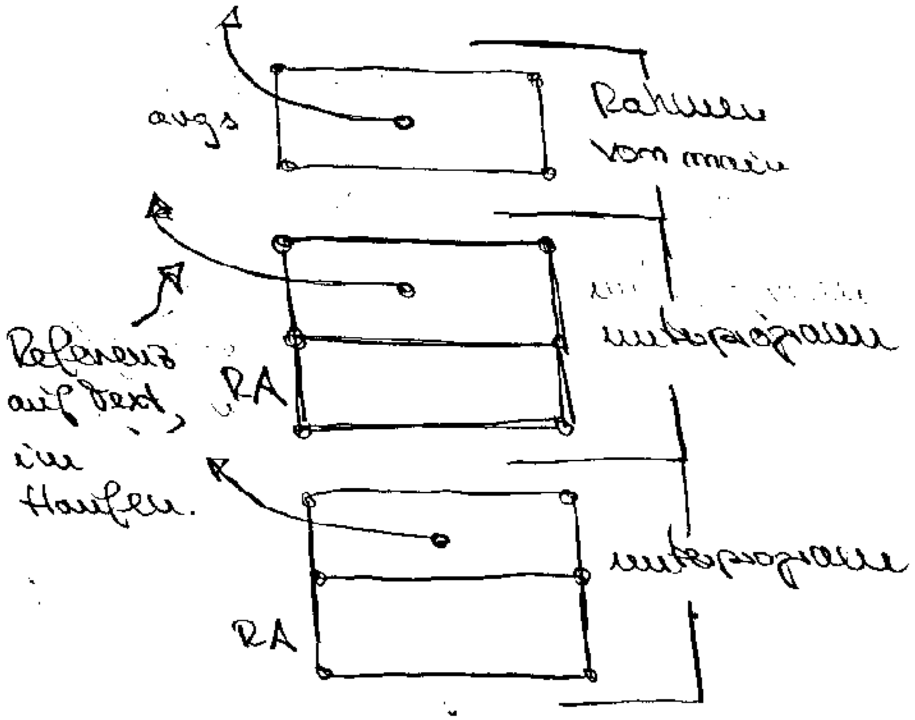
# 8. Das Wunder der Rekursion

Zunächst werden S. 176, S. 177 und S. 178 aus Ratz, Scheffler, Seese besprochen.

Dazu ein paar Anmerkungen:
Was geschieht mit unserem Laufzeitkeller beim Programm Unendlichkeit.java?



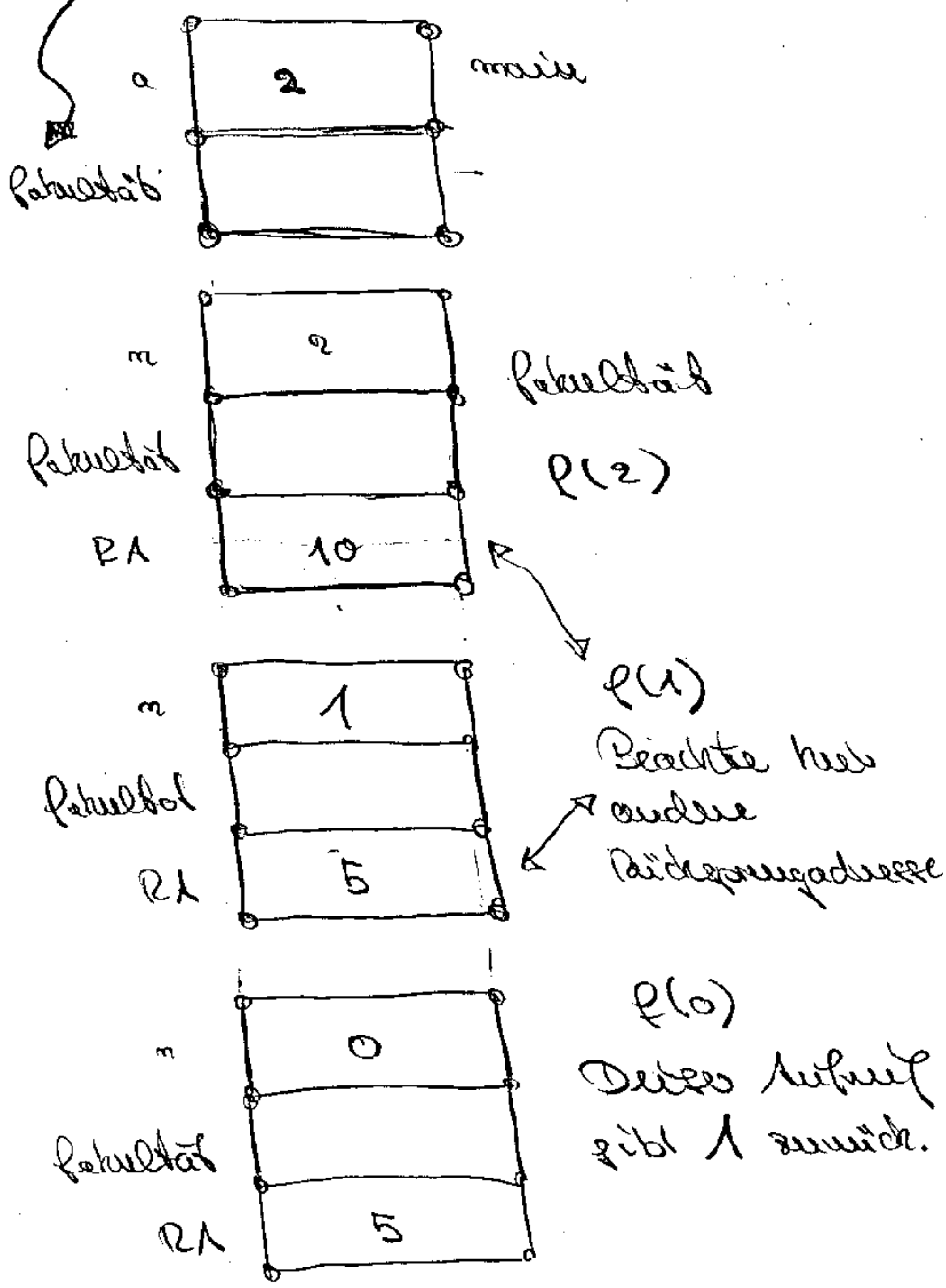
8.2

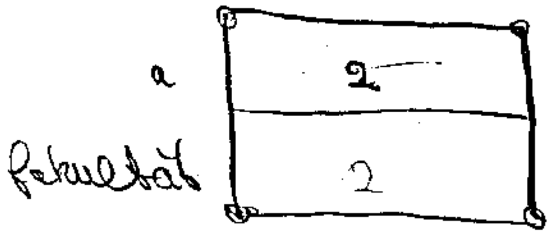
Beim Programm der Fakultät

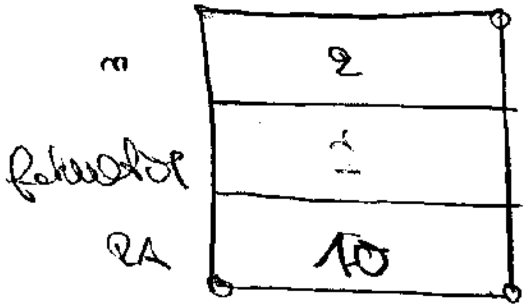
```
public class Fakultät {
  public static int fakultät(int n) {
    if (n == 0)
      return 1;
    return n * fakultät(n-1);
  }
public static void main(String[] args) {
int a;
a = readInteger("");
fakultät(a);
}
```

Rekursion
(zurücklaufen)

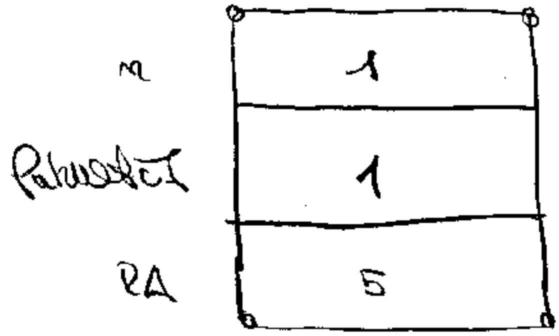
Platz für zurückgegebenen Wert.



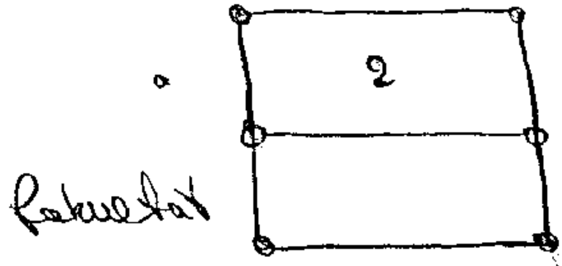


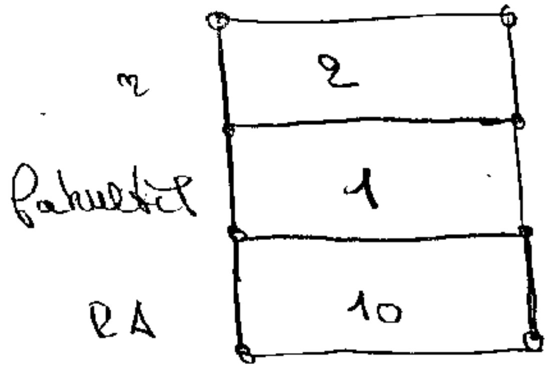




f(2)




f(1)
1 = n · fakultät
zurück, wegen
RA 5 bei f(0)!

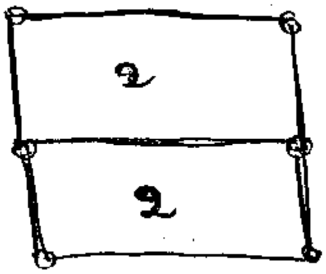



2 = n · fakultät
zurück.
Wegen RA 5
bei f(1)!

a

fakultät




– Wegen RA 10.

Korrektheit: Induktion über n zeigt die Aussage: Ein Aufruf von fakultaet(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

das zweite Argument von ggT(a,b).

Zeigen: Für alle $a \geq b$ ist

ggT(a,b) korrekt.

Ind. Anf.: $b = 0$, dann klar.

Ind. Schluß: Sei $b$ fest, $b \gneq 0$.
Gelte die Behauptung für alle (!) $b'$ mit $b' \lneq b$. Wir zeigen sie für $b$. Es ist die Menge der gemeinsamen Teiler von $a$ und $b$ gleich der von $b$ und $a \% b$. (vgl. frühere Version des Euklidischen Algorithmus).
Also folgt die Behauptung, da auf $ggT(a,b)$ die Ind.-Vor. anwendbar ist.

(8.8)

Der Binomialkoeffizient ist definiert durch: für $n \geq k \geq 0$ ist

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdot\ldots\cdot(n-k+1)}{k!}$$

für $k \geq 0$, $k > n$ ist $\binom{n}{k} = 0$.

Also ist für $k \neq 0$

$$\binom{n}{k} = \frac{n}{k} \cdot \frac{(n-1)\cdot\ldots\cdot(n-k+1)}{(k-1)!}$$

$$= \frac{n}{k} \cdot \binom{n-1}{k-1}.$$

Kombinatorische Interpretation:

$\binom{n}{0} = 1$

$\binom{n}{1} = n$ $\qquad \sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \binom{n+1}{2}$

$\binom{n}{2} = \frac{n(n-1)}{2}$

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$$

$$\vdots$$

$$\binom{n}{n} = 1$$

Es gilt:

$\binom{n}{k}$ = # Teilmengen mit genau $k$ Elementen aus $1,\dots,n$.

Beweis 1: Jede Teilmenge mit $k$ Elementen gibt uns folgender 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.

Aber: Was haben wir gewählt? Folgen $(a_1, a_2, \ldots, a_k)$, geordnet, aus $k$ verschiedenen Elementen. Die Menge, ungeordnet, $\{a_1, \ldots, a_k\}$ kommt von $k!$ Folgen: Alle Permutationen von $a_1, \ldots, a_k$. Also gilt

\# Teilmengen mit genau $k$ Elementen aus $1, \ldots, n$ $= \dfrac{n(n-1)\cdots(n-k+1)}{k!} = \binom{n}{k}$.

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: Sei nun $n > 0$. Gelte die Aussage für $n-1$ und alle möglichen $k$. Für $n$ und $k=0$ gilt die Beh. Sei $k > 0$.

# $k$-elementige Teilungen von $\{1,\dots,n\}$, die $n$ enthalten

$= \#(k-1)$-elementige Teilungen 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.

$$\binom{n-1}{k-1} + \binom{n-1}{k}$$

$$= \frac{(n-1)\cdot\ldots\cdot(n-1-(k-2))}{(k-1)!} + \frac{(n-1)\cdot\ldots\cdot(n-1-(k-1))}{k!}$$

$$= \frac{(n-1)\cdot\ldots\cdot(n-k+1)}{(k-1)!} + \frac{(n-1)\cdot\ldots\cdot(n-k)}{k!}$$

$$= \frac{k\,(n-1)\cdot\ldots\cdot(n-k+1) + n\,(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$$

$$- \frac{k\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$$

$$= \binom{n}{k}.$$

Zwei rekursive Programme:

Bin1(int n, int k)

return (Bin1(n-1, k-1) + Bin1(n-1, k))

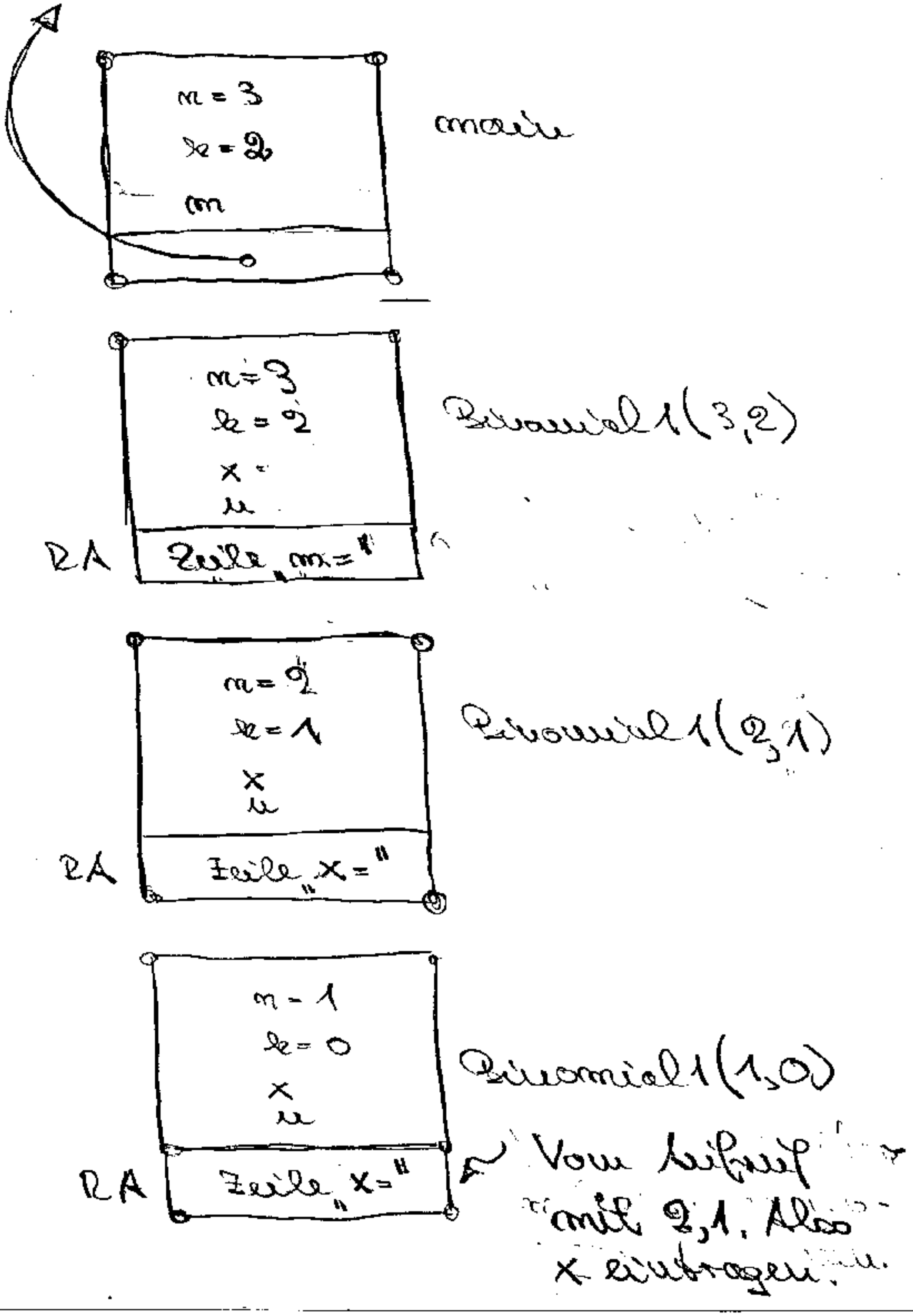
Wie Abbruchbedingung? Einmal

$k = 0$ einmal $n = k$.

