Stabilitätsabschätzungen **Vorlesung vom 4.12.15**

Auswertungsbäume zur systematischen Stabilitätsabschätzung

Auswertungsbaum: Knoten, gerichtete Kanten, Wurzel, Blätter

Zerlegung in Teilbäume

Von den Blättern zur Wurzel:

rekursive Funktionsauswertung und Stabilitätsabschätzung

Theoretische Grundlage: Satz 7.6 und Satz 7.9

Beispiele.

Summationsalgorithmen

Rekursive Summation, Auswertungsbaum, Stbilitätsanalyse

Hierarchische Summation.

bisher:

Auswirkung von (Rundungs)-Fehlern auf das Ergebnis:

Kondition eines Problems:	Eingabefehler
Stabilität eines Algorithmus:	Auswertungsfehler

bisher:

Auswirkung von (Rundungs)-Fehlern auf das Ergebnis:

Kondition eines Problems:	Eingabefehler
Stabilität eines Algorithmus:	Auswertungsfehler

jetzt:

Rechenaufwand:

Komplexität eines Problems

Aufwand und Effizienz eines Algorithmus

Komplexität und Effizienz

Komplexität und Effizienz

Wie lange muß ich auf das Ergebnis warten?

Komplexität und Effizienz

Wie lange muß ich auf das Ergebnis warten?

a) Ist das Problem schwierig? (Komplexität)

Komplexität und Effizienz

Wie lange muß ich auf das Ergebnis warten?

a) Ist das Problem schwierig? (Komplexität)

b) Ist mein Algorithmus zu langsam? (Effizienz)

Komplexität und Effizienz

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:

Komplexitätstheorie, Berechenbare Funktionen

Aufwand (Laufzeit) eines Algorithmus

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

Aufwandsmaß: reine Rechenzeit

zusätzliche Parameter:

Implementierung, Programmiersprache, Compiler, Prozessor,...

Aufwand (Laufzeit) eines Algorithmus

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

Aufwandsmaß: reine Rechenzeit

zusätzliche Parameter:

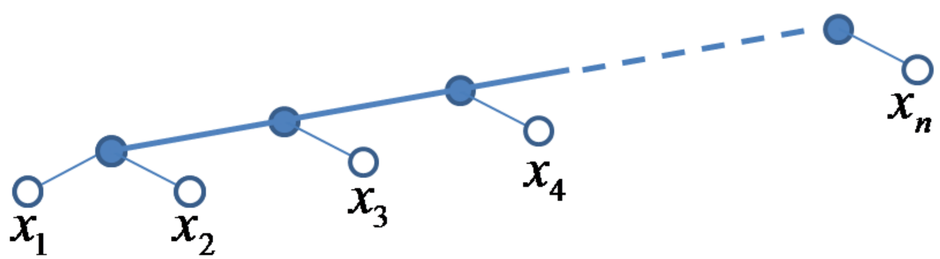
Implementierung, Programmiersprache, Compiler, Prozessor,...

Aufwandsmaß: Anzahl dominanter Operationen (problemabhängig!)

