1. Grundbegriffe und Beispiele

Literatur: Reese Kapitel 1, 2, 3, 4.

Ein einfaches Programm in Java:

```
1. public class Berechnung
2. {
3.   public static void main(String[] args)
4.   {
5.     int i;
6.     i = 3 + 4;
7.     System.out.println(i);
8.   }
9. }
```

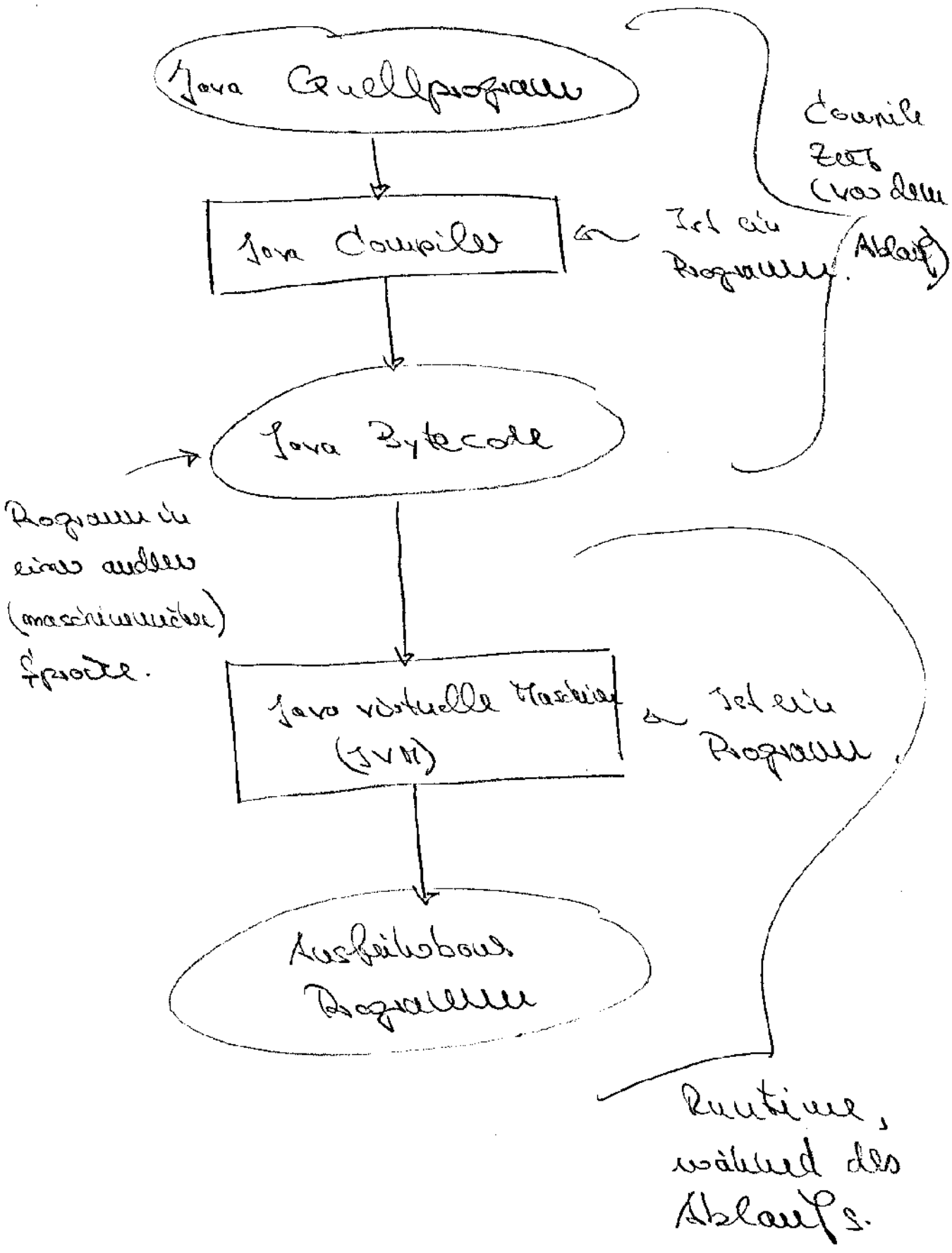
Berechnung.java

auf Programme/Woche1

Quelltext, Quellprogramm

= das ganze Programm.

Was geschieht dann?



Java Compiler = Java Übersetzer

Java virtuelle Maschine? Muß sie übersetzen? In gewisser Weise ja, arbeitet aber direkt ab. Ein

Interpreter.

Compiler übersetzt, Interpreter übersetzt und führt aus.

Was machen wir jetzt damit? Wie in der Schule? Dazu einmal das folgende Bild:

Diese Seite mußte aus rechtlichen Gründen entfernt werden!

Programm kommt auf die Festplatte.
Wie ist die Festplatte organisiert?
Sie enthält Dateien, die in
Verzeichnissen (directories) strukturiert
sind.

Die Dateien auf der Festplatte sind damit strukturiert auch vorzustellen:

Wo sind die .Dateien .login, .cshrc ?

Was ist das drinnen.

Wie jetzt das Programm

Eingeben,
Übersetzen,
Aufrufen.

Wir haben das Java Development Kit (JDK) oder Java Software Development Kit (JDK) installiert.

Programm in Datei

Berechnung.java

dann das Kommando

javac Berechnung.java

Das ist der Dateiname.

javac = Kommando für den Compiler,

also aufrufen des Namen eines Programms.

10 wichtig:

public class Berechnung

$\Leftrightarrow$

und Dateiname Berechnung.java.

Nach fehlerfreiem Lauf der Datei

Berechnung.class

Zusatz an Dateiname.

Diese enthält den sogenannten Bytecode.

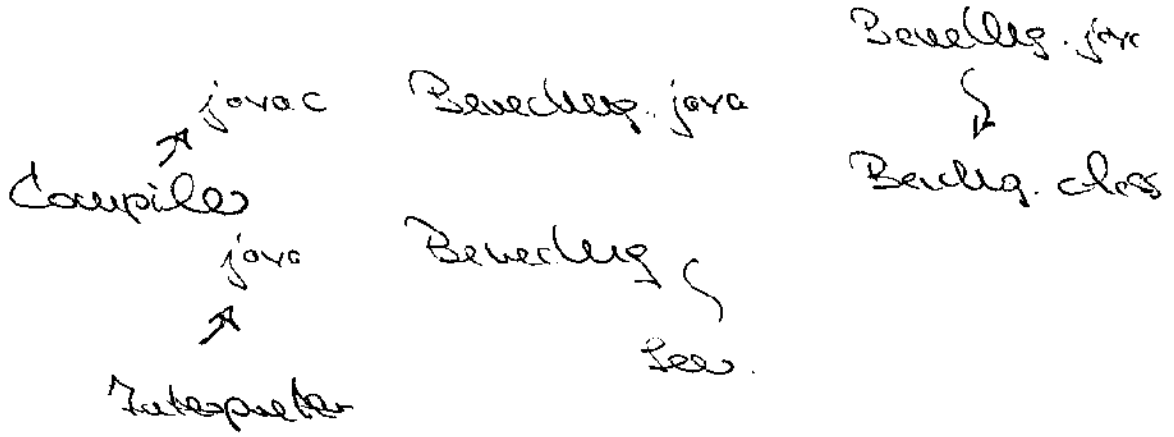
Aufruf des Interpreters mit Bytecode als Eingabe

java Berechnung

java = Name eines Programms, des Interpreters.

Die 7 sollte weiter.

Noch einmal:



Wo geschieht prinzipiell bei der Ausführung eines Programms?

Diese ist in der Zentraleinheit.

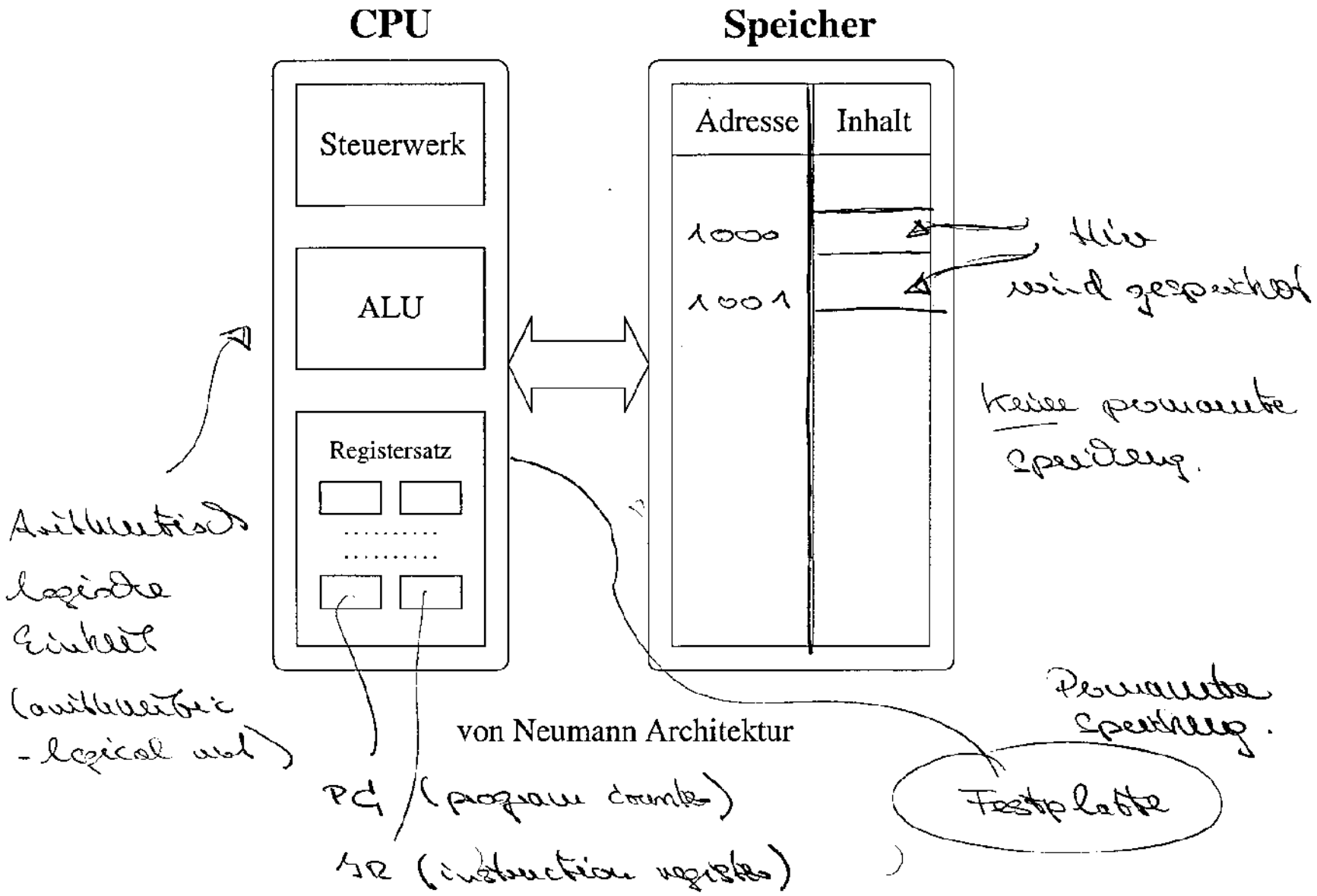
Zentraleinheit = Prozessor + Arbeitsspeicher.

CPU (central processing unit) und Hauptspeicher

Arbeitsspeicher temporär im Unterschied zu Festplatte.

Rückgesamt. Entwicklerheft.

Prozessor



Typische Befehle, die hier ausführbar sind:

Load i // Inhalt von Speicherplatz i in Register 0.

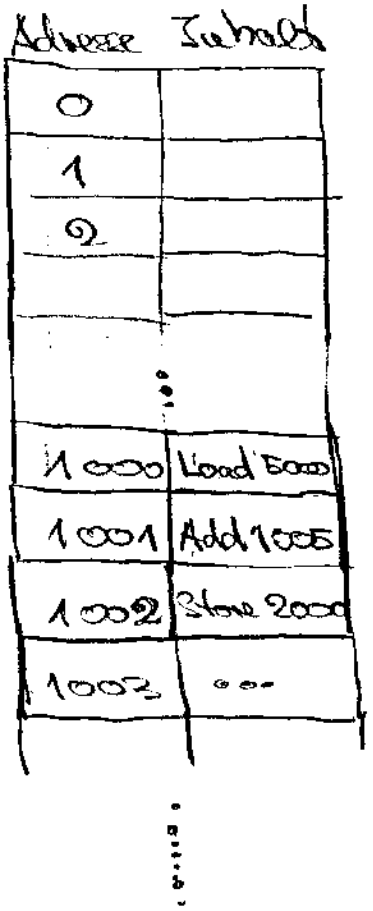
Store i // Reg 0 in Speicherplatz i

Add i // Addiert Reg 0 zu Speicherplatz i, Ergebnis in Register 0.

Programm dazu

```
Load   5000    // Addiert Speicherplatz
Add    1005    // 5000 zu 1005 und
Store  2000    // speichert in 2000.
```

Programme, die ausgeführt werden stehen auch im Hauptspeicher:



In dem Prozessor ein spezielles Register, das Programmierbare PC (programm counter), das die Adresse des aktuellen Befehls enthält.

Der Prozessor führt dann sogenannte Instruktionszyklen aus:

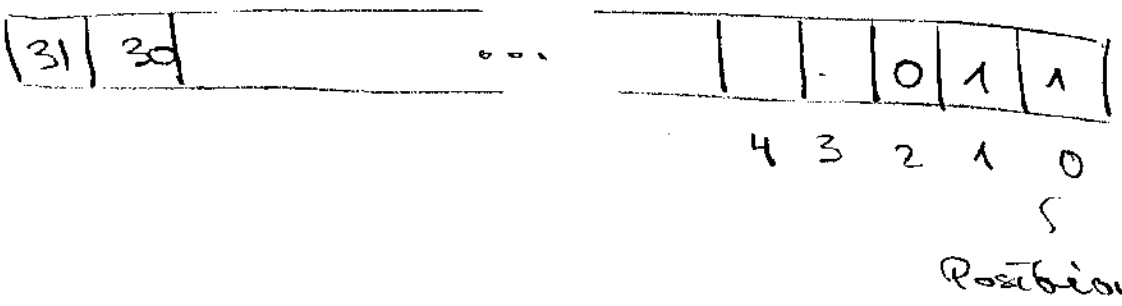
1. Fetch (Holt Befehl auf Adresse PC in IR)
2. Decode (Dekodiert die Operation)
3. Fetch Operands (Holt die Operanden aus den Speicher (= die Register))
4. Execute (Die ALU führt die Operation aus)
5. Next Instruction (PC bekommt Adresse des nächsten Befehls. In der Regel Erhöhung um 1)

Danach geht es wieder bei 1. los.

Der Arbeitsspeicher ist der RAM (random access memory) organisiert. Das heißt wahlfreier Zugriff. Das heißt: Speicherplatz zu jeder beliebigen Adresse kann gleich schnell erreicht (d.h. geladen werden).

Dagegen: Sequentieller Zugriff auf Bänder. Von der Platte werden größere Einheiten (Blocks) in den Hauptspeicher geladen. Ein Block schneller als außerhalb.

Arbeitsspeicher aus Speicherworten, die nummeriert sind. (alles eine Adresse haben). Ein Speicherwort so



Speicherwort aus 32 Bits von 0, ..., 31.

Ein Bit kann 0 oder 1 sein

(Byte = 8 Bits (hier von 0 ... 7))

Hier ist die Wortlänge 32.

Neuerdings auch Wortlänge 64.

1.16

Was passiert jetzt bei der Ausführung von unserem Programm Berechnung?

Zeile 1., ..., 5. : Vorbereitungen.

Variable i wird an einen Speicherplatz gebunden. Tabelle, die sagt i ist Speicherplatz 1004, zum Beispiel.

Variablen stehen für Speicherplätze.

Zeile 6.

Erst 3+4*7 ausrechnen, dann in Speicherplatz zu i speichern.

Platz zu i enthält dann 7 über 0,1 dargestellt.

Zeile 7.

Programmstücke zu Ausgabe.

Es werden dann eine folgende Maschinenbefehle ausgeführt:

Load x3 (Die 3 selbst (vorg. x3) in Register 0.)

Add x4 (Die 4 selbst zu 3 addieren: Ergebnis in Reg. 0.)

Store i (Reg. 0 in Speicherplatz von i. (MB evtl. modifiziert werden))

⋮

So weit zur Ausführung des Programms. Als nächstes zum Programm selbst!

Syntax = Struktur des Programms als Text.

Semantik = Das, was das Programm bewirkt.

Ein Programm besteht aus

Anweisungen.

Am Ende einer Anweisung
grundsätzlich ein ; (Semikolon).

Beispiel: int i;

Deklarationsanweisung, Variablendeklaration.

Wirkung (Semantik): Speicherplatz
wird gebunden. Inhalt eine
ganze Zahl, integer, int.

Also Zahl aus

$\mathbb{Z} = \{ \ldots -3, -2, -1, 0, 1, 2, 3, \ldots \}$.

int beschreibt den Datentyp
integer.

i ist ein benutzerdefinierter Bezeichner.

Zeile 6. i = 3+4

Eine Zuweisung. 3+4 ist ein arithmetischer Ausdruck.

Zeile 7: System.out.println(i);

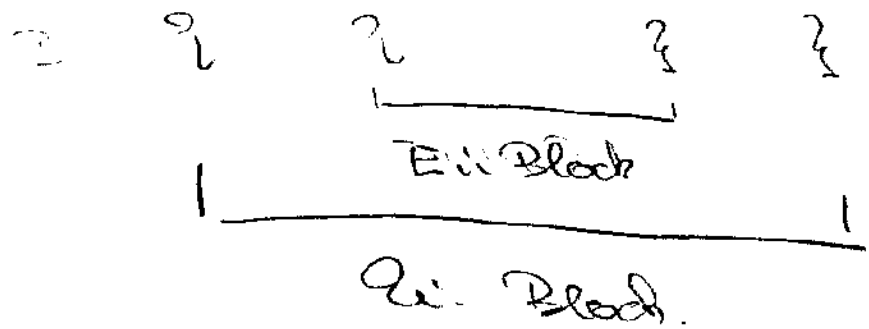
System.out.println ist eine Methode (fertiges Programmstück oder Unterprogramm) für die Ausgabe. Anweisung ist ein Methodenaufruf. i ist das Argument, der aktuelle Parameter. i meint den Inhalt des Speicherplatzes von i (als Text) meint "i" als Buchstaben.

Das noch bleibt: Der Rahmen

```
public class Bezeichnung
{
    public static void main(String[] args)
    {
    }
}
```

Klasse — Hauptmethode

Quelltext aus Blöcken, die zwischen den { } Klammern steht. Hier ein verschachtelter Block:



1. Block: Die Klasse

Java objektorientiert, arbeitet mit Klassen. Eine Klasse hat folgenden Inhalt

```
public class Berechnung
{
    Hier steht der Rest des
    Quelltexts der Klasse
    Berechnung!
}
```

Name der Klasse: Berechnung.

Die Datei mit dem Quelltext muß Berechnung.java heißen.

Von den Namen der Klasse müssen immer die Worte public und class stehen.

Klassennamen immer
mit Großbuchstaben beginnend.

2. Block: Die Hauptmethode

Klassen beinhalten Methoden.

Klassen, die selbst ablauffähig
sind, besitzen die Methode main.
Die zugehörige Programmzeile ist
dabei

public static void main(String[] args).

Diese ist ebenfalls so. String
ist mit "S" geschrieben.

Schlüsselwörter sind vordefinierte
festgelegte Wörter, die man
somit nicht verwenden sollte.

1.23.

public, class,

public, static, void

int

Benutzerdefinierte Bezeichner sind dagegen

Zeichen, main

String, args, [],

System.out.println

Bezeichner für eine Speicherplatz

Zeichen (Literalkonstanten) sind: 3, 4.

Empfehlung: 1 Anweisung pro Zeile.

Einige Modifikationen des Programms

```
7    System.out.println("Das Ergebnis ist:");
8    System.out.println(i);
```

"abc" steht für abc als Text.
System.out.println bedeutet:
Aktuellen Parameter ausgeben und
in neue Zeile gehen (line feed)
Manchmal ist auch interessant

```
7    System.out.print("Das Ergebnis ist:");
8    System.out.println(i);
```

Sollte ergeben:

Das Ergebnis ist: 7

Interessant ist auch:

7. System.out.print("Der Wert von i ist");
8. System.out.println(i);

Aus

7. System.out.print("i=");
8. System.out.print(i);

ist die Operation +
zum Aneinanderhängen von Texten
und die Bedeutung (Wirkung, Semantik)

"abc" + "chp" = "abcchp"

"abc", "chp" sind Konstanten (Literal-
konstanten) für Text (Zeichenketten)
"abc" + "chp" ist wieder Ausdruck.

Damit können wir auch schreiben

7 + System.out.println("Das Ergebnis ist:" + i)

Hier ist + die Operation, die Texte zusammenhängt, da Inhalt von i wird also als Text gesehen. Semantisch wird zunächst die Welt von

"Das Ergebnis ist:" + i ← i ist ein Ausdruck.

ausgewertet. Dieses ist der Text

"Das Ergebnis ist: 7"

Dann wird mit diesem aktuellen Parameter die Methode System.out.println ausgeführt.

Die Methoden System.out.print, System.out.println haben immer nur einen aktuellen Parameter (Argument). Dieser wird als Text verarbeitet. Also passiert bei

```
System.out.println(3+i);
```

Hier steht + für die Addition.

Noch einige Variationen des Programms

```
5    i = 100 - 10;

5    i = 10 * 10;
```

Noch ein Programm zur Übung:

Ein Programm, das einfach einen Text ausgibt

```
1  public class Uebung
1.a {                                   Uebung.java
2.    public static void main(String[], args)
2.a   {
3.      System.out.println("Algorithmen
                                  und Programmierung");
4.      System.out.println("in Chemnitz");
5     }
6   }
```

Semikolon aufpassen, ein weiteres Semikolon. Was geschieht dann?

Ausgeben können wir. Wie geben wir ein? Damit das geht, brauchen wir einen ganzen weiteren Satz von Klassen mit Methoden.

```
import Prog1Tools.IOTools;
public class EingBsp
{
  public static void main(String[] args)
  {
    int i,j;
    j = IOTools.readInteger();
    i = IOTools.readInteger();
    System.out.println("i= " + i);
    System.out.println("j= " + j)
  }
}
```

EingBsp.java

Eingabe mit „prompt":

```
System.out.print("j = ");
j = IOTools.readInteger;
```

Alternativ

```
j = IOTools.readInteger("j = ")
```

Das grundlegendste Prinzip der Programmierung ist das der Wiederholung, der wiederholten Ausführung von Anweisungen. Wiederholungen treten in verschiedener Form auf. Hier behandeln wir nur die while - Schleife.

```
1.  public class Wiederum
2.  {
3       public static void main(String[] args)
4       {
5       int i;
6       i = 10;                          <- Hier ist Stolperstelle!
7       while (i > 0)
8           {
9           i = i - 1;
10          System.out.println(i);
```

11 }
12 }
13 }

Das ist eine while-Schleife.
Diese besteht syntaktisch
aus einem Block, dem
Schleifenrumpf und dem Schleifenkopf,
der <u>while</u> $(i > 0)$. $i > 0$ ist
ein Boolscher Ausdruck, und
wahr oder falsch.

Semantik: Ausführung des
Rumpfes solange bis die
Bedingung falsch ist. Endet nicht!
Reihung: 1. Bedingung testen. 2. Rumpf
(falls Bedingung wahr) 3. testen. 4. Rumpf,
... Bis Bedingung falsch.

```
:
if (i > 0)
   {
     i: = i - 1 ;
     System. out. println(i);
   };
while (i>0)
   {
     i = i-1
     System. out. println(i)
   }
   :
```

Dasselbe mit 10-mal

```
i:=i-1;
System. out. println(i);
i:=i-1;
System. out. println(i);
   :
```

Mit Eingabe

```
import Prog1Tools.IOTools;
public class WiedomvsFini
{
   public static void main(String[] args)
   {
      int i;
      i = IOTools.readInteger("i=");
      while (i>0)
      {
         i=i-1;
         System.out.println(i);
      }
   }
}
```

Ausdel Schleifenausführung endlich, also von Eingabe abhängig!

(1.35)

Ein weiteres Beispiel: Berechnen von

$$1 + 2 + 3 + \ldots + k \,,$$

wobei k eingegeben.

```
import Prog1Tools.IOTools;

public class Summe
{
  public static void main(String[] args)
  {
    int i, j;
    j = 0;      // Initialisierung von j
    i = IOTools.readInteger("i=");
    while (i > 0)
    {
      j = i + j;
      i--;      // Für i = i - 1
      System.out.println("i= " + i + " j= " + j);
    }
  }
}
```

Berechnung des Programms bei $i=0$:

i	j
0	0

$i > 0$ ist falsch.
und Schluß.

$i = 1$

i	j
1	0

$i > 0$ wahr

$j = j + i$

1 1

$i --$

0 1

Ausgabe von j (=1)
$i > 0$ ist falsch. Schluß.

$i = 2$

⋮

Ausgabe von $j = 3$
$i > 0$ falsch. Ende.

Programm kennzeichnet Variablenbelegungen.

Alle möglichen Belegungen, d.h.:

Zustandsraum.

Hier: Menge der Paare (i, j) wobei

i, j ganze Zahlen. Deklaration bestimmt

den Zustandsraum.

Was berechnet das Programm?

Am Ende ist bei Eingabe von $k \geq 0$

$$j = 0 + 1 + 2 + 3 + \dots + k.$$

Wie sieht man das genauer? Dazu

sei für $l = 1, 2, 3, \dots$

i_l = Wert von i nach l-tem Durchlauf der Schleife

j_l = Wert von j nach l-tem ...

Außerdem i_0, j_0 Werte vor dem ersten Durchlauf.

Jetzt sei k die eingelesene Zahl. Dann ist $i_0 = 0$, $i_0 = k$.

Deshalb nach dem ersten Durchlauf

$$i_0 = k, \quad i_0 = k-1 .$$

Also auch vor dem zweiten. Dann nach dem zweiten:

$$i_2 = k+(k-1), \quad i_2 = \underbrace{k-2}_{\text{Alles } i} \quad \text{da } -1 \; -1$$

Nach dem dritten

$$i_3 = k+(k-1)+(k-2), \quad i_3 = k-3$$

$$\vdots$$

Nach dem k-ten $\quad k-k$

$$i_{k+1} = \underbrace{k+(k-1)+(k-2)+\ldots+3+2+1}_{k}, \quad i_k = 0 .$$

Wir haben insgesamt genau k Schleifen-durchläufe wegen der $i--$ und der Bedingung $i > 0$.

Für den Sonderfall $k = 0$ gilt alles ebenso. Kein Durchlauf

Betrachten wir folgende Aussage für l mit $1 \leq l \leq k$: Nach dem l-ten Durchlauf der Schleife ist

$$j = \underbrace{k + (k-1) + \dots + (k-l+1)}_{l \text{ Summanden}}, \quad i = k - l$$

j: Wert von j | i: Wert von i

Ist diese Aussage wahr und zwar für alle l?

Nach dem ersten Durchlauf, also $l=1$, ist $i \leq k$, also $j = k$ und $i = k-1$.

Nach dem zweiten auch.

(1.40)

gilt sie für $l \leq k$, dann auch für $l+1 \leq k$. Denn ist vor dem $(l+1)^{\text{ten}}$ Lauf

$$j_l = k + k-1 + \dots + k-l+1 \,, \quad i_l = k-l$$

dann, wie man sieht, danach

$$j_{l+1} = \underbrace{k + k-1 + \dots + k-l+1}_{\text{altes } j} + \underbrace{k-l}_{\text{altes } i}$$

$$i_{l+1} = k-l-1 = k-(l+1).$$

Also gilt die Aussage nach dem $l+1$-ten Durchlauf (und damit vor dem $l+2$-ten).

Zusammenfassend: Aussage gilt
nach dem ersten Durchlauf.

Haben gezeigt: Gilt sie nach
dem ℓ-ten, $1 \leq \ell < k$, dann
nach dem $(\ell+1)$-ten.

Also: Gilt nach dem ersten,
nach dem zweiten (obiges
Schluß mit $\ell=1$), nach dem
dritten ($\ell=2$), ..., nach dem
k-ten.

Haben k Durchläufe.

Also Programm ist korrekt.

Um Schleifen machen, weist man durch einige Beweise, Induktions beweise, noch eine Aufgabe noch eine Aussage über die # Durchläufe.

Für jeden l mit $1 \leq l \leq k$, ist die Aussage:

"Nach dem l-ten Durchlauf

ist

$$i = k + (k-1) + \dots + (k-l+1), \; j = k - l$$

eine Schleifeninvariante!"

Zu jeder Schleife eine Invariante. (Schwierigkeiten

der Programm kommen erst durch Schleifen.)

Noch einmal analog

```
:
j = 1;
while (i > 0)
{
    j := j * i;
    i -- ;
    System.out.println("i=" + i + "j=" + j);
}
```

Bei Eingabe $k \geq 0$ k Durchläufe.

Für $1 \leq l \leq k$ gilt: Nach l-tem Lauf

$j_l = k \cdot (k-1) \cdots (k-l+1) = (k)_l$, $i_l = k - l$.

(1.44)

Am Ende

$j_k = k(k-1)(k-2)\cdot\ldots\cdot 3\cdot 2\cdot 1 = k)_{k} = k!$

und $i_k = 0$. Für $k \geq 0$

sagt man $k!$ ist k Fakultät.

Beachte $0! = 1$.

wird die Schleife nicht

durchlaufen, also $k = 0$, dann

auch $j = k!$ und $i = 0$.

Ein Beispiel geschachtelter Schleifen

```
import Prog1Tools.IOTools

public class GeschSchl

{
  public static void main(String[] args)
  {
    int i, l, k;
    i = IOTools.readInteger("i=");
    while (i > 0)        // Äußere Schleife
    {
      k = 1;
      l = i;
      while (l > 0)      // Innere Schleife
      {
        k = l * k;
        l = l - 1;
      }                  // Haben i! in k.
      System.out.println(i + "! ist " + k);
      i--
    }
  }
}
```

Innere Schleife: auch vorher $k = 1$; $l = i$.

Innere i Durchläufe.

Nach h-tem Durchlauf $1 \leq h \leq 0$

$k_h = i(i-1) \quad (i-h+1)$, $l_h = i - k$

Schleifeninvariante

Nach i Läufen $k_i = i!$, $l = 0$.
gilt auch für $i = 0$.

Äußere Schleife: i innere i Durchläufe: für $i, i-1, \ldots, 1$.

Schleifeninvariante

Nach j-tem Lauf ist ausgehend bei Eingabe s in das i

$s!$ "ist" $s!$ // Nach dem vorherigen Durchlauf
$(s-1)!$ ist $(s-1)!$
$\vdots$
$(s-j+1)!$ ist $(s-j+1)!$

Das geht dann bis i mit 1
Vorbei ist.

Also bei Eingabe von i = 0
wird nicht 0! ausgegeben.
Änderung durch den Test
i >= 0 in der äußeren Schleife.

2. Ganze Zahlen

Natürliche Zahlen

$$\mathbb{N} := \{0, 1, 2, 3, 4, \dots\}$$

≥ 0

Positive natürliche Zahlen

$$\mathbb{N}^+ := \mathbb{N} \setminus \{0\} = \{1, 2, 3, 4, \dots\}$$

Ganze Zahlen

$$\mathbb{Z} := \{\dots -2, -1, 0, 1, 2, 3, \dots\}$$

$$= \{0, 1, -1, 2, -2, 3, -3, \dots\}$$

Primzahlen $\mathbb{P} \subseteq \mathbb{N}$

$$\mathbb{P} = \{2, 3, 5, 7, 11, 13, 17, \dots\}$$

Primzahlen sind $\neq 1$ und
nur durch 1 und sich selbst teilbar
(ohne Rest).

Für $a \in \mathbb{N}$, $b \in \mathbb{N}^+$ sagen wir

b teilt a ohne Rest, „schreibweise $b|a$

$:\Leftrightarrow$

Es gibt ein $q \in \mathbb{N}$, so daß

$$a = q \cdot b \quad (\text{auch } a : b = q)$$

Etwa $3|3$, da $9 = 3 \cdot 3$, $3|21$
aber nicht $3|20$, „schreibweise $3 \nmid 20$
$3|0$, $7|0$, aber $0|7$ nicht definiert.
Es gilt nicht $2|3$ aber $2|0$.

Es gilt zwar $3 \nmid 20$, aber

$$20 = 6 \cdot 3 + 1 \quad (20 : 6 = 3 \text{ Rest } 1)$$

↑ Rest

Eine Verallgemeinerung des Teilbarkeitsbegriffs ist Teilbarkeit mit Rest:

Für $a \in \mathbb{N}$, $b \in \mathbb{N}^+$, r mit $0 \leq r < b$

b teilt a mit Rest r

$:\Leftrightarrow$

Es gibt ein $q \in \mathbb{N}$, so dass

$$a = b \cdot q + r \quad (\text{auch } a : b = q \text{ Rest } r)$$

5 teilt 21 mit Rest 1

$$21 = 4 \cdot 5 + 1.$$

Beachte ebenfalls $21 = 4 \cdot 5 + 6$ mit

5 teilt 21 mit Rest 6. Ebenso

$21 = 6 \cdot 5 - 1$, da hier $0 \leq r \neq 5$.

Zwei Bezeichnungen in diesem

Zusammenhang: $a \in \mathbb{N}$, $b \in \mathbb{N}^+$ (Voraussetzung nicht bei $\{a<0!$)

$\mathrm{Mod}(a,b)$ ist der Rest r ($a \% b$ Java)

$\mathrm{Div}(a,b)$ ist der Quotient q (a/b Java)

Statt r ist der Rest sagt man

auch: r ist der Modulus von a ...

bei Division durch b. Das $\mathrm{Div}(a,b)$

nennen wir auch Quotient von a

und b.

Kann es 2 verschiedene Reste oder Quotienten von a bei Division durch b geben? Eindeutigkeit von Quotient und Rest.

Man beachte, nicht alles ist eindeutig: Sei $a \in \mathbb{N}$. Dann ist die Wurzel von a das $b \in \mathbb{Z}$, so daß $a = b^2$. Dann $4 = 2^2$, $4 = (-2)^2$.

Also wie ist das bei den Resten und Quotienten? Sei $a \in \mathbb{N}$, $b \in \mathbb{N}^+$ und $0 \leq r \leq b$

$$a = q \cdot b + r.$$

Gibt es einen weiteren Rest, r' und Quotienten q', dann ebenfalls

$\in \mathbb{N}$

$\wr$

$a = q \cdot b + r'$

$\wr \qquad \wr$

$\in \mathbb{N} \qquad 0 \leq r' < b.$

Also ist

$$c \cdot b + r = q' \cdot b + r' \; (= a)$$

Also ist

$$r - r' = (q - q') \cdot b.$$

Da $0 \leq r < b$ und $0 \leq r' < b$ ist

$$\underbrace{-(b-1)}_{=-b+1} \leq r - r' \leq b-1$$

das heißt eben $|r - r'| \leq b-1$.

Andererseits ist $q - q' \in \mathbb{Z}$. Also

$(c - c')\, b$ kann nur folgende Werte

annehmen: $0 \cdot b, 1 \cdot b, (-1) \cdot b, 2 \cdot b, (-2) \cdot b, \ldots$

Also ist die einzige Möglichkeit die bleibt $q = q'$ und dann muß eben auch $r = r'$ sein.

Also: Eindeutigkeit von $Mod(a,b)$, $Div(a,b)$.

Existenz: Da $b \neq 0$ ist, ist irgendwann das erste Mal

$$b + b + \dots + b \geq a .$$

Ist $b + \dots + b = a$, dann

$$Div(a,b) = \#\, b\text{'s in der Summe}$$
$$Mod(a,b) = 0.$$

Ist $b + \dots + b \neq a$, dann

$$Div(a,b) = (\#\, b\text{'s}) - 1$$
$$Mod(a,b) = a - Div(a,b) .$$

Es ist zum Beispiel

$\mathrm{Mod}(20,8) = 4$ und $\mathrm{Mod}(52,8) = 4$

In diesem Falle wird die Mod-Schreibweise auch eingesetzt.

Man schreibt dann

$20 = 52 \bmod 8$ oder $20 \equiv 52 \bmod 8$

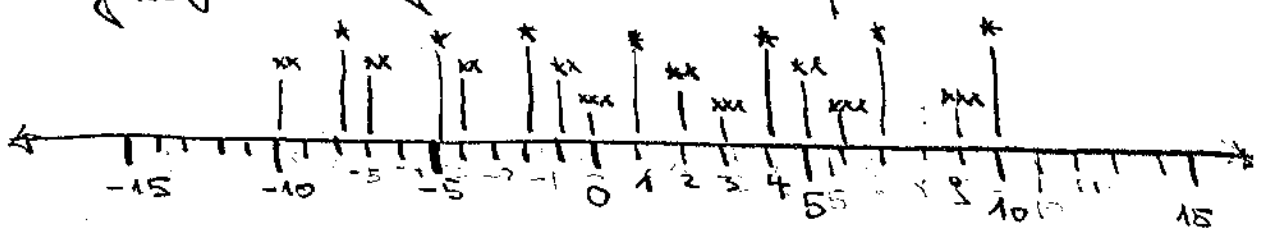
„gleicher Rest modulo 8."

Schließlich ist auch $a < 0$ möglich:

$\mathrm{Div}(-1,4) = -1 \qquad \mathrm{Mod}(-1,4) = +6$

$-1 = (-1)\cdot 7 + 6$ ← Bedingung $0 \leq r < 4$.

Grafisch: Gleichheit mod 3.



* entspricht $3\cdot\mathbb{Z}+1$ xx $3\cdot\mathbb{Z}+2$

xxx entspricht $3\cdot\mathbb{Z} = 3\cdot\mathbb{Z}+3$.

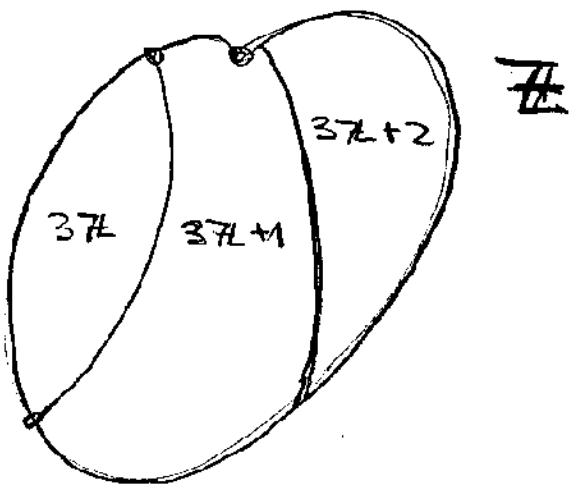
Beachte

$\mathbb{Z} = 3\cdot\mathbb{Z} \dot\cup 3\cdot\mathbb{Z}+1 \dot\cup 3\cdot\mathbb{Z}+2$

Vereinige disjunkt,

d.h. ohne gemeinsame Elemente.

Partition = disjunkte Zerlegung.



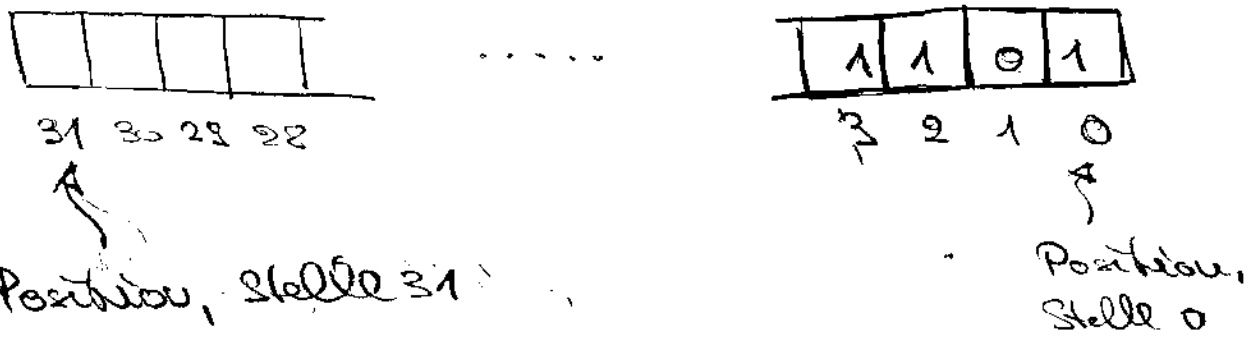
Partition entspricht Äquivalenz-
relation

Ausprobieren in Java: $-7 \,\%\, 3$, $-7 \,\%\, -3$
und auch einmal $4 \,\%\, -3$. $4 = (-3)\cdot(-2) + 1$
$-6/-3$ $\quad -6 = -3\cdot 2$ $\quad -7 \,\%\, -3$ $\quad -7 = -3\cdot 3 + 2$

2.10

Wie werden Zahlen im Rechner dargestellt, d.h. in welcher Form werden sie gespeichert?

Ein Speicherwort des Hauptspeichers



Pro Position wird 1 Bit gespeichert

8 Bits = 1 Byte.

Speicherworte üblicherweise

32 Bit = 4 Bytes

oder manchmal auch

64 Bits = 8 Bytes.

Zahlen (eigentlich alles) muß letztlich über die Bits 0, 1, also als Folge von Bits dargestellt werden.

Die Darstellung von Zahlen basiert auf dem Dualsystem. Zunächst einige Beispiele dazu:

0 dezimal ist 0 dual.

1 dezimal ist 1 dual.

2 dezimal ist 10 dual $(10)_2$

3 dezimal ist $(11)_2$

10 dezimal ist $(1010)_2$

"Ast im Dualsystem"

Das Zehner- oder Dezimalsystem beruht auf folgendem Prinzip:

$$1 = 1 \cdot 10^0$$

Beachte $10^0 = 1$, sinnvoll da $10^{a-a} = \frac{10^a}{10^a} = 1$.

$$2 = 2 \cdot 10^0, \dots, 9 = 9 \cdot 10^0$$

$$10 = 1 \cdot 10^1 + 0 \cdot 10^0$$

$$12 = 1 \cdot 10^1 + 2 \cdot 10^0$$

$$100 = 1 \cdot 10^2$$

$$0 = 0 \cdot 10^0.$$

Für $a \in \mathbb{N}$ gilt: Es gibt ein $m \geq 1$ (Anzahl Stellen, # Stellen) und Ziffern $z_0, z_1, \dots, z_{m-1}$ mit $0 \leq z_i \leq 10$

so daß

$$a = z_{m-1} \cdot 10^{m-1} + z_{m-2} \cdot 10^{m-2} + \dots + z_1 \cdot 10^1 + z_0 \cdot 10^0 .$$

Man schreibt dann auch

$$a = \sum_{i=0}^{m-1} z_i \cdot 10^i .$$

≥ 0, da $m \geq 1$ vorausgesetzt ist

(Entspricht der Zahl $(z_{m-1} z_{m-2} \dots z_0)_{10}$.)

↑ „Dezimalsystem"

Wieso ist eigentlich jedes $a \in \mathbb{N}'$ wie oben darstellbar? Das läßt sich aus unserer Division mit Rest ableiten: Ist $a \in \mathbb{N}$, dann haben wir Reste z_0 mit $0 \leq z_0 < 10$ und q_1 so daß

$$a = q_1 \cdot 10 + z_0$$

$\mathrm{Div}(a,10^0)$ $\mathrm{Div}(a,10)$ $\mathrm{Rest}(a,10)$

Weiter $\leftarrow$ Div(a, 10)

$a = \text{Div}(a,1)$
$= 10^0$

$q_1 = q_2 \cdot 10 + z_1$ $\leftarrow$ Mod(Div(a, 10), 10)

und

$q_2 = q_3 \cdot 10 + z_2$

$\vdots$

$q_{m-1} = 0 \cdot 10 + z_{m-1}$

q_m)

Div(Div(a, 10), 10)
= Div(a, 100) . Denn

$a = c \cdot 100 + r$

$a = c_1 \cdot 10 + z_0$

$q_1 = c_2 \cdot 10 + z_1$ < 100

$\Rightarrow a = q_2 100 + z_1 10 + z_0$

Einsetzen ergibt nun:

Div(Div(a, 10), 10)

$a = q_0 \cdot 10 + z_0$ $= \text{Div}(a, 100)$

$= (q_1 \cdot 10 + z_1) \cdot 10 + z_0$

$= ((q_2 \cdot 10 + z_1) 10 + z_1) 10 + z_0$

$= (((q_3 \cdot 10 + z_2) \cdot 10 + z_2) \cdot 10 + z_1) \cdot 10 + z_0$

$\vdots$

$= z_{m-1} \cdot 10^{m-1} + z_{m-2} \cdot 10^{m-2} + \dots + z_2 \cdot 10^2 + z_1 \cdot 10 + z_0$

Also: Existenz der ganzzahligen Darstellung.

Wieso ist die Darstellung im Dezimalsystem eindeutig? Angenommen es wäre für ein $a \in \mathbb{N}^+$

$$a = \sum_{i=0}^{m-1} z_i \cdot 10^i \quad \text{und} \quad a = \sum_{j=0}^{m-1} y_j \cdot 10^j$$

und $0 \leq z_i, y_i \leq 10$. Dann folgt, daß

$$z_0 = y_0$$

ist, da $z_0 = \text{Mod}(a,10)$ und $y_0 = \text{Mod}(a,10)$.

und Division mit Rest eindeutig.

Dann ist

$$\frac{a - z_0}{10} = \sum_{i=1}^{m-1} z_i \cdot 10^{i-1} = \sum_{j=1}^{m-1} y_j \cdot 10^{j-1}$$

$0 \leq z_i \leq 9$

Def. $\text{Div}(a,10)$

,

deren $q_1 = \sum_{i=1}^{m-1} z_i \cdot 10^{i-1} = \sum_{j=1}^{m-1} y_j \cdot 10^{j-1}$

Dann gilt für z_1 und y_1, daß

$$\text{Mod}(q_1, 10) = z_1 \text{ und } \text{Mod}(q_1', 10) = y_1$$

Also $z_1 = y_1$, dann $z_2 = y_2$, ... Ist etwa $m \geq n$, dann haben wir $y_{m-1} = \dots = y_m = 0$. { Dann $m-1 \geq n$ }

Also insgesamt: Eindeutigkeit abgesehen von Nullen vorne.

Statt mit 10 kann man ganz analog mit jedem $b \in \mathbb{N}^+$, $b \geq 2$ verfahren.

Dualsystem $b = 2$. ($b = 1$ geht nicht so, da $1^{z_i} = 1$, und $0 \leq z_i < 1$ $z_i = 0$ erfordert.)

Also jedes $a \in \mathbb{N}$ ist abgesehen von Nullen vorne eindeutig darstellbar als

$$a = \sum_{i=0}^{n-1} z_i \cdot 2^i, \quad n \ge 1 \text{ und } 0 \le z_i < 2$$

also Ziffern aus 0, 1. Dann auch die Schreibweise $a = (z_{n-1} \dots z_0)_2$.

$$(4)_{10} = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = (100)_2$$

$z_0 = Mod(4,2) = 0$,
$q_1 = Div(4,2) = 2$

$z_1 = Mod(2,2) = 0$,
$q_2 = Div(2,2) = 1$

$z_2 = Mod(1,2) = 1$,
$q_3 = Div(1,2) = 0$ ← Deshalb Schluß

$q_4 = 0$.

$= 2^3$

Stelle 3 von hinten, beginnend 4 = 3mal Durchlauf mit dem Wert
$Mod(Div(Div(Div(4,2),2),2),2)$

Was ist die größte Zahl, die man auf n Stellen darstellen kann?

$$(\underbrace{1 \dots 1}_{n-1})_2 = 1 \cdot 2^{n-1} + 2^{n-2} + \dots + 2^0 = 2^n - 1 .$$

Denn es ist

$$2^0 + 2^1 = 3 = 2^2 - 1$$

$$2^0 + 2^1 + 2^2 = 7 = 2^3 - 1$$

$$2^0 + 2^1 + 2^2 + 2^3 = 16 - 1 .$$

$$\vdots$$

$$\underbrace{2^0 + 2^1 + 2^2 + \dots + 2^{n-2}}_{= 2^{n-1} - 1} + 2^{n-1} = 2^n - 1 .$$

(2.12)

Das zeigt ein Induktionsbeweis.

Es wird gezeigt für alle $n \geq 1$ ist

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 .$$

Induktionsanfang: $n = 1$

Es ist

$$2^0 = 1 = 2^1 - 1.$$

Induktionsschluß. Angenommen es gilt die Behauptung für $n-1$.

Also

$$\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1.$$

Dann gilt

$$\sum_{i=0}^{n-1} 2^i = 2^{n-1} - 1 + 2^{n-1} = 2^n - 1$$

Die kleinste Zahl mit n Stellen, wo eine 1 ist 2^{n-1}.

Also

$$2^n - 1 \geq 1\underbrace{z_{n-2} \ldots z_0}_{n-1 \text{ Stellen}} \geq 2^{n-1}$$

$$2 \cdot 2^{n-1} - 1 \qquad z_{n-1}$$

Ausgerechnet

$2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 1 + 1 = 2^n$

Die Stell

Es gilt nach allgemeiner die Formel für die geometrische Reihe

$$\sum_{i=0}^{m-1} a^i = \underbrace{a^0}_{=1} + a^1 + a^2 + a^3 + \dots + a^{m-1}$$

$$= \frac{a^m - 1}{a - 1} \quad \text{für alle } a \neq 1.$$

Der Beweis ist eine Übungsaufgabe.

Analog im Dezimalsystem: Größte Zahl mit m Stellen

$$\underbrace{(9\,9 \dots 9}_{m-1}\underbrace{9)}_{0}{}_{10} = 10^{m-1} \cdot 9 + 10^{m-2} \cdot 9 + \dots + 9 \cdot 10 + 9$$

$$= 9 \cdot \sum_{i=0}^{m-1} 10^i$$

$$= 9 \cdot \frac{10^m - 1}{10 - 1} = 10^m - 1.$$

Noch einige Beispiele von Dualzahlen: (2.21)

$(8)_{10} = (1.000)_2$

$(16)_{10} = (1.0000)_2$

$(32)_{16} = (100000)_2$

2^5

$(64)_{10} = (1000000)_2$

$(128)_{10} = (10000000)_2$

Eine Stelle mehr führt zur Multiplikation mit 2. Exponentielles Wachstum, kombinatorische Explosion!

$(256)_{10} = (1\underbrace{00000000}_{8 \text{ Nullen}})_2$

2^8

$(512)_{10} = (1\underbrace{000000000}_{9 \text{ Nullen}})_2$

2^9

$(1024)_{10} = (1\underbrace{0000000000}_{10 \text{ Nullen}})_2$

2^{10}

$1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + \ldots + 0 \cdot 2^1 + 0 \cdot 2^0$.

2.22

$$(1000)_{10} \cdot (1000)_{10} = (1.000.000)_{10}$$

also ist

$$2^{10} \cdot 2^{10} \quad ? \; (1.000.000)_{10}$$

Andererseits

$$2^{10} \cdot 2^{10} = 2^{10+10} = 2^{20} = (10\ldots\ldots0)_2$$

wobei wir 20 Stellen haben.

Wieviele Stellen braucht man zur binären Darstellung von $a \in \mathbb{N}$?

0	$(0)_2$	1 Stelle
1	$(1)_2$	1 Stelle
2	$(10)_2$	2 Stellen
4	$(100)_2$	3 Stellen
⋮		

2.23

$2^{10} = 1024$ $(1\underbrace{000\,0000000}_{10\text{ Nullen}})_2$ 11 Stellen

Die 0 ist nicht von dieser Form.

Zur Darstellung von Zweierpotenzen $a = 2^m$ braucht man genau $m+1$ Stellen. (Gilt auch für $m = 0$.)

$m+1$: Anzahl der Ziffern

m Nullen

Dann ist

$$m = \log_2 a$$

$2^m : 2 = 2^{m-1}$

$2^{m-1} : 2 = 2^{m-2}$

$m+1$ = # Stellen, die man durch 2 dividieren kann, bis $\neq 2$.

Wie bei Nicht-Zweierpotenzen?

2	3	4	5 ... 7	8
$(10)_2$	$(11)_2$	$(100)_2$	3 Stellen	1000

Es ist $2 \leq \log_2 5, \log_2 6, \log_2 7 \leq 3$

etwa $\log_2 5 = 2,321\ldots$ oder ähnlich.

#Stellen von a

= der Abstand zu, so dass

$$Div(0, 2^i) = 0$$

Dann ist $\lfloor 2{,}31 \rfloor = 2{,}0$, $\lfloor 2{,}0 \rfloor = 2$, $\lfloor 2{,}9 \rfloor = 2$
(ganz klassisch floor). Also ist jedes
$a \in \{4, \dots, 7\}$ mit

$$\lfloor \log_2 a \rfloor + 1$$

Stellen darstellbar.

Beachte
$\lceil \log_2 a \rceil$ stimmt
bei $a = 4$ nicht.

Stellen
bis Division
durch $2 \leq 1$
ist. Dann
ist aber 0
das Ziel.

Ist immer, sogar für 0, wenn
$\log_2 0 := 0$ definiert wird.

Es ist für $a \geq 1$

$$\lfloor \log_2 a \rfloor + 1 = \lceil \log_2 (a+1) \rceil$$

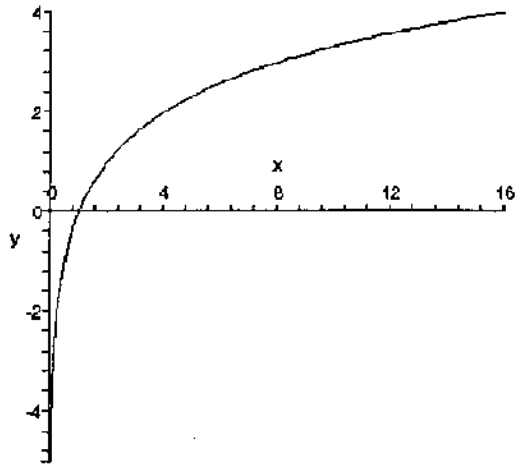
$\left(\lceil 2{,}1 \rceil = 3 \right)$

$\lceil 2{,}7 \rceil = 3$, (ceiling, Decke)

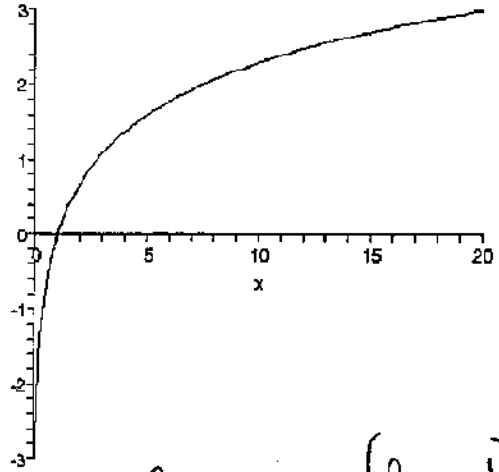
Für $a = 2^m$ Zwischenstellen haben wir $m+1$
für $a = 2^m + b$ $\quad 1 \leq b \leq 2^m$ ist
$2^m + 2 \leq a+1 \leq 2^{m+1}$ so gilt es auch.
Nicht für $a = 0$ jedenfalls ist $\lceil \log_2 \rceil = 0$.

Logarithmusfunktionen

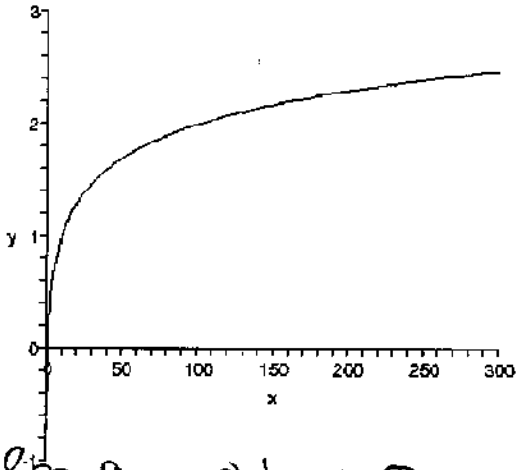
Dyadischer Logarithmus



Natürlicher Logarithmus



Dekadischer Logarithmus



$\log_q a^a = (\log_q b)\log_b a \quad q,b,a>0$

$\log_b a = x \Leftrightarrow a = \underbrace{b\cdot b\cdot b\cdot \ldots \cdot b}_{x} \cdot \underbrace{b^{\square}}_{<b}$

Logarithmengesetze:

$\log_a a = 1$

$\log 1 = 0$

$\log(ab) = \log a + \log b$

$\log\frac{a}{b} = \log a - \log b \quad (a>0, b>0)$

$\log a^n = n\cdot\log a$

$\log\sqrt[n]{a} = \frac{1}{n}\cdot\log a$

29.05

Grundsätzlich mit verschiedenen Basen kann Beispielhaft. Dezimal → dual mit Resten. Dual → Dezimal folgendermaßen.

$(100000)_2 = (32)_{10}$ wird

systematisch ermittelt durch:

$(100000)_2 : (1010)_2 = (11)_2$ Rest $(10)_2$

$-(1010)_2$

$(01100)_2$

$-(1010)_2$

$(0010)_2$

Das ist

Div $((10000)_2, (1010)_2)$

$(11)_2 : (1010)_2 = 0$ Rest $(11)_2$

deshalb Abbruch.

Also ist

$(100000)_2$

$= (11)_2 \cdot (1010)_2 + (10)_2$

$= 3 \cdot 10 + 2$

$= (32)_{10}$.

2.25

$(10000000)_2 : (1010)_2 =$

-1010

01100

1010

1000

$= (1100)_2 \text{ Rest } (1000)_2$

Nun ist $(1100)_2 > (1010)_2$ also

weiterteilen

$(1100) : (1010)_2 = 1 \text{ Rest } (10)_2$

1010

10

Es ist

$(1) : (1010)_2 = 0 \text{ Rest } (1)_2$

$(10)_2$

$(1)_2$ $(1000)_2$

Also

$(10000000)_2 = (128)_{10}$

Multiplikation mit 2 im Dualsystem: Eine Null anhängen.
(Verschieben (shift) nach links.)

$$Div((1010)_2, (10)_2) = (101)_2$$

$$Mod((10011)_2, (10)_2) = (1)_2.$$

Bei Speicherworten der Länge etwa 4 (4 Bits) können wir so die Zahlen $(0)_{10} = (0000)_2$,
$(1)_{10} = (0001)_2, \ldots, (1111)_2 = (15)_{10}$ darstellen. Bei Länge 32 etwa $0, \ldots, \underbrace{(111\ldots1)_2}_{32} = 2^{32}-1$
$= 4.000.000.000$
größte Zahl

Also $16 = 2^4$ größte Zahl $= 16-1 = 15$
4 Stellen ohne Leerstelle
$2^3 = 8$, von $(000)_2$ bis $(111)_2$ von 0 bis 7 oder 8 Werte.

Was ist, wenn die Addition aus dessen Bereich hinausführt?

Wir lassen den Übertrag, der rausfällt weg! Was heißt das dann?

$$(1111)_2 + (0001)_2 = (10000)_2 = (16)_{10},$$

wird dann aber zu Null.

$$(1111)_2 + (0010)_2 = (10001)_2 = (17)_{10},$$

wird aber dann zu 1.

$$(1111)_2 + (1111)_2 = (11110) = (30)_{10}$$

wird zu $(1110)_2 = (14)_{10}$.

Es ist

$$\mathrm{Mod}\left((z_{m-1} \dots z_0)_2, \underbrace{(10000)_2}_{=(16)_{10}}\right) = (z_3, z_2, z_1, z_0)_2$$

und allgemein:

$$\mathrm{Mod}((z_{n+1} \dots z_0)_2, \underbrace{1 0 \dots 0}_{i \text{ Nullen}}) = (z_{i-1} \dots z_0)_2$$

Abschneiden der ersten Stelle bei Binärzahlen bedeutet also Rechnen auf den Resten mod 16.

Ebenso bei etwa 32 Positionen:
mod 2^{32}. Ist $0 \leq a, b, a+b \leq 2^{32}$,
dann ist

$$\mathrm{Mod}(a+b, 2^{32}) = a+b$$

Ist aber $0 \leq a, b \leq 2^{32}$ und $a+b \geq 2^{32}$,
dann ist

$$\mathrm{Mod}(a+b, 2^{32}) = \underbrace{a+b-2^{32}}_{\leq 2^{32}},$$

da $a+b \leq 2^{33}$ ist.

Eine weitere Möglichkeit auch negative Zahlen zu bekommen, ist die Einführung eines Vorzeichenbits nach folgendem Muster:

0111	7
⋮	
0001	1
0000	0
1000	-0
1001	-1
⋮	
1111	-7

Das scheint aber auch einen eigenen Algorithmus zur Subtraktion. Es geht auch einfacher.

Das Ziel ist es, die Subtraktion auf die Addition mit Abschneiden der eventuell überzählig gen Stelle zurückzuführen. Im Beispiel von $n = 2$, also Zahlen $(x_1 x_0)_2$ sieht es so aus:

00	01	10	11
0	1	2	3

Es ist

$$11 + 10 = 100$$

das ergibt dann 00. Interpretieren wir doch einfach 11 als -1.

Dann weiter im

$$11 + 11 = 110$$

ergibt 10. Wie ist 10 zu interpretieren? Als -2.

Die neue Interpretation ist die

2'e Komplement

Darstellung:

00	01	10	11
0	1	-2	-1
	2^1-1	-2^1	-2^1+1

Allgemein auf n Positionen dezt.

n Nullen			
0 ... 0	0 ... 1	...	01...1
0	1		$2^{n-1}-1$

		n Einsen
10 ... 0	10 ... 01	11 ... 1
-2^{n-1}	$-2^{n-1}+1$	-1

Insgesamt haben wir hier 2^n

Werte.

Die offizielle Definition der 2'er kpl. Darstellung auf n Bits ist jetzt:

- Ist $0 \leq a \leq 2^{n-1} - 1$, dann ist das

2'er kpl. von a = Binärdarstg. von a auf n Bits. (Vorne eine 0)

- Ist $-2^{n-1} \leq a \leq -1$, etwa $a = -a'$, dann

2'er kpl. von a = Binärdarstg von $2^n - a' = 2^n + a$. auf n Bits (Vorne eine 1).

Man kann auch alternativ und prägnanter definieren:

2'er kpl. von a

= Binärdarstellung auf n Bits von $\text{Mod}(a, 2^n)$.

Man beachte: $0 < a \leq 2^n$, dann $\text{Mod}(a, 2^n) = a$

$-2^n \leq a < 0$, dann $\text{Mod}(a, 2^n) = 2^n + a$, $a \neq 0$.

Es ist bekanntlich $\text{Mod}(-1, 2^n) = 2^n - 1$, da $-2^n + (2^n - 1) = -1$, $\text{Mod}(-2^{n-1}, 2^n) = 2^{n-1}$.

Rest

Wir haben also jetzt 2 Möglichkeiten, eine Bitfolge als Zahl zu interpretieren: Als Binärzahl oder als 2'er kpl. Darstellung. Aus der Bitfolge $z_{n-1} \dots z_0$ ergibt sich die Binärzahl bekanntlich als

$$(z_{m-1} \dots z_0)_2 = \sum_{i=0}^{m-1} 2^i \cdot z_i .$$

Dagegen haben wir beim 2'er kpl:

$$(z_{n-1} \dots z_0)_{2\text{kpl}} = \begin{cases} (z_{m-1} \dots z_0)_2 & \text{falls } z_{n-1} = 0 \\ \underbrace{(z_{m-1} \dots z_0)_2 - 2^n}_{\neq 0} & \text{falls } z_{m-1} = 1 \end{cases}$$

Man prüft leicht noch, daß die Abbildung

$a \rightsquigarrow$ 2'er kpl Darst. von a ist $z_{n-1} \dots z_0$

$\rightsquigarrow (z_{n-1} \dots z_0)_2$ falls $z_{n-1} = 0$ bzw.