Vergleiche Programme Bin1.java, Bin1a.java. Bin1.java: Speicherzugriffe:

Fehler wegen Nichtabbruchs der Rekursion.

Laufzeitkeller bei Binomial1(3,1)



⋮

n = 2
k = 1
x = 1
u

RA Zeile x = "

Binomial1(2,1)

n = 1
k = 1
x
u

RA Zeile u = "

Binomial1(1,1)

⋮

n = 2
k = 1
x = 1
u = 1

Zeile x =

Binomial1(2,1)

Jetzt wird
x + u auf
x übergeben.

⋮

| |
|---|
| n = 3 |
| k = 2 |
| x = 2 |
| u |
| RA |

Binomial1(3,2)

| |
|---|
| n = 2 |
| k = 2 |
| x = 2 |
| u |
| RA "u = " |

Binomial1(2,2)

⋮

| |
|---|
| n = 3 |
| k = 2 |
| x = 2 |
| u = 1 |
| RA "m = " |

Binomial1(3,2)

← Rückgabe von 3 = 2+1 an m.

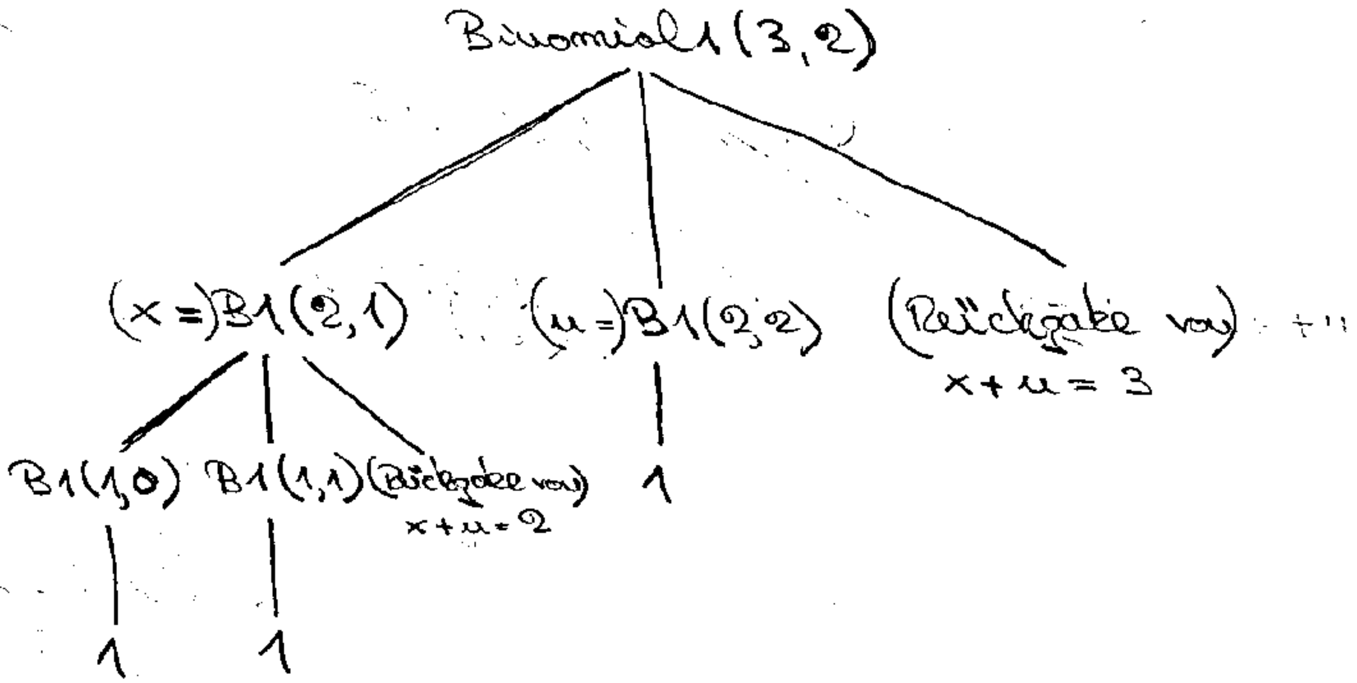
| |
|---|
| n = 3 |
| k = 2 |
| m = 3. |
| A |

main

Neu.



Wichtig ist die Darstellung des <u>Ablaufs</u> eines Programms, insbesondere bei rekursiven Methoden als sogenannten 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); $r+u=2$

  ⋮ • Vater analog.

- Nachfolger von B1(2,1) sind

  B1(2,1) (!), die Kinder und

  die Kinder der Kinder usf.

- Die Kinder sind geordnet von

  links nach rechts: Erstes Kind,

  Zweites, ... • Erster Teilbaum, zweiter Teilbaum, ...

- B1(3,2), B1(2,1), 1, 1, ... heißen

  auch Knoten. Man unterscheidet

  innere Knoten und Blätter.

  Die Blätter sind:

  1, 1, 1, $x+u=3$.

Ausführung von B1(3,2)

=

Ausführung der Kinder von B1(3,2) von links nach rechts:

B1(2,1); B1(2,2); x+u=3 ausrechnen.

Ausführung von B1(2,1) dann analog ...

Ausführung von B1(3,2)

=

Abarbeiten des Baumes in Postorder:

<u>Erst</u> die Kinder, links nach rechts, <u>dann</u> der Vater.

8.19

Das entspricht genau dem Pulsieren des Laufzeitkellers 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 Kind.

(3,2) $x+u=3$ // Drittes Kind von (3,2).

(3,2) // Hier ist (3,2) fertig.

Nun zum Programm Bin2.java.

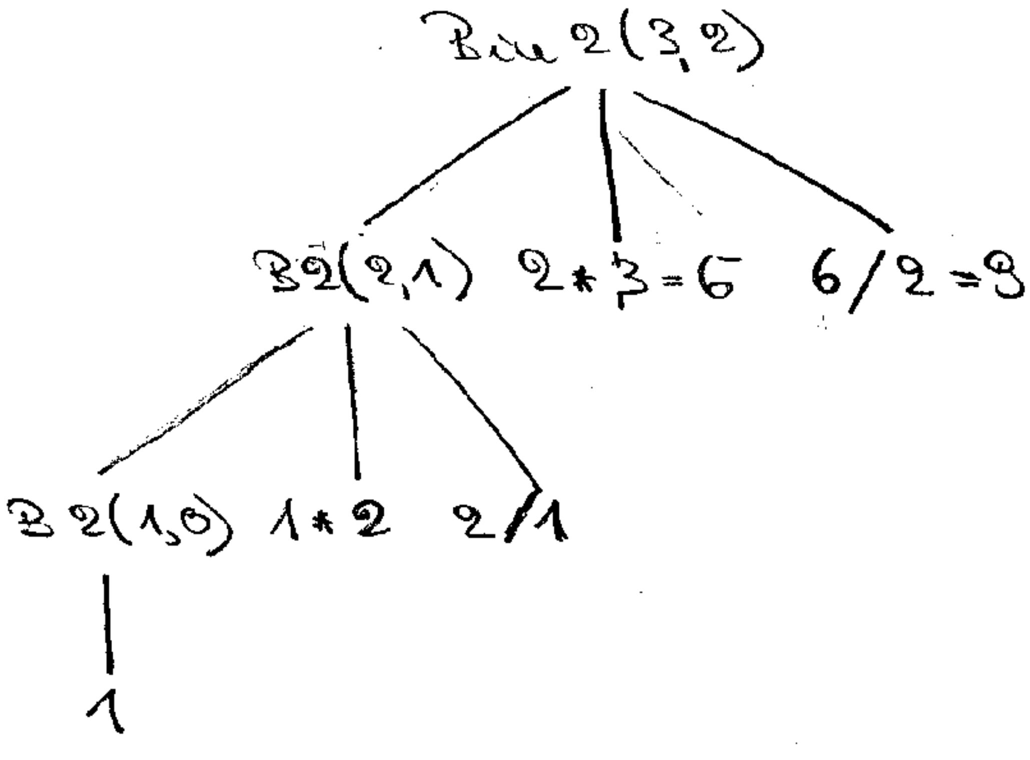
Das geht nach dem Prinzip vor:

Bin2( int n, int k)

return ((Bin2 (n-1, k-1) * n)/k)

Abbruchbedingung k = 0.

Baum bei Eingabe von 3 und 2:



Laufzeittabelle:

(3,2)

(3,2) (2,1)

(3,2) (2,1) (1,0) // Übergabe von 1

(3,2) (2,1) // Übergabe von $1 \times 2/1 = 2$

(3,2) // Übergabe von $2 \times 3/2$

// 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 $a = 4$, $b = 3$, $m = 100$.

RP(4,3)

RP(4,2) $\quad 4 * 16 = 56.$ $\quad$ b ungerade

RP(4,1) $\quad x * x = 4 * 4 = 16$ $\quad$ b gerade

RP(4,0) $\quad 4 * 1$

1

Es wird immer nur 1 rekursiver Aufruf ausgeführt, obwohl 2 im Programm sind.

8.24

Am Laufzeitkeller läßt sich noch einmal die Bedeutung der Rücksprungadresse sehen:

(4, 3, m)

RA

(4, 3, m) (4, 2, m, b) // Da 3 ungerade.

An a bezogen

(4, 3, m) (4, 2, m, b) (4, 1, m, a) // auf Vorgängerrahmen geht es weiter.

(4, 3, m) (4, 2, m, b) (4, 1, m, a) (4, 0, m, b)

// Übergabe von 1.

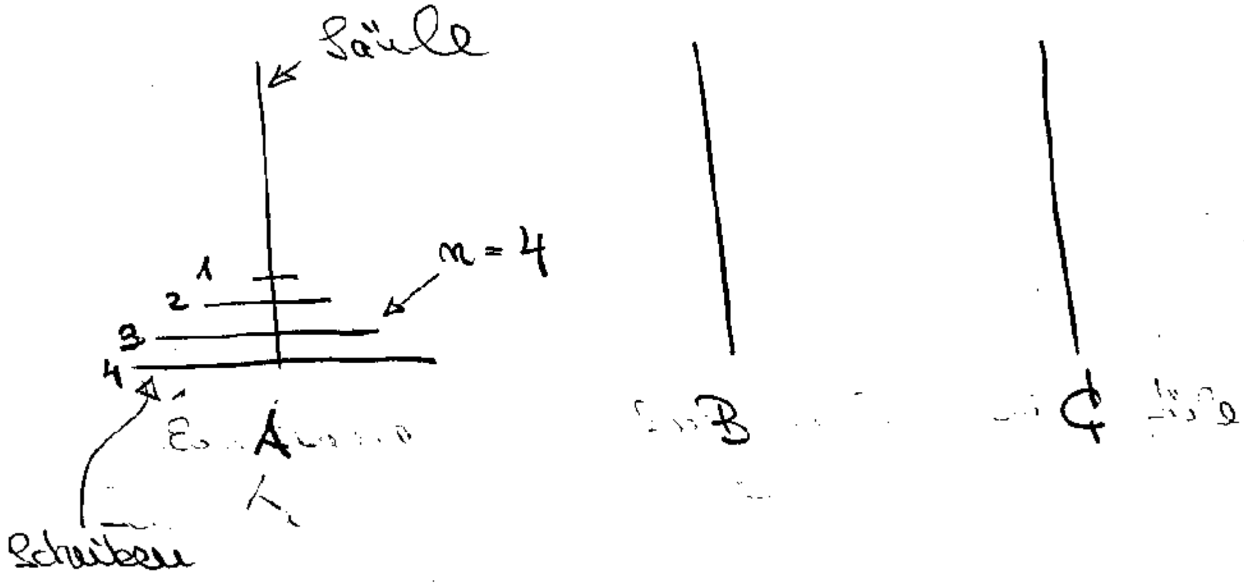
(4, 3, m) (4, 2, m, b) (4, 1, m, a) // Übergabe 4.

(4, 3, m) (4, 2, m, b) // Übergabe von 16.

(4, 3, m) // Übergabe von 56

Das Problem der Türme von Hanoi ist das Paradebeispiel für die Anwendung der Rekursion.

Ausgangssituation

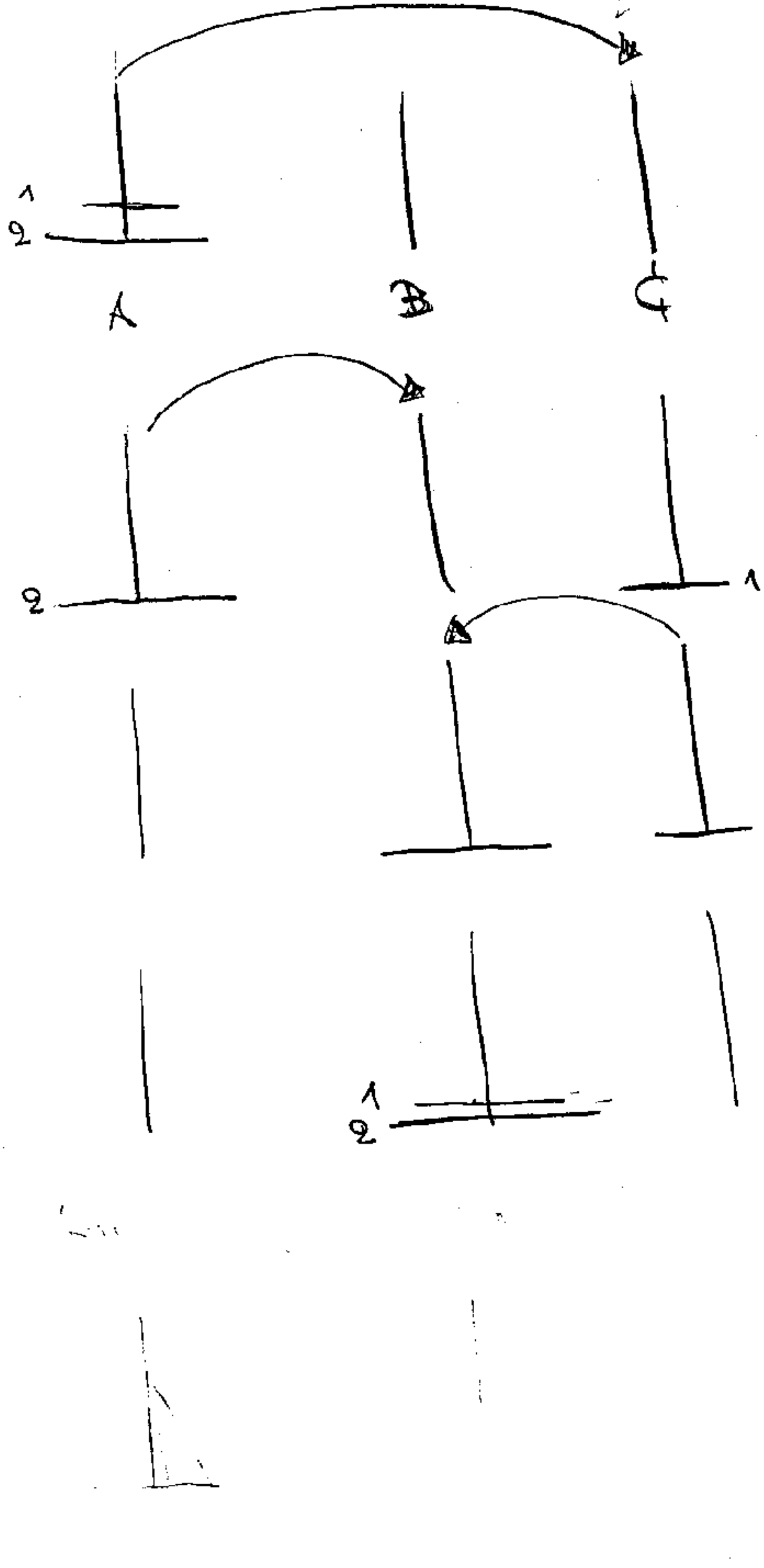



Aufgabe: Bringe die n Scheiben von A nach B nur mit Hilfe von C.

