

Algorithmen und Datenstrukturen

4. Rekursion

Prof. Dr.-Ing. Felix Freiling



Worum geht es in dieser Lehreinheit?

- Nachdem wir nun bereits mit Anweisungen, Ablaufstrukturen und Methoden zentrale Programmiertechniken kennengelernt haben, befassen wir uns in dieser Lerneinheit mit speziellen Methodendeklarationen, die in ihrem Rumpf direkt oder indirekt einen Aufruf von sich selbst enthalten.
- Diese als *Rekursion* bekannte Programmiertechnik ermöglicht bei geeigneten Problemstellungen besonders intuitive ^{直观} und leicht verständliche Algorithmen, allerdings mit der Konsequenz, dass solche rekursiven Problemlösungen als verarbeitungsaufwändiger gelten als solche, die nur die uns bisher bereits bekannten Programmiertechniken verwenden (sog. *iterative Lösungen*).
- Wir lernen verschiedene *Formen von Rekursion* ^{迭代} in Methodendeklarationen kennen und befassen uns dann auch mit der Frage, wie intuitive rekursive in effizientere iterative Lösungen überführt werden können.

Lernziele: Was sollen Sie am Ende dieser Lehreinheit können? (I)

- bei einem rekursiven Programmstück den Typ der Rekursion identifizieren können
- für eine rekursive Methode einen Korrektheits- und Terminierungsbeweis skizzieren können
- zu einer gegebenen Problemstellung einen rekursiven Algorithmus auch unter Berücksichtigung des Induktionsprinzips entwerfen und in Pseudocode-Notation angeben können
- in geeigneten Fällen eine rekursive Methode unter Verwendung von Akkumulatoren in eine endrekursive Methode umwandeln können
- eine endrekursive Methode in eine iterative Methode umwandeln können
- in geeigneten Fällen bzgl. einer Spezifikation semantisch äquivalente iterative und rekursive Variante eines Programmstücks angeben können

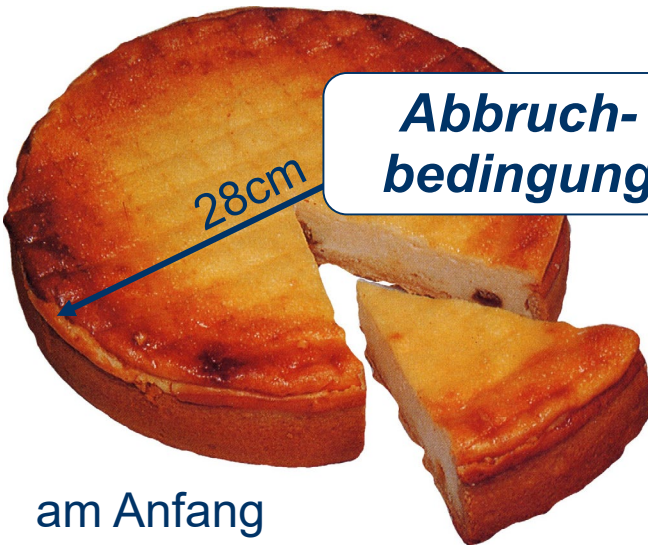
Lernziele: Was sollen Sie am Ende dieser Lehreinheit können? (II)

- einen Algorithmenentwurf in ein Java-Programm unter Verwendung primitiver Datentypen, Arrays, Strings, der Ablaufstrukturen Sequenz, ^{序列} Alternative, Mehrfachauswahl, **while**-Schleife, **do-while**-Schleife, **for**-Schleife sowie Methoden und Rekursion unter Berücksichtigung von Regeln guten Programmierstils umsetzen können
- verschiedene Induktionsformen kennen ^{归纳}
- Induktionsbeweis an geeignetem Beispiel führen können

- 4.1 Grundbegriffe
- 4.2 Lineare Rekursion und Endrekursion
- 4.3 Kaskadenartige Rekursion
- 4.4 Verschränkte und verschachtelte Rekursion

Einführendes Beispiel: Kuchen in Stücke schneiden (I)

- Schneiden Sie den Kuchen in 8 gleiche Stücke!



**Abbruch-
bedingung**

am Anfang
 $\text{rest} = \text{Umfang} = 28\text{cm} \cdot \pi$

```
static void teilen1(double rest,
                    int anzahl) {
    double breite = rest/anzahl;
    while (anzahl >= 1) {
        einSchnitt(breite);
        rest = rest - breite;
        anzahl = anzahl - 1;
    }
}
```

nötig nur für
Analogie zur
Folgefolie.

Iterative Lösung:

- Mehrmaliges Ausführen von Anweisungen in einer Schleife
- Oft zählt man dabei mit, zum wievielten Male eine Aktion ausgeführt wird. (Code verwendet nur deshalb keine **for**-Schleife, um die Analogie zur Folgefolie klarer zu machen.)

Einführendes Beispiel: Kuchen in Stücke schneiden (II)

- Schneiden Sie den Kuchen in 8 gleiche Stücke!



**Abbruch-
bedingung**

```
static void teilen2(double rest,
                   int anzahl) {
    if (anzahl >= 1) {
        double breite = rest/anzahl;
        einSchnitt(breite);
        teilen2(rest - breite,
               anzahl - 1);
    } else {
        //fertig
    }
}
```

Basisfall ohne
rekursiven Aufruf

Rekursive Lösung (I):

- Ebenfalls Form der wiederholten Ausführen von Anweisungen.
- Charakteristisch: Anweisung wird durch sich selbst beschrieben.
- **Rückführung auf ein kleineres, ähnliches Problem;**
Methode ruft sich dazu selbst auf, sog. **Rekursionsschritt**

Einführendes Beispiel: Kuchen in Stücke schneiden (III)

- Schneiden Sie den Kuchen in 8 gleiche Stücke!



**Abbruch-
bedingung**

```
static void teilen2(double rest,
                    int anzahl) {
    if (anzahl >= 1) {
        double breite = rest/anzahl;
        einSchnitt(breite);
        teilen2(rest - breite,
                anzahl - 1);
    } else {
        //fertig
    }
}
```

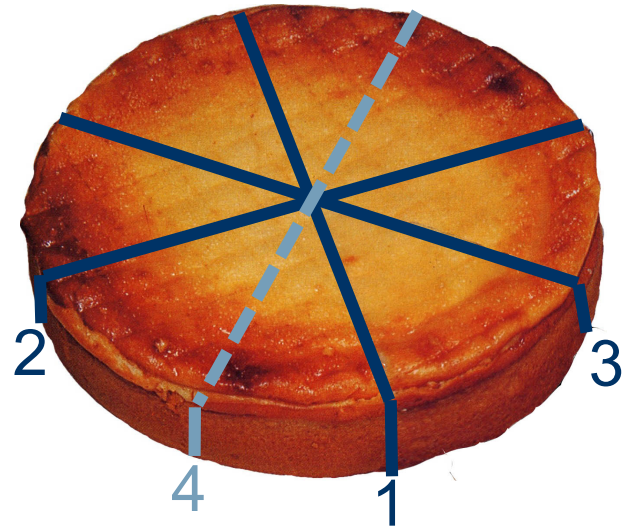
Basisfall hier nur zur Verdeutlichung

Rekursive Lösung (I):

- Methodenaufruf für kleineres Problem und Stückgrößenberechnung $1/8$, $1/7$, $1/6$, $1/5$, ... $1/2$ scheinen hier komplizierter als bei iterativer Lösung.
- Aber: Rekursion ist zentrale Lösungsstrategie für viele Algorithmen.
- Und oft findet man über Rekursion bessere Lösungen...