$(z_{n-1} - z_0)_2 - 2^n$ falls $z_{n-1} = 1$

in jedem Falle wieder a ergibt.

Zunächst ist nun der folgende Satz, der besagt, daß die normale binäre Addition hinreichend ist, um auf der 2'er Kpl'ten zu rechnen.

Satz

Sei $-2^{n-1} \leq a, b \leq 2^{n-1}-1$ und

$(x_{n-1} \dots x_0)_{2\text{Kpl}} = a$, $(y_{n-1} \dots y_0)_{2\text{Kpl}} = b$.

Sei außerdem

$$(z_n z_{n-1} \dots z_0)_2 = (x_{n-1} \dots x_0)_2 + (y_{n-1} \dots y_0)_2$$

(also eine normale binäre Addition).

(a) Für alle a, b wie oben gilt:

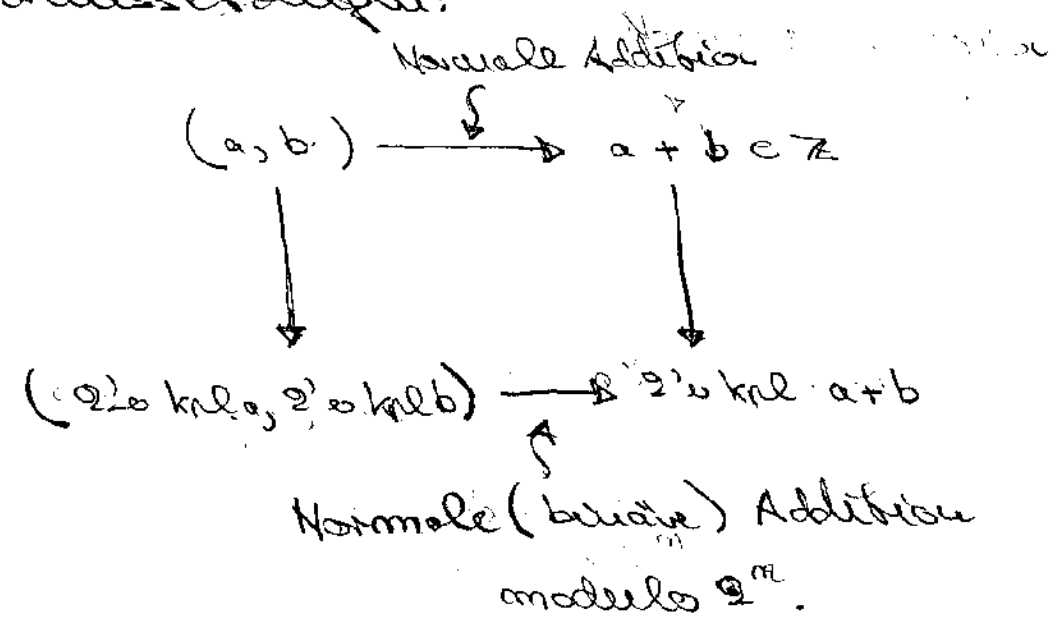
$$\underbrace{\text{Mod}\left((z_n z_{n-1} \dots z_0)_2, 2^n\right)}_{= (z_{n-1} \dots z_0)_2} = \underbrace{\text{Mod}(a+b, 2^n)}_{-2^n \leq a+b \leq 2^n - 2}$$

(b) [illegible]

(b) Ist nun auch noch $-2^{n-1} \leq a+b \leq 2^{n-1}-1$, so ist $z_{n-1} \dots z_0$ das 2's kpl. von $a+b$, das heißt

$$(z_{n-1} - z_0)_{2\,\text{kpl}} = a+b.$$

(Das heißt, addieren wir die beiden 2's kpl's einfach binär, bilden dann den Rest modulo 2^n, so haben wir das 2's kpl von $a+b$. Das folgende Diagramm „kommutiert" unter den gegebenen Voraussetzungen:



Beweis

(a) Die Reste modulo 2^m bleiben gleich, sofern wir Vielfache (positive oder negative) von 2^u zu einem Wert hinzuaddieren.

(b) Für (a) gilt, daß

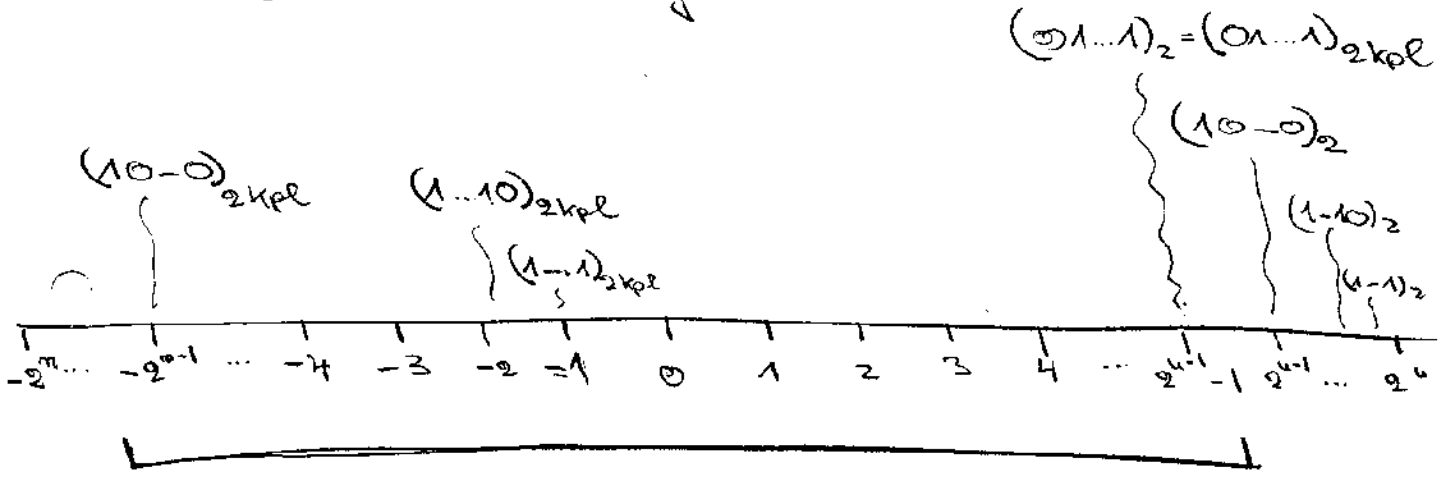
$$\underbrace{(z_{u-1} \dots z_0)_2}_{\leq 2^u} = \text{Rest}(a+b, 2^m)$$

ist. Da $a+b$ in dem fraglichen Bereich ist, $-2^{u-1} \leq a+b \leq 2^{u-1}-1$, ist nach der alternativen Definition über $z_{m-1} \dots z_0$ die 2'er kpl. Darstellung von $a+b$, also

$$(z_{m-1} - z_0)_{2\text{kpl}} = a+b. \qquad \square$$

Hinweis: Beweis 2.39 ...

Noch einmal das 2's Komplement an der Zahlengeraden:



Bereich des 2's kpl's d.

Es ist $-2^{n-1} = -2^n + 2^{n-1}$

also $\mathrm{Mod}(-2^{n-1}, 2^n) = 2^{n-1} = (1\ldots0)_2$

Ebenso $-2^{n-1}+1 = -2^n + (2^{n-1}+1)$

also $\mathrm{Mod}(-2^{n-1}+1, 2^n) =$

$= 2^{n-1}+1 = (10\ldots01)_2$

Schließlich $-1 = -2^n + (2^n - 1)$

also $\mathrm{Mod}(-1, 2^n) = (1\ldots1)_2$.

Folgender Trick zur Ermittlung der 2'er kpl-Darst. mit n Bits sollte man kennen:

Für $0 \leq a \leq 2^{n-1}-1$ gilt es mittels der Binärdarstellung von a.

Für $-2^{n-1} \leq a \leq -1$ und $a = -a'$ ist es die Binärdarst. von $2^n + a = 2^n - a'$.

Nun ist

$$2^n - 1 - 2^n - a' = 2^n - 1 - a' + 1.$$

Ist $(x_{n-1} \dots x_0)_2 = a'$, so ist die Binärdarstellung von $2^n - 1 - a'$ einfach

$$(\bar{x}_{n-1}\, \bar{x}_{n-2} \dots \bar{x}_0)_2 = 2^n - 1 - a',$$

mit $\bar{x}_i = 1 - x_i$ ($\bar{x}_i = 0$ wenn $x_i = 1$, $\bar{x}_i = 1$ wenn $x_i = 0$).

$\bar{x}_{n-1} \dots \bar{x}_0$ ist das 1'er Komplement von $x_{n-1} \dots x_0$, zu dem dann nur noch 1 addiert werden muß, um zum 2'er Komplement zu kommen.

Betrachtung des Java Programms ZuKlTest.java und ZuByteTest.java.

Wir haben in Java 4 verschiedene Typen für ganze Zahlen. Arbeiten mit dem 2'er kpl. Dstg.

byte 1 Byte = 8 Bit von -128 bis 127
($2^8 = 256$)

short 2 Byte = 16 Bit von -32768 bis 32767
($2^{16} = 65536$)

int 32 Bit von -2^{31} bis $2^{31}-1$
($2^{31} \approx 2$ Milliarden)
long 64 Bit von -2^{63} bis $2^{63}-1$.

Literalkonstanten angeben

Literal.java

Der Compiler macht folgendes:

1. Parameter vom Typ long. Ergebnis von arithmetischer Operation ist long, gibt Compilerfehler bei kleiner int.
2. Kein Parameter long, dann ist. Egal ob Typen byte oder so.

Die linke Seite muß dann mit sein. Oder explizite Verkleinerung des Bereiches rechts.

Wie in (byte) (a+b), selbst nötig, wenn a; b byte sind, da dann a+b als int gesehen wird.

Implizite Typanpassung nur dann, wenn kleinere Bereiche in den größeren: Recht ist byte long geht automatisch.

Das Programm PRIM in PRIM.java.
Korrektheitsbeweis: W ist die Schleife mit dem return statement verlassen: $\Rightarrow$ c_{stat} keine Primzahl. W ist überlegen uns: W ist die Schleife nicht mit dem return statement verlassen $\Rightarrow$ c_{stat} Primzahl.

Sei wieder

d_l = Wert von d nach l'ten Lauf,

$l \geq 1$, sofern es stattfindet. Außerdem

$d_0 = 2$.

Erste Invariante: N-> [illegible] → 2.44a

Lauf ℓ: $d_\ell = \ell + 2$

(gilt nach dem ℓ'ten Lauf (sofern es stattfindet).

Sieht man durch Inspektion des Rumpfes.

Zweite Invariante:

$2, 3, 4, \ldots, d_\ell - 1$ teilen q nicht,

gilt nach dem ℓ'ten Lauf (sofern es stattfindet). Leichte Inspektion.

Die Schleife läuft bis $d_\ell \cdot d_\ell \geq q$ ist. Da $d_\ell = \ell + 2$ hält sie also in jedem Falle an. Ist ℓ so, daß $d_\ell \cdot d_\ell \geq q$ ist, gilt

$2, 3, \ldots, d_\ell - 1$ teilen q nicht.

Da $d_\ell \geq \sqrt{q}$ ist, deshalb q Primzahl, denn wir haben keinen Teiler unter $2, \ldots, \sqrt{q}$

falls c Quadratzahl und keinen unter $2, \dots, \lfloor \sqrt{c} \rfloor$ falls c keine Quadratzahl. Die Anzahl der Durchläufe ist $\sqrt{c} - 2$ falls c Quadratzahl, $\lceil \sqrt{c} \rceil - 2$ falls c keine Quadratzahl.

Etwa: $c = 5$, dann $d_1 = 3$ Schleife und $\lceil \sqrt{5} \rceil = 3$. $c = 7$, dann $d_1 = 3$ Schleife.

$c = 11$, dann $d_1 = 3$, $d_2 = 4$ Schleife.

Table 1

USEFUL PRIME NUMBERS

N	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
2^{15}	19	49	51	55	61	75	81	115	121	135
2^{16}	15	17	39	57	87	89	99	113	117	123
2^{17}	1	9	13	31	49	61	63	85	91	99
2^{18}	5	11	17	23	33	35	41	65	75	93
2^{19}	1	19	27	31	45	57	67	69	85	87
2^{20}	3	5	17	27	59	69	129	143	153	185
2^{21}	9	19	21	55	61	69	105	111	121	129
2^{22}	3	17	27	33	57	87	105	113	117	123
2^{23}	15	21	27	37	61	69	135	147	157	159
2^{24}	3	17	33	63	75	77	89	95	117	167
2^{25}	39	49	61	85	91	115	141	159	165	183
2^{26}	5	27	45	87	101	107	111	117	125	135
2^{27}	39	79	111	115	135	187	199	219	231	235
2^{28}	57	89	95	119	125	143	165	183	213	273
2^{29}	3	33	43	63	73	75	93	99	121	133
2^{30}	35	41	83	101	105	107	135	153	161	173
2^{31}	1	19	61	69	85	99	105	151	159	171
2^{32}	5	17	65	99	107	135	153	185	209	267
2^{33}	9	25	49	79	105	285	301	303	321	355
2^{34}	41	77	113	131	143	165	185	207	227	281
2^{35}	31	49	61	69	79	121	141	247	309	325
2^{36}	5	17	23	65	117	137	159	173	189	233
2^{37}	25	31	45	69	123	141	199	201	351	375
2^{38}	45	87	107	131	153	185	191	227	231	257
2^{39}	7	19	67	91	135	165	219	231	241	301
2^{40}	87	167	195	203	213	285	293	299	389	437
2^{41}	21	31	55	63	73	75	91	111	133	139
2^{42}	11	17	33	53	65	143	161	165	215	227
2^{43}	57	67	117	175	255	267	291	309	319	369
2^{44}	17	117	119	129	143	149	287	327	359	377
2^{45}	55	69	81	93	121	133	139	159	193	229
2^{46}	21	57	63	77	167	197	237	287	305	311
2^{47}	115	127	147	279	297	339	435	541	619	649
2^{48}	59	65	89	93	147	165	189	233	243	257
2^{59}	55	99	225	427	517	607	649	687	861	871
2^{60}	93	107	173	179	257	279	369	395	399	453
2^{63}	25	165	259	301	375	387	391	409	457	471
2^{64}	59	83	95	179	189	257	279	323	353	363
10^6	17	21	39	41	47	69	83	93	117	137
10^7	9	27	29	57	63	69	71	93	99	111
10^8	11	29	41	59	69	153	161	173	179	213
10^9	63	71	107	117	203	239	243	249	261	267
10^{10}	33	57	71	119	149	167	183	213	219	231
10^{11}	23	53	57	93	129	149	167	171	179	231
10^{12}	11	39	41	63	101	123	137	143	153	233
10^{16}	63	83	113	149	183	191	329	357	359	369

The ten largest primes less than N are $N - a_1, \ldots, N - a_{10}$.

Zunächst einmal noch zwei grundsätzliche Dinge zu Primzahlen:

Es ist eines der schönsten Beweise der Mathematik (von Euklid, ca. 300 v. Christus), daß es unendlich viele Primzahlen gibt, also

$$|\mathbb{P}| = \infty$$

$$\mathbb{P} = \{2, 3, 5, 7, \ldots\}$$

Das sieht man recht leicht, indem man zu jeder endlichen Menge von Primzahlen eine neue Primzahl angibt, die nicht zu dieser Menge gehört:

(2.43)

Die Konstruktion am Beispiel:

Wenn $M = \{2, 3, 7\}$, dann

betrachte zunächst $b = 2 \cdot 3 \cdot 7 + 1 = 43$

Dann gilt zunächst einmal

$$2 \nmid b, \quad 3 \nmid b, \quad 7 \nmid b.$$

Also keine der Zahlen der Menge teilt b.

b braucht im allgemeinen keine Primzahl zu sein, hat aber einen kleinsten Teiler $\neq 1$.

Dieser muß Primzahl sein.

Dieser kleinste Teiler ist,

wie gesagt, nicht in unserer Menge

und also haben eine weitere Primzahl.

Genau sogar

$q_1 \cdots q_b + 1$

und mit

von q_j

geteilt!

(2.48)

Analog eben bei beliebige endliche Menge von Primzahlen $p_1, \dots, p_n$.

Es ist nicht direkt $p_1 \cdots p_n + 1$ Primzahl, wenn $p_1, \dots, p_n$ die ersten Primzahlen sind:

$2 \cdot 3 + 1 = 7$, $2 \cdot 3 \cdot 5 + 1 = 31$, $2 \cdot 3 \cdot 5 \cdot 7 + 1 = 211$

$2 \cdot 3 \cdot 5 \cdot 7 \cdot 9 + 1 = 1891$ keine Primzahl.

Ebenso klappt es nicht bei

$2 \cdot 3 + 1 = 7$ $2 \cdot 3 \cdot 7 + 1 = 43$,

$2 \cdot 3 \cdot 7 \cdot 43 + 1 = 1807$, auch

keine Primzahl mehr.

Haben wir aber: Der kleinste Teiler > 2 einer Zahl muß eine Primzahl sein!

2.4B

Primzahlen besitzen ihre Bedeutung (auch) aus dem Fundamentalsatz der Arithmetik:

Jede Zahl $a \in \mathbb{N}$, $a \geq 1$ läßt sich schreiben als

$$a = p_1^{e_1} \cdot \ldots \cdot p_m^{e_m}$$

$p_1, \ldots, p_m$ sind paarweise verschiedene Primz.; $e_i \geq 1$.

Diese Darstellung ist im wesentlichen eindeutig. "B.s auf Reihenfolge"

Etwa $a = 8 \; (= 7^1)$. $a = 8 = 2^3$,

$15 = 3 \cdot 5 = 5 \cdot 3$, $30 = 3 \cdot 2 \cdot 5 = 2 \cdot 3 \cdot 5 \ldots$

$24 = 3 \cdot 2^3 = 2^3 \cdot 3$. (Bruns Grundschuh, Zahlentheorie)

Wozu kann man eine solche Fallunterscheidung verwenden? Man kann zum Beispiel die ganzzahligen Teiler von n ablesen: Sei

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k},$$

dann gilt

$m \mid n$ gdw. $m = p_1^{f_1} p_2^{f_2} \ldots p_k^{f_k}$,

wobei $f_i \leq e_i$ für alle i.

„$\Leftarrow$" gilt offensichtlich.

„$\Rightarrow$" Sei $m \mid n$, dann $n = q \cdot m$,

falls zum Beispiel $m = p_1^{f_1} m'$,

$f_1 > e_1$, dann

$$n = p_1^{f_1} m' \cdot q.$$

Also hätten wir eine zweite

Primfaktorzerlegung im Widerspruch zur Eindeutigkeit. Ebenso;

falls $m = p \cdot m'$ und $p \neq p_i$

für alle i.

Primzahlen $q_1 \leq q_2 \leq q_3 \leq \dots \leq q_k$
und $a = q_1 \cdot q_2 \cdot q_3 \cdots q_k$.

Also: Wie kann man jetzt die Primfaktorzerlegung von einer Zahl a ermitteln?

Sei $2 = p_0 < p_1 < p_2 < p_3 < p_4 < \dots < p_s$ die Folge aller (!) Primzahlen $\leq \sqrt{a}$, dann:

1. Teile a durch p_0 solange es geht
2. Durch p_1 ....

⋮

Hier stellt sich folgendes Problem:

Wo bekommt man die Folge

$$p_0 < p_1 < \dots < p_s$$

her? Tafelwerke braucht

man sie gar nicht! Nehmen

wir stattdessen einfach

$$2 < 3 < 4 < 5 < 6 < 7 < \dots \leq a.$$

1. Teile a durch 2 solange es geht.
Die Vielfachen von 2 teilen danach
a sowieso nicht mehr.

2. Teile durch 3 solange es geht.
Vielfache von 3 teilen nicht mehr.

$\vdots$

a Eingabe
d = 2
while d ≤ a do
{
  while (a % d == 0) // Eliminiert // Teiler d.
  {
    d hinschreiben
    a = div(a,d).
  }
  d = d++ // Nächstes d versuchen.
}

Zur inneren Schleife: Beim

a_0, d_0 = die Werte vor der Schleife

sei für $l \geq 1$

a_l = Wert nach l-tem Durchlauf.

Läuft

sei jetzt $l \geq 1$ so daß evtl mindestens l

haben. Dann gilt folgende Schleifeninvariante:

Aufgabe ist $\underbrace{d, d, \ldots, d}_{l\text{-mal}}$

$a_l = \mathrm{Div}(a, d^l)$ und a_l ist ganzzahlig.

Anzahl Durchläufe = Anzahl Male, so daß d_0 das a_0 teilt.

Für $l = 1$ gilt die Invariante, gilt sie für l, dann für $l+1$.

Wegen der Bedingung $a \% d == 0$ gilt am Ende der inneren Schleife, daß d das a_l nicht mehr teilt (Schreibweise $d \nmid a_l$, denn das bedeutet gerade nicht $a \% d == 0$).

Damit ist für die innere Schleife klar. Bei Start mit a_0, d_0 wähle ℓ so, daß

$$\underbrace{d_0, \ldots, d_0}_{\ell\text{-mal}}$$

hintereinander sind und

$$a_\ell = a_0/(d_0)^\ell \text{ und } d \nmid a_\ell.$$

Korrektheit des Gesamtprogramms.

a_0 = der Wert des eingelesenen a

$d_0 = 2$

und sei für $k \geq 1$, so daß $\geq k$ Läufe der äußeren Schleife stattfinden.

a_k = der Wert von a nach k Läufen

d_k = der Wert von d nach k Läufen.

Sei weiterhin $m \geq 0$ (hängt von a_0 und k ab), die Schreibzahl $m = m(a_0, k)$ die Anzahl der ausgegebenen Zahlen nach k Läufen und sei

$q_1, \ldots, q_m$

die Folge dieser Zahlen.

Schleifeninvariante ist:

$c_1, \dots, c_m$ sind Primzahlen,

$a_k = a_0 / c_1 \cdot \dots \cdot c_m$ ist ganzzahlig,

a_k hat keine (Prim-)Faktoren $\leq d_k$;

$d_k = 2 + k$.

Hat das wirklich eine Invariante?

$k = 1$. Dann ist

$$c_1, \dots, c_m = 2, \dots, 2$$

$$a_1 = a_0 / c_1 \cdot \dots \cdot c_m$$ ("$c_1 \cdot \dots \cdot c_0 = 1$, das leere Produkt")

und

$$2 \nmid a_1.$$

Alles wegen der Eigenschaften der inneren Schleife! Außerdem ist $d_1 = 3 = 2 + k$.

Die Invariante gilt tatsächlich.

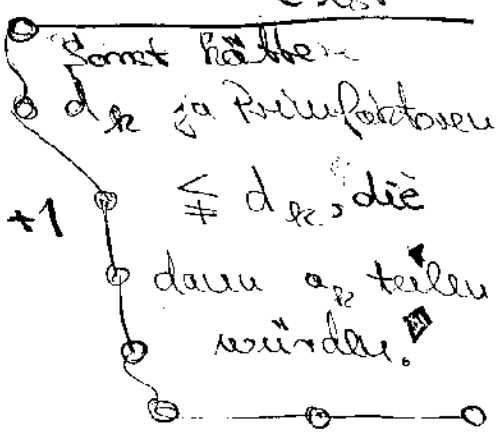
Gelte sie noch k Durchläufen
und beim $k+1^{ten}$ Lauf stop.

Also ist $d_k = 2+k$.

1. Fall d_k keine Primzahl.
Die innere Schleife wird nicht
betreten, da a_k keine (Prim-)Faktoren
$\neq d_k$ hat und d_k keine Primzahl ist
Es ist dann

$$d_{k+1} = d_k + 1 = 2 + k + 1$$

und Invariante gilt.



2. Fall d_k Primzahl

2.a. Fall $d_k \nmid a_k$ Invariante gilt.

2.b. Fall $d_k \mid a_k$, dann wird oben
bei $k=1$ und die Invariante gilt.

Die Anzahl der Durchläufe der äußeren Schleife ist endlich, weil d++ und da a_k nicht größer wird.

Ist nun die äußere Schleife im Ende, also $d_k \neq a_k$, dann gilt immer noch die Invariante. Da a_k keinen Primfaktor $\leq d_k$ hat, muß (!) $a_k = 1$ und somit ist $c_1, \ldots, c_m$ dann tatsächlich die Primfaktorenzerlegung von a_0.

Einige Beispiele:

k	a_k	d_k	Ausgabe	
0	2	2	–	äußere Schleife
1	1	3	2	Schl.

↑ ausgabe im vorletzten Lauf.

2.58

k	a_k	d_k	Ausgabe
0	3	2	–
1	2	3	–
2	1	4	3 ← Ausgabe im 2. Lauf.
			2. Durchlauf

k	a_k	d_k	Ausgabe
0	8	2	
1	1	3	2, 2, 2
			3. Durchlauf.

k	a_k	d_k	Ausgabe
0	10	2	
1	5	3	2
2	5	4	–
3	5	5	–
4	5	6	5

Das geht nicht etwas bis 11 oder ähnliches.

2.60

Ist also p eine Primzahl, dann

k	a_k	d_k	Bezeichnung
0	p	2	–
1	p	3	–
2	p	4	–
⋮	⋮	⋮	⋮
$p-2$	p	p	–
$p-1$	1	$p+1$	p

Schluß.

$d_0 = 2$

Anzahl Durchläufe der äußeren Schleife mit $k = 0, \dots, k = p-2+1$

wobei p der größte Primteiler von a_0 ist. Dann ist $d_{p-2+1} = p+1$!

und $a_{p-2+1} = 1$. Brechte auf

noch einmal die Bedingung
unserer Invariante: a_k hat keinen (!)
(Prim-)Teiler $\neq d_k$!

Kann man dieses Programm
noch verbessern?

1. Wenige d's genommen, nämlich
welche die sonstige keine Primteiler
induzieren.
2. Sind wir bei einem k mit
$d_k \cdot d_k \neq a_k$ angekommen, ist
a_k Potenzfrei und kann
als solches ausgegeben werden
(unsere Invariante gilt ja immer noch).

Als eine Verbesserung:

```
:
while  d-d ≤ a
  {
    while
      {
      :


      }
  }
```

a als letzten Primteiler ausgeben.

Am Ende ist das q_k Primzahl, da $d_k d_k \geq a$ und a keinen Primteiler $\leq d_k$ enthält nach der Invariante.

Betrachtet das folgende Problem:

Größter gemeinsamer Teiler.

Sind $g, h \in \mathbb{Z}$, dann ist

$ggT(g,h) =$ ...

$:= \max \{ b \in \mathbb{N}^+ \mid b | g \text{ und } b | h \}$.

$M :=$

Etwa $g = 10$, $h = -8$, dann

$M = \{1, 2\}$, $ggT(10, 8) = 2$.

Dagegen $g = 12$, $h = -8$, dann

$M = \{1, 2, 4\}$, $ggT(12, -8) = 4$.

Es ist $ggT(0, h) = h$, solange $h \neq 0$.

M = die Menge des Teilers von h

Ist $h \neq 0$, dann

$$ggT(0,h) = \underset{\neq 0}{|-h|}$$

Wird ist $ggT(0,0)$, dann $M = \mathbb{Z}_1$

deshalb $ggT(0,0)$ nicht definiert.

Es ist für g,h

$$ggT(-g,h) = ggT(-g,-h)$$

$$= ggT(g,-h) = ggT(g,h).$$

Es kommt also nicht aufs Vorzeichen an.

Wir können wir den $ggT(g,h)$ ermitteln? Dazu zunächst einmal die Menge der

Teiler einer Zahl (bereits gezeigt):

Sei $q \neq 1$ und Primfaktorzerlegung

$$q = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$$

dann

$$b \mid q \iff b = p_1^{f_1} p_2^{f_2} \ldots p_k^{f_k} \text{ und}$$

für alle i gilt $f_i \leq e_i$.

Habe nun auch $h \neq 1$ die Zerlegung

$$h = q_1^{a_1} \ldots q_l^{a_l} \text{ und } a_i \neq 0.$$

Seien $s_1, \ldots, s_m$, $m \leq l, k$ die in q und h vorkommenden Primteiler, dann gilt

$$M = \{ s_1^{b_1} s_2^{b_2} \ldots s_m^{b_m} \mid b_i \leq \text{Exponent von } s_i \text{ in } q \text{ und in } h \}.$$

Damit ist $ggT(z,h)$ das größte Element in $\mathbb{N}$ und das ist

Hier sieht man auch, daß alle gemeinsamen Teiler $ggT(z,h)$ teilen!

$\text{Max } M = s_1^{c_1} \cdot \ldots \cdot s_m^{c_m}$, wobei

c_i = „kleinste" Exp. von s_i in z, Exp von s_i in h.

Etwa

$ggT(30, 8) = 2^1$, denn $30 = 2 \cdot 5 \cdot 3$, $8 = 2^3$.

Hätten wir die Primfaktoren und ihre Exponenten zur Verfügung stünde sich $ggT(z,h)$ leicht ermitteln. Zur Sammlung der Primfaktoren: Datenstrukturen, etwa Mengen, Folgen.

Wir machen es zunächst einmal ohne Datenstrukturen in ??? Prüfe: Java

Tatsächlich gibt es bessere Verfahren, den $ggT(a,b)$ zu ermitteln (nach Euklid, ca. 300 v. Chr.). Zunächst gilt für $a \neq 0$

$$ggT(a,a) = ggT(a,1) = a. \quad \left\{ \text{auch } ggT(0,a) = a. \right.$$

Ferner gilt für $a \neq b$, etwa $a > b \geq 1$, daß mit

$$M_1 := \{ b \mid b \mid a \text{ und } b \mid b \}$$

$$M_2 = \{ b \mid b \mid (a-b) \text{ und } b \mid b \}$$

↳ statt dem a in M_1

gilt

$$M_1 = M_2 .$$

Denn gilt $b \in M_1$, dann

$$a = b \cdot q_1 , \text{ und } b = b \cdot q_2 , \; q_1, q_2 \geq 1$$

dann

$$a - b = b \cdot q_1 - b \cdot q_2 = b \cdot (q_1 - q_2)$$

Also: $b \mid g-h$ und $b \mid h$.

Andererseits ist für $b \in M_2$

$$g-h = b \cdot r_1 \quad \text{und} \quad h = b \cdot r_2$$

also

$$g = \overbrace{b \cdot r_1 + b \cdot r_2}^{h} = b(r_1 + r_2)$$

d.h. $b \in M_1$. Damit ist $M_1 = M_2$.

Beim Übergang von g_0, h_0, $g_0 \geq h_0 \geq 1$ auf $g_1 = g_0 - h_0$, $h_1 = h_0$ ist die Menge der gemeinsamen Teiler von g_0 und h_0 und g_1 und h_1 eine Invariante.

Unser Euklidisches Algorithmus geht dann so:

	g	h	
$10-3=$	10	3	
	7	3	
$7-3=$	4	3	
$4-3=$	1	3	$=3-1$
	1	2	$=2-1$
	1	1	
	1	0	

und $ggT(10,3) = ggT(1,1) = 1$.

Eingabe von $g, h \geq 1$, ... ≥ 1.

```
while (g != h)          // Für g ≠ h
{
   if g > h
   {
      g = g - h;
      break;            // geht zum Ende
   }                    // der Schleife.

   h = h - g;           // hier ist h > g.
}
```

Ausgabe von g oder h.

Die Sprunganweisungen:
break = geht zum Ende des aktuellen Blocks, beendet aktuelle Schleife
continue = beendet den aktuellen Lauf (nicht die ganze Schleife)
return = beendet Methode

Verifikation:

g_0 = Wert von g am Anfang.

h_0 = Wert von h am Anfang.

Sei

$M_0 = \{ b \mid b \mid g_0, b \mid h_0 \}$.

Findet für $l \geq 1$ der l'te Lauf statt, dann

$$q_l, h_l, M_l \text{ nach } l\text{'tem Lauf.}$$

Zur # Läufe: Dieser ist endlich, denn $q_0, h_0 \geq 1$ und $0 \leq q_{l+1} \leq q_l$ oder $0 \leq h_{l+1} \leq h_l$. Also $2 \leq q_{l+1} + h_{l+1} \leq q_l + h_l$. Also irgendwann ist Schluß. Beachte nicht unbedingt $|q_{l+1} - h_{l+1}| \leq |q_l - h_l|$.

Invariante: Findet ein l'ter Lauf statt, dann

$$M_0 = M_l$$

"und $q_l \geq 1$ und $h_l \geq 1$"

Inspektion des Pfützenzauberers und Überlegung oben.

Quintessenz: Ist noch ℓ Läufer das Ende erreicht, so ist $g_\ell = h_{\ell,\text{vorg. Bedingung}}$.

Dann ist

$$M_\ell = \underbrace{g_b \mid b \mid g_{\ell}}_{=h_\ell} \cdot \mathfrak{I} = M_0$$

wegen der Invarianten. Also

$$\text{Hex}\, M_\ell = g_\ell = \mathfrak{Z}\mathfrak{Z}\tau(g_0, b_0).$$

Weitere Beschleunigung nutzt Einsatz von

Div(g,h), eigentlich Mod(g,h) →
(für Programm Eukl.java)

Für $g,h \geq 1$ ist das kleinste gemeinsame Vielfache definiert durch

$$kgV(g,h) = \min\{b \mid g|b \text{ und } h|b\}.$$

Dann $kgV(g,h) \leq g \cdot h$. Sind

$p_1, \ldots, p_s$ alle Primfaktoren, die in g oder h vorkommen (mit Einsen ergänzt) wird ist

$$g = p_1^{e_1} \cdots p_s^{e_s}, \quad e_s \geq 0$$

$$h = p_1^{a_1} \cdots p_s^{a_s}, \quad a_s \geq 0,$$

dann ist

$$kgV(g,h) = p_1^{c_1} \cdots p_s^{c_s}, \text{ wobei}$$

$$q_i = \max\{e_i, a_i\}$$

Folgt mit der Eindeutigkeit der Primfaktorzerlegung von $kgV(g,k)$. (Übungsaufgabe).

Außerdem ist

$$ggT(g,k) = p_1^{d_1} \cdot \ldots \cdot p_s^{d_s}, \quad d_i \geq 0$$

und $d_i = \min\{a_i, e_i\}$. Nun sieht man leicht, daß

$$g \cdot k = ggT(g,k) \cdot kgV(g,k)$$

denn

$$\underbrace{a_i + e_i}_{\text{zu } g\cdot k} = \underbrace{\min\{a_i, e_i\}}_{\text{zum } kgV} + \underbrace{\max\{a_i, e_i\}}_{\text{zum } ggT}.$$

Programm für $kgV(g,k)$ in ggTkgV.java.

Als nächstes kleines Problem behandeln wir das Problem der

ganzzahligen Wurzel.

Für $n \in \mathbb{N}$ ist $a := \mathrm{gw}(n)$ eindeutig definiert durch

$$a^2 \leq n < (a+1)^2 .$$

In anderen Worten, ist

$$M = \{ b \in \mathbb{N} \mid b^2 \leq n \},$$

so ist $\mathrm{gw}(n) = \max M$. Also zum Beispiel ist

$$3 = \mathrm{gw}(9) = \mathrm{gw}(10) = \mathrm{gw}(11) = \dots = \mathrm{gw}(15)$$

$$4 = \mathrm{gw}(16) = \dots = \mathrm{gw}(24)$$

$$5 = \mathrm{gw}(25) = \dots = \mathrm{gw}(36) .$$

Anders geschrieben $\mathrm{gw}(n) = \lfloor \sqrt{n} \rfloor$

ganzzahliger Anteil

wobei $\sqrt{n} \in \mathbb{R}^+$ genommen wird.

Ein einfaches Programm:

n wird eingelesen

```
z = 0
while z·z ≤ n
{
    z = z+1
}
```

Ausgabe z.

n	z
0	0
	1

Ausgabe 1

n	z
3	0
	1
	2

Ausgabe 2

n	z
4	0
	1
	2
	2

Ausgabe 3

Also nicht ganz richtig.

Stattdessen

n wird eingelesen

z = 0

while $(z+1)\cdot(z+1) \leq n$

do

z = z + 1

fi

Ausgabe z

n	z
0	0

Ausgabe 0

n	z
1	0
	1

Ausgabe 1

n	z
2	0
	1

Ausgabe 1, da $2 \cdot 2 \not\leq 2$.

n	z
4	0
	1
	2

Ausgabe 2, da $\lfloor\sqrt{4}\rfloor = 2$.

Läufe: Beschränkt durch $\lfloor\sqrt{n}\rfloor$.

Alternativ $n_0 - z_\ell^2 \geq 0$ ist invariant und

$n_0 - z_{\ell+1}^2 < n_0 - z_\ell^2$.

Invariante: Nach dem ℓ'ten Lauf, sofern er stattfindet, gilt

$$z_\ell^2 \leq n_0.$$

Gilt für $\ell = 1$ und gilt bei Beh. ℓ, dann für $\ell+1$.

Quintessenz: Ist die Schleife nach dem ℓ'ten Lauf zu Ende, so ist

$(z_\ell + 1)^2 > n_0$ wg. Bedingung

und

$z_k^2 \leq n_0$ neg. Invariante.

Also ist $z_e = ggT(m)$.

Falls kein Lauf ist $n = 0$ und $z = 0$.
Damit ist die Verifikation des
Programms erbracht.

Das Programm steht in ggTEin.java

Zur Effizienzverbesserung ein
anderer Ansatz: Man könnte das
I um $d \geq 1$ hochzählen, dann
ob könnte man die $ggT(m)$
verfehlen. Deshalb arbeiten
wir mit zwei Variablen,
a und b, so daß

immer die Invariante gilt

$$a_\ell^2 \leq n \text{ und } n < b_\ell^2, \text{ und } 0 \leq a_\ell < b_\ell$$

das heißt

$$0 \leq a_\ell \leq \mathrm{gw}(n) < b_\ell .$$

Also $\mathrm{gw}(n)$ liegt in dem halboffenen Intervall $[a_\ell, b_\ell)$ und $a_\ell < b_\ell$. Beachte dass $[a_\ell, b_\ell) = \{a_\ell, \ldots, b_\ell - 1\}$.

Beachte dass:

$$|\{a_\ell, \ldots, b_\ell - 1\}| = b_\ell - a_\ell .$$

$\#$ Elemente

Am Anfang ist

$$a_0 = 0 \quad \text{und} \quad b_0 = n + 1 .$$

Ziel ist es in jeder Runde
die Länge des Intervalls,
$[a_\ell, b_\ell)$ um $d = d_\ell$ Elemente zu
verkleinern. Wie etwa in
$a_{\ell+1} = a_\ell + d$, $b_{\ell+1} = b_\ell$ oder aber
$a_{\ell+1} = a_\ell$, $b_{\ell+1} = b_\ell - d$
und die Invariante

$$\text{gew}(m) \in [a_{\ell+1}, b_{\ell+1})$$

bleibe erhalten.

Ist irgendwann $b_\ell = a_\ell + 1$,
dann ist $a_\ell = \text{gew}(m)$, denn
dann ist

$$a_\ell^2 \le m \text{ und } (b_\ell + 1)^2 \ge m.$$

wegen der Invariante.

Wie können wir ein gutes
(d.h. möglichst großes) d finden?

Beachten wir, $d=1$ geht immer,
solange $|[a_\ell, b_\ell)| = b_\ell - a_\ell \geq 2$ ist.

Ist nun das d so, daß

$$a_\ell + d \leq b_\ell - d,$$

dann gilt

$$(a_\ell + d)^2 \geq n \Rightarrow (b_\ell - d)^2 \geq n$$

(sprengen wir links zu weit,
dann geht es rechts)
und auch

$$(b_\ell - d)^2 \leq n \Rightarrow (a_\ell + d)^2 \leq n$$

(rechts zu weit, dann links)

Bei d mit $a_\ell + d \le b_\ell - d$ wird die Invariante erhalten. Das impliziert, bei d muß

$$2d \le b_\ell - a_\ell = |[a_\ell, b_\ell)|$$

eine Möglichkeit die Invariante:

$$a_{\ell+1} = a_\ell + d, \quad b_{\ell+1} = b_\ell \text{ oder}$$

$$a_{\ell+1} = a_\ell, \quad b_{\ell+1} = b_\ell - d \text{ wie.}$$

Das größte d, das es tut, ist

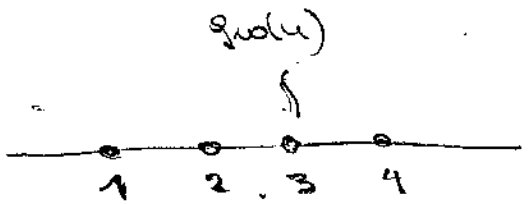
$$d_\ell = \mathrm{Div}\big(\underbrace{b_\ell - a_\ell}_{\ge 2}, 2\big) \ge 1$$

$\ge$ folgt aus $b_\ell - a_\ell \ge 2$.

Beachten wir noch einmal,

ist $a_\ell = 1$, $b_\ell = 4$, dann

$d_\ell = 1$ nach obiger Formel.



Ist etwa $\mathrm{ggT}(4) = 3$, dann

wird das Subintervall so verkleinert:

a	b	d	
1	4	1	a_2, b_2, d_2
			ein Lauf.
2	4	1	
			ein Lauf
3	4		
			Schluß

$g_{w0}(u) \in [3,4)$ also $g_{w0}(u) = \frac{3}{2}$

Ist $g_{w0}(u) = 2$, dann

a	b	d
1	4	1
2	4	1
2	$\frac{3}{2}$	

oder auch

a	b	d
1	4	1
1	3	1
2	3	

Beachte bitte:

$1 + 1 \leq g_{w0}(u)$

und

$4 - 3 \geq g_{w0}(u)$,

deshalb 2 Möglichkeiten.

Man beachte auch, daß die Wahl von $d = \left\lceil \left(\frac{b_2 - a_2}{2} \right) \right\rceil$

die Invariante nicht zwingend erhält: $gw(u) = 2$ und

a	b	d
1	4	2
1	2	

Gewichtsmaß da $2 \notin [1,2)$

a	b	d
1	4	2
3	4	

Gewicht echt reduziert.

Das gilt ebenso bei größeren Intervallen, wenn das gewählte Element $gw(m) = \left\lfloor \frac{b-a}{2} \right\rfloor$ ist. (Übungsaufgabe)

Damit haben wir folgendes Programm zu Beispiel.java. (Erstes Beispiel der binären Suche.

```
Eingabe von m;

a = 0
b = m+1

while  a+1 ≠ b                  // Vielleicht besser
{                               //  a+1 ≠ b. Beachte
   d = (b-a)/2;                 // b a+1 ≤ b ist
   if (a+d)² ≤ m                // gleichbedeutend zu
   {                            //  a ≤ b-2 oder
      a = a+d                   //  2 ≤ b-a.
   }
   if (b-d)² ≥ m                // Beachte, daß beide
   {                            // Fälle gleichzeitig
      b = b-d                   // vor kommen können
   }                            // a=1, b=6, d=2
}                               // und f(m) = b².

Ausgabe von a.                  // Hier ist a+1 = b
                                // und damit nur noch
                                // a im Suchintervall.
```

Invariante: $g_{ro}(m) \in [a_e, b_e)$.

$b_{e+1} - a'_{e+1} \leq b_e - a_e$ also

endlich viele Läufe.

Das war ja die Invariante!

Abbruch bei $a_e + 1 \geq b_e$ also

$b_e = a_e + 1$, da $g_{ro}(u) \in [a_e, b_e)$.

Dieses Beispiel zeigt deutlich

den Nutzen der Invarianten.

Nun zum dritten wagen Prinzip:

Die Reduce der Breite [illegible]

2. Ist (Breite)^2 größer als m

suche links weiter.

3. Ist (Breite)^2 kleiner als m

suche rechts weiter

ist es nicht ganz leicht, auf den

Algorithmus korrekt zu konnen.

Als letztes Beispiel die Exponentiation. Da a^b schnell zu groß wird, schauen wir uns einmal $0 \leq \mathrm{Mod}(a^b, q) < q$ an.

Zunächst einmal ist

$$\mathrm{Mod}(g \cdot h, q)$$

$$= \mathrm{Mod}(g, q) \odot_q \mathrm{Mod}(h, c),$$

wobei $g \odot_q h_c = \mathrm{Mod}(g \cdot h, q)$.

Denn ist

$$g = q_1 \cdot q + \underbrace{r_1}_{\mathrm{Mod}(g,q)}, \quad h = q_2 \cdot c + \underbrace{r_2}_{\mathrm{Mod}(h,q)}$$

dann ergibt sich

g.k =

$= (q_1 c + r_1) \cdot (q_2 \cdot c + r_2)$

$= q_1 \cdot q (q_2 c + r_2) + r_1 \cdot q_2 \cdot c + r_1 \cdot r_2$

$= q' \cdot c + r_1 \cdot r_2$

$= q' \cdot q + q'' \cdot q + r'$

Mod (g·k, q)

Vielfaches von q

Mod $(r_1 \cdot r_2, q)$

wobei $r_1 \cdot r_2 = q'' \cdot c + r'$

Dann Mod (a^b, q) nach folgendem

Prinzip: $a_1 = a \,\%\, q$,

$a_2 = (a \cdot a_0) \,\%\, q$, $a_3 = a \cdot a_1 \,\%\, q$,

$a_4 = a \cdot a_2 \,\%\, q, \ldots, a_b = a \cdot a_{b-1} \,\%\, q$.

in EinfModExp.java. Es gilt ähnliches

Dazu stellen wir b in

Binärdarstellung

$$b = \sum_{i=0}^{m-1} z_i \cdot 2^i = (z_{m-1} \dots z_0)_2.$$

Dann (wir vergessen jetzt einmal das $-$%)

$$a^b = a^{\sum z_i 2^i}$$

$$= a^{z_{n-1} 2^{n-1}} \cdot a^{z_{n-2} \cdot 2^{n-2}} \cdot \dots \cdot a^{z_1 2^1} \cdot a^{z_0 \cdot 2^0}$$

und beobachten, daß

$$a^{(2^j)} = a^{(2^{j-1}) \cdot 2}$$

$$= \left(a^{(2^{j-1})}\right)^2 = a^{(2^{j-1})} \cdot a^{(2^{j-1})}$$

ist. Wir benötigen also 2^{j-1} Faktoren a mit einer Multiplikation hin.

Vorgehensweise: Generieren

$a, a^2, a^4, a^8, \ldots, a^{2^{n-1}}$.

Wenn $b_i = 1$ multiplizieren

mit $a^{(2^i)}$ an das Teilergebnis.

Selbstverständlich alles modulo.

```
ahilf = 1;
while b ≠ 0
{
   while (b%2 == 0 )
   {
    a = (a * a) % q;
    b = b/2
   }
   ahilf = (ahilf * a) % q
   b--
}
```

Ausgabe von ahilf.

Hier die Invarianten zu finden ist nicht ganz so offensichtlich.

Innere Schleife

$a^{(0)} \geq 1,\ b^{(0)} \geq 1,\ a^{(\ell)}, b^{(\ell)}$ wie gehabt.

Invariante ist etwa (alles modulo q):

$$\left(a^{(0)}\right)^{b^{(\ell)}} = \left(a^{(0)}\right)^{b^{(0)}}$$

denn $\left(a^{(\ell+1)}\right)^{b^{(\ell+1)}} = \left(a^{(\ell)} \cdot a^{(\ell)}\right)^{b^{(\ell)}/2} = \left(a^{(\ell)}\right)^{b^{(\ell)}} = \left(a^{(0)}\right)^{b^{(0)}}$

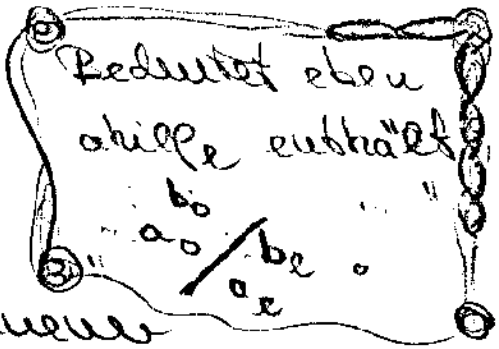
Induktion der inneren Schleife. (Ind. Vor. für ℓ)

Äußere Schleife

$a_0 \geq 1,\ b_0 \geq 1,\ a_\ell, b_\ell$, ähnlich wie gehabt, aber bezogen auf die äußere Schleife:

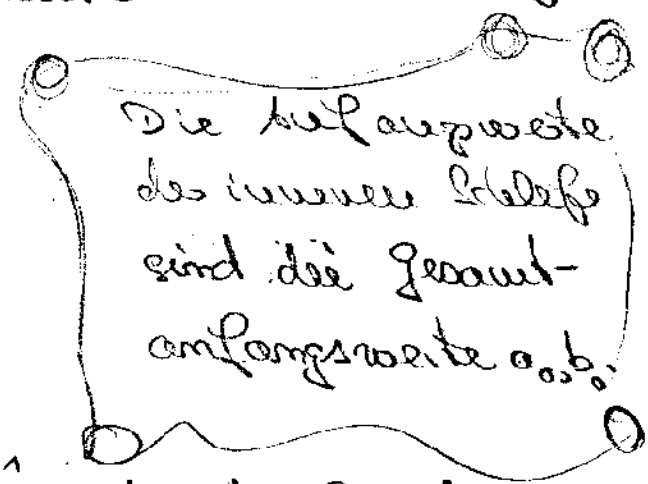
Die Invariante lautet hier:
Am Ende des ℓ'ten Laufs der
äußeren Schleife gilt

$$ahilf_\ell \cdot a_\ell^{b_\ell} = a_0^{b_0}.$$



$\ell = 1$, dann z. Nach innerer
Schleife haben wir das a_1
der äußeren Schleife und das
$b_1 + 1$ (!) der äußeren Schleife. Es
ist wegen Invariante innerer Schleife

$$a_1^{b_1+1} = a_0^{b_0}$$



Dann weiter:

$ahilf_{-1} = a_1$, da $ahilf_0 = 1$.

Dann

$$ahilf_1 \cdot \underbrace{a_1^{b_1}}_{= a_1^{b_1+1}} = a_0^{b_0}, \text{ Invariante gilt.}$$

$\Leftarrow$ wegen $a_1^{b_1+1} = a_0^{b_0}$,

Induktionsschluß: Invariante bei l,

$$\text{ahilf}_l \cdot a_l^{b_l} = a_0^{b_0}$$

Wir betreten die innere Schleife mit

$$a_l, b_l \quad (\text{statt } a_0, b_0!)$$

Am Ende der inneren Schleife haben wir bereits das a_{l+1} und das $b_{l+1} + 1$ (!).

Wegen Inv. innerer Schleife ist

$$a_l^{b_l} = a_{l+1}^{b_{l+1}+1}$$

Dann gilt

$$\text{ahilf}_{l+1} = a_{l+1}^{b_{l+1}} =$$

Wegen der Zuweisung!

$$= \text{ahilf}_l \cdot a_{l+1}^{b_{l+1}} \cdot a_{l+1}^{b_{l+1}}$$

$$= \text{ahilf}_l \cdot a_{l+1}^{b_{l+1}+1}$$

Da sind ja die Eingangswerte der inneren Schleife.

Inv. dieser Schleife:

$$a_\ell^{b_\ell} = a_{\ell+1}^{b_{\ell+1}+1}$$

$= \text{hilfe}_\ell \cdot a_\ell^{b_\ell}$

$= a_0^{b_0}$.

Quintessenz: Ist $b_\ell = 0$, das muß am Ende der Schleife sein, dann ist $\text{hilfe}_\ell = a_0^{b_0}$.

Bemerkung zur Multiplikation im ℓ'ten Komplement.

2^ℓ'tes Kompl. von a = Einschränkung von $\text{Mod}(\cdot, 2^\ell)$.

Es gilt

$$\text{Mod}(a \cdot b, 2^\ell) = \text{Mod}(a, 2^\ell) \odot_{2^\ell} \text{Mod}(b, 2^\ell).$$

Können wir überlegt auf den 2^ℓ'ten Komplementen multiplizieren, ebenso wie addieren!

3. Reelle Zahlen

$r \in \mathbb{R}$

$\Leftrightarrow$

Es gibt ein $n \geq 0$, Ziffern z_i, $0 \leq z_i \leq 10$ für $n-1 \geq i \geq -\infty$, so daß

$$r = \pm \sum_i z_i \cdot 10^i$$

$$= \pm \left(z_{n-1} z_{n-2} \dots z_0, z_{-1} z_{-2} \dots \right)_{10}$$

↳ Darstellg als Dezimalbruch.

Es ist $\sum_{i=-1}^{-\infty} z_i 10^i \leq 9 \cdot \sum_{i=1}^{-\infty} 10^i$

↑

$\Delta b - 1$.

$$= 9 \cdot \sum_{i=1}^{\infty} \left(\frac{1}{10}\right)^i \quad \text{geom. Reihe.}$$

$$= 9 \cdot \left(\frac{1}{1 - \frac{1}{10}} - 1 \right) = 1$$

Also: Alle vor dem Komma stehenden Ziffern ≤ 1.

Also ist diese unendliche Summe immer eine endliche Zahl. Schon hier sieht man: Nicht-Eindeutigkeit der Darstellung:

Gleichheit als abstrakte Zahl gesehen.

$$0{,}999\ldots999\ldots = 1 \;!$$

Wie kommt man an diese Darstellung, insbesondere für $a \lneq 1$?

Das kann auch 0 sein.

$$a \cdot 10 = z_{-1} + a_1, \quad z_{-1} \in \mathbb{N}, \; 0 \le a_1 \lneq 1$$

Tatsächlich $0 \le z_{-1} \le 9$

$$a_1 \cdot 10 = z_{-2} + a_2$$

$$a_2 \cdot 10 = z_{-3} + a_3$$

$$\vdots$$

Dann

$$a = z_{-1} \cdot \frac{1}{10} + z_{-2} \cdot \frac{1}{100} + z_{-3} \cdot \frac{1}{1000} + \ldots$$

Etwa $\frac{1}{4} \cdot 10 = 2 + \frac{1}{2}$

$$\frac{1}{2} \cdot 10 = 5 + 0. \quad \frac{1}{4} = 0{,}25.$$

Verbietet man unendliche Folgen von neun noch 9 so ist die angegebene Darstellung eindeutig. Das gilt genauso im Dualsystem:

$$x = \pm \sum_i s_i \cdot 2^i$$

Etwa $\frac{1}{2} = (0,1)_2$, $\frac{1}{4} = (0,01)_2$

$\frac{3}{4} = (0,11)_2$. Mit oben angegebener Methode berechnet man insbesondere

Beispiel:

$$\frac{1}{10} \cdot 2 = 0 + \frac{2}{10} \qquad 0,0$$

$$\frac{2}{10} \cdot 2 = 0 + \frac{4}{10} \qquad 0,00$$

3.4

$\frac{4}{10} \cdot 2 = 0 + \frac{8}{10}$ 0,000

$\frac{8}{10} \cdot 2 = 1 + \frac{6}{10}$ 0,0001

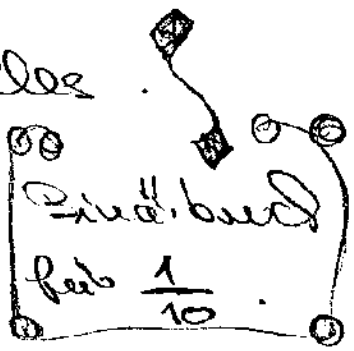
$\frac{6}{10} \cdot 2 = 1 + \frac{2}{10}$ 0,000 11

$\frac{2}{10} \cdot 2 = 0 + \frac{4}{10}$ 0,000 11 0011

0011 ...

Ab hier wiederholt sich alles.

Es ist zu sehen



$0,\overline{0011}$

$= 3 \cdot \left(\frac{1}{16} + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^3 + \dots \right)$

$= 3 \cdot \sum_{i=1}^{\infty} \left(\frac{1}{16}\right)^i$

Und $\frac{1}{10} = \frac{1}{2} \cdot \frac{2}{10}$

Geometrische Reihe

$= 3 \cdot \left(\frac{1}{1 - \frac{1}{16}} - 1 \right) = 3 \cdot \left(\frac{16}{15} - 1 \right) = 3 \cdot \frac{1}{15} = \frac{2}{10}$.

Andererseits ist $\frac{1}{2} = (0,1)_2 = (0,5)_{10}$;

Also: Endlicher Binärbruch

$\Rightarrow$ endlicher Dezimalbruch.

Endlicher Dezimalbruch, dann i.a.

kein endlicher Binärbruch.

(Endsteigkeitsprobleme bei Binärbrüchen
in der Übungsaufgabe.)

Bisherige Zahlendarstellung beim
Typ int: Festkommadarstellung (fixed point).
Man stellt sich die Zahlen als
Vielfache von 1 (ähnliche Darstellung)
oder Vielfache von $\frac{1}{10}$ oder $\frac{1}{100}$, ...
von. Das Komma ist immer an der
gleichen Stelle!

Jetzt: Gleitkommadarstellung (floating point). Diese Darstellung besteht aus 3 Bestandteilen wie in

$+0{,}3725 \cdot 10^{10}$, $-0{,}001 \cdot 10^{-10}$, ...

Vorzeichen — (Dezimal-) Bruch auch Mantisse, Signifikand. — Exponent

Zur Darstellung von Zahlen im Gleitkommadarstellung hat Java die Typen float und double. Literalkonstanten (vom Typ double) sehen folgendermaßen aus:

33.505 e4 = $33{,}505 \cdot 10^4$

33.505 E10 = $33{,}505 \cdot 10^{10}$

.333 E-3 $= 0{,}333 \cdot \frac{1}{1000}$,

-.275 E 3 $= -0{,}275 \cdot 1000$, ...

Die interne Darstellung von Zahlen vom Typ **double** verwendet 64 Bits:

$x_{63} x_{62} x_{61} x_{60} \ldots\ldots \quad x_3 x_2 x_1 x_0$.

$\hat{=}$ 32 Bit Wörter.

x_{63} Vorzeichen, $x_{63} = 1$ für −

$x_{63} = 0$ für +.

$x_{62} \ldots x_{52}$ 11 Bits (!) für den Exponenten.

Der eigentliche Exponent zur Basis 2 ergibt sich also:

$(x_{62} \ldots x_{52}) - 1023$ ← wegen Exponenten $\neq 0$.

$(11 \text{---} 1)_2 = 2047 = (11111111111)_2$

11 Stellen

Das ist nicht (!) die 2's Komplement-darstellung.

Also: $-1023 \leq$ Exponent $\leq 2046 - 1023 = 1023$

Falls Exponentbits alle 1, dann Sonderfall, d.h. entweder Unendlich, oder keine Zahl (siehe unten).

Schreibweise $x_{51} \ldots x_0$ der Bruch

(Signifikand, Mantisse).

Exponent $\neq -1023$ (d.h. $(x_{62} \ldots x_{52})_2 \geq 1$)

dann normalisierte Darstellung.

Das heißt Interpretation als Binärbruch

$$(1, x_{51} x_{50} \ldots x_0)_2$$

↑

wird automatisch ergänzt, bei Exponentbits nicht alle 0, dadurch eine mehr.

3.9

Also etwa $(0,00011)_2$ wird

zu $.10\ldots0$ und $(x_{62}-x_{52})_2 = 1013$, dann

$1013 - 1023 = -4$.

Exponent $= -1023$ (d.h. $(x_{62}-x_{52})_2 = 0$)

dann denormalisiert, indem eine

0 vor das Komma kommt.

Normalisiert kleinste positive Zahl

bei $f = x_{51}\ldots x_0 \neq 0$

$$1,f \cdot 2^{\text{Exponent}-1023} \quad \geq -1021$$

$$\geq 1 \quad \text{Exponent} - 1022$$

$$= 0,1f \cdot 2^{\text{Exponent}-1022}$$

Denormalisiert bei $f = x_{51}\ldots x_0 \neq 0$

$$0,f \cdot 2^{1-1023}$$

$$= 0,f \cdot 2^{-1022}$$

float: Darstellung auf 32 Bits

$x_{31}\ x_{30}\ x_{29}\ \dots\ x_3\ x_2\ x_1\ x_0$

Vorzeichen : x_{31}

Exponent : 30 – 23 (8 Bits)

Bruch : 22 – 0 (23 Bits)

$\frac{1}{2}$ $\left(\frac{1}{2}\right)^{23}$

Bias −127 wird zum Exponenten addiert.

Spezielle Werte (bei float und double):

0: alle Bits außer Vorzeichen.

±0 sind damit existent.

Unendlich (infinity)

+ infinity, − infinity. Exponent alles 1'en, Bruch alles 0'en.

Not a number (NaN)

Keine reelle Zahl, Exponent alles 1 und Bruch nicht 0.

- Quiet NaN – quiet NaN, Vorzeichen 1
- Signalling NaN, Vorzeichen 0.

Bei float. Ab wann Genauigkeitsverlust? Bruch hat 23 Bits.

Normalerweise 24 Bits. Also jede $a \in \mathbb{N}$ mit

$$a \leq 2^{24} - 1$$

gut darstellbar

$$a + 1 = 2^{24}$$

auch, mit den Exponenten. Also

$$a + 2 = 2^{24} + 1 = 16.777.217$$

Die 1:
Exponent
$(0111\ldots11)_2 = 1023$
Bruch:
Alles Nullen.
2: Exp $(10\ldots0)_2 = 1024$
Bruch: Alles Nullen.

wird zu $16.777.216.$

Größte Exponentialanteil ist:

$$2^{127} \approx 10^{38,23},$$

Da alles '1' zu nicht vorkommt.

außerdem $2 \cdot 10^{0,23} = 3,33$.

Kleinster Exponent ist

$$2^{-127} \approx 10^{-38,23}$$

Bildet von Bruchkehr.

Auf der Position des Bruches kann 100....01 stehen. Das heißt sich also $1 \cdot \frac{1}{2^{23}} = 2^{-23} = 10^{-6,9}$.

Also können kleinste positive Zahlen bis zu

$$10^{-45}$$

dargestellt werden.

4. Die Basisbausteine von Java

Kapitel 4,5 Ratz, Scheffler, Seese

Kapitel 6.1 bis 6.3 Kaiblitz, Weber

und die offizielle Java Beschreibung

von Arnold, Gosling

http://java.sun.com/docs/books/jls/

second_edition/html/j.title.doc.html.

Syntax = Wie Objekte (hier Programme) aussehen.

Semantik = Bedeutung eines Objekts.

(Was ein Programm macht,

welche Zustandstransformation?)

