Prof. Andreas Goerdt
M. Pippig, M. Riedel, U. Schönke

TU Chemnitz
Wintersemester 2003/2004

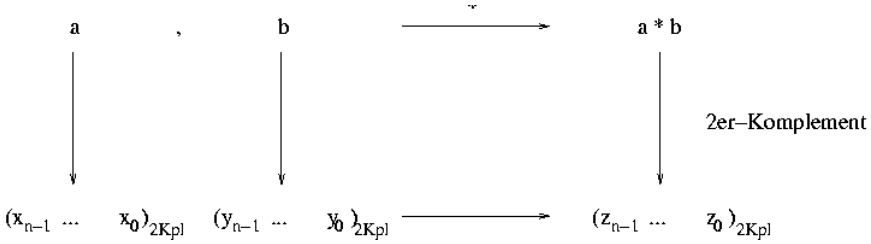
Algorithmen und Programmierung

5. Übung – Lösungsvorschläge

1. Aufgabe:

   Ausführlicher Beweis für die Multiplikation:

   Veranschaulichung



   Wir arbeiten zunächst im Binärsystem (nicht mit dem Zweier-Komplement).

   Voraussetzung: Sei $-2^{n-1} \leq a,\ b \leq 2^{n-1}-1$ und $x = \left(x_{n-1}\ \ldots\ x_0\right)_2, y = \left(y_{n-1}\ \ldots\ y_0\right)_2$ (x und y sind die binären Interpretationen von a und b.). Dann ist $\left(x_{n-1}\ \ldots\ x_0\right)_2 \cdot \left(y_{n-1}\ \cdots\ y_0\right)_2 = \left(z_{m-1}\ \ldots\ z_0\right)_2$ das Ergebnis der binären Multiplikation.

   1) Es gilt ganz allgemein

   $$\mathrm{MOD}\left(a \cdot b, c\right) = \mathrm{MOD}\left(\mathrm{MOD}\left(a,c\right) \cdot \mathrm{MOD}\left(b,c\right), c\right) \text{ für } c > 0$$

   2) Es gilt weiterhin die Def. des Zweierkomplementes:

   $$\begin{array}{lcll} \left(x_{n-1}\ \ldots\ x_0\right)_2 = \mathrm{MOD}\left(a, 2^n\right) & \Leftrightarrow & \left(x_{n-1}\ \ldots\ x_0\right)_{2Kpl} & = a \\ \left(y_{n-1}\ \ldots\ y_0\right)_2 = \mathrm{MOD}\left(b, 2^n\right) & \Leftrightarrow & \left(y_{n-1}\ \ldots\ y_0\right)_{2Kpl} & = b \end{array}$$

   3) Die Reste modulo 2^n bleiben gleich, sofern ein Vielfaches (positiv oder negativ) von 2^n zu einem Wert hinzuaddiert wird. Im Binärsystem steht jede weitere Stelle m, $m \geq n$ für ein Vielfaches von 2^n (z. B.: $m = n+1 = 2^{n+1} = 2 \cdot 2^n$). Somit ist ganz allgemein

   $$\begin{array}{rcl} \mathrm{MOD}\left(\left(z_{2n-1}\, z_{2n-2} \ldots\, z_1\, z_0\right)_2, 2^n\right) & = & \mathrm{MOD}\left(\left(z_{2n-2} \ldots\, z_1\, z_0\right)_2, 2^n\right) \\ & = & \vdots \\ & = & \mathrm{MOD}\left(\left(z_n \ldots\, z_1\, z_0\right)_2, 2^n\right) \\ & = & \mathrm{MOD}\left(\left(z_{n-1} \ldots\, z_1\, z_0\right)_2, 2^n\right) \\ & = & \left(z_{n-1} \cdots\, z_1\, z_0\right)_2 \end{array}$$

Sind also die letzten n Ziffern von zwei Zahlen gleich, so haben beide modulo 2^n den gleichen Rest.