Einführendes Beispiel: Kuchen in Stücke schneiden (IV)

- Schneiden Sie den Kuchen in 8 gleiche Stücke!



```
static void teilen3(double rest,  
                  int anzahl) {  
  
    if (anzahl >= 1) {  
        halbierungsSchnitt();  
        teilen3(anzahl/2);  
    } else {  
        //fertig  
    }  
  
}
```

**Abbruch-
bedingung**

**Rekursions-
schritt**

Basisfall ohne
rekursiven Aufruf,
zur Verdeutlichung

Rekursive Lösung (II):

- braucht nur 4 Schnitte statt 8.
- Ebenfalls Rückführung auf ein kleineres, ähnliches Problem.
- (Funktioniert nur für 2er-Potenzen.)

Lösungsstrategie bei Verwendung von Rekursion (intuitiv)

■ Vorgehensweise:

- zerlege gegebenes Probl. in kleinere, gleichartige Probleme
- löse Teilproblem(e) und zwar wie folgt:
 - zerlege gegebenes Probl. in kleinere, gleichartige Probleme
 - löse Teilproblem(e) und zwar wie folgt:
 - zerlege gegebenes Probl. in kleinere, gleichartige Probleme
 - löse Teilproblem(e) und zwar wie folgt
 - usw.
 - wenn zu lösendes Problem klein genug ist, dann löse es direkt
 - setze Teillösungen zu Gesamtlösung zusammen
 - setze Teillösungen zu Gesamtlösung zusammen
- setze Teillösungen zu Gesamtlösung zusammen

Voraussetzungen, damit rekursiver Ansatz erfolgreich

- Nicht direkt lösbare Teilprobleme müssen dem Ausgangsproblem ähnlich sein (**Selbstähnlichkeit**)
- Teilprobleme werden bei jedem weiteren rekursiven Aufruf bzgl. einer Ordnung kleiner (**Monotonieeigenschaft**)
- Größe der Teilprobleme ist nach unten beschränkt (**Beschränktheit**)
- Beispiele:
 - Restkuchen ist wie ein Kuchen, lediglich um eine Stück kleiner (Selbstähnlichkeit).
 - Restkuchen wird mit jedem rekursiver Aufruf um ein Stück kleiner (Monotonieeigenschaft).
 - wenn der Restkuchen kein Stück mehr hergibt, folgen keine weiteren rekursiven Aufrufe (Beschränktheit).



Gliederung der Lehreinheit

4.1 Grundbegriffe

→ 4.2 Lineare Rekursion und Endrekursion

4.3 Kaskadenartige Rekursion

4.4 Verschränkte und verschachtelte Rekursion



Zu Oberflächlich!
Problemgröße wird nicht
(monoton) verkleinert.
Basisfall fehlt.

Fakultätsfunktion (I)

- **Fakultätsfunktion:** $f(n) = n!$

- **iterative Definition:** $n! = 1 \cdot 2 \cdot \dots \cdot n = \prod_{k=1}^n k$

Beispiel (iterativ): $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

- **rekursive Definition:**
$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n \cdot (n-1)! & \text{falls } n > 1 \end{cases}$$

Beispiel (rekursiv):

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! \\ &= 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned}$$

Arbeitsauftrag für Sie

Erstellen Sie eine iterative Fassung der Fakultätsfunktion in Java.



$$n! = 1 \cdot 2 \cdot \dots \cdot n = \prod_{k=1}^n k$$

Fakultätsfunktion (III)

- rekursive Definition (für $n > 0$):
$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n \cdot (n - 1)! & \text{falls } n > 1 \end{cases}$$

- Umsetzung in Java:

Variante 1: mit **if-else**-Anweisung (Fallunterscheidung)

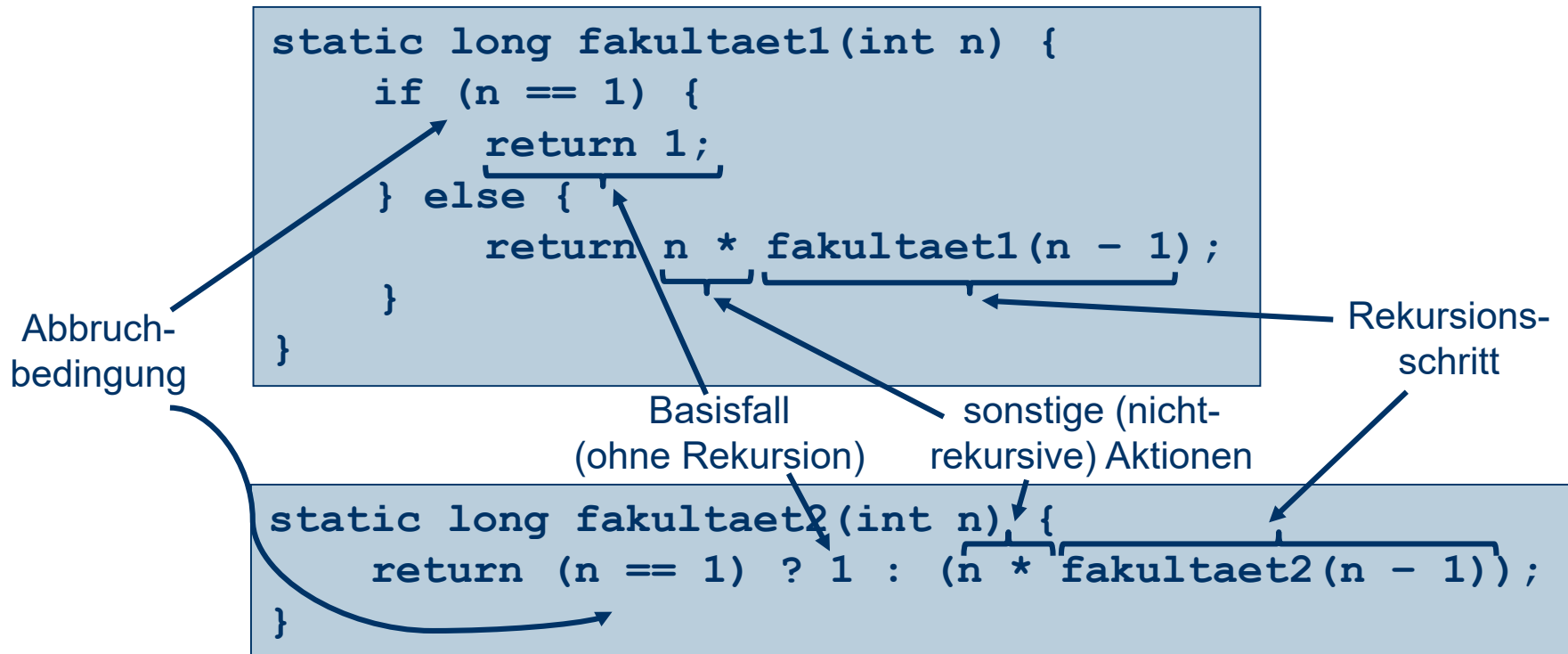
```
static long fakultaet1(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * fakultaet1(n - 1);  
    }  
}
```

Variante 2: mit **?-Operator** (bedingter Ausdruck)

```
static long fakultaet2(int n) {  
    return (n == 1) ? 1 : (n * fakultaet2(n - 1));  
} boolean
```

Fakultätsfunktion (IV)

- rekursive Definition (für $n > 0$):
$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n \cdot (n - 1)! & \text{falls } n > 1 \end{cases}$$
- hier: typischer Aufbau mit Fallunterscheidung von Basisfall (Ende der Selbstaufrufe) und Fall mit Rekursionsschritt



- **Rekursion:** allgemeines Prinzip zur Lösung von Problemen, bei der die Berechnung auf die eigene Berechnung direkt oder indirekt zurückverweist
- mit dieser wird auf dieselbe Weise verfahren
- Durchführung bricht ab, wenn eine vorgegebene Abbruchbedingung erreicht wird
- Beispiel für die Anwendbarkeit von Rekursion ist das „**Teile und Herrsche/Divide and Conquer-Prinzip**“:
 1. Zerlege ein großes Problem in kleinere Versionen des Problems.
 2. Falls die kleineren Versionen noch zu groß sind, löse sie auf die gleiche Weise.
 3. Falls die kleineren Versionen hinreichend klein sind, löse sie direkt.
 4. Füge die Lösung kleinerer Versionen des Problems zu einer Lösung des großen Problems zusammen.

