

3. Entwurfsprinzipien von Algorithmen

- Konventionen für lesbaren Code
- Entwurfsprinzipien
 - Schrittweise Verfeinerung
 - Rekursive Algorithmen



3.1 Kommentare & Konventionen

- Standard Java Konventionen:
<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
- Namenskonventionen
- Formatierungsanweisungen
- Kommentarregeln
- Programmierstil
- **Siehe auch Konventionen auf ILIAS**

Namenskonventionen

- Parameter / Variablen:
 - Beginn mit Kleinbuchstaben; Anfangsbuchstaben weiterer Wortteile groß; in der Regel ein Nomen
 - `radius`
 - `lowerBound`
- Methoden:
 - Beginn mit Kleinbuchstaben; Anfangsbuchstaben weiterer Wortteile groß; meistens ein Verb (Imperativ)
 - `ggt`
 - `kreisFlaeche`
 - `oeffne`
- Konstanten:
 - ausschließlich Großbuchstaben; Trennung von Wortteilen durch Unterstrich (`_`)
 - `PI`
 - `MAX_VALUE`

Formatierung

- Hilfsmittel
 - Zeilenumbruch
 - Einrückung
- Eine Deklaration pro Zeile, am Blockanfang

- Beispiele
 - Format von Methodenköpfen

- sample(int i, int j) {
 ...
}

- Format von Anweisungen

- if (*condition*) {
 ...
} else {
 ...
}

Bei if, while, etc. am besten immer Anweisungsblöcke statt atomarer Anweisungen

Blöcke konsistent einrücken

Kommentare

- Wichtige Zusatzinformationen zur Dokumentation von Algorithmen
 - Bedeutung der Eingabe (Vorbedingungen)
 - Bedeutung der Ausgabe (Nachbedingungen)
- Kommentare werden vom Java-Compiler ignoriert.
- Arten von Kommentaren in Java

Kommentartyp	Beginn	Ende
einzeilig	//	Zeilenende
mehrzeilig	/*	*/
JavaDoc	/**	*/

Dieser Kommentar wird von dem Werkzeug javadoc genutzt, um aus dem Inhalt des Kommentars z.B. html-Seiten zu erzeugen.

Das Werkzeug javadoc

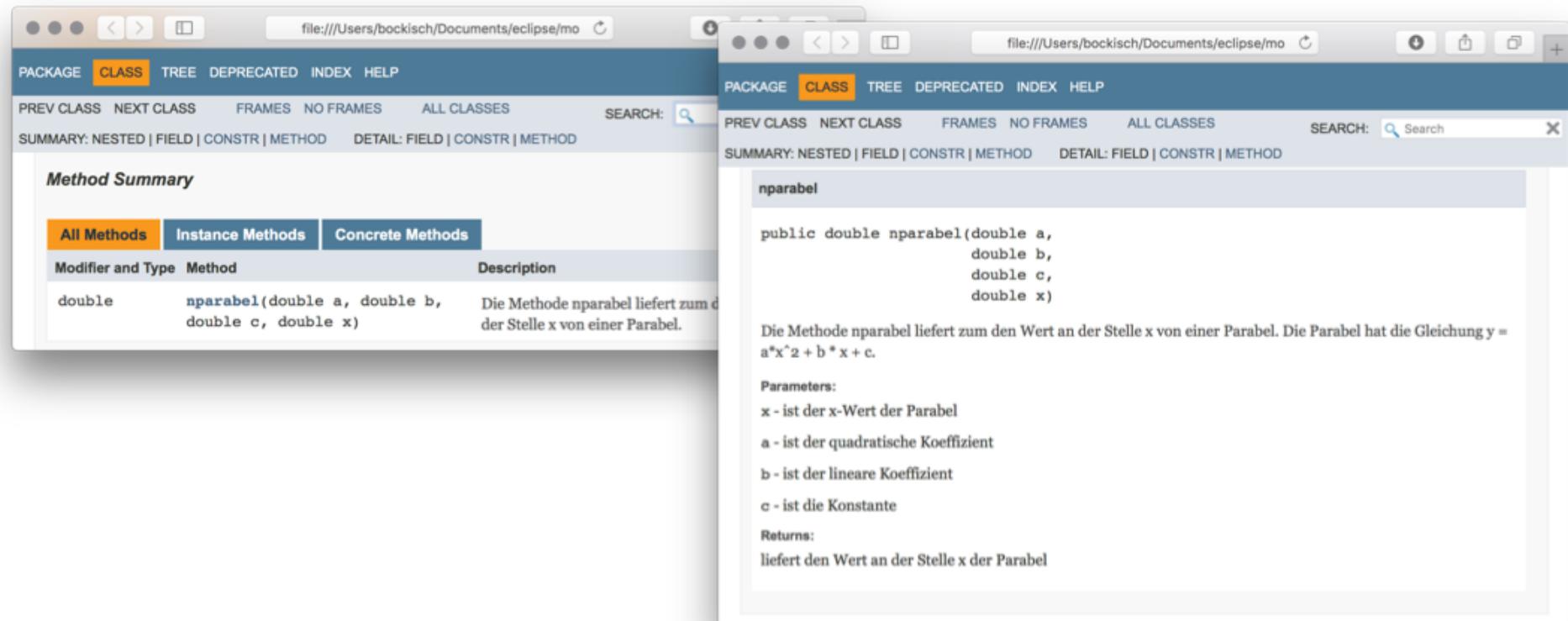
- javadoc erzeugt aus einer java-Datei unter Verwendung der Kommentare `/** ... */` eine **html-Datei**
 - und noch ganz viele andere Dateien.
- Spezielle **Tags mit dem Präfix @** im Kommentar haben eine Bedeutung.
 - Allgemein verwendbare Tags
 - `@author` für Namen des Autors
 - `@version` für die Version des Programms
 - `@see` für Verweise
 - Tags für Methoden
 - `@param` für die Methodenparameter
 - `@return` für das Ergebnis
 - `@see` für Verweise
- Erster Satz muss eine vollständige Kurzbeschreibung sein
- Weitere Sätze für zusätzliche Erläuterungen

Verwendung später,
wenn wir Klassen haben.