Da nach obiger Voraussetzung $(x_{n-1} \dots x_0)_2 \cdot (y_{n-1} \dots y_0)_2 = (z_{2n-1} \dots z_0)_2$ das Ergebnis der binären Multiplikation von a und b ist, gilt:

$$(z_{n-1} \dots z_0)_2 = \text{MOD}\left((x_{n-1} \dots x_0)_2 \cdot (y_{n-1} \dots y_0)_2, 2^n\right)$$

Ziel: Zeigen, dass

$$(z_{n-1} \dots z_0)_{2Kpl} = a \cdot b$$

Dies gilt genau dann, wenn

$$(z_{n-1} \dots z_0)_2 = \text{MOD}(a \cdot b, 2^n)$$

Eigentlicher Beweis:

Mit 1) ist

$$\text{MOD}(a \cdot b, 2^n) = \text{MOD}(\text{MOD}(a, 2^n) \cdot \text{MOD}(b, 2^n), 2^n)$$

Mit 2)

$$= \text{MOD}\left((x_{n-1} \dots x_0)_2 \cdot (y_{n-1} \dots y_0)_2, 2^n\right)$$

Mit 3)

$$= (z_{n-1} \dots z_0)_2$$

Daraus folgt

$$(z_{n-1} \dots z_0)_{2Kpl} = a \cdot b$$

nach Definition.

2. Aufgabe:

a)

```
import Prog1Tools.IOTools;

public class EuklidSub {

  public static void main (String[] args) {

    int a=IOTools.readInteger("Bitte a eingeben: ");
    int b=IOTools.readInteger("Bitte b eingeben: ");

    while (a != b ) {      // a <> b?
      if (a > b) {         // a > b
        a = a - b;
      }
      else {               // a < b
        b = b - a;
      }
    }
    System.out.println("ggT ist " + a);
  }
}
```

Seien a_{start} und b_{start} die eingelesenen Werte, von denen der ggT gefunden werden soll. Sei weiterhin M_{start} die Menge der gemeinsamen Teiler von a_{start} und b_{start}. Das größte Element der Menge ist nach Definition der ggT(a_{start}, b_{start}). Seien nun a_l und b_l die Werte der Variablen a und b nach dem l-ten Schleifendurchlauf sowie M_l die Menge der gemeinsamen Teiler von a_l und b_l. Es gilt folgende Invariante:

$$M_l = M_{start} \text{ sowie } a_l, b_l \geq 1 \text{ für } 0 \leq l \leq n$$

n ist die Anzahl der stattfindenden Schleifendurchläufe, abhängig von a_{start} und b_{start}. Diese ist endlich, da die Werte der Variablen a und b im Laufe der Abarbeitung immer kleiner und durch die Differenzbildung irgendwann gleich werden.

Induktionsanfang:

Für $l = 0$ (vor dem ersten Schleifendurchlauf) gilt $M_0 = M_{start}$, da $a_0 = a_{start}$ und $b_0 = b_{start}$. Außerdem gilt $a_0 \geq 1$ und $b_0 \geq 1$, da der Algorithmus den ggT nur für positive Zahlen berechnet.

Induktionsschritt:

Sei $1 \leq l \leq n$ fest. Dann gilt für $l-1$ nach Voraussetzung (Invariante):

$$M_{l-1} = M_{start} \text{ sowie } a_{l-1}, b_{l-1} \geq 1$$

Während des l-ten Schleifendurchlaufs verändern sich die Variablen a und b wie folgt (2 Fälle):

1) $a_{l-1} > b_{l-1}$
   Dann ist $a_l = a_{l-1} - b_{l-1}$ und $b_l = b_{l-1}$.
2) $a_{l-1} < b_{l-1}$
   Dann ist $a_l = a_{l-1}$ und $b_l = b_{l-1} - a_{l-1}$.

$a_{l-1} = b_{l-1}$ kann nicht auftreten, da danach noch (mindestens) ein Schleifendurchlauf kommt ($l \leq n$).

Da wir immer nur den (streng) kleineren Wert vom größeren abziehen, muss die Differenz mindestens 1 betragen, so dass gilt $a_l, b_l \geq 1$.