Beschränkung: Pro Schritt eine Scheibe. Nie größere über kleinere Scheibe.

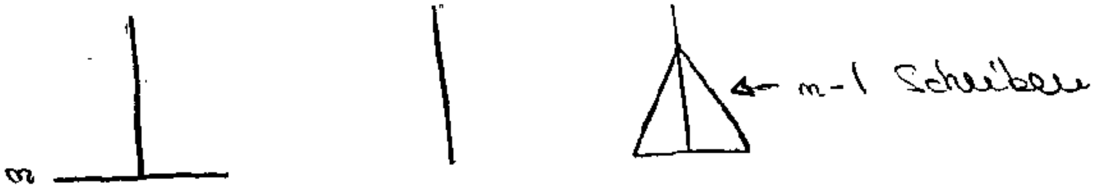
Mit $n = 2$ :

2.26



Im allgemeinen Hanoi(A, B, C, n)


Hanoi(A, C, B, n-1)




Scheibe n von A nach B


Hanoi(C, B, A, n-1)

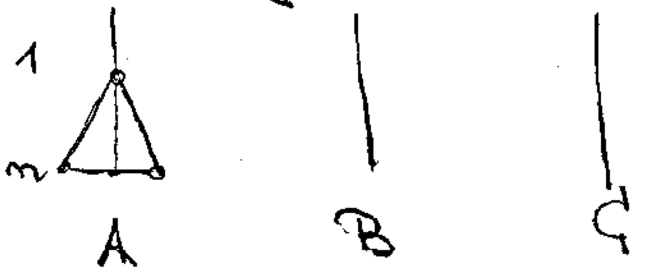
Programm gemäß

```
Hanoi(A, B, C, n)
if (n = 1)
    "Scheibe n = 1 von A nach B";
    return;        // Beachte: return beendet
                   // den aktuellen
                   // Aufruf.
if (n > 1) {
(a)  Hanoi(A, C, B, n-1);
(b)  "Scheibe n von A nach B";
(c)  Hanoi(C, B, A, n-1);
     // return unnötig, da Ende erreicht.
}
```

8.22

Zur Korrektheit ist zu zeigen:

Für alle $n \geq 1$ liefert Hanoi$(A,B,C,n)$ eine Folge von regelkonformen Umsetzungen bezogen auf

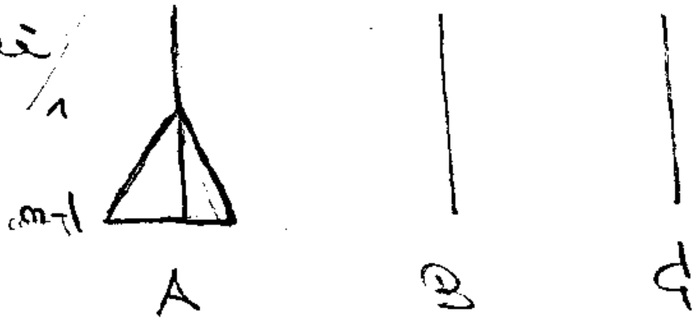



Induktion über $n$.

$n = 1$ ✓

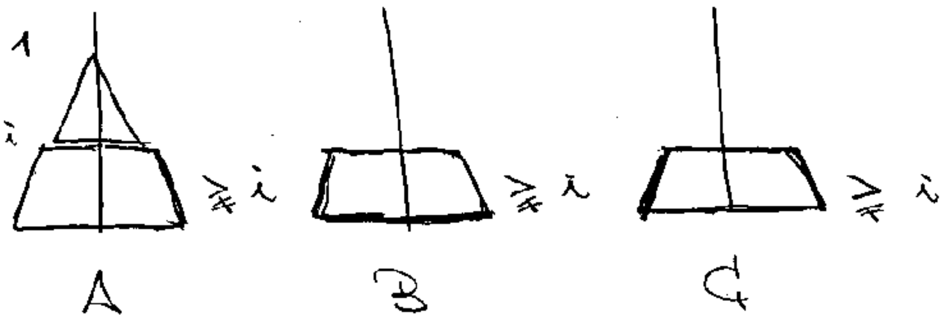
$n > 1$ Ind.-Vor. lautet nun:

Bei

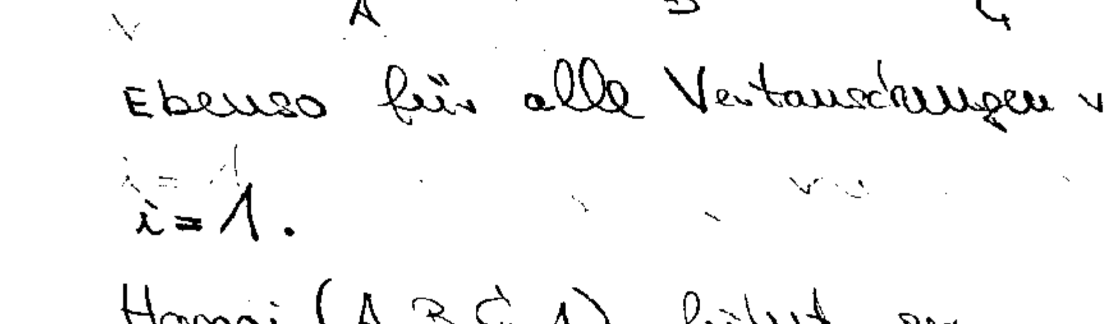



ist Hanoi$(A,B,C)$ richtig. Wir wissen nichts über Hanoi$(A,C,B)$.

Tatsächlich müssen wir wesentlich allgemeiner vorgehen und zeigen induktiv über $i \geq 1$: Für jede mögliche Situation der Art




liefert Hanoi(A,B,C,i) eine regelkonforme Folge von Umsetzungen zu




Ebenso für alle Vertauschungen von A,B,C.

$i = 1$.

Hanoi(A,B,C,1) liefert nur

„Schiebe 1 von A nach C". Vertauschungen

8.31

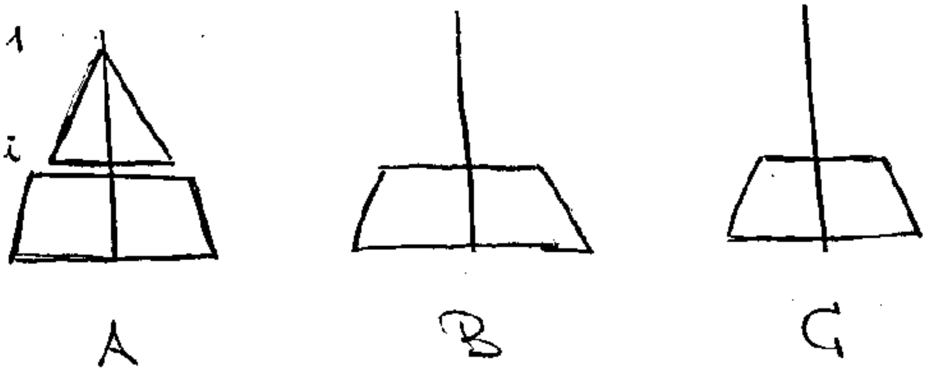
von A, B, C werden analog behandelt.

Ind.-Schluß: $i > 1$. Nach Ind.-Vor. gilt die Behauptung für $i-1$ und alle Vertauschungen von A, B, C.

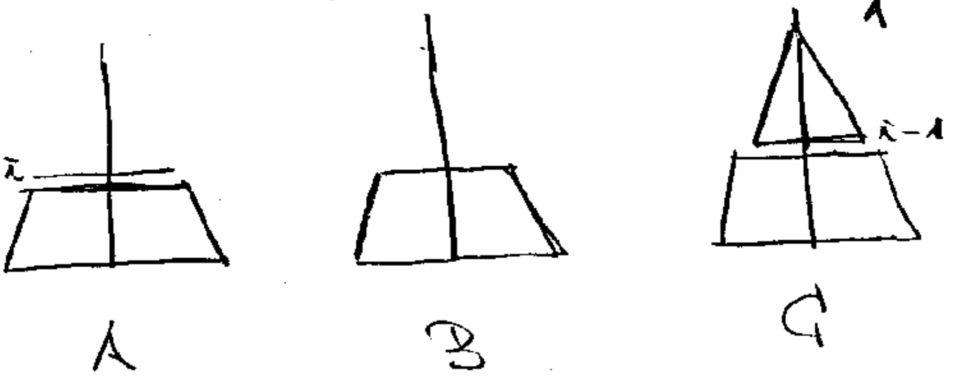
Betrachten wir den Aufruf Hanoi(A, B, C, i). Da $i \ngtr 1$ ist liefert das

Hanoi(A, C, B, i-1);
„Schreibe i von A nach B)";
Hanoi(C, B, A, i-1);

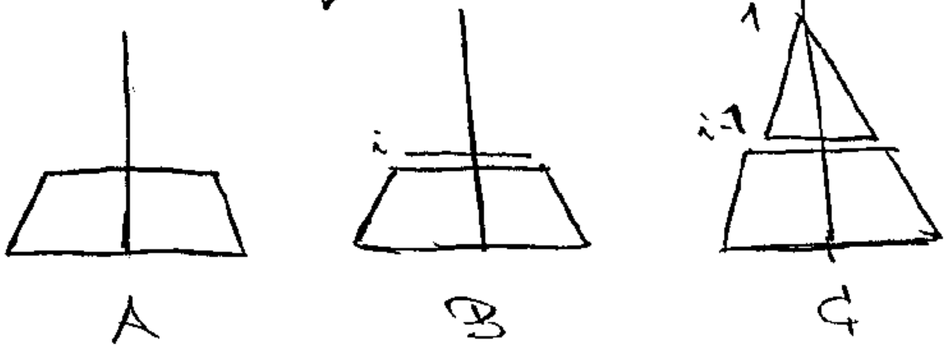
Das liefert eine regelkonforme Folge von Umsetzungen:



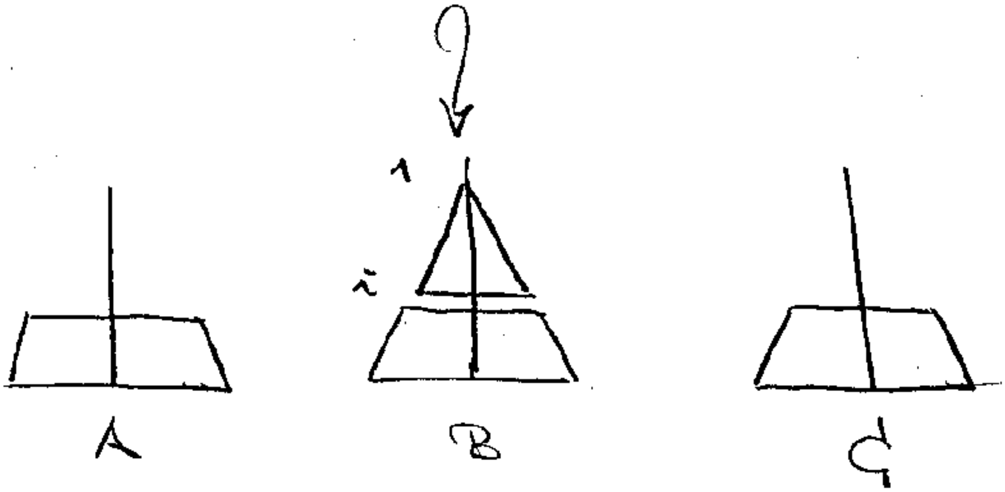

Hanoi(A, C, B, i-1)
regelkonform nach
Ind.-Vor. bzw. Vertauschungen.




„S. i von A nach B"




Hanoi(C, B, A, i-1)
Ind.-Vor.




Man beachte, daß die Ind.-Vor. für jede(!) mögliche Situation gilt, also jede Situation in der die unteren Scheiben $\geq i-1$ sind.

Programm in Hanoi.java.

Der Aufrufbaum

$$
\begin{tikzcd}
& H(A,B,C,2) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
H(A,C,B,1) \arrow[d, dash] & A \xrightarrow{2} B & H(C,B,A,1) \arrow[d, dash] \\
A \xrightarrow{1} C & & C \xrightarrow{1} B
\end{tikzcd}
$$

„Schreibe 1 von C nach B"

Tatsächlich
liegt auch Scheibe 1
oben auf C!

Verfolgen des Laufzeitkellers bei $n=2$

Rücksprungadresse an das Ende des Hauptprogramms

$(\underbrace{A, B, C, 2}_{\text{Aktuelle Parameter}}, E)$

Aktuelle Parameter

$(A, B, C, 2, E)$

$(A, C, B, 1, b)$

$A \xrightarrow{1} C$

$(A, B, C, 2, E)$

$A \xrightarrow{2} B$

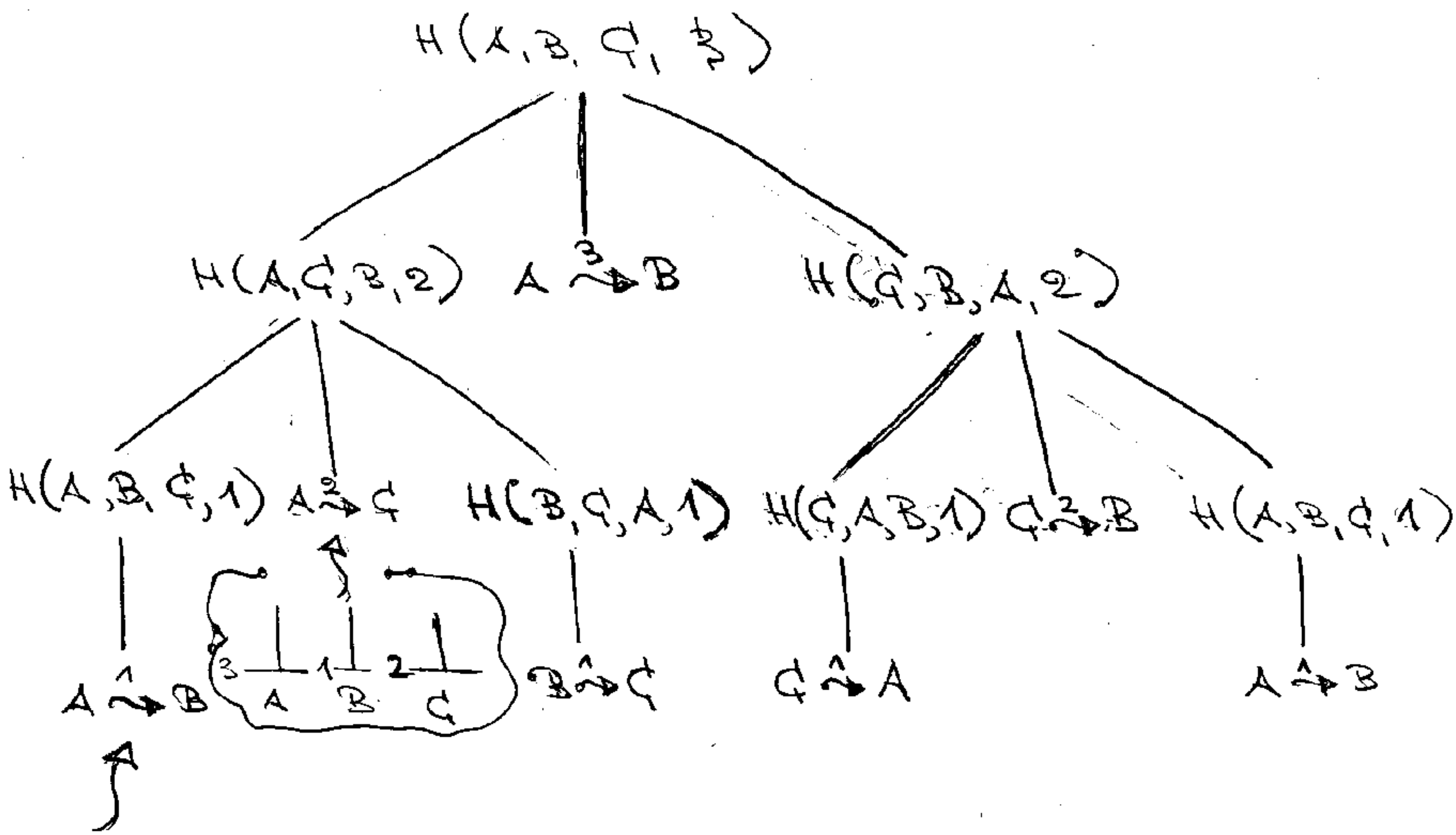
$(A, B, C, 2, E)$

$(C, B, A, 1, E)$

$C \xrightarrow{1} B$

$(A, B, C, 2, E)$

Programm zu Ende




Das wird als erstes umgesetzt:

Postorder Bearbeitung des Baums.