Beispiel: Parabel (1)

```
/** Die Methode nparabel liefert zum den y-Wert an der Stelle x von einer Parabel.  
 * Die Parabel hat die Gleichung y = a*x^2 + b * x + c.  
 * @param x ist der x-Wert der Parabel  
 * @param a ist der quadratische Koeffizient  
 * @param b ist der lineare Koeffizient  
 * @param c ist die Konstante  
 * @return liefert den Wert an der Stelle x der Parabel  
 */  
  
public double nparabel(double a, double b, double c, double x) {  
    return a*x*x + b*x + c;  
}
```

JavaDoc



- JavaDoc Tool ist Klassenbasiert
 - Daher Anwendung erst später
 - Aber: kommentieren Sie Methoden bereits nach diesen Regeln!

Lesbarkeit von Programmen

- Die anderen Arten von Kommentaren sind primär **für Programmentwickler** gedacht und sind deshalb auch wichtig.
 - Methoden werden nicht nur benutzt, sondern können bei großen Softwareprojekten durch einen anderen Entwickler verändert werden.
- Generell ist beim Verstehen von Programmcode auch die **Formatierung des Programmtexes** wichtig.
 - Die Formatierung hat keinen Einfluss auf die syntaktische Korrektheit eines Programms, sondern dient „nur“ dazu, die Lesbarkeit zu verbessern.
 - Zeilenumbrüche, Leerzeichen und Einrücken sollten verwendet werden, auch wenn dies nicht durch die Programmiersprache gefordert wird.
 - *Wir werden deshalb darauf achten, die Programmierrichtlinien (siehe ILIAS) einzuhalten.*

3.2 Schrittweise Verfeinerung von Algorithmen

- Entwurf der Algorithmen ist relativ einfach solange die Problemstellungen einfach sind.
 - Leider sind nahezu alle interessanten Problemstellungen komplex!

Schrittweise Verfeinerung (Top-Down Ansatz)

- Idee
 1. Zerlege das Problem geeignet in mehrere Einzelprobleme
 2. Verfahre mit jedem in Schritt 1 erzeugten Teilproblem folgendermaßen:
 - Falls das Teilproblem genügend einfach ist: Entwirf Algorithmus für das Problem
 - Ansonsten: Wende das Verfahren der schrittweisen Verfeinerung auf dieses Problem erneut an.

Dieser Top-Down Ansatz erinnert mich an das Teile und Hersche Pardigma:
Ist eine Problem groß, dann zerlege das Problem in Teilprobleme
ist ein Teilproblem klein und einfach dann löse ist
ist das Teilproblem immer noch groß dann teile sie wieder in andere
Teilprobleme bis sie einfach genug sind um es zu lösen, sind die jetzt
klein genug dann löse es.

Beispiel

- Algorithmus Entkalken-Komplett
 1. Vorbereitung
 2. Entkalken
 3. Spülen
- Jeder dieser Schritte muss wieder durch einen eigenen Algorithmus genau beschrieben werden.



Algorithmus zum Spülen der Kaffeemaschine

1. Setze den vollen Wassertank in das Gerät ein.
2. Stelle das Auffanggefäß in die Mitte des Auffanggitters.
3. Drücke die grün leuchtende Taste.

Beispiel

- **Problem**

- Schritt 1 ist **noch nicht detailliert genug**, d.h. der Schritt kann nicht interpretiert und deshalb auch nicht ausgeführt werden.

- **Erneute Verfeinerung des Algorithmus**

Algorithmus zum Einsetzen des vollen Wassertanks

1. Fülle den Wassertank mit Trinkwasser bis zur max-Marke auf.
2. Klappe die Wassertankhalterung des Geräts auf.
3. Setze den vollen Tank auf die Halterung.
4. Bringe die Wassertankhalterung wieder in die senkrechte Position.

Aspekte bei der schrittweisen Verfeinerung

- Kenntnisse über die Fähigkeiten des Prozessors sind von zentraler Bedeutung bei der schrittweisen Verfeinerung.
 - Wenn ein Schritt noch nicht vom Prozessor ausgeführt werden kann, muss noch eine Verfeinerung vorgenommen werden.
 - Wünschenswert ist auch eine Verfeinerung von Schritten, deren Ausführung noch verbessert werden kann.
 - Die Fähigkeiten des Prozessors legen die Richtung der Verfeinerung fest.
- Bisher entwickelte Algorithmen, die der Prozessor ausführen kann, können wieder beim Entwurf anderer Algorithmen als elementare Schritte genutzt werden, z.B. „Einsetzen Wassertank“
 - Wichtig beim Spülen der Kaffeemaschine und beim Zubereiten von Kaffee
→ Prozessor erweckt dadurch den Anschein, komplexere Operationen direkt ausführen zu können.

Beispiel: Maximum – Version 1

- Aufgabe
 - Berechnung des Maximums von 3 Zahlen
- Lösungsidee
 - Nehmen wir an, dass wir bereits eine Methode max2 hätten, um das Maximum von zwei ganzen Zahlen zu berechnen.
 - Dann können wir das Maximum von drei Zahlen darauf zurückführen.

```
/**  
 * Der Algorithmus berechnet das Maximum von drei ganzen Zahlen  
 * @param m erste ganze Zahl  
 * @param n zweite ganze Zahl  
 * @param p dritte ganze Zahl  
 * @return Maximum von m, n und p  
 */  
int max3(int m, int n, int p) {  
    int tmp = max2(n,p); // Berechne zunächst das Maximum von n und p  
    return max2(m, tmp); // Danach das Maximum von m und tmp  
}
```

Beispiel: Maximum – Version 1

- Aufgabe
 - Berechnung des Maximums von 3 Zahlen
- Lösungsidee
 - Nehmen wir an, dass wir bereits eine Methode max2 hätten, um das Maximum von zwei ganzen Zahlen zu berechnen.
 - Dann können wir das Maximum von drei Zahlen darauf zurückführen.

```
/**  
 * Der Algorithmus berechnet das Maximum von drei ganzen Zahlen  
 * @param m erste ganze Zahl  
 * @param n zweite ganze Zahl  
 * @param p dritte ganze Zahl  
 * @return Maximum von m, n und p  
 */  
int max3(int m, int n, int p) {  
    return max2(m, max2(n,p));  
}
```

Die Variable tmp können wir einsparen, indem wir den Aufruf direkt dort hinschreiben, wo auf tmp lesend zugegriffen wird.

