Tafelübung 03
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Ablaufdiagramme

Schleifen

- `while`-Schleifen
- `for`-Schleifen
- `do-while`-Schleifen
- Endlosschleifen

Hinweise zu den Aufgaben

- Primfaktorzerlegung
- Gaußsches Eliminationsverfahren

Ablaufdiagramme

Ablaufdiagramme

- Ablaufdiagramme dienen zur grafischen Darstellung eines Algorithmus
  - Knoten stellen auszuführende Anweisungen und Bedingungen dar
  - Kanten stellen den möglichen Kontrollfluss dar

- Elemente eines Ablaufdiagramms:
  - zur Darstellung von Beginn und Ende einer Methode
  - `Anweisung` zur Darstellung einer einzelnen Anweisung
  - `Bed.` zur Darstellung einer Bedingung einer Verzweigung
  - $\longrightarrow$ zur Darstellung des Kontrollflusses

- für Zuweisungen in Anweisungen wird üblicherweise „$\leftarrow$“ verwendet

Beispiel: Ablaufdiagramm einer `if`-Anweisung

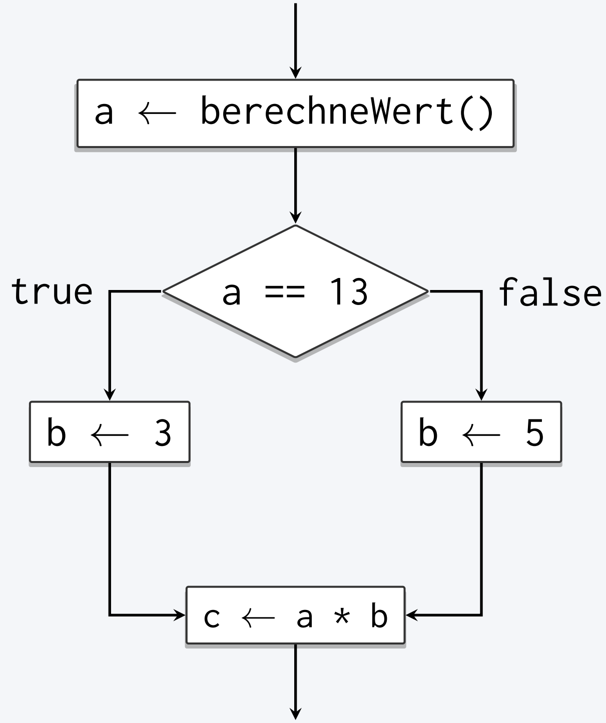
Beispiel: `if`-Anweisung

```
a = berechneWert();

if (a == 13) {
  b = 3;
} else {
  b = 5;
}

c = a * b;
```

Ablaufdiagramm

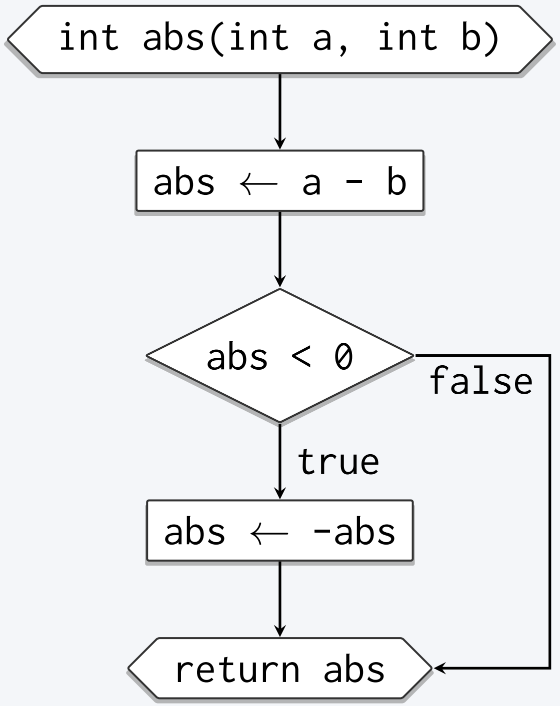


Beispiel: Ablaufdiagramm einer Methode

Beispiel: Methode

```java
public static int abs(int a,
                      int b) {
  int abs = a - b;
  if (abs < 0) {
    abs = -abs;
  }
  return abs;
}
```