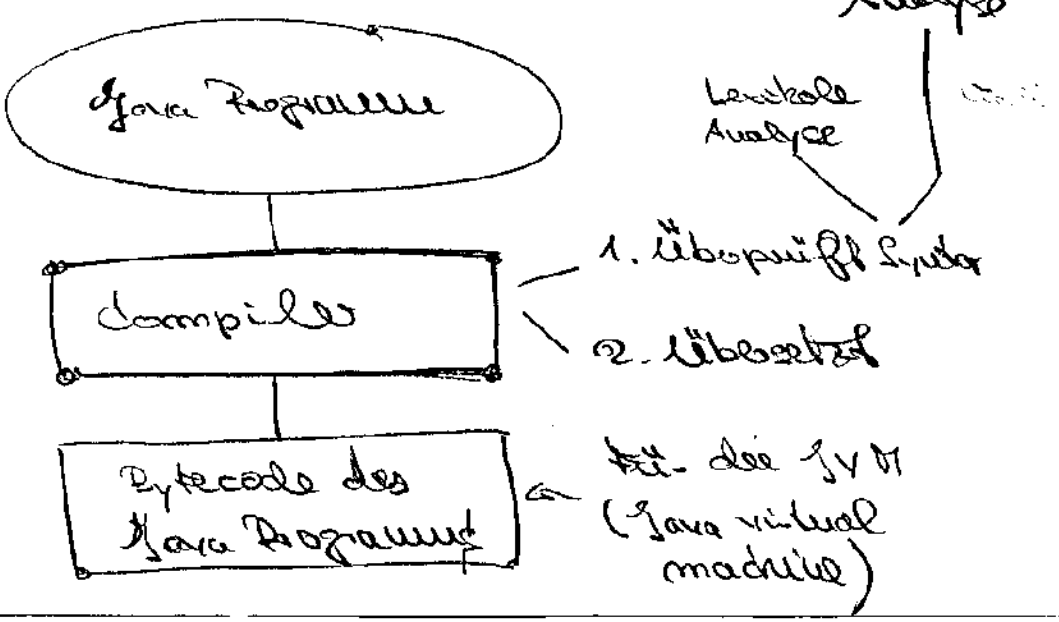
Ein Beispiel

$(+0,z_{-1}z_{-2}z_{-3}\ldots)_{10}$ ist eine syntaktisch korrekte Bezeichnung einer reellen Zahl, wenn $0 \leq z_{-i} \leq 9$.

Die Semantik, der bezeichneten Zahl, ist

$$x = \sum_{i=-1}^{-\infty} z_{-i} \cdot 10^{-i} \in \mathbb{R}.$$

Die Zahl $1 = 0,999\overline{9}$ wird auf 2 Arten bezeichnet.



Die lexikale Analyse bestimmt die Struktur der Basissymbole von (ca. Wenn der Compiler ein Programm bearbeitet, erscheint es zunächst einfach als Folge von Bits ,0,1. Das Ziel der lexikalen Analyse ist es, zu verstehen, was (= welche Bitfolgen)

- Namen oder Bezeichner
- Schlüsselworte
- Literale (und von welchem Typ)
- Trennzeichen
- Operatoren

sind. Man bezeichnet die so

erkannten Bedeutteile als

Token, aus denen das Programm dann

weiter aufgebaut ist.

In einem 1. Schritt interpretiert

der Compiler die Bitfolge als

Folge von gemäß Unicode codierten

Zeichen.

http://www.unicode.org

2 Byte = 16 Bit pro Zeichen

Insgesamt $2^{16} - 1 = 65.535$ Möglichkeiten.

Vorher ASCII (American Standard

Code for Information Interchange)

ASCII: 128 Zeichen + 33.

33 nicht druckbare Steuerzeichen

(Zeilenvorschub)

4.5

Tab. 2.2. Der ASCII Zeichensatz

oct	hex	0 0	20 10	40 20	60 30	100 40	120 50	140 60	160 70
0	0	nul	dle		0	@	P	‘	p
1	1	soh	dc1	!	1	A	Q	a	q
2	2	stx	dc2	“	2	B	R	b	r
3	3	etx	dc3	#	3	C	S	c	s
4	4	eot	dc4	$	4	D	T	d	t
5	5	enq	nak	%	5	E	U	e	u
6	6	ack	syn	&	6	F	V	f	v
7	7	bel	etb	’	7	G	W	g	w
10	8	bs	can	(	8	H	X	h	x
11	9	ht	em	)	9	I	Y	i	y
12	A	lf	sub	*	:	J	Z	j	z
13	B	vt	esc	+	;	K	[	k	{
14	C	ff	fs	,	<	L	\	l	|
15	D	cr	gs	-	=	M	]	m	}
16	E	so	rs	.	>	N	^	n	~
17	F	si	us	/	?	O	_	o	del

Codierung = Zuordnung von

Zeichen + Speicherwort

a wird in ASCII dargestellt durch

$(60+1)_{16} = (61)_{16} = (97)_{10} = \underbrace{(1100001)_2}_{7 \text{ Bits}}$

ASCII → Unicode: Die ersten 8 Positionen auf 0 setzen. Also wird a zu

$$\underbrace{00000000}_{=0}\underbrace{01100001}_{=97}$$

Oder bei ö wird zu

$$(30)_{16} = \underbrace{(011}_{=3}\underbrace{0000)}_{=0}{}_{2} = (48)_{10},$$

und in Unicode 0...0|0011|0000.

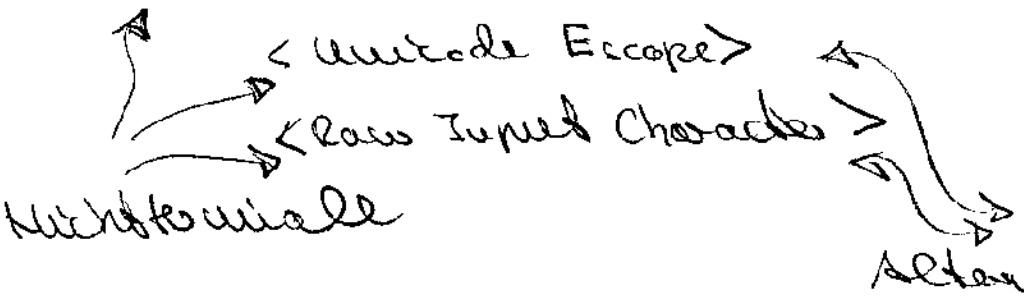
Wie kommt das Compiler an die Unicode Zeichen? Teile jeweils 16 Bits ab (oder evtl. auch 7 falls ASCII) und behandle diese Bits als ein Zeichen.

Ganz so leicht ist es nicht, wegen
der Unicode escapes. So steht
\u abcd nicht für diese Folge von
Zeichen, sondern für das Zeichen,
dessen Unicode Kodierung $(abcd)_{16}$ ist.
So ist \u 0061 das a zu lesen,
\u 0030 das 0, ... Man kann alle
Unicode Zeichen explizit angeben
durch \u $x_3x_2x_1x_0$, wobei x_i hexadezi-
male Ziffern sind.

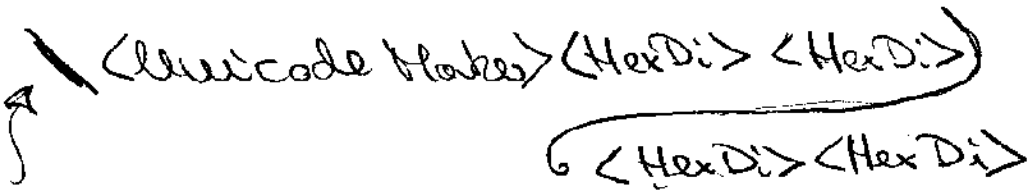
Damit sind wir also bei
folgendem Problem:

Zunächst müssen die Unicode Escapes richtig ersetzt werden.

<Unicode Input Character> :

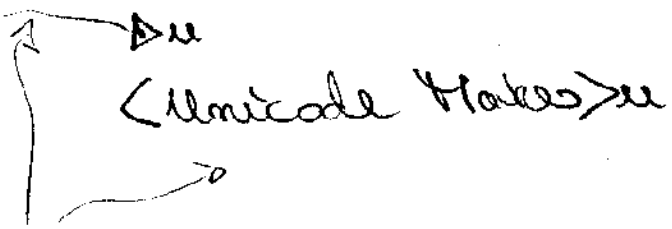


<Unicode Escape> :



Terminalzeichen

<Unicode Marker> :



Rekursive Definition. Gleichbedeutend zu: Unicode Marker ist folgende Menge:

- $u \in$ Unicode Nokes
- $\forall w \in$ Unicode Nokes, dann auch $wu \in$ Unicode Nokes.

Also Unicode Nokes $= \{u, uu, uuu, \ldots, uuuuu, \ldots\}$,

sofern man impliziert, daß die Menge Unicode Nokes nur mit den obigen Regeln aufgebaut ist,

Alternativ: Die kleinste Menge mit obigen Eigenschaften. Man spricht von einer induktiven Definition einer Menge. Ebenso kann $\mathbb{N}$ definiert werden

- $0 \in \mathbb{N}$
- $\forall n \in \mathbb{N}$, dann auch $n+1 \in \mathbb{N}$.

<RawTypChar> :

" Jedes Unicode Zeichen "

<HexDi> :

0
1
2
...
a
...
f
A
...
F

Hier sind auch die Steuerzeichen dabei.

Alternativ.
Und : one of
0 1 2 3 — F

Ein Backslash \ :

Einmal Unicode Escape,
einmal \ als Zeichen: \\

"Die Eingabe \\u2297 = \u2297
\ als Zeichen } Unicode Escape.
ergibt die Unicode Zeichenfolge
"\u2297 = ⊗ .

(4.11)

\\\\\\ gibt \\\. Ein Fehler ist

\uuu XYab, da nicht hexadezimal.

\u005c\u005a wird zu \u005a,

Unicode von \, binär: 0000000 1011100 (5= 12=),

was nicht als Unicode escape gelesen

wird, sondern so bleibt, wie es ist.

Nach Beobachtung des Eingabe-

ablaufs nach angegebenen Regeln

hat der Compiler eine Folge

von Unicode Zeichen. Diese

wird als nächstes mittels der

vorhandenen Zeilenendezeichen in

Zeilen eingeteilt (z.B. zwecks

Numerierung, Erkennung des Endes

von Kommentaren // ...)

<Line Term>:

LF steht für (line feed, newline)

CR " (carriage return, return)

CR LF "

Haben jetzt Eingabestrom in Zeilen getrennt.
Jetzt hat man (der Compiler) eine Folge von InputChar's gemäß Unicode und LineTerm's vorliegen.

Es ist alles bereit, die Basisbestandteile eines Java Programms, die Token, zu identifizieren.

<InputElement> :

<InputElement>
<InputElement> <InputElement>

Rekursiv <...> :

<Token Element> :

<White Space>
<Comment>
<Token>

<Token> :

<Identifier>
<Keyword>
<Literal>
<Separator>
<Operator>

<White Sp> :

SP char. (space)
FF char (form feed)
⋮

<Comment> :

<Trad. Comment>
<End of line Comment>

<End of line Comment> :

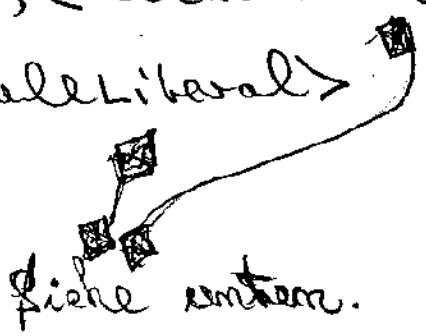
// <Char of line> or
<Line Term>

<Identifier>:

<IdentiChars> nicht

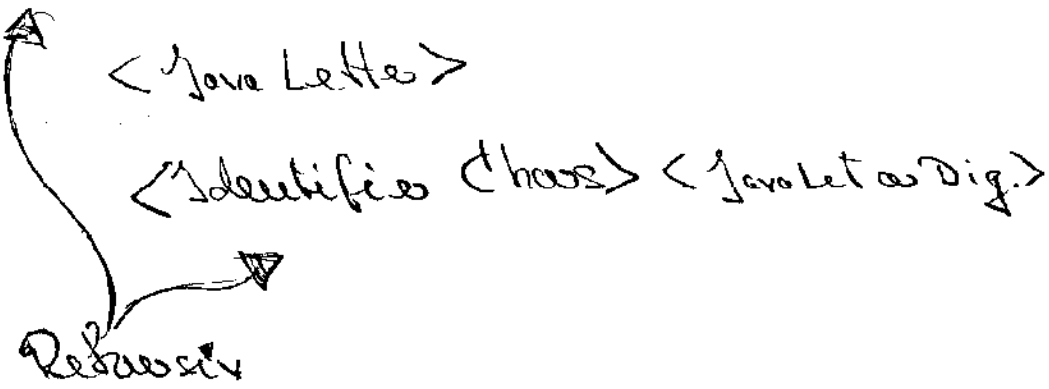
<Keyword>, <BooleanLiteral>

oder <NullLiteral>



sieht so aus.

<IdentiChars>:



<JavaLetter>:

A...

<JavaLetterOrDigit>:

...

A, ..., Z, a, ..., z,
_ (\u005F)
$ (\u0024)
auch α, β, ...

zusätzlich
0, ..., 3

4.15

<Keyword> ::=

one of

abstract	double	interface	switch
assert	else	long	synchronized
boolean	extends	native	this
break	final	new	throw
byte	finally	package	throws
case	float	private	transient
catch	for	protected	try
char	goto	public	void
class	if	return	volatile
const	implements	short	while
continue	import	static	
default	instanceof	strictfp	
do	int	super	

Die Schlüsselwörter `goto` und `const` werden momentan nicht verwendet; `assert` und `strictfp` sind in Java 2 neu hinzugekommen. Zusätzlich sind die Wörter `true`, `false` und `null` reserviert.

const, goto kommen ausschließlich nicht als Schlüsselworte vor.

true, false, null stehen nur Schlüsselworte, sind aber Literale.

<Literals>:

<IntLit>
<FloatLit>
<BoolLit>
<CharLit>
<StringLit>
<NullLit>

<FloatLit>:

<Digits>. <Digits>$_{opt}$ <ExPart>$_{opt}$ <TypeSuf>$_{opt}$

Nicht zurückgeben.

Terminalzeichen

: Digits ExPart$_{opt}$ TypeSuf$_{opt}$

:

<Digits>:

one of 0 1 2 3 4 ... 9

<Digits> one of 0 ... 9

Rekursiv

4.17

<TypeSuf>:

one of f F d D.

<BooleanLit>:

one of true false

<CharacterLit>:

' <Single char> '

' <Esc Sequence> '

Terminalsymbolen. (Ist dieses anders als Unicode Escape.)

<Single char>:

<Input Char> aber nicht ' \ ,

CR, LF nicht hier!, da <Line Term>. (Auch Unicode escapes)

Zum Beispiel sind 'a',
'\uFFFF' CharLit's.
Das Unicode escape für line feed
'\u000a' ist nicht zulässig.
Die line feed Zeichen sind ja
auch schon „weg", d.h. als „neue
Zeile" interpretiert. Stattdessen
benötigen wir character escapes:

<EscSequence> :

```
\b          // \u0008, backspace
\t
\n          // Linefeed
\r
\"          // "
\'          // '
 :
```

Terminalsymbol (4.19)

<StringLit> :

"<StringChars>opt"

<StringChars> :

<StringChar>

<StringChar> <StringChars>

Rekursiv

<StringChar> :

<InputChar> ohne " , \ (auch Unicode escapes)

<EscSeq>

Vorsicht S. 4.18.

Etwa " " (leerer String)

"Das ist" + "zu lang"

(Zusammengesetzter StringLit)

§4.20

<NullLit> :

null

<Separator> :

one of ( ) { } [ ] ; , .

<Operator> : one of

=	>	<	!	~
==	<=	>=	!=	&&
+	-	*	/	&
+=	-=	*=	/=	&=
?	:			
||	++	--		
|	^	%	<<	>>	>>>
|=	^=	%=	<<=	>>=	>>>=

Hauptschlüsselwörter
Literale
Trennzeichen
Operatoren

Nach der lexikalischen Analyse liegt dem Compiler das Programm strukturiert in abgespeicherter Form vor. Er kennt die einzelnen Lexeme und weiß, ob es sich um Schlüsselwörter, ob es sich um Identifier, ...

Siehe Aufbau des Grundbestandteile.
↓ Dazu lexikale Analyse des Compilers.

Jetzt kommt es zum schwierigeren

Teil: Aufbau des Programms. Dazu syntaktische Analyse.

Hier spielt es eine wichtige Rolle, daß Java eine stark getypte Sprache (strongly typed language) ist. Was heißt das?

In einer typfreien Sprache werden alle möglichen Wertebereiche als Mengen von Bitfolgen gesehen und gleich behandelt. Typisch typfrei sind

```
x := 5;        oder   x = 3.70;
y := true;            y = 'a';
                      y = x + 7;
z = x + y;            :
if (z = true) then y = 5
              else x = true;
:
```

Das ist also fehlerhaft.

Die erste Möglichkeit ist es, die Typen zur Laufzeit zu überprüfen. Das Programm

```
int x;
boolean y;
x := 5;
y = true;
z := x + y;
if (z = true) ...
```

wird dann zwar problemlos übersetzt, aber die Zuweisung z = x + y gibt zur Laufzeit einen Typfehler. Typüberprüfung zur Laufzeit erfordert natürlich Aufwand!

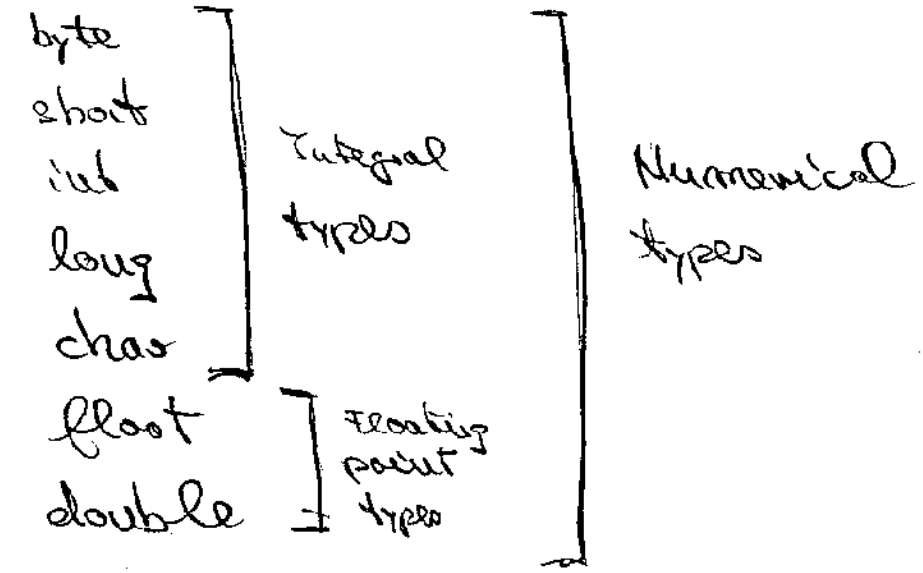
4. 2½ P

Im stark getypten Fall ist die Sprache dadurch, daß Typfehler wie x + y eben von vornherein, syntaktisch ausgeschlossen sind. Da beim reinen statischen Typprüfung ist es, Fehlerquellen schon zur Compile-Zeit aufzudecken. Jedes Ausdruck hat deshalb bereits zur Compile-Zeit einen Typ. Und es ist sichergestellt, daß dieser Typ auch bei jedem Lauf - wie abgelegen auch immer - respektiert wird. Zur eigentlichen Laufzeit ist dann keine Typüberprüfung mehr erforderlich.

Noch einmal Definition des Typen

<Type>:
    <Primitive Type>
    <Reference Type>

<Prim. type>:



Semantik:

boolean : true, false

byte : Von -128 bis 127
ausschließlich

⋮

char : Von '\u0000' bis zu
'\uffff'.
65535

Zahl mit zu den integral types.

Operatoren für Werte des integral

types: Argumente.

Ergebnis boolean:

< , <= , > , >= Semantik: Vergleiche

== , != Semantik: Gleichheitstest, Ungleichheitstest.

Ergebnis im allgemeinen int oder long:
außer bei den shift Operatoren.

+ , - Einstellige + , -.
Semantik:
+ positiv, - negativ.

* , / , % Multiplikative Operatoren.
/ rundet zur Null hin.
% kann negative Zahl ergeben.

4.26

+ , −	Zweistellige additive Operatoren
--	Dekrement um 1 Präfix oder Postfix
++	Präfix: Erst erhöhen, dann Wert geben. Postfix umgekehrt. Inkrement um 1 Präfix oder Postfix
<<	a = 5; b = a++; ergibt b = 5! x<<n x um n Bits nach links. Von rechts Nullen. Entspricht 2^n. (-1 << 1) = -2
>>	x>>n n Bits nach rechts. Von links Vorzeichenbit. Arithmetisches shift (-1 >>1) = -1 (-4>>2) = -1 (-1/2) = 0.

4.27

>>> $x >>> n$ x www n

nach rechts. Von links Nullen. Logisches shift.

$(-1 >>> 1) = 2^{31} - 1$.

Ist bei den shift Operationen das n größer als die #Bits von x, so wird um Mod(n, #Bits von x) geshiftet!

~ Bitweise Negation

&, |, ^ Bitweises Und, Oder, exklusives Oder.

? : $(x == 0)\ ?\ a\ :\ b$
Falls $x == 0$, dann a, sonst b.

(byte), (short), ... Type cast. Typumwandlung

+ String Konkatenation
"ab¢" + 3,5 = "abc3.5"

Für floating point types.

Ebenso außer, shift und ~, &, |, ^.

Operatoren für boolean sind

==, !=

!

Overloading, da auch bei integral types

Negation, etwa !b

&, ^, |

&&

Lazy evaluation

||

wie &, aber ist linker Argument false keine weitere Auswertung des rechten.

wie | aber links true, rechts keine Ausw.

Q :

+ "abc" + true = "abc true".

Beachte type cast bei boolean

aus wie

(boolean)(<Ausdruck>)

und <Ausdruck> hat Typ boolean.

Zur Verdeutlichung. Andere casts

sind auch boolean auch zulässig!

4.30

Bisher kannten wir nur lokale Variablen. Lokale Variablen beziehen sich auf Blöcke.

<Block> :

{ Blockstatements$_{opt}$ }

<Blockstatements> :

<Blockstatement>

<Blockstatements> <Blockstatement>

Rekursiv

<Blockstatement> :

<Loc. Var Decl>

<Class Decl>

<Statement>

<LocalVarDecl> :

final$_{opt}$ <Type> <Variable Declarators> ;

< VariableDeclarator >:

Beispielhaft x, y, x = 5, x = 3+7, ...

Jede LocalVarDecl. ist direkt in einem Block enthalten.

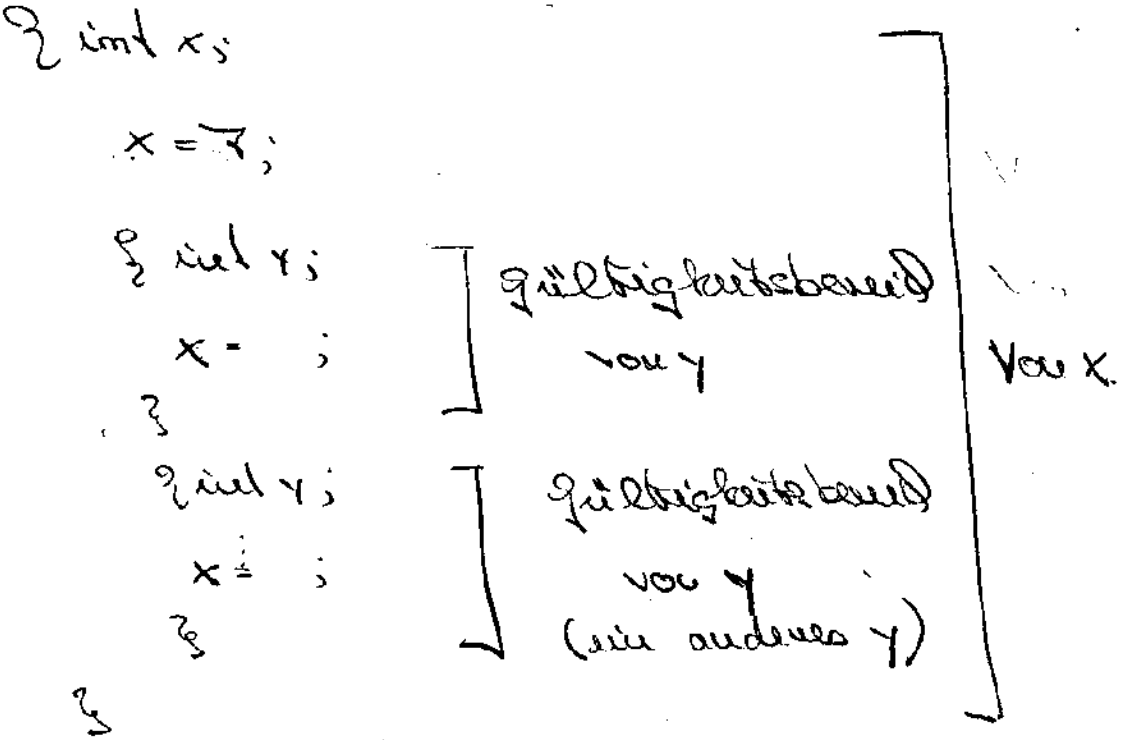
Zwei Probleme lokaler Variablen:

- Gültigkeitsbereich (scope)
- Initialisierung

```
:
{ int x;
    x = 5;
    { int x;
      :
      }
}
```

Gültigkeitsbereich des äußeren x

Obiges Programm gibt einen Syntaxfehler.
Lokale Variablen dürfen nicht
innerhalb ihres Gültigkeitsbereichs
noch einmal deklariert werden.
Zulässig ist dagegen



Vor Gebrauch muß eine lokale Variable initialisiert sein, sonst Syntaxfehler.

Zuweisung:
- Ändert Wert einer Variablen.
- Hat Wert, wie ein Ausdruck. Der zugewiesene Wert.

```
{m
   :
{ int k;
   if(x>0 && (k = IOTools.readInteger())>=0)
      System.out.println(k);
}
```

wird k als initialisiert erkannt.

Also kein Syntaxfehler. Ebenso in

```
{ int k;
  while(true)
  { k = m;
    if(k >= 5) break;
    m = 6
  }
{ System.out.println(k)
```

wird k als initialisiert erkannt.

Dagegen wird in

```
{ int k;
  int m = 5;
  if (m > 2)  k = 9;
  System.out.println(k)
}
```

k nicht als initialisiert erkannt.
Syntaxfehler. Boolsche Ausdrücke
werden nicht ausgerechnet, selbst
wenn das zur Compile-Zeit möglich
ist. Es wird nur „schematisch"
geprüft, ob eine Initialisierung
sicher stattfindet.

Alternativ wird auch

```
:
{ int k;
  if (x>y)
      k=3;
      else
          k=21;
  System.out.println(k)
}
```

dann es wird abgefragt, daß k von

Gebrauch einen Wert bekommt.

Jedoch nicht gut

```
:
{ int k;
  if (x>y)
      k=3;
  if !(x>y)          // wird x<>y.
      k=21;
  System.out.println(k)
}
```

Zusätzlich zu den oben aufgeführten Operatoren hat Java noch die Zuweisungsoperatoren (assignment operators):

= Linke Seite: Speicherplatz (Variable). Rechte Seite wird zugewiesen.

x = x x = e bedeutet

x = (Typ von x)(x * e)

/= , %= ,

+= , -= , <<= ,

>>= , >>>= , &= ,

^= , |=

Analog.

Syntaktisch korrekt ist dann

```
   ⋮
short x = 3;
x += 4.6;
   ⋮
```

und ergibt im Resultat x = 7.

Denn es ist äquivalent zu

```
   ⋮
short x = 3;
x = (short) (x + 4.6)
   ⋮
```

Das ist notwendigerweise vom Typ double!

Ausdrücke werden mit den bisher erwähnten Operatoren induktiv aufgebaut. Zu Auswertung Semantik

wird von links nach rechts
vorgegangen. Mehrere Operatoren
gleicher Bindungskraft = linksassoziativ.

Nur Zuweisungsoperatoren assoziativ
nach rechts:

$a = b = c$ wird als $a = (b = c)$

ausgewertet. Also a bekommt den
Wert von b nachdem die Zuweisung
$b = c$ stattgefunden hat.

Eine Zuweisung ist auch ein
Ausdruck. Ihre Semantik ist 2-fach:

- Ermittle Wert (normalerweise
  des zugewiesenen Werts, also $i+1, \ldots$)
- Ändere den Wert der Variablen
  auf der linken Seite (Seiteneffekt)

21.33

Operatoren nach ihrer Bindungskraft (Präzedenz):

Tab. 6.2. Präzedenzen von Java-Operatoren

Die Reihenfolge ist von großer Präzedenz absteigend zu kleiner Präzedenz angegeben. Operatoren gleicher Stufe stehen in einer Zeile.

Postfix Operatoren	`[] .` *(params)* *expr++ expr--*
Unäre Operatoren	*++expr --expr +expr -expr* `~ !`
Erzeugung und Anpassung	`new` *(type) expr*
Multiplikative Op.	`* / %`
Additive Op.	`+ -`
Schiebe-Op.	`<< >> >>>`
Relationale Op.	`< > >= <= instanceof`
Gleichheits-Op.	`== !=`
Bitweises und log. UND	`&`
Bitweises und log. XOR	`^`
Bitweises und log. ODER	`|`
Bedingtes logisches UND	`&&`
Bedingtes logisches ODER	`||`
Bedingung	`?:`
Zuweisungs-Operator	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

Noch ein paar Beispiele:

$y = (x >= 0 \;?\; x : -x)$

setzt y auf $|x|$, den Betrag von x.

a && b ist gleichbedeutend zu $(a \;?\; b : false)$

Also:

a + b + c = (a + b) + c

b + a++ = b + (a++)

a || b ist gleichbedeutend zu (a ? true : b)

a && b && c ist gleichbedeutend zu

((a ? b : false) ? c : false) .

c wird nur ausgewertet,
wenn a und b true sind.

Man beachte, daß alle Bestandteile von Ausdrücken zur Compile Zeit einen Typ erhalten. Man könnte natürlich ganz streng + einmal nur auf 2 float Argumenten, dann auf 2 double Argumenten, ... zulassen. Der Typisierungsmechanismus erlaubt aber

Typ Aufweitung (type promotion).

Bei einfachen arithmetischen Operatoren gilt: Ergebnis hat Typ int.
byte, char, short werden zu int.

Bei zweistelligen Operatoren gilt:
Ein Operand double, anders zu double aufgeweitet. Andernfalls float, wenn einer float. Andernfalls long, wenn einer long, andernfalls beide auf int ausgeweitet. Also ist das Ergebnis immer vom Typ int - oder höher.

Auch bei Shift. Beachte: Rechnerarithmetik nur für int.

: hat den Typ

↓

1 + 1.0f : float

1 + 1.0 : double

1L + 1.0f : float

1 + 1L : long

Bei int → long
Exakten 32 Bit = 0
falls ≥ 0, sonst 1!
int → float Genauigkeit geht verloren.

Nach Deklaration byte a, b, c;

a + b : int

- b : int

a = - b Syntaxfehler

c = a + b "

Beachte aber korrekt ist

byte b = 7;

byte c = b;

Dagegen geht nicht

byte c = -b.

Interessant ist, daß

final short s = 5;

short t = -s;

short t = s + 17

wiederum gelt, da die Werte rechts zur Compile Zeit feststehen und von der Größe her passen.

Implizite Typkonversionen bei „Vergrößerung" des Wertebereichs wie (oben.)

Bei einer Verkleinerung ist eine explizite Typkonversion (type cast) erforderlich:

Einige Beispiele, syntaktisch korrekt:

```
int x = (int) 1L
char z = (char) 127
```

127, da Literal vom Typ int wird
in ein Zeichen umgewandelt.

```
byte q = (byte) (a+b)
```

Beachte: type cast
bindet stärker als +.

Type cast auf boolean gibt nicht.
Was geschieht bei einem cast ?
Bei integral types fallen die
höherwertigen Bits weg. Dabei
Vorzeichenänderung möglich.

Bei double und float gibt Rückgabe
Wert oder es entsteht Infinity.

Was passiert bei floating point types
und integral types? Problem der Größe.
Übungsaufgabe.

Einige Beispiele

(byte) 128 = -128
↓
10000000

(byte) 129 = -127

(byte) 13.5 = 13

(byte) 255 = -1
↑
11111111

(byte) 13.5 = 13

Bei gemischten Ausdrücken:
Typanpassung von innen nach außen.
Ein paar Beispiele

```
double d = 1/2
```

ergibt d = 0.0 . Zwar gibt 1/2
eine Typanpassung 0 ; dann 0.0
durch Typzuweisung.

```
0.8 + 1/2 = 0.8,
```

da 1/2 zunächst zu 0.0 wird.

```
double d = 1.0/2
```

ergibt d = 0.5 und schließlich

```
0.8 + 1.0/2 = 1.3,
```

da zunächst 2 zu double umgewandelt wird.

4.4

Schließlich noch die Anweisungen (statements) von Java.

Im Unterschied zu Zuweisungen (assignment statements)

Wir wollen nun beispielhaft von:

<Statement> :

<Block>

<Empty Statement>

<Expression Statement>

⋮

<If Then Stat>

<If Then Else Stat>

<Statement We Should Jp>

⋮

<Expr. Statement>:

<Assignment>

<Pre Increment Expr> //++i

⋮

<If Then Stat>:

if (<Expression>) <statement>

<If Then Else Stat>:

if (<Expression>)

<Statement No Shal If>

else

<statement>

Hier rekursive Vorkommen von Nichtterminalen.

keine if (<Expr>) <statement>

<Statement No Short If> :

Im Prinzip kann

if ((<Expression>)

<Statement>

Hier <Expression> vom Type boolean.

Beachte das Problem des dangling

else :

```
if (b)
    if(c)
        x = 7
    else
        x = 8
```

Hier kann die Java Anweisung helfen

```
if (b)
    if (c)
        x = 7;
    else;
else    // Bei b=false
    x = 8
```

In welchem if gehört das else ?

Java stellt sicher: Im Zweifel

zum Innern.

4.50

Das switch statement dient der Fallunterscheidung:

```
switch (c)
{
    case konst1 : { statement 1; break;}
        :
    case konstn: { statement n; break; }
    default:   { statement d;}
}
```

c ist ein Ausdruck. Statement i bei c = konst i. Das break bewirkt daß ans Ende des switch Anweisung gegangen wird. Auch ohne break syntaktisch zulässig. Dann auch die weiteren Zeilen

Zusammenfassen mehrere Fälle ist möglich:

```
switch (i)
{
   case 0 : case 1: case 2 : case 3 : { res=1; break;}
      :
   default : { res =0;}
}
```

Jetzt noch die Schleifen:

<while Stat> :

while (<Expression>) <Statement>

<Expression> vom Typ boolean.

Beachte:

if (b) while (c) if c then x = a else y = b

Wäre ein while mit Sprung?

Rekursion von oben!

<Do Statement> :

do <Statement> while <Expression>

<Expression> vom Typ boolean. Semantik:

<Statement> wird ausgeführt,

dann wird <Expr.> getestet, usf. ...

Die for Schleife. Zunächst

ein Beispiel

```
int i;
int res = 0;
for (i=1; i<10; i++)
  {
    res = res+i
  }
```

4.53

ist gleichbedeutend zu:

```
int i;
int res = 0;
i = 1;
while i < 10
{
   res = res + i;
   i++;
}
```

```
for (;;)
{
  //
}
```

ist eine Endlosschleife.

<For Statement> :

for (<For Init>$_{opt}$; <Expression>$_{opt}$; <For Update>$_{opt}$)

<Statement>

Hier Rekursion von oben.

i++, i < 10

<For Init> :

<Statement Exp. List>

<Local Var Decl>

int i = 5

Sonst: Die ganze Schleife.

<For Update>:

<Statement Exp. List>

<Statement Exp. List> :

<Statement Exp>

<Stat Exp. List> <Statement Exp.>

<Expression>$_{opt}$ vom Typ boolean.

Semantik:

- Initialisierung von <For Init>
- Teste ob <Expression>$_{opt}$ true. Nicht vorhanden ist wie true.
- Ausführung von <Statement>
- <For Update>$_{opt}$ wird gemacht
- <Expression>$_{opt}$ wird getestet, bei true <Statement> ...

Ist bei <Expression>$_{opt}$ kein Ausdruck, dann Termination nur durch eine break in der Schleife.

Hier noch das leere Statement

<EStatement>:

;

Schließlich noch Marken und Sprunganweisungen.

Jeder Anweisung kann eine Marke (label) vorgesetzt werden:

<LabeledStatement>:

<Identifier>:<Statement>

Die Marke des Statements.

Das unmarkierte break-Statement:

break;

Semantik: Bewirkt sofort die unmittelbar umgebende switch; while-, do- oder for-Anweisung. Fehler, falls diese nicht existiert.

Markiertes break:

break <Identifier>;

Bewirkt die umgebende Anweisung, die durch den Identifier markiert ist.

Die continue Anweisung ohne Marke

continue;

Beispiel; daß bei einer umgebenden Iteration zur nächsten Iteration gegangen wird.

Nächste continue Anweisung

continue <Identifier>;

entspricht, daß die nächste umgebende Iterationsanweisung mit dem Label <Ident> gewählt wird.

Schließlich noch return; bewirkt das Ende des aktuellen (Unter-) Programms.

Es folgt das Beispiel noch S. 107 Ratz, Scheffler, Seese.

5. Felder

Zunächst wird Kapitel 6.1
des Buchs von Ratz, Scheffler,
Seese durchgearbeitet.

Man kann Felder deklarieren
wie in

```
int[][][] feld;
```

oder auch (nicht im Buch)

```
int feld[][][];
int[] feld[][];
   :
```

Das Partitionsproblem.
Eingabe: Ein Feld feld und ein x
so daß x = feld[k] für ein k.

Ausgabe: Eine Umspeicherung von feld, so daß für ein $0 \leq i \leq n-1$

feld[0], feld[1], ... feld[i] $\leq x$

und

feld[i+1], ..., feld[n-1] $\geq x$.

Problem

Eingabe

$$feld = (f(0), f(1), \ldots, f(y-1))$$

$x = x = f(k)$

Ausgabe

$$feld' = (\underbrace{f'(0), f'(1), \ldots, f'(i)}_{\leq x^\circ}, \overbrace{f'(i+1), \ldots f'(n-1)}^{\geq x^\circ})$$

Einige Beispiele:

Eingabe: $(1, 2, 3, 4, 5, 6)$, $x = 6$,

Ausgabe unverändert,

$i = 4$ oder $i = 5$.

Eingabe: $(1, 2, 3, 4, 4, 5)$, $x = 4$

Unverändert, $i = 3$ oder $i = 4$.

Eingabe: $(1, 2, 3, 4, 4, 4)$, $x = 4$

Unverändert, $i = 3$, $i = 4$

oder auch $i = 5$.

Im Falle $i = 5$ ist die rechte Hälfte leer. Da $i = 0$ ist die linke Hälfte nie leer.

Das Problem ist leicht zu lösen:

1. Speichere die Elemente feld[i] $\leq x$ in einem neuen Feld. $feld_1$
2. Speichere die feld[i] $\geq x$ in den Rest von $feld_1$.
3. Speichere $feld_1$ auf feld um.

Ungünstig, da doppelter Speicherplatz und feld zunächst einmal 2-mal gelesen wird (feld. laufe kann sehr groß sein > 1.000.000 zum Beispiel). Ziel: Umordnen nur in dem array feld.

Idee des Algorithmus Partition am Beispiel:

feld = (5, 4, 3, 2, 7, 8, 1, 10, 11), x = 4

Zusätzliche Variablen i, j, deren Inhalt Adressen von feld darstellen.

(5, 4, 3, 2, 7, 8, 1, 10, 11)

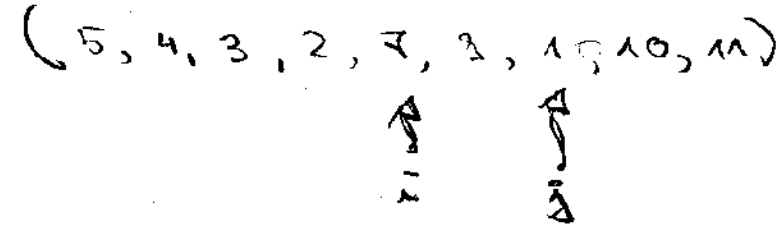
↑ i = 0 ↑ j = feld.length − 1

Gehe von links mit i bis zum ersten Element von feld, das ≥ x ist.

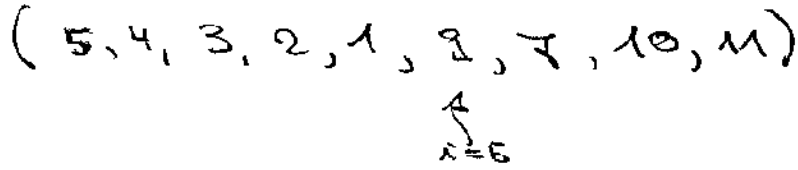
(5, 4, 3, 2, 7, 8, 1, 10, 11)

↑ i = 4 ↑ j

Von rechts mit j bis zum ersten Element $\leq x$.



Vertausche feld[i] und feld[j]



Dabei gehen wir mit i und j gleich noch einen weiter.

Ein Tauschen nicht hier, aber beachte etwa

$(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$ $x = 5$.

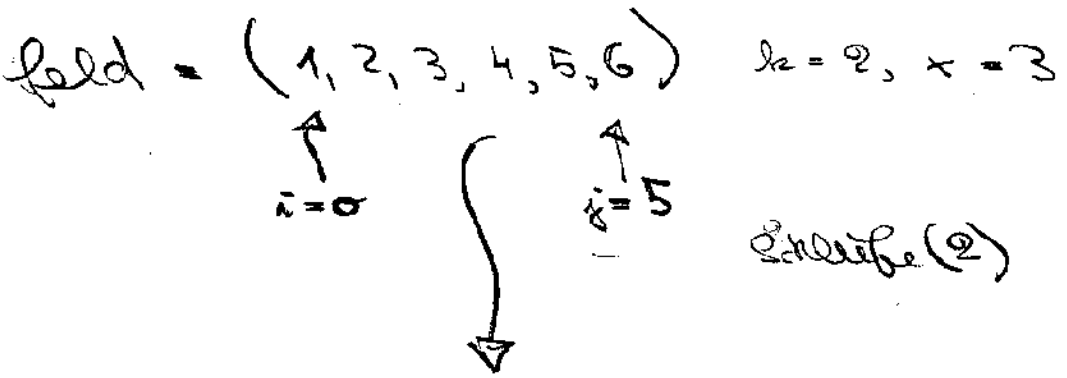
(5.7)

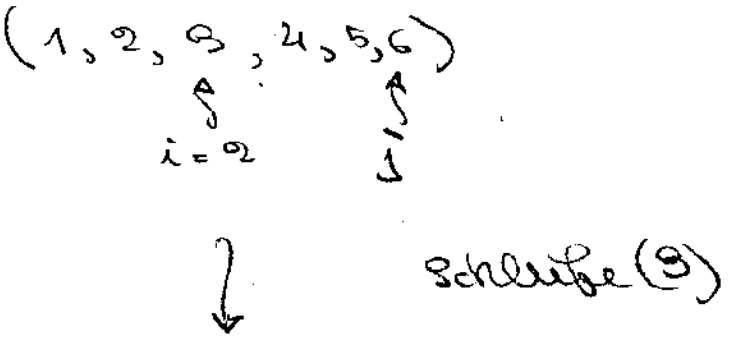
```
import Prog1Tools.IOTools;
public class Partition{
  public static void main(String[] args) {
  int [] feld;
  int i,k, j, x, n, temp;


  // Initialisierung des Feldes
  n = IOTools.readInteger("Bitte die Feldgroesse eingeben:");
  feld = new int[n];
  System.out.println();
  for (i=0; i<feld.length; i++)
    feld[i] = IOTools.readInteger("Bitte feld[" + i + "] eingeben: ");
  System.out.println();
  //
  //
  //
  k = IOTools.readInteger("Das k zur Partitionierung eingeben: ");
  if (k < 0 | k > feld.length - 1) {
     System.out.println("Das k = " + k + " liegt nicht im fraglichen Bereich.");
     return;
     }
  //
  // Ausgabe des Feldes
  //
  for (i=0; i<feld.length; i++)
    System.out.print(feld[i] + "  ");
  System.out.println();
  //
  // Ausgabe des k und des feld[k]
  //
  System.out.println(k+ " ist das k und feld["+k+"] ist " + feld[k]);
  //
  // Jetzt facngt das eigentliche Partition an.
  //
  i = 0;
  j = feld.length - 1;
  x = feld[k];
  //
  // Noch einmal zur Sicherheit die Ausgabe des Pivotelements.
  //
  System.out.println("Pivotelement: " + x);
  //
  //
```

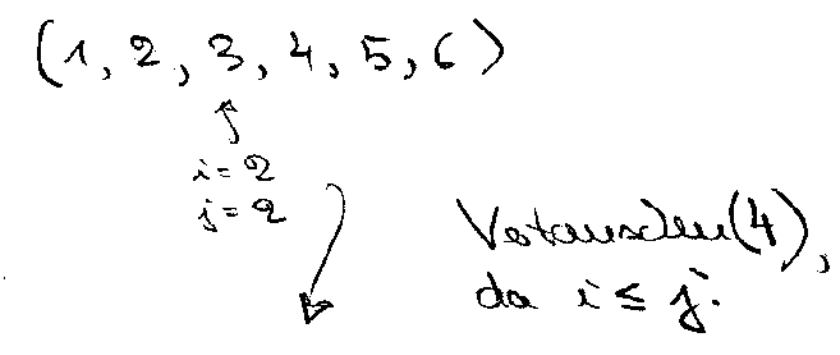
5.8

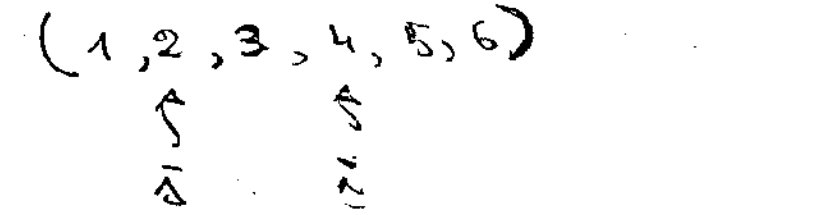
```
    do {                          // Schleife (1)
      //
      //   Schleife (2)
      //
      while (feld[i] < x)
        i++;     // Suche von links her
      //
      //   Schleife (3)
      //
      while (feld[j] > x)
        j--;     // Suche von rechts her
      //
      //    Vertauschen (4)
      //
      if (i <= j) {
        temp = feld[i];
        feld[i] = feld[j];
        feld[j] = temp;
        i++;
        j--;
      //
      //
      }

    } while (i <= j);      // Ende der Schleife bei (5)
    //
    //
    //
    System.out.println("Indexueberschneidung bei i = " + i + " und j = " + j);
    //
    // Ausgabe
    //
    for (i=0; i<feld.length; i++)
      System.out.print(feld[i] + "  ");
    System.out.println();
    return;
    }
}
```

Einige Beispiele zum Algorithmus Partition.



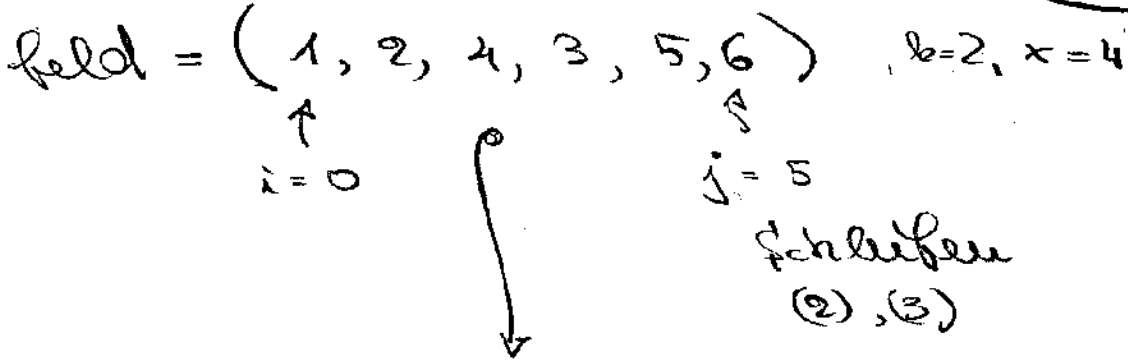


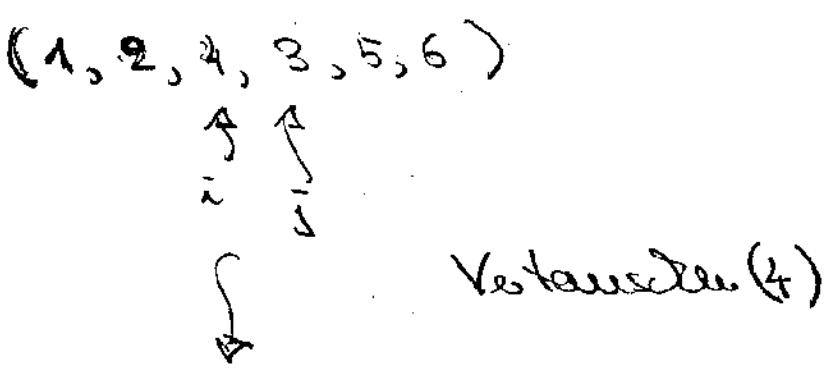


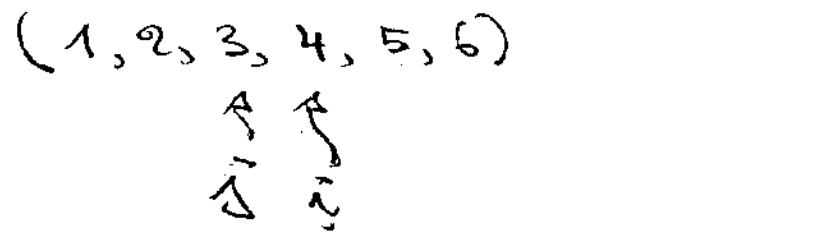


$i - j = 2 \geq 0$ da Ende.

5.10





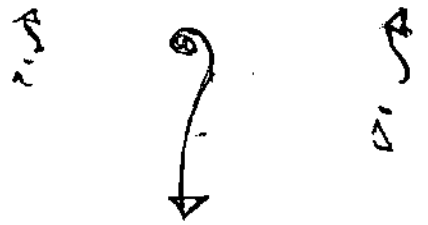


i - j = 1 ≥ 0 Ende.

Das waren 2 Fälle, in denen die Schleife nach Ausführung von (4) verlassen wird. Verlassen ohne (4) ist auch möglich.

5.11

feld = (4, 2, 3, 5, 6) $k = 2$, $x = 3$

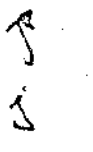


(2), (3), (4)

(3, 2, 4, 5, 6)

$j \geq i$ ist wahr, da in Schleife (1), dann in (2).

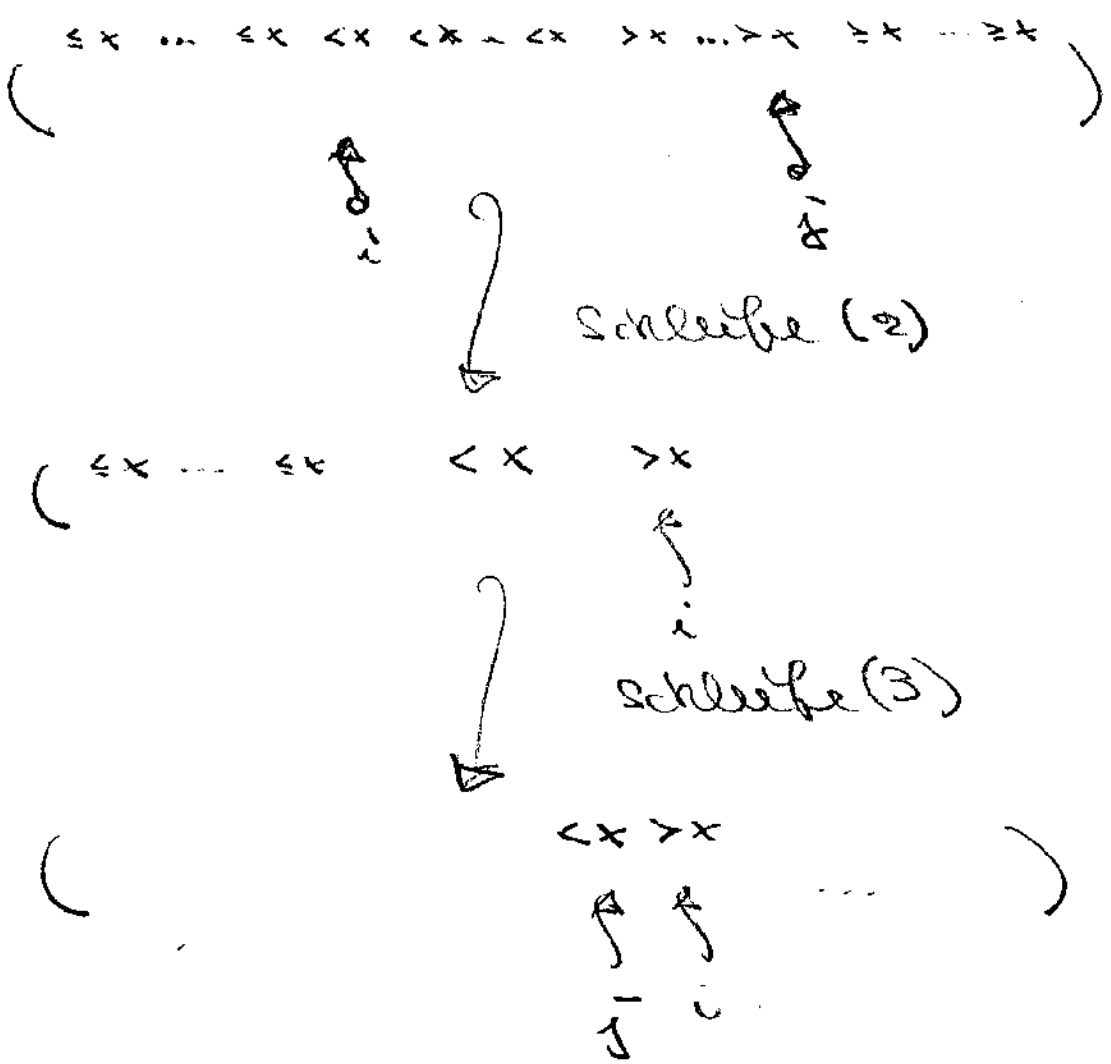
(3, 2, 4, 5, 6)



Schleife (3) wird nicht gemacht, vertauscht (4) auch nicht da Ende da

$i - j \geq 1$.

Schleife (1) endet ohne Vertauschen (4) auszuführen, wenn:



Ende, da $i - j \geq 1$, und

es findet kein Vertauschen (4) statt.

5.13

Der Beweis einer Verifikation.

Wir betrachten Schleife (1).

Für $\ell \geq 1$ sei

$$f_\ell = \text{feld}_\ell, i_\ell, j_\ell$$

wie bekannt. Für $\ell = 0$ seien es die Anfangswerte. $x = x_0$ und $k = k_0$ ist fest.

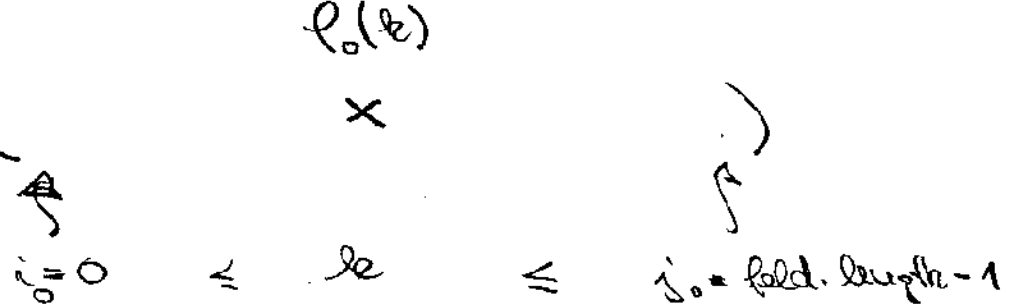
Zuerst überlegen wir einmal, daß die Schleifen (2) und (3) nicht über die Ränder von feld_ℓ hinauslaufen. Dazu folgende Invariante:

(Inv 1) Ist $i_\ell \leq j_\ell$, dann:
Es gibt $i' \geq i_\ell$ und $f_\ell[i'] \geq x$.
Es gibt $j' \leq j_\ell$ und $f_\ell[j'] \leq x$.

(Inv1) gilt für i_0, j_0 mit

$i_0 = i' \wedge j_0 = j' = k$.

Situation ist ja am Anfang



Jetzt zum Induktionsschluß. Falls (Inv1) nach dem l'ten, also vor dem $l+1$'ten Lauf von Schleife (1) mit

i_l, j_l, f_l, i' und j'.

Dabei ist $l \geq 0$ vorausgesetzt.

Wir betrachten den $\overbrace{l+1}^{\geq 1}$'ten Lauf. Also ist $i_l \leq j_l$.

Schleife (2) hält an mit einem $i_{l,1}$ wobei, wg. (Inv 1)

$$i_l \leq i_{l,2} \leq i'$$

(Gleichheit ist möglich). Dann ist $f_{t_l}(i_{l,1}) \geq x$.

Ebenso Schleife (3): wg. (Inv 1)

$$j_l \geq j_{l,3} \geq j'$$

wobei $f_{t_l}(j_{l,3}) \leq x$.

Ist $i_{l,2} - j_{l,3} \geq 1$, so sind wir fertig. (Inv 1) gilt mit

$$i' = i_{\ell,2} = i_{\ell+1}$$

$$j' = j_{\ell,2} = j_{\ell+1}$$

Ist aber $i_{\ell,2} \leq j_{\ell,2}$, so wird

$f_\ell(i_{\ell,2})$ und $f_\ell(j_{\ell,2})$ in (4) vertauscht.

Außerdem wird

$$i_{\ell+1} = i_{\ell,2} + 1$$

$$j_{\ell+1} = j_{\ell,2} - 1 .$$

Ist $i_{\ell+1} \leq j_{\ell+1}$, so gilt

(Inv 1) mit

$$i' = j_{\ell,3}$$

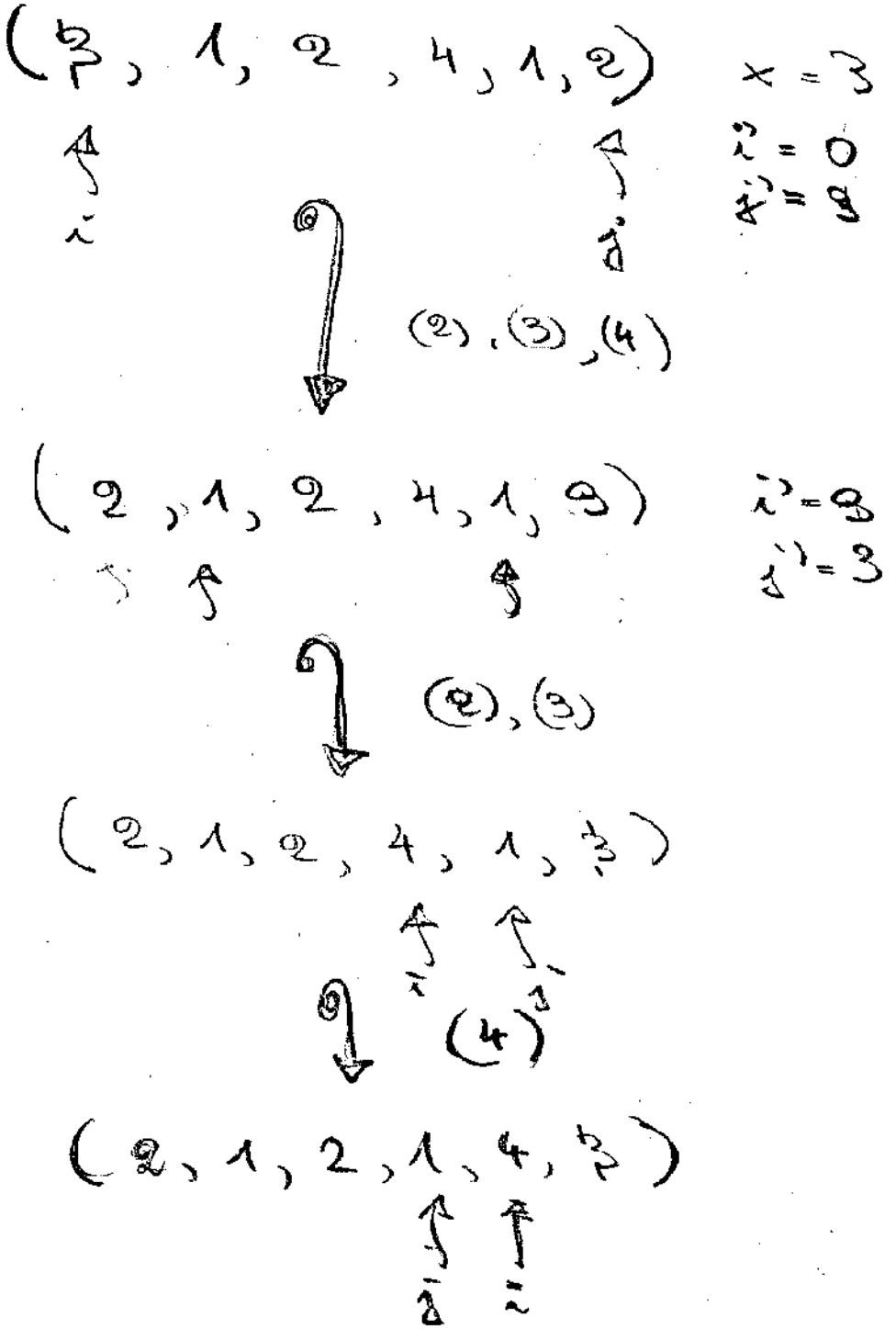
$$j' = i_{\ell,2} .$$

Passiert, wenn
$i_{\ell,2} = j_{\ell,3}$
oder aber
$i_{\ell,2} = j_{\ell,3} - 1$

Ist aber $i_{\ell+1} \geq j_{\ell+1}$, so gilt

(Inv 1) von selbst

Ein kleines Beispiel:



Die eigentliche Korrektheit von Schleife (1) folgt mit der Invariante

(Inv 2) Es ist
$f_\ell(0), \ldots, f_\ell(i_\ell - 1) \leq x$
und
$f_\ell(i_\ell + 1), \ldots$ Beachte (4), d.h. $i_\ell < j_\ell$!
$\ldots, f_\ell(\text{fld.length} - 1) \geq x$

(Inv 2) gilt für ℓ_0, i_0, j_0, da dann (formal juristisch) die leeren Folgen von Elementen von f_0 angesprochen sind.

Gelte (Inv 2) nach dem

ℓ'ten Lauf also vor dem

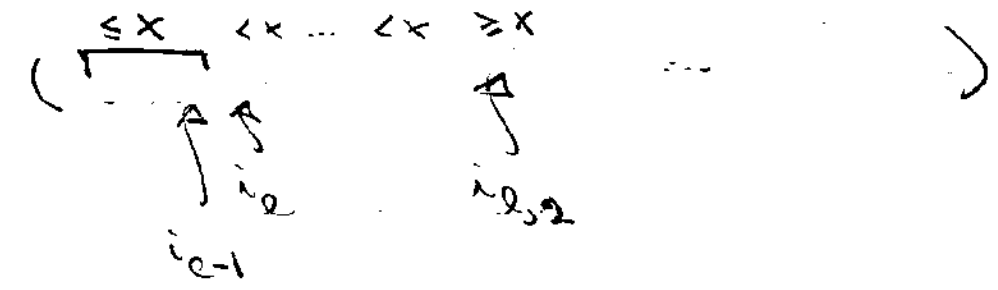
$\ell+1$'ten mit i_ℓ, j_ℓ, f_ℓ.