Hier haben wir das Problem vereinfacht in dem wir das Problem gebrochen haben in ein Problem das einfacher zu lösen ist

Vorteile der schrittweisen Verfeinerung

- Komplexe Problemstellungen werden in einfachere Probleme aufgeteilt. Jedes der Probleme kann durch eine eigene Methode gelöst werden.
 - Änderungen einer Methode haben keinen Einfluss auf das aufrufende Programm, solange das Verhalten der Methode gleich bleibt.
 - Beim Benutzen einer Methode ist nur erforderlich, **was** die Methode leistet, aber **nicht, wie** die Methode dies erledigt (**prozedurale Abstraktion**).
 - Korrektheit eines großen Programms kann einfacher überprüft werden, da dies auf die Korrektheit der Methoden abgebildet werden kann.
 - Methoden können unabhängig voneinander im **Team** entwickelt werden.
 - Fertig implementierte Methoden können beim Entwurf anderer Algorithmen **wiederverwendet** werden (→ **Softwarebibliotheken**)
 - In der Java-Klasse Math sind bereits viele mathematische Funktionen vorhanden. Wir können diese benutzen, indem wir vor dem Methodennamen noch den Klassennamen gefolgt von einem Punkt schreiben.

Math.log(5)

3.3 Rekursion

- Methodenaufruf
 - Die Methodenausführung hat einen eigenen lokalen Speicher für lokale Variablen

```
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max3(int m, int n, int p) {
    int tmp = max2(n,p); // Berechne zunächst das Maximum von n und p
    return max2(m, tmp); // Danach das Maximum von n und tmp
}
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max2(int m, int n) {
    if (m > n) {
        return m;
    }
    else {
        return n;
    }
}
int tmp = 21;
max3(1, 2, 3);
tmp;
```



3.3 Rekursion

- Methodenaufruf
 - Die Methodenausführung hat einen eigenen lokalen Speicher für lokale Variablen

```
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max3(int m, int n, int p) {
    int tmp = max2(n,p), // Bei der
    return max2(m, tmp); // Danach
}
/** JavaDoc Kommentar fehlt aus Platzgründen
 */
int max2(int m, int n) {
    if (m > n) {
        }
    else {
        }
}
int tmp = 21;
max3(1, 2, 3);
tmp;
```

Zuweisen von 3 an tmp im Kontext der Methodenausführung

Zuweisen von 21 an tmp im Standard-Kontext.

Ergibt 21, da wir im Standard-Kontext sind.



Speicherreservierung

Methodeninstanzen

Aufruf von `max3(1, 2, 3)`

→ Aufruf von `max2(2, 3)`

→ Aufruf von `max2(1, 3)`

Das sind Methodeninstanzen im Speicher für die Parmatervariablen, in diesem Fall haben wir keine lokalen Variablen

Wir haben 1x eine Methodeninstanz für die Methode `max3` und 2x Methodeninstanzen für die Methode `max2`

Variablenname	Wert
m	1
n	2
p	3
tmp	3

Variablenname	Wert
m	2
n	3

Variablenname	Wert
m	1
n	3

Speicherreservierung

Aufruf von `max3(1, 2, 3)`

→ Aufruf von `max2(2, 3)`

→ Aufruf von `max2(1, 3)`

Ein neuer Speicherbereich
wird für **jeden Aufruf
einer Methode** angelegt!

Lokale Variablen für max3	
Variabien	vwert
m	1
n	2
p	3
tmp	3

Lokale Variablen für max2	
Variabien	vwert
m	2
n	3

Lokale Variablen für max2	
Variabien	vwert
m	1
n	3

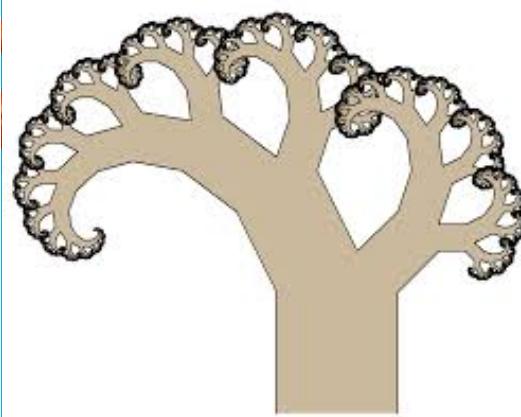
Wichtige Eigenschaft

was es passiert im Speicher wenn man ein rekursive Methode aufruft

- Bei jedem Aufruf einer Methode wird eine neue Methodeninstanz erzeugt. Diese bekommt ihren eigenen Speicherbereich, in dem insbesondere die Werte der Parametervariablen und der lokalen Variablen hinterlegt werden.
- Nach dem Beenden dieser Methodeninstanz wird dieser Speicher zurückgegeben.
- Dies gilt auch, wenn dieselbe Methode mehrmals aufgerufen wurde und somit mehrere Instanzen der Methode existieren können.

Rekursion

- Bisher
 - Lösung eines Problems P werden auf die Lösung eines anderen Problems Q zurückgeführt. [In dem Beispiel finde das Max von 3 zahlen wir haben es zurückgeführt an die Lösung von einem Problem mit finde das Maximum von 2 Zahlen](#)
- Jetzt
 - Lösung eines Problems P durch Zurückführen auf die Lösung eines Problems vom Typ P oder mehrerer einfacher Probleme vom Typ P.