(Linear) Rekursive Methode, rekursive Methodendefinition

- Eine Methode f heißt **rekursiv** (lat. recurrere: zurücklaufen, -kehren), gdw. zur Berechnung von f diese Methode f wieder benutzt wird.
- Eine Methode f heißt **linear rekursiv**, wenn sie in jedem Zweig einer Fallunterscheidung ihres Rumpfes höchstens einmal aufgerufen wird.
- Eine Methodendeklaration $f(x) \{ \dots \}$ heißt **rekursive Methodendefinition**, gdw. f im Block $\{ \dots \}$ aufgerufen wird.

Variante (sog. verschränkte Rekursion, Details folgen):

- im Rumpf von f steht der Aufruf einer Methode g ,
im Rumpf von g steht ein Aufruf der Methode f
 - f ist dann zwar rekursiv, hat aber keine rekursive Methodendefinition
- Beispiel: `fakultaet1/2` sind rekursiv,
sogar linear rekursiv - haben rekursive Methodendefinition

Auswertung des Methodenaufrufs fakultaet(4)

`fakultaet(4)` = 4 * fakultaet(3) = 4 · 6 = 24

`fakultaet(3)` = 3 * fakultaet(2) = 3 · 2 = 6

`fakultaet(2)` = 2 * fakultaet(1) = 2 · 1 = 2

`fakultaet(1)` = 1

Korrektheitsüberlegungen

- Wir haben nun eine Java-Implementierung zur Fakultätsfunktion angegeben.
- Frage: Inwieweit können wir sicher sein, dass die Implementierung korrekt ist und immer das gewünschte Ergebnis liefert?
- **Variante 1: Testansatz** (für Auswahl von Eingaben):
 - möglich: für jede Eingabe kann das zugehörige Ergebnis durch schrittweises Ausführen ermittelt werden
 - Problem: nicht alle erdenklichen Eingaben ausprobierbar (Zeit!)
 - letzte Unsicherheit bleibt bzgl. der übrigen Eingaben
- **Variante 2: Korrektheitsbeweis** (für alle möglichen Eingaben):
 - mit geeigneten Beweismethoden den Nachweis erbringen, dass die Methode für jede beliebige Eingabe immer das korrekte Ergebnis bzgl. der Spezifikation berechnet
 - im Zusammenhang mit rekursiven Methoden kommt dabei oft die Beweistechnik der **vollständigen Induktion** zum Einsatz
完全归纳

Einschub: Beweistechnik vollständige Induktion (I)

- **Vollständige Induktion:** mathematische Beweismethode, mit der eine Aussage für alle natürlichen Zahlen n bewiesen wird
- da Menge der natürlichen Zahlen unendlich ist, kann ein solcher Beweis nicht für alle Einzelfälle durchgeführt werden
- Beweis erfolgt daher i. d. R. in zwei Etappen:
 1. sog. **Induktionsanfang**: beweise die Aussage für eine kleinste Zahl n_0 (meist 0 oder 1)
 2. sog. **Induktionsschluss** (oder: **-schritt**): beweise unter der Hypothese, dass die Aussage für ein beliebiges $n \geq n_0$ bereits gilt (sog. **Induktionsvoraussetzung**), dass die Aussage dann auch für $n + 1$ gilt

wenn das gelingt, folgt daraus direkt, dass die Aussage für alle natürlichen Zahlen gilt, denn mit 1. gilt die Aussage für $n = n_0$, mit 2. damit auch für $n = n_0 + 1$ und damit auch für $n = n_0 + 2$ usw.

Einschub: Beweistechnik vollständige Induktion (II)

- Induktion als Beweistechnik für Aussagen, die sich auf die natürlichen Zahlen beziehen
- Ziel: Nachweis der Gültigkeit einer Aussage $A(n)$ für alle $n \in \mathbb{N}$

a) Basisform der Induktion (Variante 1: „+1“)

- Bedingungen (zu beweisen):
 - $A(1)$ ist wahr
 - Für jedes $n \geq 1$ gilt: $A(n) \Rightarrow A(n + 1)$
- Dann gilt $A(n)$ für alle n .

b) Basisform der Induktion (Variante 2: „-1“)

- Bedingungen (zu beweisen):
 - $A(1)$ ist wahr
 - Für jedes $n > 1$ gilt: $A(n - 1) \Rightarrow A(n)$
- Dann gilt $A(n)$ für alle n .

Einschub: Beweistechnik vollständige Induktion (III)

- hier: k Anfangswerte $A(1), \dots, A(k)$ gegeben
- Strategie nun: „ k parallele Pfade durch \mathbb{N} , die \mathbb{N} ganz überdecken“
- Ziel: Nachweis der Gültigkeit einer Aussage $A(n)$ für alle $n \in \mathbb{N}$

c) ***k -Anfangsform der Induktion (Variante 1: „+ k “)***

- Bedingungen (zu beweisen):
 - $A(1), \dots, A(k)$ sind wahr
 - Für jedes $n \geq 1$ gilt: $A(n) \Rightarrow A(n + k)$
- Dann gilt $A(n)$ für alle n .

d) ***k -Anfangsform der Induktion (Variante 2: „- k “)***

- Bedingungen (zu beweisen):
 - $A(1), \dots, A(k)$ sind wahr
 - Für jedes $n > k$ gilt: $A(n - k) \Rightarrow A(n)$
- Dann gilt $A(n)$ für alle n .

Einschub: Beweistechnik vollständige Induktion (IV)

- Ziel: Nachweis der Gültigkeit einer Aussage $A(n)$ für alle $n \in \mathbb{N}$

e) *strenge Induktion*

- Bedingungen (zu beweisen):
 - $A(1)$ ist wahr
 - Für jedes $n > 1$ gilt: $(\forall m < n: A(m)) \Rightarrow A(n)$,
also: $A(1) \wedge \dots \wedge A(n-1) \Rightarrow A(n)$
d.h.: $A(m)$ gilt nicht nur für einen Vorgänger, sondern für alle
- Dann gilt $A(n)$ für alle n .

f) *Zweierpotenz-Induktion*

- Bedingungen (zu beweisen):
 - $A(1)$ ist wahr
 - Für jedes $n = 2^k > 1$ gilt: $A\left(\frac{n}{2}\right) \Rightarrow A(n)$
- Dann gilt $A(n)$ für alle $n = 2^k$.