8.36

Laufzeittabellen zum letzten Beispiel mit Programm von S. 8.28.

← Programmende

(A, B, C, 3, E)

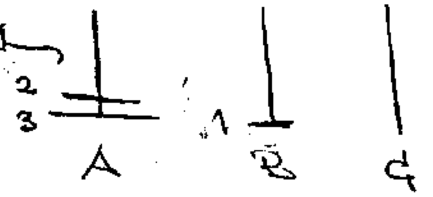
(A, B, C, 3, E) Rahmen von H(A,B,C,3)

(A, C, B, 2, b) Rahmen von H(A,C,B,2)

(A, B, C, 3, E)

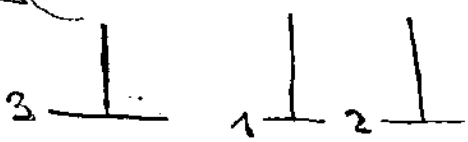
(A, C, B, 2, b)

(A, B, C, 1, b)




(A, B, C, 3, E) Aufruf mit 1.

(A, C, B, 2, b)

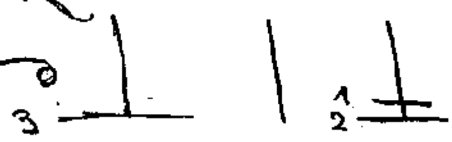



Wegen RA b oben.

(A, B, C, 3, E)

(A, C, B, 2, b)

(B, C, A, 1, E)



(A, B, C, 3, E)

Weg. RA b

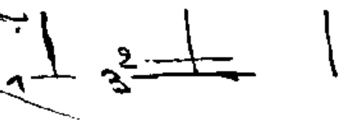
(A, B, C, 3, E)

(C, B, A, 2, E)

(C, A, B, 1, b)

(A, B, C, 3, E)

(C, B, A, 2, E)



(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 zu Ende.

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

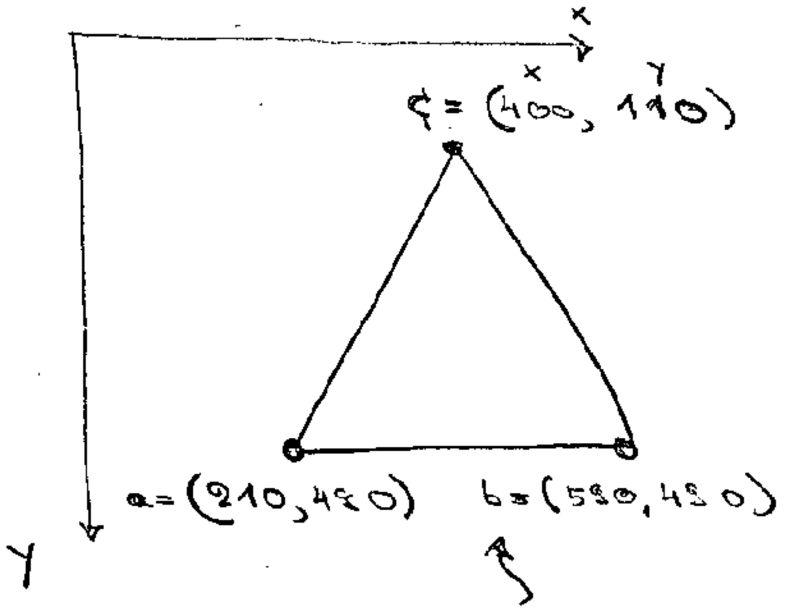
Sierpinski.java

Sierpinski.java.

# Das Sierpinski Dreieck.

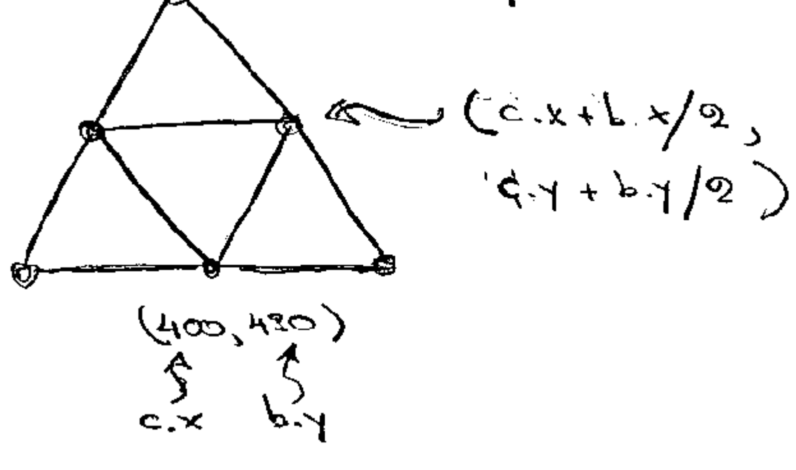
Level 0: Ein einfaches Dreieck.

Werte des Programms Sierpek.java:

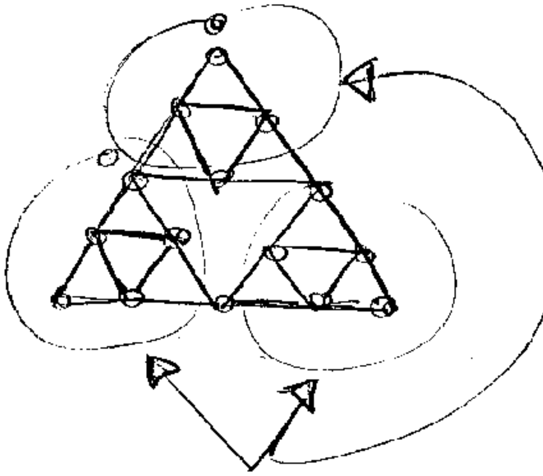



Punkt ist Objekt einer Klasse mit Komponentenvariablen x und y

Level 1:



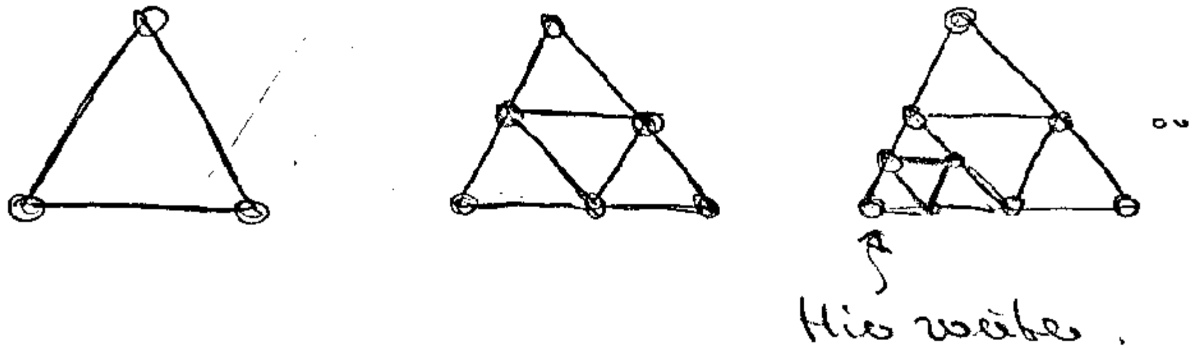
Level 2



Hier rekursiv weiter.

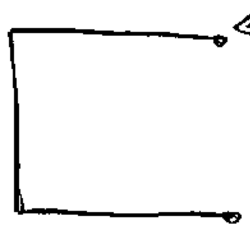
Formale Definition:
S-Dreieck von level 0 ist Dreieck.
Von level $n \geq 1$ ist Dreieck mit 3 S-Dreiecken von level $n-1$.

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:



Hier der Anfang

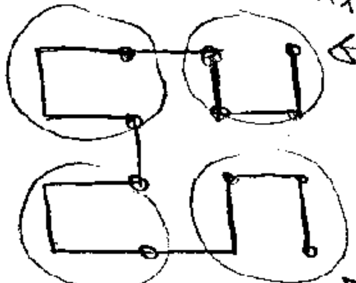
Hier das Ende.

$H_1$

Daraus dann

$H_1$

$H_1$ gedreht, Öffnung oben



Anfang

$H_1$ gedreht, Öffnung unten.

$H_1$

Ende.

$H_2$

(8.44)

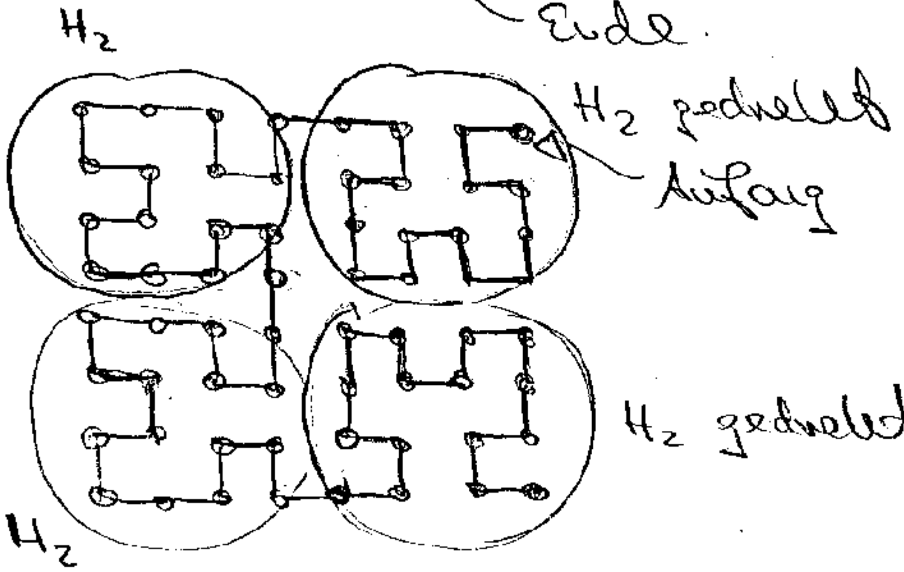
Dann bekommen wir:

← Anfang

$H_2$ $H_2$ Öffnung oben

$H_2$ $H_2$ Öffnung unten

↖ Ende




$H_3$.

So geht es weiter:

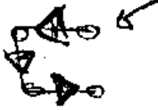
$H_3$ $H_3$ $H_4$ $H_4$

$H_3$ $H_3$ $H_4$ $H_4$ ...

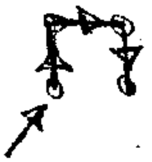
$H_4$ $H_5$

Schauen wir uns $H_3$ an, dann haben wir 4 Typen A, B, C, D von $H_1$:

Durchlaufrichtung

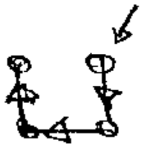


A



B

C (einmal in $H_3$, unten rechts)



D

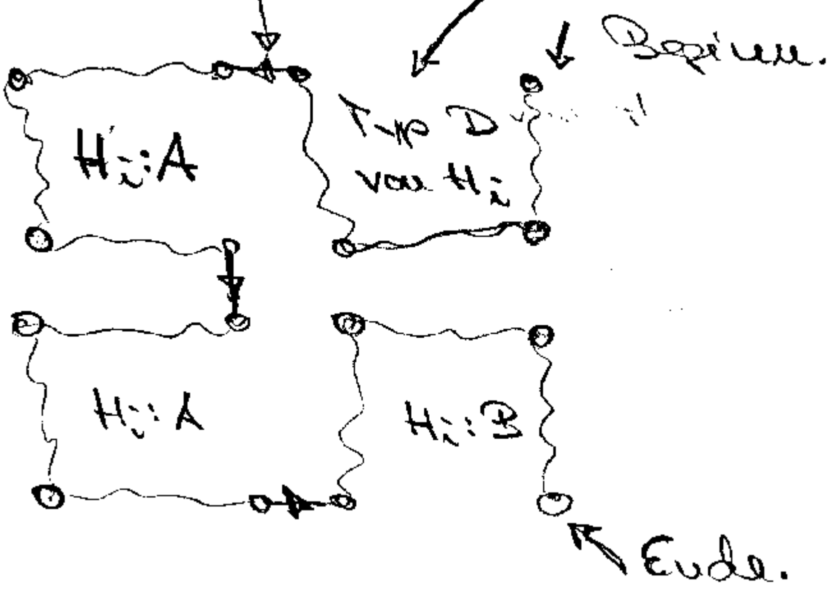
Analog definieren wir 4 Typen A, B, C, D von $H_2, H_3, \ldots$.

(8.46)

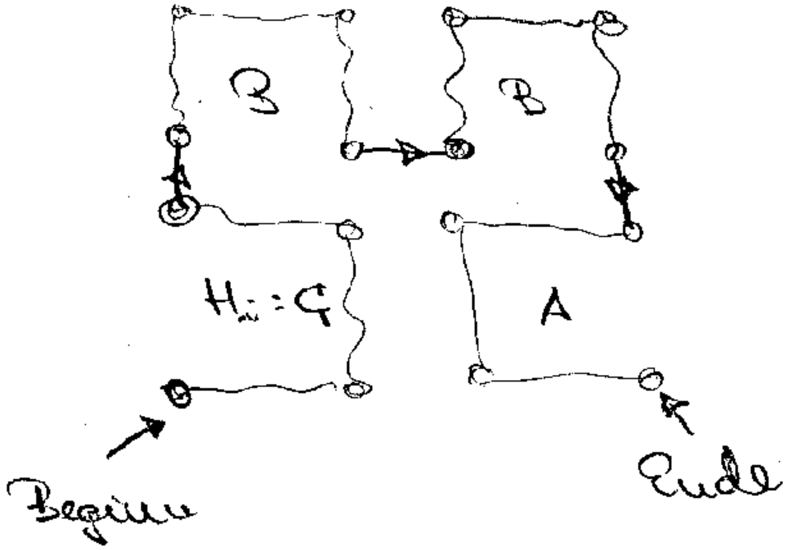
Dann ergibt sich folgende Vorschrift $H_{i+1}$ aus $H_i$ zu bauen:

Verbindung nach links.