Prof. Christoph Bockisch

The screenshot shows a Mozilla Firefox browser window with the title "Rekursion - Google-Suche - Mozilla Firefox". The address bar displays the URL <https://www.google.de/search?q=Rekursion>. The search term "Rekursion" is entered in the search bar. Below the search bar, there are tabs for "rekursion_vom_spiegel...", "Es war einmal ein Man...", and "Rekursion - Google-Su...". The main content area shows the Google search results for "Rekursion". The "Web" tab is selected, and the results indicate approximately 210,000 results found in 0.23 seconds. A red link "Meinten Sie: [Rekursion](#)" is highlighted. A cookie consent message at the bottom states: "Cookies helfen uns bei der Bereitstellung unserer Dienste. Durch die Nutzung dieser Seite erklären Sie sich damit einverstanden, dass wir Cookies setzen." There are "Cookie-Einstellungen" and "Akzeptieren" buttons at the bottom.



Beispiel (Matrjoschka-Puppen)

Ich habe nicht wirklich verstanden was er hier will

/* Öffnet rekursiv die Puppen, entfernt die massive Puppe und schließt die Puppen. */

```
void nimmMassivePuppe(Puppe M) {
```

```
    if (M ist massiv) { Hier glaube ich wir haben die Basisbedingung
        nimm M;
    }
    else {
        öffne M;
        nimmMassivePuppe(Inhalt von M);
        schließe M;
    }
}
```

- Zweiter Schritt im **else**-Zweig bewirkt, dass die gleiche Methode wieder aufgerufen wird, jetzt aber mit der nächst kleineren Puppe.
- Algorithmus arbeitet in 2 Phasen
 1. Phase: Öffnen aller Puppen
 2. Phase: Schließen aller Puppen



Berechnung der Fakultät

- Aufgabenstellung: Berechnung von $n!$, wobei
$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n = (1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1)) \cdot n = ((n-1)!) \cdot n$$
- Wir nutzen dabei aus, dass folgende Eigenschaft gilt.

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

- Direkte Umsetzung als rekursive Methode in Java

```
int fact(int n) {  
    if (n == 0)                                Basisfall  
        return 1;  
    else  
        return n * fact(n-1);  rekursiver Fall  
}
```

→ Das ist ein Beispiel für eine Lineare Rekursion

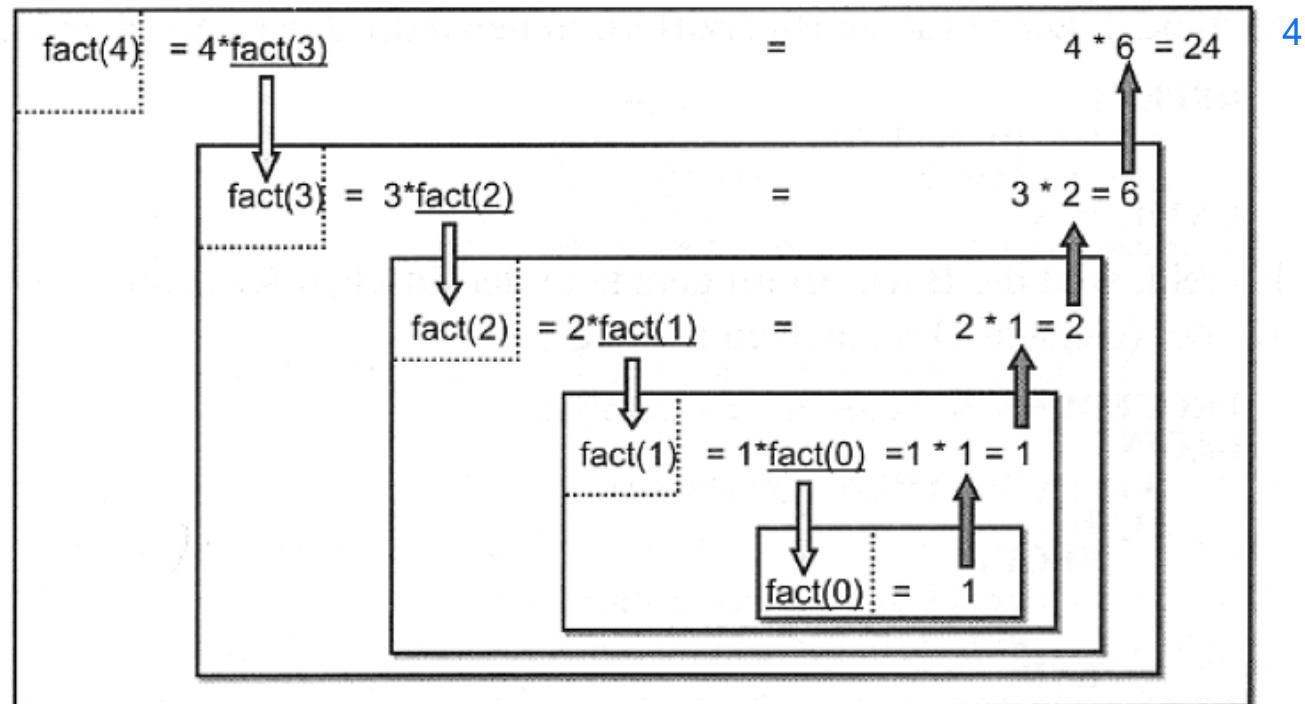
Veranschaulichung des Prozesses für fact(4)

Methodenaufrufe

Ergebnisrückgabe

Unterprogrammaufrufe

Ergebnisrückgabe



Wichtige Anforderungen an rekursive Algorithmen

- Folge von rekursiven Methodenaufrufen muss endlich sein
 - Eingabe für den nächsten rekursiven Aufruf muss „einfacher“ werden.
 - Eine bedingte Anweisung (bzw. eine entsprechende Kontrollstruktur) muss den einfachsten Fall behandeln, an dem die Rekursion nicht fortgesetzt wird. *sogenannter Basisfall*
- Beispiel
 - Matrjoschka
 - Bei jedem Rekursionsaufruf nähern wir uns der massiven Puppe.
 - Eine bedingte Anweisung führt zu dem Fall, wenn wir auf die massive Puppe treffen.
 - Fakultätsberechnung
 - Bei jedem Rekursionsaufruf nähern wir uns der Null.
 - Eine bedingte Anweisung behandelt den Fall der Berechnung von $0!$.

sogennter rekursiver Fall

Wir müssen bei einem rekursiven Algorithmus darauf achten:
dass bei jedem rekursiven Aufruf dass wir uns den Basisfall annähren
die Basisfall ist dafür gedacht dass der Algorithmus nach endlich vielen
Schritten aufhört und ein Ergebnis liefert , da der Algorithmus sich selbst
aufruft müssen wir daran denken, dass der Algorithmus irgendwann endetnach endlich
vielen Schritten

Lineare Rekursion ist eine spezielle Art der Rekursion, bei der eine Funktion sich selbst nur einmal pro Aufruf aufruft, und dies immer auf eine direkte Weise. Bei linearer Rekursion gibt es eine klare und einfache Struktur, bei der der rekursive Aufruf die nächste Stufe des Problems bearbeitet, ohne sich mehrfach oder in komplexen Wegen selbst aufzurufen.