Schleife(2) liefert also $i_{\ell,2}$ das

nächste $i' \geq i_\ell$ mit

$$f_\ell(i') \geq x.$$

Situation



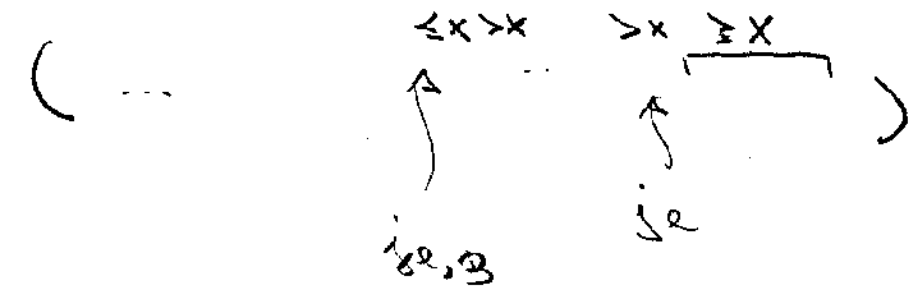
Dann ist jedenfalls

$$f_\ell(0), \ldots, f_\ell(i_{\ell,1} - 1) \leq x.$$

Schritt (3) liefert als $j_{\ell,3}$ das nächste $j' \le j_\ell$ mit

$$f_\ell(j') \le x .$$

Situation



Dann

$$f_\ell(\text{feld.länge}(\ell)-1) \dots f_\ell(j_{\ell,1}+1) \ge x .$$

Nun geht es weiter. Ist

$$i_{\ell,2} \le j_{\ell,3}$$

wird (4) ausgeführt.

Nach den Vorbereitungen haben wir f_{e+1} vorliegen. Es ist

$$f_{e+1}(0), \dots, f_{e+1}(i_{e,2}) \leq x$$

$\Leftarrow$ wg. Voraussetzung (4).

$$f_{e+1}(\text{feld.laenge}(b-1)), \dots, f_{e+1}(j_{e,3}) > x$$

Außerdem wegen Hochzählen.

$$i_{e+1} = i_{e,2} + 1$$

$$j_{e+1} = j_{e,3} - 1$$

Also gilt die Invariante (Inv 2), da eben

$$i_{e,2} = i_{e+1} - 1$$

$$j_{e,3} = j_{e+1} + 1.$$

Nun zur Quintessenz des ganzen. Schleife (1) endet, mit dem e'ten Lauf, wenn

$$i_e - j_e \geq 1$$

ist. Wegen (Inv 2) ist dann in jedem Falle

$$f_e(0), \ldots, f_e(i_e - 1) \leq x$$

$$f_e(\text{feld.length} - 1), \ldots, f_e(j_e + 1) \geq x.$$

Da $i_e > j_e$ ist, ist

$$i_e - 1 \geq j_e.$$

Also steht das Gewünschte da:

$f_e(0), \ldots, f_e(j_e) \leq x$

$f_e(\text{feld.length}-1), \ldots, f_e(j_{e+1}) \geq x.$

Trennung an j_e.

Man könnte auch sagen

$$j_{e+1} \leq i_e$$

und analog wie oben mit

i_e und $i_e - 1$ statt j_e und j_{e+1}.

Schleifbed. zu # Durchläufe der Schleifen: (2) hält immer an wg. $(\text{Inv } 1)$.
Ebenso (3). (1) hält an, da in (4) i++, j--. Wird nun (4) nicht gemacht, so ist $i > j$ und (1) hält danach an.
Es werden i und j nun hoch- und runtergezählt.

Wie sieht es genau am Ende aus? 1/4

Dazu ganz genau eine weitere Invariante für Schleife (1):

Es ist

$(\text{Inv } 3)$ $\quad i_\ell - j_\ell \leq 2.$

Beweis: Übungsaufgabe 8. Übung.

Ist $i_\ell - j_\ell = 2$, dann

ist $i_\ell - 1 = j_\ell + 1$ und wg. (Zus 2)

$$p_\ell(i_\ell - 1) = p_\ell(j_\ell + 1) = x.$$

Ist $i_\ell - j_\ell = 1$, dann $i_\ell - 1 = j_\ell$

oder $j_\ell + 1 = i_\ell$ und wir können

mit (Zus 2) nun sagen

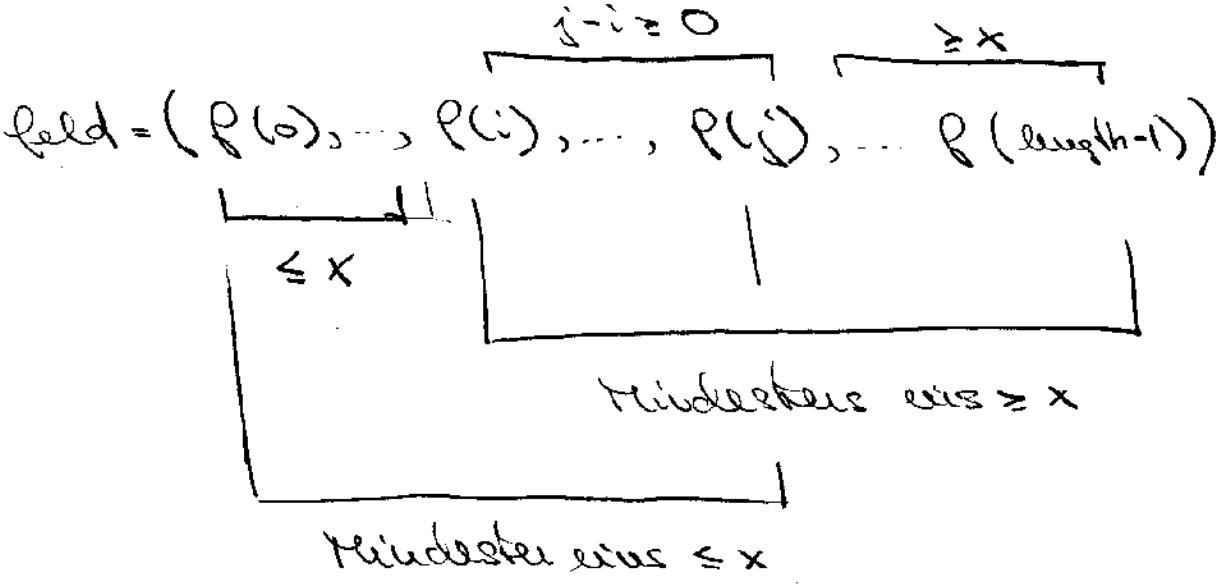
$$p_\ell(i_\ell - 1) = p_\ell(j_\ell) \leq x$$

$$p_\ell(i_\ell) = p_\ell(j_\ell + 1) \geq x.$$

Eine alternative Möglichkeit der Verifikation von Partition: Induktion über die Größe des Restes, angegeben durch $j_\ell - i_\ell$.

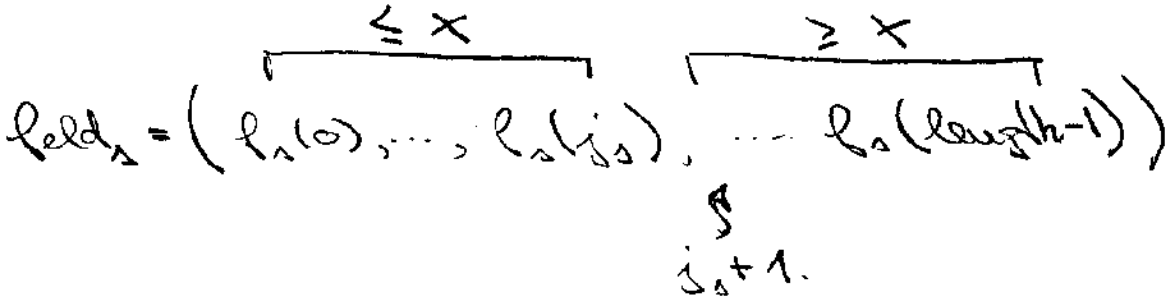
Dazu zeigen wir Behauptung von (1)

mit einer Situation der Art



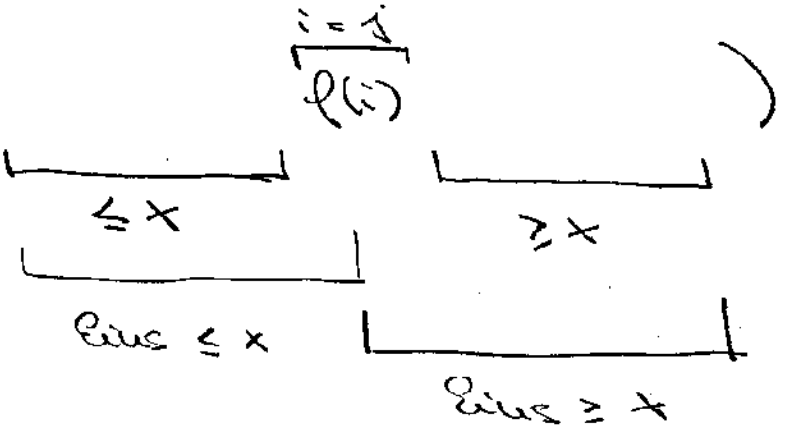
so hält (1) für ein $s \geq 1$ nach s

Läufen an mit



Ind. über $j - i \geq 0$. Ind.-Anf.

Folgen für alle (!)



gilt die Behauptung: Also,

betreten (1).

1. Fall $f(i) \neq x$

(2) hält mit $i+1$ (, fehlerfrei da mindestens eins $\geq x$(!)).

(3) bewirkt keine Änderung an j

Da jetzt $i > j$ gilt Beh. mit $s = 1$.

2. Fall $f(i) > x$

Analog 1. Fall, i bleibt stehen,
(3) endet mit $j-1$.

3. Fall $f(i) = x$.

(2), (3) ändern i und j nicht.
Am Ende wird in (4) gezählt,
Behauptung gilt.

Nun zum Induktionsschluß: $j - i \geq 0$
Nach Ind.-Vor gilt Beh. für alle (!)
Situationen, in denen die Differenz
um mindestens (!)1 kleiner ist.

Also, betreten wieder (1). Dann (2).

1. Fall: $f(i) < x$

Im ersten Lauf von (2) wird
i um eins hochgezählt.

„kürzestes echtes $\geq x$" gibt!

Betreten wir (1) gleich mit $i+1$
statt i, greift die Sud-Va. Hier
wird (2) mit derselben Situation
betreten, die aber nach dem
ersten Lauf von (2) vorliegt.
Also folgt die Behauptung
wegen der Sud.-Var.

2. Fall $f(i) \geq x$, $f(j) > x$

Ganz analog zum ersten Fall
j wird eins runtergezählt im
ersten Lauf von (3). Ind.-Vor
ist anwendbar.

3. Fall $f(i) \geq x$, $f(j) \leq x$

(2), (3) machen nichts. (4) tauscht
und zählt i eins hoch, j eins
runter. Sei

$$i_1 = i+1 \quad , \quad j_1 = j-1 \, .$$

Ist $i_1 \not\geq j_1$, dann gilt die
Behauptung direkt. Denn dann $j = i+1$,
$j_1 = i_1 - 1$.

Ist also $i_1 \leq j_1$ dann

$$j_1 - i_1 \leq j - i .$$

und wir haben eine Situation vorliegen, daß die Ind.-Vor mit j_1, i_1 dem Feld noch dem Vertauschen anwendbar ist.

Man sieht: Diese Induktion über die Größe der noch zu bearbeitenden Struktur ist etwas kürzer als die Invarianten.

Anwendung von Partitionieren beim Problem: k'tes kleinstes Element.

Eingabe: Ein Feld feld von Zahlen und ein k mit $0 \leq k \leq$ f.length -1.

Ausgabe: Das Element von feld, das bei einer Sortierung auf die k'te Stelle kommt (bezieht sich bei k=0 auf das Minimum).

feld = (1, 1, 3, 3, 2)

k = 2, Ausgabe 2
k = 1, Ausgabe 1
k = 3, Ausgabe 3
k = 4, Ausgabe 3

(10, 20, 5)
k = 1, Ausgabe 10

$k = 0$, Minimum finden.

Einmal Feld durchgehen;
kleinstes Element merken.

$k = 1$,

kleinstes und nächstes Element
merken. 2 Elemente vergleichen.

$k = 2$

3 kleinste Elemente vergleichen.

$k = \lceil n/2 \rceil - 1$ $\quad n$ = Feld-Länge

$\lceil n/2 \rceil + 1$ Elemente vergleichen.

Alternative: Sortieren.

Find für das Problem: k-tes Element.

```
import Prog1Tools.IOTools;
public class Find{
  public static void main(String[] args) {
  int [] feld;
  int i, j, k, x, n, links, rechts, temp;

  // Initialisierung des Feldes
  n = IOTools.readInteger("Bitte Feldgroesse eingeben: ");
  feld = new int[n];
  for (i=0; i<feld.length; i++)
    feld[i]=IOTools.readInteger("Bitte a[" + i + "] eingeben:");
  System.out.println();

  // Initialisierung von k
  System.out.println("Bitte k eingeben (0 <= k < " + n + ")");
  System.out.println("0 steht dabei fuer das kleinste Element");
  System.out.println((n-1) + " steht fuer das groesste Element");
  k = IOTools.readInteger("k = ");
  System.out.println();

  // Ausgabe
  for (i=0; i<feld.length; i++)
    System.out.print(feld[i] + "  ");
  System.out.println();

  // Find
  links = 0;
  rechts = feld.length-1;
```

```
while (links < rechts) {
  i = links;
  j = rechts;
  x = feld[k];                    //Pivotelement für Partition

  // Aufteilen
  do {
    while (feld[i] < x)
      i++;     // Suche von links her
    while (feld[j] > x)
      j--;     // Suche von rechts her
    if (i <= j) {
      temp = feld[i];
      feld[i] = feld[j];
      feld[j] = temp;
      i++;
      j--;
    }
  } while (i <= j);
  if (j < k)
    links = i;
  if (k < i)
    rechts = j;
}
// Ausgabe
for (i=0; i<feld.length; i++)
  System.out.print(feld[i] + "  ");
System.out.println();
System.out.println("Das " + k + "-kleinste Element ist " + feld[k]);
return;
}
}
```

wie Partition

äußere Schleife

Ausgabe von feld[k].

Einige Beispiele.

feld = (1) , k = 0

Kein Betreten der äußeren Schleife

Ausgabe von 1.

feld = (2,1) , k = 0

(2,1) l = 0, r = 1

↓

i = 0; j = 1

↓

(1,2), i = 1, j = 0

↓ Es ist k ≤ i, k ≥ j

l = 0, r = 0

Ende und Ausgabe von 1.

feld = (1 2 3) $k' = 1$

↓

(1 2 3) $l = 0$ $r = 2$

↓ Partition wf $f(1) = 2$

(1 2 3) $i' = 0$, $i = 2$

i' i

Dann $l = 2$, $r = 0$ am Ende.

und $f(1) = 2$

Zur Verifikation ist zu zeigen, daß wenn der k'-te Lauf der letzte der äußeren Schleife ist

$feld_e = ($ $\underbrace{\quad\quad}_{\leq f_e(k)}$ $f_e(k)$ $\underbrace{\quad\quad}_{\geq f_e(k)}$ $)$.

k-te Stelle.

Als zur Verifikation. (Inv 1)
von Partition explizierch, daß
uns nur über die Zweier von
feld kaufen. Wir brauchen
öfter (Inv 2) von Partition

(Inv 2) $\quad fe[0], \ldots, fe[i-1] \leq x$ (Pivotelement)
und
$fe[j+1], \ldots fe[f.length-1] \geq x$.

Wir zeigen zunächst die
Invariante (Fin 1) der
äußeren Schleife:

$links_\ell$ $rechts_\ell$

Ist $li_\ell \neq re_\ell$, so gilt

(Find 1) $\quad li_\ell \leq k \leq re_\ell$

Also k zwischen li_ℓ und re_ℓ.

(Find1) gilt am Anfang, bei $\ell = 0$.

gelte (Find1) bei $\ell - 1$, wobei $\ell \geq 1$ ist.

Was passiert im ℓ-ten Lauf?

li_ℓ, re_ℓ werden in den if-statements am Ende gesetzt. Wegen dem Ende von Partition ist dort $i \geq j$.

Linker Index | Rechter Index

Sei nun $li_\ell \leq re_\ell$. Dann

ist nun ein if-statement ausgeführt

worden. Also ist [illegible]

$k = i$ oder $k = j$. { Nicht $k < i$, nicht $k > j$.

Ist $k = i \neq j$ dann ist mit Induktionsvor.

$$links_\ell = i \leq k \leq rechts_{\ell+1} = rechts_\ell$$

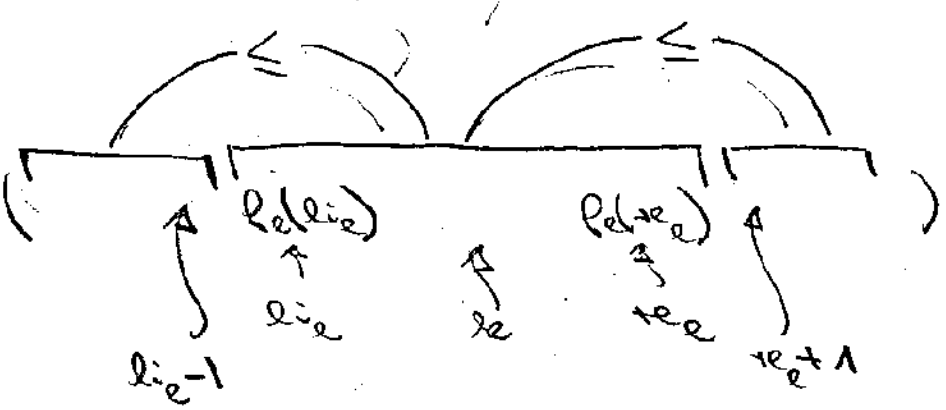
und (Find1) gilt. Ebenso für

$k < j < i$. Dann ist

$$rechts_\ell = j \geq k \geq links_{\ell+1} = links_\ell.$$

und (Find1) gilt.

Die Invariante (Find2) der äußeren Schleife:

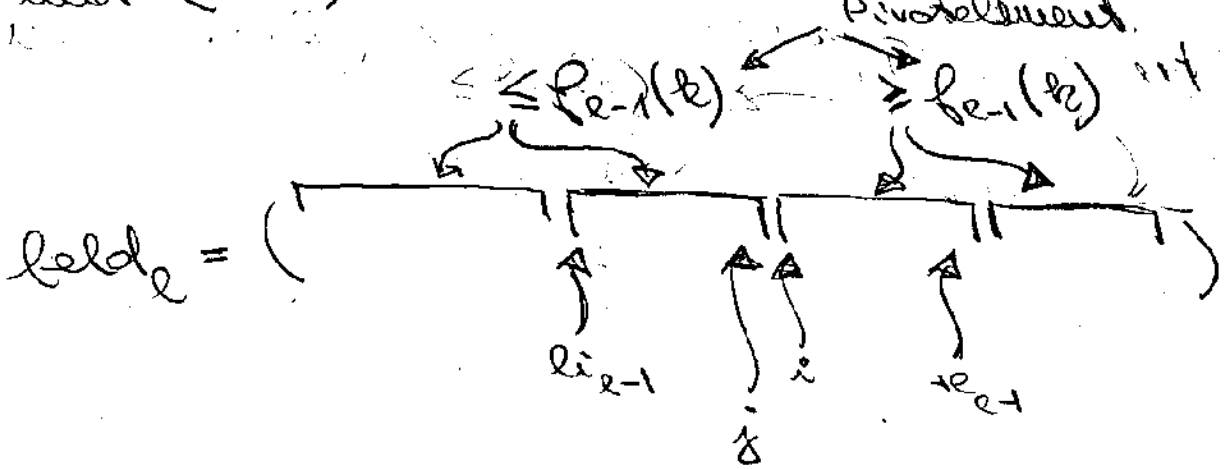
(Find2) $fild_\ell = ($ 

$)$

(Fid 2) gilt mit $\ell = 0$. Gilt (Fid 2) mit $\ell - 1$ wobei $\ell \geq 1$. Wird die äußere Schleife betreten ist mit (Fid 1)

$$li_{\ell-1} < le \leq re_{\ell-1}$$

Am Ende von Partition liegt $feld_\ell$ vor. Es ist wegen Id.-Vor. und (Inv 2) von Partition



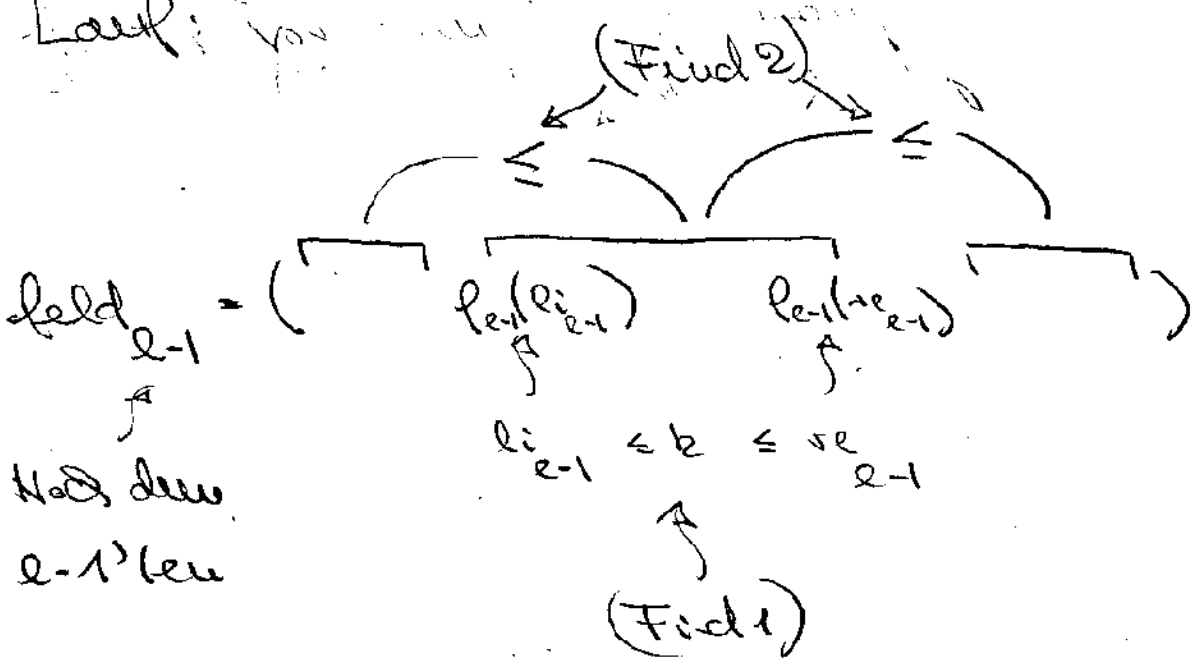
Da $links_\ell = i$ wurde kann oder $rechts_\ell = j$, gilt (Fid 2).

durch Durchläufe der
äußeren Schleife endlich,
da am Ende von Partition
$i \gneq j$ ist. Dann gilt
$i \geq j \geq k$ und rechts $= j$
oder
$j < k \leq i$ und links $= i$.

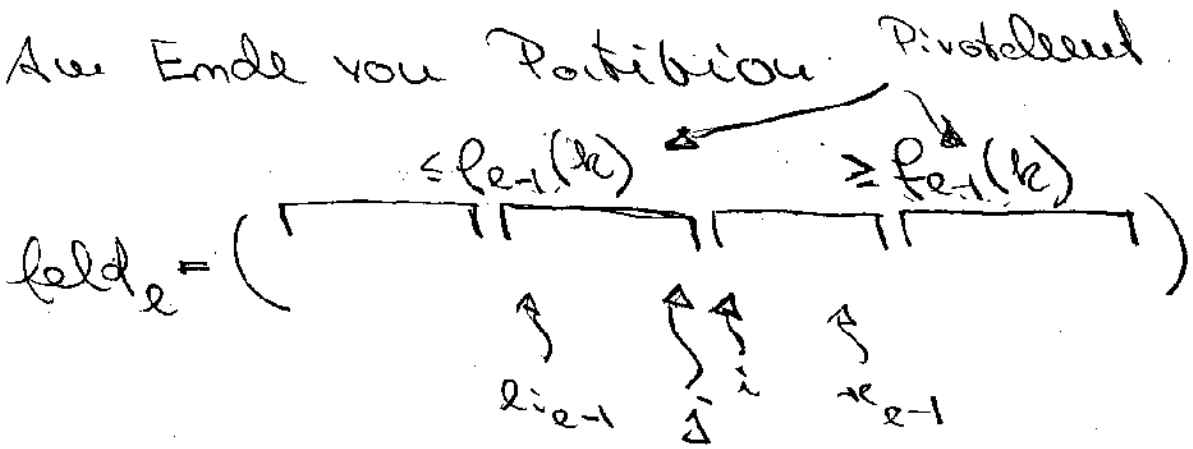
Damit wird rechts - links
echt kleiner und die Anzahl
der Läufe ist endlich.

Schließlich die Qualität des ganzen Ist der ℓ-te Lauf der letzte Lauf der äußeren Schleife.

Ist $\ell = 0$, dann ist die Verifikation erbracht. Ist $\ell > 0$, dann von ℓ'ten Lauf:



Am Ende von Partition



wegen (Inv 2) Partition und (Find 2). Da wir fertig sind wissen

$$li_\ell = re_\ell .$$

1. Fall $li_\ell = re_\ell$

Da $i > j$ wird nur ein If-Statement wirklich ausgeführt. Also

$$li_\ell = i \text{ und } re_\ell = re_{\ell-1}$$

oder

$$re_\ell = j \text{ und } li_\ell = li_{\ell-1} .$$

Im zweiten Fall

$$j < k \text{ und } k \geq i$$

also wegen $re_\ell = li_\ell$ $k = li_\ell$.

(Find 2) ergibt die Verifikation.

Im zweiten Fall ist

$$i > k \text{ und } k \leq \gamma$$

also $k = re_k$ und wieder mit (Find 2).

2. Fall: $li_k \geqq re_k$

Dann sind beide If-Statements ausgeführt worden. Dann ist

$$\gamma < k \text{ und } k < i$$

Die Verifikation von Find folgt mit (Inv 1) von Partition und (Find 2).

Die V

6. Klassen

Die Komponenten von Feldern sind immer von ein und demselben Typ. Bei Klassen braucht das nicht mehr der Fall zu sein.

Hier wird Kapitel 6.2 des Buches von Ratz, Scheffler, Seese besprochen. Klassen sind auch ein Referenzdatentyp.

(6.2)

In Aufgabe 6.8

public class Komponente

public int wert

public Komponente ref

Was macht dieses Programm?

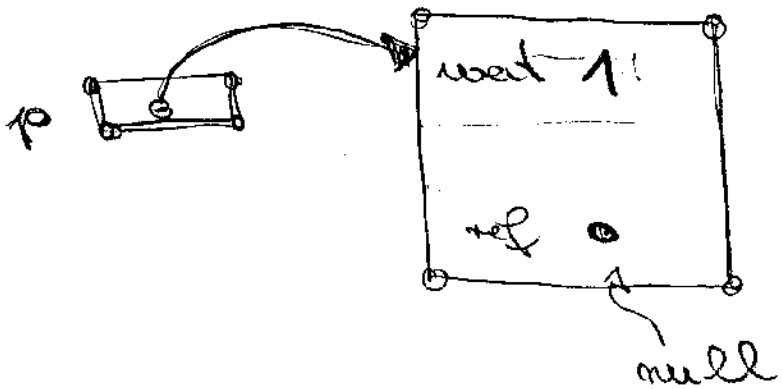
wert K0 = 1021

2 Referenzvariablen

p 

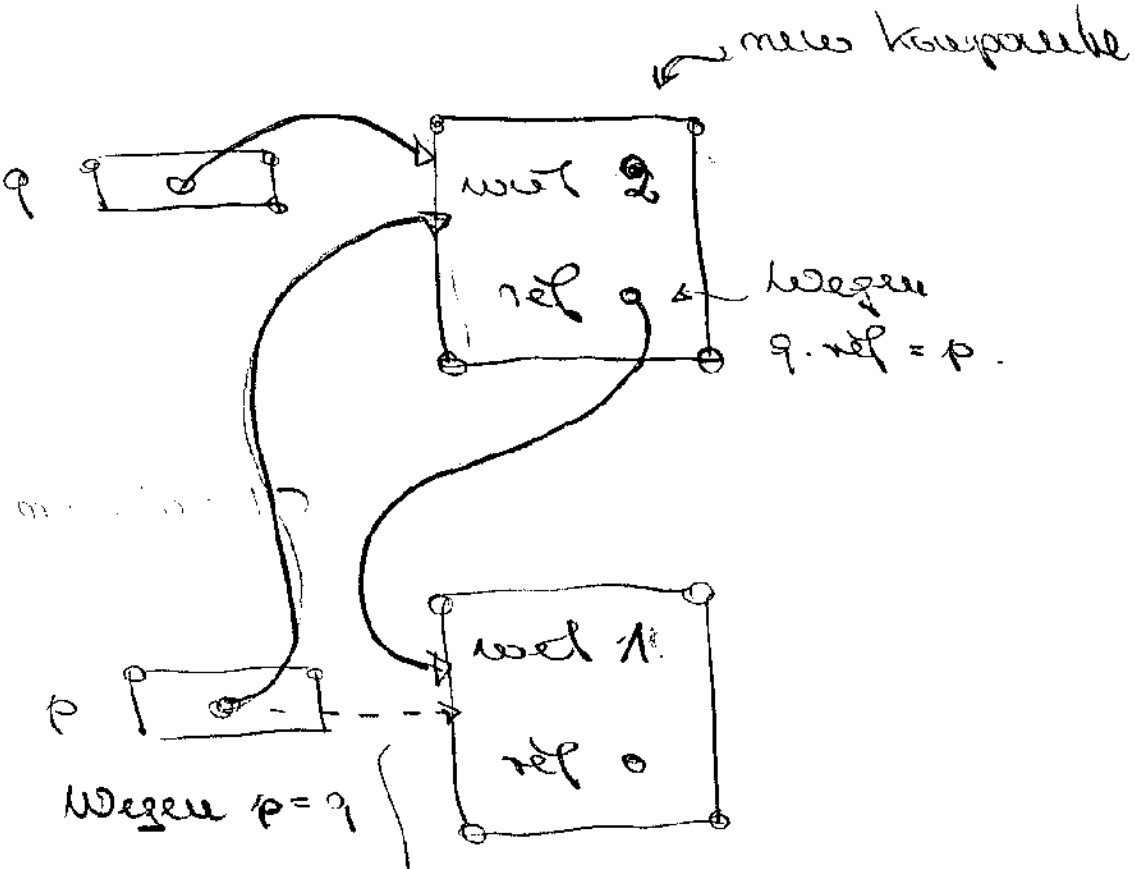
q 

11, 12, 13



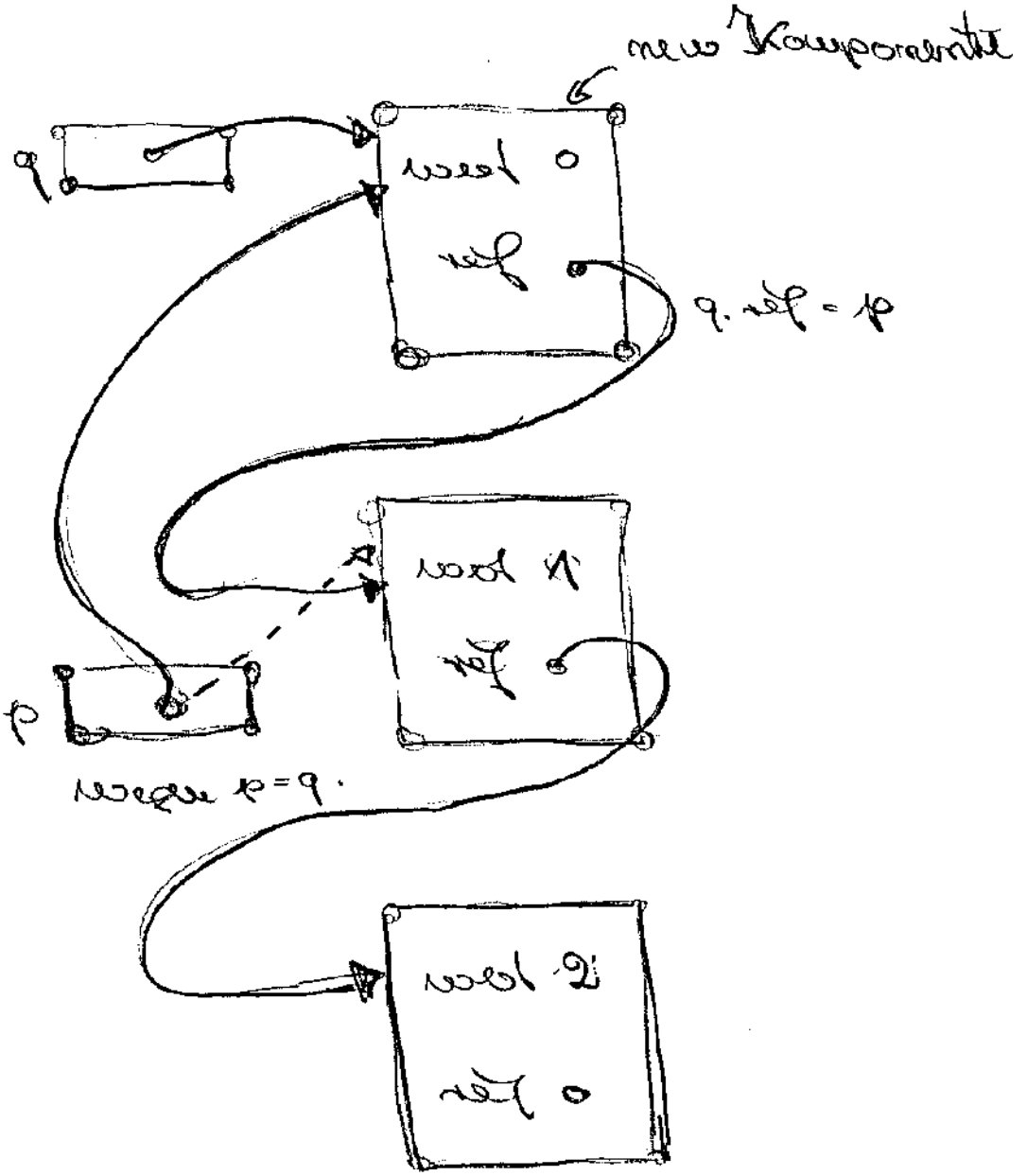
14 mete Nr = 102

15 - 20 mit i = 2



mete Nr = 10

15-90 mit i=3



p.Feld = 0

(p.Ref).Feld = 1

((p.Ref).Ref).Feld = 2

Man hat dann dynamische Datenstrukturen.

7. Mehrprogramme, Funktionen, Methoden

Kapitel 7.1 aus Ratz, Scheffler, Seese von S. 163 – 173.

Das Programm Methtest.java zum Ausprobieren.

Wir haben unsere Programme bis jetzt die Struktur:

```
public class <Programmname> {
public static int <Methodenn><Param.> {
}
public static boolean[] <Methodenn><Param> {
}        // Hier evtl Methodendeklarationen
---------
public static class <Klassenname> {
      public int x
}
public static class <Klassenname> {
      public <Klassenname> ref
}        // Hier evtl Klassendeklarationen.
---------
public static void main(String[] args) {
}        // Hauptprogramm.
```

Zum Überladen von Methodennamen:

Es ist möglich mehrere Methoden gleichen Namens zu deklarieren:

```
public static float plus(int a, int b)
{
return (float)(a+b)
}

public static float plus(float a, float b)
{... //irgendetwas
}

float c;
int d;

plus(c,d) // Die zweite Definition
          // wird genommen.
```

Implizite Typanpassungen, sofern zulässig werden gemacht. Es wird die Definition genommen, für

die die wenigsten Typumwandlungen erforderlich sind. Im Zweifelsfall Compilerfehler:

```
public static void f(int a, long b)
{ ... }

public static int f(long c, long d)

f(1,2)  // Erste Definition, da
        // eine implizite Typumwandlung
        // 2. long. Die zweite Defi-
        // nition braucht 2 Umwandlungen.

f(1L,2L) // Die zweite Definition, klar.

// Weitere Deklaration
public static void f(long a, int b)
{ }

f(1,2)  // Fehlermeldung vom Compiler
        "ambiguous".
```

Zuerst die Methodendeklaration wird gewünscht, bei der die wenigsten Typangaben erforderlich sind.
Fehlermeldung vom Compiler bei Mehrdeutigkeit.

Das folgende ist von grundsätzlicher Wichtigkeit zum Verständnis von Methoden und deren Aufrufen. Was geschieht bei der Ausführung eines solchen Methodenaufrufs ?

Das Programm beginnt das Hauptprogramm mit

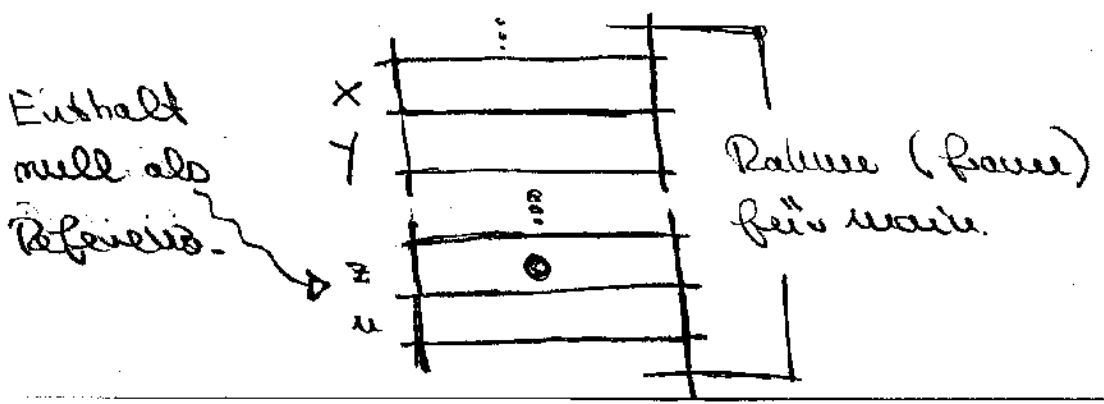
main(String[] args)

auszuführen. Die lokalen Variablen darf sind durch den Compiler bekannt.

Es wird im Hauptspeicher ein Bereich (Rahmen, Frame) angelegt, wo diese Variablen zu finden sind. Also sowas etwa

```
public class <Klassenname>
{

public static void main(String[] args){
int x;
float y;
int[] z;
double u;
}
}
```

Sei der Rahmen etwa



Kommt es jetzt zu einem Aufruf einer Methode, etwa

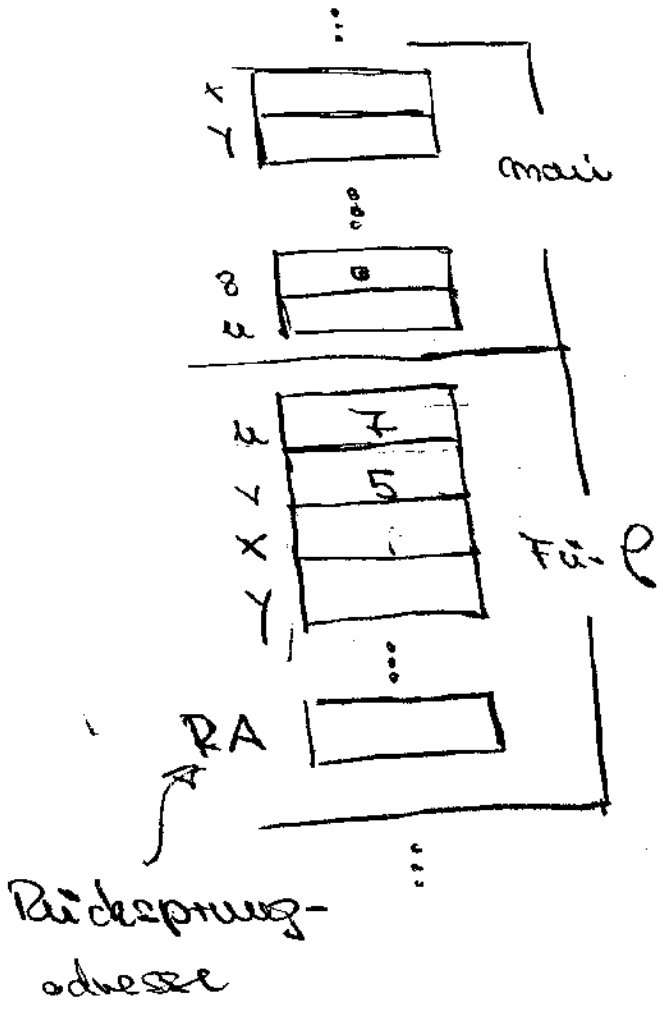
f(x, 5),

wobei x den Wert 7 hat und die Deklaration von f

```
public static int f(int u, int v)
{
  int x
  int y
}
```

den Aufruf in Frage liegt, so wird ein neuer Rahmen vor den von main gesetzt.

Unser Hauptspeicher sieht jetzt so aus:



Das Programm zu f wird in dem Rahmen von f ausgeführt.

Beim Aufruf p(x, 5) muß also folgendes geschehen:

- Ein neuer Rahmen wird angelegt. D.h. es werden Register gesetzt, in denen sich der Prozessor merkt, wo der aktuelle Rahmen liegt, ...
- In den Rahmen wird Speicherplatz für die formalen Variablen der Methode und die lokalen Variablen der Methode plaziert.
- Es werden die Werte der aktuellen Parameter (hier x = 7 und 5) ausgerechnet und bei den formalen Parametern (hier u und v) gespeichert.

Das ist im Prinzip (!) alles - fast alles.

Irgendwann ist die Auffüllung von L am Ende. Wenn geschieht dann ? Man muß im mainz Inhalt der Stelle des Aufrufs von f(x,5) weitermachen. Jetzt kann es mehrere Aufrufe von f geben. Deshalb merkt man sich im Rahmen, wo am von f(x,a) Ende weitergemacht werden soll: die

Rücksprungadresse (RA).

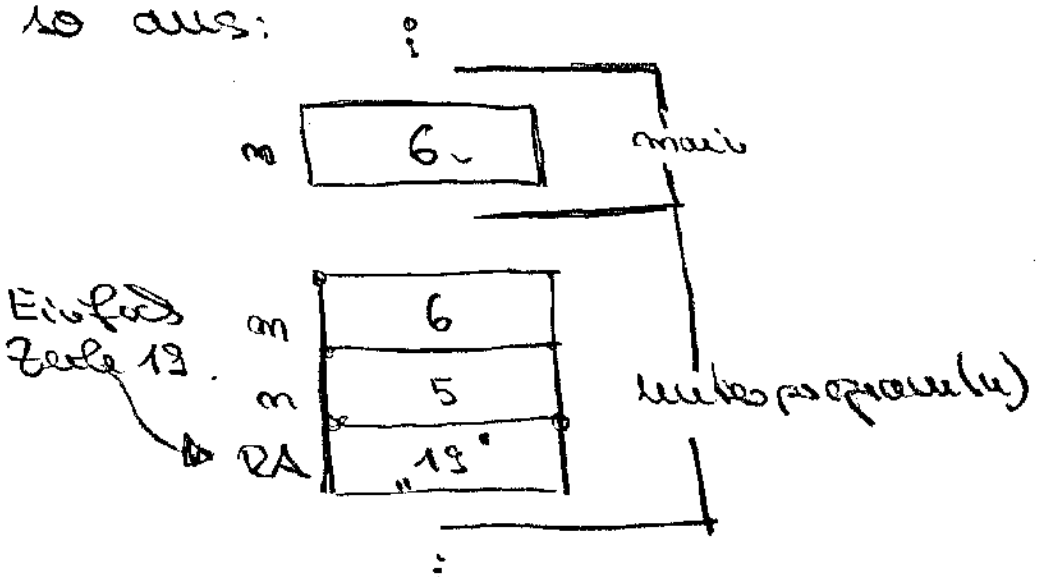
Wenn f(x,a) zu Ende geschieht folgendes:

- Eine eventuelle Ausgabe von P(x,5) wird übergeben (d.h. in den Rahmen von main eingetragen).
- Der Prozessor merkt sich RA in einem Register.
- Der Rahmen von P wird

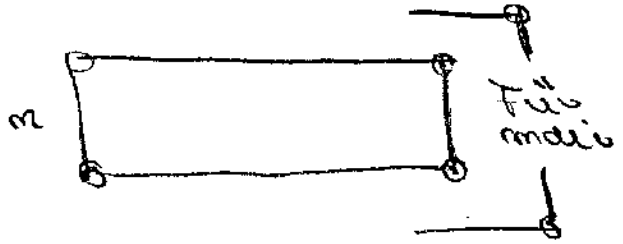
Bei dem Programm Aufruf Test.java auf S. 169 steht die Hauptspeicher nach Ausführung des Beginns des Aufrufs

unterprogramm(n)

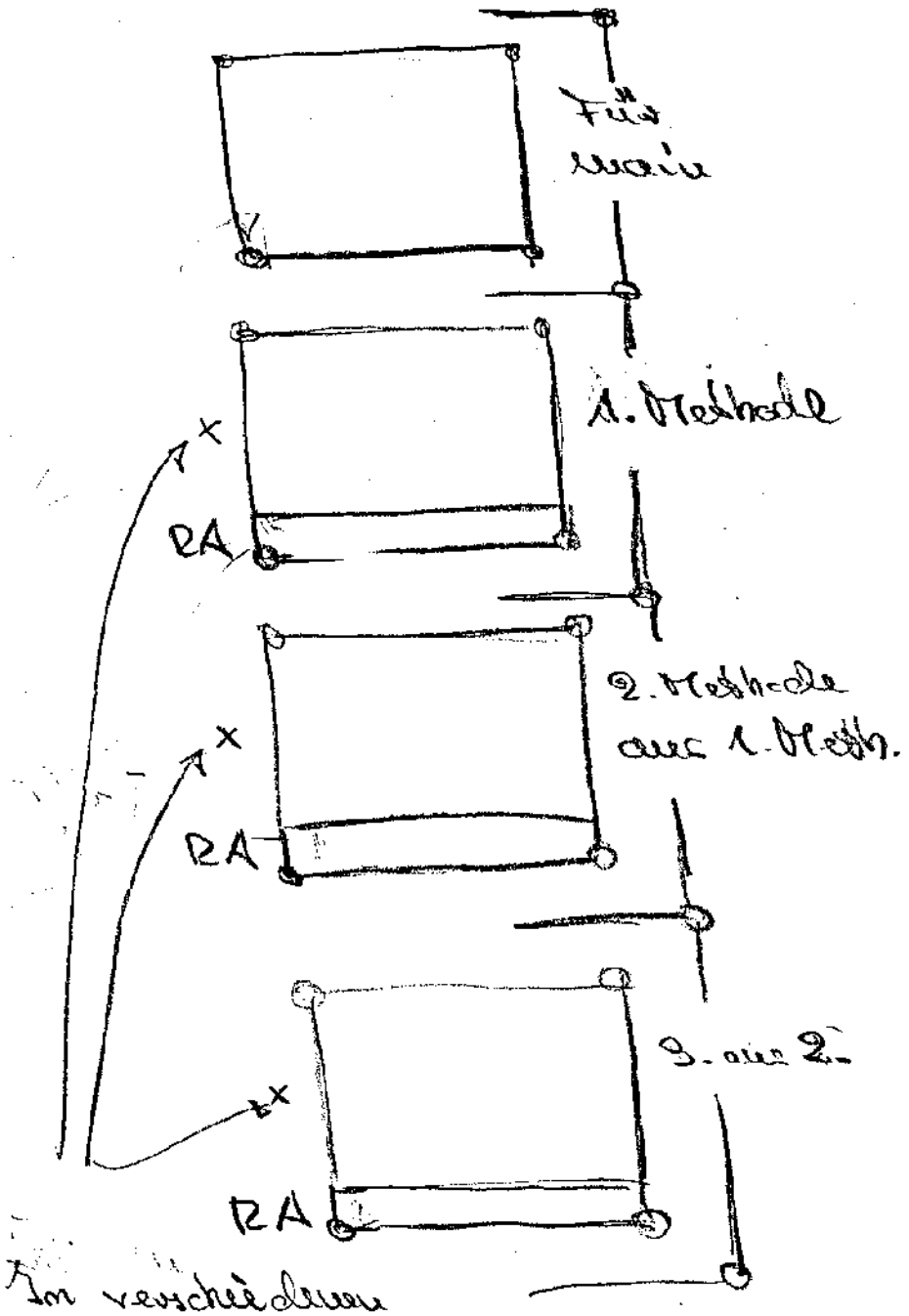
so aus:



Am Ende von mehrprogramm(4) bezieht sich wieder alles auf den aktuellen Rahmen.



Hätten wir innerhalb von mehrprogramm(n) weitere Methodenaufrufe; so würden weitere Rahmen generiert, etwa nach folgendem Prinzip:



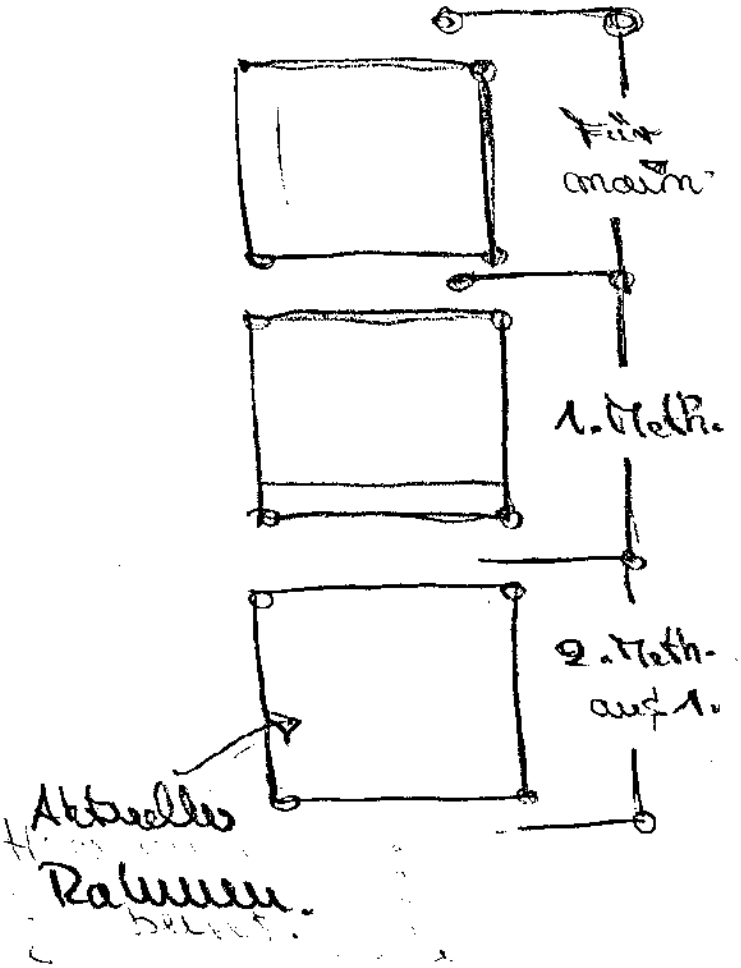
In verschiedenen

Räumen können

Variablen gleichen Namens

vorkommen!

Nach dem Ende von 3



Die Rahmen prüfen mit
den Aufrufen. Das Programm
arbeitet im aktuellen Rahmen,
dem vordersten (obersten, unterstem
– je nachdem) Rahmen.

Die Rahmen sind nach dem

- Kellerprinzip, Stapelprinzip, pushdown oder stack Prinzip

organisiert. Das heißt:

- Aktuelles Rahmen
  = Zuletzt erzeugter Rahmen
  ↑
  Neuster Rahmen.

Das ganze ist der

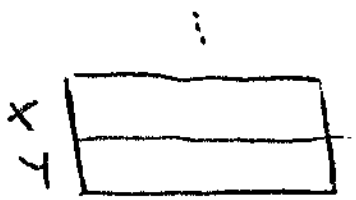
- Laufzeitstapel, run time stack, Prozedurkeller, ...

Da die Regeln zu gewöhnlichen
und in Methoden lokal deklarierten
Variablen behandelt, liegt sich
in dem vorliegenden Kontext auch
die Behandlung lokaler Variablen
in Blocks ein:

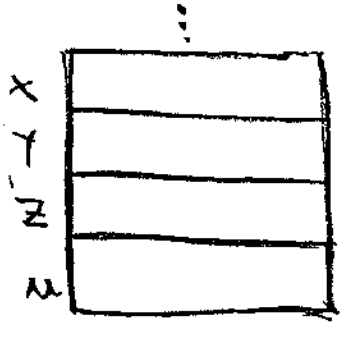
```
    ⋮

{ int x
  int y
    ⋮
  { int z      // Beachte: Hier noch
    int x      // einmal int x über-
    { ⋮        // setzungsfähig im
    { int z    // Unterschied zu Methoden!
      int u
      ⋮
    }
```

Man stellt sich am besten den aktuellen Rahmen vor, der dann in sich passiert:

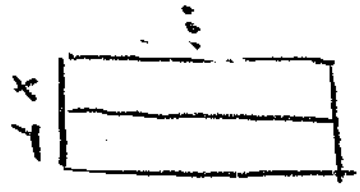
Zunächst



danach, x,y bleiben sichtbar,

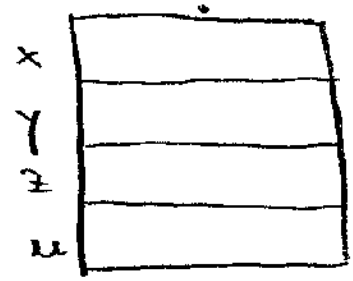


dann



z,u sind nach dem Block weg.

dann wieder:



Die Übergabe des Parameters erfolgt
in Java als

- Wertaufruf, call-by-value,
d.h. aktueller Parameter wird
ausgewertet und dem
aktuellen Frame übergeben.

Alternativen

- Referenzaufruf, call-by-reference.
Ist aktueller Parameter Variable
(Speicherplatz, Adresse). Der
formale Parameter wird ein
Alias des aktuellen.
Vermeidet Kopieren im
Laufzeit. Evtl. effizienter als

Verschiedene Fälle von Seiteneffekten.

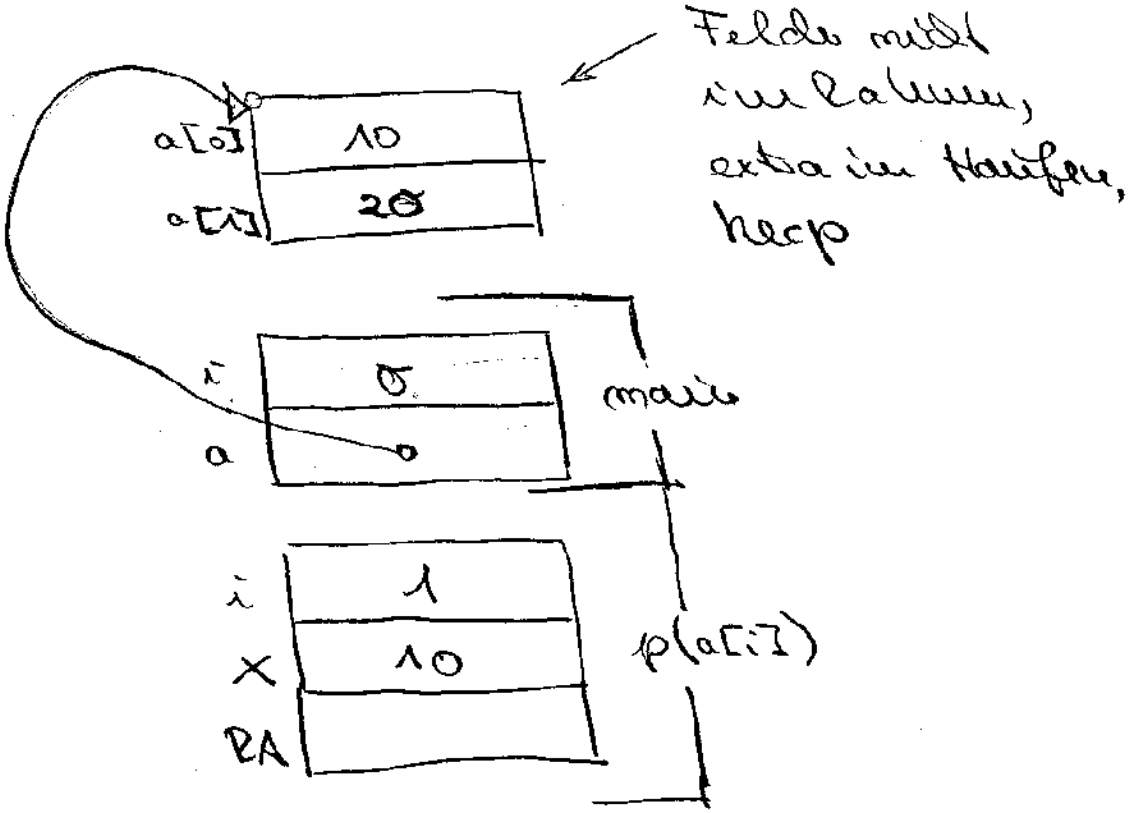
- Namensaufruf, call-by-name.
  Beim Aufruf einfaches schematisches Einkopieren des aktuellen Parameters so wie er ist.

Dazu ein Beispiel (Küchlin S. 154).

```
int [] a = new int [] {10, 20};

public static void p(int x) {
    int i = 0
    i = i + 1;
    x = x + 2
}
```

```
public static void main(String[] args)
{
    p(a[i])
    System.out.println(a[0]);
    System.out.println(a[1]);
}
```

Weiterlauf (Java):



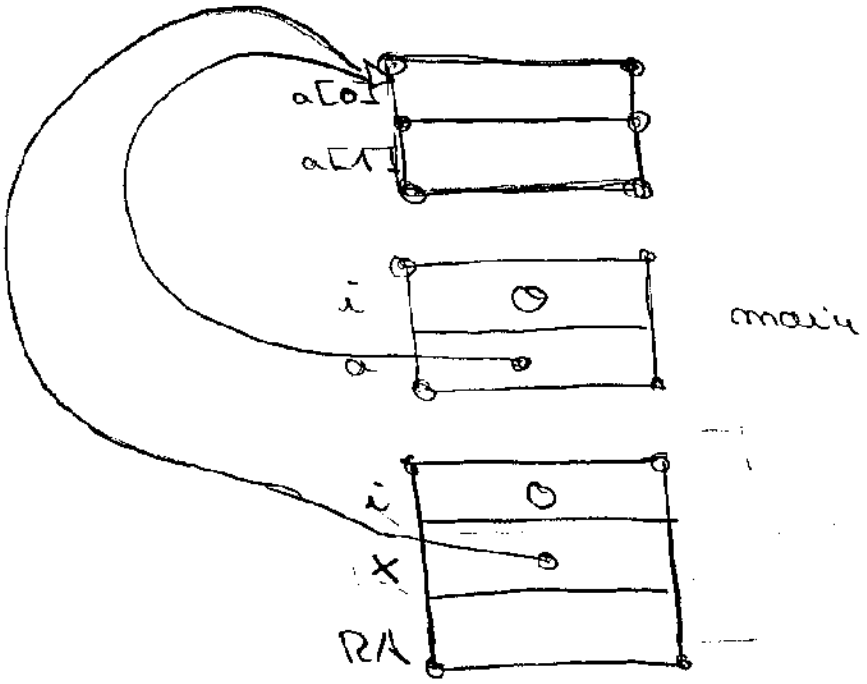
Dann x = 10. Ausgabe

10

20

(7.20)

Referenzaufruf (so nicht in Java):

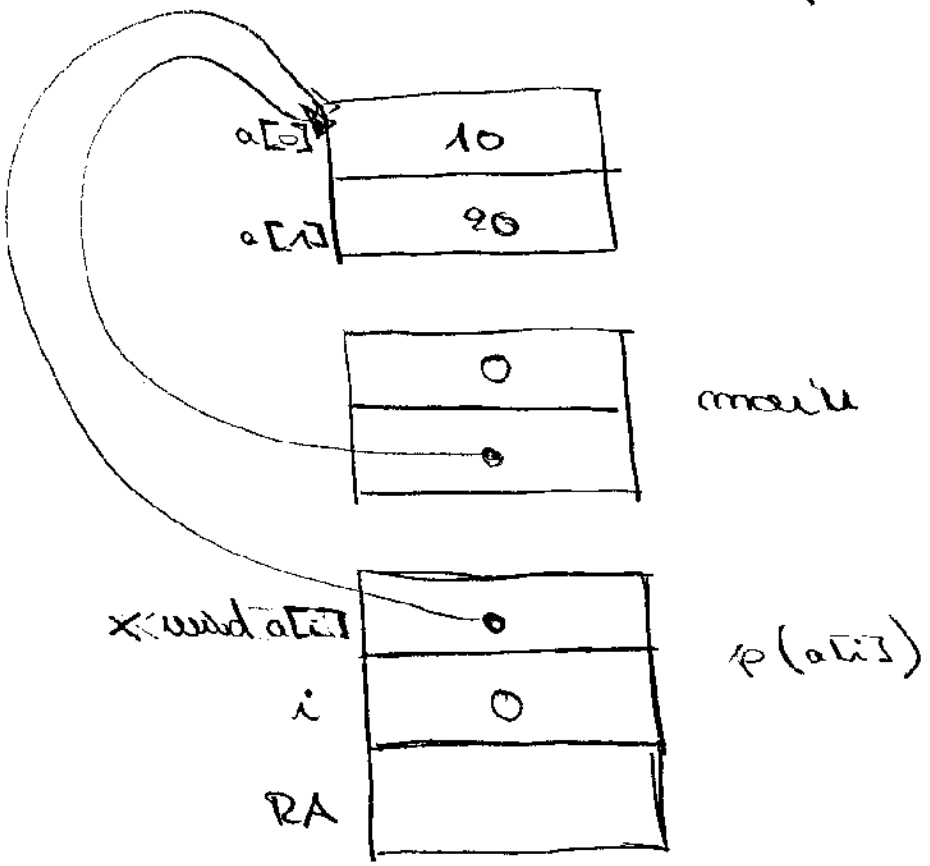


Da formale Parameter x wird zum Verweis auf den aktuellen a[0]. Ausgabe

12
20

(1.21)

Momentaufnahme (schon ganz nützlich im Labor):



Durch i = i+1 in p(a[i])

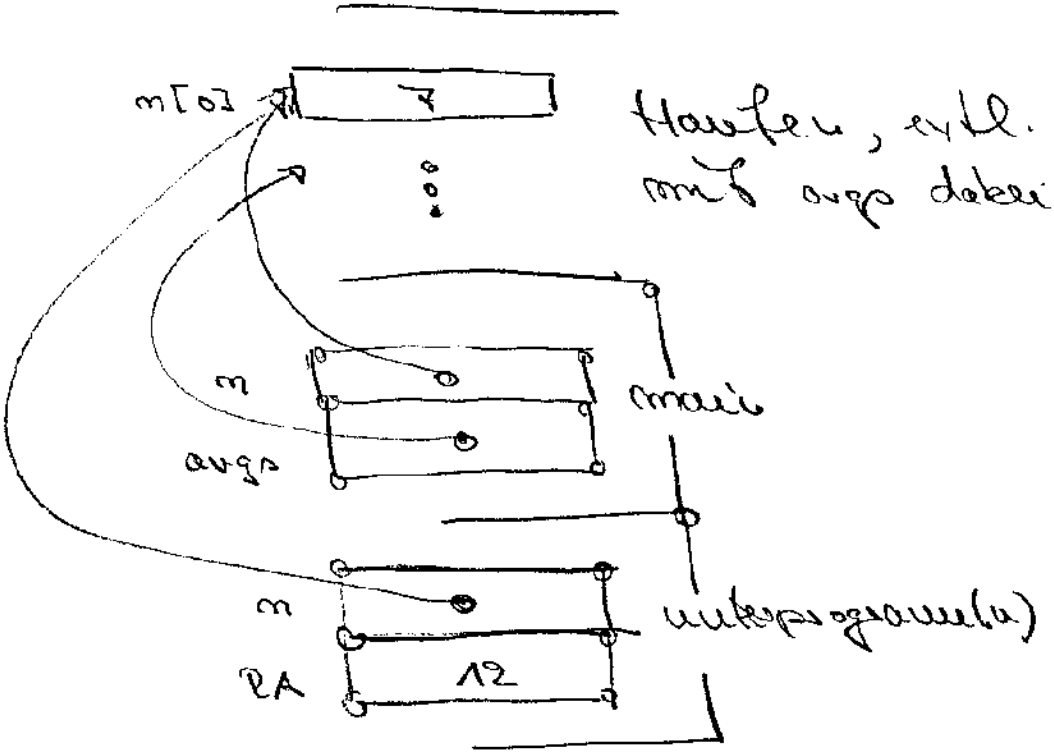
zeigt dann a[i] auf a[1].