Ablaufdiagramm



Schleifen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Schleifen: Motivation (I)

- Was tun, wenn Anweisungen mehrfach ausgeführt werden sollen?
  - Beispiel: 10x „ Hallo Welt“ ausgeben
  - Idee: Anweisungen einfach mehrfach hintereinder schreiben
    - aufwändig und äußerst unschön ☹
    - klappt nicht, wenn man die Anzahl an Ausführungen vorher nicht kennt ☹
  - Lösung: Schleifen als Kontrollstruktur
    - wiederholte Ausführung von Anweisungen solange Bedingung erfüllt ist

- Java kennt verschiedene Arten von Schleifen:
  - `while`-Schleife
  - `for`-Schleife
  - `do-while`-Schleife

while-Schleife

Syntax: while-Schleife

```
while (Bedingung) {  // Schleifenkopf
  Anweisungen;       // Schleifenrumpf
}
```

Hinweise

- der Schleifenrumpf besteht aus den zu wiederholenden Anweisungen
- die Bedingung ist ein beliebiger boolescher Ausdruck
  - wird vor jedem Schleifendurchlauf ausgewertet
    - falls `true`: Anweisungen im Schleifenrumpf werden ausgeführt
    - falls `false`: Schleife wird beendet

Ablaufdiagramm der `while`-Schleife

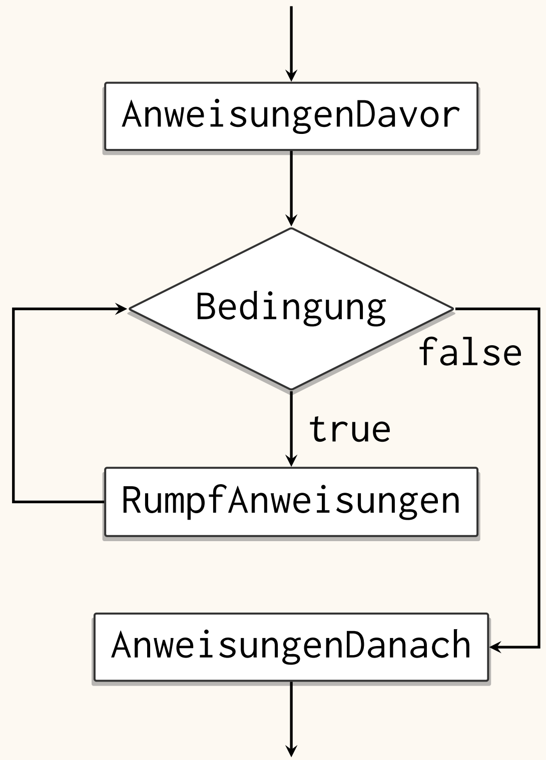
Syntax: `while`-Schleife

```
AnweisungenDavor;

while (Bedingung) {
  RumpfAnweisungen;
}

AnweisungenDanach;
```

Ablaufdiagramm



Beispiel: Ausgabe aller Quadratzahlen $\leq$ y (I)

Aufgabe

Gegeben sei eine positive, ganze Zahl y. Gesucht ist ein Algorithmus, der alle ganzzahligen Quadratzahlen $\leq y$ berechnet und ausgibt.

Beispiel

$y = 42 \quad \Rightarrow \quad 1, 4, 9, 16, 25, 36$

Überlegungen

- Berechnung der Quadratzahlen:
  - eine Quadratzahl ist eine Zahl $q = n \cdot n$
  - um die nächstgrößere Quadratzahl zu berechnen, wird n um eins inkrementiert

$\leadsto$ Speicherung der Basis n, Berechnung der Quadratzahl q, Inkrementierung von n

- Bedingung für die zu schreibende Schleife:
  - falls die *nächste* Quadratzahl *größer* als y ist: Berechnung abbrechen

$\leadsto$ Rumpf ausführen, solange die nächste Quadratzahl $\leq y$ ist

Beispiel: Ausgabe aller Quadratzahlen $\leq$ y (II)

Aufgabe

Gegeben sei eine positive, ganze Zahl y. Gesucht ist ein Algorithmus, der alle ganzzahligen Quadratzahlen $\leq y$ berechnet und ausgibt.