Sei nun c ein gemeinsamer Teiler von a_{l-1} und b_{l-1}, also $c \in M_{l-1}$. Laut Teilbarkeitseigenschaften (siehe Tafelwerk) gilt:

$$\begin{aligned} c|a_{l-1} \text{ und } c|b_{l-1} \quad &\Rightarrow \quad c| \quad \underbrace{(a_{l-1} - b_{l-1})}_{a_l} \quad \text{(im 1. Fall)} \\ &\Rightarrow \quad c| \quad \underbrace{(b_{l-1} - a_{l-1})}_{b_l} \quad \text{(im 2. Fall)} \end{aligned}$$

Also ist $c|a_l$ und $c|b_l$ und $c \in M_l$. Alle Elemente aus M_{l-1} sind folglich auch in M_l enthalten.

Nun müssen wir noch nachweisen, dass M_l keine zusätzlichen Elemente enthält. Sei daher d ein gemeinsamer Teiler von a_l und b_l, also $d \in M_l$. Laut Teilbarkeitseigenschaften gilt:

$$\begin{aligned} d|a_l \text{ und } d|b_l \quad &\Rightarrow \quad d| \quad \underbrace{(a_l + b_l)}_{a_{l-1}} \quad \text{(im 1. Fall)} \\ &\Rightarrow \quad d| \quad \underbrace{(b_l + a_l)}_{b_{l-1}} \quad \text{(im 2. Fall)} \end{aligned}$$

Also ist $d|a_{l-1}$ und $d|b_{l-1}$ und $d \in M_{l-1}$. Alle Elemente aus M_l sind somit auch in M_{l-1} enthalten. Folglich gilt $M_l = M_{l-1} = M_{start}$.

Wir betrachten nun den n-ten (also letzten) Schleifendurchlauf. Wegen $a_n = b_n$ erfolgt der Abbruch der Schleife. Da $a_n = b_n$, ist a_n auch gemeinsamer Teiler von a_n und b_n, also $a_n \in M_n$. Außerdem ist a_n natürlich das größte Element von M_n.

Quintessenz: Nach dem n-ten Schleifendurchlauf ist die Invariante korrekt und die Schleifenbedingung nicht mehr erfüllt. Folglich liefert der Algorithmus mit a_n den korrekten Wert für ggT(a,b) zurück.

b)

```
import Prog1Tools.IOTools;

public class EuklidDiv {

  public static void main (String[] args) {

    int a=IOTools.readInteger("Bitte a eingeben: ");
    int b=IOTools.readInteger("Bitte b eingeben: ");

    while ((a != 0 ) && (b != 0)) {   // Ende?
      if (a > b) {                    // a > b
        a = a % b;
      }
      else {                          // a <= b
        b = b % a;
      }
    }

    if (a>0) {
      System.out.println("ggT ist " + a);
    }
    else {
      System.out.println("ggT ist " + b);
    }
  }
}
```

Wir wählen die Bezeichner identisch wie bei Teilaufgabe a). Es gilt folgende Invariante:

$$M_l = M_{start} \text{ sowie } a_l + b_l \geq 1 \text{ und } a_l, b_l \geq 0 \text{ für } 0 \leq l \leq n$$

n ist die Anzahl der Schleifendurchläufe. Diese Anzahl ist endlich, da die Werte der Variablen a und b durch die Restbildung immer kleiner werden und irgendwann 0 erreicht wird.

Induktionsanfang:

Für $l = 0$ (vor dem ersten Schleifendurchlauf) gilt $M_0 = M_{start}$, da $a_0 = a_{start}$ und $b_0 = b_{start}$. Außerdem gilt $a_0 + b_0 \geq 1$ und $a_0, b_0 \geq 0$, da der Algorithmus den ggT nur für positive Zahlen berechnet, wobei in dieser Realisierung eine davon gleich 0 sein darf.

Induktionsschritt:

Sei $1 \leq l \leq n$ fest. Dann gilt für $l-1$ nach Voraussetzung (Invariante):

$$M_{l-1} = M_{start} \text{ sowie } a_{l-1} + b_{l-1} \geq 1 \text{ und } a_{l-1}, b_{l-1} \geq 0$$

Während des l-ten Schleifendurchlaufs verändern sich die Variablen a und b wie folgt (2 Fälle):

1) $a_{l-1} > b_{l-1}$
   Dann ist $a_l = \text{MOD}(a_{l-1}, b_{l-1})$ und $b_l = b_{l-1}$.
2) $a_{l-1} \leq b_{l-1}$
   Dann ist $a_l = a_{l-1}$ und $b_l = \text{MOD}(b_{l-1}, a_{l-1})$.