Es wird a[1] geändert. Ausgabe

10

22 ,

Schließlich zu Feldern. Um das
unmöglich, langwierige Umkopieren
von Feldern zu vermeiden, werden
diese nicht in den Rahmen gespeichert,
sondern in den sogenannten Haufen
(hap). Keine Zugriffsbeschränkungen
wie im Lautzentkopf. Vgl. auch
Retz, Schiffler, Lesen S. 171 bis S. 175.
Lautzentheller und Haufen bei
Aufruf Test 2. java:



An den formalen Parameter n, eine Referenzvariable, wird der Wert von der Referenzvariable n übergeben. Im unterprogramm(n) wird das main im Hanoi geändert.

Abhilfe: Kopieren des Feldes. Kopie ans Hanoi. Problem: Nicht mehr

erweitbaus Speicherplatz (garbage collection im Laufzeitsystem integriert).

Am Buch bis S. 175. Dazu einige Erläuterungen.

Bisher hatten wir unsere Klassen mit Instanzvariablen ausgestattet:

```
public class <Klassenname> {
    public int x
    public float[] y
    :
}
```

Klassenvariablen werden mit static gekennzeichnet:

```
public class <Name> {
    public int x
    public static float a
    static long b
    :
}
```

Methoden

public static void <Name> <Para>

Klassenvariablen sind in allen

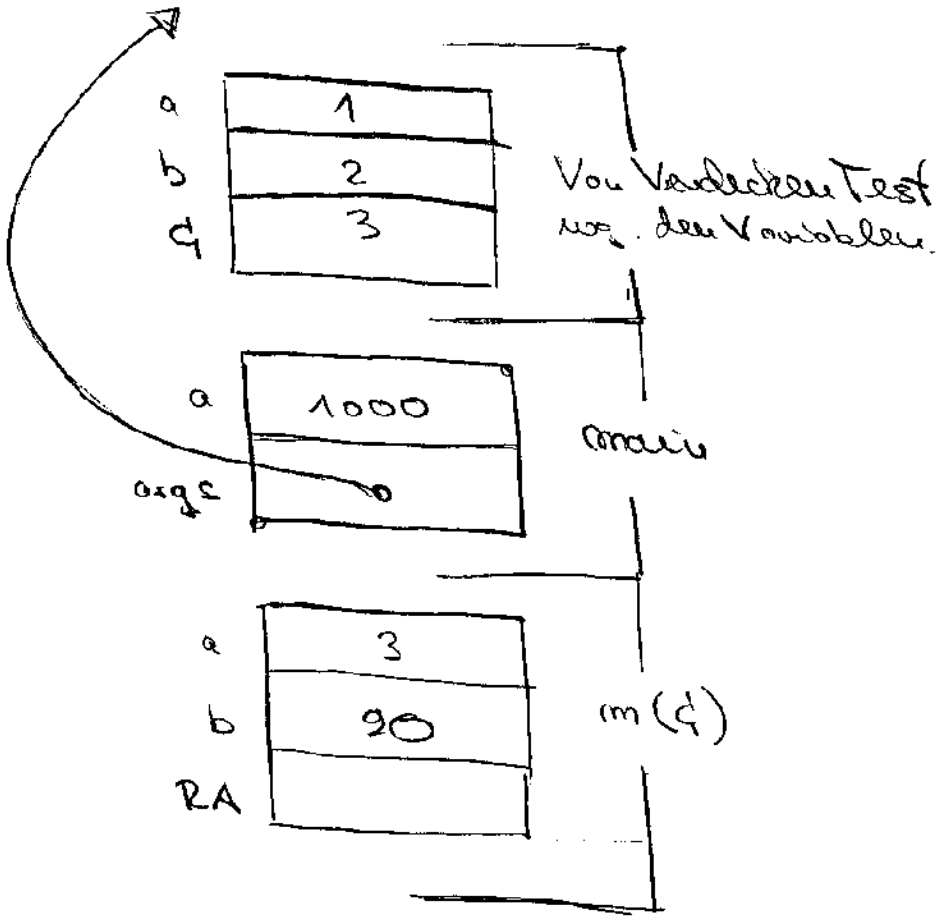
Instanzen einer Klasse gleich.

Für alle Instanzen existiert nur

eine Klassenvariable sozusagen.

Dazu eventuell später noch mehr.

Laufzeitbild von Verdeckertest:



Situation nach Beginn von m(q)

Ausgabe ausgesamt:

a= 1000 (Zeile 11

b= 2 12

m( )= 13

$a = 3$	Teile 5
$b = 28$	6
$q = 3$	7
$am(q) = 100$	13

↑

Erst wird das Ausgabeunde ausgerechnet als ganzes, dann wird ausgegeben.

7,22

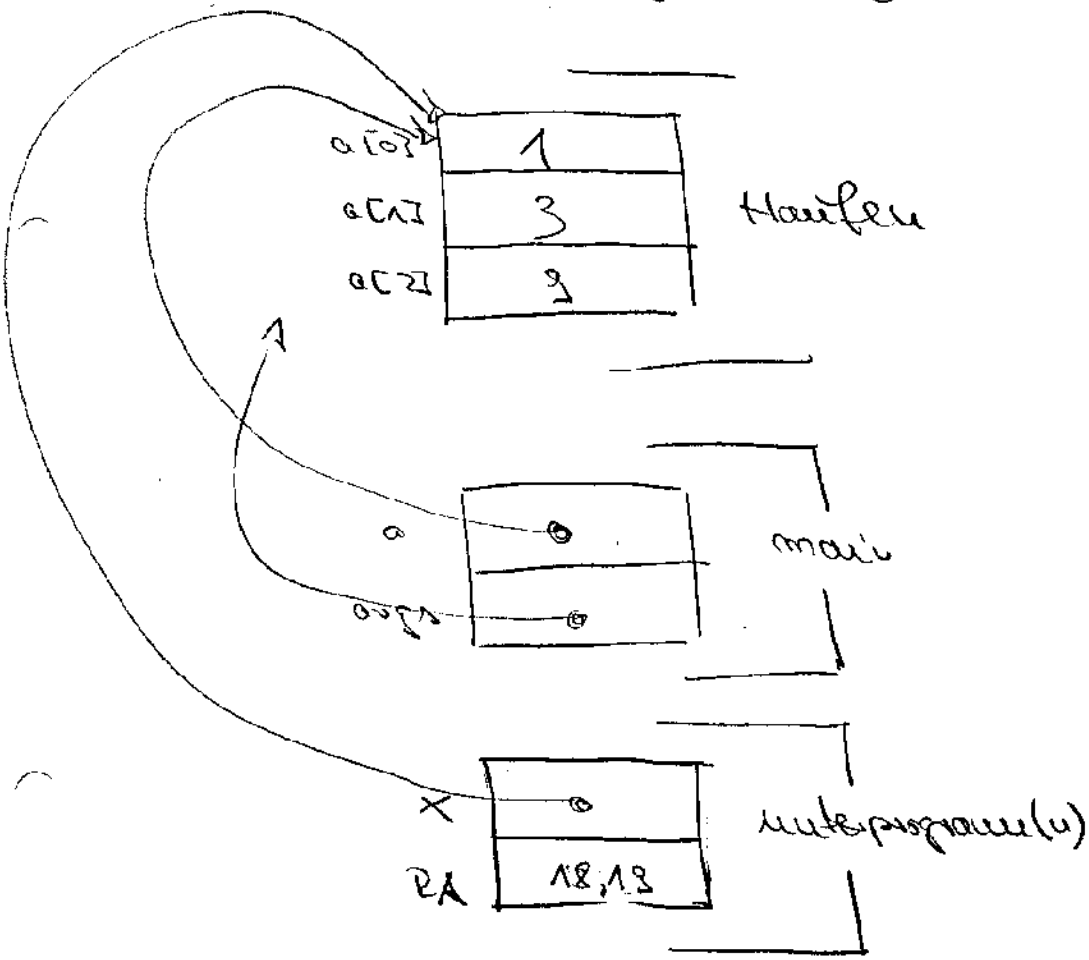
```
public class NocheinTest{
  //
  //
  //
  public static int[] unterprogramm(int[] x) {  // (1)
  x[0]= 7;  // (2)
  x = new int[2];  //(3)
  return x;
  }
  //
  // Hier ist das Unterprogramm zu Ende
  //
  public static void main(String[] args){
  int[] a = new int[3];
  a[0] = 1;
  a[1] = 3;
  a[2] = 9;
  a = unterprogramm(a); // (4)
  System.out.println(a[0] + "und " + a[1] );  //a[2] gibt Grenzueberschreitung,
  }
  }
```

// zum Laufzeit.

Schleifbild zu abschließendem Beispiel:

Nochmal Text. java von letzte Seite.

Haufen und Laufzeitkelle nach (1):

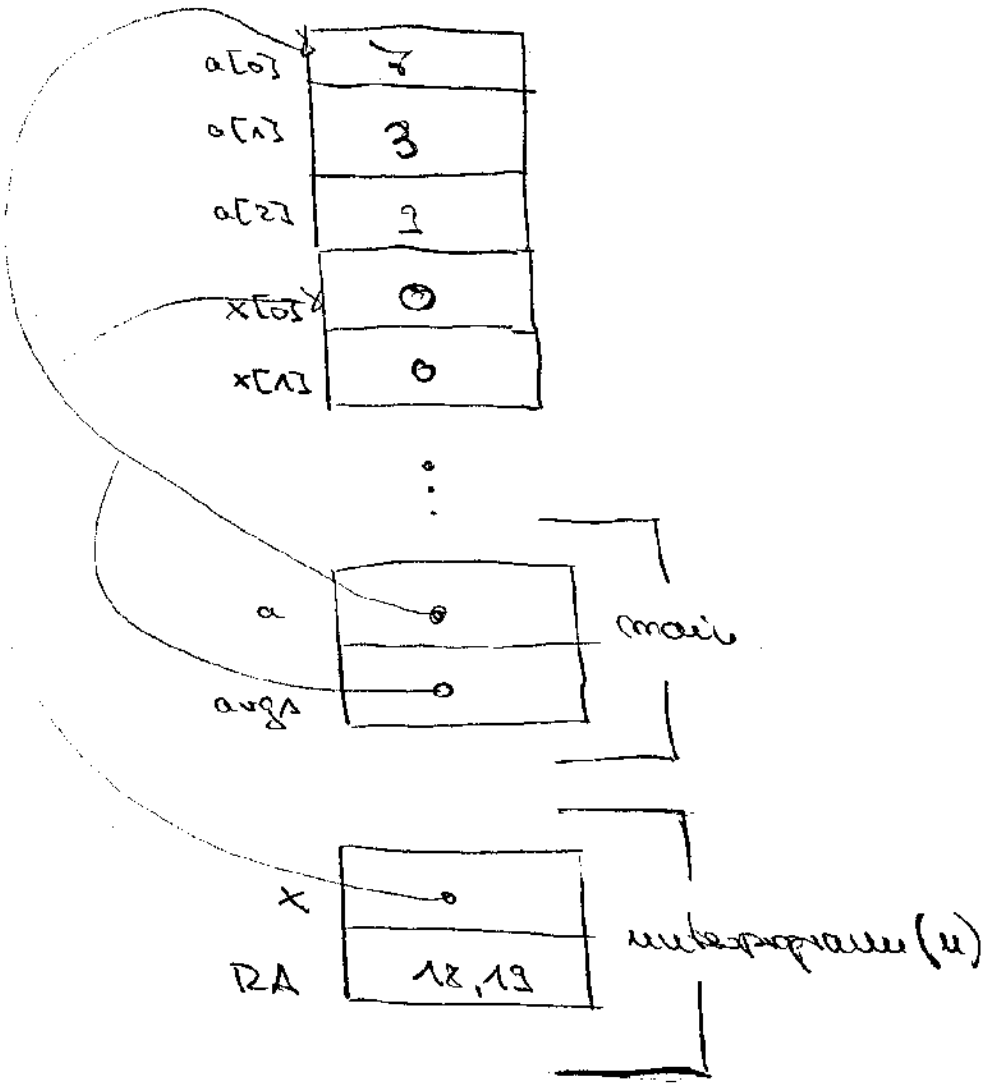


Nach (2) alles gleich außer

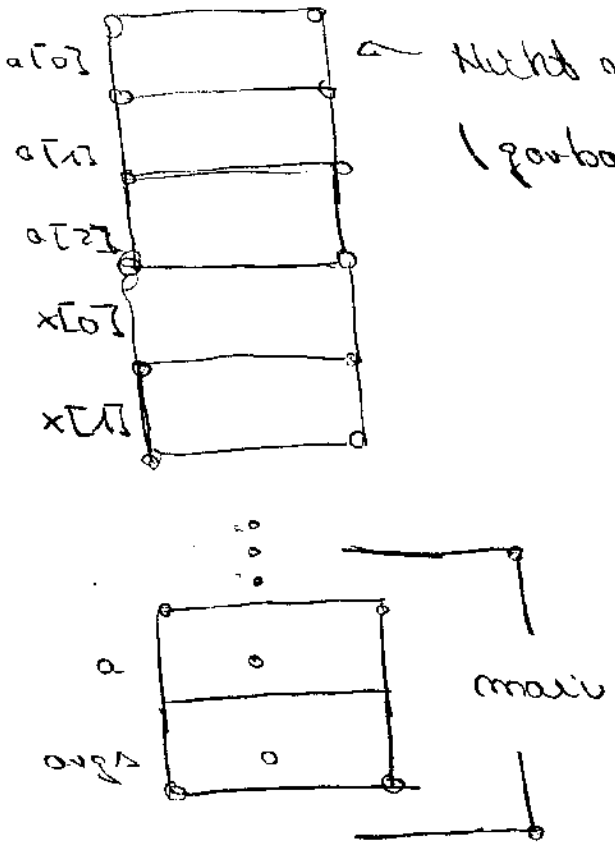
x[0] = a[0] = 7.

x wird ja wie die Referenzvariable a übergeben.

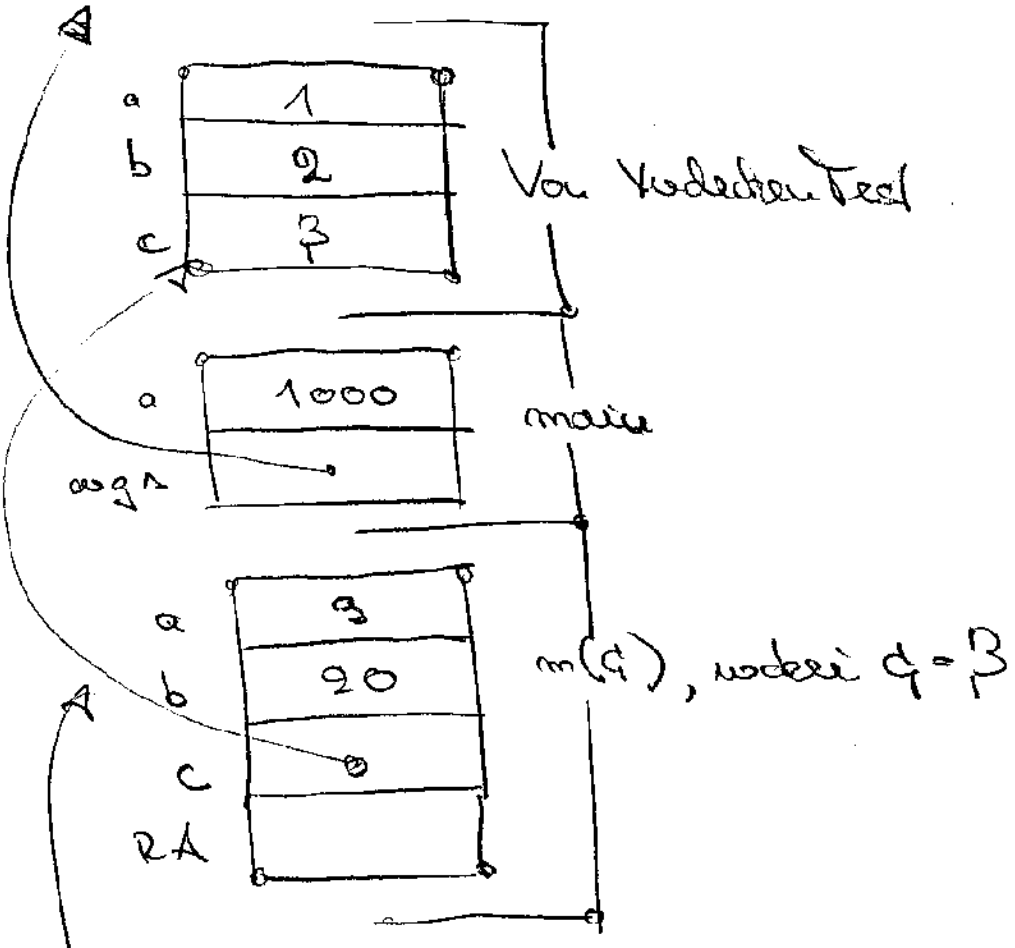
Nodo (3)



Schleifblock mode(4), s. auch a = x



Schließlich noch eine Korrektur zu S. 7.26.



Die Klassenvariable 4 bleibt sichtbar. Es ist auch möglich, 4 in m(a int) zu ändern.

1.33

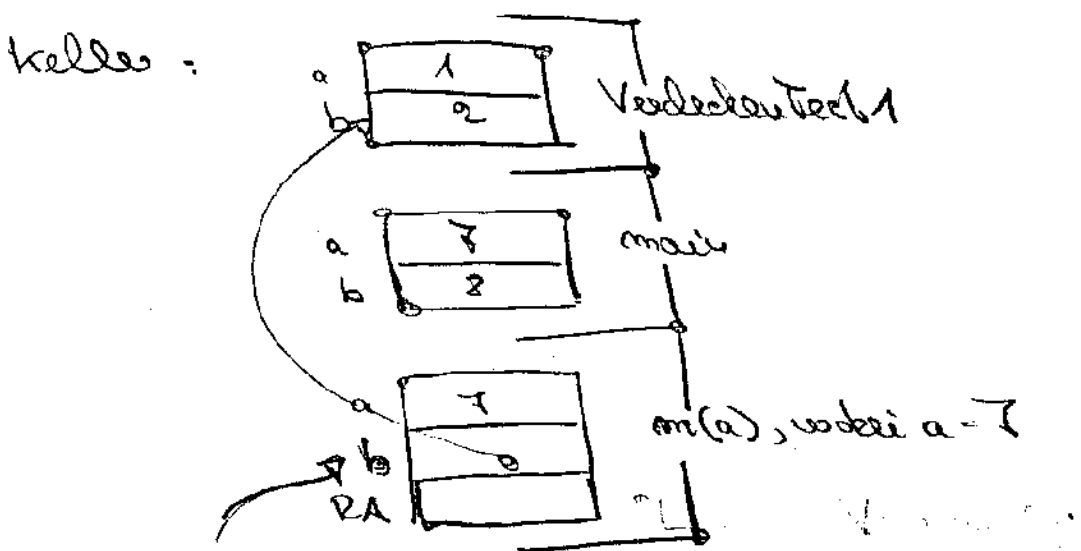
Noch ein Nachtrag zum Verdeckungsproblem, die Programme auf den folgenden Seiten

Verdecken Test 1. java

Verdecken Test 2. java.

```
public class VerdeckenTest1{  // Hier ein Test von Methoden.
  //
  // Die Deklaration beginnt mit
  // 2 Klassenvariablen: "static"
  //
  static int a = 1, b = 2;
  //
  // Jetze eine Methode
  //
  public static void m(int a){
     System.out.println("Der aktuelle " +
                        " Parameter fuer a  ist " + a);
     System.out.println("Der aktuelle " +
                        " Wert  fuer b  ist " + b);
     }
  //
  // Die Deklarationen enden.
  //
  public static void main(String[] args) {
  int a = 7, b= 8;
  //
  //
  // Deklaration von lokalen Variablen .
  //
  //
  //
  m(a) ;// Die Frage ist, welches b wird in diesem
        // Aufruf genommen.  Das von der Klasse, b= 2.
   }
  }
```

Evgibl Ausgabe a = 7, b = 2.



Wieso das lokale b aus main?

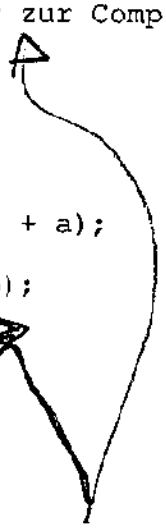
Das b kann in m(int a) auch geändert werden.

Also Regel: Lokale Variablen einer aufrufenden Methode sind in einer aufgerufenen Methode prinzipiell nicht sichtbar.

Das ist auch vollkommen sinnvoll. Sonst würde sich ein Programm nach Änderung von Namen von lokalen Variablen anders verhalten.

Klassenvariablen sind sichtbar, sofern nicht überschrieben.

```
public class VerdeckenTest2{  // Hier ein Test von Methoden.
  //
  // Die Deklaration beginnt mit
  // 2 Klassenvariablen: "static"
  //
  static int a = 1 ;// Im Vergleich zu VerdeckenTest1 , b = 2;
  //                  Bekommen Compilerfehler, da b in Zeile 15
  //                  nicht bekannt ist. Offensichtlich werden alle
  //                  Variablen ausser Parameter zur Compile-Zeit
  //                  sozusagen identifiziert.
  //
  // Jetze eine Methode
  //
  public static void m(int a){
     System.out.println("Der aktuelle " +
                        " Parameter fuer a  ist " + a);
     System.out.println("Der aktuelle " +
                        " Wert  fuer b  ist " + b);
     }
  //
  // Die Deklarationen enden.
  //
  public static void main(String[] args) {
  int a = 7, b= 8;
  //
  //
  // Deklaration von lokalen Variablen .
  //
  //
  //
  m(a) ;// Die Frage ist, welches b wird in diesem
        // Aufruf genommen.
   }
  }
```

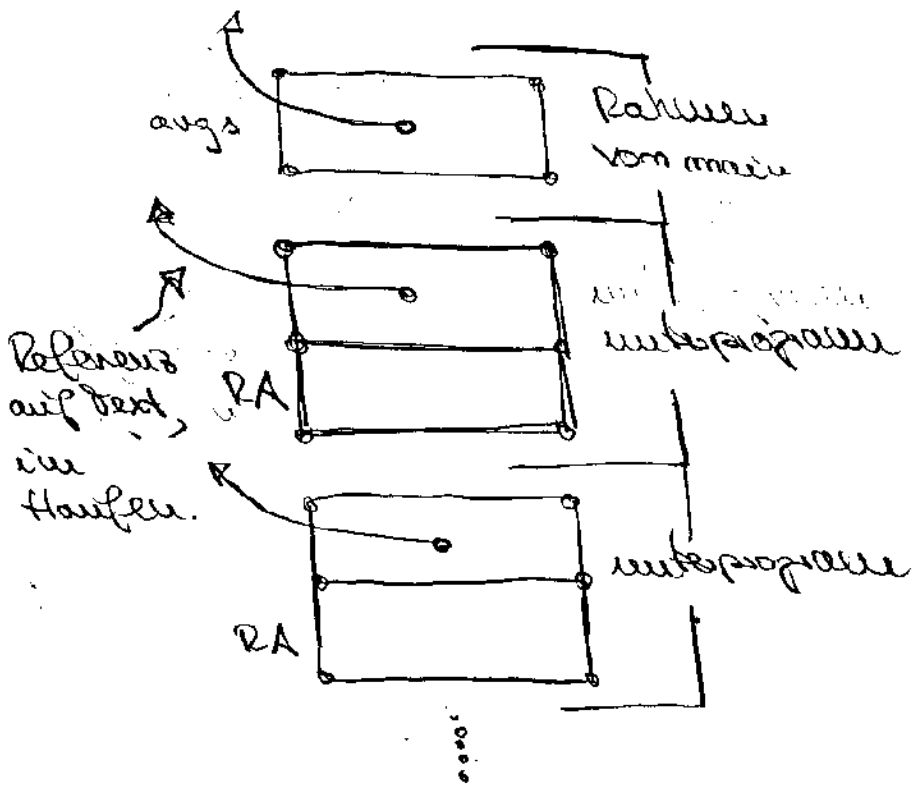


Dieses Programm übersetzt
gar nicht erst. Aber der
b ist vom Compiler nicht
identifiziert.

8. Das Wunder der Rekursion

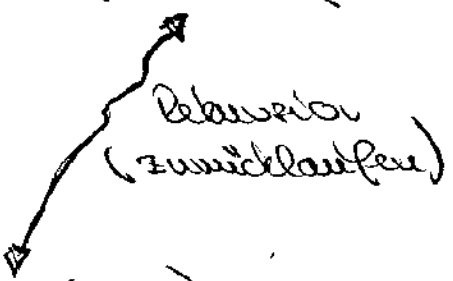
Zunächst wurden §. 176, §. 177 und §. 178 aus Rote, Schaffler, Seese besprochen.

Dazu ein paar Anmerkungen:
Was geschieht mit unseren Laufzeitbildern beim Programm Anwendlichkeit.java ?

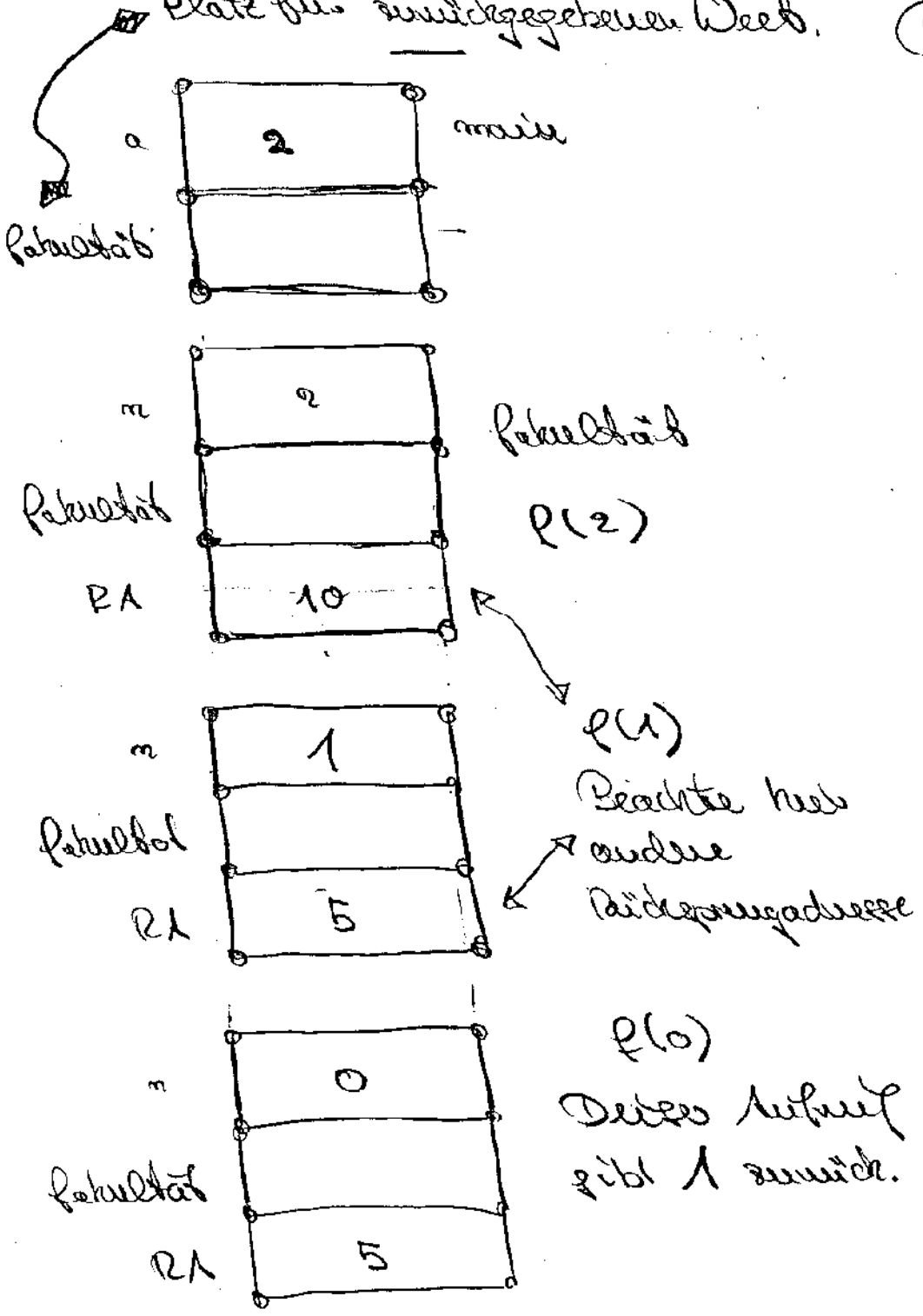


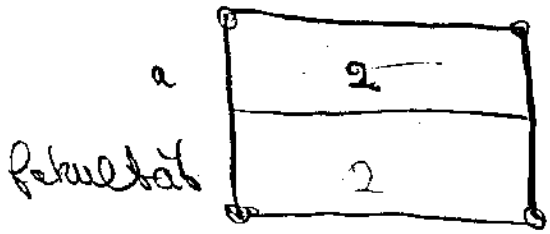
Beim Programm der Fakultät

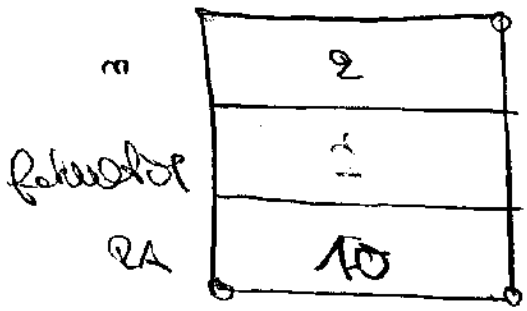
```
1  public class Fakultät {
2    public static int fakultät(int n) {
3      if (n == 0)
4        return 1;
5      return n * fakultät(n-1);
6    }
7  public static void main(String[] args) {
8  int a;
9  a = readInteger( );
10 fakultät(a);
11 }
```



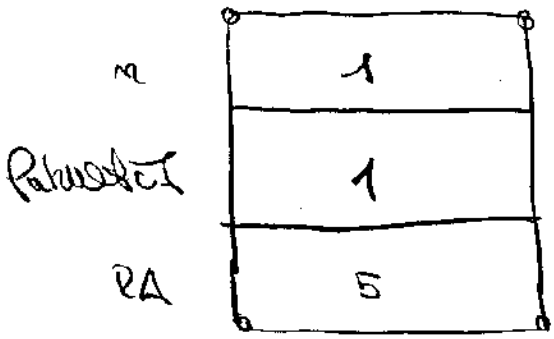
Platz für zurückgegebenen Wert.



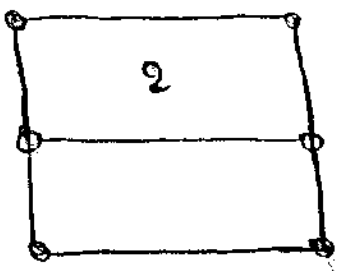


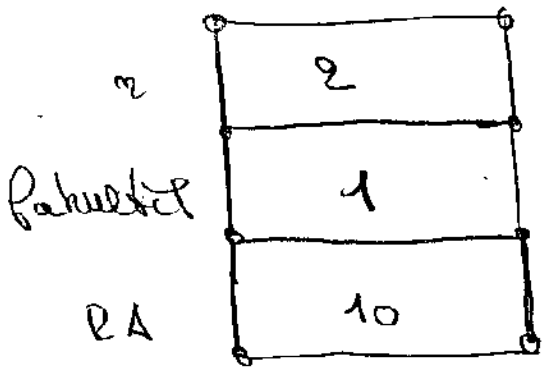


f(2)

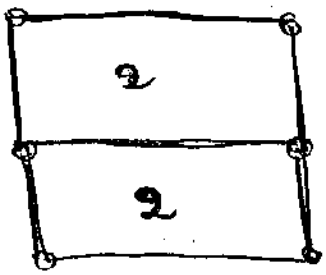


f(1)
1 = n · Fakultät
zurück, Wegen
RA 5 bei f(0) !





2 = n · Fakultät
zurück.
Wegen RA 5
bei f(1) !

a 

Fakultät – Wegen RA 10.

Korrektheit: Induktion über n. Zeige die Aussage: Ein Aufruf von Fakultät(n) liefert n! .

Es folgen weitere Beispiele zur Rekursion.

Eine rekursive Version des Euklidischen Algorithmus:

```
int ggT(int a, int b)
        // a >= b >= 0, a != 0
  if (b == 0) return a;
  return ggT(b, a % b)
```

Korrektheit: Induktion über den zweiten Argument von $ggT(a,b)$.

Zeigen: Für alle $a \geq b$ ist $ggT(a,b)$ korrekt.

Ind. Anf.: $b = 0$, dann klar.

(47)

Ind.-Schluß: Sei $b \neq 0$, $b \geq 0$.
Gelte die Behauptung für alle (!)
b' mit $b' \leq b$. Wir zeigen sie
für b. Es sei r der Rest des
gemeinsamen Teils von
a und b gleich dem von b und
$a \% b$. (vgl. frühere Version
des Euklidischen Algorithmus).
Also folgt die Behauptung, da
auf $ggT(a,b)$ der Ind.-Vor. anwendbar
ist.

Der Binomialkoeffizient ist definiert durch: Für $n \geq k \geq 0$ ist

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

Also ist für $k \neq 0$

$$\binom{n}{k} = \frac{n}{k} \cdot \frac{(n-1)\cdots(n-k+1)}{(k-1)!}$$

$$= \frac{n}{k} \cdot \binom{n-1}{k-1}.$$

Für $k=0$, $k>n$ ist $\binom{n}{k} = 0$.

Kombinatorische Interpretation:

$\binom{n}{0} = 1$

$\binom{n}{1} = n$

$\binom{n}{2} = \frac{n(n-1)}{2}$

$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \binom{n+1}{2}$

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$$

$$\vdots$$

$$\binom{n}{n} = 1$$

Es gilt:

$\binom{n}{k}$ = # Teilmengen mit genau k Elementen aus $1, \dots, n$.

Beweis 1: Jede Teilmenge mit k Elementen gibt aus folgendes Auswahlprozess:

1. Wähle 1. Element aus $1, \dots, n$: n Möglichkeiten

2. Wähle 1 Element aus dem Rest: $n-1$

$\vdots$

k. " : $n-k+1$

Also: $n(n-1)(n-2)\cdots(n-k+1)$ Möglichkeiten.

Also: Was haben wir gewählt?
Folgen $(a_1, a_2, \dots, a_k)$, angeordnet,
aus k verschiedenen Elementen. Die
Menge, ungeordnet, $\{a_1, \dots, a_k\}$ kennt
von $k!$ Folgen: Alle Permutationen
von $a_1, \dots, a_k$. Also gilt

Teilmengen mit genau k Elementen
aus $1, \dots, n = \frac{n(n-1)\cdots(n-k+1)}{k!} = \binom{n}{k}$.

Beweis 2: Induktion über n. Zeigen
für alle k mit $0 \le k \le n$ ist
$\binom{n}{k}$ = # der Teilmengen.

Ind.-Anf.: $n = 0$, dann $k = 0$ dann ✓

Ind.-Schluß: Ich muss $n > 0$. Gelte die Aussage für $n-1$ und alle möglichen k. Für n und $k = 0$ gilt die Beh. Sei $k > 0$.

k-elementige Teilmengen von $\{1, \dots, n\}$, die n enthalten

$= \#(k-1)$-elementige Teilmengen von $\{1, \dots, n-1\}$

$= \binom{n-1}{k-1}$ nach Ind.-Vor.

k-elementige, die n nicht enthalten

$= \binom{n-1}{k}$ nach Ind.-Vor.

$$\binom{n-1}{k-1} + \binom{n-1}{k}$$

$$= \frac{(n-1) \cdot \dots \cdot (n-1-(k-2))}{(k-1)!} + \frac{(n-1) \cdot \dots \cdot (n-1-(k-1))}{k!}$$

$$= \frac{(n-1)\cdot\ldots\cdot(n-k+1)}{(k-1)!} + \frac{(n-1)\cdot\ldots\cdot(n-k)}{k!}$$

$$= \frac{k\,(n-1)\cdot\ldots\cdot(n-k+1) + n(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$$

$$- \frac{k\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$$

$$= \binom{n}{k}.$$

Iwei rekursive Programme:

Bin1(int n, int k)

return (Bin1(n-1, k-1) + Bin1(n-1, k))

Wie Abbruchbedingung? Einmal

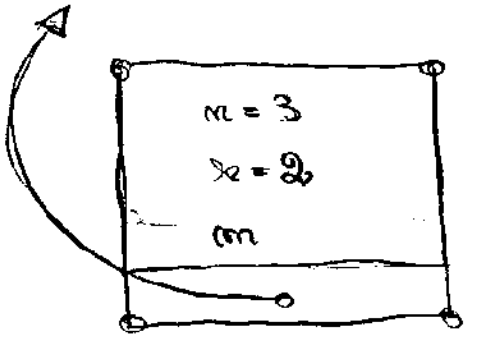
$k = 0$ einmal $n = k$.

Vergleiche Programme Bin1.java,

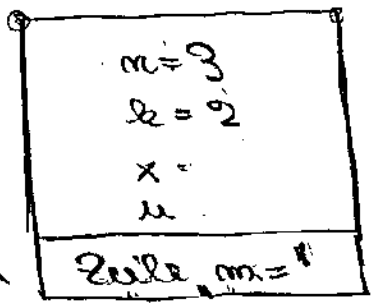
Bin1a.java. Bin1.java: Speicherzugriffe:

Fehler wegen Nichtabbruch der Rekursion.

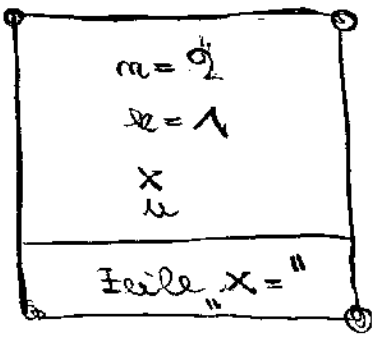
Laufzeittabelle bei Binomial1(3,1)



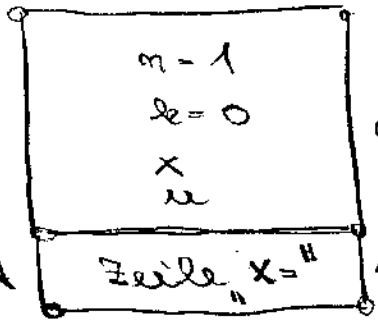
main



Binomial1(3,2)

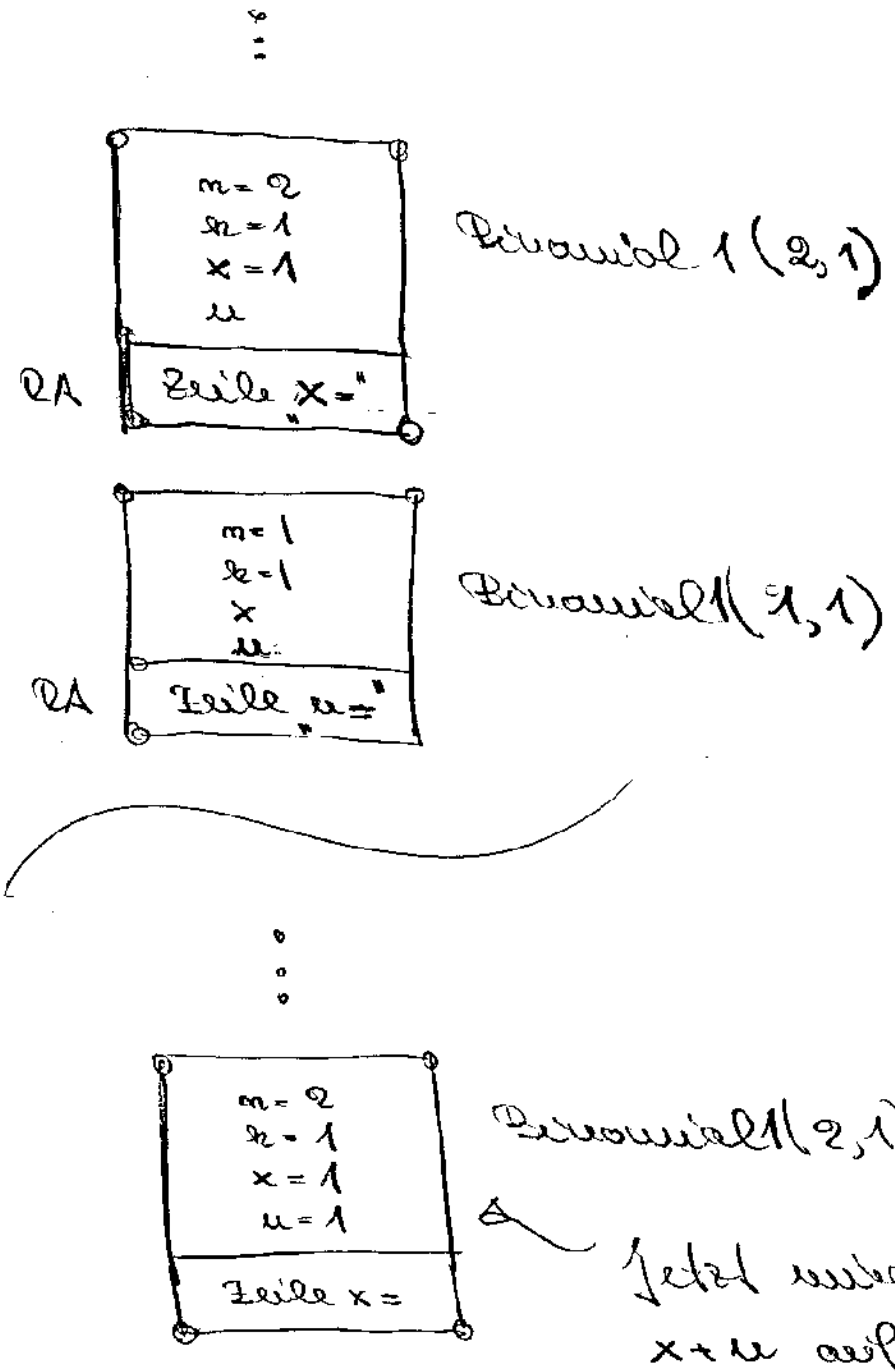


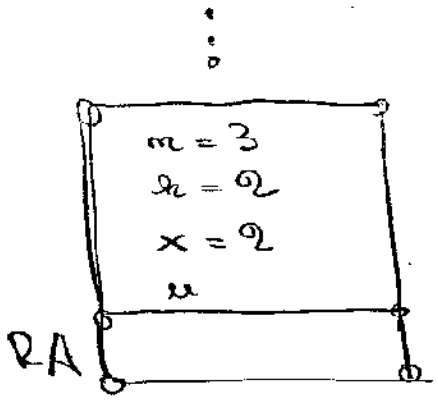
Binomial1(2,1)



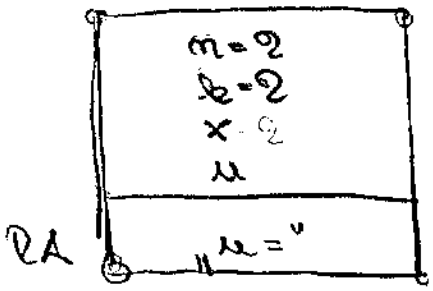
Binomial1(1,0)

Vom Aufruf mit 2,1. Also x eintragen.

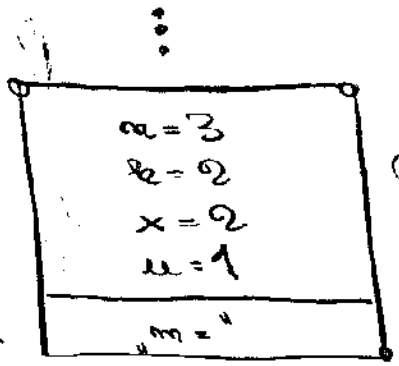

m = 2
k = 1
x = 1
u
Zeile x =
Binomial 1 (2,1)
m = 1
k = 1
x
u
Zeile x =
Binomial 1 (1,1)
m = 2
k = 1
x = 1
u = 1
Zeile x =
Binomial 1 (2,1)
Jetzt wird x + u auf x übergeben.



Binomial1(3,2)

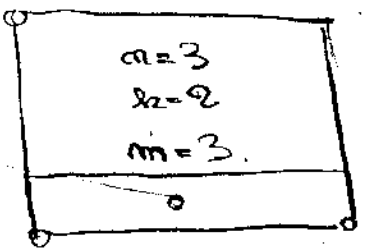


Binomial1(2,2)



Binomial1(3,2)

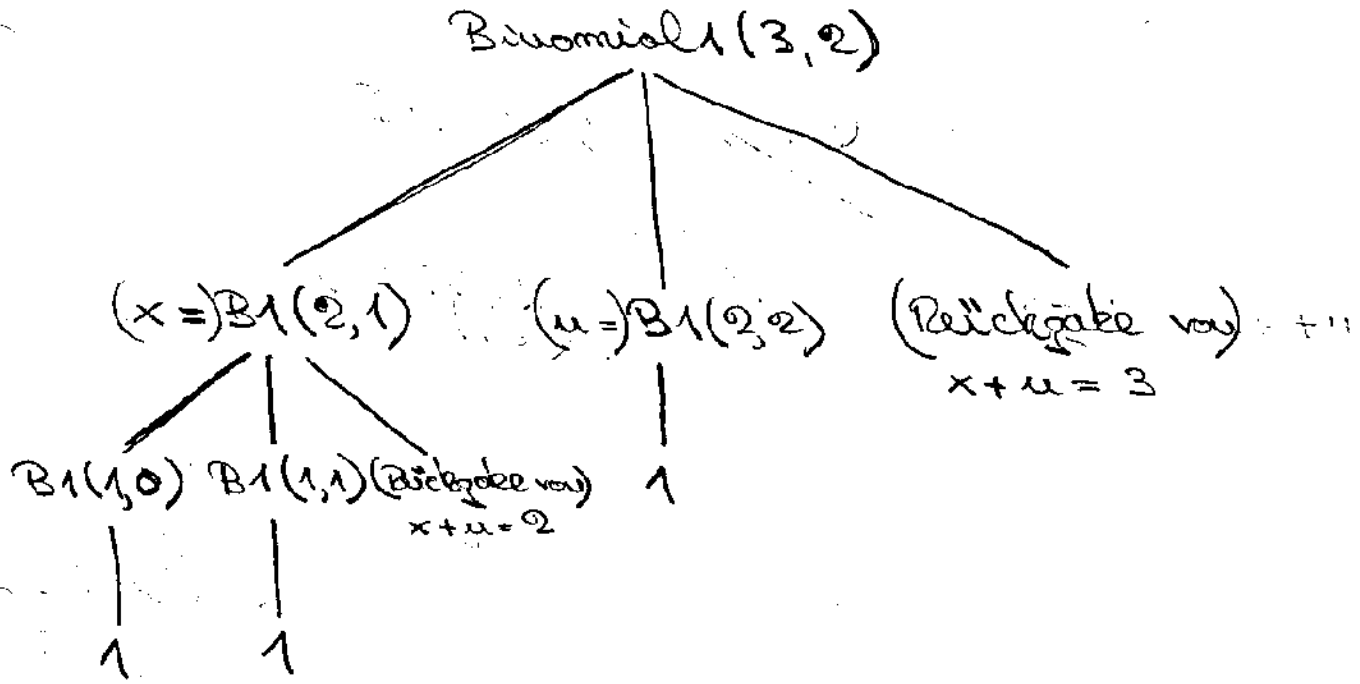
← libérée
vaut z = 2+1
en m.



mais

Neu.

Wichtig ist die Darstellung des Ablaufs eines Programms, insbesondere bei rekursiven Methoden als sogenannter Baum.



Einige Begriffe:

- Wurzel ist B1(3,2)
- Kinder (auch direkte Nachfolger) von B1(3,2) sind B1(2,1), B1(2,2), x+u=3

- Kinder von B1(2,1) sind

  B1(1,0), B1(1,1), x+y=2

  ⋮ • Vater analog.

- Nachfolger von B1(2,1) sind

  B1(2,1) (!), die Kinder und

  die Kinder des Vaters usf.

- Die Kinder sind geordnet von

  links nach rechts: Erstes Kind,

  Zweites, ... • Erstes Teilbaum,

  zweites Teilbaum, ...

- B1(3,2), B1(2,1), 1, 1, ... heißen

  auch Knoten. Man unterscheidet

  innere Knoten und Blätter.

  Die Blätter sind:

  1, 1, 1, x+y=3.

Ausführung von B1(3,2)

=

Ausführung des Kinder von B1(3,2)
von links nach rechts:

B1(2,1); B1(2,2); x+y=3 ausrechnen.

Ausführung von B1(2,1) dann
analog ...

Ausführung von B1(3,2) ...

=

Abarbeiten des Baums in
Postorder:

Erst die Kinder, links nach rechts,
dann der Vater.

Das entspricht genau dem Beispiel
der Laufzeitbelastung und der Bedeutung
der Rücksprungadresse:

(3,2)

(3,2) (2,1) // Erstes Kind von (3,2)

(3,2) (2,1) (1,0) // Erstes Kind von (2,1)

(3,2) (2,1) (1,1) // Zweites Kind von (2,1)

(3,2) (2,1) x+u=2 // Drittes Kind von (2,1)
// Erstes Kind von (3,2)
// ist fertig.

(3,2) (2,2)

(3,2) (2,2) 1 // (2,2) hat nur 1 Kid.

(3,2) x+u=3 // Drittes Kind von (3,2).

(3,2) // Hier ist (3,2) fertig.

Nun zum Programm Bin2.java.

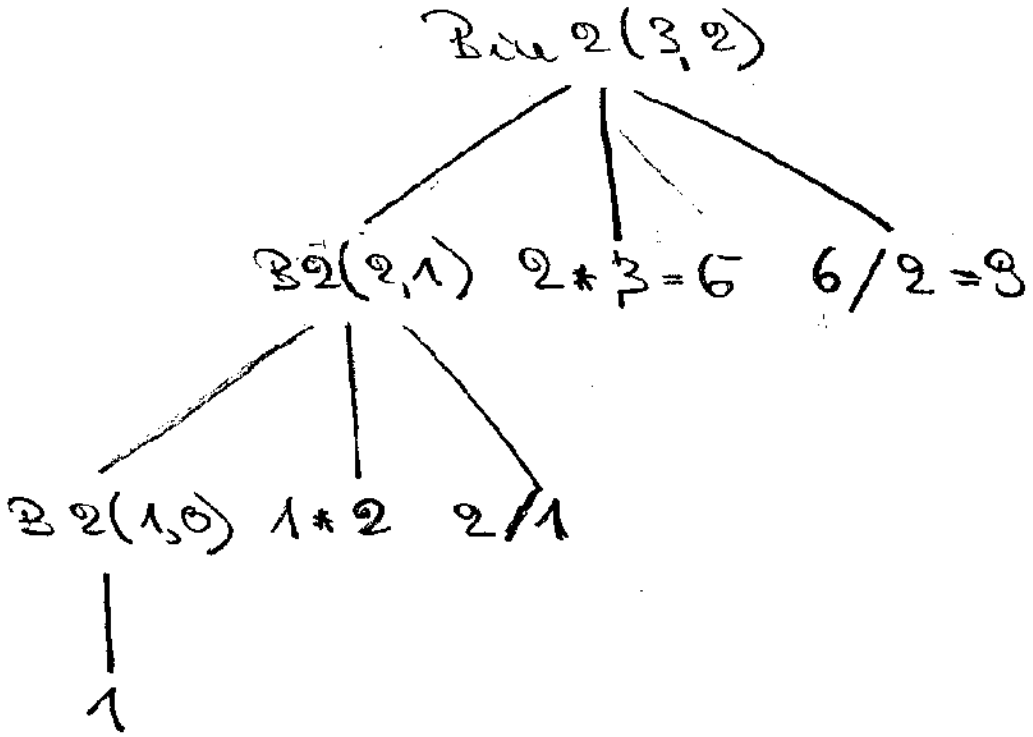
Das geht nach dem Prinzip vor:

Bin2(int n, int k)

return ((Bin2 (n-1, k-1) * n)/k)

Abbruchbedingung k = 0.

Baum bei Eingabe von 3 und 2:



Laufzeitbeispiel:

(3,2)

(3,2) (2,1)

(3,2) (2,1) (1,0) // Übergabe von 1

(3,2) (2,1) // Übergabe von 1×2/1=2

(3,2) // Übergabe von 2×3/2

// an main.

Die Speicherung der Rücksprung-
adresse im aktuellen Frame
stellt sicher, daß die übrigen
Kinder eines Knotens in der
richtigen Reihenfolge bearbeitet
werden.

Das (modulare) Potenzieren geht ganz leicht nach dem Prinzip:

$F(a,b)$

if $(b == 0)$ return 1 ;

if b gerade $x = F(a, b/2)$ (a)

return $(x * x)$

if b ungerade return $(a * F(a, b-1))$ (b)

Korrektheit: Induktion über b.

- Ind.-Anf.: $b = 0$ ✓

Induktionsschluß: b gerade.

Dann ist nach Ind.-Vor

$$x = F(a, b/2) = a^{b/2}.$$

Also Ausgabe $x \cdot x = a^b$.

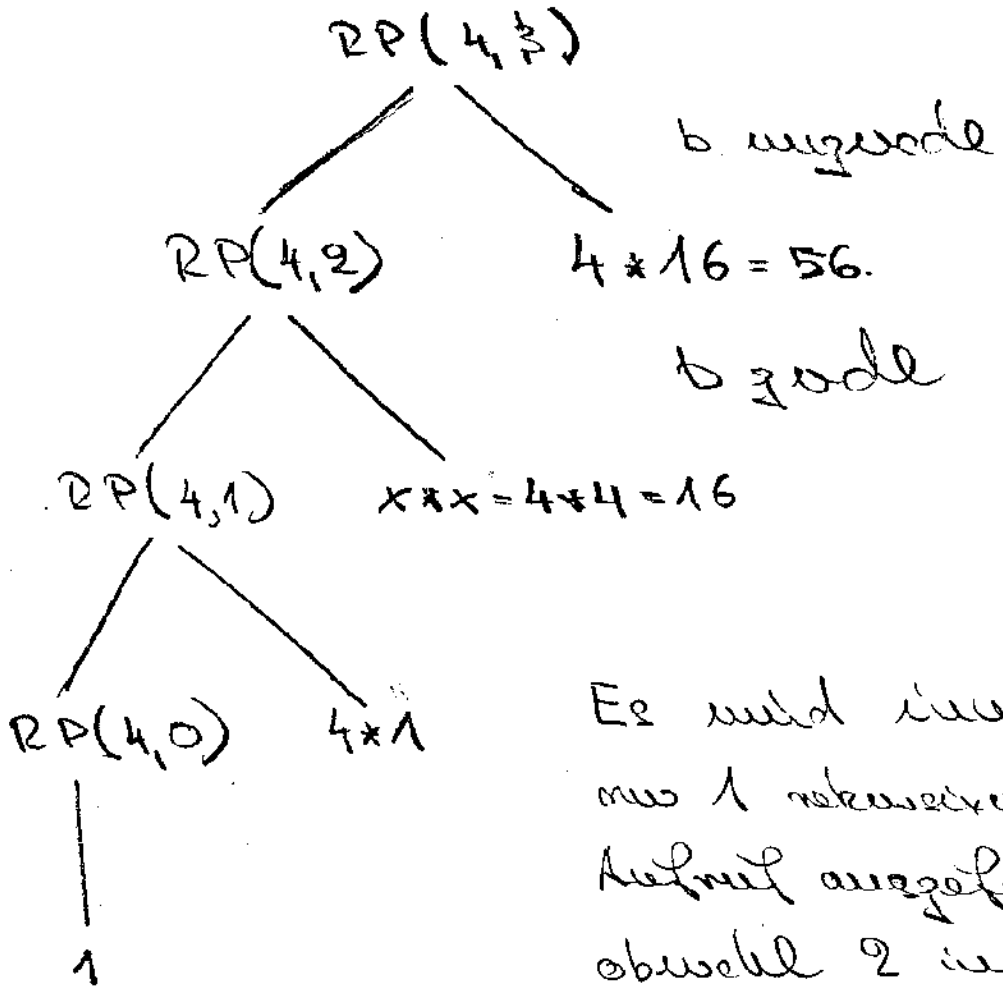
b ungerade, dann Ausgabe von

$$a \cdot F(a, b-1) = a \cdot a^{b-1} = a^b$$

mit Ind.-Vor.

Das Programm RekPot.java.

Eingabe etwa a = 4, b = 3, m = 100.



Es wird immer nur 1 rekursiver Aufruf ausgeführt, obwohl 2 im Programm sind.

Am Laufzeittableau läßt sich noch einmal die Bedeutung des Rückspringpointers zeigen:

(4, 3, m) ← RA

(4, 3, m) (4, 2, m, ⓑ) // Da 3 ungerade.

(4, 3, m) (4, 2, m, ⓑ) (4, 1, m, ⓐ) // Au ⓐ bezogen auf Vorgängerelement gibt es weiter.

(4, 3, m) (4, 2, m, ⓑ) (4, 1, m, ⓐ) (4, 0, m, ⓑ)

// Übergabe von 1.

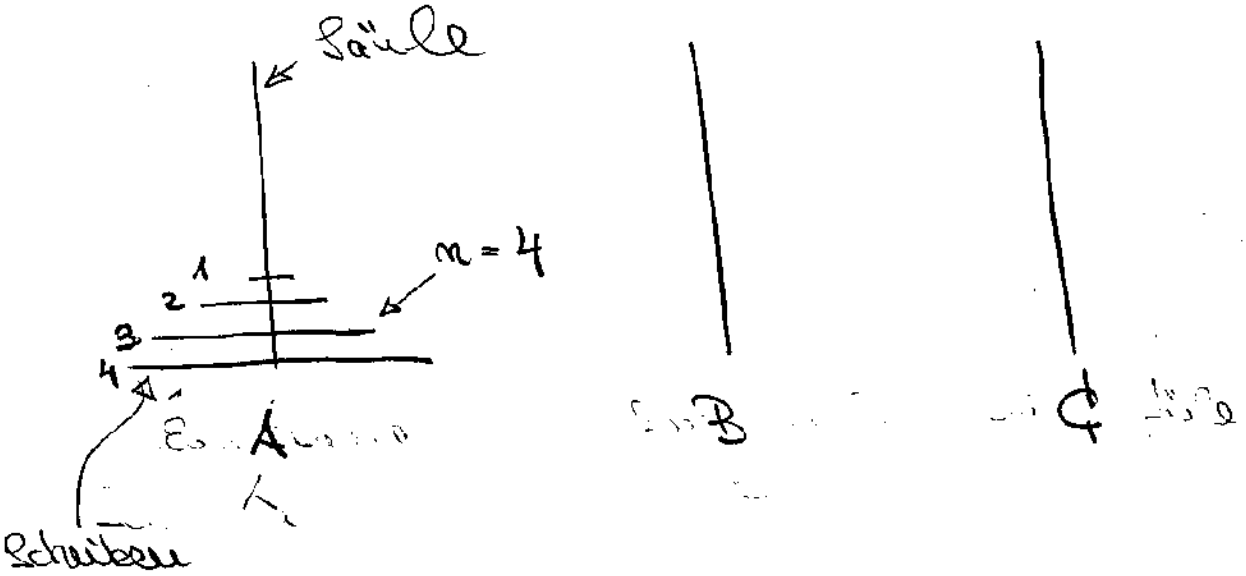
(4, 3, m) (4, 2, m, ⓑ) (4, 1, m, ⓐ) // Übergabe 4.

(4, 3, m) (4, 2, m, ⓑ) // Übergabe von 16.

(4, 3, m) // Übergabe von 56

Das Problem der Türme von Hanoi ist das Paradebeispiel für die Anwendung der Rekursion.

Ausgangssituation

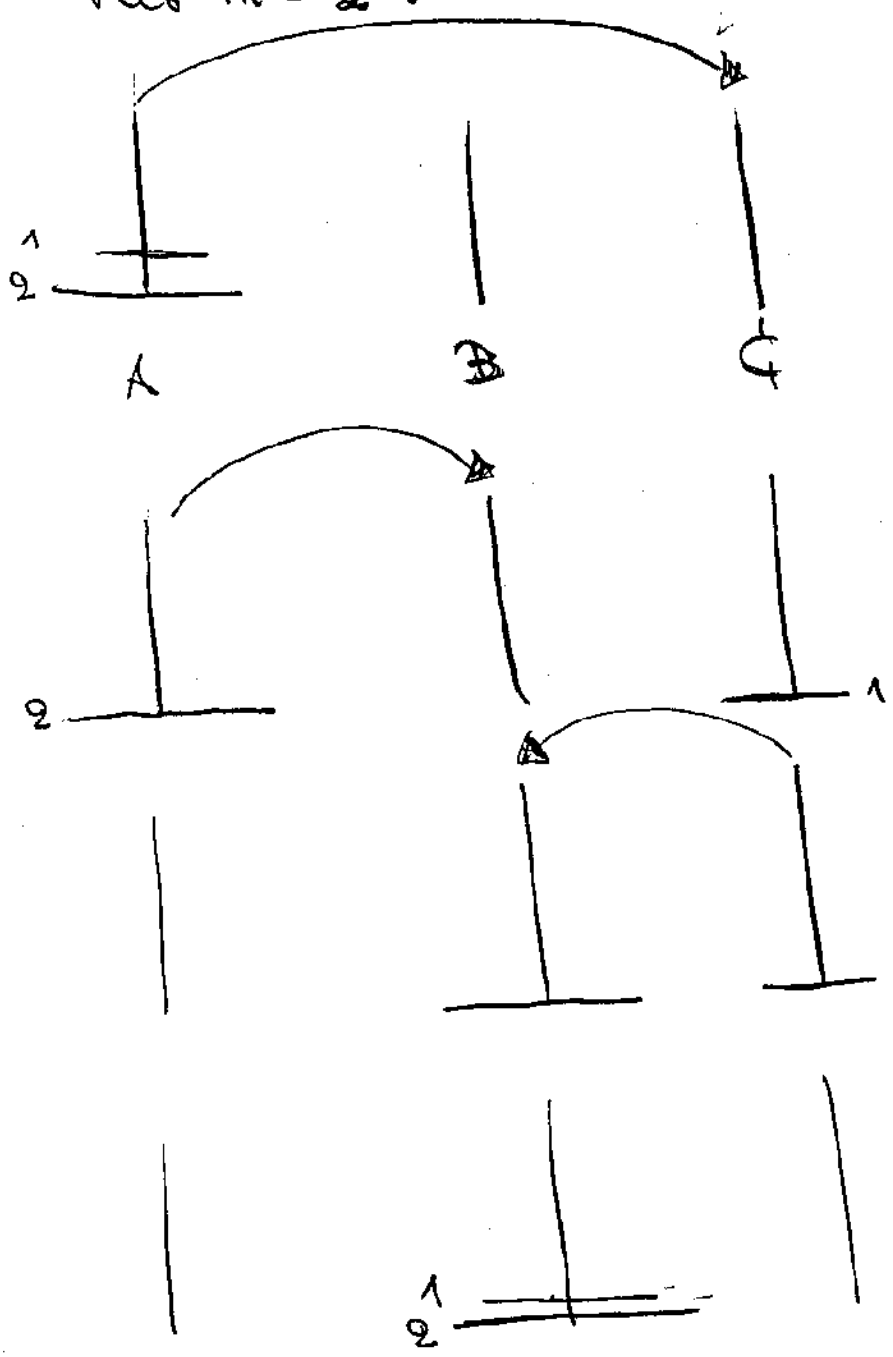


Aufgabe: Bringe die n Scheiben von A nach B nur mit Hilfe von C.