Mögliche Lösung

```
int n = 1;  // Basis

while (n*n <= y) {
  int q = n*n;  // Berechnung der aktuellen Quadratzahl
  System.out.println(q);  // Ausgabe der aktuellen Quadratzahl
  n = n + 1;  // Inkrementierung der Basis n
}
```

Tipp

Eigene Programme immer auch mit Grenzwerten testen, hier z.B. für $y \in \{48, 49, 50\}$.

for-Schleife

Syntax: for-Schleife

```
for (Initialisierung; Bedingung; Fortsetzung) { // Schleifenkopf
  Anweisungen;                                 // Schleifenrumpf
}
```

Hinweise

- Schleifenrumpf und Bedingung wie bei der `while`-Schleife
- die `Initialisierung` wird einmalig vor Betreten der Schleife ausgeführt
  - kann leer bleiben (Semikolon nicht vergessen!)
- die `Fortsetzung` wird nach jedem Schleifendurchlauf ausgeführt
  - kann leer bleiben (Semikolon nicht vergessen!)

Ablaufdiagramm der `for`-Schleife

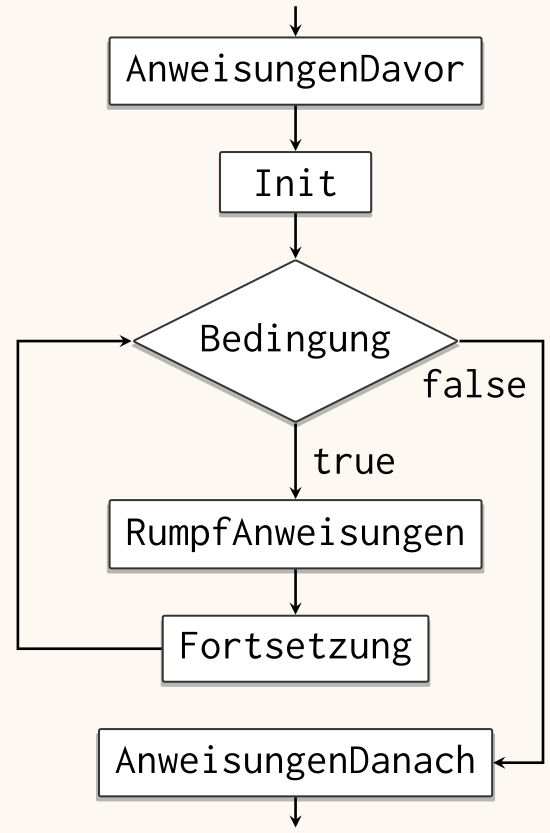
Syntax: `for`-Schleife

```
AnweisungenDavor;

for (Init; Bedingung; Fortsetzung) {
  RumpfAnweisungen;
}

AnweisungenDanach;
```

Ablaufdiagramm



Schleifenvariable, Laufvariable, Zählvariable

- die Bedingung hängt üblicherweise vom Wert einer ganzzahligen Variable ab

⇝ Schleifenvariable oder Laufvariable oder Zählvariable

- eine solche Laufvariable wird üblicherweise...
  - ... in der Initialisierung deklariert und initialisiert
  - ... in der Fortsetzung inkrementiert oder dekrementiert

Beispiel

```
for (int i = 1; i <= 3; ++i) {
  System.out.println(i);
}
```

Alternative (unschön!)

```
int i = 1;
// evtl. weitere Anweisungen ...
for ( ; i <= 3; ) {
  System.out.println(i);
  ++i;
}
```

Beispiel: Berechnung der n-ten Potenz (I)

Aufgabe

Gegeben seien zwei positive, ganze Zahlen x und n. Gesucht ist ein Algorithmus, der die n-te Potenz x^n von x berechnet und ausgibt.

Beispiel

$x = 3, n = 4 \quad \Rightarrow \quad 81$

Überlegungen

- $x^n \;=\; x \cdot x \cdot x \cdot \ldots \cdot x \;=\; 1 \cdot x \cdot x \cdot x \cdot \ldots \cdot x$

$\leadsto$ in jeder Iteration eine Multiplikation mit x durchführen

$\leadsto$ Variable, um das aktuelle Zwischenergebnis zu speichern

  - Variable mit 1 initialisieren (neutrales Element der Multiplikation)

- wir brauchen n solcher Multiplikationen

$\leadsto$ Laufvariable, die Anzahl an Iterationen mitzählt

$\leadsto$ Bedingung, die die Schleife nach n Iterationen abbricht

Beispiel: Berechnung der *n*-ten Potenz (II)

Aufgabe

Gegeben seien zwei positive, ganze Zahlen x und n. Gesucht ist ein Algorithmus, der die n-te Potenz x^n von x berechnet und ausgibt.