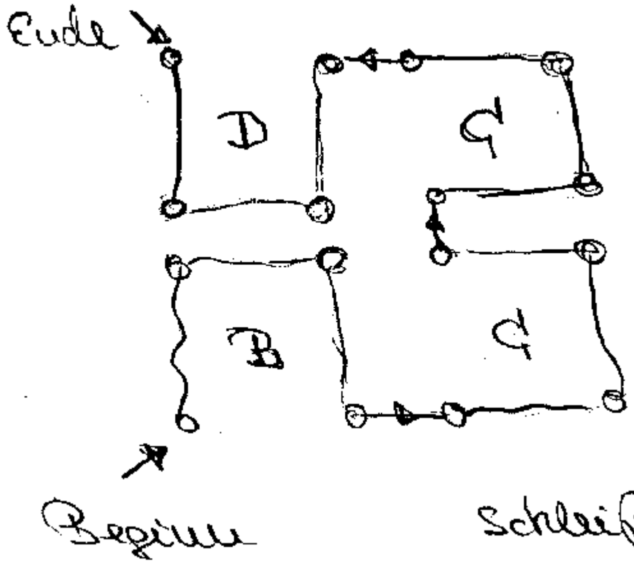
Typ A von $H_{i+1}$: Öffnung oben.



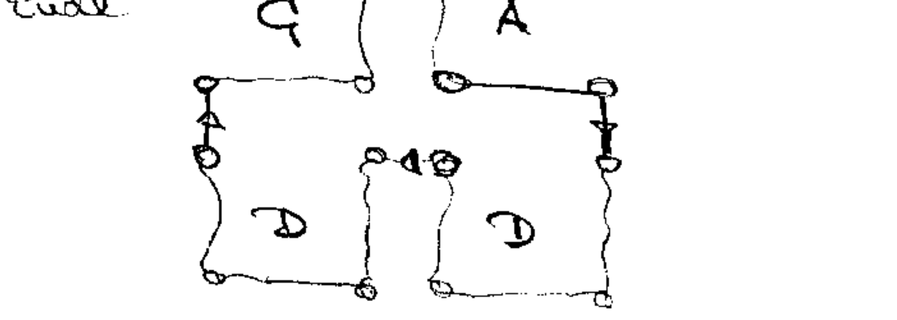
Typ B von $H_{i+1}$:




$H_{i+1}$: C




schließlich $H_{i+1}$: D



Wie zeichnet man das ganze?

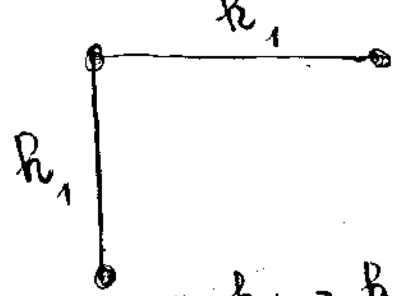
Ist $h = h_1$ gegeben, Länge bei level $= 1$.

Dann bei level $= i$ ist

$$h = h_i = h_1/2^{i-1}$$

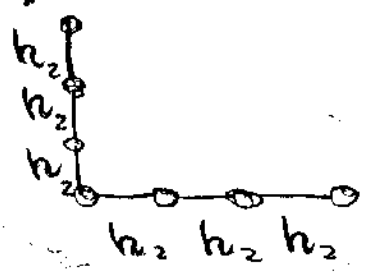
Dann haben wir:

$H_1$ :



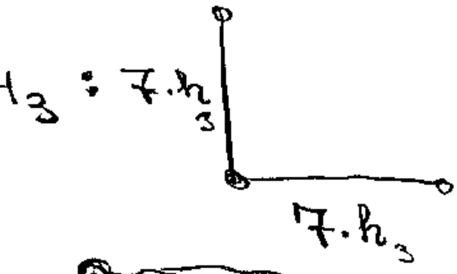

$h_2 = h_1/2$

$H_2$ :



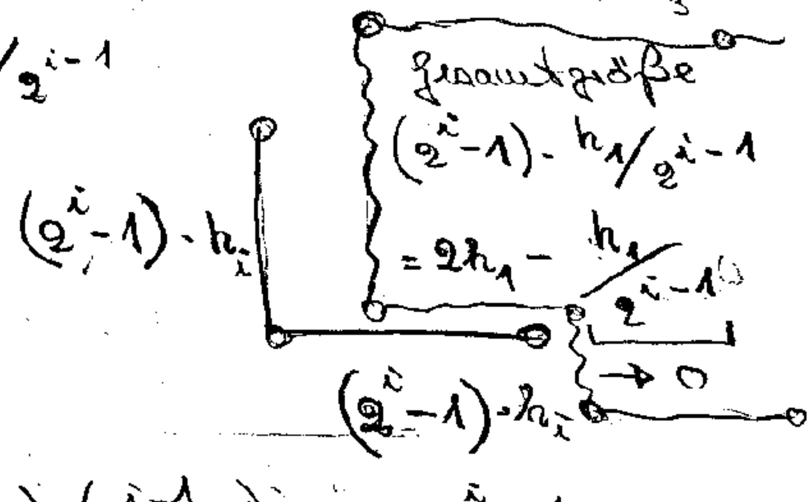

$h_3 = h_1/4$

$H_3$ :




$$h_i = h_1/2^{i-1}$$

$H_i$ :




Induktiv: $(2^{i-1}-1)+(2^{i-1}-1)+1 = 2^i-1$.

Dazu das Programm Hilber.java.

Verbesserung in Hilbertverb.java.

Methoden zum Ziehen der Linien. Die Basislänge $h = h_i$ 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 des Stiftes (stetig)
- ohne Überkreuzungen (injektiv)
- vollständig (surjektiv)

ausfüllt.

Die kombinatorische Suche, d.h. Suche von kombinatorischen Objekten wie endlichen Mengen, 0-1-Vektoren oder ähnlichem mit bestimmten Eigenschaften ist ein weiteres wichtiges 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 $n \geq 1$ 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 der Art

$\overbrace{b_0 b_1 - b_{d-1}}^{\text{Das sind } d \text{ Stück.}}$ Alle der Länge n-d

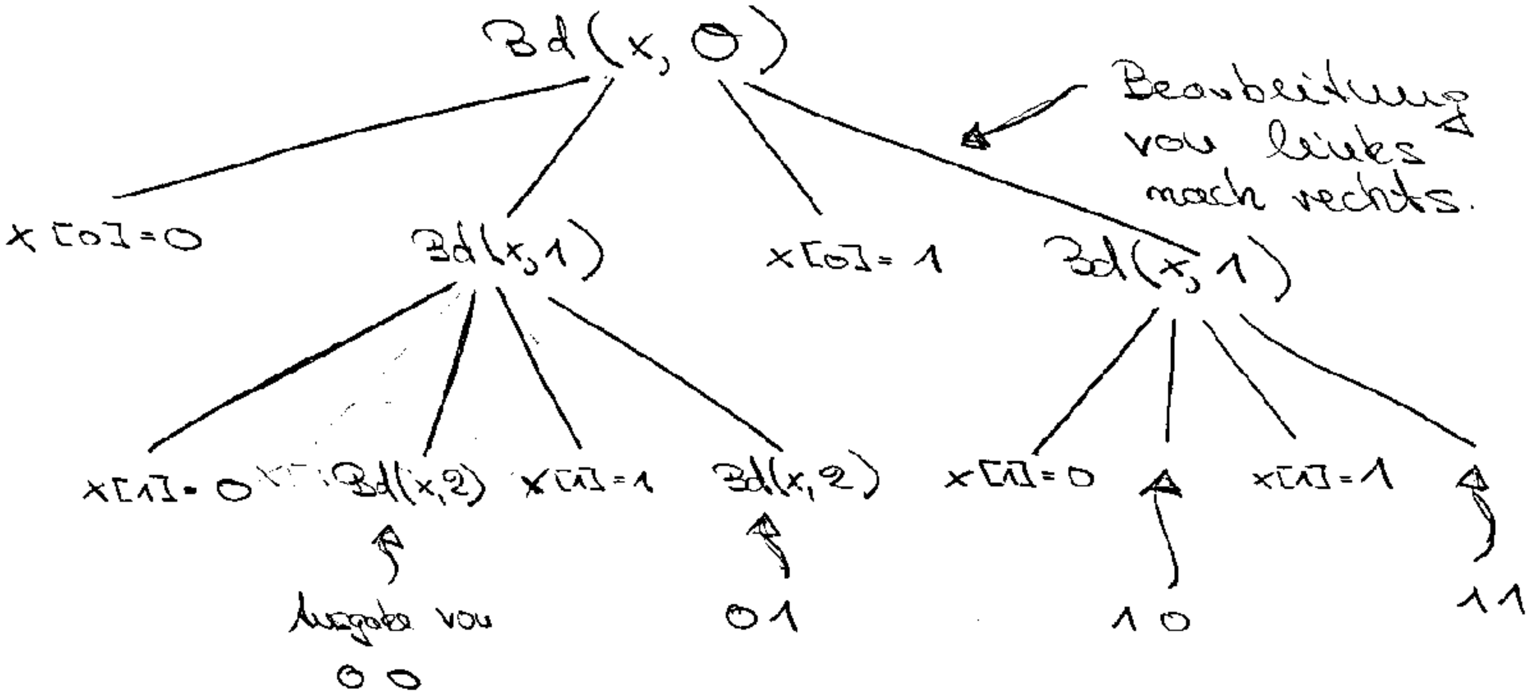
ergeben sich als

$b_0 b_1 - b_{d-1} 0$ „Alle von $n-(d+1)$"

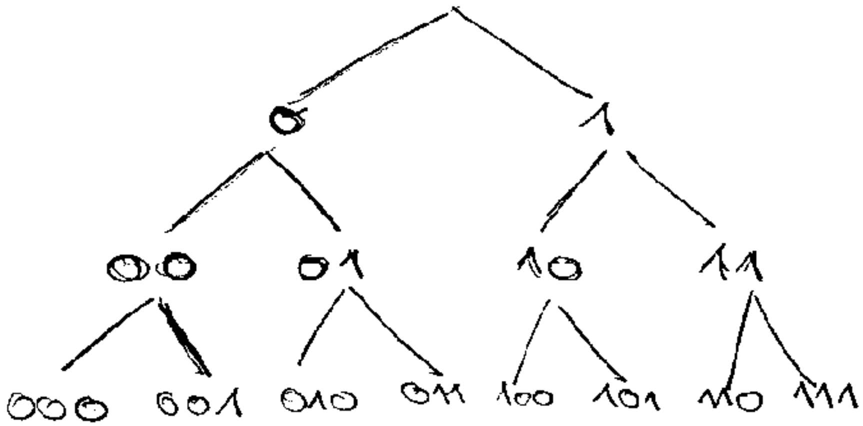
gefolgt von

$b_0 b_1 - b_{d-1} 1$ „Alle von $n-(d+1)$".

Programm in Bindru.java.




Das Muster ist bei x.length = 3:




Vollständiger binärer Baum der Tiefe 3:

o Tiefe 0, Tiefe 1

Tiefe 2, ...

8.53

Schwieriger 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

$$1 \; \text{Permutation auf} \; \underbrace{2 \dots n}_{n-1 \text{ Elemente}}$$

2. Alle der Art

2 Permutationen auf 1, 3 .., n

m. Alle der Art

m 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)      // In der while-Schleife
    i++               // Perm(x, d+1) n-d-mal
}                     // Jeweils anderes Feld x.
```

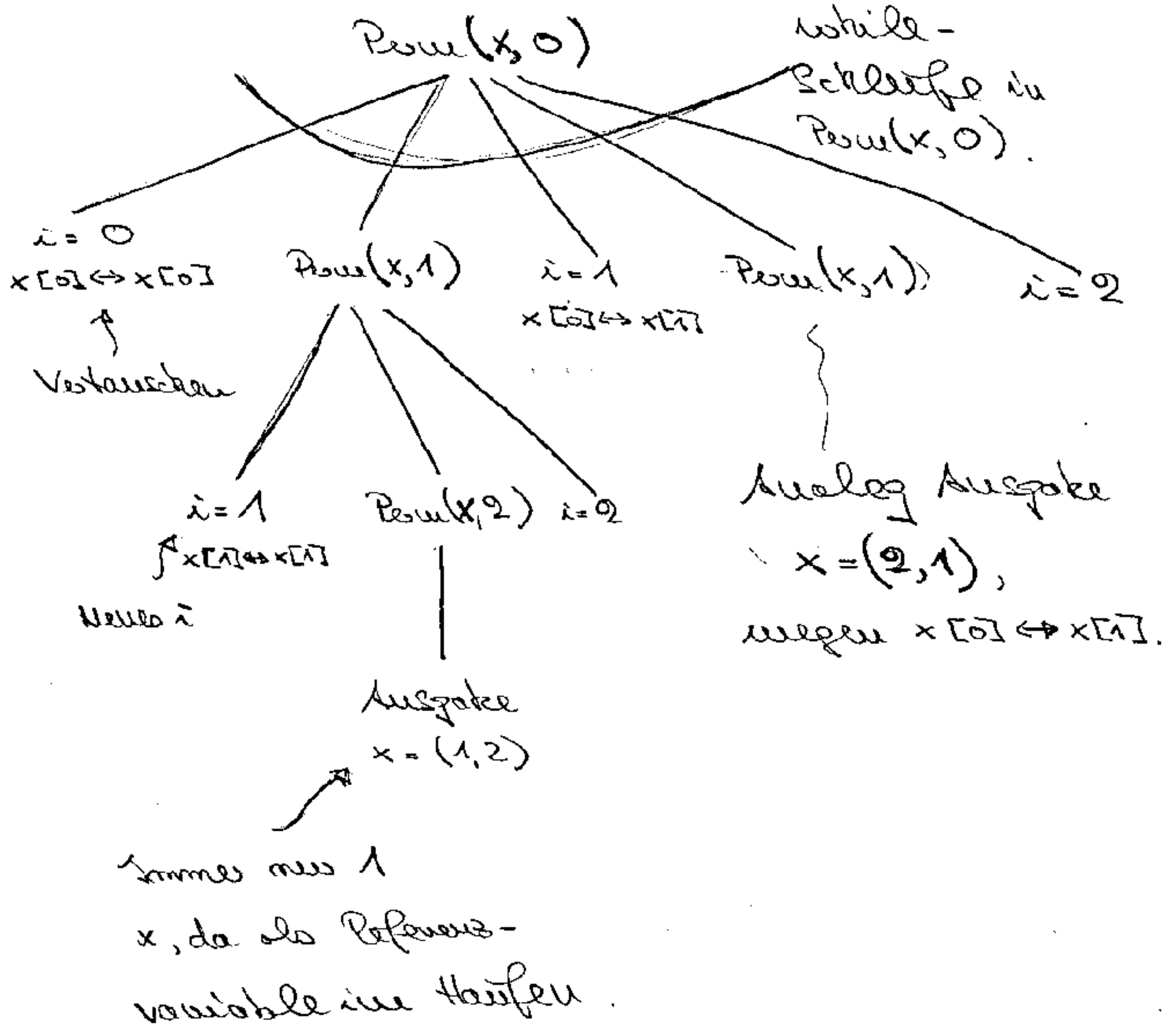
Startpermutation ist dann

$x[0] = 1,\ x[1] = 2,\ \dots,\ x[n-1] = n$

Startaufruf ist dann

```
Perm(x, 0)   // Führt zu n Aufrufen
             // von Perm(x, 1).
```

Aufrufbaum bei n = 2:



$n = 3$

Perm(x, 0)

$i = 0$
$x[0] \leftrightarrow x[0]$

Perm(x,1)

$i = 1$
$x[0], x[1]$

Perm(x,1)

$i = 2$
$x[0], x[2]$

Perm(x,1)

$i = 3$

Analog Perm(x,0) auf $n = 2$

Ausgabe von

1 2 3
1 3 2

Hier wird getauscht

Ausgabe von

3 1 2
3 2 1

Hier wird getauscht

Ausgabe von

1 2 3
1 3 2

Nicht ganz richtig.

Das Austauschen muß sich immer auf die Ausgangspermutation beziehen, den aktuellen Parameter bei Aufruf.

Der Modifikation in der while-Schleife:

```
while (i < x.length) {

    Vertausche x[d] mit x[i]

    Perm(x, d+1)