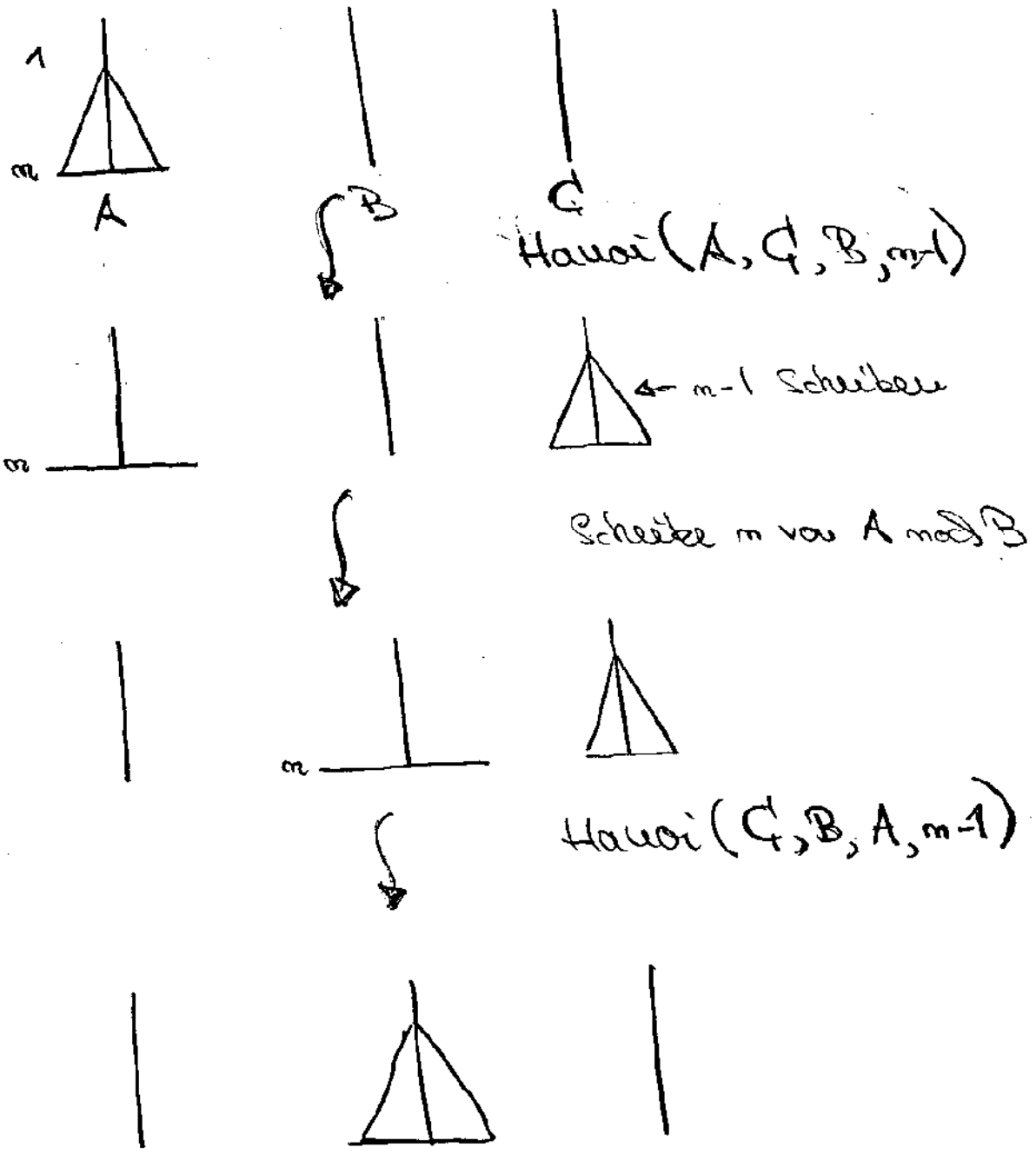
Beschränkung: Pro Schritt eine Scheibe. Nie größere über kleinere Scheibe.

Pour $n = 2$:

(2.26)



Im allgemeinen Hanoi(A, B, C, n)



Programm gemäß

```
Hanoi(A,B,C,n)
{ if (n = 1)
    { "Schiebe n=1 von A nach B";
      return;   // Beachte: return beendet
                // den aktuellen
                // Aufruf.
    }
  if (n > 1) {
    ⓐ Hanoi(A, C, B, n-1);
    ⓑ // Schiebe n von A nach B";
    ⓒ Hanoi(C, B, A, n-1);
      // return unnötig, da Ende erreicht.
  }
```

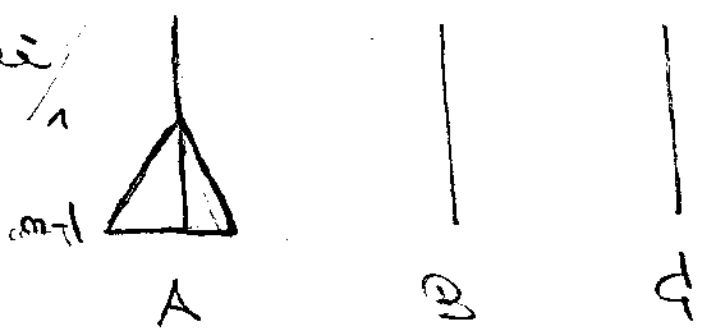
Zur Korrektheit ist zu zeigen:
Für alle $n \geq 1$ liefert Hanoi(A,B,C,n)
eine Folge von regelkonformen
Umsetzungen bezogen auf

Induktion über n.

$n = 1$ ✓

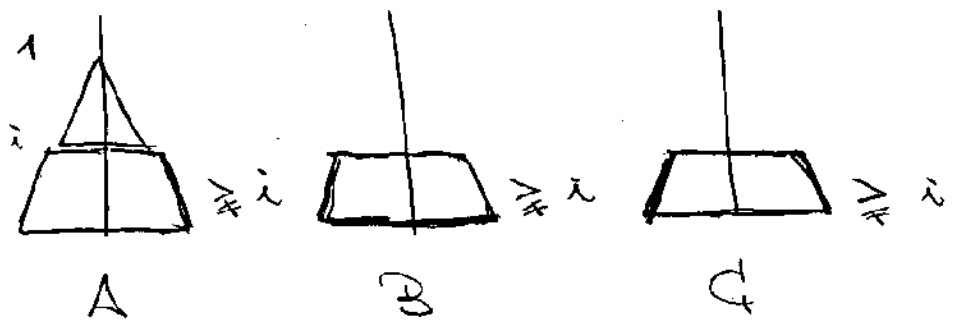
$n > 1$ Ind.-Vor. lautet nun:

Bei

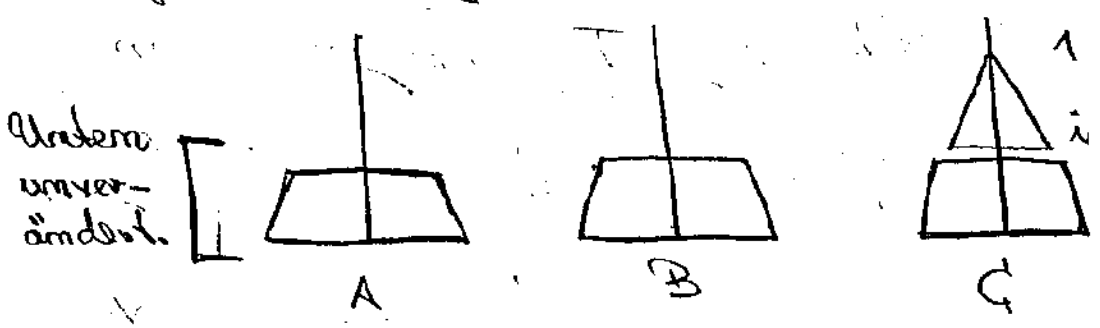


ist Hanoi(A,B,C) richtig. Wie
müssen nichts über Hanoi(A,C,B)

Tatsächlich müssen wir wesentlich allgemeiner vorgehen: wir zeigen induktiv über $i \geq 1$. Für jede mögliche Situation der A, t



liefert Hanoi(A, B, C, i) eine regelkonforme Folge von Umsetzungen so



Ebenso für alle Vertauschungen von A, B, C.

$i = 1$.

Hanoi(A, B, C, 1) bildet so

„Scheibe 1 von A nach B". Vertauschungen

von A, B, C werden analog behandelt.

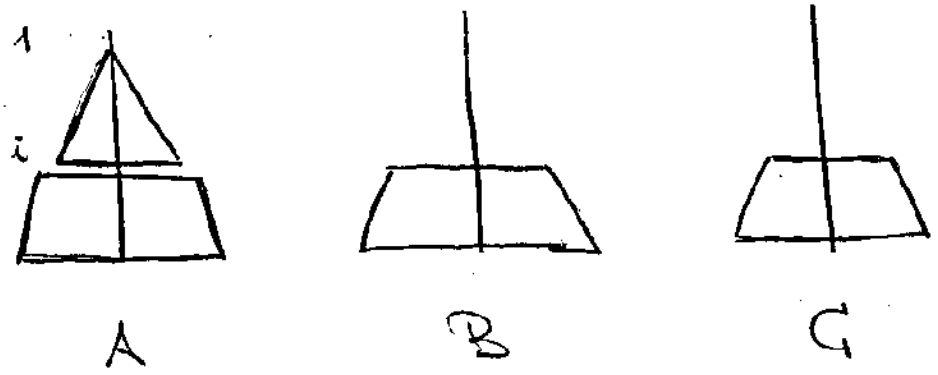
Ind.-Schluß: $i > 1$. Nach Ind.-Vor. gilt die Behauptung bis $i-1$ und alle Vertauschungen von A, B, C.

Betrachten wir den Aufruf

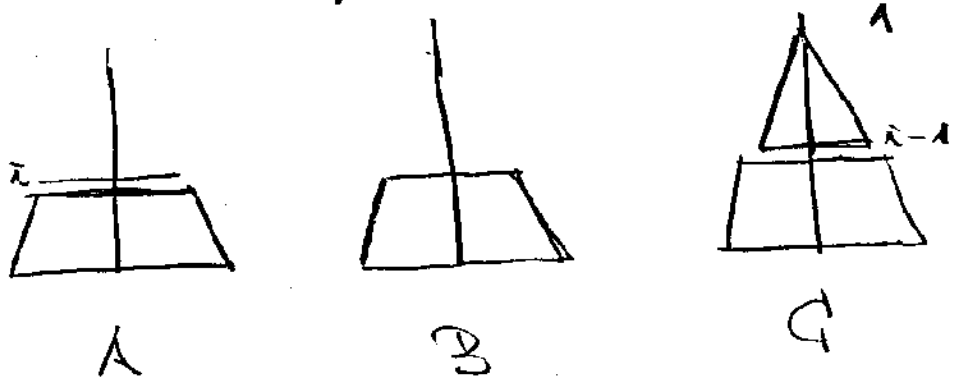
Hanoi (A, B, C, i). Da $i \geq 1$ ist liefert das

Hanoi (A, C, B, i-1);
„schreibe i von A nach B)";
Hanoi (C, B, A, i-1);

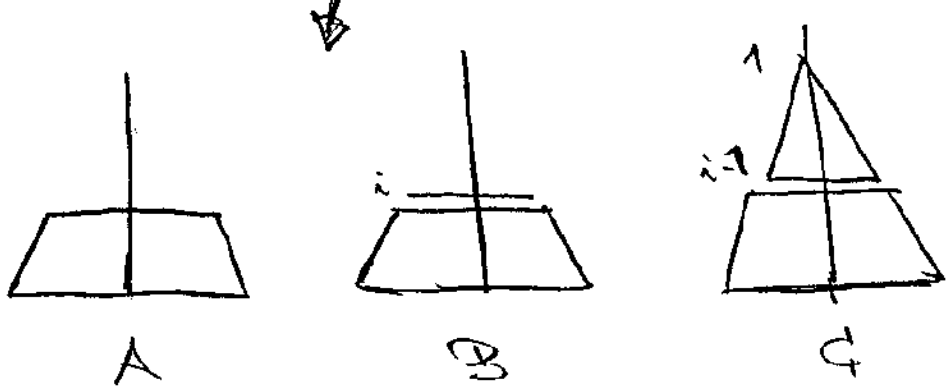
Das liefert eine regelkonforme Folge von Umsetzungen:



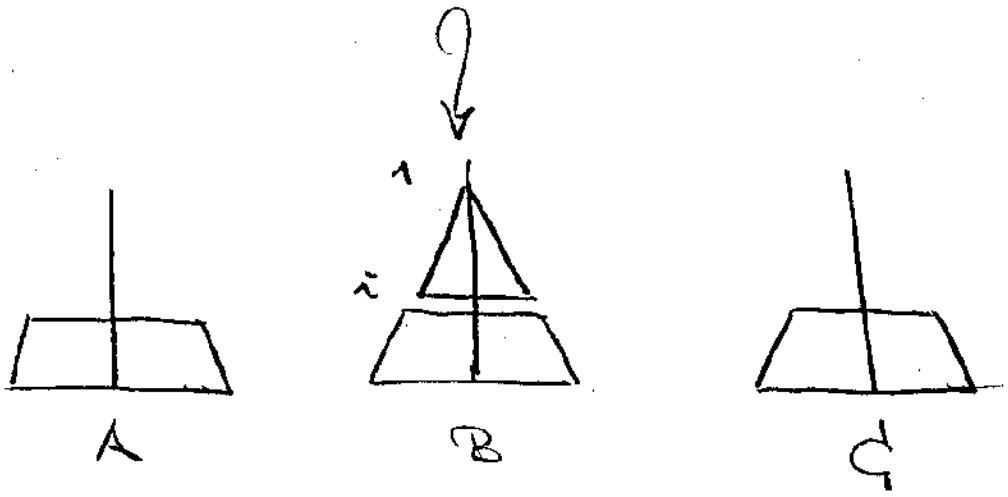
Hanoi(A, C, B, i-1)
rekursiver Aufruf
Id.-Voo. Bez. Verbindung



„2. i von A nach B"



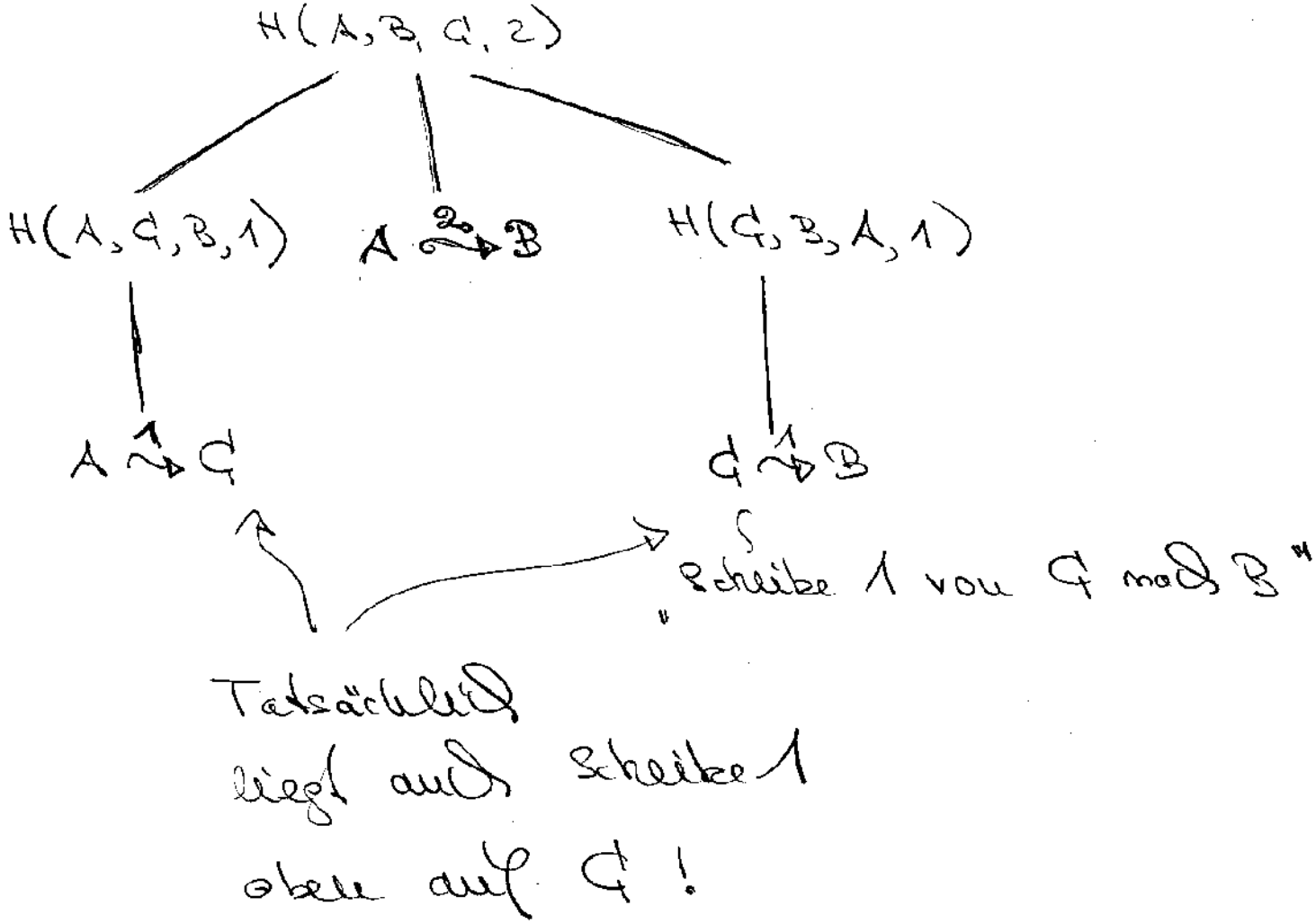
Hanoi(C, B, A, i-1)
Sud.-Voo.



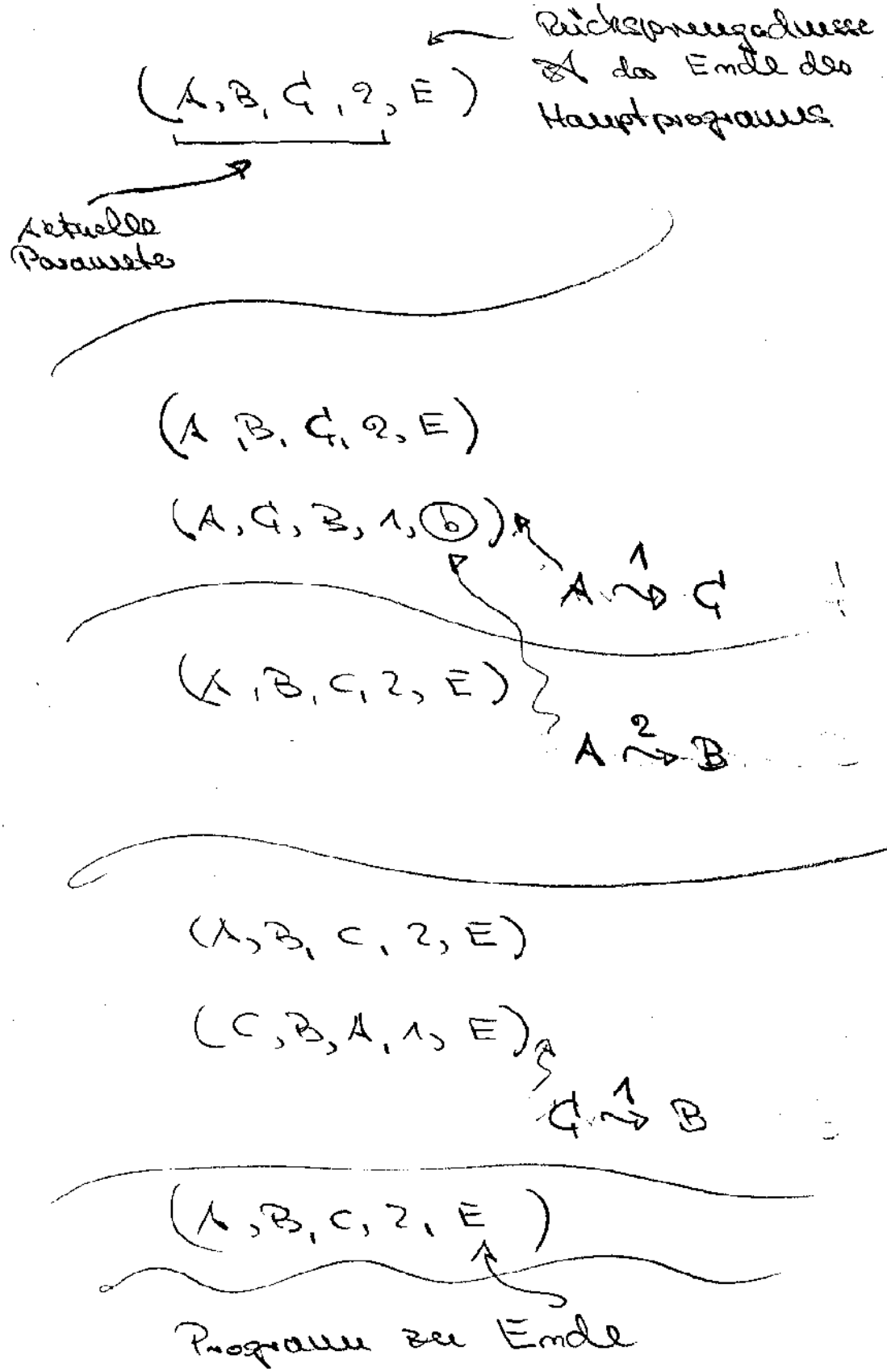
Man beachte, daß die Ind.-Vor. für jede (!) mögliche Situation gilt, also jede Situation in der alle unteren Schieber $\geq i-1$ sind.

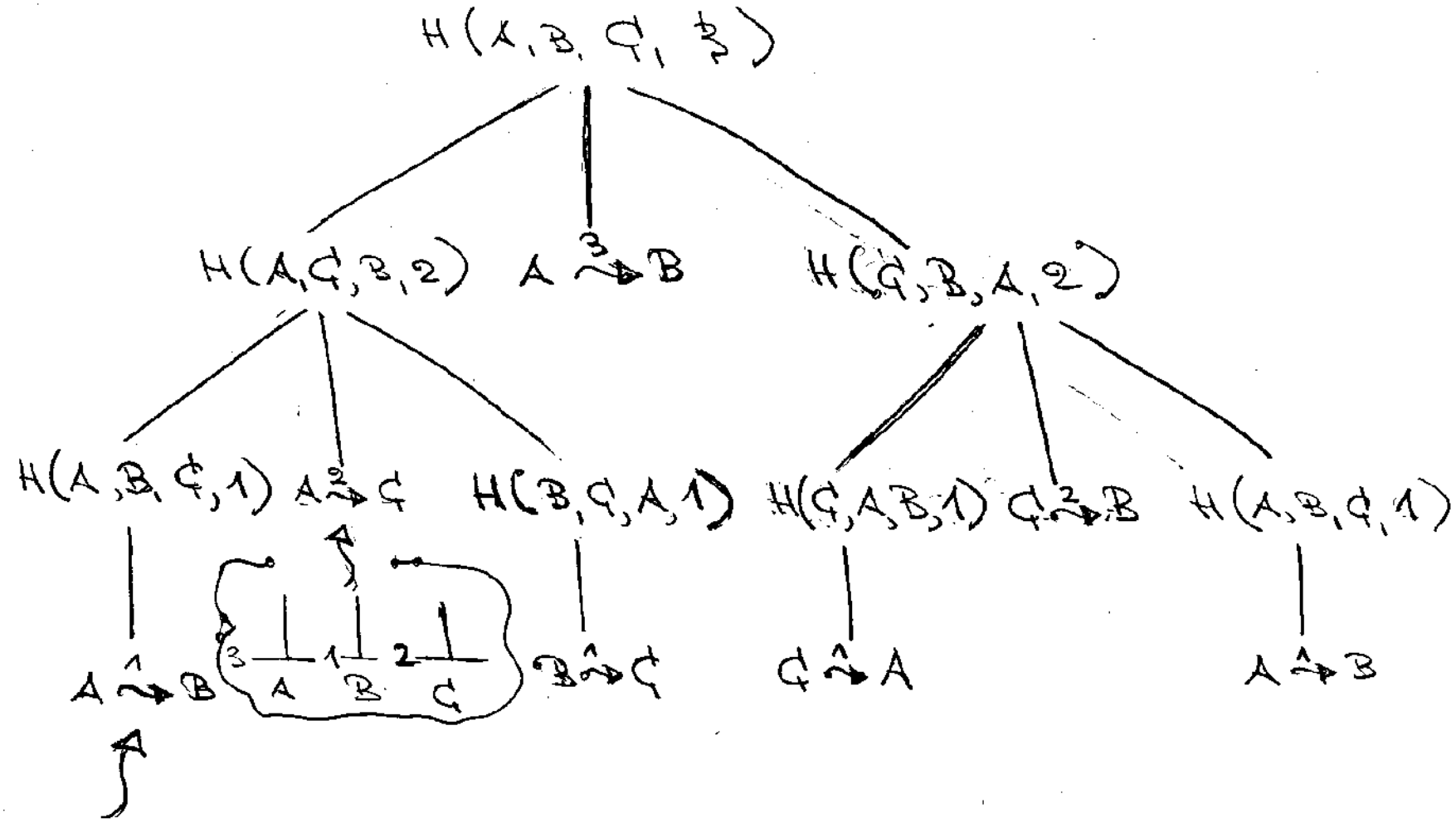
Programm in Hanoi.java.

Der Lösungsbaum



Verfolgen des Laufzeitkellers bei n = 2





Das wird als erstes ausgeführt:

Postorder Beschreibung des Baums.

Laufzeitbeispiel zum letzten Beispiel mit Programm von S. 28.

Programmende

(A, B, C, 3, E)

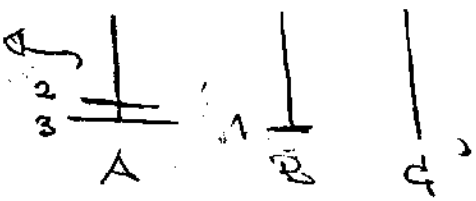
(A, B, C, 3, E) Rekursion von H(1, B, C, 3)

(A, C, B, 2, b) Rekursion von H(1, C, B, 2)

(A, B, C, 3, E)

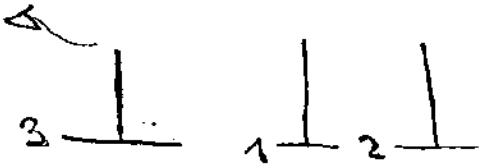
(A, C, B, 2, b)

(A, B, C, 1, b)



(A, B, C, 3, E) Aufruf mit 1.

(A, C, B, 2, b)



Wegen RA ⓑ.

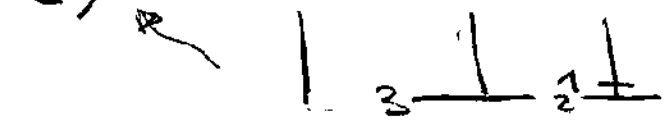
oben.

(A, B, C, 3, E)

(A, C, B, 2, (b))

(B, C, A, 1, E)

(A, B, C, 3, E)

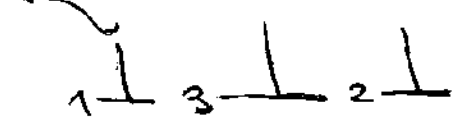


Nog. 2A (b)

(A, B, C, 3, E)

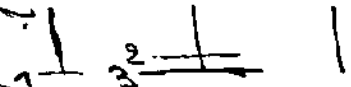
(C, B, A, 2, E)

(C, A, B, 1, (b))



(A, B, C, 3, E)

(C, B, A, 2, E)



8.38

(A, B, C, 3, E)

(C, B, A, 2, E)

(A, B, C, 1, E)

(A, B, C, 3, E)

(C, B, A, 2, E)

(A, B, C, 3, E)

Programm Ende.

8.40

Applets (von application) erlauben die einfache Programmierung von Zeichnungen in Java. Beispiele in

Gruesse.java

Gruesse.html

Übersetzen mit

javac Gruesse.java

Aufrufen mit

appletviewer Gruesse.html

Hier die Programme

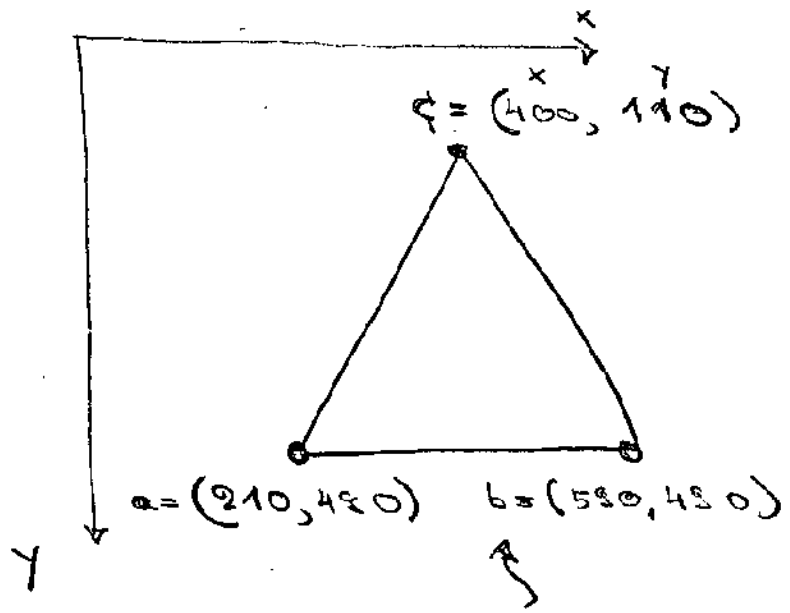
Sierpinski.java

Sierpi.java.

Das Sierpinski Dreieck.

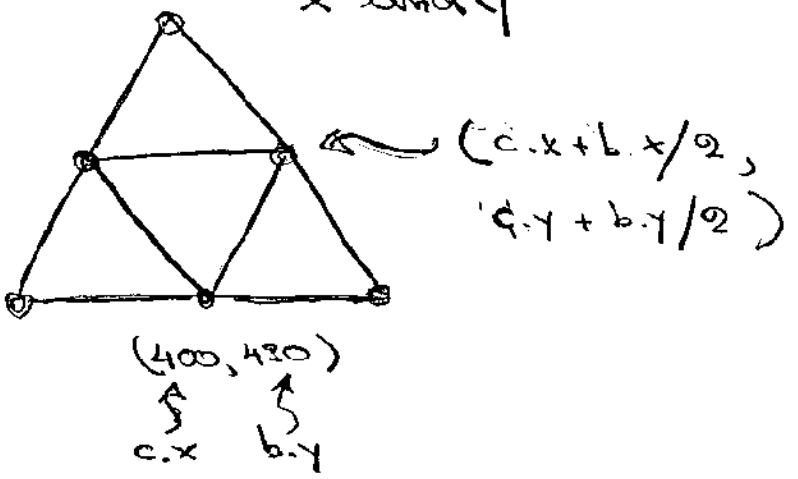
Level 0 : Ein einfaches Dreieck.

Liste der Programmzeilen (pro):

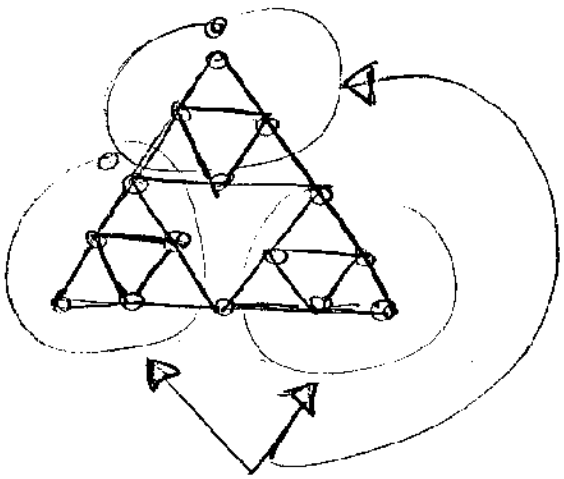


Punkt ist Objekt einer Klasse mit Komponentenvariablen x und y

Level 1 :



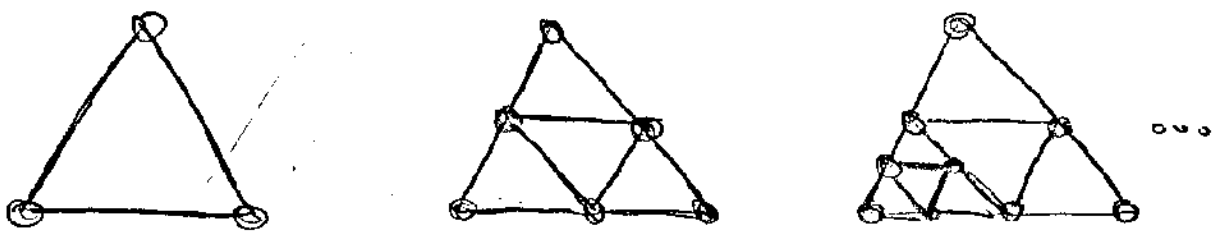
Level 2



Hier rekursiv weiter.

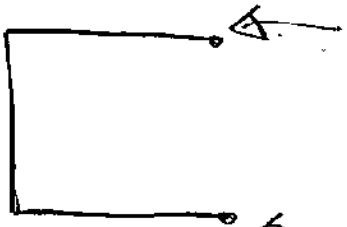
Formale Definition:
§-Dreieck von level 0 ist Dreieck.
Von level $n \geq 1$ ist Dreieck mit 3 §-Dreiecken von level $n-1$.

Die Methode triangle hat 3 rekursive Aufrufe von triangle. Der Ablauf der Zeichnung geht so:



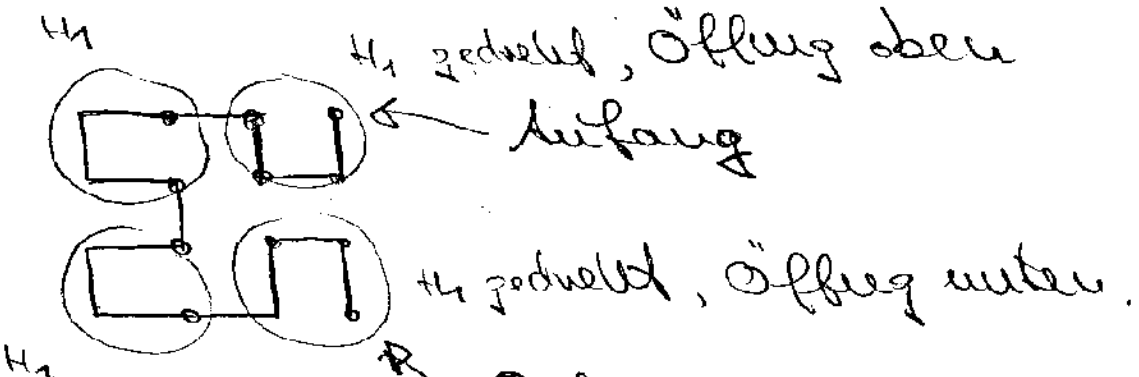
Hier weiter.

Ein anderes Beispiel dieser Art ist die Hilbert Kurve. Diese besteht aus folgendem Basiselement:



H_1

Daraus dann



H_2

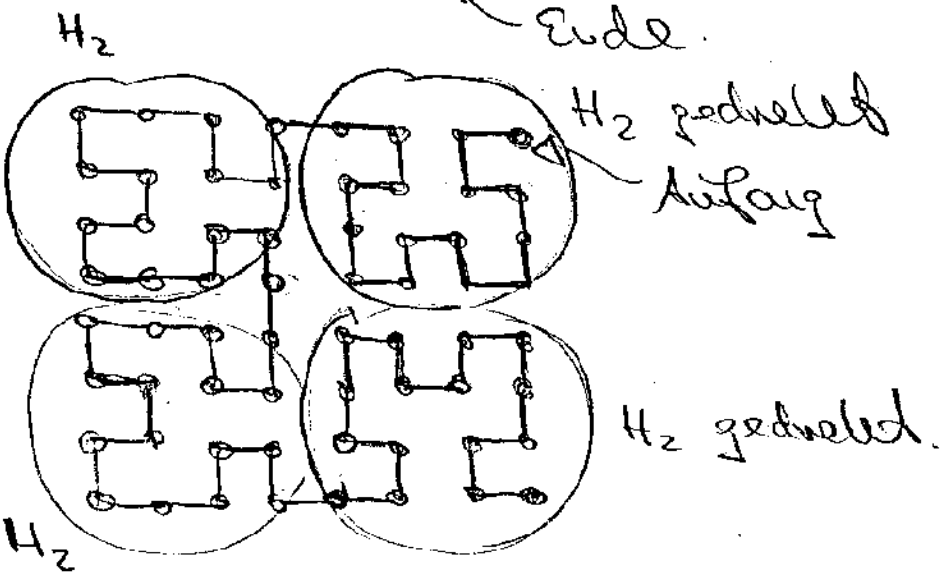
Dann bekommen wir:

← Anfang

H_2 H_2 öffnen dann

H_2 H_2 öffnen wieder

← Ende



H_3.

So geht es weiter:

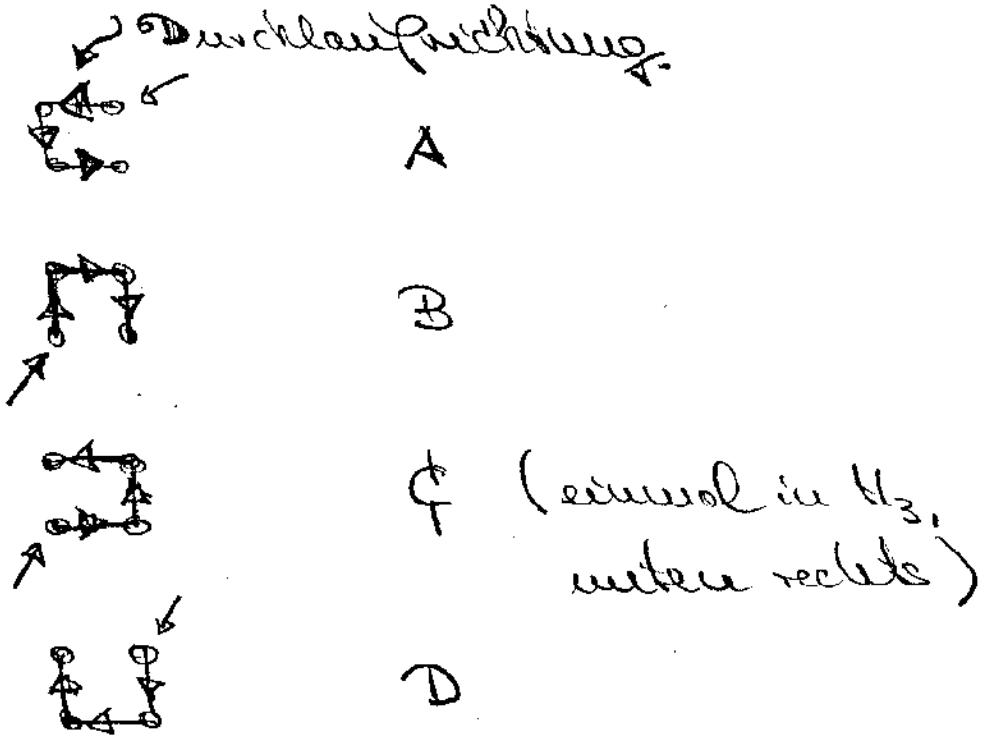
H_3 H_3 ← H_4 H_4 ←

...

H_3 H_3 → H_4 H_4 →

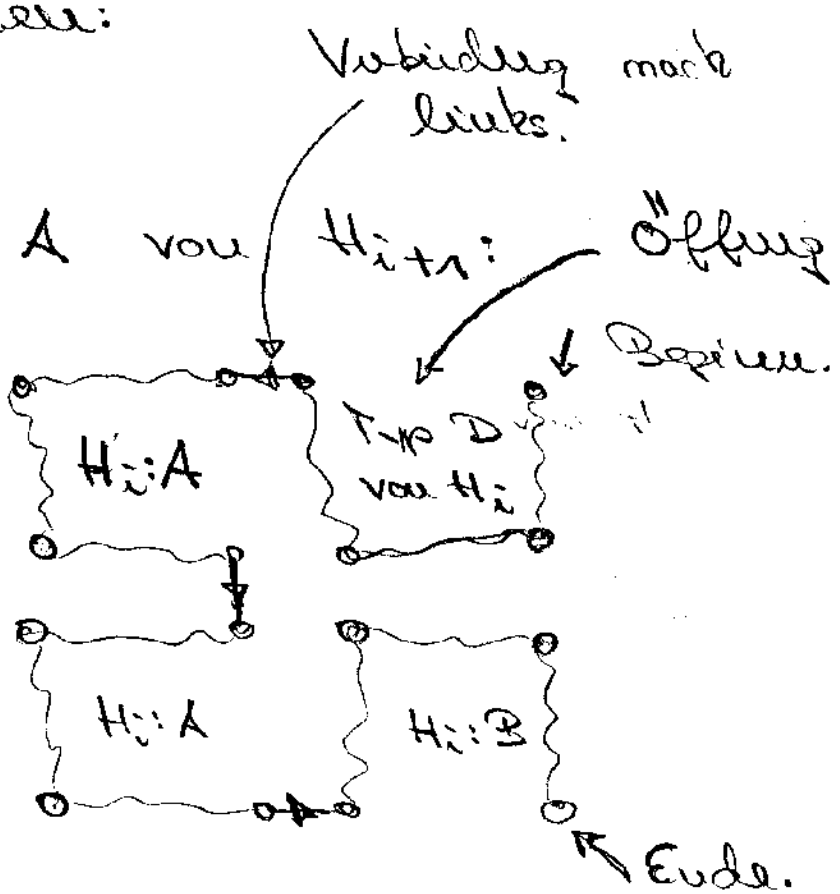
H_4 H_5

Schauen wir uns H_3 an, dann haben wir 4 Typen A, B, C, D von H_1:

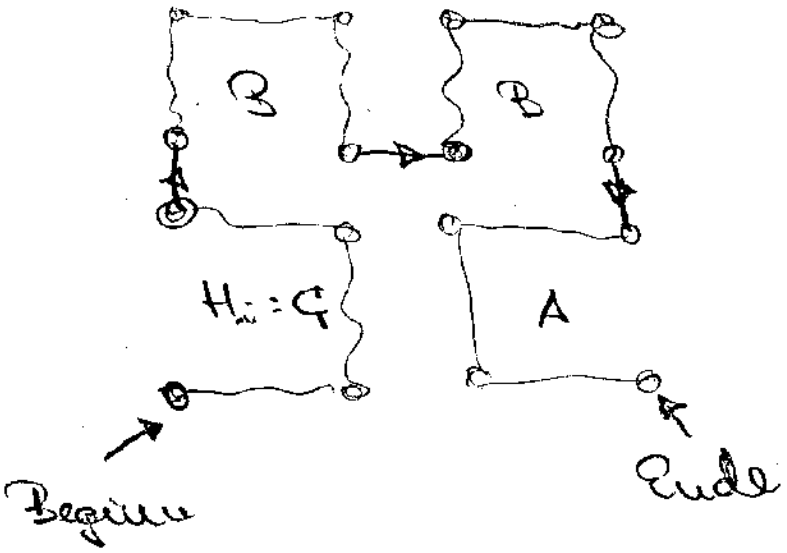


Analog definieren wir 4 Typen A, B, C, D von $H_2, H_3, \ldots$.

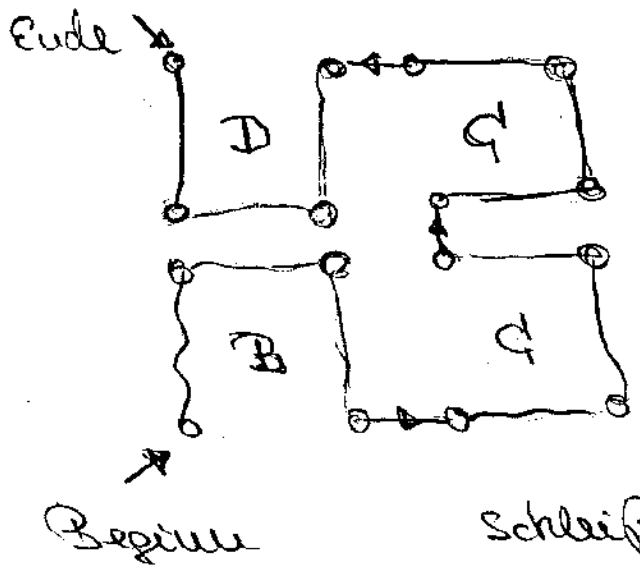
Dann ergibt sich folgende Vorschrift H_{i+1} aus H_i zu bauen:



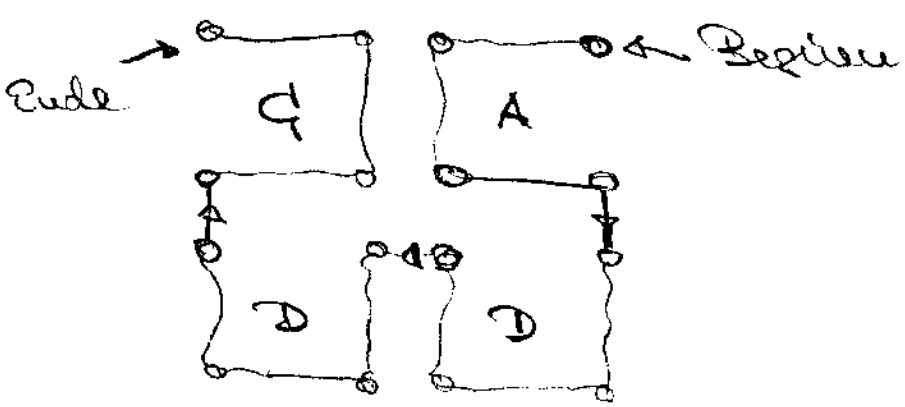
Typ B von H_{i+1}:



H_{i+1}: C



schließlich H_{i+1}: D



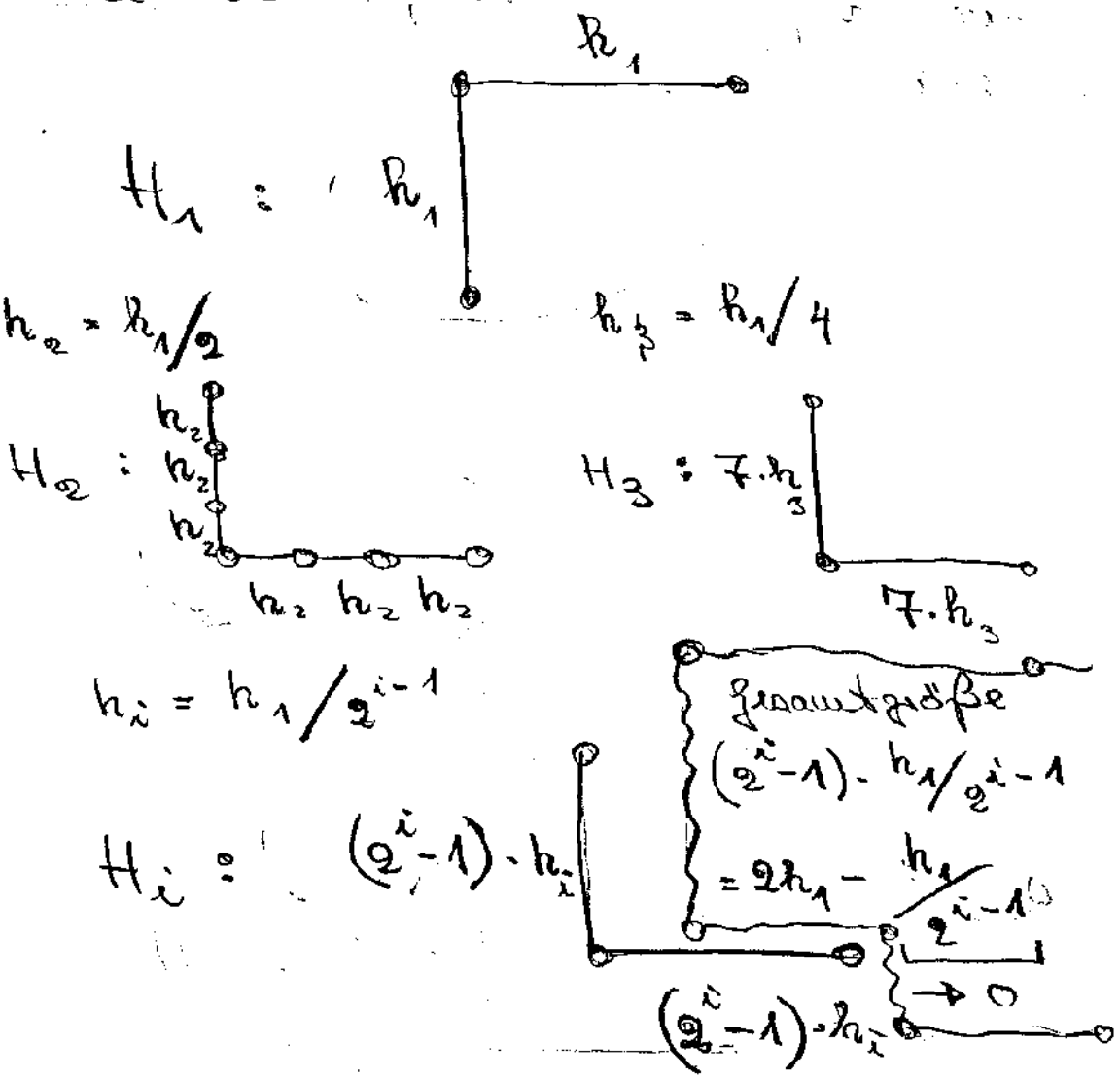
Wir zeichnet man das ganze T_i?

Sei $h = h_1$ gegeben, Länge bei level = 1.

Dann bei level = i ist

$= h_i = h_1 / 2^{i-1}$.

Dann haben wir:



Induktiv: $(2^{i-1} - 1) + (2^{i-1} - 1) + 1 = 2^i - 1$.

Dazu das Programm Hilbs.java.

Verbesserung in Hilbsverb.java.

Methoden zum Zeichnen der Linien. Die Basisklänge $h = h_i$ wird mit eingegeben.

Diese Hilbertkurve hat eine Bedeutung. Für i groß, d.h. $i \to \infty$ ist H_i eine Kurve, die das Quadrat mit Seitenlänge $2h_1$

- ohne Absetzen die Fläche (stetig)
- ohne Überkreuzungen (injektiv)
- vollständig (surjektiv)

ausfüllt.

Die kombinatorische Suche, d.h.
Suche von kombinatorischen Objekten
wie endlichen Mengen, 0-1-Vektoren
oder abzählbaren mit bestimmten
Eigenschaften ist ein weiterer
wichtiger Anwendungsbereich der
Rekursion. Dazu behandeln wir
zunächst die algorithmische Erzeugung
aller möglicher Objekte.

Erzeugung aller (!) 0-1 Vektoren
der Länge n.

Array x mit Länge n, also
Einträgen $x[0], \ldots, x[n-1]$.

Rekursives Muster:

Alle 0-1-Folgen der Länge n ergeben sich als

0 „alle der Länge $n-1$",

gefolgt von

1 „alle der Länge $n-1$".

Ist $n = 0$ so haben wir die leere Folge.

Die Rekursion geht dann so weiter:

Die Folgen

0 „alle der Länge $n-1$"

ergeben sich als

00 „Länge n-2"

gefolgt von

01 „Länge n-2".

Also im allgemeinen haben wir es mit beliebigen Anfangsstücken zu tun: Alle die mit

Das sind d Stück.

$b_0\, b_1 - b_{d-1}$ Alle die Länge n-d

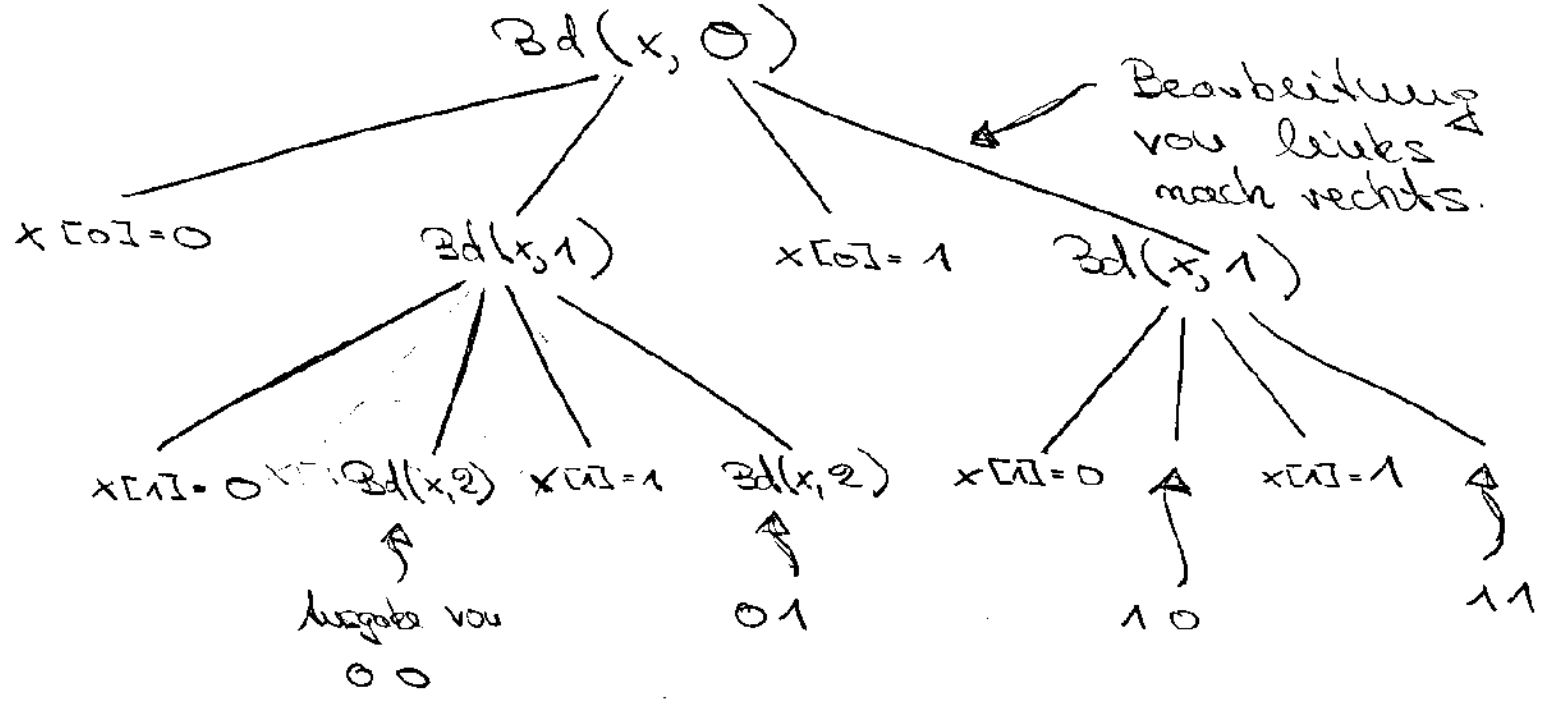
ergeben sich als

$b_0\, b_1 - b_{d-1}\,0$ „Alle von n-(d+1)"

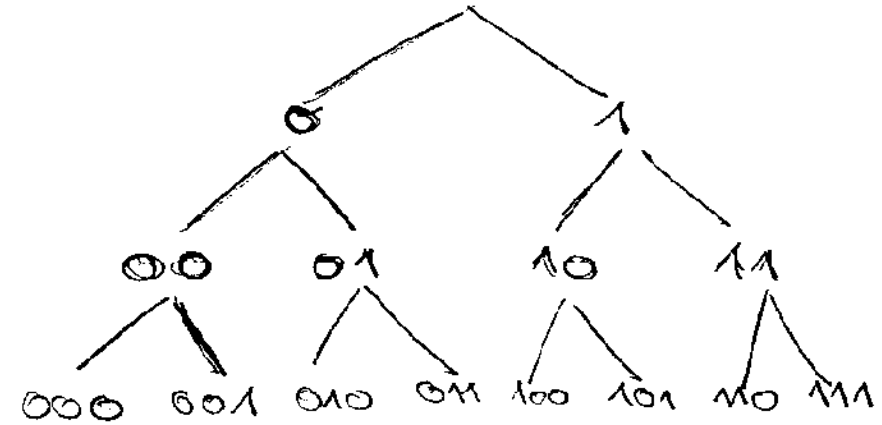
gefolgt von

$b_0\, b_1 - b_{d-1}\,1$ „Alle von n-(d+1)".

Programm in Binom.java.



Das Muster ist bei x.length = 3:



Vollständiger binärer Baum der Tiefe 3:
o Tiefe 0, Tiefe 1
Δ Tiefe 2, ...

Schwierig ist es, alle Permutationen über n Elementen, etwa $1, \dots, n$ aufzulisten. Nehmen wir - rekursiv - an, wir könnten bereits alle Permutationen über $n-1$ Elementen auflisten. Dann können wir alle Permutationen über n Elementen folgendermaßen bekommen:

Beginne mit der Permutation

$$1\ 2\ 3\ 4 \dots n.$$

1. Erzeuge rekursiv alle Permutationen der Art

$n-1$ Elemente

1 Permutation auf $2 \dots n$

2. Alle der Art

2 Permutationen auf 1,3-,4

$\S$

n. Alle der Art

n Permutationen auf 1 ... n-1.

Also Methode

Perm(int[] x, int d)

erzeugt beim Aufruf Perm(x,d)

alle Permutationen der Art

x[0] x[1] ... x[d-1] Alle Permutationen

der Elemente

x[d], ..., x[x.length-1].

Also Methode

```
Perm(int[] x, int d) {
if  d >= x.length { Ausgabe(x);
                    return;
                  }

int i = d;
while (i < x.length) {
    Vertausche x[d] mit x[i];
    Perm(x, d+1)      // Zu der while-Schleife
    i++               // Perm(x;d+1) n-d-mal
}                     // Jeweils anderes Feld x.
```

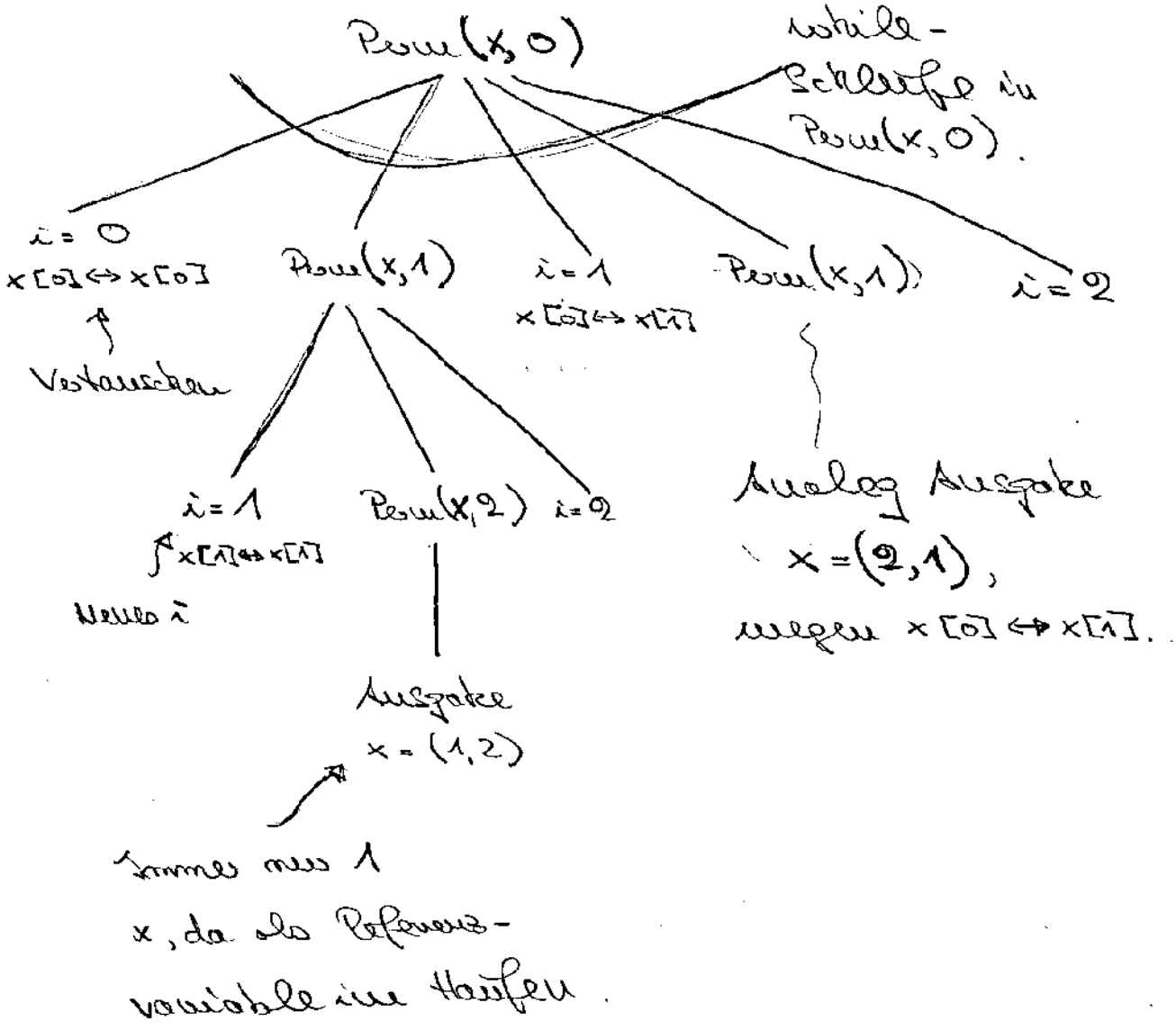
Startpermutation ist dann

$x[0] = 1,\ x[1] = 2,\ \ldots,\ x[n-1] = n$

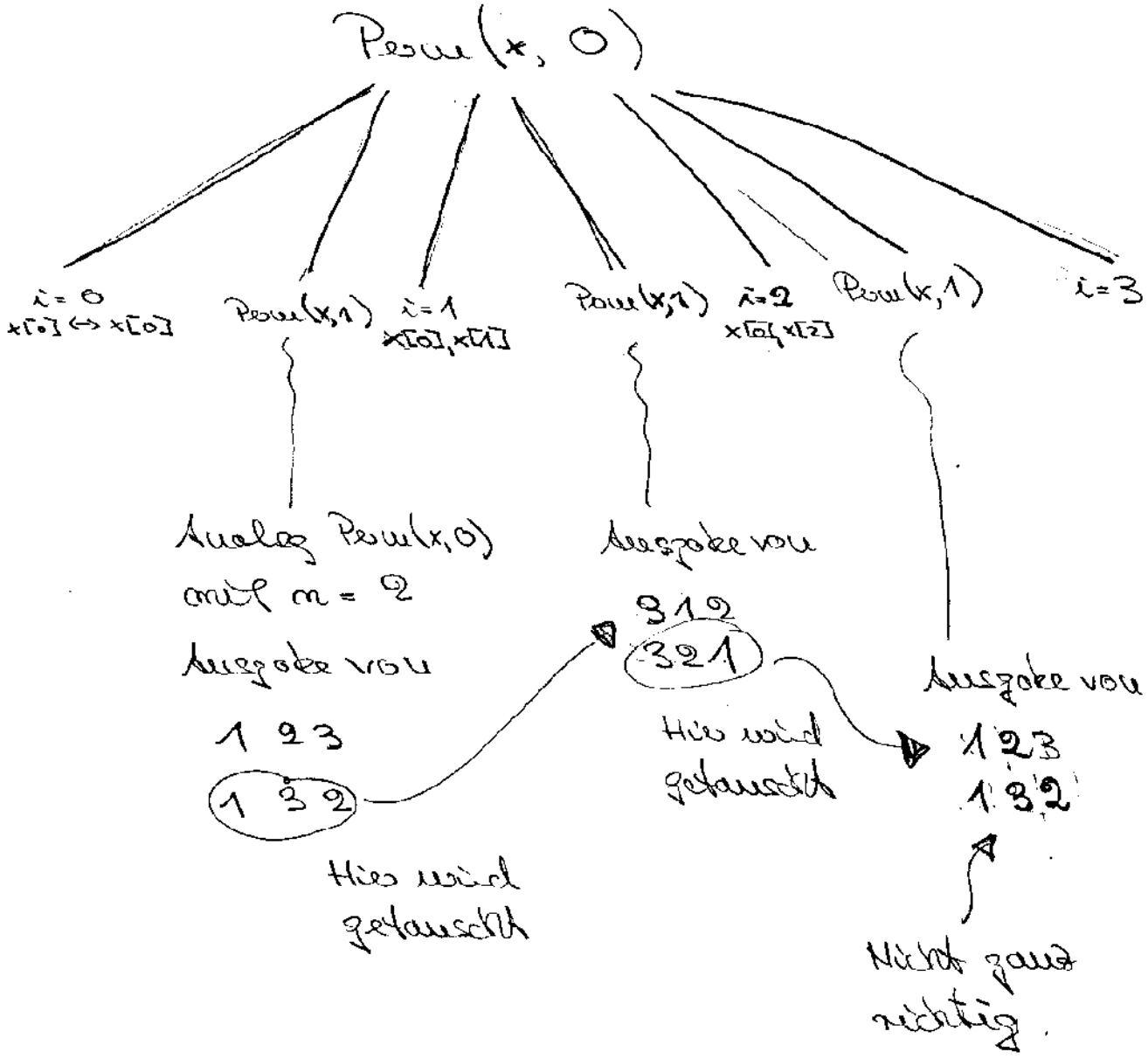
Startaufruf ist dann

```
Perm(x, 0)   // Führt zu n Aufrufen
             // von Perm(x, 1).
```

Aufrufbaum bei $n = 2$:



$n = 3$



Das Austauschen muß sich immer auf die Ausgangspermutation beziehen, den aktuellen Parameter bei Aufruf.