implementierungsunabhängige Resultate

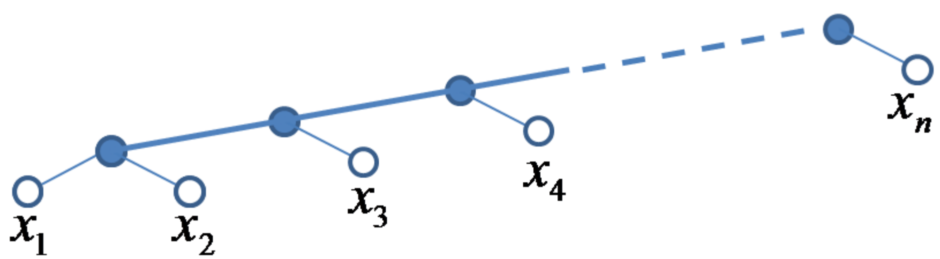
Rekursive Summation positiver Zahlen

rekursive Summation: `S = a[1]; for i=2:1:m S = S + a[i]; end`



Rekursive Summation positiver Zahlen

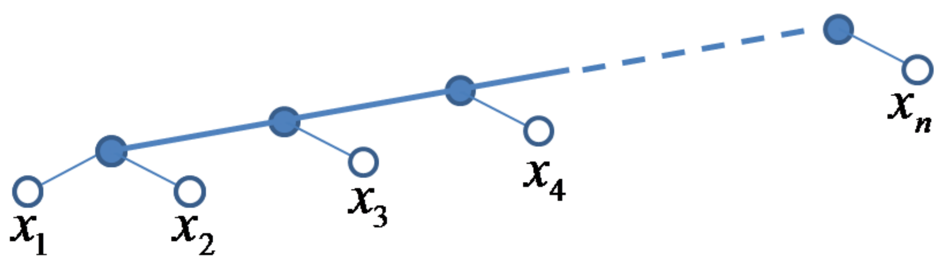
rekursive Summation: `S = a[1]; for i=2:1:m S = S + a[i]; end`



Aufwandsmaß: Anzahl der Additionen

Rekursive Summation positiver Zahlen

rekursive Summation: `S = a[1]; for i=2:1:m S = S + a[i]; end`



Aufwandsmaß: Anzahl der Additionen

Aufwand des rekursiven Algorithmus: $n-1$

Hierarchische Summation

$$
\begin{array}{ccccccccccccccc}
x_1 & & x_2 & & x_3 & & x_4 & & x_5 & & x_6 & & x_7 & & x_8 \\
& \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & \\
& s_1^{(1)} & & & & s_2^{(1)} & & & & s_3^{(1)} & & & & s_4^{(1)} & \\
& & \searrow & & \swarrow & & & & & & \searrow & & \swarrow & & \\
& & & s_1^{(2)} & & & & & & & & s_2^{(2)} & & & \\
& & & & \searrow & & & & & & \swarrow & & & & \\
& & & & & & & s_1^{(3)} & & & & & & &
\end{array}
$$

Hierarchische Summation

$$
\begin{array}{ccccccccccccccc}
x_1 & & x_2 & & x_3 & & x_4 & & x_5 & & x_6 & & x_7 & & x_8 \\
& \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & \\
& s_1^{(1)} & & & & s_2^{(1)} & & & & s_3^{(1)} & & & & s_4^{(1)} & \\
& & \searrow & & \swarrow & & & & & & \searrow & & \swarrow & & \\
& & & s_1^{(2)} & & & & & & & & s_2^{(2)} & & & \\
& & & & \searrow & & & & & & \swarrow & & & & \\
& & & & & & & s_1^{(3)} & & & & & & &
\end{array}
$$

Aufwandsmaß: Anzahl der Additionen

Hierarchische Summation

$$
\begin{tikzcd}[column sep=tiny]
x_1 \arrow[dr] & & x_2 \arrow[dl] & & x_3 \arrow[dr] & & x_4 \arrow[dl] & & x_5 \arrow[dr] & & x_6 \arrow[dl] & & x_7 \arrow[dr] & & x_8 \arrow[dl] \\
& s_1^{(1)} \arrow[drr] & & & & s_2^{(1)} \arrow[dll] & & & & s_3^{(1)} \arrow[drr] & & & & s_4^{(1)} \arrow[dll] & \\
& & & s_1^{(2)} \arrow[drrrr] & & & & & & & & s_2^{(2)} \arrow[dllll] & & & \\
& & & & & & & s_1^{(3)} & & & & & & &
\end{tikzcd}
$$

Aufwandsmaß: Anzahl der Additionen

Aufwand des hierarchischen Algorithmus: $(n = 2^J)$:

$$
2^{J-1} + \cdots + 1 = \sum_{j=0}^{J-1} 2^j
$$

Hierarchische Summation

$$
\begin{array}{cccccccccccccccc}
x_1 & & x_2 & & x_3 & & x_4 & & x_5 & & x_6 & & x_7 & & x_8 \\
& \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & & & & \searrow\swarrow & \\
& s_1^{(1)} & & & & s_2^{(1)} & & & & s_3^{(1)} & & & & s_4^{(1)} & \\
& & \searrow & & \swarrow & & & & & & \searrow & & \swarrow & & \\
& & & s_1^{(2)} & & & & & & & & s_2^{(2)} & & & \\
& & & & \searrow & & & & & & \swarrow & & & & \\
& & & & & & & s_1^{(3)} & & & & & & &
\end{array}
$$