Einschub: Beweistechnik vollständige Induktion (V)

- Ziel: Nachweis der Gültigkeit einer Aussage $A(n)$ für alle $n \in \mathbb{N}$

g) Rückwärtsinduktion

- Bedingungen:
 - $A(n)$ ist wahr für alle n aus einer unendlichen Teilmenge von \mathbb{N}
 - Für jedes $n > 2$: $A(n) \Rightarrow A(n - 1)$
 - Für ein m lässt sich immer ein größeres k in dieser Teilmenge finden.
 - Von diesem k „arbeitet man sich rückwärts bis m vor“ und zeigt so $A(m)$.
- Dann gilt $A(n)$ für alle n .

h) strukturelle Induktion

- Beweis über die Konstruktionsregeln einer Struktur

Korrektheitsbeweis für die Methode `fakultaet`

- Mathematische Definition der Fakultät: $n! = 1 \cdot 2 \cdot \dots \cdot n = \prod_{k=1}^n k$
- Java-Implementierung:

```
static long fakultaet2(int n) {  
    return (n == 1) ? 1 : (n * fakultaet2(n - 1));  
}
```

- Nachweis von `fakultaet2(n) = n!` für alle $n \geq 1$:

- Induktionsanfang: `fakultaet2(1) = 1 = $\prod_{k=1}^1 k = 1 = 1!$` ✓
- Induktionsschluss (Schritt von n auf $n + 1$):

$$\text{fakultaet2}(n + 1) = (n + 1) \cdot \text{fakultaet2}(n)$$

Def.von `fakultaet2`

$$= (n + 1) \cdot n!$$

Induktionsvorausstzg.

$$= (n + 1) \cdot \prod_{k=1}^n k$$

Definition von $n!$

$$= \prod_{k=1}^{n+1} k \quad \checkmark$$

Definition von \prod

Achtung: Korrektheitsnachweis nur für alle n , die einen Funktionswert innerhalb des Wertebereichs des Datentyps `long` liefern

Zur Terminierung der Methode `fakultaet`

- Gefahr: fehlerhafter Entwurf einer rekursiven Methode (z. B. nicht-rekursiver Basis-/Abbruchfall vergessen) führt möglicherweise dazu, dass diese nicht terminiert (sog. **Endlosrekursion**)
- Beispiel: *fehlerhafte Version* von `fakultaet2` *ohne Basisfall*

```
static long fakultaet2f(int n) {  
    return n * fakultaet2f(n - 1);  
}
```

- Auswertung des Aufrufs `fakultaet2f(2)` führt zur Berechnung $2 * 1 * 0 * (-1) * (-2) * \dots$
- Die ineinander geschachtelten Methodenaufrufe lassen das Programm irgendwann mit einem sog. **StackOverflow** abbrechen

→ Frage: Wie kann man nachweisen, dass ein rekursives Programm zum Ende kommt (**terminiert**)?

- **Variante 1: Testansatz** (für Auswahl von Eingaben)
 - Hoffnung: Ausführung endet nach gewisser Zeit
 - im Erfolgsfall Terminierung damit feststellbar
 - Problem:
 - Was tun, wenn Ausführung lange dauert?
 - Kann man sich sicher sein, dass die Bearbeitung des Aufrufs irgendwann noch zu einem Ergebnis kommt?
- **Variante 2: Terminierungsbeweis** (für alle möglichen Eingaben)
 - Vorgehensweise:
 - e_0, e_1, e_2, \dots sei Argumentfolge der Rekursionsschritte bei Auswertung eines Methodensaufrufs
 - finde ganzzahlige **Terminierungsfunktion** $t(e_i)$ und beweise (z. B. per Induktion):
 - t fällt bei jedem Rekursionsschritt streng monoton
 - t ist nach unten beschränkt

Terminierungsbeweis für die Methode `fakultaet`

- Java-Implementierung:

```
static long fakultaet2(int n) {  
    return (n == 1) ? 1 : (n * fakultaet2(n - 1));  
}
```

- Argumentfolge betrachten: $n, n-1, n-2, \dots$
- Terminierungsfunktion finden: $t(n) = n$
- Nachweisskizze (für ganzzahlige $n > 0$):
 - $t(n)$ ist auf der Folge der Argumente streng monoton fallend bei jedem Rekursionsschritt (und zwar jeweils um 1), beweisbar mittels vollständiger Induktion
 - bei der impliziten Annahme, dass n ganzzahlig (`int`) und $n > 0$ ist, ist $t(n)$ nach unten durch 1 beschränkt
 - zeigt zusammen mit Nachweis der Übereinstimmung mit Spezifikation sog. **totale Korrektheit** (für Argumentwerte größer als 0)

Zusammenhang von Rekursion und Induktion (I)

Rekursion

Programmiertechnik
in der Informatik

Rekursive Methoden
werden strukturiert in:

(nicht-rekursiver)
Basisfall

Rekursionsfall mit

- Basisoperationen
- Rekursionsschritt

Induktion

Beweistechnik
in der Mathematik

induktive Beweise
werden strukturiert in:

Induktionsanfang

Induktionsschritt

Größter gemeinsamer Teiler (ggT) (I)

- größter gemeinsamer Teiler zweier ganzer Zahlen m und n
 $ggT(m, n)$ ist die größte natürliche Zahl, durch die sowohl m als auch n ohne Rest teilbar sind
- Verfahren zur Bestimmung des ggTs:
 - **Variante 1: Primfaktorzerlegung**
因式分解
经典的欧几里得算法
 - **Variante 2: klassischer Euklidischer Algorithmus**
 - subtrahiere die kleinere Zahl von der größeren
 - ersetze größere ursprüngliche Zahl durch die so gewonnene Differenz
 - wiederhole diese Schritte solange, bis die beiden Zahlen gleich groß sind
 - Ergebnis ist der ggT der beiden Zahlen

Größter gemeinsamer Teiler (ggT) (II)

- Pseudocode zum klassischen Euklidschen Algorithmus

`ggT(a, b) :`

`wenn a = b ist, dann gib a zurück;`

`sonst wenn a < b dann gib ggT(a, b - a) zurück;`

`sonst gib ggT(a - b, b) zurück;`

- Implementierung in Java

```
static int ggT(int a, int b) {  
    if (a == b) {  
        return a;  
    } else if (a < b) {  
        return ggT(a, b - a);  
    } else {  
        return ggT(a - b, b);  
    }  
}
```

Auswertung von `ggT(18, 12)` (I)

```
static int ggT(int a, int b) {  
    if (a == b) {  
        return a;  
    } else if (a < b) {  
        return ggT(a, b - a);  
    } else return ggT(a - b, b);  
}
```

```
...  
ggT(18, 12);  
...
```

anlegen

Methodenschachtel

```
a: 18  
b: 12  
erg:  
Rücksprungziel  
konzeptuell:  
...  
else {  
    erg = ggT(6, 12);  
    return erg;  
}  
...
```

→ zur Laufzeit existieren viele Kopien
der methodenlokalen Variablen

Methodenschachtel

```
a: 6  
b: 12  
erg:  
Rücksprungziel  
konzeptuell:  
...  
... if (a < b) {  
    erg = ggT(6, 6);  
    return erg;  
}  
...
```

Methodenschachtel

```
a: 6  
b: 6  
erg: 6  
Rücksprungziel  
konzeptuell:  
...  
if (a == b) {  
    erg = 6;  
    return erg;  
}  
...
```

Argumente werden in die
Parametervariablen kopiert

Auswertung von ggT(18, 12) (II)

```
static int ggT(int a, int b) {  
    if (a == b) {  
        return a;  
    } else if (a < b) {  
        return ggT(a, b - a);  
    } else return ggT(a - b, b);  
}
```

```
...  
ggT(18, 12);  
...
```

anlegen

Methodenschachtel

a: 18
b: 12
erg:
Rücksprungziel
konzeptuell:
...
else {
 erg = ggT(6, 12);
 return erg;
}
...

Methodenschachtel

a: 6
b: 12
erg: 6
Rücksprungziel
konzeptuell:
...
... if (a < b) {
 erg = ggT(6, 6);
 return erg;
}
...

→ bei der Rückkehr
aus der aufgerufenen
Methode werden die
Methodenschachteln
wieder abgebaut

Auswertung von ggT(18, 12) (III)

```
static int ggT(int a, int b) {  
    if (a == b) {  
        return a;  
    } else if (a < b) {  
        return ggT(a, b - a);  
    } else return ggT(a - b, b);  
}
```

...
ggT(18, 12);
...

anlegen

Methodenschachtel

a: 18
b: 12
erg: 6
Rücksprungziel
konzeptuell:
...
else {
 erg = ~~ggT(18, 12);~~ 6
 return erg;
}
...