Die Modifikation in der while-Schleife:

```
while (i < x.length) {

    Vertausche x[d] mit x[i]

    Perm(x, d+1)                    Neu.

    Vertausche x[d] mit x[i]

    i++

}
```

Dann können wir induktiv über x.length − d argumentieren, daß am Ende eines Aufrufs Perm(x,d) wieder der Ausgangs-x vorliegt.

$x.length - d = 0$, dann $d = x.length$

dann gilt die Behauptung, da

nichts an x gemacht wird.

$x.length - d \geq 0$, dann $d \leq x.length$.

Dann folgt die Behauptung

mit der I.d.-Vor., da die

lokale Schleife für $i = d$ bis

$i = x.length - 1$ ausführt:

$x[d] \leftrightarrow x[i]$

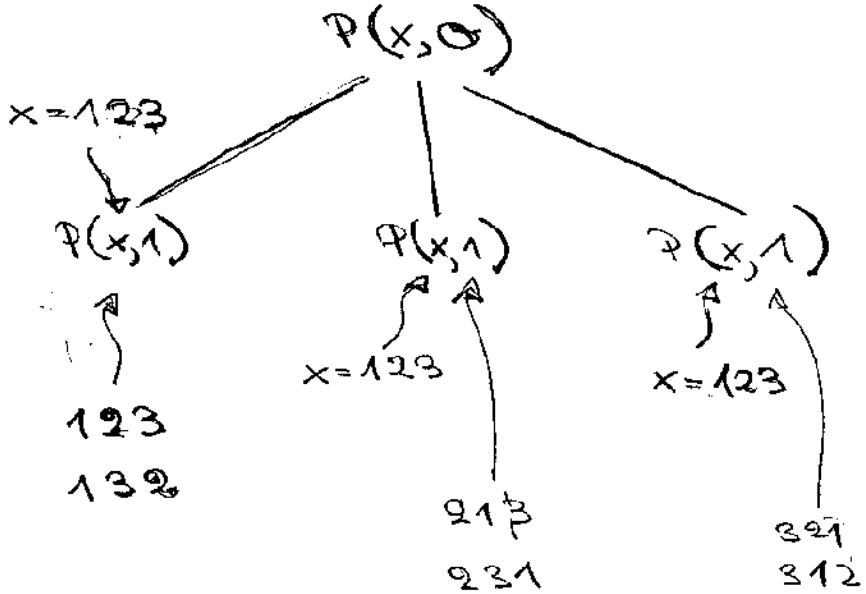
Perm$(x, d+1)$

$x[d] \leftrightarrow x[i]$

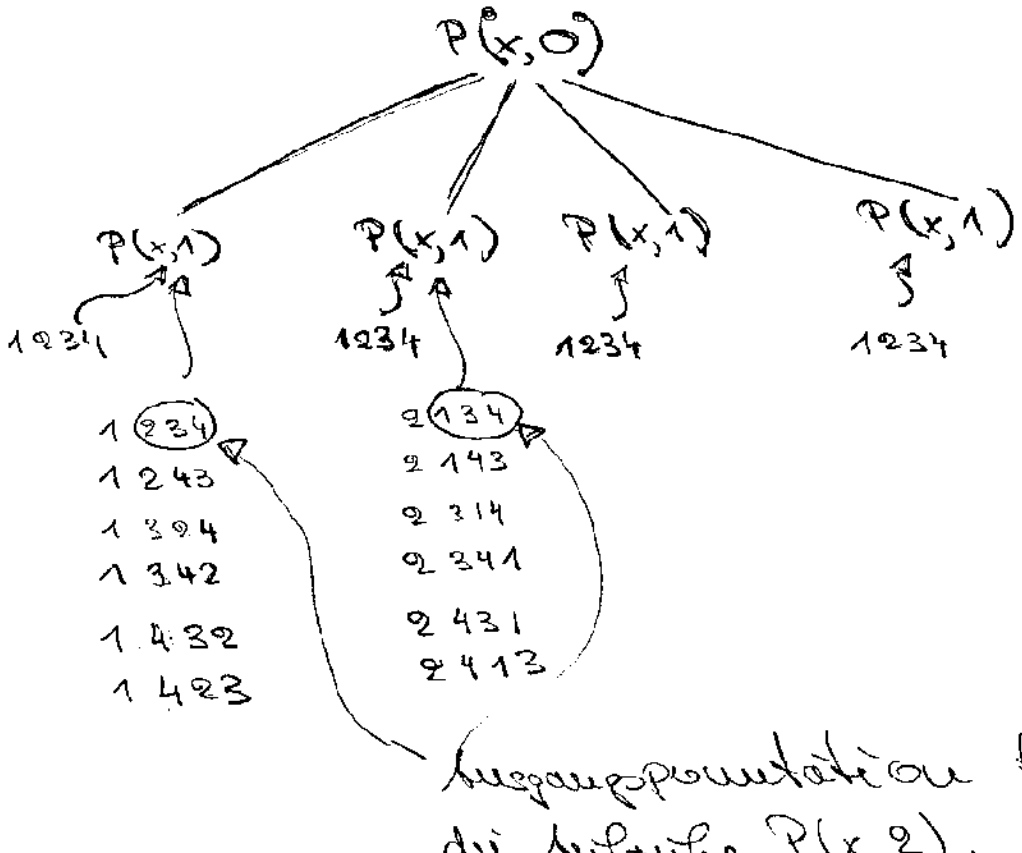
und nach I.d.-Vor. jedesmal wieder

die Vertauschpermutation π vorliegt! ∎

$n = 3$



$n = 4$



Ausgangspermutation für die Aufrufe $P(x,2)$.

Ein Standardbeispiel der kombinatorischen Suche ist das 8-Damenproblem:

Setze 8 Damen auf ein Schachbrett, so daß sie sich nicht gegenseitig „schlagen".

Feld brett der Länge 8. Einträge 1, ..., 8.

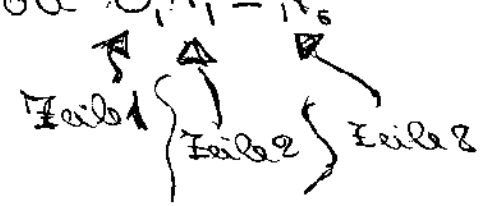
brett[0] = 4 $\Leftrightarrow$ Dame in Spalte 1 in Zeile 5.

brett[1] = 7 $\Leftrightarrow$ Spalte 2 Zeile 8.

⋮

brett[7] = 0 $\Leftrightarrow$ Spalte 8 Zeile 1.

Lösungskandidat: Permutation von $0, 1, \ldots, 7$



Lösungsraum: Alle Permutationen auf $1 \dots 8$.

$$8! = \underbrace{8 \cdot 7 \cdot 6 \cdot 5}_{4 \text{ Faktoren}} \cdot 4 \cdot 3 \cdot 2 \cdot 1 \geq 4^4 = 2^8$$

Allgemein

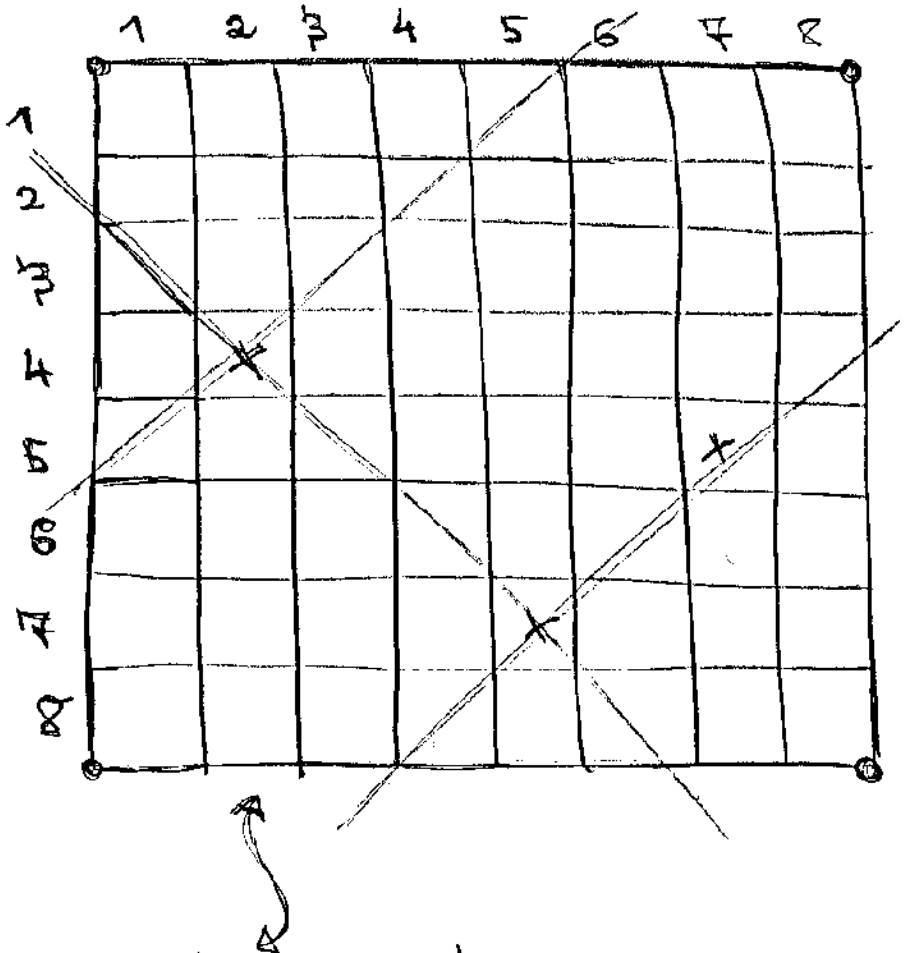
$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{((\log n) - 1) \cdot \frac{n}{2}} > 2^n .$$

Wie testen wir, ob eine Permutation Lösung ist?

Gemeinsame Zeilen nicht möglich, da stets Permutation ist.

Gemeinsame Spalten ebenfalls nicht.

Diagonalen



brett[1] = 3

Diagonale ↘ :

brett[l] = k, dann

l-1	k-1
l-2	k-2
l-3	k-3

l+1	k+1
l+2	k+2
l+3	k+3

Dieser Teil reicht schon.

Berechne bis spalte = l.

```
for(int i = spalte - 1, j = brett[spalte] - 1;
                    i >= 0 ; i-- ; j-- )

    if (brett[i] == j ) return true
```

Ebenso Diagonale / nach linke unten:

(durch Feldgrenzen.)

```
for(int i = spalte - 1, j = brett[spalte] + 1 ;
                    i >= 0 ; i-- ; j++ )

    if (brett[i] == j ) return true.
```

Invariante für die erste Schleife:

Falls nicht return true nach l'ten Lauf, sind $l \geq 1$:

brett[sp-1] ≠ brett[sp] - 1 und

brett[sp-2] ≠ brett[sp] - 2, ... und

brett[sp-l] ≠ brett[sp] - l.

Falls return true, dann ... aber

brett[sp-l] = brett[sp] - l.

Entweder mit $l = sp$, das entspricht $i = 0$.

Methode, die für gegebenes
brett prüft, ob kein Schlagen
möglich ist.

```
public static boolean bedroht(int[] brett)
  int spalte = 0;
  for (spalte = 0; spalte < brett.length; spalte++)
    for (int i = spalte - 1, j = brett[spalte] - 1;
         i >= 0; i--; j--)
      if (brett[i] == j) return true;
  // Test nach links oben.
```

Testet auch die Feldgrenzen.

Hier der Test nach
links unten /

Invariante der äußeren Schleife:

Nach k-tem Lauf ohne return true sind die ersten k Spalten noch links jedenfalls gut.

Quintessenz: Am Ende sind alle Spalten noch links gut, also insgesamt gut.

Programm ADE.java

Permutationen auf $0,\dots,7$ drucken.

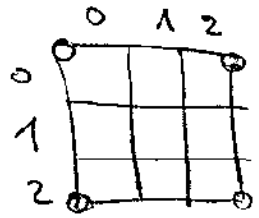
Jede Permutation testen.

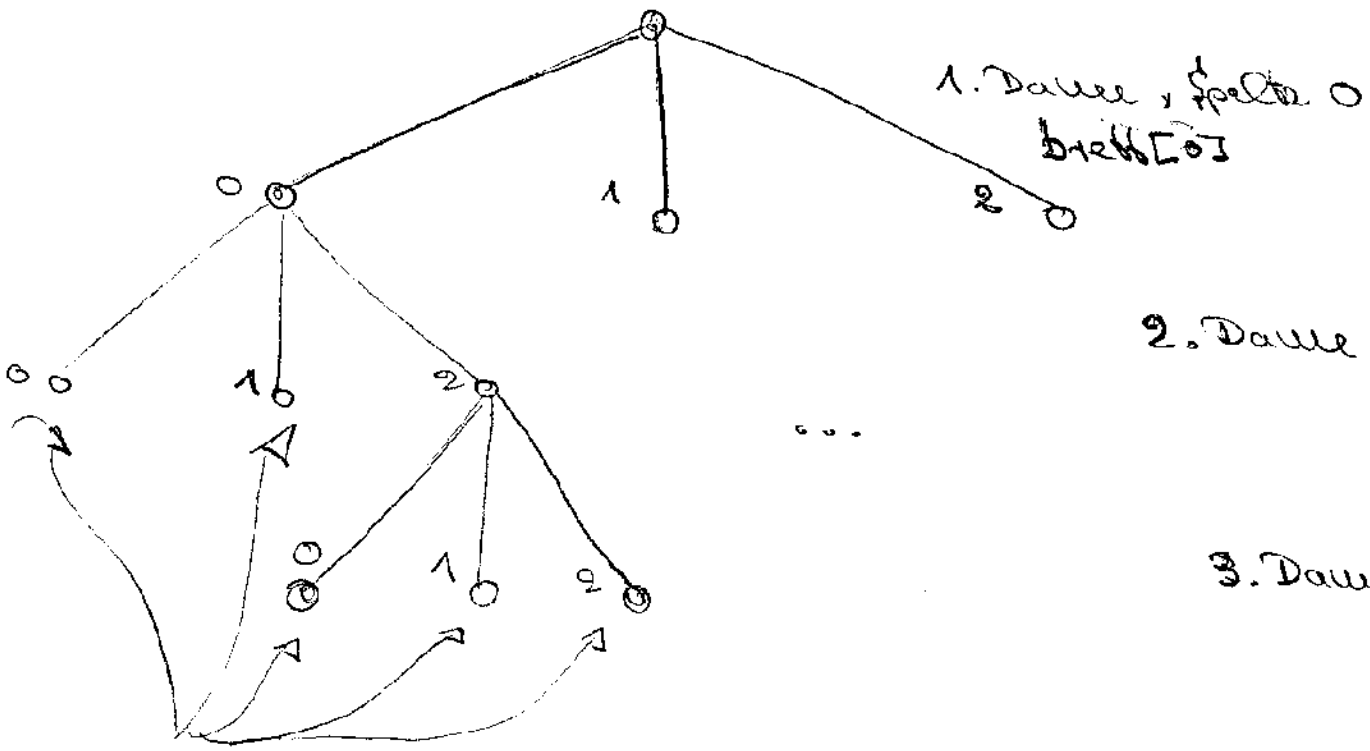
$n = 8$, Einhöge auf $0, \dots, 7$ ergibt 92 Lösungen. ✓

Das Programm ADE.java testet jede der n! Permutationen des Lösungsraums einzeln. Die Methode des backtracking entwickelt die mögliche Lösung schrittweise und versucht so früh wie möglich zu erkennen, wann ein Weiterentwickeln keinen Sinn mehr hat (Sackgasse, dann backtracking). Dann werden alle Permutationen, die sich aus dieser Sackgasse ergäben in einem Schritt als Nicht-Lösungen erkannt.

Im Prinzip kommen wir einen Postordnungsdurchlauf durch den folgenden Baum:

$n = 3$





Zum Beispiel wird 012 gar nicht betrachtet. Ebenso 120 nicht ...

Bei beliebigem n benötigt
man eine wesentlich größere
Expansion im Vergleich zu ADE.java

01 — 01 „alle (n-2)! Permutationen"

021 — 021 „alle (n-3)! Permutationen"

⋮

Wie programmiert man das?

```
public static void
        suche(int[] brett, int spalte) {

if (spalte == brett.length) {
                Ausgabe(brett);
                return }
                // Bis i = brett.length!
for(int i=0; i < brett.length; i++) {
    brett[spalte] = i; // Heißt, Dame an
                // spalte in Zeile i.
        if (! bedroht(brett, spalte))
                suche(brett, spalte+1)
                // Nächste Dame.
}
```

bedroht(brett, spalte) testet nach links in der Zeile und den Diagonalen.

Korrektheit der Rekursion. Dazu zeigen wir: Ist

brett[0], ..., brett[spalte-1]

eine mögliche Teillösung, so liefert

suche(brett, spalte)

genau diejenigen Lösungen, die Fortsetzung dieser Teillösung sind

Induktion von oben. ← Von den Blättern zu der Wurzel des Baums

spalte = brett.length, dann gilt die Behauptung.

8.7.9

spalte < brett.length . wo
kommen in die for-Schleife.
Setzen die 'brett[spalte]' auf
0, 1, 2, —, brett.length - 1.
Testen jedesmal, ob zulässige
Teillösung. Behauptung folgt
mit Ind.-Vor. Haben wir
keine zulässige Teillösung
wird auch nichts ausgegeben.

Laufzeitunterschied von

Achtdamen.java

und

ADE.java !

Unsere Programme geben immer
alle Lösungen des im allgemeinen
n-Damen Problems aus. Wir
ist das Programm Schritt für
Schritt zu modifizieren, so daß nur
eine Lösung ermittelt wird (Effizienz).
Das ist das Programm im Buch
auf S. 185.

Die Unterschied ist, daß nun
public static boolean (Neu:) setze(int i, int spalte)
definiert wird.

Ist eine Lösung gefunden wird
true zurückgegeben (Zeile 32).
In dem darüberliegenden Verzweig

wird dann direkt wieder getestet,
ob der Befund nun ausgeglichen hat
und es wird wieder gleiche Luft
zurückgegeben usw ...

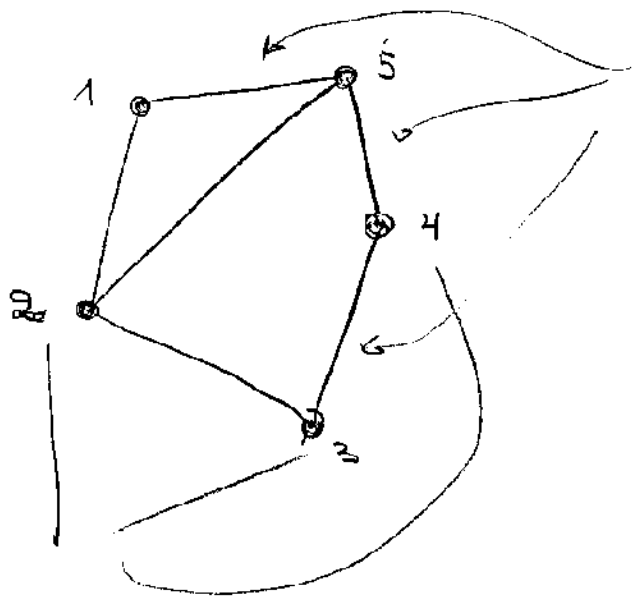
Als letztes Suchproblem: Das

- Färbungsproblem

bei

- Graphen.

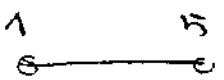
Typische Graphen sind



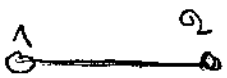
Kante (edge)

Knoten (node, vertex)

Formale Beschreibung



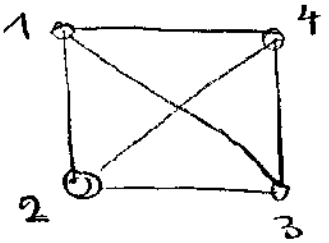
$\{1,5\}$



$\{1,2\}$

Inzidenz von Knoten

⋮



Vollständiger Graph (Clique)

mit 4 Knoten.

Menge der Kanten

$$= \{ \{1,2\}, \{2,3\}, \{3,4\}, \{4,1\}, \{1,3\}, \{2,4\} \}$$

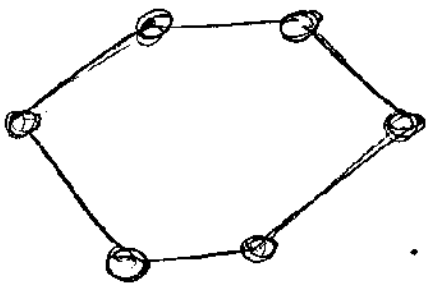
Also 6 Kanten.

$$\binom{4}{2} = \frac{4 \cdot 3 \cdot 2 \cdot 1}{2 \cdot 2} = 6$$

Bei n Knoten sind $\binom{n}{2}$ Kanten möglich, aber nicht mehr.

Hier noch ein Graph mit

#Knoten = #Kanten.



<u>Definition</u> (k-Färbbar)

Sei $k \in \mathbb{N}$, $k \geq 0$. Ein Graph mit

- Knotenmenge V

und zugehöriger

- Kantenmenge E

ist k-Färbbar

gdw.

es gibt eine Abbildung

$$f : V \longrightarrow \{1, \dots k\}$$

f: Die Färbung des Knotens

$\{1, \dots k\}$: Die Farben (alternativ auch $\{0, \dots, k-1\}$)

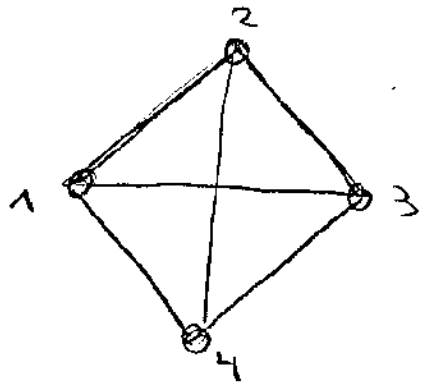
so daß gilt:

$\forall \{v,w\} \in E$, so ist $f(v) \neq f(w)$. □

"Benachbarte" Knoten haben also die gleiche Farbe.

8.80

$E = \emptyset$ ist 1-färbbar, alles
bei gleiche Farbe.

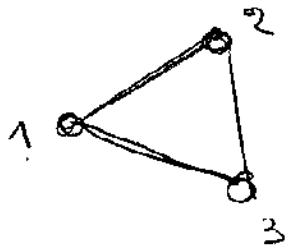


Ist 4-färbbar

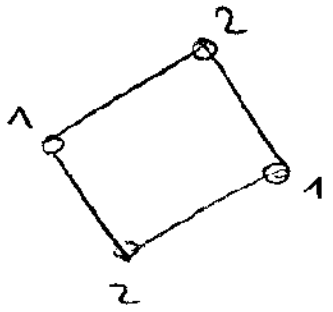
$$F(v) = 4.$$

Ist $|V| = n$, dann immer n-färbbar.

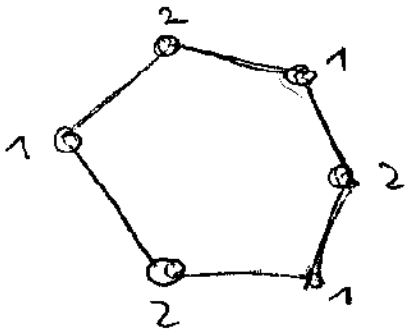
8.81



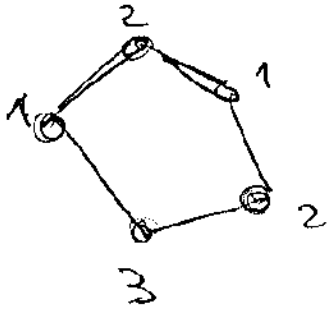
Nicht 2-, aber 3-färbbar.



2-färbbar.



2-färbbar.



Nicht 2-färbbar,
3-färbbar.

Satz

Der Graph G mit

$$V = \{1, \dots, n\}$$

$$E = \{\{i, i+1\} \mid 1 \leq i < n\} \cup \{\{n, 1\}\}$$

ist immer 3-färbbar. Er ist 2-färbbar gdw. n gerade ist.

Beweis:

$F(1) = 1$ ✓

$F(2) = 2$ ✓ — Dann $f(n-2) = 1$

$F(3) = 1$ ✓

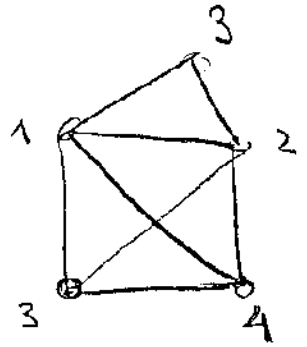
⋮

$$F(n-1) = \begin{cases} 2 & \text{falls } n-2 \text{ ungerade} \\ 3 & \text{falls } n-2 \text{ gerade.} \end{cases}$$

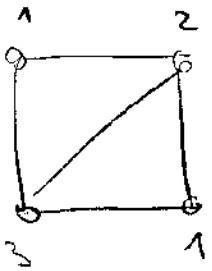
Dann $f(n-2) = 2$

□

8.83



4 - färbbar.



3 - färbbar.

Die stellt neun die

3 - Färbbarkeit eines Graphen fest.

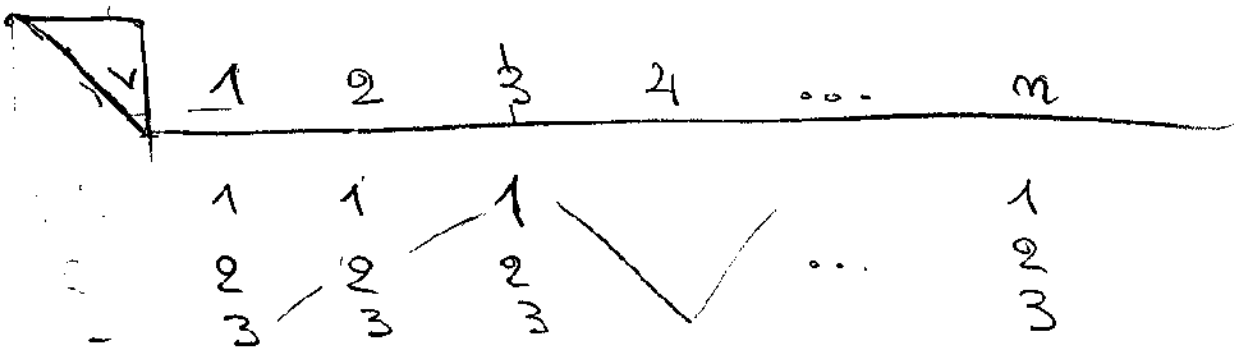
Sei G mit

- $V = \{1, \ldots, n\}$
- E beliebig.

Wie sieht hier der Lösungsraum aus? Alle prinzipiell möglichen 3-Färbungen von V sind alle Abbildungen

$$f : V \to \{1,2,3\}$$

Wieviele Abbildungen F gibt es?



Jeder „Weg" = 1 f

$3 \cdot 3 \cdot \ldots \cdot 3 = 3^n$ Wege

3^n 3-Färbungen.

Darstellung von G durch

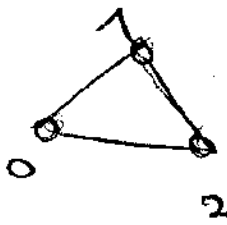
graph[n][n] der Adjazenzmatrix.

graph[v][w] = 1 $\Leftrightarrow$ $\{v,w\}$ Kante

graph[v][w] = 0 $\Leftrightarrow$ $\{v,w\}$ nicht Kante.

Natürlich graph[v][w] = graph[w][v]

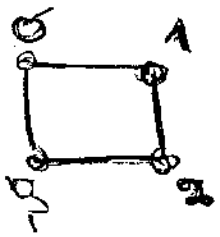
und graph[v][v] = 0.



graph

	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

symmetrisch an der Diagonalen.



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

Adjazenzmatrix

Es bietet sich wieder das Backtracking an. Wir wollen nur eine Färbung haben, falls existent, sonst Meldung, daß keine Färbung existiert.

```
public static boolean

    Faerb(int[][] graph, int[] F,
                int knoten)
        // Färben die Knoten
        // in der Reihenfolge
        // 0, 1, 2, ..., n-1.
        // F[] = partielle Färbung
        // knoten = zu färbender Knoten

if (knoten == F.length) return true // Gut gefärbt.

naechs: for(int i=1; i<=3; i++)
            Teste ob i zulässige Farbe
            für knoten ist.
            Falls nein: Nächste Farbe,
                        continue naechs.
```

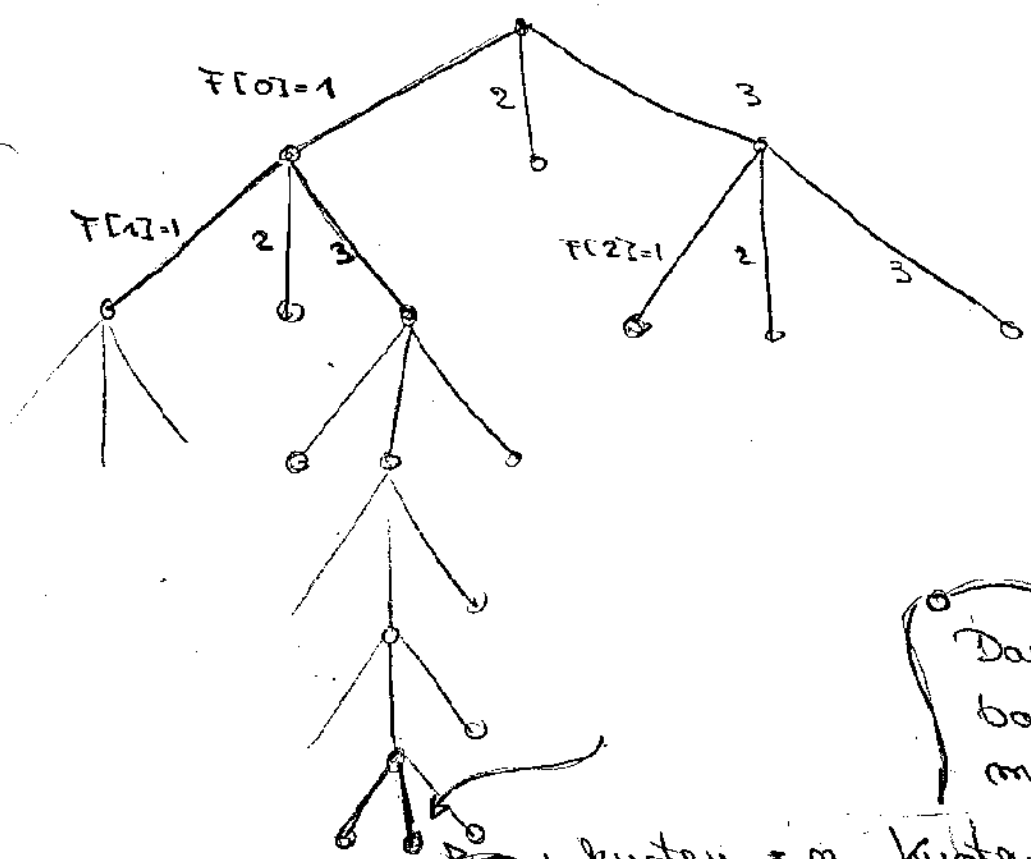
Nächsten Knoten färben.

Falls ja:

```
F[knoten] = i
test = Faerb(3, F, knoten+1)
if (test) return true;
```

return false // Heißt F ist nicht zu zulässige Färbung erweiterbar.

Prozedurbaum:



Dann kein Backtracking mehr.

knoten = n, Knotennr $0, \ldots, n-1$.

Programm DreiFach.java,

Erklärung per Hand.

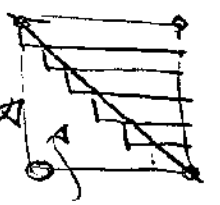
Jetzt zufällige Graphen. Was
heißt das?

Ecken.

- Knotenmenge $V = \{0, \ldots, \text{Knoten} - 1\}$

- Kantenmenge E zufällig.

```
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
        if (Math.random() * n <= q)
            graph[i][j] = 1;
        graph[j][i] = graph[i][j];
```



Symmetrisch besetzt.

Math.random() liefert

zufällige double Zahl aus [0,1].

10 - mal Math.random() $\leq 1/2 = \frac{1}{2}$.

100 - Züge dann ≈ 50 - mal $\leq \frac{1}{2}$.

Math.random() * n $\leq$?, dann

$c = \frac{n}{2}$ bedeutet Math.random() $\leq 1/2$

$c = 3$ bedeutet Math.random() $\leq \frac{c}{n}$,

also eine Klasse, ...

$c = n$ bedeutet 1 - mal n.

8.21

Einige Testläufe:

Grösse = n = 30

$c = 4$

W-keit Kante $\approx \frac{4}{30}$

\# Kanten $\approx \binom{30}{2} \cdot \frac{4}{30}$

$$= \frac{30 \cdot 29}{2} \cdot \frac{4^{2}}{30} = 2 \cdot 29 \approx 54.$$

Programm funktioniert mit

$n = 30$, $m = 40$. $c = 3$ immer

3-färbbar, $c = 7$ nicht mehr

3-färbbar.

Es gibt einen Umschlagpunkt,

etwa bei $c = 5{,}2$ und n groß.

Das nächste Problem ist: Gegeben ein Graph, finde eine Färbung, die mit den wenigsten Farben auskommt. (Klassisches (kombinatorisches) Optimierungsproblem wie bei kürzeste Wege, geringste Kosten, kürzeste Zeit, ...)

Dazu einige Beobachtungen:

- Bei n Knoten werden in jedem Fall n Farben.
- Es nicht alle Färbungen so:

  Knoten 0 — Farbe 0
  Knoten 1 — Farbe 0,1
  ⋮
  Knoten i — Farbe 0,1,...,i

auszuprobieren. Denn sei

$F : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$

eine zulässige Färbung eines Graphen.

Falls $F(0) = j \neq 0$,

vertausche Farbe 0 und j.

Falls $F(1) = j \neq 1$,

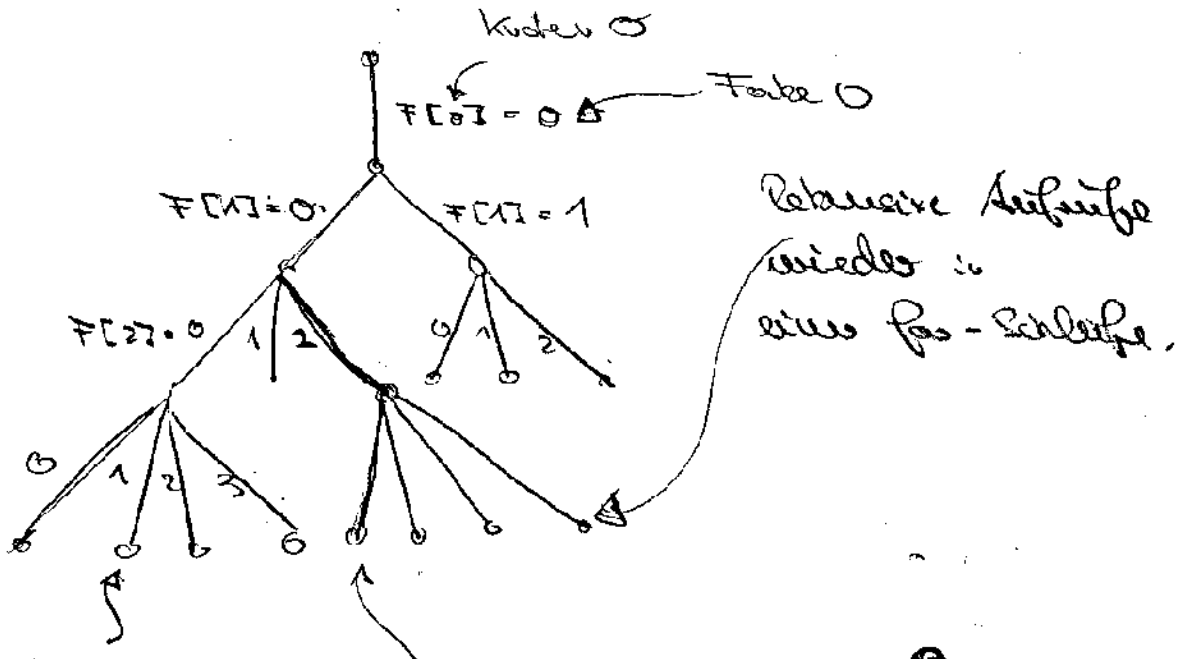
vertausche 1 und j.

Falls $F(2) = j \neq 2$

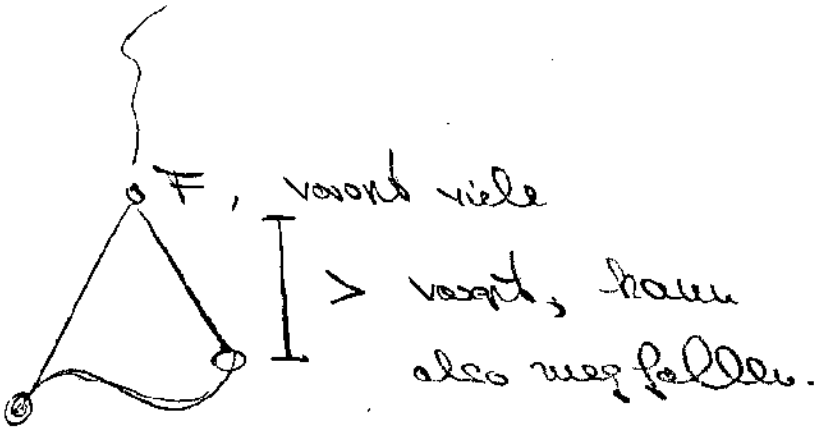
vertausche 2 und j.

Beachte: Ist $F(2) = 0$ von vorher, so geschieht hier nichts

Sei ein Graph G mit Knotenmenge $0, \dots, n-1$ gegeben, so können wir prinzipiell folgenden Baum in Postordnung rekursiv abarbeiten:



Beobachtung: Haben wir eine feste Färbung mit sagen wir vorab (vorläufiges Optimum) vielen Farben so braucht jede partielle Färbung f mit vorab vielen Farben nicht mehr weiter betrachtet werden.



Prinzip: branch and bound.

public static void Faerbb

(int[][] graph, int[] F;
int[] Fvorgh, knoten)

anz = # verschiedene Farben von F.

// interessanter Trick
// da Entwicklung. Nur
// 2-mal durch das array

Falls knoten = F.length.

Haben fertige Färbung.

Testen ob anz < vorgh.

Falls ja, vorgh = anz;

Fvorgh = F'.

In jedem Falle neben.

Falls knoten < F.length

Falls anz > vorgb return,
es hat keinen Sinn mehr.

Falls anz ≤ vorgb,

Farben 0, ..., knoten bei
knoten ausprobieren;
Testen ob momentan zulässig
F entsprechend setzen

Farb(gph, F, Fvorgb, knoten+1)

// Hier werden dann
// alle Fortsetzungen
// von F abgesucht.

Man kann 2 grundsätzlich verschiedene Arten relevante Methoden unterscheiden:

- Aufrufbaum aus Parametern direkt ermittelbar (Fakultätfunktion, Hanoi, 0-1-Vektoren).
- Aufrufbaum erst durch Ausführung der Routinen ermittelbar (alle Backtracking, branch-and-bound Verfahren). Bäume stark unterschiedlich in Abhängigkeit von der konkreten Eingabe.

Manchmal kann man die Rekursion umkommen (Fakultät, Exponentation, 0-1-Vektoren). Insbesondere im zweiten den genannten Fall ist eine Simulation ohne Rekursion schwierig, sogar unmöglich.

Immer kann die Rekursion eliminiert werden, indem man den Prozedurballs per Hand simuliert. Demonstration an Hanoi.java. Für $m = 2$

Zeigen wir das Prinzip,
indem wir den Keller zeigen:

(Pfahl 1, Pfahl 2, Pfahl 3, 2)

(Pf 3, Pf 2, Pf 1, 1)

("Scheibe 2 von Pf 1 nach Pf 2")

(Pf 1, Pf 3, Pf 2, 1)

Spitze

Programmzeiger im Keller = das was noch zu machen ist. Entspricht Rücksprungadresse.

(Pf 3, Pf 2, Pf 1, 1)

("Sch 2 von Pf 1 nach Pf 2")

("Sch 1 von Pf 1 nach Pf 3")

Wegen des 1 oben

(PP 3, PP 2, PP 1, 1)

Ausgabe von

Sch 1 von 1 nach 3

Sch 2 von 1 nach 2.

("Sch 1 von 3 nach 2")

Ausgabe von

"Sch 1 von 3 nach 2"

Hallo Welt, Programm

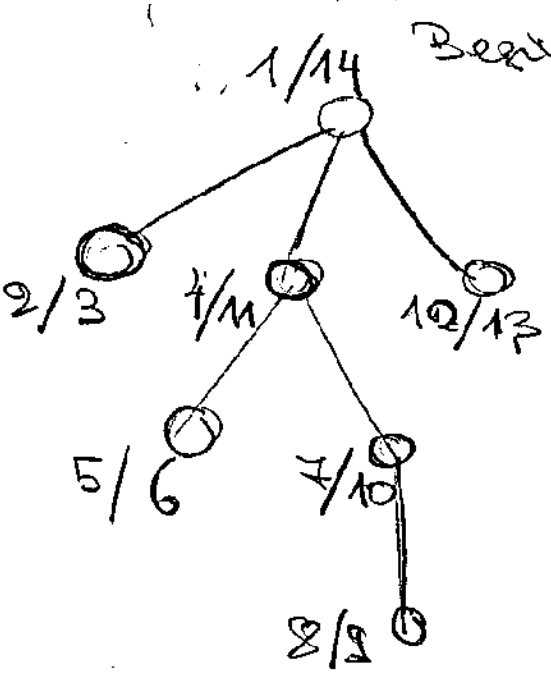
zu Ende.

Ausprogrammiert in HanoiTurm.java.

Noch 2 Nachträge.

Zu Abarbeitungsreihenfolge eines

Prozessbaums:



Ordnet man die Kanten nach

dem Beginn, so ergibt das einen

Präordnungsdurchlauf des Baums:

Erst die Wurzel, dann die Kinder

von links nach rechts, rekursiv wieder

(prae-ver).

Nach dem Ende benennen
wir die Postordnung: Erst die
Kinder, dann die Wurzel und
alles rekursiv wieder.

Zum 3-Färbbarkeitsproblem:

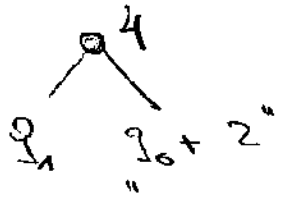
Wir betrachten den Algorithmus:
Gehe die Knoten in der Reihenfolge
1, 2, 3, ..., n durch und färbe jeweils
mit der kleinsten möglichen Farbe.

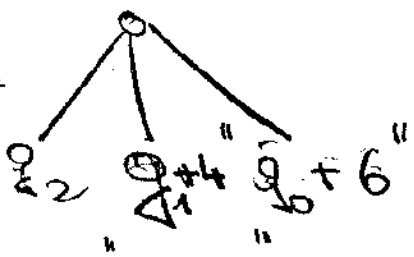
Dieser Algorithmus führt schon
bei 2-färbbaren Graphen zu

beliebig vielen Farben. Dann

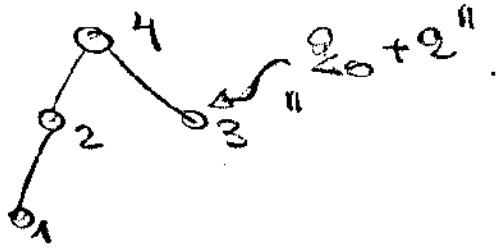
g_0 =

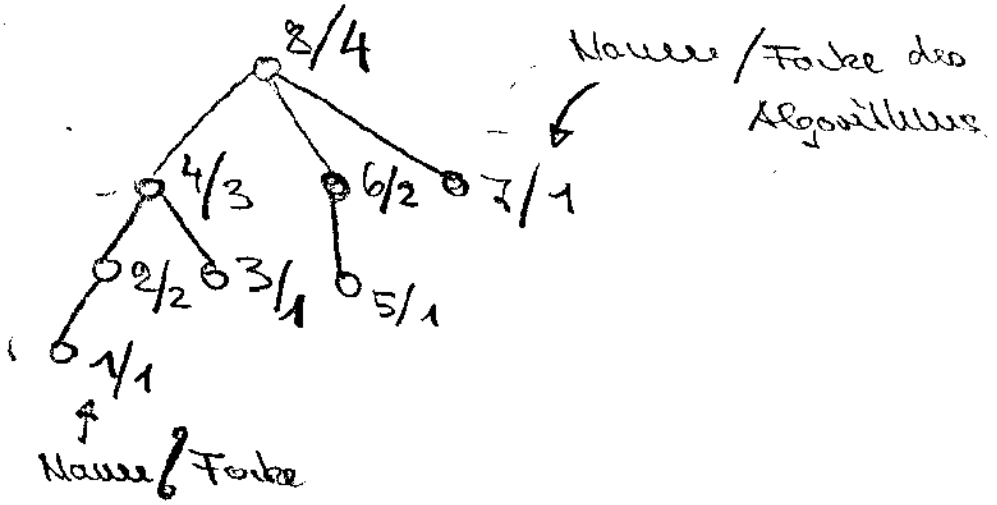
, g_1 = 

g_2 = 

g_3 = 

Also

g_2 = 

g_3 = 

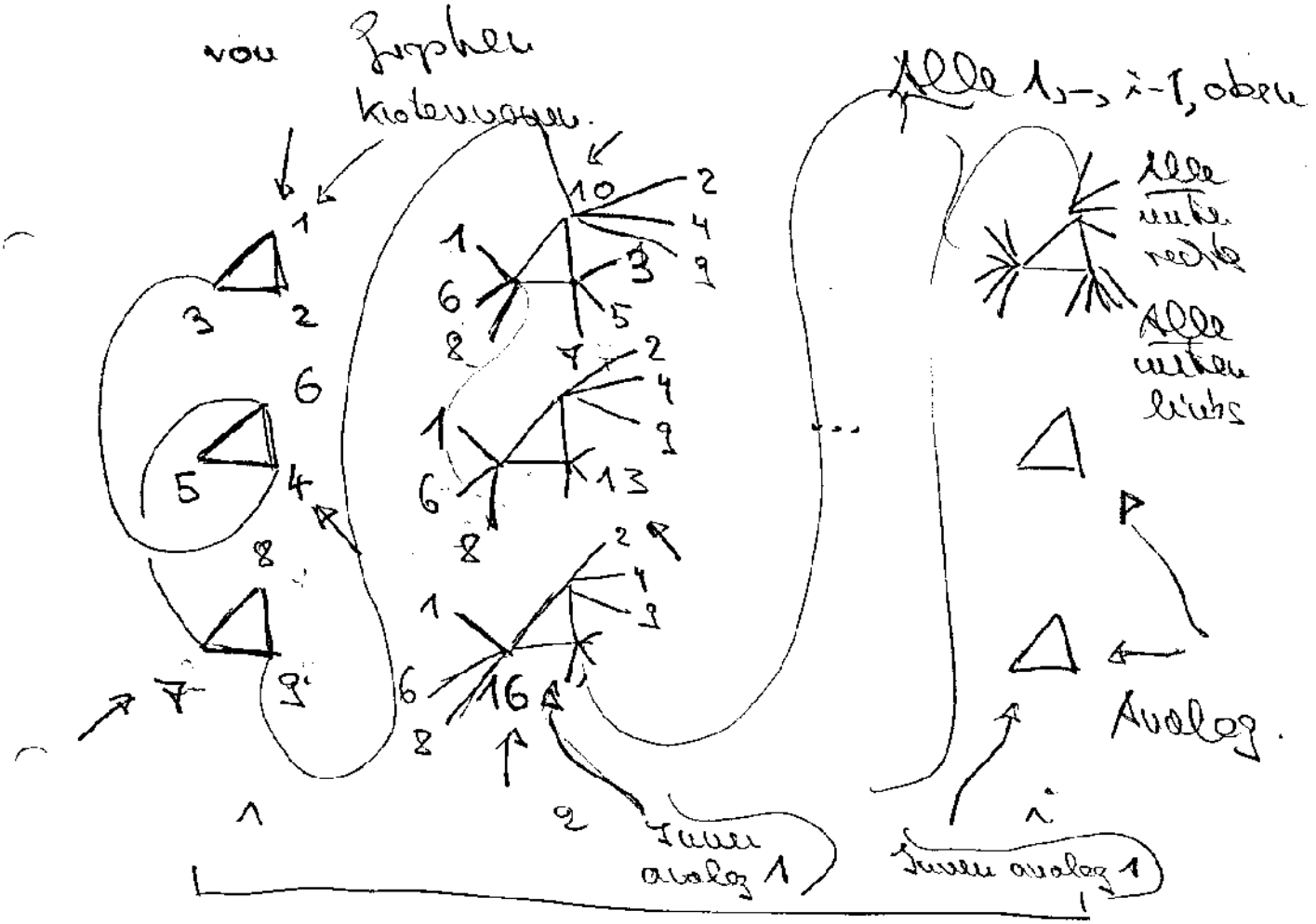
Dann braucht unser Algorithmus bei q_i $i+1$ Farben. Dabei reichen 2 !

Ein etwas besseres Algorithmus:
Gehe zum noch nicht gefärbten Nachbarn kleinsten Namens und färbe ihn mit der kleinsten Farbe.

Das gibt in obigem Beispiel eine 2-Färbung. Allerdings können auch 3-färbbare Graphen finden, für die der Algorithmus

beliebig viele Farben braucht.

Dann folgende Menge (Familie)



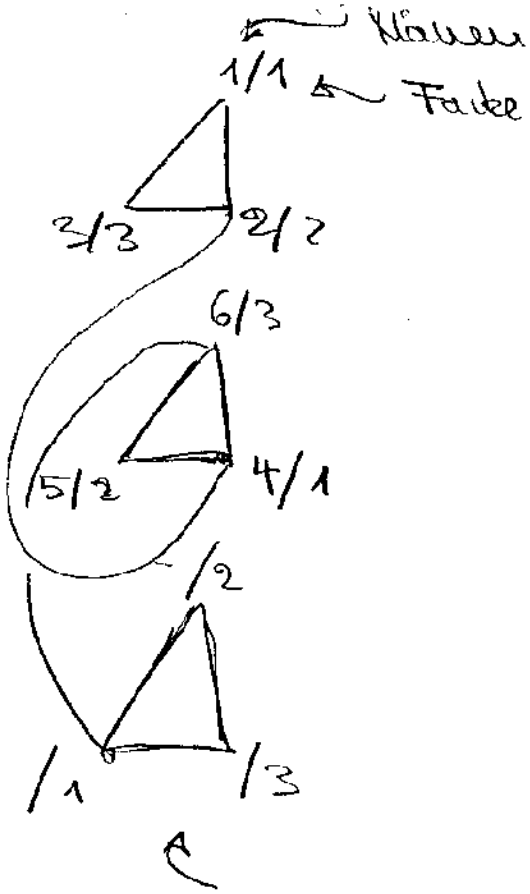
$\mathcal{F}_i :=$

3-Färbung. Jeder Dreieck

8.107

Wie färbt der Algorithmus 4?

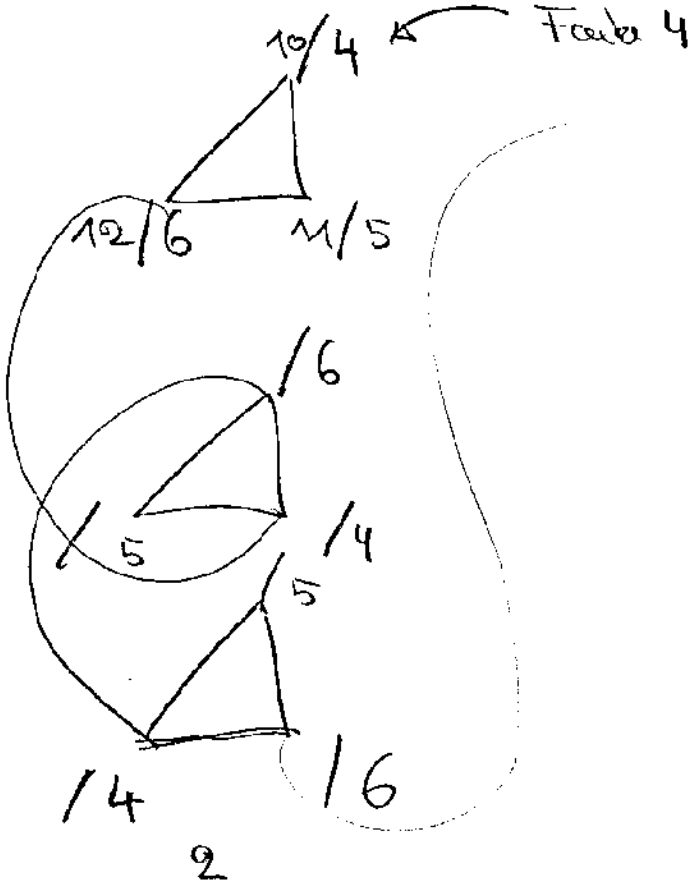
Die Dreiecke über 1 so:



Beachte die Vertauschung der Färbungen.

Dann kosten 10.

Die Dreiecke über 2 so:



Die über 3 müssen nun mit Farbe 4 anfangen. Am Ende 9 Farben vgl. 9; hat also 3 i Farben mit dem einfachen Algorithmus.

2. Zeit und Platz

Neben experimentell beobachtbaren Aussagen über Zeit- und Platzbedarf eines Programms sind fundierte theoretische Voraussagen von Interesse. Wie solche Voraussagen zu treffen sind, wird im folgenden behandelt.

Zunächst etwas zum Platzbedarf.

Ein Zeichen nach ASCII:

7 Bit pro Zeichen. Das 8. Bit eines Byte ist Prüfbit (erlaubt das Erkennen eines falsch stehenden Bits – nicht dessen Korrektur)

$2^7 = 128$ Zeichen.

Ein Zeichen nach Unicode:

2 Byte, also $2^{16} = 2^8 \cdot 2^8 = 65.536$

Zeichen.

Der Übergang von ASCII nach Unicode ist ein schönes Beispiel des „kombinatorischen Explosion": Die einfache Verdopplung der Anzahl Bits von 8 auf 16 bewirkt eine Explosion der Anzahl Möglichkeiten: Von 256 auf 65.536. Exponentielle Zusammenhänge:

- 1 Bit mehr = Verdopplung ($2^{n+1} = 2 \cdot 2^n$)
- Verdopplung der absolut Zahl
  = Quadrierung ($2^{2^n} = (2^n)^2$).
- absolut Zahl mal k
  = hoch k ($2^{k \cdot n} = (2^n)^k$)
- Quadrierung der absolut Zahl
  = hoch absolut Zahl
  ($2^{(n^2)} = 2^{n \cdot n} = (2^n)^n$).

Dagegen bei etwa n^4, 4 konstant, statt 2^n:

- Verdopplung der n
  = Mult. mit Konstante ($(2n)^c = 2^c \cdot n^c$)

Konstante

- Mult. von n mit k
  = Mult. mit Konstante $\left((k \cdot n)^c = k^c \cdot n^c\right)$.

- Quadrierung des n
  = Quadrierung $\left((n^2)^c = (n^c)^2\right)$.

- Addition von 1
  = Addition von a, $n \leq a \leq n^c$:
  $\left(n^c + n^c \leq (n+1)^c \leq (2n)^c\right)$

Bei linearer Abhängigkeit zusätzlich:

- Addition von 1:
  = Addition einer Konstanten.
  $\left(c \cdot (n+1) = c \cdot n + c\right)$

Größere Mengen von Byte

1 Kilobyte (kB): 2^{10} = 1024 Byte.

1 Megabyte (MB): 2^{20} = 1.048.576 Byte

1 Gigabyte (GB): 2^{30} = 1.073.741.824 Byte.

1 Terabyte : 2^{40} Byte.

Ggf. auch: Kilo = 1000, Mega = 1.000.000, ... Manchmal auch in der Informatik.

Beispielwerte

Seite eines Buches	Ungefähr
40 Zeilen à 80 Zeichen.	3 - 6,5 kB
Insgesamt 300 Seiten.	
1000 Seiten	3 - 6,5 MB

Diskette	1,4 MB
Windows Programm	5 MB
Musikstück	40 MB
Hauptspeicher	56 - 128 MB
DVD	5 GB
Festplatte	20 GB bis 100 GB

Speicherplatzbedarf von unseren Programmen:

PRIM.java 2 long

Immer gleich viel Speicherplatz

Elemente vom Typ String wird im Programm angegeben.

ggTPrim.java — Eine Anzahl von long Variablen mit im Programm wird etwas Text mit im Programm.

Immer gleich viel Speicherplatz.

Partition.java — Auf jeden Fall Speicherplatz für ein array mit n Einträgen vom Typ int, also n Plätze für int. (4 Byte)

Hängt von dem eingelesenen n ab!

Außerdem eine Zahl von weiteren Variablen. Auch nicht-explizite mit der Länge

Diese Zahl hängt nur vom Programm ab. Nicht von der Eingabe!

Fakn.java

Einige int-Variablen.

Der Laufzeitkeller,

maximal n Rekursion

n ist eingegeben.

Größe eines Rekursions?

Eine konstante Zahl viel Plätze. Also insgesamt etwa

$c + d \cdot n$

bei Eingaben n, c, d Konstanten programmabhängig.

Abhängig vom Programm, vom Übersetzer. Unabhängig von der Eingabe.

DreiFaeb.java

Wir haben 2 arrays graph und f, graph mit n^2 int Speicherplätzen,

f mit n int Speicherplätzen.

Von Eingabe abhängig!

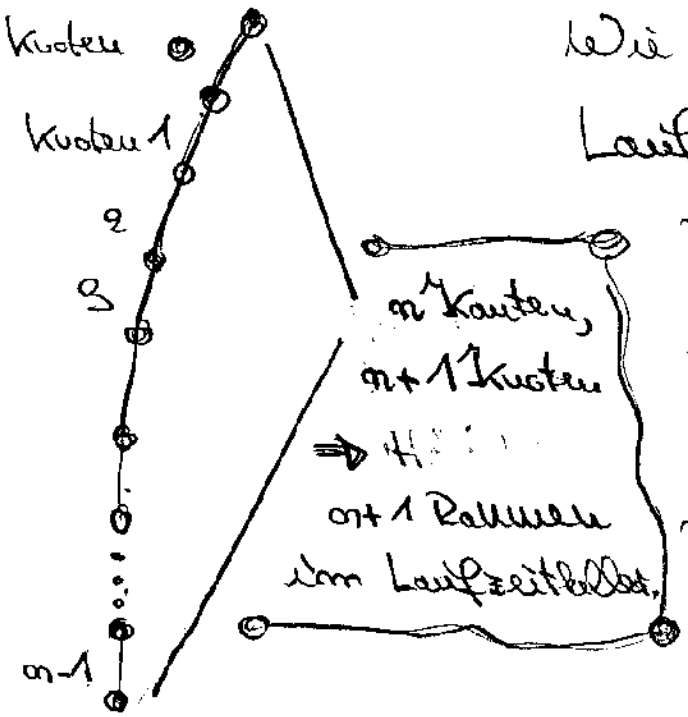
Konstant, programmabhängig.

(1.9)

Aufgaben einige mit Variablen, boolean V

Variablen usf.

Wie groß wird der Laufzeittabelle?



Pro Zeilen eine konstante Zahl von Speicherplätzen.

Dann insgesamt

$cm^2 + m + c(m+1) + d$

also

$m^2 + m + cm + (d+c)$.