Mögliche Lösung

```java
int potenz = 1;

for (int i = 1; i <= n; ++i) {
  potenz = potenz * x;
}

System.out.println(potenz);
```

„Übliche“ Lösung

```java
int potenz = 1;

for (int i = 0; i < n; ++i) {
  potenz = potenz * x;
}

System.out.println(potenz);
```

Laufvariablen beginnen üblicherweise bei 0

Vergleich `for`-Schleife ↔ `while`-Schleife

Beide Schleifen-Arten sind gleich mächtig und können ineinander umgewandelt werden.

`for`-Schleifen

`for`-Schleifen eignen sich inbesondere dann, wenn die Anzahl an Iterationen vor Betreten der Schleife bekannt bzw. berechenbar ist.

`while`-Schleifen

`while`-Schleifen eignen sich inbesondere dann, wenn die Anzahl an Iterationen vor Betreten der Schleife unbekannt bzw. nicht (einfach) berechenbar ist.

Beispiel: `for`-Schleife

```java
int potenz = 1;

for (int i = 0; i < n; ++i) {
  potenz = potenz * x;
}
System.out.println(potenz);
```

Beispiel: `while`-Schleife

```java
int n = 1;

while (n*n <= y) {
  int q = n*n;
  System.out.println(q);
  n = n + 1;
}
```

`do-while`-Schleife: Motivation

- **bisher** (`for`, `while`):
  - Überprüfung der Bedingung vor jedem Schleifendurchlauf

  $\rightsquigarrow$ auch *vor* dem *ersten* Schleifendurchlauf!

$\rightsquigarrow$ Schleifenrumpf wird unter Umständen gar nicht ausgeführt

$\rightsquigarrow$ sogenannte kopfgesteuerte oder abweisende Schleifen

- **jetzt** (`do-while`):
  - Überprüfung der Bedingung nach jedem Schleifendurchlauf

  $\rightsquigarrow$ erstmalig *nach* dem *ersten* Schleifendurchlauf!

$\rightsquigarrow$ Schleifenrumpf wird mindestens einmal ausgeführt

$\rightsquigarrow$ sogenannte fußgesteuerte oder nicht-abweisende Schleife

do-while-Schleife

Syntax: do-while-Schleife

```
do {
  Anweisungen;          // Schleifenrumpf
} while (Bedingung); // "Schleifenfuß"
```

Hinweise

- Schleifenrumpf und Bedingung wie bei der while-Schleife
  - allerdings andere Auswertungsreihenfolge (siehe oben)

Achtung!

Semikolon hinter der Bedingung nicht vergessen!

Ablaufdiagramm der do-while-Schleife

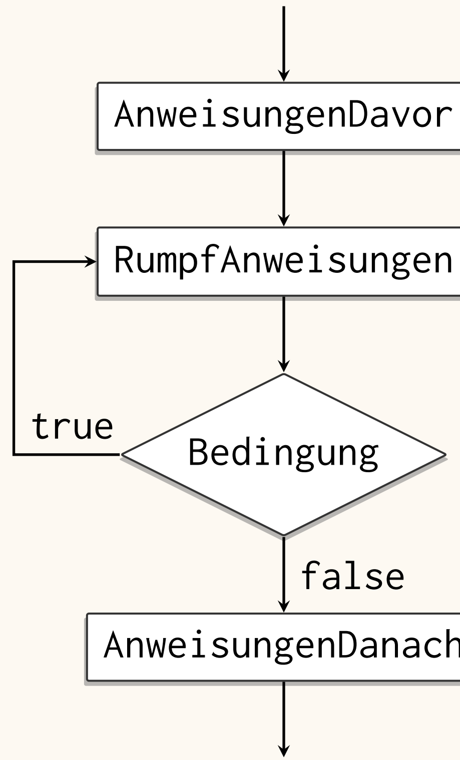
Syntax: do-while-Schleife

```
AnweisungenDavor;

do {
  RumpfAnweisungen;
} while (Bedingung);

AnweisungenDanach;
```

Ablaufdiagramm



Beispiel: Sechser würfeln (I)

Aufgabe

Gesucht ist ein Programm, das solange „würfelt“, bis eine 6 fällt.

