

Option 1: Aufwandsmaß: reine Rechenzeit

Nachteil: Abhängigkeit von zusätzlichen Parametern:

Implementierung, Programmiersprache, Compiler, Prozessor,...

Option 2: Aufwandsmaß:

Anzahl dominanter Operationen

# Aufwand (Laufzeit) eines Algorithmus

hängt ab von: Algorithmus, Eingabedaten ("Größe" des Problems)

Option 1: Aufwandsmaß: reine Rechenzeit

Nachteil: Abhängigkeit von zusätzlichen Parametern:

Implementierung, Programmiersprache, Compiler, Prozessor,...

Option 2: Aufwandsmaß:

Anzahl dominanter Operationen

Wir wollen allgemeingültige Resultate!

# Beispiel: Sortier-Algorithmen

Problemklasse: Sortieren von $n$ ganzen Zahlen

Problemgröße: $n$

Aufwandsmaß: Anzahl der Vergleiche

Merge-Sort: Anzahl der benötigten Vergleiche: $\leq n \log_2 n$

Definition: (Landau-Symbol $\mathcal{O}$ und das Symbol $\Theta$)

Seien $f : \mathbb{R}^+ \to \mathbb{R}^+ = \{n \in \mathbb{R} \mid n \geq 0\}$ und $g : \mathbb{R}^+ \to \mathbb{R}^+$ zwei Funktionen.

Wir schreiben: $f(n) = \mathcal{O}(g(n))$ für $n \to \infty$, falls

$$\exists C > 0 \text{ und } \exists n_0 \in \mathbb{R} \text{ mit } f(n) \leq C \cdot g(n) \text{ für alle } n \geq n_0$$

Definition: (Landau-Symbol $\mathcal{O}$ und das Symbol $\Theta$)

Seien $f : \mathbb{R}^+ \to \mathbb{R}^+ = \{n \in \mathbb{R} \mid n \geq 0\}$ und $g : \mathbb{R}^+ \to \mathbb{R}^+$ zwei Funktionen.

Wir schreiben: $f(n) = \mathcal{O}(g(n))$ für $n \to \infty$, falls

$$\exists C > 0 \text{ und } \exists n_0 \in \mathbb{R} \text{ mit } f(n) \leq C \cdot g(n) \text{ für alle } n \geq n_0$$

Wir schreiben: $f(n) = \Theta(g(n))$ für $n \to \infty$, falls

$$f(n) = \mathcal{O}(g(n)) \quad \text{und} \quad g(n) = \mathcal{O}(f(n))$$

Satz:

a) $f(n) = O(g(n))$ für $n \to \infty \quad \Longleftarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \quad c \in \mathbb{R}$

b) $f(n) = \Theta(g(n))$ für $n \to \infty \quad \Longleftarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0$

Definition:(Komplexität)
Sei $(P(n))$ eine Familie von Problemen, die für große $n \in \mathbb{N}$ gelöst werden sollen und **A** die Menge aller Algorithmen, die $(P(n))$ lösen.
Der Aufwand von einem Algorithmus $\mathcal{A} \in \mathbf{A}$ sei $L_{\mathcal{A}}(n)$.
Dann heißt die größte Zahl $\mathcal{K}(n)$ mit

$$\mathcal{K}(n) \leq L_{\mathcal{A}}(n) \qquad \forall \mathcal{A} \in \mathbf{A}$$

die Komplexität des Problems / der Problemfamilie $(P)$.

Definition:(Komplexität)
Sei $(P(n))$ eine Familie von Problemen, die für große $n \in \mathbb{N}$ gelöst werden sollen und **A** die Menge aller Algorithmen, die $(P(n))$ lösen.
Der Aufwand von einem Algorithmus $\mathcal{A} \in \mathbf{A}$ sei $L_{\mathcal{A}}(n)$.
Dann heißt die größte Zahl $\mathcal{K}(n)$ mit

$$\mathcal{K}(n) \leq L_{\mathcal{A}}(n) \qquad \forall \mathcal{A} \in \mathbf{A}$$

die Komplexität des Problems / der Problemfamilie $(P)$.

Definition (Effizienz) in der Numerischen Mathematik
Ein Algorithmus $\mathcal{A} \in \mathbf{A}$ heißt effizient, falls

$$L_{\mathcal{A}}(n) = \Theta(\mathcal{K}(n))$$

Definition:(Komplexität)
Sei $(P(n))$ eine Familie von Problemen, die für große $n \in \mathbb{N}$ gelöst werden sollen und **A** die Menge aller Algorithmen, die $(P(n))$ lösen.
Der Aufwand von einem Algorithmus $\mathcal{A} \in \mathbf{A}$ sei $L_{\mathcal{A}}(n)$.
Dann heißt die größte Zahl $\mathcal{K}(n)$ mit

$$\mathcal{K}(n) \leq L_{\mathcal{A}}(n) \qquad \forall \mathcal{A} \in \mathbf{A}$$

die Komplexität des Problems / der Problemfamilie $(P)$.