Da wir immer nur einen Wert verändern und dieser als Ergebnis einer Modulo-Operation nicht kleiner als 0 werden kann, gilt $a_l + b_l \geq 1$ und $a_l, b_l \geq 0$.
Sei nun c ein gemeinsamer Teiler von a_{l-1} und b_{l-1}, also $c \in M_{l-1}$. Laut Teilbarkeitseigenschaften gilt:

$$\begin{aligned} c|a_{l-1} \text{ und } c|b_{l-1} &\Rightarrow c|(a_{l-1} - b_{l-1}) \\ &\Rightarrow c|(b_{l-1} - a_{l-1}) \end{aligned}$$

Dies kann man natürlich erweitern, indem wir x bzw. y mal subtrahieren:

$$\begin{aligned} c|a_{l-1} \text{ und } c|b_{l-1} &\Rightarrow c|(a_{l-1} - (x \cdot b_{l-1})) \\ &\Rightarrow c|(b_{l-1} - (y \cdot a_{l-1})) \end{aligned}$$

Setzen wir $x = \text{DIV}(a_{l-1}, b_{l-1})$, dann ist (1. Fall)

$$a_{l-1} - (x \cdot b_{l-1}) = \text{MOD}(a_{l-1}, b_{l-1}) = a_l$$

Analog gilt für $y = \text{DIV}(b_{l-1}, a_{l-1})$ (2. Fall)

$$b_{l-1} - (y \cdot a_{l-1}) = \text{MOD}(b_{l-1}, a_{l-1}) = b_l$$

Also ist $c|a_l$ und $c|b_l$ und $c \in M_l$. Alle Elemente aus M_{l-1} sind folglich auch in M_l enthalten.
Nun müssen wir noch nachweisen, dass M_l keine zusätzlichen Elemente enthält. Sei daher d ein gemeinsamer Teiler von a_l und b_l, also $d \in M_l$. Laut Teilbarkeitseigenschaften gilt:

$$\begin{aligned} d|a_l \text{ und } d|b_l &\Rightarrow d|(a_l + b_l) \\ &\Rightarrow d|(b_l + a_l) \end{aligned}$$

Dies kann man wiederum erweitern, indem wir x bzw. y mal addieren:

$$\begin{aligned} d|a_l \text{ und } d|b_l &\Rightarrow d|(a_l + (x \cdot b_l)) \\ &\Rightarrow d|(b_l + (y \cdot a_l)) \end{aligned}$$

Setzen wir $x = \text{DIV}(a_{l-1}, b_{l-1})$, dann ist (1. Fall: $b_l = b_{l-1}$)

$$a_l + (x \cdot b_l) = \text{MOD}(a_{l-1}, b_{l-1}) + \text{DIV}(a_{l-1}, b_{l-1}) \cdot b_{l-1} = a_{l-1}$$

Analog gilt für $y = \text{DIV}(b_{l-1}, a_{l-1})$ (2. Fall: $a_l = a_{l-1}$)

$$b_l + (y \cdot a_l) = \text{MOD}(b_{l-1}, a_{l-1}) + \text{DIV}(b_{l-1}, a_{l-1}) \cdot a_{l-1} = b_{l-1}$$

Also ist $d|a_{l-1}$ und $d|b_{l-1}$ und $d \in M_{l-1}$. Alle Elemente aus M_l sind somit auch in M_{l-1} enthalten.
Folglich gilt $M_l = M_{l-1} = M_{start}$.
Wir betrachten nun den n-ten (also letzten) Schleifendurchlauf. Da die Schleife nach diesem Durchlauf abgebrochen wird, muss $a_n = 0$ oder $b_n = 0$ sein. Im n-ten Schleifendurchlauf muss also a ein Vielfaches von b oder umgekehrt sein, damit die Modulo-Operation das Ergebnis 0 liefert. Der Wert der Variable, die nicht gleich 0 ist, ist natürlich gemeinsamer Teiler von a_n und b_n und damit das größte Element von M_n.