Aufwandsmaß: Anzahl der Additionen

Aufwand des hierarchischen Algorithmus: $(n = 2^J)$:

$$2^{J-1} + \cdots + 1 = \sum_{j=0}^{J-1} 2^j = \frac{2^J - 1}{2 - 1} = 2^J - 1$$

Hierarchische Summation

$$
\begin{tikzcd}[column sep=small]
x_1 \arrow[dr] & & x_2 \arrow[dl] & & x_3 \arrow[dr] & & x_4 \arrow[dl] & & x_5 \arrow[dr] & & x_6 \arrow[dl] & & x_7 \arrow[dr] & & x_8 \arrow[dl] \\
& s_1^{(1)} \arrow[drr] & & & & s_2^{(1)} \arrow[dll] & & & & s_3^{(1)} \arrow[drr] & & & & s_4^{(1)} \arrow[dll] & \\
& & & s_1^{(2)} \arrow[drrrr] & & & & & & & & s_2^{(2)} \arrow[dllll] & & & \\
& & & & & & & s_1^{(3)} & & & & & & &
\end{tikzcd}
$$

Aufwandsmaß: Anzahl der Additionen

Aufwand des hierarchischen Algorithmus: $(n = 2^J)$:

$$2^{J-1} + \cdots + 1 = \sum_{j=0}^{J-1} 2^j = \frac{2^J - 1}{2 - 1} = 2^J - 1 = n - 1$$

Komplexität der Summation positiver Zahlen

Kann es einen Algorithmus geben, der weniger als $n-1$ Additionen benötigt?

Komplexität der Summation positiver Zahlen

Kann es einen Algorithmus geben, der weniger als $n-1$ Additionen benötigt?

Nein!

Komplexität der Summation positiver Zahlen

Kann es einen Algorithmus geben, der weniger als $n-1$ Additionen benötigt?

Nein!

Folgerung: Die Komplexität der Addition von n Zahlen ist $n-1$.

Aufwand und Komplexität

Definition 2.4

Problem (P), Algorithmen $\mathcal{A}(P)$ zur Lösung von (P)

Eingabedaten der Länge n, Referenzoperation (problemspezifisch!)

Aufwand und Komplexität

Definition 2.4

Problem (P), Algorithmen $\mathcal{A}(P)$ zur Lösung von (P)

Eingabedaten der Länge n, Referenzoperation (problemspezifisch!)

Der Aufwand $T_A(n)$ eines Algorithmus $A \in \mathcal{A}(P)$ ist das

Maximum der benötigten Anzahl von Referenzoperationen

über alle zulässigen Eingabedaten der Länge n.

Aufwand und Komplexität

Definition 2.4

Problem (P), Algorithmen $\mathcal{A}(P)$ zur Lösung von (P)

Eingabedaten der Länge n, Referenzoperation (problemspezifisch!)

Der Aufwand $T_A(n)$ eines Algorithmus $A \in \mathcal{A}(P)$ ist das

Maximum der benötigten Anzahl von Referenzoperationen

über alle zulässigen Eingabedaten der Länge n.