Definition (Effizienz) in der Numerischen Mathematik
Ein Algorithmus $\mathcal{A} \in \mathbf{A}$ heißt effizient, falls

$$L_{\mathcal{A}}(n) = \Theta(\mathcal{K}(n))$$

Bem.: Definition in der Theoretischen Informatik: $L_{\mathcal{A}}(n) = \mathcal{O}(n^p)$

# Sortieren von $n$ Zahlen

Problemklasse: Sortieren von $n$ ganzen Zahlen

Problemgröße: $n$

Aufwandsmaß: Anzahl der Vergleiche

# Sortieren von $n$ Zahlen

Problemklasse: Sortieren von $n$ ganzen Zahlen

Problemgröße: $n$

Aufwandsmaß: Anzahl der Vergleiche

Algorithmus Merge-Sort:

$$L_{\mathcal{A}}(n) \leq n \log_2 n$$

Komplexität:

$$\mathcal{K}(n) \geq \frac{1}{4} n \log_2 n$$

# Sortieren von $n$ Zahlen

Problemklasse: Sortieren von $n$ ganzen Zahlen

Problemgröße: $n$

Aufwandsmaß: Anzahl der Vergleiche

Algorithmus Merge-Sort:

$$L_{\mathcal{A}}(n) \leq n \log_2 n$$

Komplexität:

$$\mathcal{K}(n) \geq \frac{1}{4} n \log_2 n$$

Merge-Sort ist ein effizienter Algorithmus!

# Problem: Berechnung von $ggT(a, b)$

**Definition:** (größter gemeinsamer Teiler – ggT)

Eine Zahl $d$, die zwei ganze Zahlen $a, b \in \mathbb{N}$ teilt ($d|a$ und $d|b$), heißt gemeinsamer Teiler von $a$ und $b$.
Die größte positive Zahl $d$, die gemeinsamer Teiler von $a$ und $b$ ist, heißt größter gemeinsamer Teiler von $a$ und $b$ oder kurz $ggT(a, b)$.

**Problem (P):**
Berechne $ggT(a, b)$ für Eingabedaten $a \geq b \in N \setminus \{0\}$

# Ausprobieren: Algorithmus 1

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

# Ausprobieren: Algorithmus 1

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

*ggT := 1*

*for i = 2:b*
  *if i|a and i|b*
    *ggT := i*
  *endif*
*endfor*

*return ggT*

# Ausprobieren: Algorithmus 1

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

*ggT := 1*

*for i = 2:b*
  *if i|a and i|b*
    *ggT := i*
  *endif*
*endfor*

*return ggT*

Aufwandsmaß: Anzahl der Divisionen

# Ausprobieren: Algorithmus 1

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

*ggT := 1*

*for i = 2:b*
  *if i|a and i|b*
    *ggT := i*
  *endif*
*endfor*

*return ggT*

Aufwandsmaß: Anzahl der Divisionen
Aufwand: $2(b-1)$

# Geschickteres Ausprobieren: Algorithmus 2

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

# Geschickteres Ausprobieren: Algorithmus 2

```
Input: positive Zahlen a >= b>0
Output: ggT(a,b)

for i := b:2
   if i|a and i|b
      return i
   endif
endfor

return 1
```

# Geschickteres Ausprobieren: Algorithmus 2

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

```
for i := b:2
   if i|a and i|b
      return i
   endif
endfor

return 1
```

Aufwandsmaß: Anzahl der Divisionen

# Geschickteres Ausprobieren: Algorithmus 2

```
Input: positive Zahlen a >= b>0
Output: ggT(a,b)

for i := b:2
   if i|a and i|b
      return i
   endif
endfor

return 1
```

Aufwandsmaß: Anzahl der Divisionen

Aufwandsschranke: $2(b-1)$

Definition (Kongruenzen)

Der bei Division von $a$ durch $b$ bleibende Rest heißt

$$a \bmod b = a - \lfloor a/b \rfloor b \; .$$

$m, n \in \mathbb{Z}$ heißen kongruent modulo $b$, falls $m \bmod b = n \bmod b$.

Definition (Kongruenzen)

Der bei Division von $a$ durch $b$ bleibende Rest heißt

$$a \bmod b = a - \lfloor a/b \rfloor b \, .$$

$m, n \in \mathbb{Z}$ heißen kongruent modulo $b$, falls $m \bmod b = n \bmod b$.