    Vertausche x[d] mit x[i]

    i++

}
```

Neu.

Dann können wir induktiv über x.length − d argumentieren, daß am Ende eines Aufrufs Perm(x,d) wieder das Ausgangs-x vorliegt.

$x.\text{length} - d = 0$, dann $d = x.\text{length}$

dann gilt die Behauptung, da nichts an $x$ gemacht wird.

$x.\text{length} - d \gneq 0$, dann $d \lneq x.\text{length}$.

Dann folgt die Behauptung mit der I.d.-Vor., da die while Schleife für $i = d$ bis $i = x.\text{length} - 1$ ausführt:

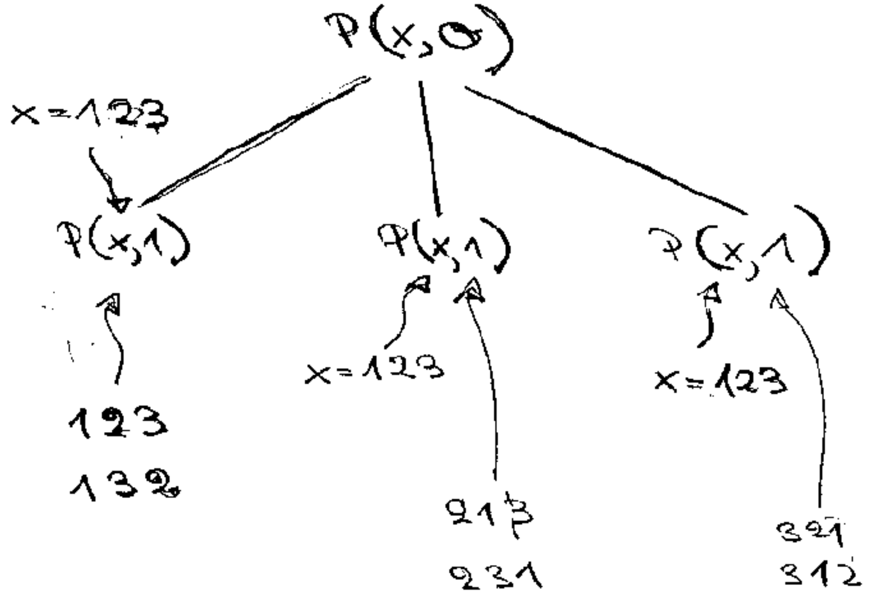
$x[d] \leftrightarrow x[i]$

$\text{Perm}(x, d+1)$

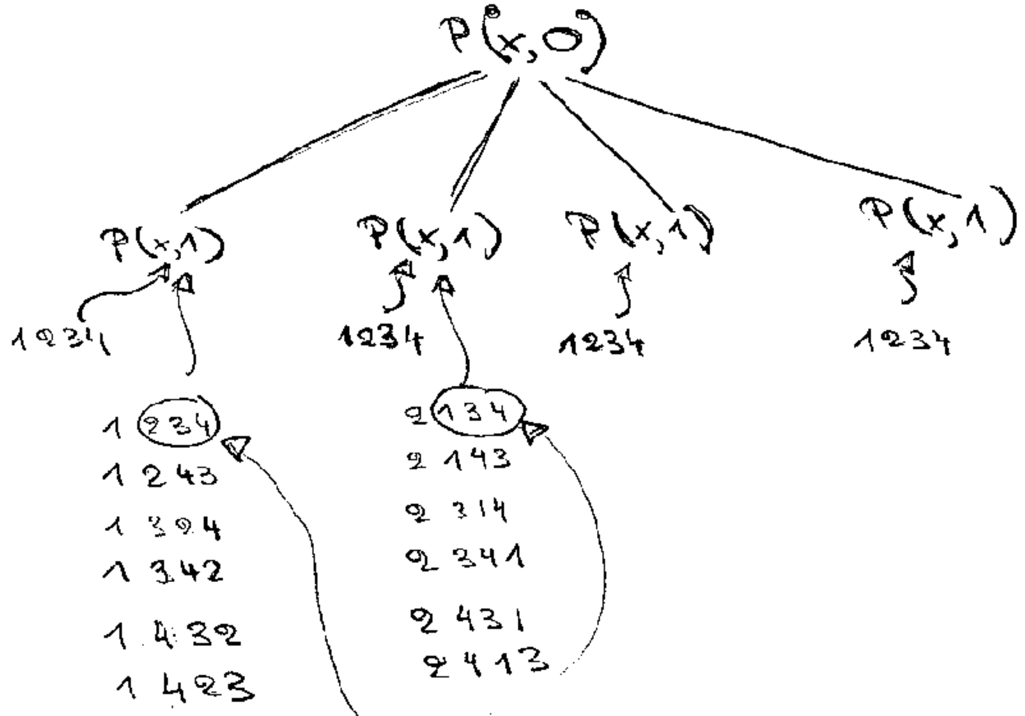
$x[d] \leftrightarrow x[i]$

und nach I.d.-Vor. jedesmal wieder die Ausgangspermutation $x$ vorliegt. $\boxtimes$

$n = 3$




$n = 4$




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.

$\vdots$

brett[7] = 0 $\Leftrightarrow$ Spalte 8 Zeile 1.

Lösungskandidat: Permutation

von 0, 1, ..., 7.

Zeile 1 Zeile 2 Zeile 8

Lösungsraum: Alle Permutationen

auf 1 ... 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

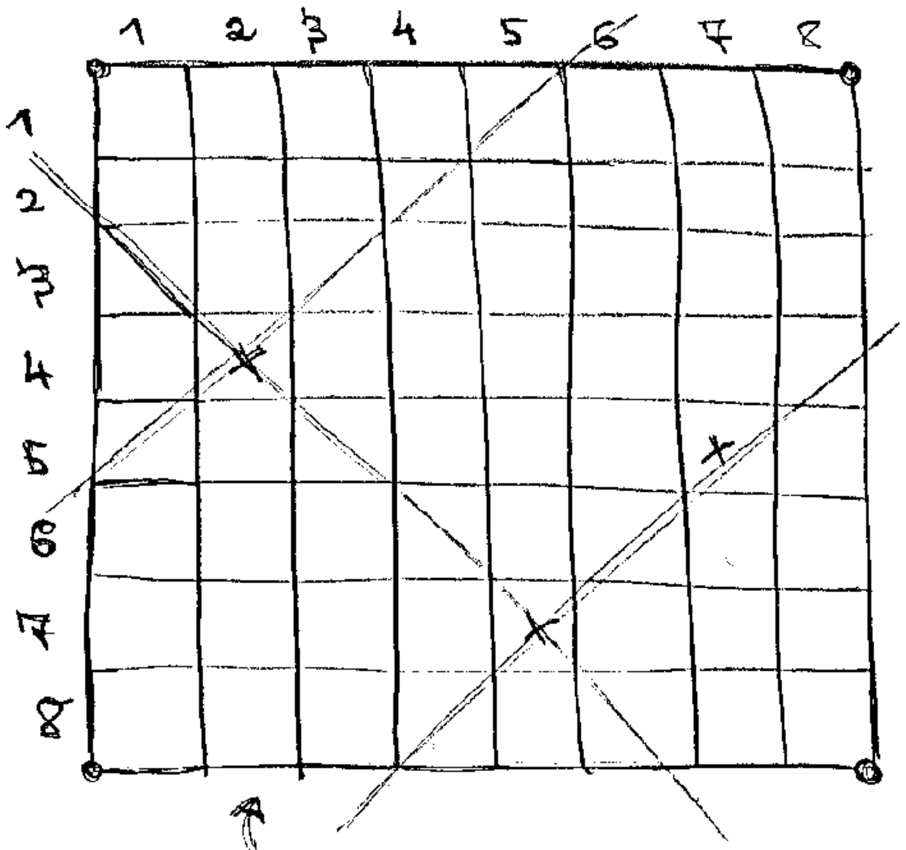
$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{((\log n)-1)\cdot\frac{n}{2}} > 2^n .$$

Wie testen wir, ob eine Permutation Lösung ist?

Gemeinsame Zeilen nicht möglich, da Brett Permutation ist.

Gemeinsame Spalten erst recht nicht.

Diagonalen ↗:




brett[1] = 3

Diagonale ↘:

brett[l] = k, dann

| | | | |
|---|---|---|---|
| l-1 | k-1 | l+1 | k+1 |
| l-2 | k-2 | l+2 | k+2 |
| l-3 | k-3 | l+3 | k+3 |

Dieser Teil reicht schon.

Bezeichne hier spalte = l.

```
for(int i = spalte - 1, j = brett[spalte] - 1;
                        i >= 0 ; i-- ; j-- )

        if (brett[i] == j) return true
```

Auch Feldgrenzen.

Ebenso Diagonale / nach links unten:

```
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, und $l \geq 1$:

brett[sp-1] $\neq$ brett[sp] - 1, und

brett[sp-2] $\neq$ brett[sp] - 2, ... und

brett[sp-$l$] $\neq$ brett[sp] - $l$.

Falls return true, dann ... aber

brett[sp-$l$] = brett[sp] - $l$.

Quintessenz mit $l$ = sp, das entspricht $i = 0$.

Methode, die für gegebenes brett prüft, ob kein Schlagen möglich ist.

```java
public static boolean bedroht(int[] brett)
 int spalte = 0;
```

Testet auch die Feldgrenzen.

```java
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.
```

Hier der Test nach

links unten /

Invariante der äußeren Schleife:

Nach $l$-tem Lauf ohne return true sind die ersten $l$ Spalten von links jedenfalls gut.

Quintessenz: Am Ende sind alle Spalten nach links gut, also insgesamt gut.

Programm ADE.java

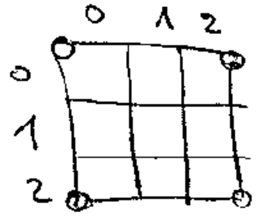
Permutationen auf $0,\ldots,7$ durchgehen.
Jede Permutation testen.

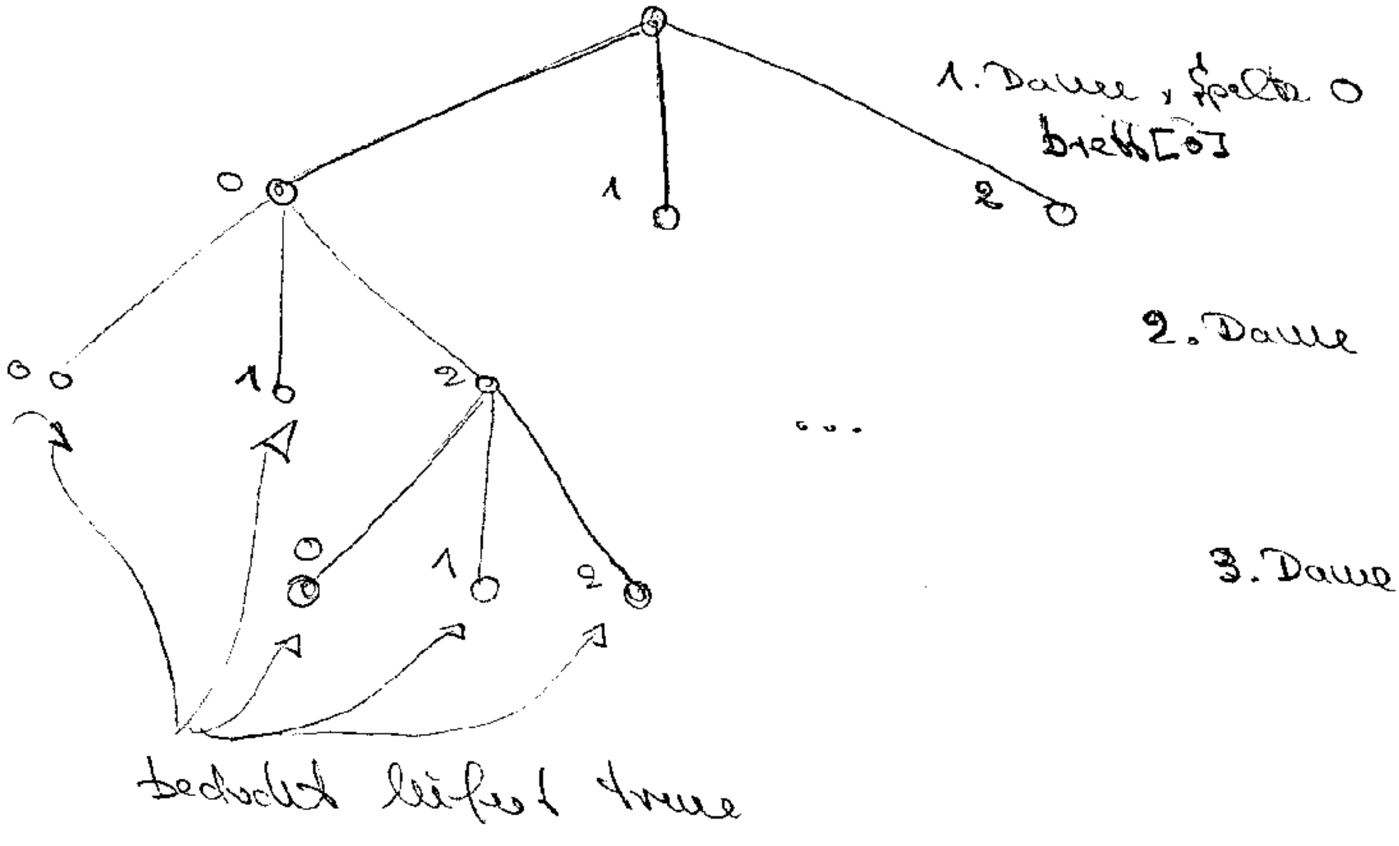
$n = 8$, Einträge auf $0,\ldots,7$ ergibt 92 Lösungen. ✓

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 Permutation, die sich aus dieser Sackgasse ergeben in einem Schritt als Nicht-Lösungen erkannt.

Im Prinzip kommen mit einem Postordnungsdurchlauf durch den folgenden Baum:

$n = 3$




Zum Beispiel wird 0 1 2 gar nicht betrachtet. Ebenso 1 2 0 nicht ...

Bei beliebigem n bekommen
wir eine wesentlich größere
Ersparnis im Vergleich zu ADE-java

O1 — O1 „Alle (n-2)! Permutationen"

O21 — O21 „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 zu
                       // 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 Baumes

spalte = brett.length, dann gilt die Behauptung.

spalte < brett.length. Wir 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 !

S. 74

Unsere Programme geben immer alle Lösungen des im allgemeinen n-Damen Problems auf. Wie ist das Programm Achtdamen.java zu modifizieren, so daß nur eine Lösung ermittelt wird (Effizienz).
Das ist das Programm im Buch auf S.185.

Der Unterschied ist, daß nun
public static boolean setze(int[] brett, int spalte)
Hier
definiert wird.

Ist eine Lösung gefunden wird true zurückgegeben (Zeile 32).
In dem darüberliegenden Aufruf

wird dann direkt wieder getestet,

ob der [illegible] ausgegeben hat

und es wird wieder gleich [illegible]

zurückgegeben und ...

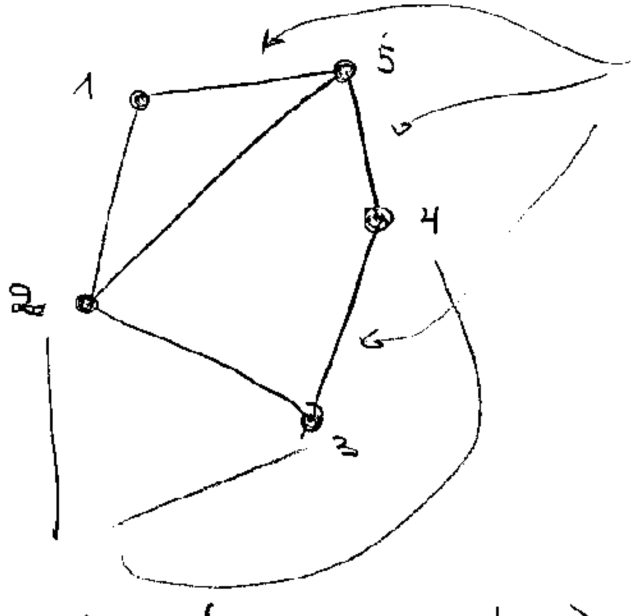
Als letztes Suchproblem: Das

- Färbungsproblem

bei

- Graphen.