Hier sind die Hauptmerkmale der linearen Rekursion:

Einzelner rekursiver Aufruf:

Die Funktion enthält genau einen rekursiven Aufruf pro Instanz, der zu einem Basisfall führt. Die Struktur der Rekursion ist daher einfach und geradlinig.

Basisfall:

Wie bei allen rekursiven Funktionen gibt es auch bei linearer Rekursion einen Basisfall, der die Rekursion beendet und die Rückkehr der Kontrolle zu den vorherigen Aufrufen ermöglicht. Der Basisfall stellt sicher, dass die Rekursion irgendwann endet und nicht unendlich weiterläuft.

Lineare Rekursion (auch: Struktureller Rekursion)

- Problem P wird auf genau ein einfacheres Problem des gleichen Typs rekursiv zurückgeführt.
 - Diese Art der Probleme können dann auch direkt durch eine while-Schleife gelöst werden (und umgekehrt).
 - Aufgabe: Berechne die Fakultät der Zahl n ohne Verwendung einer Rekursion.

```
int facultaetWhile(int n) {  
    int erg = 1;  
    while (n > 0) {  
        erg = erg*n;  
        n = n-1;  
    }  
    return erg;  
}
```

Baumartige Rekursion

- Problem P wird in einem Rekursionsschritt auf **mehrere Probleme** des gleichen Typs zurückgeführt.
 - Jedes dieser neuen Probleme muss einfacher sein als das ursprüngliche Problem.
 - Diese Programme sind i. A. nicht mehr so einfach in nicht-rekursive Programme mit Schleifen umzusetzen.

Baumartige Rekursion ist eine Form der Rekursion, bei der sich eine Funktion in mehreren parallelen rekursiven Aufrufen verzweigt. Diese Art der Rekursion wird oft als „Baumrekursion“ bezeichnet, weil die Struktur der Rekursionsaufrufe wie ein Baum aussieht, bei dem jeder Knoten (Aufruf) mehrere untergeordnete Knoten (rekursive Aufrufe) haben kann.

Merkmale der Baumartigen Rekursion:
Verzweigte Aufrufe:

Ein einzelner Funktionsaufruf kann mehrere rekursive Aufrufe erzeugen, die parallel bearbeitet werden. Diese Vielzahl an Aufrufen bildet die „Äste“ eines Rekursionsbaums.
Komplexität:

Baumartige Rekursion kann komplexer sein als lineare Rekursion, weil die Anzahl der Aufrufe exponentiell ansteigt, insbesondere wenn die Tiefe des Baums groß wird.

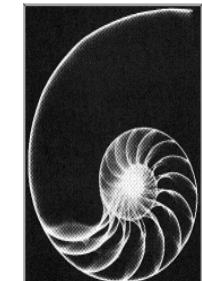
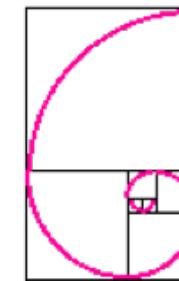
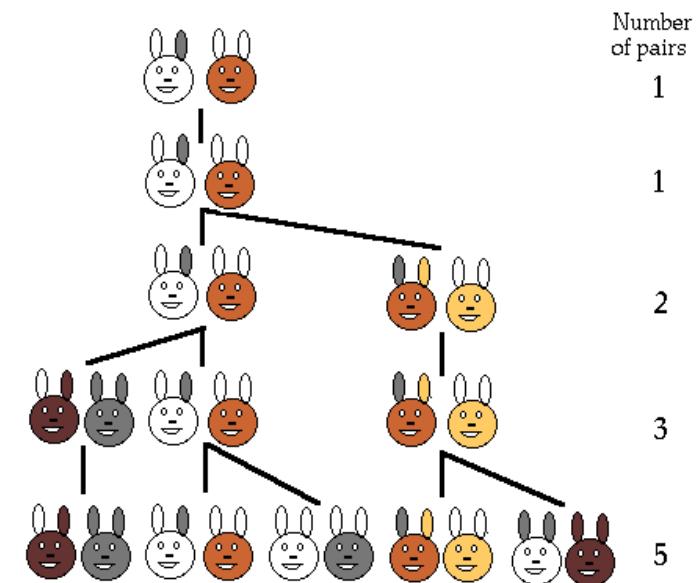
Die Fibonacci-Folge

- $(F_n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$
- Bildungsgesetz:
 - $F_0 = 1, F_1 = 1,$
 - $F_{n+1} = F_n + F_{n-1}$



Trivia:

- 1202 entwickelt um Wachstum von Kaninchenpopulationen zu beschreiben
- Weitere Wachstumsprozesse: Blütenstände, Tannenzapfen, Schneckenhäuser
- Verhältnis einer Fibonacci-Zahl zur vorangegangenen nähert sich Goldenem Schnitt (Ästhetik) an