Hinweis

Zufallszahl zwischen 1 und 6 erzeugen:

```
1 + (int)(Math.random()*6)
```

Überlegungen

- in jeder Iteration eine Zahl „würfeln“
  - falls 6: Schleife abbrechen
  - falls keine 6: Schleife erneut durchlaufen
- es muss mindestens einmal „gewürfelt“ werden

$\rightsquigarrow$ fußgesteuerte Schleife

Beispiel: Sechser würfeln (II)

Aufgabe

Gesucht ist ein Programm, das solange „würfelt“, bis eine 6 fällt.

Mögliche Lösung

```
int wuerfel;

do {
  wuerfel = 1 + (int)(Math.random()*6);
  System.out.println("es wurde eine " + wuerfel + " gewürfelt");
} while (wuerfel != 6);
```

Endlosschleifen

- für alle Schleifen-Arten gilt:
  - Ausführung des Schleifenrumpfes ist abhängig von einer Bedingung
    - Wiederholung des Schleifenrumpfes *solange* die Bedingung erfüllt ist
    - Abbruch der Schleife sobald die Bedingung *nicht* mehr erfüllt ist

$\leadsto$ wenn die Bedingung immer erfüllt ist, also nie `false` wird:

- Schleifenrumpf wird „unendlich oft“ ausgeführt $\leadsto$ Endlosschleife

Unerwünschtes Verhalten

Bei einer Endlosschleife terminiert das Programm nicht. In den allermeisten Fällen ist dieses Verhalten unerwünscht!

Beispiele für Endlosschleifen

Offensichtliche Endlosschleife

```java
while (true) {
  System.out.println("Hallo, Welt!");
}
```

Weniger offensichtliche Endlosschleife

```java
int i = 1;
while (i <= 30) {
  if ((i % 5 == 0) || (i % 7 == 0)) {
    continue; // bricht *aktuellen* Durchlauf ab
  }
  System.out.print(i + " ");
  i = i + 1;
}
```

Hinweise zu den Aufgaben

Primfaktorzerlegung

Primfaktorzerlegung

Die Primfaktorzerlegung einer natürlichen Zahl n ist die Darstellung von n als Produkt aus Primzahlen. Diese ist (bis auf kommutative Vertauschungen) eindeutig.

Beispiele

- $9 = 3 \cdot 3 = 3^2$
- $24 = 2 \cdot 2 \cdot 2 \cdot 3 = 2^3 \cdot 3^1$
- $28 = 2 \cdot 2 \cdot 7 = 2^2 \cdot 7^1$
- $13 = 13^1$

Primfaktorzerlegung: Idee für einen Algorithmus

- um eine Zahl n in seine Primfaktoren zu zerlegen:
  - beginne mit dem Teiler 2
  - falls n ganzzahlig ohne Rest durch den Teiler dividiert werden kann:
    - dividiere n durch den Teiler und „merke“ Division
    - wiederhole die Berechnung mit dem alten Teiler
  - andernfalls:
    - erhöhe den Teiler und wiederhole die Berechnung mit dem neuen Teiler
  - breche die Berechnung ab, sobald $n = 1$ ist

Hinweis

Dass hier auch Nicht-Primzahlen als Teiler-Kandidaten betrachtet werden, beeinflusst das Ergebnis nicht, denn es werden nur Primzahlen als tatsächliche Teiler berücksichtigt.

Aufgabe 3.4: Darstellung der Primfaktoren

Aus der Aufgabenstellung

Die Methode `primfaktorzerlegung` (sic!) ermittelt die Primfaktorzerlegung ihres Arguments n und gibt sie so als 2-dimensionales Feld z zurück, dass:

$$n = p_0^{e_0} \cdot p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_{k-1}^{e_{k-1}}$$

wobei `z[0]` = $\{p_0, p_1, p_2, \ldots, p_{k-1}\}$ die sortierten ($p_0 < p_1 < p_2 < \ldots < p_{k-1}$) Primfaktoren sowie `z[1]` = $\{e_0, e_1, e_2, \ldots, e_{k-1}\}$ ihre jeweiligen Potenzen ($e_i > 0$) sind.