Satz: (Rekursionssatz für den ggT)

Es gilt

$$ggT(a, b) = ggT(b, a \bmod b) \qquad \forall a, b \in \mathbb{N} \, .$$

Aus der Definition $\mathrm{mod}(a,b) = a - \lfloor a/b \rfloor b$ ergibt sich

$$a = \lfloor a/b \rfloor b + \mathrm{mod}(a,b) \ .$$

Damit erhält man

$$d|a \text{ und } d|b \quad \Longleftrightarrow \quad d|b \text{ und } d|\ \mathrm{mod}\ (a,b) \ .$$

Die Mengen aller gemeinsamen Teiler von $a$, $b$ und $b$, $\mathrm{mod}(a,b)$ stimmen also überein und damit auch deren größtes Element, d.h.

$$ggT(a,b) = ggT(b, a \bmod b) \qquad \forall a,b \in \mathbb{N} \ .$$

# Umsetzung: Der Euklidische Algorithmus

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

```
m = a;
n = b;
while n>0
   r = m modulo n
   m = n
   n = r
endwhile

return m
```

# Umsetzung: Der Euklidische Algorithmus

*Input: positive Zahlen a >= b>0*
*Output: ggT(a,b)*

```
m = a;
n = b;
while n>0
   r = m modulo n
   m = n
   n = r
endwhile

return m
```

Satz:
Der Euklidische Algorithmus liefert nach endlich vielen Schritten das richtige Ergebnis.

Definition (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 0\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

Definition (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 0\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

Lemma
Es sei $a > b$. Terminiert der Euklidische Algorithmus nach genau $k \geq 0$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.

Induktionsanfang: Termination nach $k = 0$ Schritten impliziert $b = 0 = F_0$. Nach Voraussetzung ist dann $a > b = 0$ also $a \geq 1 = F_1$.

Induktionsschritt: Im ersten Schritt werden $m = b$ und $n = \mathrm{mod}(a, b)$ berechnet. Nach Voraussetzung sind zur Bestimmung von $ggT(m, n)$ dann noch weitere $k - 1$ Schritte nötig. Nach Induktionsvoraussetzung gilt

$$b = m \geq F_k, \qquad \mathrm{mod}(a, b) = n \geq F_{k-1} \ .$$

Also ist $a \geq F_{k+1}$ zu zeigen. Aus $a > b > 0$ folgt $\lfloor a/b \rfloor \geq 1$ und damit

$$b + \mathrm{mod}(a, b) = b + (a - \lfloor a/b \rfloor b) \leq a.$$

Daraus folgt

$$a \geq b + \mathrm{mod}(a, b) \geq F_k + F_{k-1} = F_{k+1}.$$

Definition (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 1\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

**Definition** (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 1\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

**Lemma**
Es sei $a > b$. Terminiert der Euklidische Algorithmus nach genau $k \geq 1$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.

**Definition** (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 1\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

**Lemma**
Es sei $a > b$. Terminiert der Euklidische Algorithmus nach genau $k \geq 1$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.

**Satz** (Lamé, 1848)
Gilt $a > b$ und ist $k$ die kleinste Zahl aus $\mathbb{N}$, so dass $b < F_{k+1}$, so terminiert der Euklidische Algorithmus nach höchstens $k$ Schritten.

**Definition** (Fibonacchi–Zahlen)
Die durch die Drei–Term–Rekursion

$$F_{n+1} - F_n - F_{n-1} = 0\,, \qquad F_0 = 1\,, \quad F_1 = 1$$

definierten Zahlen $F_k$, $k = 0, 1, \ldots$ heißen Fibonacchi–Zahlen.

**Lemma**
Es sei $a > b$. Terminiert der Euklidische Algorithmus nach genau $k \geq 1$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.

**Satz** (Lamé, 1848)
Gilt $a > b$ und ist $k$ die kleinste Zahl aus $\mathbb{N}$, so dass $b < F_{k+1}$, so terminiert der Euklidische Algorithmus nach höchstens $k$ Schritten.

**Bemerkung:**
Im Falle $b = F_k$, $a = F_{k+1}$ braucht man $k$ Schritte.

Formel von Moivre-Binet

$$F_k = \frac{1}{\sqrt{5}} \left( \phi^k - (1-\phi)^k \right) \geq \phi^{k-1}, \qquad \phi = \frac{1+\sqrt{5}}{2} \ .$$

Satz:

Der Euklidische Algorithmus termiert nach höchstens $\log_\Phi(b) + 1$ Schritten.

Formel von Moivre-Binet

$$F_k = \frac{1}{\sqrt{5}} \left( \phi^k - (1-\phi)^k \right) \geq \phi^{k-1}, \qquad \phi = \frac{1+\sqrt{5}}{2} .$$

Satz:
Der Euklidische Algorithmus termiert nach höchstens $\log_\Phi(b) + 1$ Schritten.

Bemerkung:
Das ist ein qualitativer Sprung gegenüber dem naiven Algorithmus!
Beispiel: $\log_\Phi(10.000) < 20 \ll 10.000$