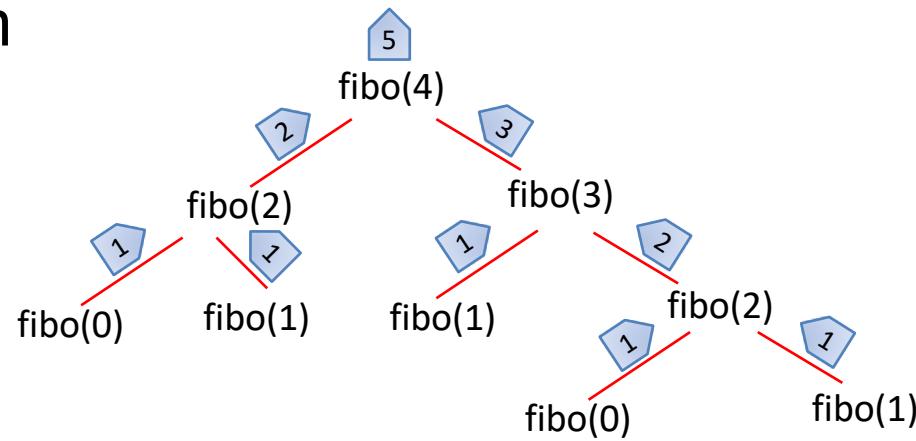


Lösung

- Direkte Umsetzung als rekursive Methode

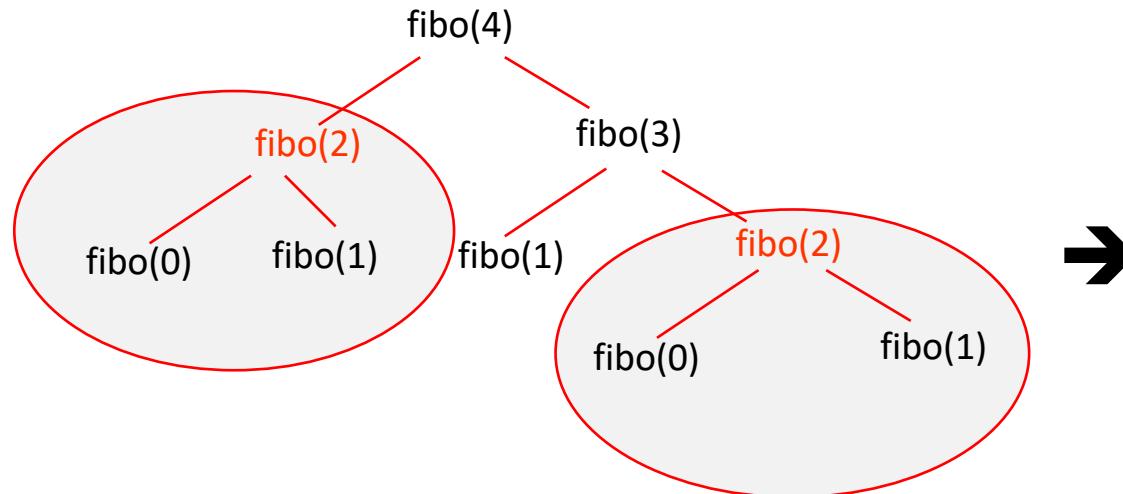
```
int fibo(int n){  
    if (n < 2)  
        return 1;  
    else  
        return fibo(n-2)+fibo(n-1);  
}
```

- Aufrufbaum



3.3.1 Laufzeit

- Durch eine schlecht programmierte Rekursion kann ein Programm viel Zeit benötigen.
 - Das gleiche Problem in einer Rekursion kann **mehrfach berechnet** werden.
- Beispiel: Aufrufbaum von `fibo(4)`



5-mal `fibo(1)` oder `fib(0)`
2-mal `fibo(2)`
1-mal `fibo(3)`

Laufzeit von fibo

- Um es etwas einfacher zu machen, sei n gerade.
- Für die Lösung von $\text{fibo}(n)$ muss für $n > 2$ mindestens 2-mal die Lösung von $\text{fibo}(n-2)$ berechnet werden. Damit folgt:

- 4-mal $\text{fibo}(n-4)$ [bei $n > 4$]
- 8-mal $\text{fibo}(n-6)$ [bei $n > 6$]
- 16-mal $\text{fibo}(n-8)$ [bei $n > 8$]
-
- $2^{n/2} - \text{mal fibo}(0)$

- **Vermutung**
 - Falls $n > 1$ gerade ist, wird $\text{fib}(0) 2^{n/2}$ -mal berechnet.
 - (Formaler Beweis mit Hilfe vollständiger Induktion möglich, wird aber in dieser Vorlesung nicht gezeigt)

Die Laufzeit der Fibonacci-Algorithmus lässt sich optimieren. Siehe VL "Datenstrukturen und Algorithmen", "Deklarative Programmierung".



3.3.2 Terminierung

- Wichtig dabei ist die Vereinfachung des Problems und die Behandlung des einfachsten Sonderfalls.
- Einfachheit lässt sich nicht immer auf einen ganzzahligen Parameter zurückführen.
- Beispiel ggt

```
int ggtRekursiv(int a, int b) {  
    if (a==b) // Stoppbedingung für die Rekursion  
        return a;  
    else {  
        if (a>b) // Rückführung auf ein einfaches Problem  
            return ggtRekursiv(a-b,b);  
        else  
            return ggtRekursiv(b-a,a);  
    }  
}
```

Terminierung rekursiver Funktionen

- Damit eine rekursive Funktion

```
typerg rekFunk (typ1 p1, ..., typk pk) {  
    ... rekFunk (e1, ..., ek) ...  
}
```

terminiert, müssen die Parameterwerte e_1, \dots, e_k des rekursiven Aufrufes „einfacher“ sein als die Parameterwerte p_1, \dots, p_k des originalen Aufrufs

- Was heißt „einfacher“ ?