Beispiel: Primfaktorzerlegung von $n = 6600 = 2^3 \cdot 3^1 \cdot 5^2 \cdot 11^1$

	0	1	2	3
0	2	3	5	11
1	3	1	2	1

Größter gemeinsamer Teiler (ggT)

Größter gemeinsamer Teiler (ggT)

Der größte gemeinsame Teiler (ggT) zweier natürlicher Zahlen a und b ist die größte natürliche Zahl $ggT(a, b)$, durch die sich a und b ganzzahlig ohne Rest teilen lassen.

Beispiele

- $ggT(36, 24) = ggT(24, 36) = 12$
- $ggT(12, 24) = ggT(24, 12) = 12$
- $ggT(13, 3) = ggT(3, 13) = 1$

Berechnung aus den Primfaktorzerlegungen zweier Zahlen

Wenn die Primfaktorzerlegungen von a und b bekannt sind, multipliziert man für den $ggT(a, b)$ die Primfaktoren, die in *beiden* Zerlegungen auftreten:

$$ggT(36, 24) = ggT(2^2 \cdot 3^2, 2^3 \cdot 3^1) = 2^2 \cdot 3^1 = 12$$

Kleinstes gemeinsames Vielfaches (kgV)

Kleinstes gemeinsames Vielfaches (kgV)

Das kleinste gemeinsame Vielfache (kgV) zweier natürlicher Zahlen a und b ist die kleinste natürliche Zahl $kgV(a, b)$, die ein ganzzahliges Vielfaches von a und b ist.

Beispiele

- $kgV(36, 24) = kgV(24, 36) = 72$
- $kgV(12, 24) = kgV(24, 12) = 24$
- $kgV(13, \ \ 3) = kgV(\ \ 3, 13) = 39$

Berechnung aus den Primfaktorzerlegungen zweier Zahlen

Wenn die Primfaktorzerlegungen von a und b bekannt sind, multipliziert man für das $kgV(a, b)$ die Primfaktoren, die in *mindestens einer* Zerlegung auftreten:

$$kgV(36, 24) = kgV(2^2 \cdot 3^2, 2^3 \cdot 3^1) = 2^3 \cdot 3^2 = 72$$

Gaußsches Eliminationsverfahren

Das Gaußsche Eliminationsverfahren...

... dient zur Lösung von linearen Gleichungssystemen der Form:

$$\left\{\begin{array}{ccccccc} a_{00} \cdot x_0 & + & a_{01} \cdot x_1 & + & a_{02} \cdot x_2 & = & b_0 \\ a_{10} \cdot x_0 & + & a_{11} \cdot x_1 & + & a_{12} \cdot x_2 & = & b_1 \\ a_{20} \cdot x_0 & + & a_{21} \cdot x_1 & + & a_{22} \cdot x_2 & = & b_2 \end{array}\right\}$$

bzw. in Matrix-/Vektor-Schreibweise: $A\vec{x} = \vec{b}$

Gaußsches Eliminationsverfahren: Beispiel

$$(A|b) := \left(\begin{array}{ccc|c} 3 & 11 & 19 & 101 \\ 5 & 13 & 23 & 103 \\ 7 & 17 & 29 & 107 \end{array}\right) \xmapsto{piv_0} \left(\begin{array}{ccc|c} 7 & 17 & 29 & 107 \\ 5 & 13 & 23 & 103 \\ 3 & 11 & 19 & 101 \end{array}\right) \xmapsto{elim_0} \left(\begin{array}{ccc|c} 7 & 17 & 29 & 107 \\ 0 & -6 & -16 & -186 \\ 0 & -26 & -46 & -386 \end{array}\right) \xmapsto{piv_1}$$

$$\xmapsto{piv_1} \left(\begin{array}{ccc|c} 7 & 17 & 29 & 107 \\ 0 & -26 & -46 & -386 \\ 0 & -6 & -16 & -186 \end{array}\right) \xmapsto{elim_1} \left(\begin{array}{ccc|c} 7 & 17 & 29 & 107 \\ 0 & -26 & -46 & -386 \\ 0 & 0 & -140 & -2520 \end{array}\right) \xmapsto{loese} \vec{x} := \begin{pmatrix} -18.0 \\ -17.0 \\ 18.0 \end{pmatrix}$$

Fragen? Fragen!

(hilft auch den anderen)