Größter gemeinsamer Teiler (ggT) (III)

□ **Variante 3: optimierter Euklidischer Algorithmus**

- Nachteil des klassischen Euklidischen Algorithmus: bei großen Ausgangszahlen ggf. viele Subtraktionen erforderlich
- deshalb:
 - in aufeinander folgenden Schritten jeweils Division mit Rest
 - Divisor wird im nächsten Schritt zum neuen Dividenden und Rest zum neuen Divisor
 - Divisor, bei dem sich Rest 0 ergibt, ist der ggT der Ausgangszahlen
- Beispiel:

3528 : 2100 = 1 Rest 1428

2100 : 1428 = 1 Rest 672

1428 : 672 = 2 Rest 84

672 : 84 = 8 Rest 0

□ Implementierung in Java:

```
static int ggT2(int a, int b) {  
    if (b == 0) {  
        return a;  
    } else {  
        return ggT2(b, a % b);  
    }  
}
```

Endrekursion (engl: tail recursion, auch: Rumpfrekursion, repetitive Rekursion, iterative Rekursion)

- Eine Methode f heißt **endrekursiv** (engl. *tail recursive*, auch: *rumpfrekursiv*, *repetitiv rekursiv*, *iterativ rekursiv*), wenn der rekursive Methodenaufruf stets die letzte Aktion des Zweigs zur Berechnung von f ist.
- Endrekursion ist ein Spezialfall der linearen Rekursion.
- Eine endrekursive Methode kann unmittelbar **entrekursiviert** werden. Das ist deshalb erstrebenswert, da rekursive Lösung zwar i. d. R. intuitiv aufzuschreiben, iterative Lösungen i. d. R. aber schneller in der Ausführung sind.
- Beispiele:
 - Die Methoden `ggT`, `teilen2/3` sind endrekursiv (und damit auch linear rekursiv).
 - Die Methoden `fakultaet1/2` sind linear rekursiv, aber nicht endrekursiv.

Entrekursivierung endrekursiver Methoden

- **Entrekursivierung:** überführe eine rekursive Methode in eine bzgl. der Spezifikation äquivalente iterative Methode

- Endrekursive Methode:

$$f(x) = \begin{cases} g(x) & \text{falls } \textit{praed}(x) \quad \text{(nicht-rekursiver Basisfall, wenn } x \text{ Eigenschaft } \textit{praed}(x) \text{ erfüllt)} \\ f(r(x)) & \text{sonst} \quad \text{(rekursiver Aufruf; durch Methode } r \text{ modifizierter („kleinerer“) Parameter)} \end{cases}$$

- entrekursivierte Fassung:

```
arg = x;                // Hilfsvariable
while (!praed(arg)) {
    arg = r(arg);        // modifiziere Parameter
}
return g(arg);           // nicht-rekursiver Basisfall
```

Einschub: endrekursive vs. linear rekursive Methode

- Endrekursive Methode (Spezialfall der linearen Rekursion):

$$f(x) = \begin{cases} g(x) & \text{falls } \textit{praed}(x) & \text{(nicht-rekursiver Basisfall, wenn } x \text{ Eigenschaft } \textit{praed}(x) \text{ erfüllt)} \\ f(r(x)) & \text{sonst} & \text{(rekursiver Aufruf; durch Methode } r \text{ modifizierter („kleinerer“) Parameter)} \end{cases}$$

- linear rekursive Methode:

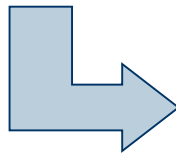
$$f(x) = \begin{cases} g(x) & \text{falls } \textit{praed}(x) & \text{(nicht-rekursiver Basisfall, wenn } x \text{ bestimmte } \textit{praed}(x) \text{ erfüllt)} \\ h(x, f(r(x))) & \text{sonst} & \text{(rekursiver Aufruf; durch Methode } r \text{ modifizierter („kleinerer“) Parameter; Rekursionsergebnis wird anschließend mittels Methode } h \text{ weiterverarbeitet)} \end{cases}$$

Entrekursivierung am Beispiel der Methode `sum` (I)

1. rekursive (aber nicht endrekursive) Implementierung

```
static int sum1(int n) {  
    if (n == 0) {  
        return 0;  
    } else {  
        return n + sum1(n-1);  
    }  
}
```

umwandeln



2. endrekursive Implementierung

```
static int sum2(int n) {  
    return add_sum(0, n);  
}  
  
static int add_sum(int m,  
                   int n) {  
    if (n == 0) {  
        return m;  
    } else {  
        return add_sum(m+n, n-1);  
    }  
}
```

Entrekursivierung am Beispiel der Methode `sum` (II)

2. endrekursive Implementierung 可以直接取消循环

- endrekursive Hilfsmethode `add_sum` liefert als Ergebnis die Summe aus `m` und der Summe der ersten `n` natürlichen Zahlen
- während des Ablaufs der Endrekursion in `add_sum` werden die Zwischenergebnisse im `m`-Parameter **akkumuliert**
- Auswertung von `sum2(3)`:
$$\begin{aligned} \text{sum2}(3) &= \text{add_sum}(0, 3) \\ &= \text{add_sum}(3, 2) \\ &= \text{add_sum}(5, 1) \\ &= \text{add_sum}(6, 0) \\ &= 6 \end{aligned}$$

```
static int sum2(int n) {  
    return add_sum(0, n);  
}  
  
static int add_sum(int m,  
                  int n) {  
    if (n == 0) {  
        return m;  
    } else {  
        return add_sum(m+n, n-1);  
    }  
}
```

Entrekursivierung am Beispiel der Methode `sum` (III)

Hinweis: **x** und **arg** bezeichnen hier Argumentvektor

repetitiv rekursive Methode:

$$f(x) = \begin{cases} g(x) & \text{falls } \textit{praed}(x) \\ f(r(x)) & \text{sonst} \end{cases}$$

```
static int sum2(int n) {  
    return add_sum(0, n);  
}  
  
static int add_sum(int m, int n) {  
    if (n == 0) {  
        return m;  
    } else {  
        return add_sum(m+n, n-1);  
    }  
}
```

if-else 和 while

entrekursivierte Implementierung:

```
arg = x;  
while (!praed(arg)) {  
    arg = r(arg);  
}  
return g(arg);
```

```
static int sum3(int n) {  
    int m = 0;  
    while (n > 0) {  
        {  
            m = m + n;  
            n = n - 1;  
        }  
    }  
    return m;  
}
```


Entrekursivierung am Beispiel der Methode `sum` (IV)

- typische Frage im Zusammenhang der Algorithmik:
gibt es eine noch bessere (effizientere) Lösung?
- ja: Gaußsche Summenformel anwenden
- Warum ist diese Lösung effizienter?
 - Anzahl der Operationen als Vergleichsmaß
 - bei `sum3` ist die Anzahl der Schleifendurchläufe abhängig von `n`:
Schleife wird `n` mal durchlaufen
 - bei `sum4` ist die Anzahl der Operationen konstant und unabhängig
vom Parameter `n`

```
static int sum4(int n) {  
    return (n * (n + 1)) / 2;  
}
```

```
static int sum3(int n) {  
    int m = 0;  
    while (n > 0) {  
        {  
            m = m + n;  
            n = n - 1;  
        }  
    }  
    return m;  
}
```

Warum Rekursion und warum Entrekursivierung? (I)

- Frage: Warum führen wir Rekursion als Programmiertechnik ein, wenn wir dann anschließend versuchen, rekursiv gestaltete Problemlösungen wieder zu entrekursivieren?
- für Rekursion spricht:
 - Elegante, intuitive, übersichtliche Problemlösungen.
 - Programmtext ist im Allg. lesbarer und weniger fehleranfällig.
 - Besonders geeignet für Algorithmen, die auf induktiv definierten Daten arbeiten, wodurch das Problem leicht in kleinere selbstähnliche Teilprobleme zerfällt.
 - Beweise von Eigenschaften, wie Übereinstimmung mit Spezifikation und Terminierung, sind oft nahe liegend per vollständiger Induktion.