Quadratisch. Graph

Rest

Laufzeittabelle

Obwohl Baum groß (maximal 3^n) nur n+1 Zeilen.

Bei Zeitangaben wird das übliche dezimale System benutzt:

1 Millisekunde (ms) : $\frac{1}{10^3} = \frac{1}{1000}$ sek.

1 Mikrosekunde (µs) : $\frac{1}{10^6} = \frac{1}{1.000.000}$ sek

1 Nanosekunde (ns) : $\frac{1}{10^9} = \frac{1}{1.000.000.000}$ sek

1 Hertz = 1 Schritt pro Sekunde

2 Hertz = 2 Schritt pro Sekunde

1 MegaHertz = 1024 (1000) Schritt

1 Gigahertz = 1.000.000.000 Schritt pro Sekunde

1 Schritt = $\frac{1}{10^9}$ Sekunden

1 Nanosekunde.

Heutzutage: 1 Taktschritt in 1 Nanosekunde.

Also Taktfrequenz von 1 Gigahertz.

1 Taktschritt = 1 elementare Aktion.

1 Instruktionszyklus = Eine endliche Zahl von Taktschritten.

1. Befehl zwecks Befehlsadresse holen.
2. Befehl interpretieren
3. Operanden holen
4. Befehl ausführen
5. Befehlszähler auf Adresse des nächsten Befehls.

Im Nanosekunden Bereich.

Im der Informatik: Millisekunde eher lang.

Latenzzeit der Festplatte = Zeit von Anforderung bis Daten der Festplatte im Hauptspeicher sind

Millisekunden Bereich

Zeitverbrauch von unseren Programmen.

PRIM.java Schleifendurchläufe für

$d = 1, d = 2, \ldots$

$d = d_0$, wobei

$d_0 \leq \sqrt{q}$ aber $(d_0+1) \geq \sqrt{q}$.

Also

Schleifendurchläufe $\leq \sqrt{q}$

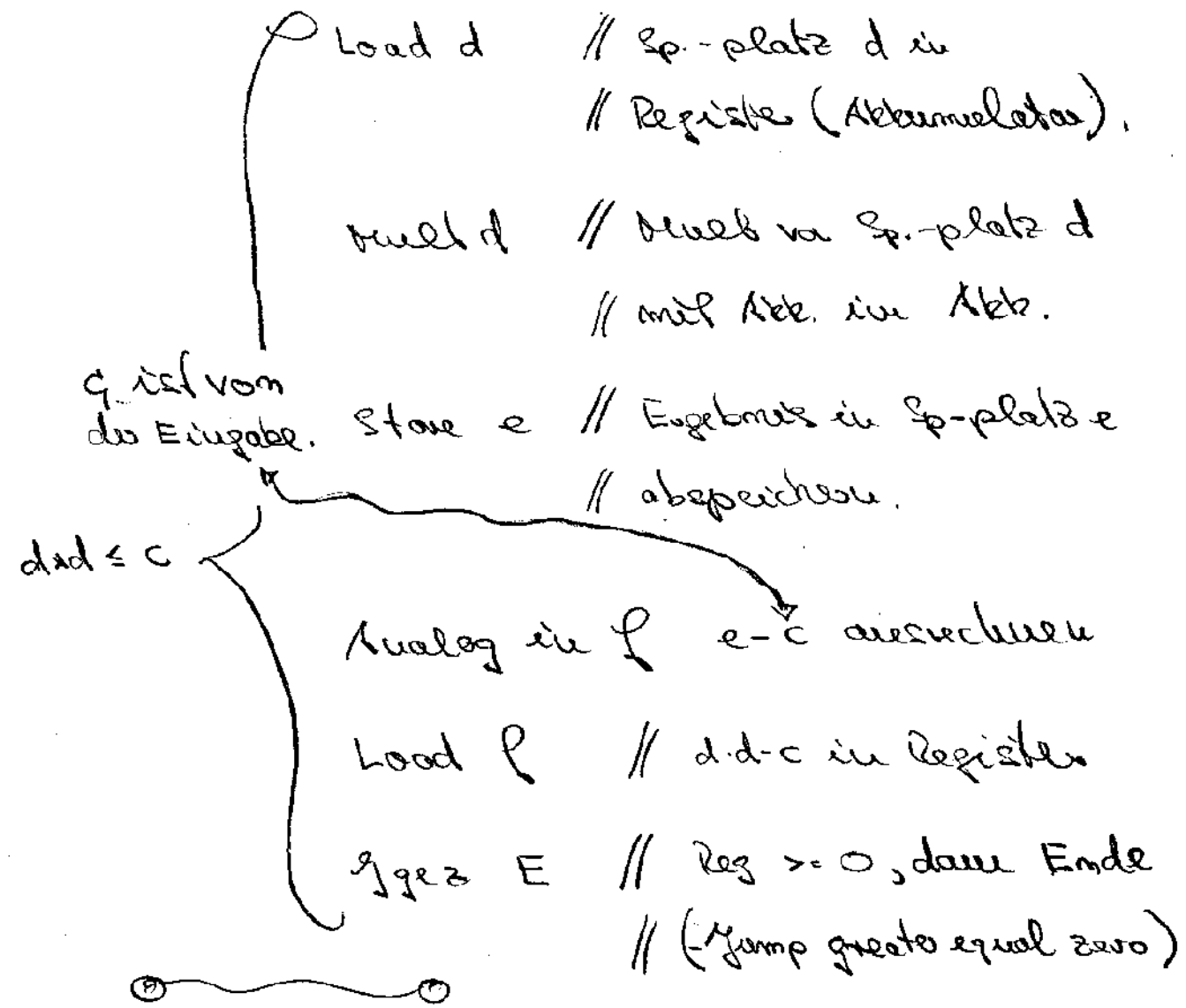
Was passiert in einem Schleifendurchlauf?

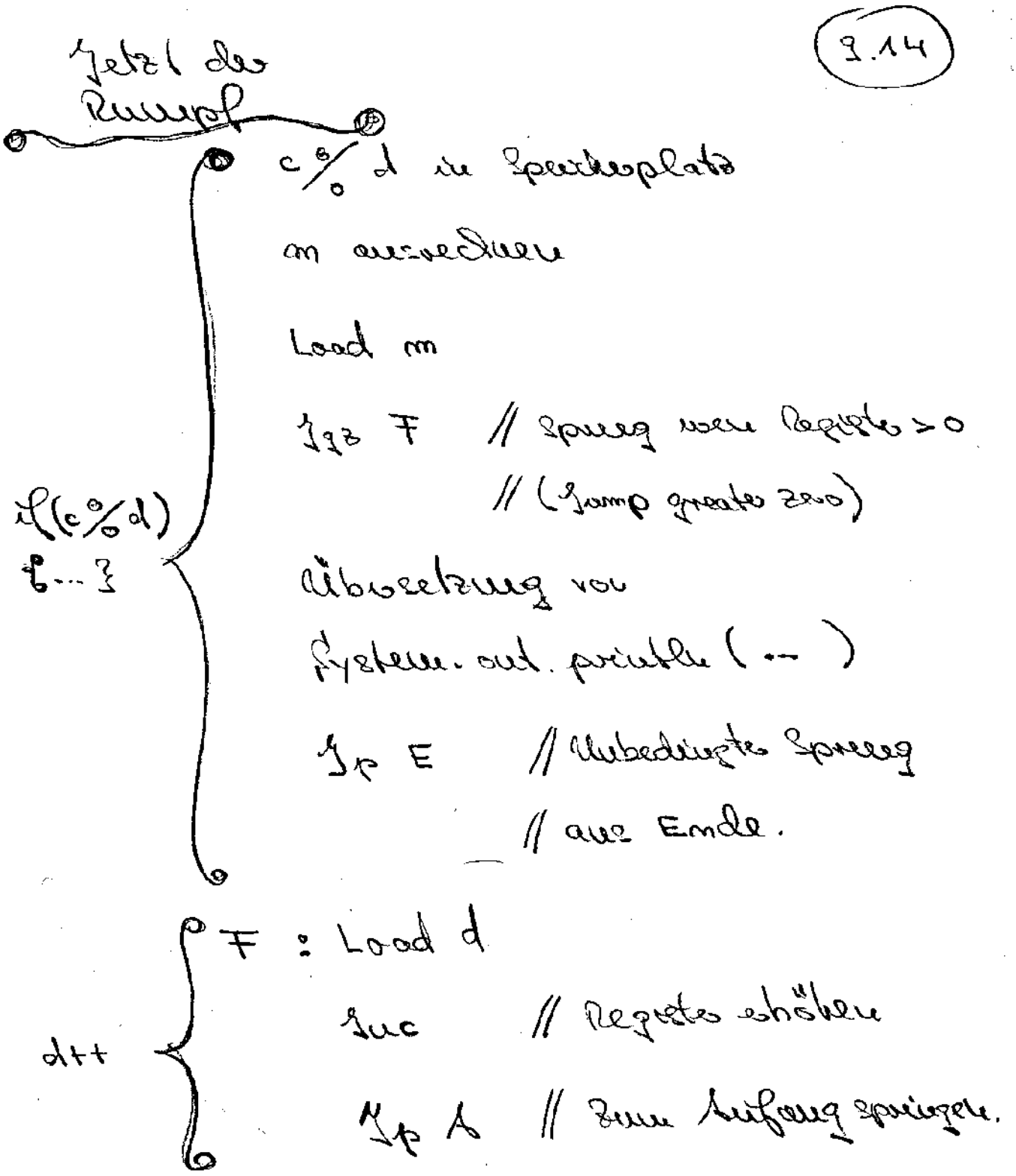
Wie lang dauert einer?

1.

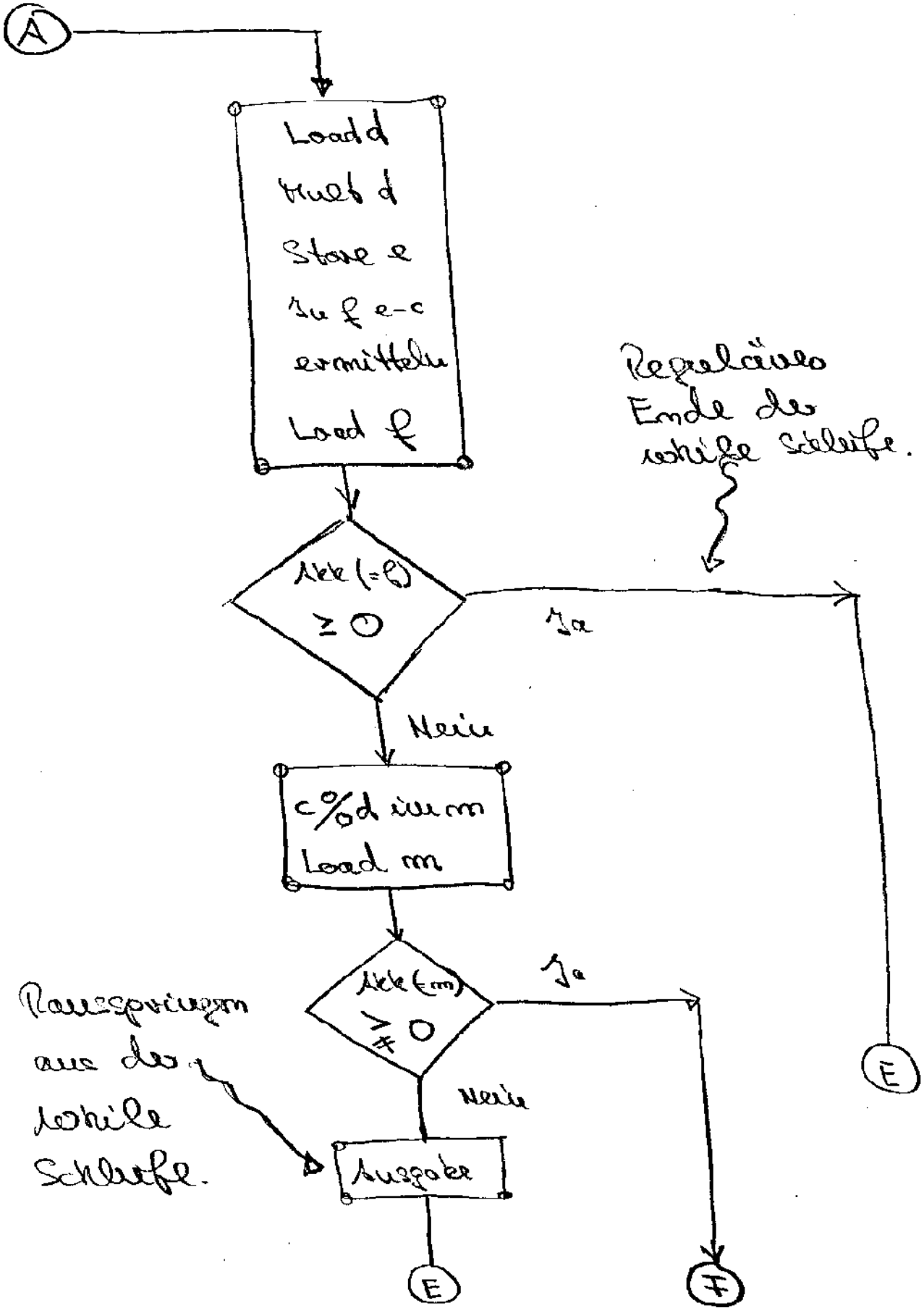
Klar von der Eingabe, hier q bezeichnet, abhängig.

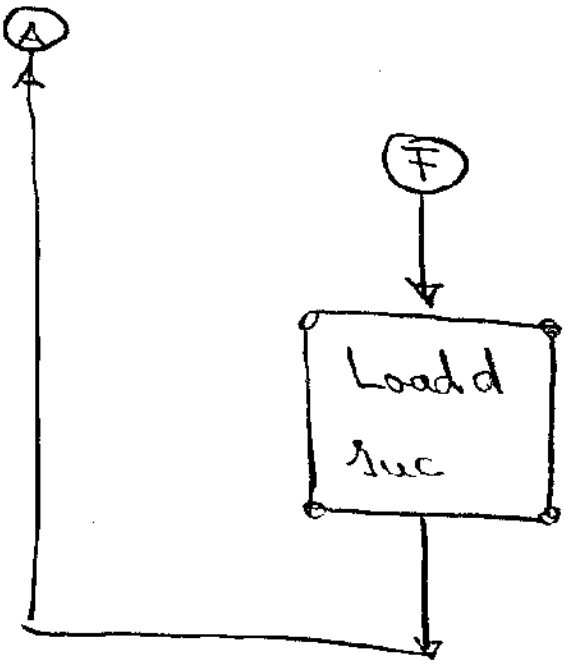
Die while-Schleife wird folgendermaßen übersetzt:

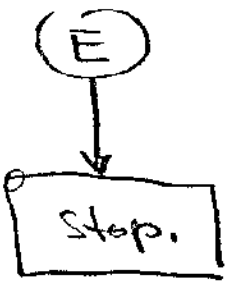



Jetzt do
Rumpf
c % d in Speicherplatz
m auswerten
Load m
Jgz F // Sprung wenn Register > 0
// (Jump greater zero)
if(c%d)
{...}
Übersetzung von
System.out.println(...)
Jp E // Unbedingter Sprung
// ans Ende.
F: Load d
Suc // Register erhöhen
Jp A // Zum Anfang springen.
dtt
E: Stop

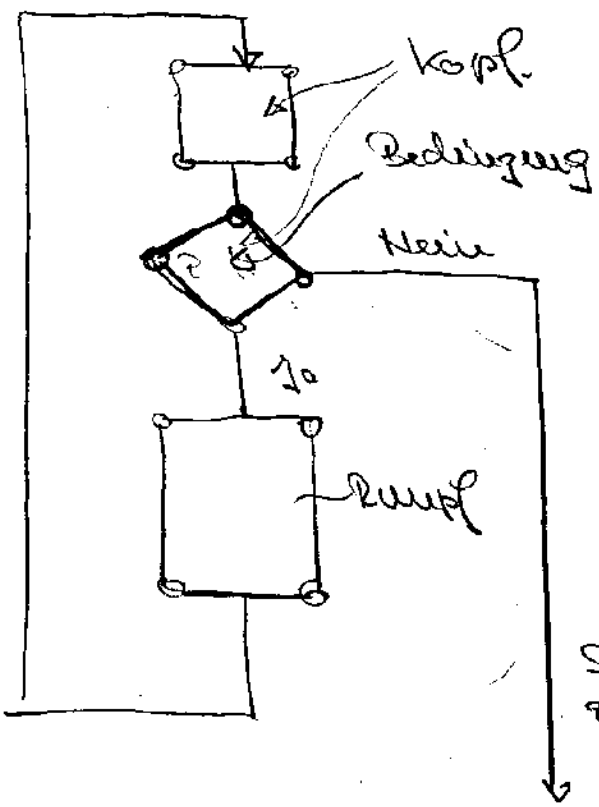
...ls Flußdiagramm:



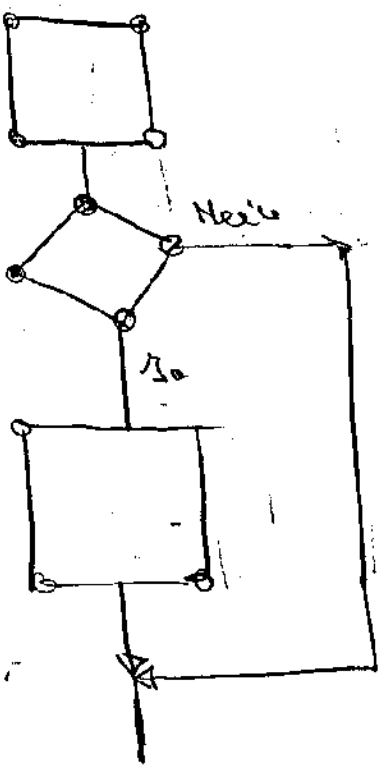




while Schleife



if - then



Beobachtung: Jede einzelne Zeile des Java Programms führt zur Abarbeitung einer konstanten (d.h. unabhängig von c) Anzahl von Maschinenbefehlen.

Damit gilt für unser Programm

Ausführungen von Maschinenbefehlen bei Eingabe von c

$$\leq \underbrace{d \cdot \sqrt{c}}_{\text{Schleife}} + \underbrace{f}_{\text{Anfang und Ende}}.$$

Ist etwa c durch 3 teilbar sind wir schnellstens fertig, also $\leq$.

Also auch

Zeit bei Eingabe von q:

$\leq \left(d \cdot \sqrt{q} + f\right)$ · Zeit für einen Befehl

$= \left(d \cdot \sqrt{q} + f\right) \cdot z$ ns

Nanosekunden

Und auch

durchlaufene elementare Programmzeilen

$\leq d' \cdot \sqrt{q} + f'$

d': Schleife, f': Rest

9.19

Fazit: Laufzeit entspricht
der # ausgeführten elementaren
Programmzeilen. Ob in ms
oder in Maschinenbefehlen nur
Unterschied um einen konstanten
programmabhängigen, eingabeunabhängigen
Faktor.

Je komplizierter die
Zeilen umso größer.

Definition (O-Notation)

Seien $f, g : \mathbb{N} \to \mathbb{R}$ (oder auch $f, g : \mathbb{R} \to \mathbb{R}$) zwei Funktionen.

$f(n)$ ist von $O(g(n))$

($f(n)$ hat Wachstums(größen-)ordnung höchstens $g(n)$)

gdw. gilt:

Es gibt eine Konstante $c \geq 0$,

ein $n_0 \in \mathbb{N}$ so daß

$$f(n) \leq c \cdot g(n) \quad \text{für alle } n \geq n_0$$

Ab n_0 interessant

Gilt $f(n) < 0$, so ist g $|f(n)|$ gemeint.

□

Die Laufzeit unseres Primzahl-
programms ist also $O(\sqrt{m})$,
wenn $m = q$ die Eingabe ist.

Es ist $O(\sqrt{m})$

$$d \cdot \sqrt{m} + p \leq \overline{(d+1)\sqrt{m}}$$

für $\sqrt{m} > p$, also $m_0 > p^2$ gelt.

Getestet wurde das Programm
Beispiel.java. Ganzzahlige Wurzel
nach dem Muster des binären
Suche.

Zeit bis zum Beginn der Schleife:

konstant $\leq d$, man sagt $O(1)$.

Wieviele Läufe maximal durch
die while Schleife? Suchintervall

$[0, n+1)$, n Eingabe, $n+1 =$ [illegible]

$[0, n/2)$, $[n/2, n+1)$, =

$[0, \frac{n}{4})$ $[\frac{n}{4}, \frac{n}{2})$ $[\frac{n}{2}, \frac{n}{2} + \frac{n+1-n/2}{2})$

$[$ ↙ $, n+1)$

Ist $n+1 = 3$, dann

3	Elemente im Suchintervall.
≤ 5	Elemente
≤ 3	Elemente
≤ 2	Elemente
1	Element, Ende.

Beobachtung: Hat ein Suchintervall
m Elemente, also $[\ell, \ell+m)$, dann
haben die nächsten Suchintervalle

$$\left\lfloor \frac{m}{2} \right\rfloor \quad \text{und} \quad \left\lceil \frac{m}{2} \right\rceil \leq \frac{m}{2} + 1$$

=

Elemente. Also bekommen
wir die neue Situation:

$m = n+1$ Elemente

$$\leq \frac{m}{2} + 1 \quad \text{"}$$

$$\leq \frac{\frac{m}{2} + 1}{2} + 1 = \frac{m}{4} + \frac{1}{2} + 1$$

$$\leq \frac{m}{8} + \frac{1}{4} + \frac{1}{2} + 1$$

$$\vdots$$

$$\leq 1 + \frac{1}{2^\ell} + \frac{1}{2^{\ell-1}} + \dots + \frac{1}{2} + 1$$

wobei $l+1 \geq \lceil \log m \rceil$ ist.

Nun ist

$$\sum_{i=1}^{l} \left(\frac{1}{2}\right)^i \leq \frac{1}{2} \cdot \frac{1}{1-\frac{1}{2}} = 1$$

(geometrische Reihe). Also ist die Anzahl von Elementen ≤ 2 und wir machen noch einen Lauf durch die Schleife.

Damit ist die Laufzeit von Binf. java

Pro Lauf konstant viel.

$$f + d \cdot (\log(n+1) + 1)$$

Rest, Anfang, Ende. — # Läufe

Das ist $O(\log n)$. Denn

es ist

konstant

$\log$ $O(\log n)$, da $\log n \to \infty$

Weiter ist

$\log(n+1)$

$\leq$ Für $n \geq 1$ ist $n+1 \leq 2n$.

$\log(2n)$

$=$

$(\log n) + 1$

$\leq$ Für $n \geq 2$ ist $\log n \geq 1$.

$2 \cdot (\log n)$,

also $\log(n+1)$ ist $O(\log n)$.

Schließlich ist

$$d \quad O(\log n).$$

Also haben wir etwas von der Form

$$O(\log n) + \underbrace{O(\log n) + O(\log u)}_{d \cdot \log(n+1)}$$

Das ist aber

$$3 \cdot O(\log n),$$

also

$$O(\log n).$$

Das Programm Eukl.java,
Euklidische Algorithmus mit
modulo Operation.

Eingabe $g \geq 2$, $h \geq 2$.

Wird im Laufe der while
Schleife $g = 1$ oder $h = 1$
kommt es noch zu einem Lauf
der Schleife; d.h. $O(1)$ Eval,
und der ggT ist 1.

Schleife:

```
while (g>0 && h>0)
  if (g>h)
    g = g%h;   // danach g<=h.
  if (h>g)
    h = h%g;   // danach h<=g.
```

Für die $g\%h$, $h\%g$ oben gilt $g\%h \le g/2$, $h\%g \le h/2$.

Denn da $g>h$, ist

$$g = \underbrace{g/h}_{\ge 1} \cdot h + \underbrace{g\%h}_{<h} > 2 \cdot g\%h$$

$\underbrace{\qquad\qquad}_{>g\%h}$

Ebenso für $h\%g$.

Wir bekommen folgende Werte:

$$g_0 > h_0$$

$$\leq g_0/2 \qquad \leq h_0/2$$

$$\leq g_0/4 \qquad \leq h_0/4$$

$$\vdots$$

Also nach dem l-ten Lauf ist da Wert auf $g \leq \frac{g_0}{2^l}$.

Nun ist $\frac{g_0}{2^l} \leq 1 \Leftrightarrow \log_2 g_0 \leq l$.

Also Laufzeit $O(\log_2 g_0)$!

Bei Eingabe $g \geq h > 0$ und $\frac{g}{h} \leq c$ mit konstant vielen Mach-befehlen.

9.30

Hanoi. java

Methodenaufruf, welche Zeit?

Laufzeitkeller verwalten

+ Parameter übergeben + Rückgabewert übg.

+ Zeit im Rumpf

Laufzeitkeller: Rahmen einrichten,

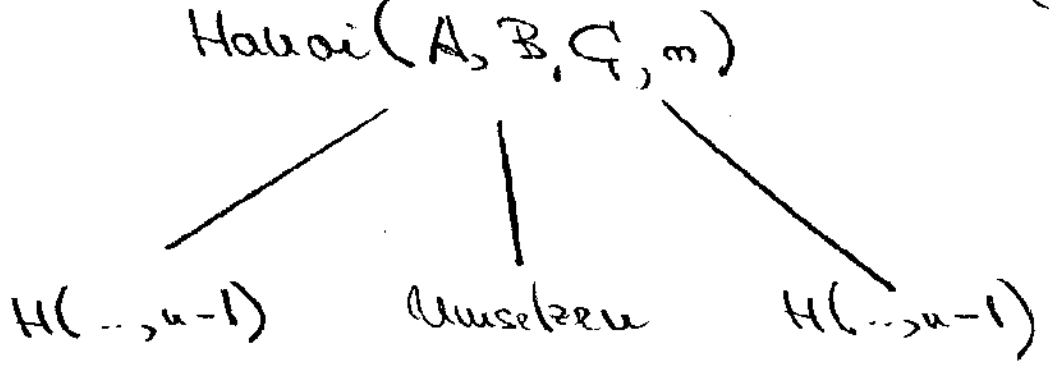
Rahmen löschen: O(1)

Parameter übergeben: O(1)

(Kein Kopieren von Feldern,
Instanzen von Klassen!)

Rückgabewert: O(1)

Alle Konstanten natürlich
programmabhängig.



T_n = # Knoten des Aufrufbaums von $H(\ldots n)$.

Dann gilt

$$T_1 = 1$$

$$T_{n+1} = 1 + 2 \cdot T_n$$

Dann gilt induktiv

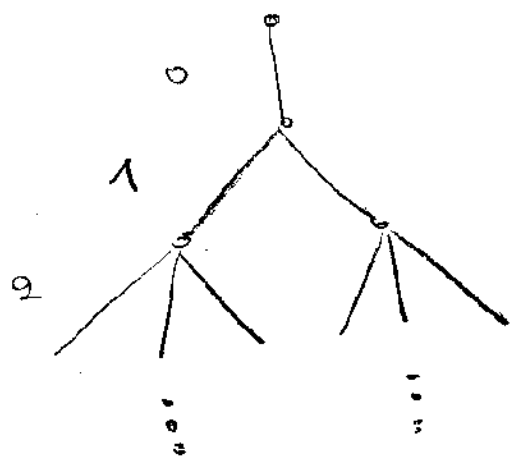
$$T_n = 2^n - 1$$

Pro Knoten $O(1)$ Zeit,

also Zeit $O(2^n)$.

Schleifblech Faerb.java

Aufrufbaum, schlimmstenfalls:



Tiefe 0

Tiefe 1

Tiefe 2

Tiefe 3

Tiefe n

n Farben testen.

9.33

Wieviele Knoten hat der Baum?

Der Baum hat ...

Knoten in Tiefe $l = l!$.

$l = 0$ ✓, $l = 1$ ✓, $l = 2$ ✓

$l > 2$, dann

$l \cdot$ # Knoten in Tiefe $l-1$

$=$ Ind.-Vor

$l \cdot (l-1)!$

$= l!$ Knoten in Tiefe l.

$\square$

Insgesamt

$$\sum_{l=0}^{n} l!$$

$$= n! + \underbrace{(n-1)! + (n-2)! + \cdots + 3! + 2! + 0!}_{\leq n \cdot (n-1)!}$$

$\leq 2(n!)$ viele Knoten.

Dann Teil von $O(n!)$

ist hinreichend.

Bei allen Input mit n Knoten sind wir somit in $O(n!)$ fertig.

Worst-case Zeitaufwand.

Wir betrachten folgenden

Aspekt einer Laufzeitfunktion $f(n)$.

Sei eine Zeit x gegeben. D.h.

$$f(n_0) = x$$

bedeutet das, wir können Eingaben

bis zu größe n_0 (sofern $f(n)$

monoton) in Zeit x lösen.

Nehmen wir jetzt an, daß

sich die Rechengeschwindigkeit

verdoppelt. Die Zeit ist also $\frac{1}{2} \cdot f(n)$.

Wir fragen wieder nach dem n_1

mit

$$\frac{1}{2} f(n_1) = x \quad \text{oder} \quad f(n_1) = 2x.$$

$f(n)$	n_0	n_1
$\log_2 n$	2^x	$2^{2x} = (2^x)^2$
$c \cdot n$	x/c	$2 \cdot (x/c)$
n^k	$\sqrt[k]{x}$	$\sqrt[k]{2} \cdot \sqrt[k]{x}$
		$\uparrow$ Konstant > 1.
2^n	$\log_2 x$	$(\log_2 x) + 1$
$n^{\log\log n}$	$x^{\frac{1}{\log\log x}}$	$2^{\overbrace{\frac{1}{\log\log x}}^{\to 0}} \cdot x^{\frac{1}{\log\log x}}$
		$\to 1$
		$2^0 = 1$
$n!$	keine Vereinfachung	
	feststellbar.	

Doppeln etwa: Bessere

Algorithmus

$$x = 2^{u_0} \quad , \quad x = 2^{1/2\, u_1} \quad ,$$

dann

$$u_0 = \log_2 x \quad , \quad u_1 = 2 \cdot \log_2 x$$

↑

Verdopplung.

Ebenso

$$x = u_0^4 \qquad x = u_1^2$$

dann

$$u_0 = \sqrt[4]{x} \qquad u_1 = \sqrt[2]{x} = \sqrt[4]{x} \cdot \sqrt[4]{x}$$

↑

Quadrierung.

Also: Bessere Reduktion

und bessere Algorithmen!

$k(N)$	Bezeichnung	10	100	1000	10^4	10^5	10^6
1	konstant	1	1	1	1	1	1
$\log(N)$	logarithmisch	3	7	10	13	17	20
$\log^2(N)$		10	50	100	170	300	400
$\sqrt{N}$		3	10	30	100	300	1000
N	linear	10	100	1000	10^4	10^5	10^6
$N \cdot \log(N)$	log-linear	30	700	10^4	10^5	$2 \cdot 10^6$	$2 \cdot 10^7$
$N^{3/2}$		30	1000	$3 \cdot 10^4$	10^6	$3 \cdot 10^7$	10^9
N^2	quadratisch	100	10^4	10^6	10^8	10^{10}	10^{12}
N^3	kubisch	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^N	exponentiell	1000	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

Abb. 4-4: Häufig auftretende Komplexitäten und ungefähre Werte für C=1.

k konstant.

Polynomial n^k. Beschleunigung

des Rechners um konstanten Faktor c

$\Rightarrow$ Vergrößerung der Eingaben

in gegebener Zeit um konstanten Faktor $\sqrt[k]{c}$.

Exponentiell $2^{\epsilon n}$, $\epsilon > 0$, c^n, $c > 1$.

oder auch $n^{\epsilon \frac{a_1}{\log n}}$, $\epsilon > 0$.

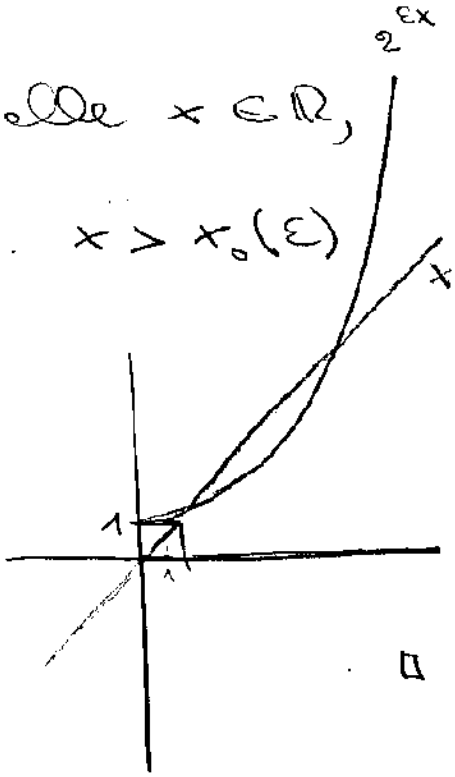
Auslegung: Nur Vergrößerung

der Eingaben um konstanten Summanden

Satz

Sei $\varepsilon > 0$ fest. Für alle $x \in \mathbb{R}$, x hinreichend groß, d.h. $x > x_0(\varepsilon)$ für ein $x_0(\varepsilon)$, gilt

$$2^{\varepsilon \cdot x} > x.$$



□

Daraus folgt das ganze Sprichwort „exponentiell geht es polynomial":

Für alle $\varepsilon > 0$ konstant, alle $z > 0$ konstant, alle $x > x_0(\varepsilon, z)$

$$(1) \quad 2^{\varepsilon \cdot x} > x^z.$$

Interessanter Trick, um Exponenten loszuwerden.

Folgt, da gilt $2^{\frac{1}{z}\varepsilon \cdot x} > x$ indem

da letzte Satz mit $\frac{1}{2} \cdot \varepsilon > 0$ statt ε verwendet wird, Die die Operation hoch 2 ist monoton, erhält also die Ungleichung. Dann insbesondere $2^{\varepsilon \cdot x}$ nicht $\mathcal{O}(x^2)$.

Unter den Voraussetzungen wie vorher gilt dann auch

$$x^{\varepsilon} > (\log x)^2 . \qquad (2)$$

Das folgt, da auch $\log_2 x \to \infty$ geht und deshalb mit (1)

für hinreichend großes x,

(groß genug, daß $\log_2 x$ groß genug)

gilt

$$x^\varepsilon = 2^{\varepsilon \log_2 x} > (\log_2 x)^k .$$

Dann ist insbesondere

$$x^\varepsilon > c \cdot \log_2 x$$

für jede Konstante c, x groß genug.

Daraus ergibt sich wegen der

Monotonie von „2 hoch"

$$2^{(x^\varepsilon)} > 2^{\log_2 x \cdot c} = x^c \qquad (3)$$

für alle $\varepsilon > 0$, c.

Beweis des Satzes von S. 9.40

Mit Analysis zeigen wir:

$$2^{\varepsilon \cdot x} - x \to \infty .$$

Dazu zeigen wir, daß $2^{\varepsilon \cdot x} - x$ streng monoton steigend ist, daß die Ableitung > 0 ist für x groß genug. Es ist

$$2^{\varepsilon \cdot x} = e^{\overbrace{\ln 2}^{<1} \cdot \varepsilon \cdot x}$$

und die Ableitung ist (Kettenregel)

$$\ln 2 \cdot \varepsilon \cdot e^{\ln 2 \cdot \varepsilon \cdot x} = (\ln 2) \varepsilon \cdot 2^{\varepsilon \cdot x} \to \infty$$

Die Ableitung von $-x$ ist -1.
Die Behauptung gilt.

Es geht auch ohne Analysis:

Dazu zeigen wir, daß für alle

$n \in \mathbb{N}$, hinreichend groß

$$2^{\xi \cdot n} > n+1 \qquad (4)$$

ist. Dann folgt für $x \in \mathbb{R}$, groß genug,

↝ Abrunden nicht vergessen.

$$2^{ex} \geq 2^{\lfloor x \rfloor} > \lfloor x \rfloor + 1 \geq x,$$

$$\uparrow$$

Wegen Obigem.

was die Behauptung des Satzes ist.

Die Behauptung (4) folgt aus

$$2^n > \xi \cdot n + 1, \qquad (5)$$

für alle $c \in \mathbb{R}$ und n groß genug,

dann setze man $q = 1/\varepsilon$ gilt

$$2^n > 1/\varepsilon \cdot n + 1$$

≡ ein wichtiger Trick: Das Substitutionsverfahren

für alle n groß genug. Da

$\varepsilon n \to \infty$ können wir substituieren

ε u. für u

$$2^{\varepsilon u} > \underbrace{n}_{= 1/\varepsilon \cdot \varepsilon u} + 1.$$

(5) ist nun auch einfach: Für

$c \geq 5$ ist $2^c > c^2 + 1$: gilt für

$c = 5$ und $(c+1)^2 + 1 = c^2 + 2c + 1 + 1$

≡ ist $2^c > c^2 + 1 > 2c + 1$. Also

$\uparrow$ Ind.-Vor. $\quad \downarrow$ $c \geq 5$

$$2^{c+1} > (c+1)^2 + 1$$

und für alle $c \geq 5$ $2^c > c^2 + 1$.

Nun eine Induktion über n.

Ind.-Anf. $n = c_1$, dann

$$2^{c_1} > c_1^2 + 1,$$

wie oben gezeigt. Nun zum

Ind.-Schluß: Sei $n \geq c_1$ fest.

$$2^n > c_1 n + 1$$

nach Ind.-Vor. Dann

$$\begin{aligned} 2^{n+1} &= \\ &> 2 \cdot 2^n \\ &\overset{\text{Ind.-Vor.}}{>} (cn+1) + \underbrace{(cn+1)}_{\geq 1} \\ &> cn + 1 + c_1 \\ &= c(n+1) + 1. \end{aligned}$$

□

Hier sieht man: c muß hinreichend groß sein

Für $k > 1$ ist Beh.: $n^k > c \cdot n$. Das impliziert $n > c^{\varepsilon} n^{\varepsilon}$, impliziert $\forall n > n^{\varepsilon}$ für $\varepsilon < 1$

Einige kombinatorische Formeln:

$n! = n(n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 =$

Bijektive Abbildungen zwischen
& n-elementigen Mengen, =

Permutationen auf $1, \ldots, n$.

Induktion über n.

$k^n =$ (keine Bijektionen oder so etwas.)

Abbildungen von M nach N
mit $|M| = n$, $|N| = k$.

Induktion über n. Für jedes
weitere Element k neue
Möglichkeiten.

2^n .

Abbildungen von M mit $|M| = n$ nach $N = \{0,1\}$.

Teilmengen von M, $|M| = n$.

$i \in M$	$f(i) \in \{0,1\}$
1	0
2	1
3	1
⋮	0
	1
	1
n	1

Dann entspricht $f : M \to \{0,1\}$

der Teilmenge

$$T = \{ i \in M \mid f(i) = 1 \}.$$

Fü. $k > n$ ist

$$\binom{n}{k} = 0$$

Fü. $k \leq n$ gilt

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

Teilmengen mit genau k Elementen von M mit $|M| = n$.

Induktion mit

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Be-

Pascalsches Dreieck

3/2e

$$\binom{0}{0} = 1$$

$$\binom{1}{0} = 1 \qquad \binom{1}{1} = 1$$

$$\binom{2}{0} = 1 \qquad \binom{2}{1} = 2 \qquad \binom{2}{2} = 1$$

$$\binom{3}{0} = 1 \qquad \binom{3}{1} = 3 \qquad \binom{3}{2} = 3 \qquad \binom{3}{3} = 1$$

$$\binom{4}{0} = 1 \qquad \binom{4}{1} = 4 \qquad \binom{4}{2} = 6 \qquad \binom{4}{3} = 4 \qquad \binom{4}{4} = 1$$

$$\vdots$$

Geometrische Reihe:

$$\sum_{i=0}^{m} q^i = \frac{q - q^{m+1}}{1-q} \quad \text{für } q \neq 1$$

$$\sum_{i=0}^{m} q^i = m+1 \quad \text{für } q = 1$$

(3.51)

Für $|q| < 1$ ist $q^{n+1} \to 0$ für $n \to \infty$.

Also dann

$$\sum_{i=0}^{\infty} q^i = \lim_{n \to \infty} \sum_{i=0}^{n} q^i$$

$$= \lim_{n \to \infty} \frac{1 - \overbrace{q^{n+1}}^{\to 0}}{1-q} = \frac{1}{1-q} \; !$$

$q = 1/2$, dann

$$\sum_{i=0}^{\infty} q^i = \frac{1}{1-\frac{1}{2}} = 2$$

$$= 1 + \frac{1}{2} + \frac{1}{4} .$$

Daher

$$\sum_{i=1}^{\infty} q^i = \frac{1}{2}\left(\frac{1}{1-\frac{1}{2}}\right) = 1 .$$

↑

10. Der grundlegende Umgang

mit Klassen

Es wird Kapitel 10 von

Ratz / Scheffler / Seese

besprochen.

Wir haben bisher folgenden Programmaufbau:

global sichtbar - wenn nicht verdeckt.

```
public class <Name> {
  static int a=1, b=2, c=3; // Klassen-
                            // variablen
  public static class <KName> { // innere
                                // Klasse
    public int Alter;
    public String Name; } // Komponenten-
                          // variablen, Instanzvariablen

  public static int <MName>(<Par>) {
                          // Methodendeklaration
    int x
    :
  }

  public static void main(String[] args){
    :
  }
```

Auslagern von Klassen zwecks
mehrfacher Benutzung:
Das static Feld.

```
public class <KName> ? // Top-level;
                       // Eluent-Klasse
                    ?
```

in Datei <KName>.java.

Beim Übersetzen

<KName>.class

Bei inneren Klassen

<Name> $ <KName>.class

<Name>: Umgebende Top-level-Klasse

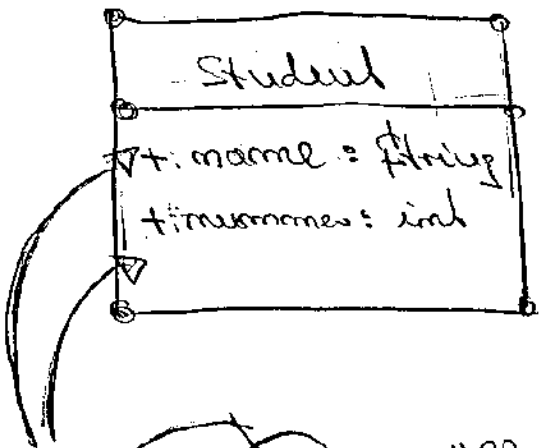
<KName>: Name der inneren Klasse.

Die Idee der Objektorientierung, gebräuchlich in der modernen Softwareentwicklung basiert auf dem Konzept der Klasse. Einführung am Beispiel des Begriffes Student. Eine Top-level Klasse bei seiner Modellierung:

```
public class Student {
    // Stellen Studenten dar.

    public String name;

    public int nummer;
}
```

Klassendiagramm in UML

(unified modelling language)



wegen (public), öffentliche Zugriffsrechte

Instantiierung (d.h. Objekte aus Klasse)

```
Student studi = new Student( );

studi.name = ... ;

studi.matrikelnr = ... ;
```

10.6

Strukturierung durch Kapselung

Bisher Änderung an Klasse Student (etwa zu public String name, zwecks Speicherung) erfordert Änderungen an allen Punkten, wo Student vorkommt!

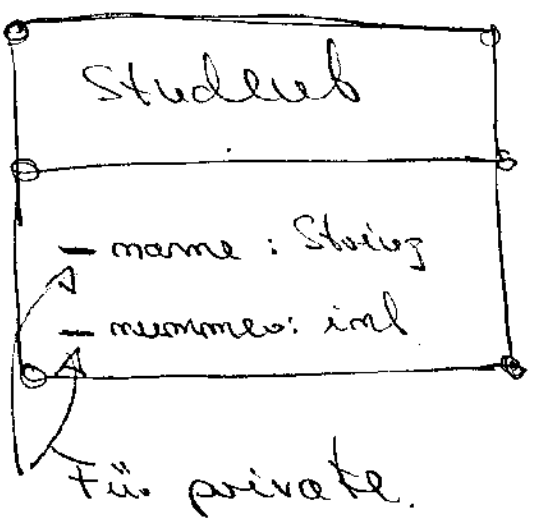
Zugriffsmethoden helfen hier ab.

```
public class Student {
    private String name;
    private int nummer;
}
```

public = jede Klasse kann auf die Variablen zugreifen.

private = keine Klasse kann zugreifen, (außer die Klasse Student selbst).

UML Diagramm:



Zum Zugriff durch andere Klassen:

static fehlt — Instanzmethoden.

public <Rückgabetyp> <MName>(<Pos>) {

// gewünschtes Programm

}

Bisher waren unsere Methodendeklarationen

public static <RTyp> <OMName>(<Pos>) {

// Programm

}

Instanzmethode (ohne static) ist
an spezielles Objekt gebunden, das
existiert ohne ein konkretes Exemplar.
Hat Zugriff auf Instanzvariablen
des zugehörigen Exemplars.

10.3

Erweitern wir die Klasse Student um Zugriffsmethoden, so sieht die Klassendeklaration so aus:

```
public class Student {

    private String name;
    private int nummer;

    public String getName( ) {
        return this.name; }

            // "this" ist Referenz auf
            // Objekt selbst, aus dem
            // aufgerufen wird. "this"
            // ist implizite Komponenten-
            // variable.
```

10.10

```
public void setName(String name) {
    this.name = name; }
    // aus Objekt.

public int getMatrNr() {
    return matrNr; }
    // aus Klasse.matrNr

public void setMatrNr(int m) {
    matrNr = m; }
```

UML-Diagramm in Abbildung 10.3.

Aufruf dann so:

```
...
matrNr = 123
Student stud = new Student();
stud.setName("Hülle");
stud.setMatrNr(78710);
int Mat.Nr = stud.getMatrNr();
```

10.11

Beste

int Matr Nr = studi. nummer

bzgl. Fehlermeldung (sofern nicht in der Klasse Student selbst).

Also: Verlassen von static in der Methodendeklaration

⇔ Binden an ein Objekt.

Die Testausmethode

```
public boolean valNummer {

    return

        nummer >= 10000

            && nummer <= 99999

            && nummer %2 != 0; }
```

überprüft die Gültigkeit einer Matrikelnummer. Darauf muss

setNummer:

```
public void setNummer(int n) {

    int alteNummer = nummer;

    nummer = n;          // war ist neue
                         // nummer noch nicht
    if (!valNummer())    // initialisiert &
        nummer = alteNummer;
}
```

Bild nach außen.

Validiert: gleiche Schnittstelle wie

Klasse Student, auch bei neuer (!)

Regeln für die Matrikelnummer.

Erleichterung von Modifikationen.

16.1b

Weitere Methode:

```
public String toString() {
    return name + "(" + nummer
        + ")"; }
```

Aufruf dann so:

```
Student studi = new Student();
studi.setName("k.k");
studi.setNummer(12345);

System.out.println(studi.toString());
```

Ergibt k.k.(12345) als Ausgabe.

Auch möglich wäre:

System.out.println(studi);

für studi.toString

da toString bei jeder Instanz

als Methode dabei ist. Wir haben

das alte toString überschrieben.

Unsere früheren Methoden gewiß

public static <Rückgabetyp> <Name>(<Par>){

// Programm }

sind Klassenmethoden oder

statische Methoden. Auch ohne

Instanz existiert. Das

Programm 12ITest.java

zeigt noch einmal, daß unsere

bisher geschriebenen Programme

einfach Top-level Klassen sind.

Es können Instanzen generiert

und ausgegeben werden. Auch dem Objekt!

Zum Nutzen der als statisch

definierten Methoden.

Wollen alle existierenden Studenten

zählen. Variable zählen einmal

für alle Instanzen der Klasse;

gemäß:

Kein Zugriff von außen möglich!

Für alle Instanzen.

```
private static int zaehler = 5;
```

Hatten früher bereits:

```
public static int ...
static int ...
```

am Beginn unserer Klasse.

Zugriff von außen durch:

Klassenmethode, ohne Objekt aufrufbar.

```
public static int getZaehler() {
    return zaehler }
```

Aufruf

Die Klasse! Kein Objekt.

```
System.out.println(Student.getZaehler())
```

10.17

Um zaehler aktuell zu halten:

Rückgabetyp.
↓
public static Student createSt() {

Zaehler++; // sind in der Klasse // Student.

return new Student(); }

Aufruf

Student studi = Student.createSt();
System.out.println (Student.getZaehler)

Haben Student in Student. Vergleiche
↓
public class Knup {
public int Wert;
public Knup ref; }
↑

Problem: Was ist wenn auch irgendwo noch new Student() nicht innerhalb createSt() ?

Abbildung 10.5 ist jetzt die aktuelle Klasse Student.

Weitere statische Komponenten einer Klasse sind Konstanten.

Als Beispiel das Studienfach: gemäß Tabelle 10.1.

Erweiterung der Klasse Student:

```
private int fach;

public int getFach() {
    return fach; }
```

Zugriffsmethode.

```
public void setFach(int fach) {
    this.fach = fach }
```

(10.19)

2 Möglichkeiten: Klassenmethode

```
Student $studi = Student.createSt();
studi.setFach(3)
```

Zustandsmethode

- Wenig schreibgeschützt.
- Fehleranfällig.

Abhilfe mit Konstanten in der Klassendeklaration gemäß:

```
private final static int
        MAXID = 1;
```

Haben auch innerhalb einer Methode die Möglichkeit der Konstantendeklaration:

```
final int MAX = 5;
```

(siehe S. 67)

Beispiel gemäß S. 265. Aufruf

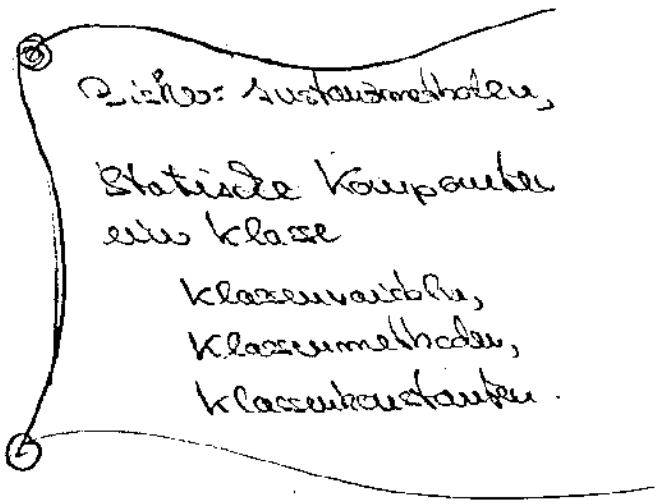
also

Student.Informatikstudium;

Dagegen gibt

Student.Zulstudium = 2;

Fehlermeldung:



Wir kommen zur Instantiierung.

= Erzeugung von Objekten. Siehe

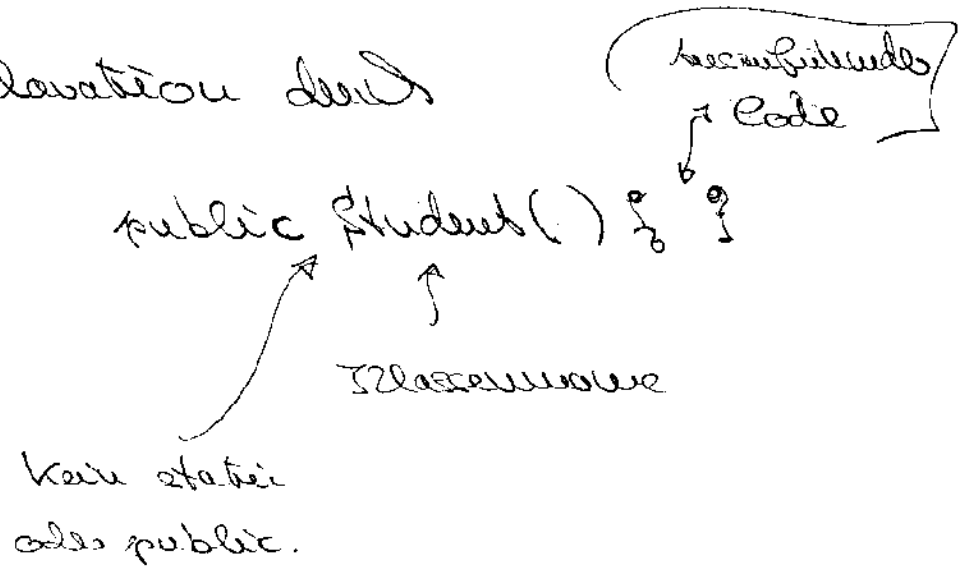
- new Student(), new PKW()
- createPkt().

Klassenmethode.

Konstruktoren erlauben es, den Erzeugungsprozess eines Objekts zu steuern.

Bisher nur `Student()`.

Deklaration durch



Standardkonstruktor, Defaultkonstruktor.

Aufruf wie gehabt

```
Student studi = new Student();
```

Ein anderer Konstruktor:

```
public Student() {
    zaehler++
}
```

Dann

```
public static Student create Student() {
    return new Student ();
}
```

↑

Erhöhung von
Zähler implizit
dabei.

Überladen von Konstruktoren,

Seite S. 269, 270, 271.

10.23

Zur Objekterzeugung.

Zu der Klassenelevation

private String name = "Dirk".

Aber im Konstruktor

this.name = "Namenlos".

Wenn ich das Ergebnis zwei

Student stud = Student.createStudent();

10.24

Was geschieht genau beim Aufruf eines Konstruktors ?

① Speicherplatz für Instanzvariablen.

② Standardwerte auf Instanzvariablen. (Tabelle 10.2, Seite)

③ Konstruktor wird mit seinen Parametern ausgeführt.

ⓐ 1. Zeile this ( ), dann letzten ausführen. Dann Rest des Konstruktors.

ⓑ 1. Zeile nicht this, dann erst alle Initialisierungen der Klassendeklaration, dann Rest des Konstruktors.

wird nach 2. ausgeführt wenn nicht 1. Zeile this (Überlader)

11. Vererbung und Polymorphismus -

das fortgeschrittene Konzept

mit Klassen in Java

Siehe Kap. 9, Kap. 11, Kap. 10.4.4.

Kap. [illegible] 4.4 [illegible]

Zusätzlich Kap. 9.3.2 aus Buch.

Das Konzept der Vererbung

```
public class Adresse {
    public String name;
    public String strasse;
    public int hnr;
    public int plz;
    public String ort;
    public String mail;
    public String kom.
```

11.2

Walter Adresse erweitern.

public class FaxAdr. extends Adresse {

public String tel

public String fax ;

UML Diagramm in Abbildung 9.5.
Die Ist-ein Pfeil !

FaxAdr. übt die Eigenschaften von
Adresse. Vererbung. Umgekehrt ist
Adresse eine Generalisierung von FaxAdr.
FaxAdr. auch eine Spezialisierung
oder Erweiterung von Adresse.

Adresse Superklasse.
FaxAdr. Subklasse.

(1.39)

Polymorphismus - Überschreiben einer Methode der Superklasse in der Subklasse

Zusammenfassend die 4 Säulen der objektorientierten Programmierung in Abbildung 9.2.

Machen wir einige Beispiele:

Aufgabe 10.4

a) Keine Zugriffsmethode für Geburtsjahr da vorhanden. Kein direkter Zugriff auf die private Variable der Superklasse.

11.4

public Taxi (int z) {

super(z) // Konstruktor der
// Superklasse, statt
// this auch möglich

}

public boolean validateNummer() {

Führen den Test mit der
Komponentenvariable nummer
durch und entsprechende Ausgabe }

Damit wird innerhalb (...)

public void setNummer (int n)

bei z.B. Nummer und

t = new Konstruktor Student ( )

das neue validate Nummer genommen.

11.5

Die Instanz bestimmt dann, welche Methode genommen wird. Man erkennt den Sinn von Instanzmethoden.

Aufgabe 10.4

Die Klasse Klang:

public Klang(int b, int a)

ein Konstruktor. Beachte den Unterschied zu

public ref Klang;

```
public String toString()
    :
public void mehrPower(int b)
    :
```

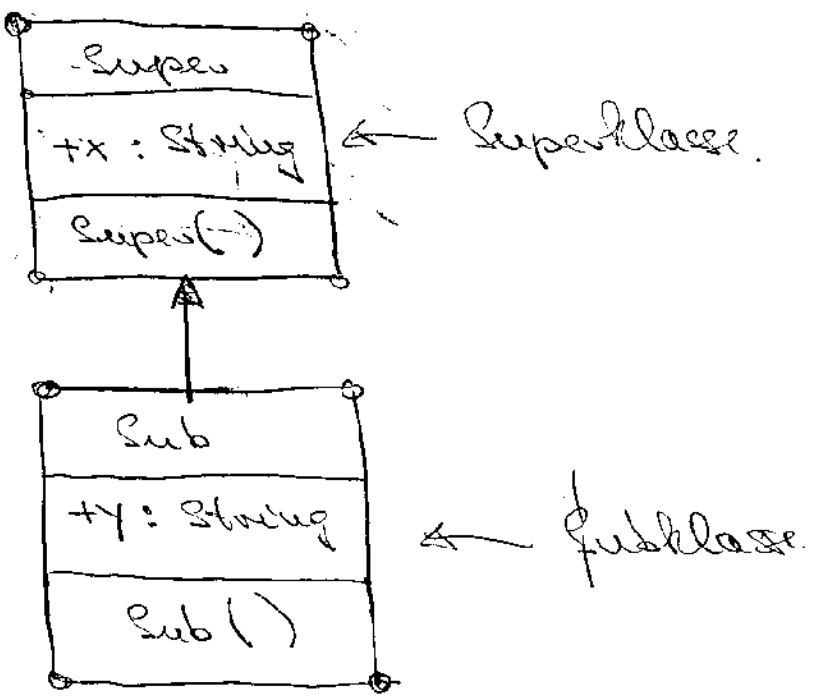
Das sind dann 2 Instanzmethoden.

Klasse Brad

```
Intelligenter Konstruktor
public Brade(int l, int b, int h) {
    super(b,h) // Entsprechender
               // Konstruktor der Superklasse.
}
```

Wie verhält es sich mit
der Objekterzeugung bei
Vererbung? Dazu noch einmal
Klasse 10.4.4. Ohne Vererbung
auf S. 10.24.

Situation jetzt:



Was passiert bei einem
Aufruf des Konstruktors der Subklasse

Sub Instanz = new Sub( )

① Speicherplatz für Instanzvariablen organisieren.

Hier heißt x und y.

② Defaultwerte für Instanzvariablen
nach Tabelle 10.2.

③ Aufgerufenen Konstruktor mit
Parametern ausführen.

Hier kann Sub( ) in der
Konstruktor von Sub ersten Zeile enthalten
→ this (...) oder
Von Super → super (...).

11.9

(a) 1. Zeile keine andere Konstruktor (nicht this(..) noch super(..)) so:
Wenn Superklasse vorhanden, Ergänzung des Konstruktors der Superklasse (Standard Konstruktor)

super();

wieder nach den Regeln (a), (b), (c).
Danach alle Initialisierungen der Klasse. Dann Rest des Konstruktors Sub().

Konstruktor nur einmal in erste Zeile.

(b) 1. Zeile super(...) so: Konstruktor der Superklasse, Dann Initialisierungen der Klasse, dann Rest des Konstruktors

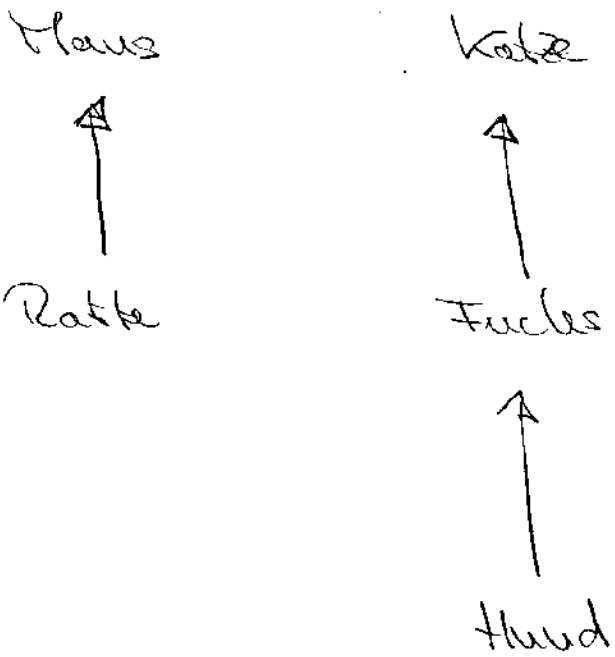
(c) 1. Zeile this(...) dann entsprechender Konstruktor, Dann Rest des Konstruktors.

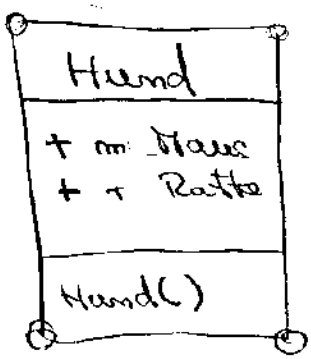
gemäß (c), der nächste Initialisierung.

Ausführung eines Konstruktors: Erst Initialisierungen der Deklaration, dann der Konstruktor selbst!

11.10

Aufgabe 10.5





UML Diagramm

(11.11)

Aufgabe 10.7

Fehler 1

Rückgabe null. (null ist Initialisierungswert für den Typ String).

Fehler 2

Alter als String wird gewandelt.

gibt die Adresse aus. Nicht die Werte der Instanzvariablen!

Fehler 3

gibt ein valides aus. oder Compilerfehler. wenn Fehler 2 nicht behoben.

Fehler 4 ✓

Fehler 5

Rückgabetyp void bei Konstruktor fehlt.

Fehler 6

Kein Defaultkonstruktor.

Was bewirkt man new (Hund()) ? (11.11)

Hund() → Fuchs()

→ Katze

Ausgabe : Katze
Fuchs // wg. super
Maus // wg. Initialisierung
Maus // wg. Initialisierung
Ratte
Hund.

M.12

Fehler 4 ✓

Fehler 5

Rückgabetyp void bei Konstruktor falsch.

Fehler 6

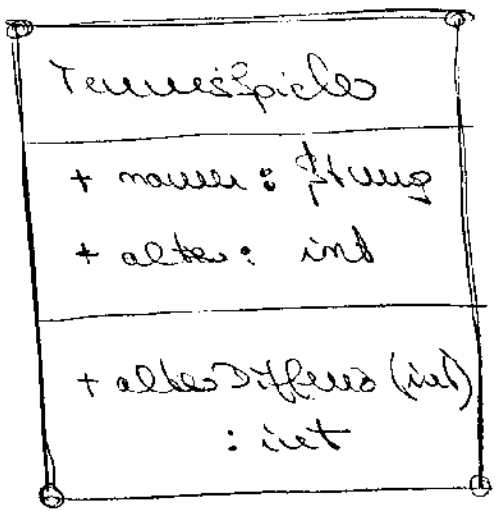
kein Defaultkonstruktor vorhanden.

public Fehler6() { }

ergänzen.

Aufgabe 10 &

a)



b) Instantiierung von main aus nicht möglich.

d) Über ein Objekt.

c) Wenn es so dann auch noch

Tennisspieler() { }.

g)

:

public TennisSpieler reifalge;

public int stabnummer;

:

h) public static int folgeNr.

k) Klassenvariablen dürfen nur auf Klassenvariablen zugreifen können der Main-Operator nicht. Es sei denn, es genommen selbst ein Objekt.

Weiter in Kapitel 11.

Nicht instanziierbar

public abstract class Wölfung

public abstract double dolBe()

// soll Wolf die Instanz

// in Dollar zurückgeben.

Abbildung 11.11 kurz für abstrakt.