Typische Graphen sind




Kante (edge)

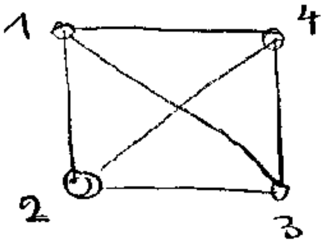
Knoten (node, vertex)

Formale Beschreibung

1 o——o 5 $\{1,5\}$ Zweiermenge von Knoten

1 o——o 2 $\{1,2\}$

⋮




Vollständige Graph (Clique) mit 4 Knoten.

Menge der Kanten

$= \{\{1,2\},\{2,3\},\{3,4\},\{4,1\},$
$\{1,3\},\{2,4\}\}$

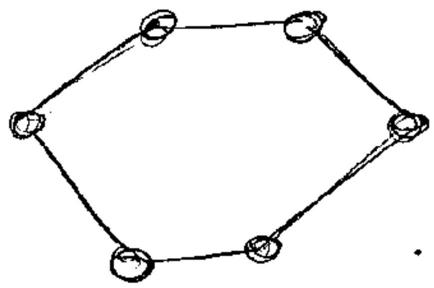
Also 6 Kanten.

$$\binom{4}{2} = \frac{4\cdot 3\cdot 2\cdot 1}{2\cdot 2} = 6.$$

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

$$f : V \to \{1, \dots k\}$$

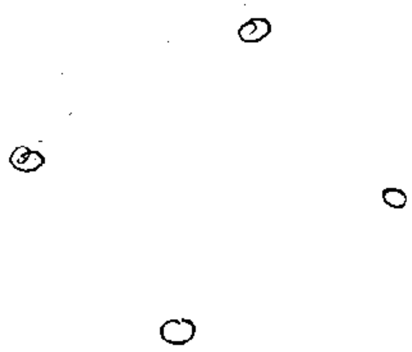
Die Färbung des Knoten

Die Farben (alternativ auch $\{0, \dots, k-1\}$)

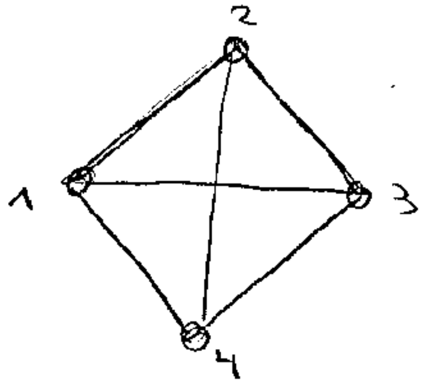
so daß gilt: ist

$\{v, w\} \in E$, so ist $f(v) \neq f(w)$. □

„Benachbarte" Knoten haben nie die gleiche Farbe.



$E = \emptyset$ ist 1-färbbar, alles die gleiche Farbe.

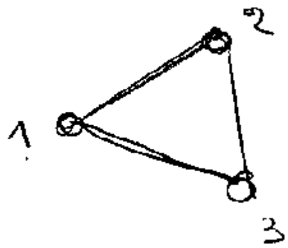



Ist 4-färbbar

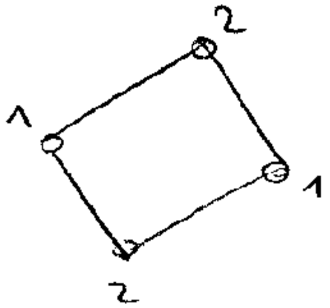
$$F(V) = 4.$$

Ist $|V| = n$, dann immer $n$-färbbar.

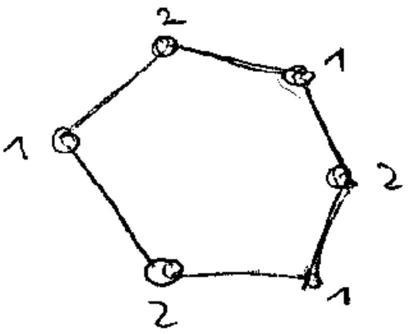
8.21



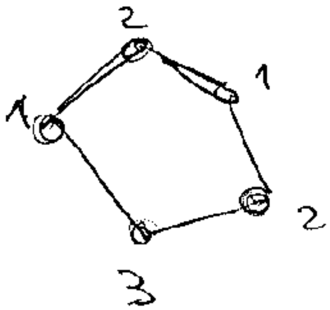

Nicht 2-, aber 3-färbbar.




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 $f(n-2) = 1$

$F(3) = 1$ ✓

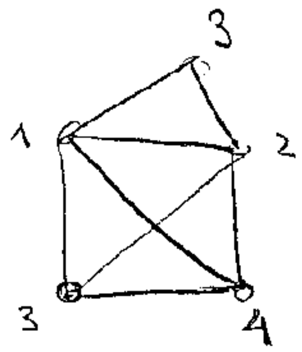
$\vdots$

$$F(n-1) = \begin{cases} 2 & \text{falls } n-2 \text{ ungerade} \\ 3 & \text{falls } n-2 \text{ gerade.} \end{cases}$$

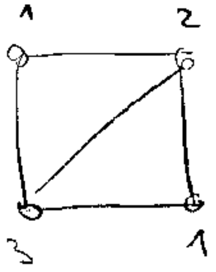
Dann $f(n-2) = 2$

1.

□




4 - färbbar.




3 - färbbar.

Wie stellt man die 3-Färbbarkeit eines Graphen fest?

Graph G mit

- $V = \{1, \dots, n\}$
- E beliebig.

Wie sieht nun der Lösungsraum aus? Alle prinzipiell möglichen 3-Färbungen von V sind alle Abbildungen

$$F : V \to \{1,2,3\}.$$

Wieviele Abbildungen F gibt es?

| $v$ | 1 | 2 | 3 | 4 | ... | $n$ |
|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | | 1 |
| | 2 | 2 | 2 | | ... | 2 |
| | 3 | 3 | 3 | | | 3 |

Jeder „Weg" = 1 F

$$3 \cdot 3 \cdot \ldots \cdot 3 = 3^n \text{ Wege}$$

$3^n$ 3-Färbungen.

Darstellung von $G$ durch

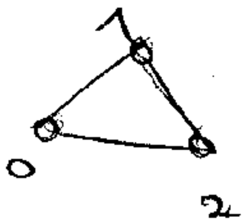
graph[n][n] der Adjazenzmatrix.

graph[v][w] = 1 $\Leftrightarrow$ $\{v,w\}$ Kante

graph[v][w] = 0 $\Leftrightarrow$ $\{v,w\}$ nicht Kante.

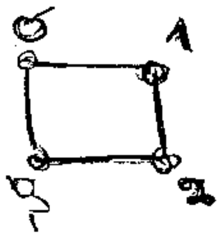
Natürlich graph[v][w] = graph[w][v]

und graph[v][v] = 0.




$$
\text{graph}\quad
\begin{array}{c|ccc}
 & 0 & 1 & 2 \\
\hline
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
2 & 1 & 1 & 0
\end{array}
$$

Symmetrisch an der Diagonalen.




$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 1 & 0 & 1 & 0 \end{array}$$

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 der Knoten
                   // in der Reihenfolge
                   // 0, 1, 2, ..., n-1.
                   // F[] = partielle Färbung
                   // knoten = zu färbenden Knoten.

if (knoten == F.length) return true // gut gefärbt.