Es muss eine mathematische Funktion

$$F : t_1 \times \dots \times t_k \rightarrow \mathcal{N}$$

geben mit

$$F(p_1, \dots, p_k) > F(e_1, \dots, e_k) \geq 0.$$

Beispiele

```
static int fibo(int n){  
...  
}
```

$$F(n) = n$$

```
static int ggT(int m,int n){  
...  
}
```

$$F(m,n) = m+n$$

```
static int McCarthy(int n){  
    if (n > 100)  
        return n-10;  
    else  
        return McCarthy(McCarthy(n+11));  
}
```

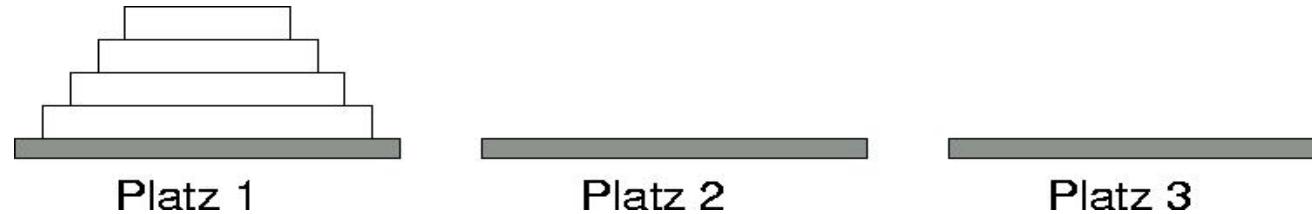
- Experimentieren Sie mit dieser Funktion
- Zeigen Sie, dass sie immer terminiert

John McCarthy war ein Logiker und Informatiker, dem seine großen Beiträge im Feld der Künstlichen Intelligenz den Turing Award von 1971 einbrachten.

3.3.3 Türme von Hanoi

Problem

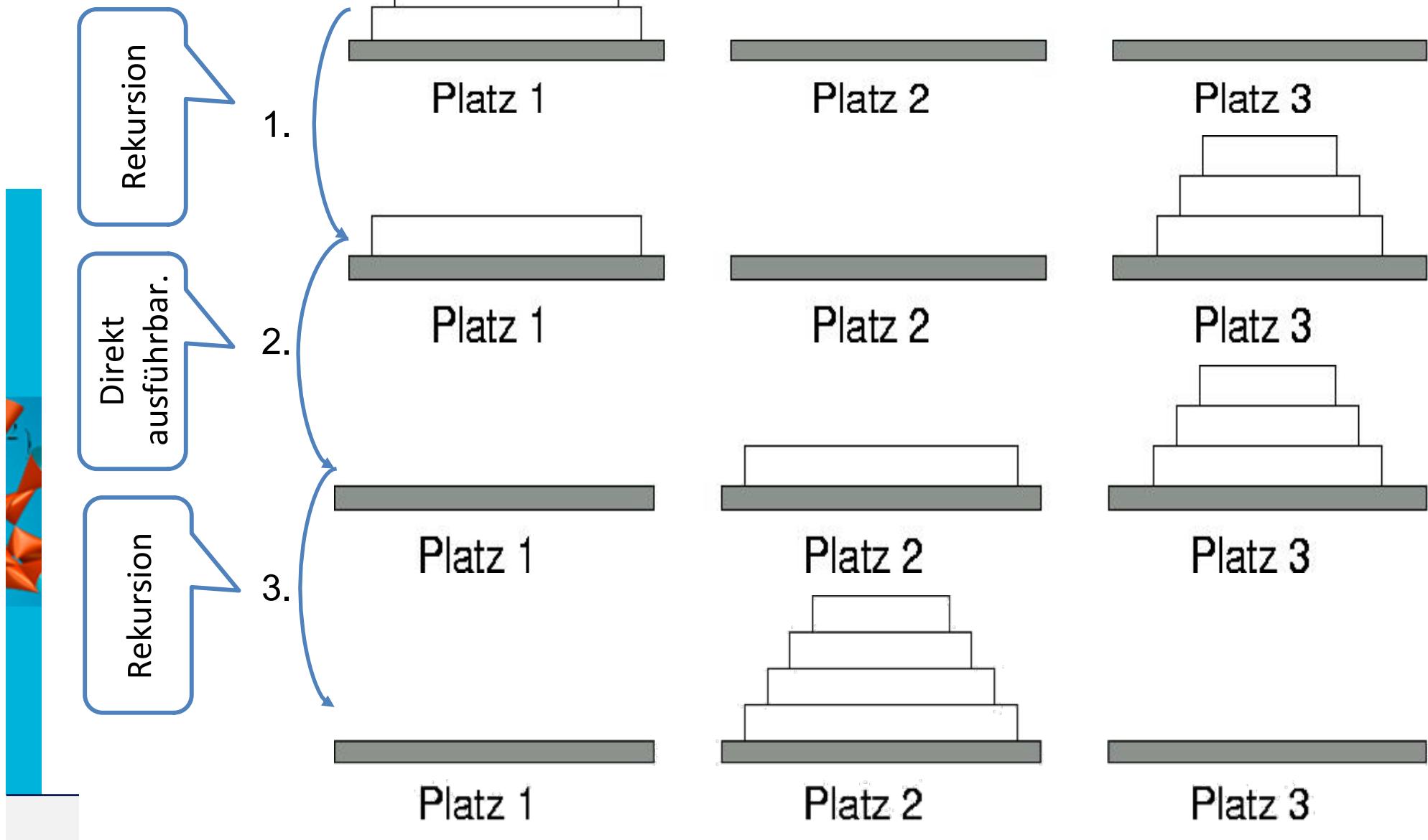
- Gegeben 3 Plätze und ein Stapel von Scheiben, die auf einem der Plätze ihrer Größe nach aufeinander liegen



- Ziel ist es den Stapel unter Beachtung folgender Regeln auf einen anderen Platz zu legen.
 - Pro Zug wird immer **nur eine Scheibe** bewegt.
 - Es darf **nie eine größere auf einer kleineren Scheibe** liegen.
 - Es dürfen alle drei Plätze zur Zwischenlagerung von Scheiben benutzt werden.

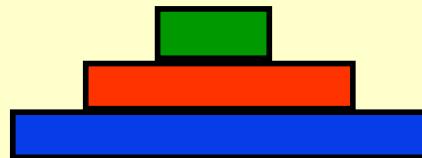
Rekursive Lösung

- Algorithmus für 4 Scheiben von Platz 1 auf Platz 2 (3 dient als Zwischenlager)
 1. Übertrage die oberen 3 Scheiben von Platz 1 nach Platz 3 (Rekursion!)
 2. Bewege die untere Scheibe von Platz 1 nach Platz 2
 3. Übertrage die 3 Scheiben von Platz 3 nach Platz 2 (wieder Rekursion!)
- Verallgemeinerung für n Scheiben von Platz 1 auf Platz 2
 1. Übertrage die oberen $n-1$ Scheiben von Platz 1 nach Platz 3 (Rekursion!)
 2. Bewege die untere Scheibe von Platz 1 nach Platz 2
 3. Übertrage die $n-1$ Scheiben von Platz 3 nach Platz 2 (wieder Rekursion!)