Warum Rekursion und warum Entrekursivierung? (II)

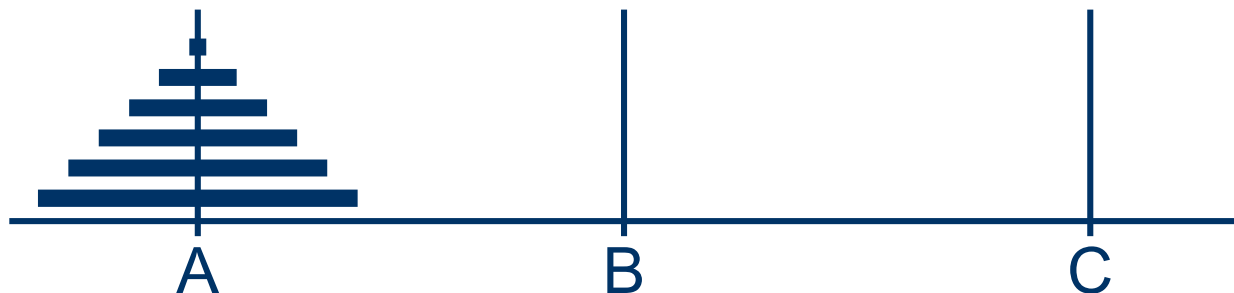
- Verwendung von Endrekursion zur Effizienzsteigerung:
 - Abarbeitung einer rekursiven Methode:
 - Speicherplatzverbrauch steigt linear mit der Rekursionstiefe.
 - Bei jedem Methodenaufruf wird Speicherplatz auf dem Laufzeitstapel für das Sichern der Rücksprungadresse, der Methodenparameter und ggf. methodenlokaler Variablen belegt.
 - Bei endrekursivem Methodenaufruf:
 - Die im für die aufrufende Methode belegten Speicherbereich abgelegten Werte werden nur noch für die Parameterübergabe an die endständig aufgerufene Methode benötigt.
 - Dieser Speicherbereich kann wiederverwendet werden.
→ Speicherbedarf unabhängig von Rekursionstiefe.
 - Manche optimierende Übersetzer entrekursivieren Endrekursion automatisch.

- 4.1 Grundbegriffe
- 4.2 Lineare Rekursion und Endrekursion
- 4.3 Kaskadenartige Rekursion 级联递归
- 4.4 Verschränkte und verschachtelte Rekursion

Türme von Hanoi

Problemspezifikation

- gegeben:
 - k Scheiben unterschiedlichen Durchmessers, der Größe nach geordnet auf Position A
 - Positionen B und C ohne Scheiben



- Problem: versetze Scheiben-Turm von A nach B , wobei
 - jede Scheibe nur einzeln bewegt werden kann
 - immer nur die oberste Scheibe eines Turmes bewegt werden kann
 - niemals eine größere Scheibe auf einer kleineren liegen darf und
 - C zur Zwischenablage (als Hilfsstapel) benutzt werden kann

Videos zu den Türmen von Hanoi (Auswahl)



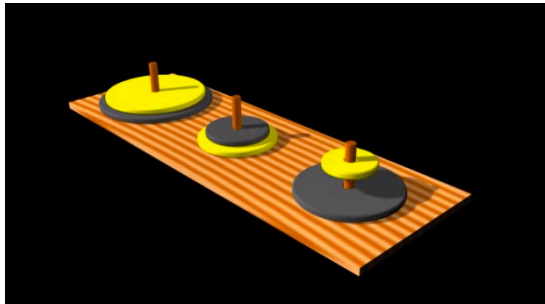
<http://www.youtube.com/watch?v=wzzD8SVPcZU>



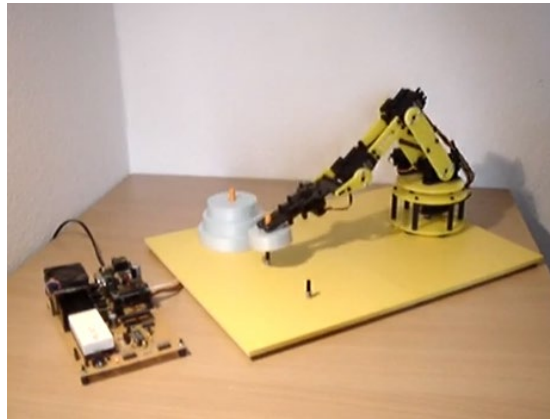
<http://www.youtube.com/watch?v=gadQDCRr7U8>



<http://www.youtube.com/watch?v=w9LgLiW9YHU>



<http://www.youtube.com/watch?v=1bIHt5HVVDs>



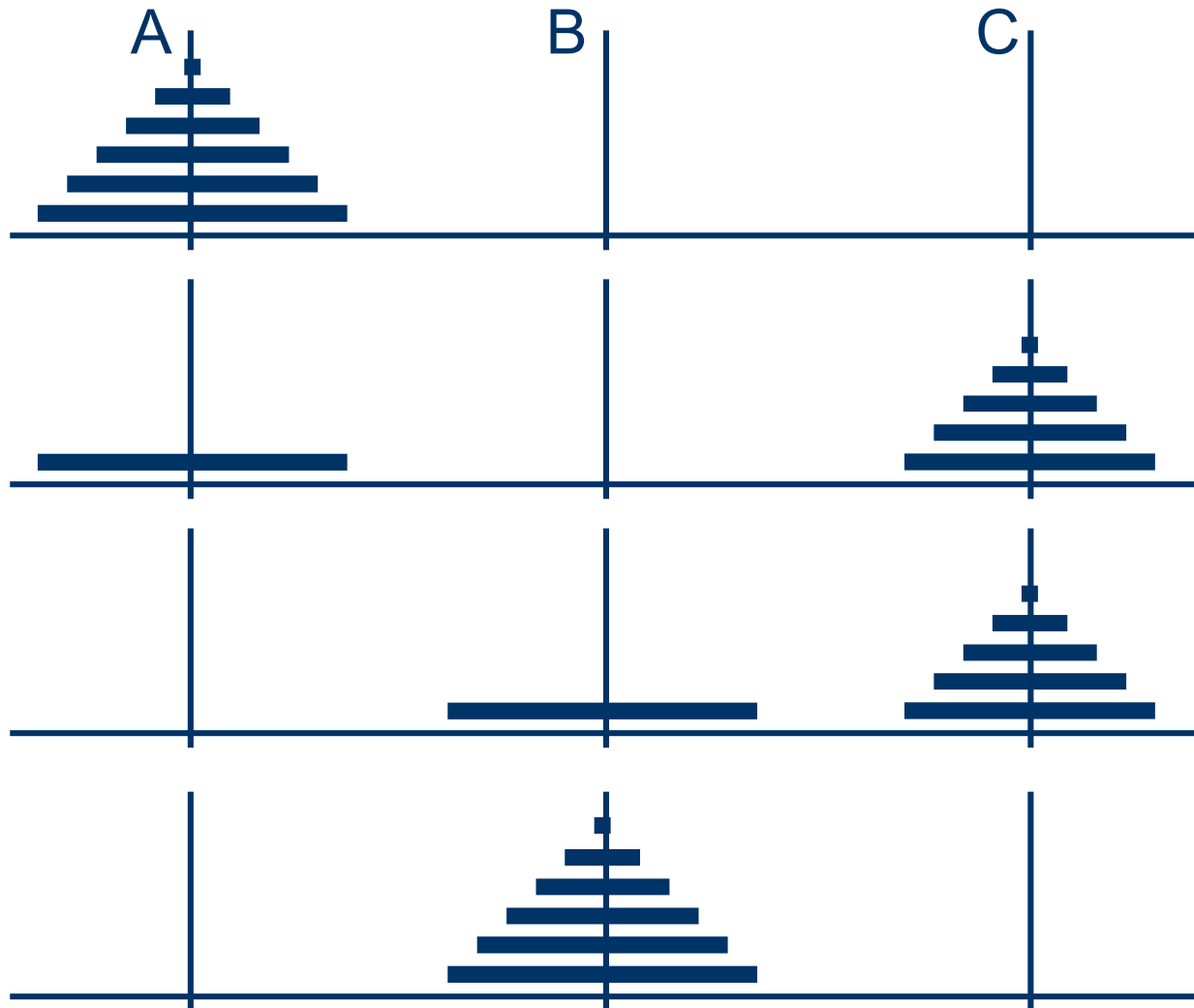
<http://www.youtube.com/watch?v=D2X46zBND6g>