Definition 2.5

Die Komplexität $\mathcal{K}_P(n)$ von (P) ist

$$\mathcal{K}_P(n) = \inf_{A \in \mathcal{A}(P)} T_A(P)$$

Effiziente Algorithmen

Problem (P), Algorithmus $A \in \mathcal{A}(P)$ zur Lösung von (P)

Effiziente Algorithmen

Problem (P), Algorithmus $A \in \mathcal{A}(P)$ zur Lösung von (P)

Effizienz eines Algorithmus' A:

Theoretische Informatik: $T_A(n) = O(n^p)$

Numerische Mathematik: $T_A(n) = O(\mathcal{K}_P(n))$

Effiziente Algorithmen

Problem (P), Algorithmus $A \in \mathcal{A}(P)$ zur Lösung von (P)

Effizienz eines Algorithmus' A:

Theoretische Informatik: $T_A(n) = O(n^p)$

Numerische Mathematik: $T_A(n) = O(\mathcal{K}_P(n))$

Definition: Landau-Symbol $O(\cdot)$ für $f(n)$, $g(n) \to \infty$ für $n \to \infty$

$$f(n) = O(g(n)) \quad \text{für } n \to \infty \qquad \Longleftrightarrow \qquad \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

Effiziente Algorithmen

Problem (P), Algorithmus $A \in \mathcal{A}(P)$ zur Lösung von (P)

Effizienz eines Algorithmus' A:

Theoretische Informatik: $T_A(n) = O(n^p)$

Numerische Mathematik: $T_A(n) = O(\mathcal{K}_P(n))$

Definition: Landau-Symbol $O(\cdot)$ für $f(n)$, $g(n) \to \infty$ für $n \to \infty$

$$f(n) = O(g(n)) \quad \text{für } n \to \infty \qquad \Longleftrightarrow \qquad \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

Beispiel: $18n^3 + 3n^2 + \sin(e^n) = O(n^3)$

Sortieren

gegeben: $z_1,\ z_2,\ \ldots,\ z_n \in \mathbb{R}$

gesucht: Permutation (Umordnung) π: $z_{\pi(1)} \leq z_{\pi(2)} \leq \cdots \leq z_{\pi(n)}$

Sortieren

gegeben: $z_1, \; z_2, \; \ldots, \; z_n \in \mathbb{R}$

gesucht: Permutation (Umordnung) π: $z_\pi(1), \; z_\pi(2), \; \ldots, \; z_\pi(n)$

Algorithmus: TumbSort: Alle Umordnungen durchprobieren.

Sortieren

gegeben: $z_1,\ z_2,\ \dots,\ z_n \in \mathbb{R}$

gesucht: Permutation (Umordnung) π: $z_\pi(1),\ z_\pi(2),\ \dots,\ z_\pi(n)$

Algorithmus: TumbSort: Alle Umordnungen durchprobieren.

Aufwandsmaß: Vergleich zweier Zahlen

Sortieren

gegeben: $z_1,\ z_2,\ \ldots,\ z_n \in \mathbb{R}$

gesucht: Permutation (Umordnung) π: $z_\pi(1),\ z_\pi(2),\ \ldots,\ z_\pi(n)$

Algorithmus: TumbSort: Alle Umordnungen durchprobieren.

Aufwandsmaß: Vergleich zweier Zahlen

Aufwand von TumbSort: $O(n^n)$

Sortieren

gegeben: $z_1,\ z_2,\ \ldots,\ z_n \in \mathbb{R}$

gesucht: Permutation (Umordnung) π: $z_\pi(1),\ z_\pi(2),\ \ldots,\ z_\pi(n)$

Algorithmus: TumbSort: Alle Umordnungen durchprobieren.

Aufwandsmaß: Vergleich zweier Zahlen

Aufwand von TumbSort: $O(n^n)$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$S = \{7, 1, 5\}\ ,\ k = 3: \qquad SMax = \max\{7, 1, 5\} = 7, \qquad iMax = 1$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$S = \{7, 1, 5\}\ ,\ k = 3: \qquad SMax = \max\{7, 1, 5\} = 7, \qquad iMax = 1$$

$$S = \{5, 1, 7\}\ ,\ k = 2: \qquad SMax = \max\{5, 1\} = 5, \qquad iMax = 1$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$S = \{7,1,5\}\ ,\ k = 3: \qquad SMax = \max\{7,1,5\} = 7, \qquad iMax = 1$$
$$S = \{5,1,7\}\ ,\ k = 2: \qquad SMax = \max\{5,1\} = 5, \qquad iMax = 1$$
$$S = \{1,5,7\}$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$S = \{7,1,5\}\ ,\ k = 3: \qquad SMax = \max\{7,1,5\} = 7, \qquad iMax = 1$$
$$S = \{5,1,7\}\ ,\ k = 2: \qquad SMax = \max\{5,1\} = 5, \qquad iMax = 1$$
$$S = \{1,5,7\}$$