naeFar: for (int i = 1; i <= 3; i++)
            Teste ob i zulässige Farbe
            für knoten ist.
            Falls nein: Nächste Farbe,
                        continue naeFar.
```

8.88

Nächsten Knoten färben.

Falls ja:

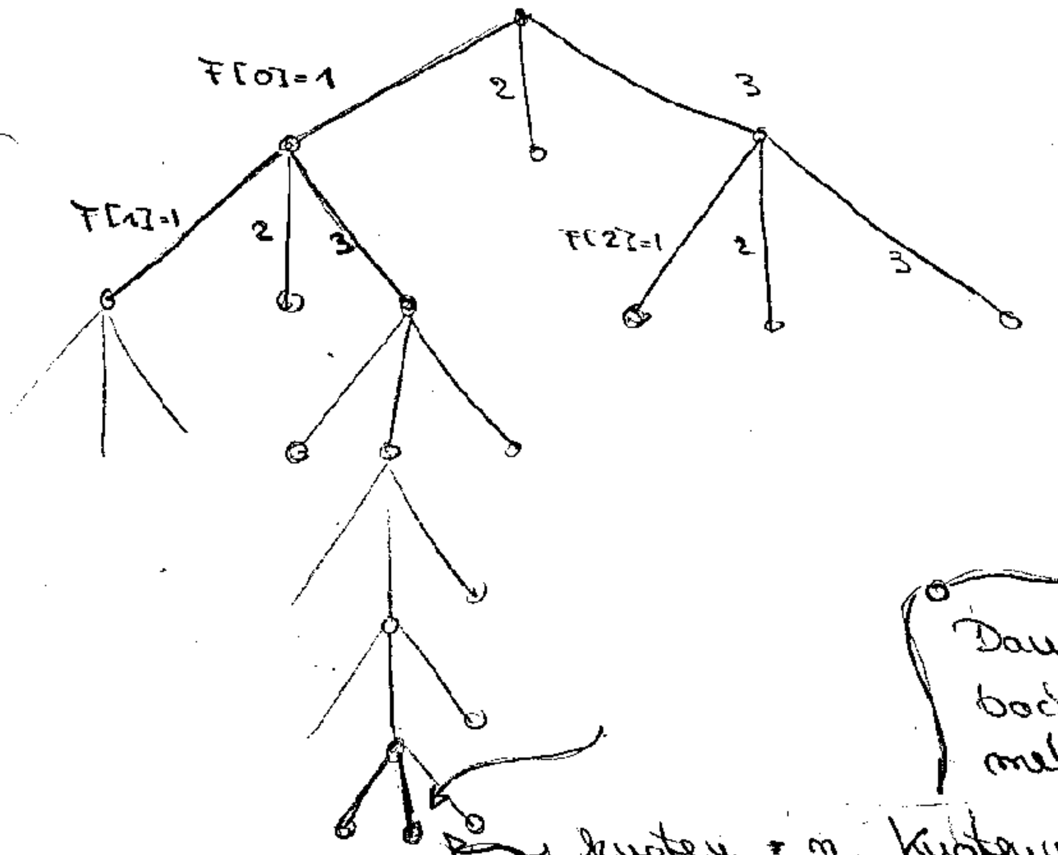
F[knoten] = i

test = Farb(3, F, knoten+1)

if (test) return true;

return false // Heißt F ist nicht zu zulässiger Färbung erweiterbar.

Prozedurbaum:




Dann kein backtracking mehr.

knoten = n, Knotenmenge 0,...,n-1.

Programm DreiFarb.java,

Einlesen per Hand.

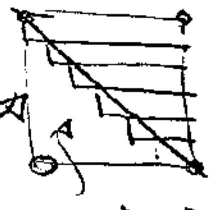
Jetzt zufällige Graphen. Was heißt das?

Einlesen.

- Knotenmenge $V = \{0, \ldots, \text{grösse} - 1\}$
- 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].

W-keit Math.random $\leq 1/2 = \frac{1}{2}$.

100 Züge dann $\approx$ 50-mal $\leq \frac{1}{2}$.

Math.random() * n $\leq c$, dann

$c = \frac{n}{2}$ bedeutet Math.random() $\leq 1/2$

$c = 3$ bedeutet Math.random() $\leq \frac{3}{n}$,

also eher klein, ....

$c = n$ bedeutet W-keit m.

Einige Testläufe:

grösse $= n = 30$

$c = 4$

W-keit Kante $\approx \frac{4}{30}$

$\# \text{Kanten} \approx \binom{30}{2} \cdot \frac{4}{30}$

$$= \frac{\cancel{30} \cdot 29}{\cancel{2}} \cdot \frac{\cancel{4}^{2}}{\cancel{30}} = 2 \cdot 29 \approx 54.$$

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 kürzester Weg, geringste Kosten, kürzeste Zeit, ...)

Dazu einige Beobachtungen:

- Bei $n$ Knoten reichen in jedem Fall $n$ Farben.

- Es reicht alle Färbungen der A.f

  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.

Ist $F(0) = j \gneq 0$,

vertausche Farbe 0 und $j$.

Ist $F(1) = j \gneq 1$,

vertausche 1 und $j$.

Ist $F(2) = j \gneq 2$

vertausche 2 und $j$.

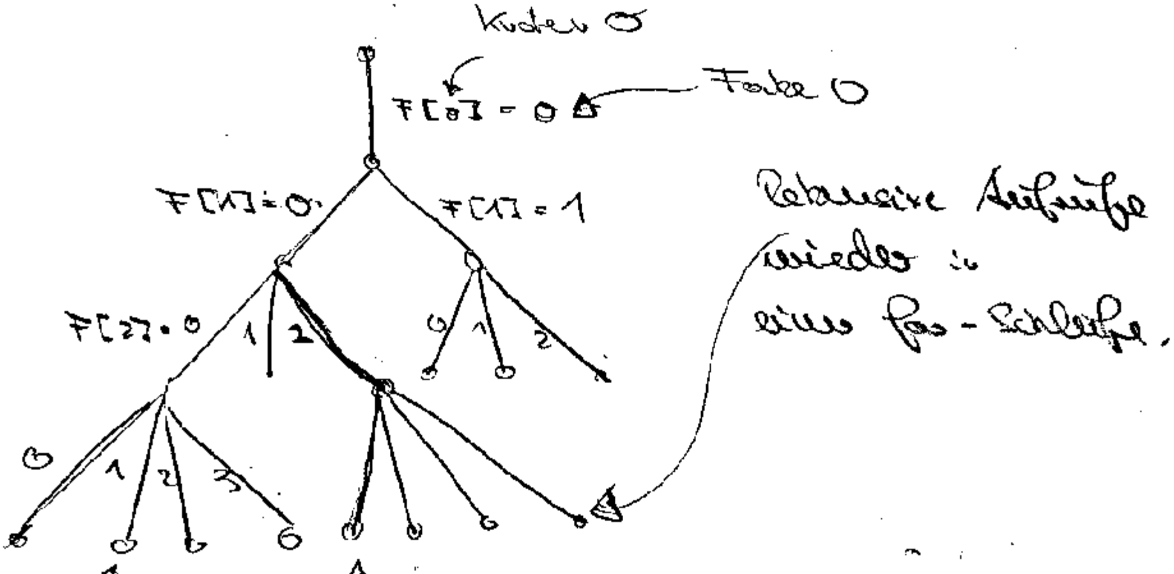
Beachte: Ist $F(2) = 0$ von vorher, so geschieht hier nichts

Ist ein Graph G mit Knotenmenge

0, ..., n-1 gegeben, so können

wir prinzipiell folgenden Baum

in Postordnung rekursiv abarbeiten:

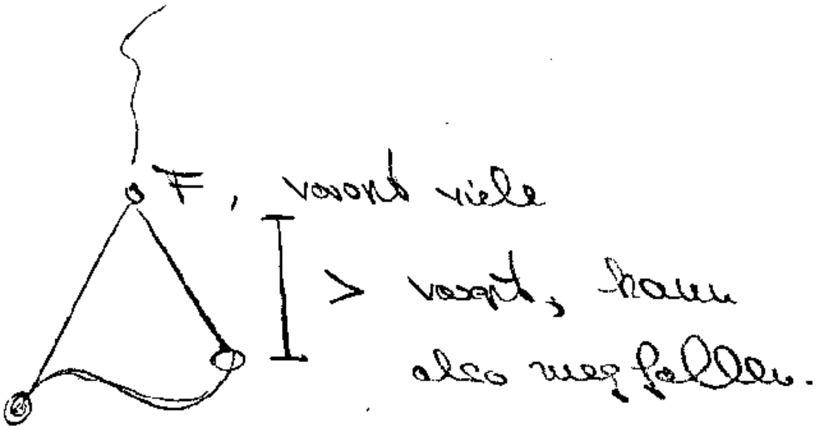



Nur 1 Farbe gebraucht.

3 Farben.

n Farben, diese Färbung tut es immer.

Beobachtung: Haben wir eine fertige Färbung mit sagen wir voropt (vorläufiges Optimum) vielen Farben so braucht jede partielle Färbung F mit voropt vielen Farben nicht mehr weiter betrachtet werden.




Prinzip: branch and bound.

public static void Faerbb

(int[][] graph, int[] F,

int[] Fvorgb, knoten)

anz = # verschiedene Farben von F

// interessanter Trick

// der Ermittlung. Nur

// 2-mal durch das array!

Falls knoten = F.length.

Haben fertige Färbung.

Testen ob anz < vorgb.

Falls ja, vorgb = anz;

„Fvorgb = F".

In jedem Falle return.

Falls knoten < F.length

Falls anz > vorgf return,
es hat keinen Sinn mehr.

Falls anz ≤ vorgb, Tu

Farben 0, ..., knoten bis
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 rekursiver Methoden unterscheiden:

- Aufrufbaum aus Parametern direkt ermittelbar (Fakultätsfunktion, Hanoi, 0-1-Vektoren).

- Aufrufbaum erst durch Ausführen der Rekursion ermittelbar (alle backtracking, branch-and-bound Verfahren). Bäume stark unterschiedlich in Abhängigkeit von der konkreten Eingabe.

Manchmal kann man ohne Rekursion auskommen (Fakultät, Exponentation, 0-1-Vektoren). Insbesondere im zweiten der genannten Fall ist eine Simulation ohne Rekursion schwierig, sogar unmöglich.

Immer kann die Rekursion eliminiert werden, indem man den Prozedurkeller per Hand simuliert. Demonstration an Hanoi.java. Für $n=2$

zeigen nun das Prinzip,
indem wir den Keller zeigen:

(Pfahl1, Pfahl2, Pfahl3, 2)

(Pf3, Pf2, Pf1, 1)
("Schiebe 2 von Pf1 nach Pf2")
(Pf1, Pf3, Pf2, 1)

Spitze

(Pf3, Pf2, Pf1, 1)
("Sch 2 von Pf1 nach Pf2")
("Sch 1 von Pf1 nach Pf3")

Wegen des 1 oben

Programmtext im Keller = das was noch zu machen ist. Entspricht Rücksprungadresse.

8.101

(Pf 3, Pf 2, Pf 1, 1)

Ausgabe von
Sch 1 von 1 nach 3
Sch 2 von 1 nach 2.

("Sch 1 von 3 nach 2")

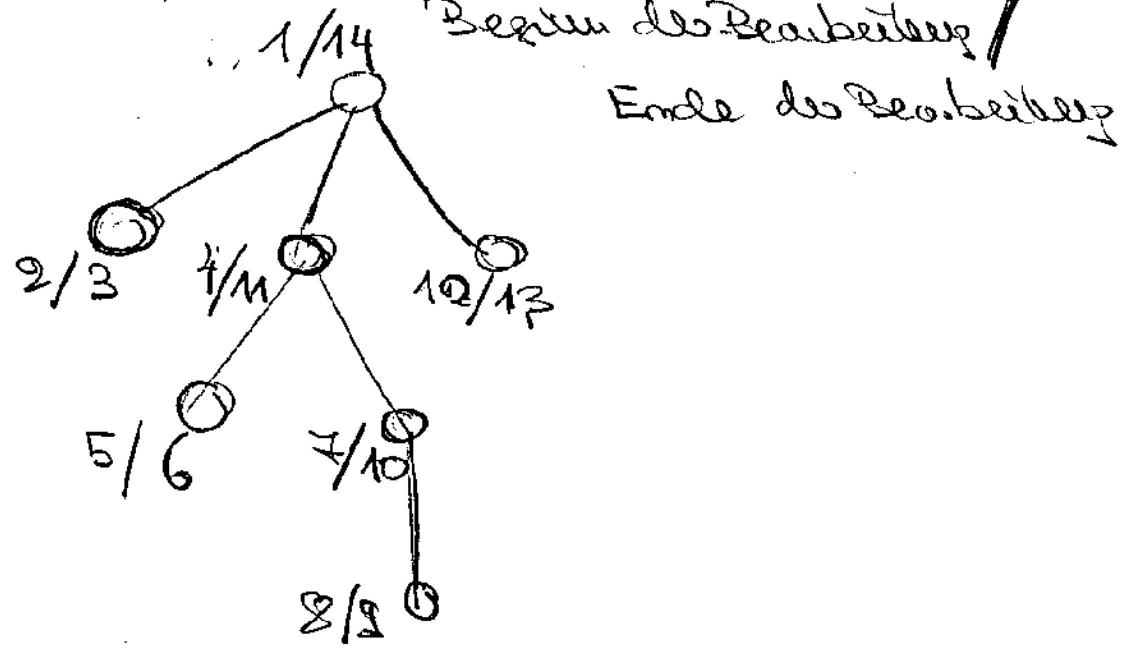
Ausgabe von
"Sch 1 von 3 nach 2".

Keller leer, Programm
zu Ende.

Ausprogrammierung in HanoiRek.java.

Noch 2 Nachträge.

Zur Abarbeitungsreihenfolge eines Prozedurbaums:




Ordnet man die Knoten nach dem Beginn, so ergibt das einen Präordnungsdurchlauf des Baums:

Erst die Wurzel, dann die Kinder von links nach rechts, rekursiv weiter (prae = vor).

Nach dem Ende bekommen wir die Postordnung: Erst die Kinder, dann die Wurzel und alles rekursiv weiter.

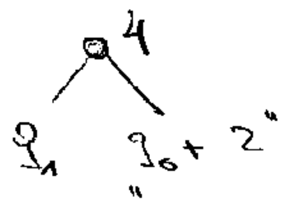
Zum 3-Färbbarkeitsproblem:

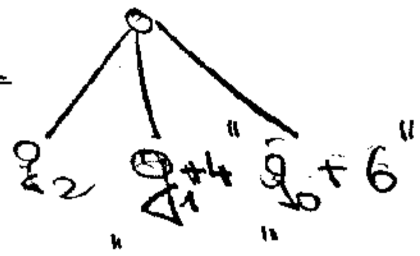
Wir betrachten den Algorithmus: Gehe die Knoten in der Reihenfolge 1, 2, 3, ..., n durch und färbe jeweils mit der kleinsten möglichen Farbe.

Dieser Algorithmus führt schon bei 2-färbbaren Graphen zu beliebig vielen Farben. Dann

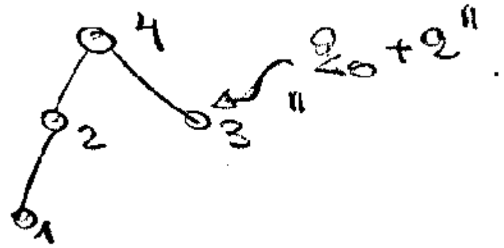
$g_0 =$ 

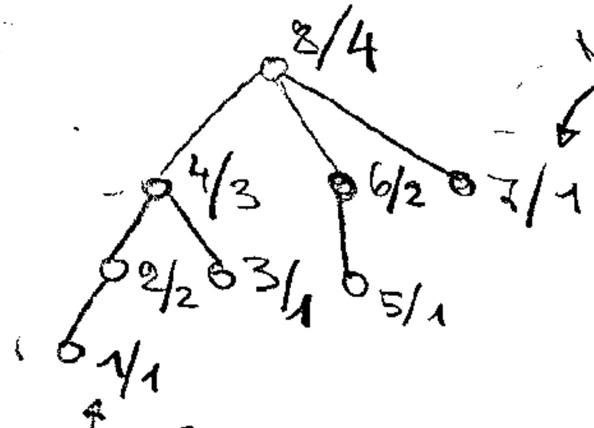
, $g_1 =$ 


$g_2 =$ 


$g_3 =$ 


Also

$g_2 =$ 


$g_3 =$ 


Name / Farbe des Algorithmus

Name / Farbe

8.10.5

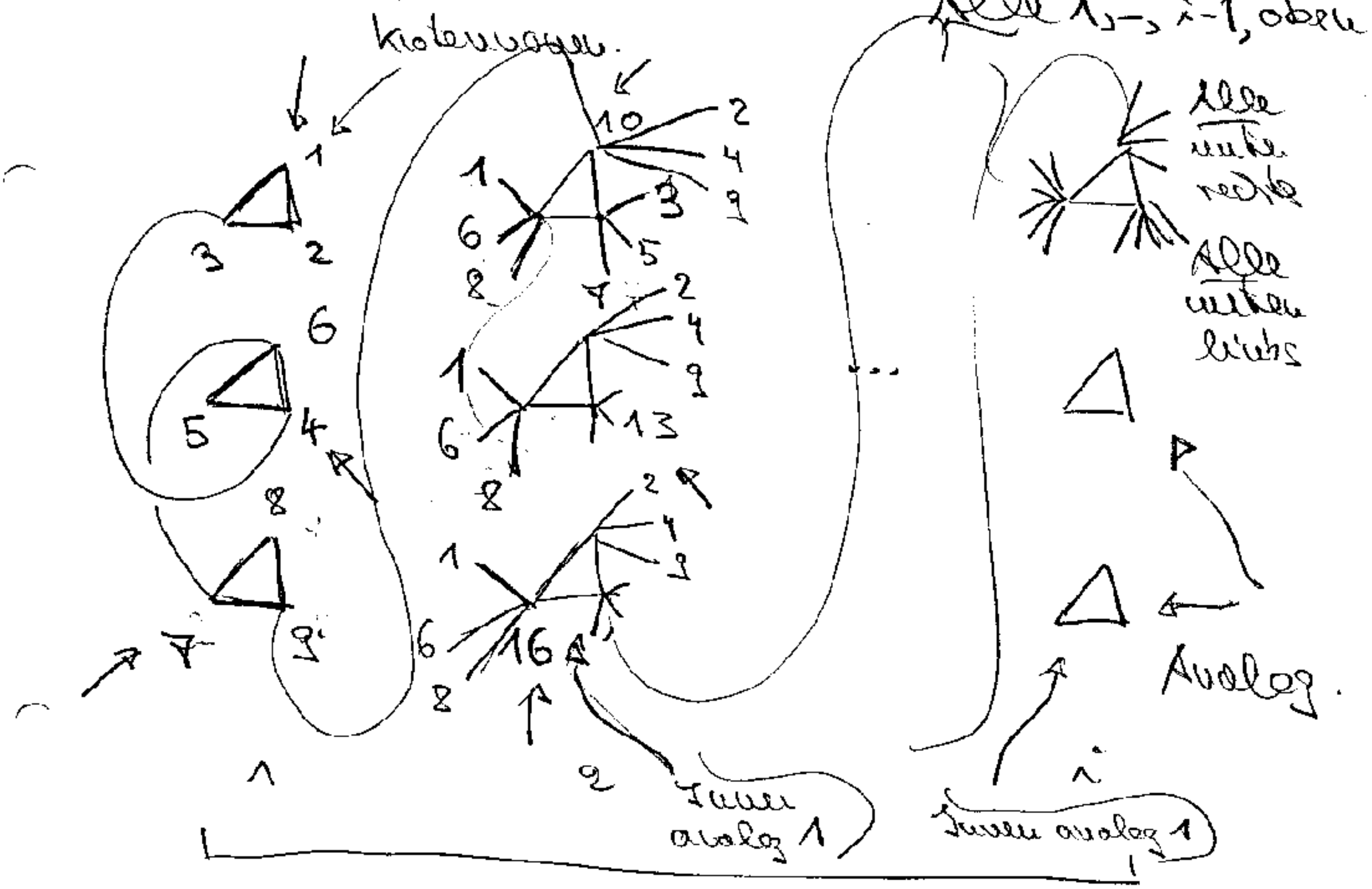
Dann braucht unser Algorithmus bei $G_i$ $i+1$ Farben. Dabei reichen 2!

Ein etwas besserer Algorithmus: Gehe zum noch nicht gefärbten Nachbarn kleinsten Namens und färbe ihn mit der kleinsten Farbe.

Das gibt im obigen Beispiel eine 2-Färbung. Allerdings können wir auch 3-färbbare Graphen finden, für die der Algorithmus beliebig viele Farben braucht.

Dazu folgende Menge (Familie)

von Graphen

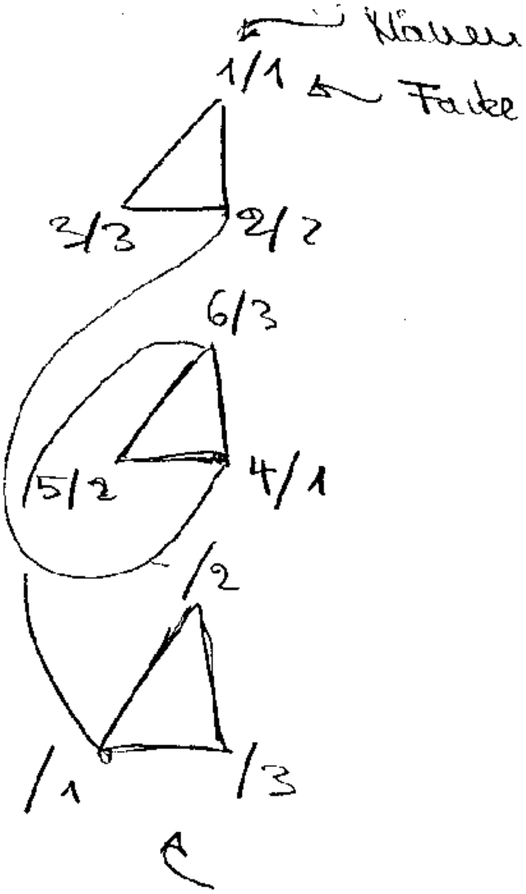



$G_i$ =

3-Färbung: Jedes Dreieck

Wie färbt der Algorithmus G

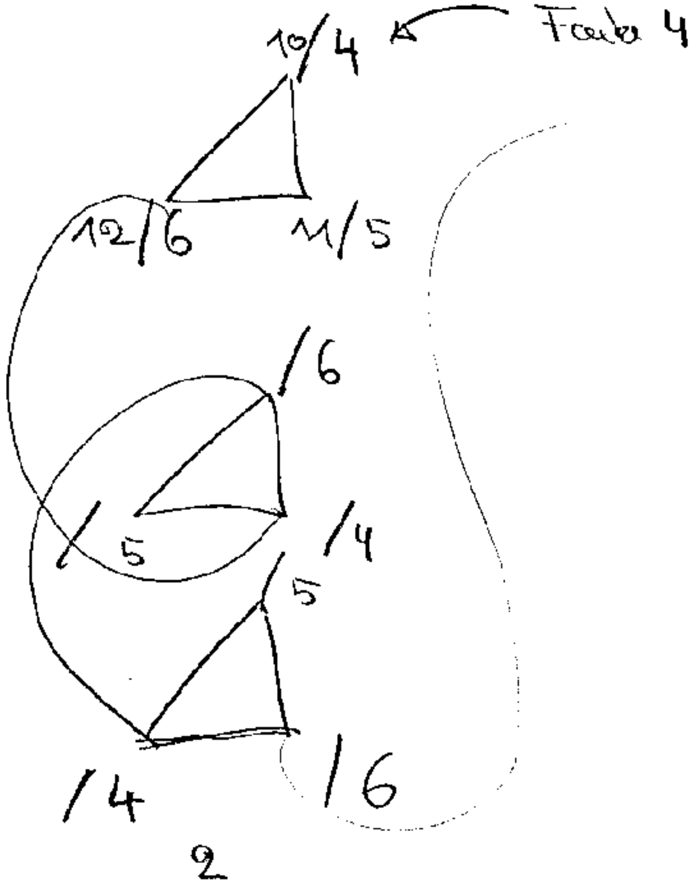
Die Dreiecke über 1 so:




Beachte die Verdrehung der Färbungen.

Dann Knoten 10.

Die Dreiecke über 2 so:




Die über 3 müssen nun mit Farbe 7 anfangen. Am Ende 9 Farben usf. $G_i$ hat also $3 \cdot i$ Farben mit dem einfachen Algorithmus.