<http://www.youtube.com/watch?v=UMPneeBzQHk>

Algorithmenentwurf (I)

- Anwendung des Induktionsprinzips



$k - 1$ Scheiben
von A nach C

1 Scheibe
von A nach B

$k - 1$ Scheiben
von C nach B

Algorithmenentwurf (II)

- Anwendung des Induktionsprinzips
 - Basisfall $k = 0$: keine Scheiben; trivial, weil sofort fertig
- Rückführung auf kleinere Probleme ($k > 0$)
 - Teilproblem 1: versetze die oberen $k - 1$ Scheiben von A nach C unter Verwendung von B als Zwischenablage; kleineres Problem: rekursiv lösbar
 - Teilproblem 2: versetze die verbleibende k -te Scheibe von A nach B
 - Teilproblem 3: Versetze die $k - 1$ Scheiben von C nach B unter Verwendung von A als Zwischenablage; kleineres Problem: rekursiv lösbar
 - Verknüpfung: Hintereinanderausführung der 3 Teillösungen

Induktionsannahme:
das Problem ist für die
Versetzung von $k - 1$
Scheiben lösbar

琐碎的
triviale Elementar-
operation

Induktionsannahme:
das Problem ist für die
Versetzung von $k - 1$
Scheiben lösbar

Java-Methode hanoi

entspricht k

```
static void hanoi(int scheiben,  
                  char start, char ziel, char hilfe) {  
    if (scheiben > 0) {  
        hanoi(scheiben - 1, start, hilfe, ziel);  
        System.out.println("Versetze Scheibe " + scheiben +  
                           " von " + start + " nach " + ziel);  
        hanoi(scheiben - 1, hilfe, ziel, start);  
    }  
    // else Fall = Basisfall der Induktion ist leer  
}
```

- Rekursion nicht nur am Ende des Rumpfs
- Fall k wird auf den Fall $k - 1$ zurückgeführt (Induktionsannahme wird doppelt angewendet)

Methodenaufruf `hanoi(3, 'A', 'B', 'C');`

```
> java Hanoi
```

```
Versetze Scheibe 1 von A nach B
```

```
Versetze Scheibe 2 von A nach C
```

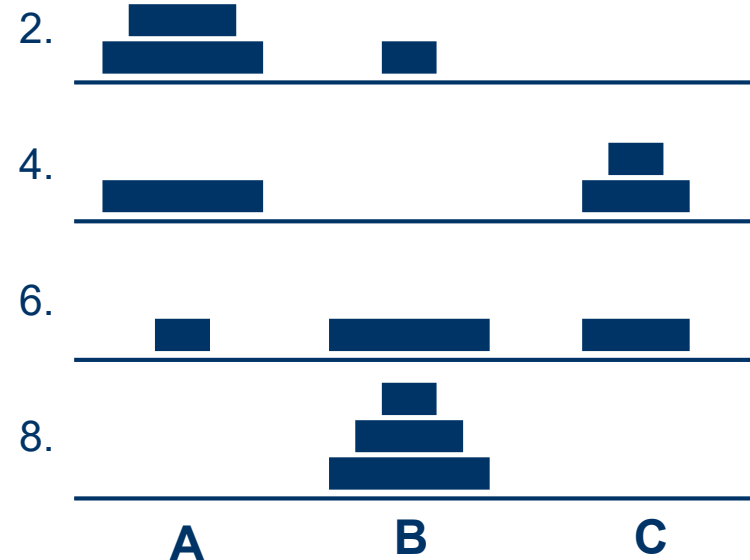
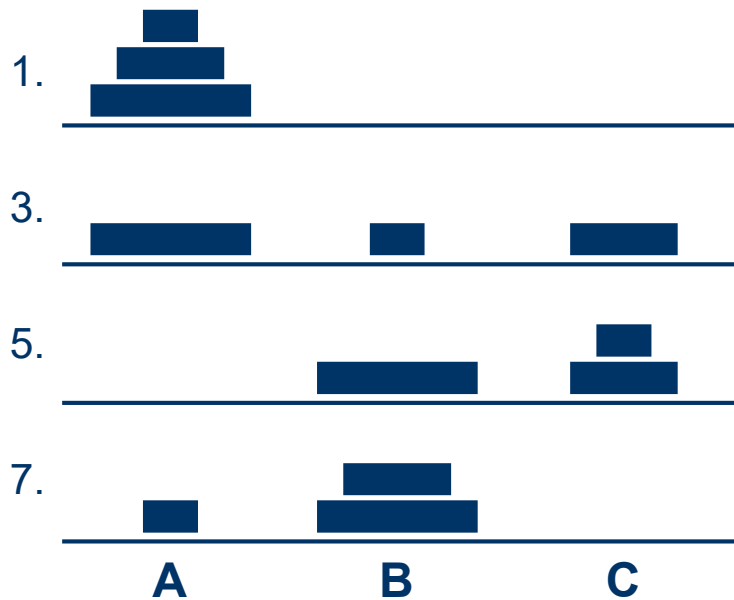
```
Versetze Scheibe 1 von B nach C // zwei Scheiben A → C
```

```
Versetze Scheibe 3 von A nach B // größte Scheibe am Ziel
```

```
Versetze Scheibe 1 von C nach A
```

```
Versetze Scheibe 2 von C nach B // zweitgrößte Scheibe am Ziel
```

```
Versetze Scheibe 1 von A nach B // fertig
```



Korrektheit und Terminierung von `hanoi`

- *Korrektheit:*

Beweis per Induktion (kann direkt aus Rekursion abgelesen werden)

- *Terminierung:*

Terminierungsfunktion $T(x) = x$ (Anzahl zu versetzender Scheiben)

- streng monoton fallend (nachweisbar per Induktionsbeweis)
- nach unten durch 0 beschränkt

Kaskadenartige Rekursion

- Eine Methode `f` heißt **kaskadenartig rekursiv**, wenn in einem Zweig einer Fallunterscheidung im Rumpf von `f` zwei oder mehr Aufrufe von `f` auftreten.
- Kaskadenartig rekursive Methoden sind gekennzeichnet durch eine baumartige Aufrufstruktur und ein lawinenartiges Anwachsen der anfallenden rekursiven Aufrufe.
- Kaskadenartige Rekursion lässt sich nicht (leicht) in Iterationen umwandeln.
- Beispiele:
 - Die Methode `hanoi` ist kaskadenartig rekursiv.
 - Das nachfolgend betrachtete Beispiel `fibonacci` ist ebenfalls kaskadenartig rekursiv.