Aufwand von BubbleSort:

$$(n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n-1} k$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$
\begin{array}{lll}
S = \{7,1,5\}\ ,\ k = 3: & SMax = \max\{7,1,5\} = 7, & iMax = 1 \\
S = \{5,1,7\}\ ,\ k = 2: & SMax = \max\{5,1\} = 5, & iMax = 1 \\
S = \{1,5,7\} & &
\end{array}
$$

Aufwand von BubbleSort:

$$(n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n-1} k = \tfrac{1}{2} n(n-1)$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$S = \{7, 1, 5\}\ ,\ k = 3: \qquad SMax = \max\{7, 1, 5\} = 7, \qquad iMax = 1$$
$$S = \{5, 1, 7\}\ ,\ k = 2: \qquad SMax = \max\{5, 1\} = 5, \qquad iMax = 1$$
$$S = \{1, 5, 7\}$$

Aufwand von BubbleSort:

$$(n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n-1} k = \tfrac{1}{2} n(n-1) = O(n^2)$$

BubbleSort

Algorithmus: BubbleSort: Sukzessive Maximierung

Beispiel:

$$
\begin{array}{lll}
S = \{7,1,5\}\ ,\ k=3: & SMax = \max\{7,1,5\} = 7, & iMax = 1 \\
S = \{5,1,7\}\ ,\ k=2: & SMax = \max\{5,1\} = 5, & iMax = 1 \\
S = \{1,5,7\} & &
\end{array}
$$

Aufwand von BubbleSort:

$$(n-1)+(n-2)+\cdots+1 = \sum_{k=1}^{n-1} k = \tfrac{1}{2}n(n-1) = O(n^2) \ll O(n^n)$$

MergeSort

$$
\begin{tikzcd}[column sep=small]
 & & & & & & & U \arrow[dllll] \arrow[drrrr] & & & & & & & \\
 & & & U_1^{(1)} \arrow[dll] \arrow[drr] & & & & & & & & U_2^{(1)} \arrow[dll] \arrow[drr] & & & \\
 & U_1^{(2)} \arrow[dl] \arrow[dr] & & & & U_2^{(2)} \arrow[dl] \arrow[dr] & & & & U_3^{(2)} \arrow[dl] \arrow[dr] & & & & U_4^{(2)} \arrow[dl] \arrow[dr] & \\
U_1^{(3)} & & U_2^{(3)} & & U_3^{(3)} & & U_4^{(3)} & & U_5^{(3)} & & U_6^{(3)} & & U_7^{(3)} & & U_8^{(3)}
\end{tikzcd}
$$

MergeSort

$$
\begin{array}{ccccccccccccccc}
 & & & & & & & U & & & & & & & \\
 & & & \swarrow & & & & & & & & \searrow & & & \\
 & & & U_1^{(1)} & & & & & & & & U_2^{(1)} & & & \\
 & \swarrow & & & & \searrow & & & & \swarrow & & & & \searrow & \\
 & U_1^{(2)} & & & & U_2^{(2)} & & & & U_3^{(2)} & & & & U_4^{(2)} & \\
 \swarrow & & \searrow & & \swarrow & & \searrow & & \swarrow & & \searrow & & \swarrow & & \searrow \\
U_1^{(3)} & & U_2^{(3)} & & U_3^{(3)} & & U_4^{(3)} & & U_5^{(3)} & & U_6^{(3)} & & U_7^{(3)} & & U_8^{(3)}
\end{array}
$$