Türme von Hanoi mit drei Scheiben

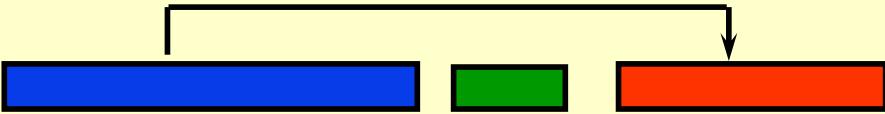
Ausgangslage



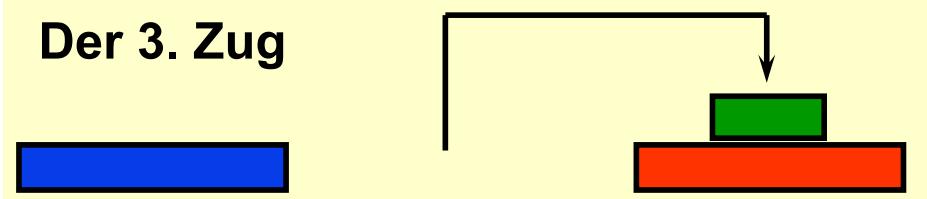
Der 1. Zug



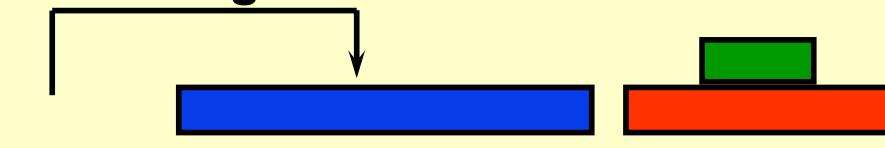
Der 2. Zug



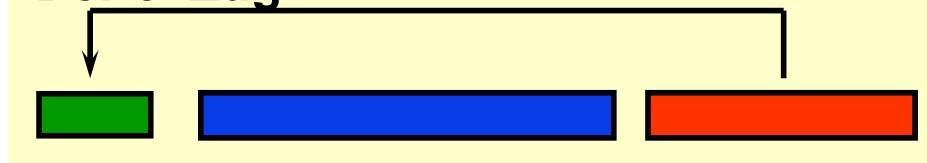
Der 3. Zug



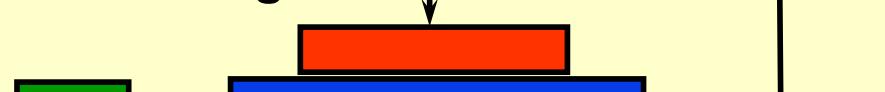
Der 4. Zug



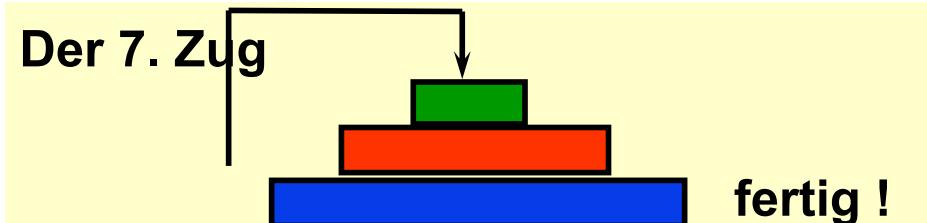
Der 5. Zug



Der 6. Zug



Der 7. Zug



Rekursiver Algorithmus

```
/** Überträgt einen Stapel von n Scheiben von Platz „von“ auf Platz „nach“ und
 * benutzt
 * dabei den Platz „über“. n ist dabei die Anzahl der Scheiben
 */
void bewegeStapel (int n, int von, int nach, int ueber) {
    if (n == 1) // Stopbedingung für die Rekursion
        bewegeScheibe(von, nach);
    else {          // Zurückführung auf einfachere Probleme
        bewegeStapel(n - 1, von, ueber, nach);
        bewegeScheibe(von, nach);
        bewegeStapel(n - 1, ueber, nach, von);
    }
}
```

- Was ist die Ausgabe des Algorithmus
 - In der Methode bewegeScheibe wird der Zug mit System.out.println ausgegeben.
 - Geht dies nicht etwas besser?
 - Ja, aber die Antwort dazu kommt in einem späteren Kapitel.

Anzahl der Züge beim Hanoi-Spiel

- Wenn wir **eine** Scheibe haben, benötigen wir **einen** Zug.
- Wenn wir **zwei** Scheiben haben, benötigen wir **drei** Züge.
- **A(n)** sei die Anzahl der Züge, falls wir **n** Scheiben haben.
- Dann gilt offenbar:

- $A(1) = 1$
- $A(n+1) = A(n) + 1 + A(n) = 1 + 2 \cdot A(n)$

- Die resultierende Folge lautet:
 $1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots$
- Allgemein scheint zu gelten, dass

$$A(n) = 2^n - 1$$

Mehr dazu in
späteren Semestern.

- Dies kann induktiv bewiesen werden:

$$A(1) = 1 = 2^0$$

$$A(n+1) = 1 + 2 \cdot A(n) = 1 + 2 \cdot (2^n - 1) = 1 + 2^{n+1} - 2 = 2^{n+1} - 1$$

Zusammenfassung

- Entwurf von Algorithmen durch schrittweise Verfeinerung
 - Wiederverwendung
- Rekursion ist ein wichtiges Entwurfsprinzip
 - Lösungen werden auf ein Problem geringerer Komplexität des gleichen Typs zurückgeführt.
 - Arten von Rekursion
 - Linear und baumartig
 - Terminierung
 - Laufzeit
- Was kann alles mit Rekursion berechnet werden?
 - Genauso viel wie auf einer Turing-Maschine → Theoretische Informatik.