Quintessenz: Nach dem n-ten Schleifendurchlauf ist die Invariante korrekt und die Schleifenbedingung nicht mehr erfüllt. Folglich liefert der Algorithmus den korrekten Wert als $\text{ggT}(a_{start}, b_{start})$ zurück.

c) Man kann in Java die aktuelle Zeit messen, in Millisekunden z.B. mit der Methode `System.currentTimeMillis()`. Eine Laufzeitmessung könnte also folgendermaßen aussehen:

```
long start = System.currentTimeMillis();

// zu messender Programm-Teil

long stop = System.currentTimeMillis();
long time = stop - start;

System.out.println("Laufzeit " + time + "ms");
```

Einige Beispiele für so gemessene Laufzeiten der Algorithmen:

Eingabe für a	Eingabe für b	Laufzeit EuklidSub	Laufzeit EuklidDIV
1 000 000 000	2	1435 ms	0 ms
353430	395010	0 ms	0 ms
435357934	1	1270 ms	0 ms
345345345	234234234	0 ms	0 ms

Wir sehen, dass diese Werte nicht sehr aussagekräftig sind. Bei Zahlen, deren Differenz sehr groß ist, benötigt der Algorithmus EuklidSub jedoch vergleichsweise lange.

d) Schauen wir uns einige einfache Beispiele an:

a_{start}	b_{start}	EuklidSub	EuklidDiv
10	2	1. a=8 b=2 2. a=6 b=2 3. a=4 b=2 4. a=2 b=2 $\Rightarrow$ 4 Subtraktionen	1. a=0 b=2 $\Rightarrow$ 1 Restbildung
7	10	1. a=7 b=3 2. a=4 b=3 3. a=1 b=3 4. a=1 b=2 5. a=1 b=1 $\Rightarrow$ 5 Subtraktionen	1. a=7 b=3 2. a=1 b=3 3. a=1 b=0 $\Rightarrow$ 3 Restbildungen

Bei einer Abschätzung nach oben überlegen wir uns, was im „schlimmsten Fall“ (dem sog. *worst case*) passieren könnte.

Beim Euklidischen Algorithmus mit Subtraktionen sieht man leicht, dass im schlechtesten Fall eine der beiden Zahlen gleich 1 ist, dann wird nämlich die größere immer nur um 1 vermindert. Bei $a = 5$ und $b = 1$ braucht der Algorithmus 4 Subtraktionen, bis $a = b = 1$ gilt. Wir können daher verallgemeinern und sagen, dass dieser Algorithmus höchstens

$$\max(a,b) - 1$$

Schleifendurchläufe benötigt.

Beim Euklidischen Algorithmus mit Restbildung ist es nicht ganz so einfach. Wir dividieren immer die größere durch die kleinere Zahl und arbeiten mit dem Rest der Division weiter, bis 0 erreicht ist. Das „Schlimmste“, was nun passieren kann, ist, dass das Ergebnis einer Restbildung den größtmöglichen Wert annimmt. Wir schauen uns das anhand eines Beispiels an: Eine der Ausgangszahlen sei 499. Ist die andere Zahl 250, dann erhalten wir einen Divisionsrest von 249. Ist die andere Zahl jedoch z.B. 100, dann erhalten wir einen Rest von 99. Man sieht leicht, dass kein größerer Rest als 249 bleiben kann. Dies entspricht der Division der Ausgangszahl (499) durch 2. Also: MOD(499, 250)= 249, DIV(499, 2)= 249.

Im schlimmsten Fall dividiert unser Algorithmus also in jedem Schritt durch 2. Wenn wir eine Zahl n systematisch durch 2 dividieren, benötigen wir $\lfloor \log_2 n \rfloor + 1$ Divisionen, bis wir bei 0 angelangt sind (siehe Aufgabe 3.e)). Die Anzahl der Schleifendurchläufe des Euklidischen Algorithmus mit Restbildung können wir also nach oben abschätzen mit

$$\lfloor \log_2 \max(a,b) \rfloor + 1$$

e) EuklidSub: Die Differenz von a und b sollte sehr groß sein.