Aufwand von MergeSort: $T_{\mathrm{MergeSort}}(n) \leq n \log_2 n$

MergeSort

$$
\begin{tikzcd}[column sep=small]
 & & & & & & & U \arrow[dllll] \arrow[drrrr] & & & & & & & \\
 & & & U_1^{(1)} \arrow[dll] \arrow[drr] & & & & & & & & U_2^{(1)} \arrow[dll] \arrow[drr] & & & \\
 & U_1^{(2)} \arrow[dl] \arrow[dr] & & & & U_2^{(2)} \arrow[dl] \arrow[dr] & & & & U_3^{(2)} \arrow[dl] \arrow[dr] & & & & U_4^{(2)} \arrow[dl] \arrow[dr] & \\
U_1^{(3)} & & U_2^{(3)} & & U_3^{(3)} & & U_4^{(3)} & & U_5^{(3)} & & U_6^{(3)} & & U_7^{(3)} & & U_8^{(3)}
\end{tikzcd}
$$

Aufwand von MergeSort: $T_{\text{MergeSort}}(n) \leq n \log_2 n$ optimal!

Berechnung von $ggT(a, b)$

Definition 4.1 (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: TumbGGT (Algorithmus 4.3)

```
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: TumbGGT (Algorithmus 4.3)

```
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 mit Rest

Ausprobieren: TumbGGT (Algorithmus 4.3)

```
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 mit Rest   Aufwand: $2(b-1)$

Geschickteres Ausprobieren: TumbGGT++

```
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: TumbGGT++

```
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: TumbGGT++

```
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: $2(b-1)$

Strukturelle Einsicht

Definition 4.6 (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 $m \bmod b = n \bmod b$.

modulo-Funktion: $\quad \text{mod}(a, b) = a \bmod b$

Strukturelle Einsicht

Definition 4.6 (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 $m \bmod b = n \bmod b$.

modulo-Funktion: $\quad \mathsf{mod}(a, b) = a \bmod b$

Lemma 4.12 (Rekursionssatz für den ggT)
Es gilt

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

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 terminiert nach höchstens b Schritten.

Aufwandsanalyse

Definition (Fibonacci–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 Fibonacci–Zahlen.

Aufwandsanalyse

Definition (Fibonacci–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 Fibonacci–Zahlen.

Lemma 4.14
Es sei $a > b$. Terminiert der Euklidische Algorithmus nach genau $k \geq 1$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.

Aufwandsanalyse

Definition (Fibonacci–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 Fibonacci–Zahlen.

Lemma 4.14
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 4.15 (Lamé, 1848)
Gilt $a > b$ und $b < F_{k+1}$ mit $k \in \mathbb{N}$, so terminiert der Euklidische Algorithmus nach höchstens k Schritten.

Aufwandsanalyse

Definition (Fibonacci–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 Fibonacci–Zahlen.

Lemma 4.14
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 4.15 (Lamé, 1848)
Gilt $a > b$ und $b < F_{k+1}$ mit $k \in \mathbb{N}$, so terminiert der Euklidische Algorithmus nach höchstens k Schritten.

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

Aufwandsschranke für den Euklidischen Algorithmus

Moivre-Binet: $F_k = (\phi^k - (1-\phi)^k)/\sqrt{5} \quad \phi = \frac{1+\sqrt{5}}{2}$

Aufwandsschranke für den Euklidischen Algorithmus

Moivre-Binet: $F_k = (\phi^k - (1-\phi)^k)/\sqrt{5} \quad \phi = \frac{1+\sqrt{5}}{2}$

Satz 4.16
Für den Aufwand $T_{\mathrm{EA}}(n)$ des Euklidische Algorithmus gilt

$$T_{\mathrm{EA}}(n) \leq \log_\phi(b) + 1$$

Aufwandsschranke für den Euklidischen Algorithmus

Moivre-Binet: $F_k = (\phi^k - (1-\phi)^k)/\sqrt{5} \quad \phi = \frac{1+\sqrt{5}}{2}$

Satz 4.16
Für den Aufwand $T_{\mathrm{EA}}(n)$ des Euklidische Algorithmus gilt

$$T_{\mathrm{EA}}(n) \leq \log_\phi(b) + 1$$

Bemerkung:

Das ist eine exponentielle Verbesserung gegenüber TUMBGGT!

Beispiel: $\log_\Phi(10.000) < 20 \ll 10.000$