EuklidDiv: Die Zahlen sollten so gewählt sein, dass die Division der größeren durch die kleinere Zahl 1 ergibt und ein möglichst großer Rest bleibt. Dies sollte auch wieder für das nächste Zahlenpaar gelten usw. Am besten baut man seine Zahlen daher „von unten“ auf, indem man immer die kleinere zur größeren addiert.

3. Aufgabe:

a)

$$\log_b a = c \Leftrightarrow b^c = a$$

b)

$$\underbrace{\log_b a}_{x} = \underbrace{\log_b c}_{y} \cdot \underbrace{\log_c a}_{z}$$

Nach Definition (s. o.) ist

$$\begin{aligned} \log_b a = x &\Leftrightarrow b^x = a \\ \log_b c = y &\Leftrightarrow b^y = c \\ \log_c a = z &\Leftrightarrow c^z = a \end{aligned}$$

Wir setzen nun die Gleichungen ineinander ein:

$$\begin{aligned} b^x &= a \\ &= c^z \\ &= b^{y \cdot z} \end{aligned}$$

Damit dies gilt, muss

$$x = y \cdot z$$

sein.

c)

$$\underbrace{\log_b \frac{a}{c}}_{x} = \underbrace{\log_b a}_{y} - \underbrace{\log_b c}_{z}$$

Nach Definition (s. o.) ist

$$\begin{aligned} \log_b \tfrac{a}{c} = x &\Leftrightarrow b^x = \tfrac{a}{c} \\ \log_b a = y &\Leftrightarrow b^y = a \\ \log_b c = z &\Leftrightarrow b^z = c \end{aligned}$$

Wir setzen nun die Gleichungen ineinander ein:

$$\begin{aligned} b^x &= \frac{a}{c} \\ &= \frac{b^y}{b^z} \\ &= b^{y-z} \end{aligned}$$

Damit dies gilt, muss

$$x = y - z$$

sein.

d) Nach Definition ist

$$b^{log_b a} = a$$
$$a^{log_a b} = b$$

Also ist

$$\begin{aligned} \left(b^{log_b a}\right)^{\log_a b} &= b \\ b^{log_b a \cdot \log_a b} &= b \end{aligned}$$

Damit dies wahr ist, muss gelten

$$\begin{aligned} log_b a \cdot \log_a b &= 1 \\ log_b a &= \frac{1}{\log_a b} \end{aligned}$$

$\log_a b$ ist demnach das Reziproke von $\log_b a$.

e) $\lfloor \log_2 n \rfloor + 1$ entspricht der Anzahl der Divisionen $n = n$ div 2 bis $n < 1$.
Beispiel:

DIV(1000,2)	=	500
DIV(500,2)	=	250
DIV(250,2)	=	125
DIV(125,2)	=	62
DIV(62,2)	=	31
DIV(31,2)	=	15
DIV(15,2)	=	7
DIV(7,2)	=	3
DIV(3,2)	=	1
DIV(1,2)	=	0

Die Anzahl der Divisionen ist 10 und

$$\begin{aligned} \lfloor \log_2 1000 \rfloor + 1 &= \lfloor 9,96578 \rfloor + 1 \\ &= 9 + 1 \\ &= 10 \end{aligned}$$

4. Aufgabe:

```
import Prog1Tools.IOTools;

public class BinSucheNeu {

  public static void main(String[] args){

    long n, min , el, d;
    n =  IOTools.readLong("n␣eingeben:␣");
    min = 0;
    el = n+2;

    //min=a;
    //el=b–a+1;

    while (min+1 < (min + el – 1))  {

      d = (el – 1) / 2;
      System.out.println("d␣=␣" +d);
      if ((min + d)*(min+d) <= n ) {
        min = min + d;
        el = el – d;
        System.out.println("min␣=␣" +min);
        System.out.println("el␣=␣" +el);
      }

      if (((min + el – 1) – d )*((min + el – 1) – d) > n){
        el = el – d;
        System.out.println("el=␣"+ el);
      }
    }
   System.out.println(min + "␣ist␣ganzzahlige␣Wurzel␣von␣" + n);
  }
}
```

Invariante:

$$min_l \leq \lfloor \sqrt{n} \rfloor < min_l + el_l - 1$$

bzw.

$$\mathrm{gW}(n) \in [min_l, min_l + el_l - 1)$$