Fibonacci-Zahlen (I)

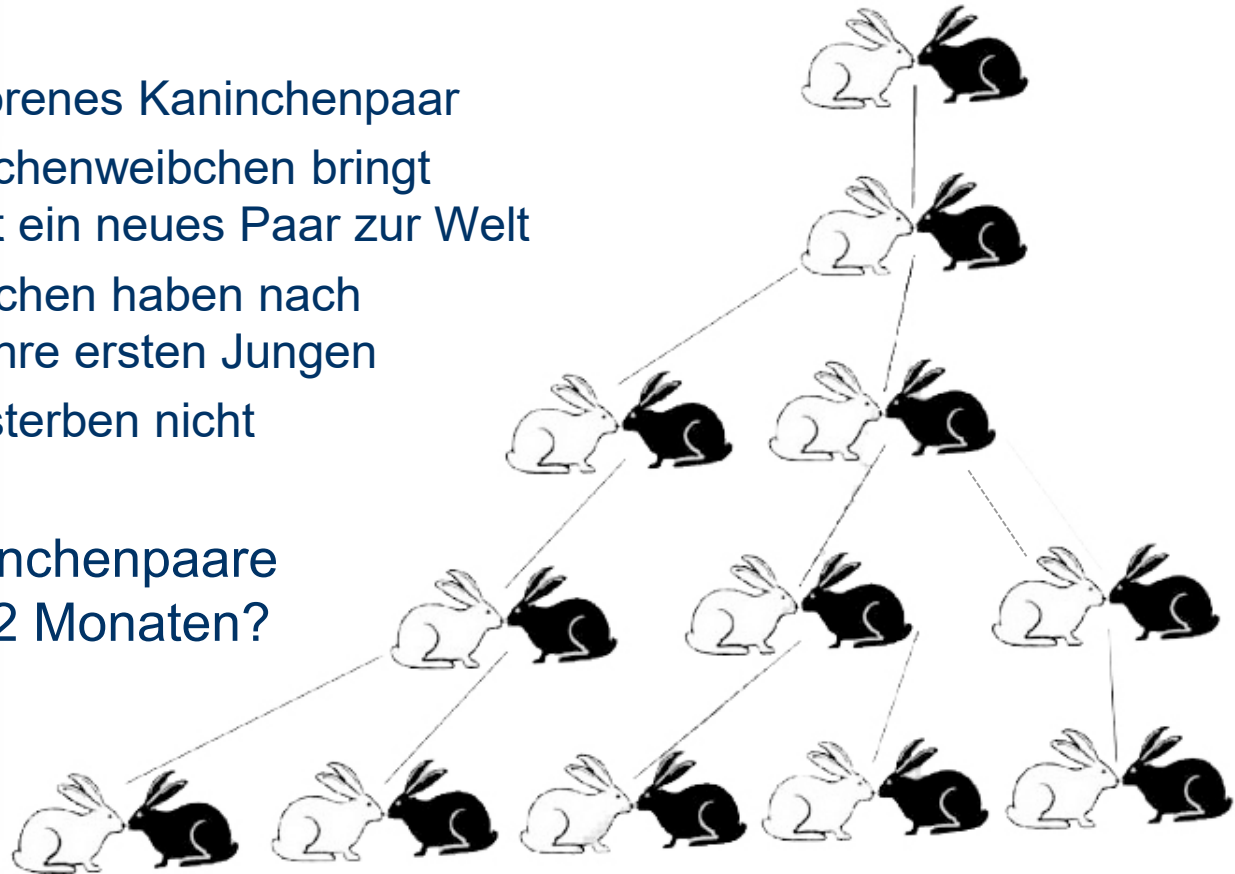
- ursprüngliche Fragestellung aus dem Jahr 1202:

- gegeben:

- ein neugeborenes Kaninchenpaar
 - jedes Kaninchenweibchen bringt jeden Monat ein neues Paar zur Welt
 - junge Kaninchen haben nach 2 Monaten ihre ersten Jungen
 - Kaninchen sterben nicht

- gesucht:

Wie viele Kaninchenpaare gibt es nach 12 Monaten?



Fibonacci-Zahlen (II)

Monat 1:

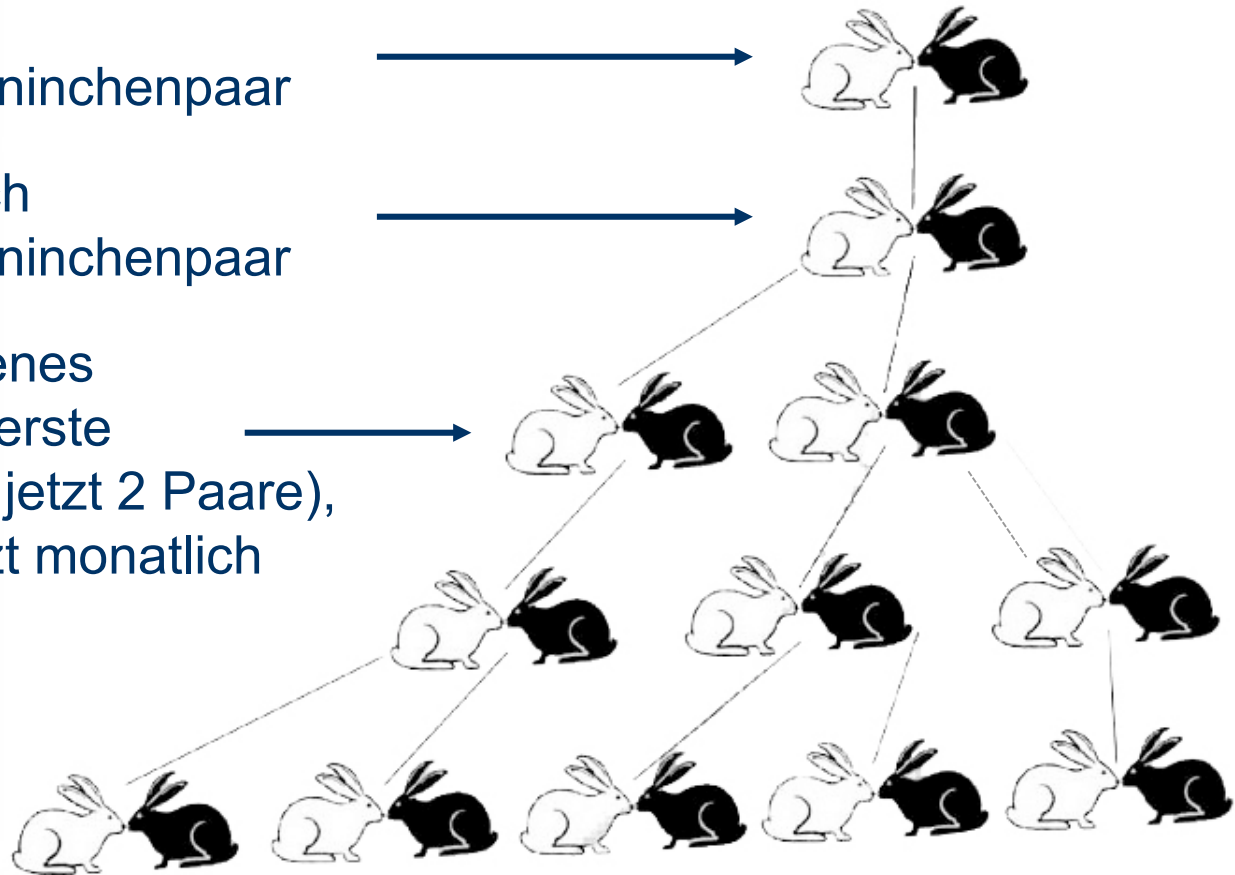
1 neugeborenes Kaninchenpaar

Monat 2: immer noch

1 neugeborenes Kaninchenpaar

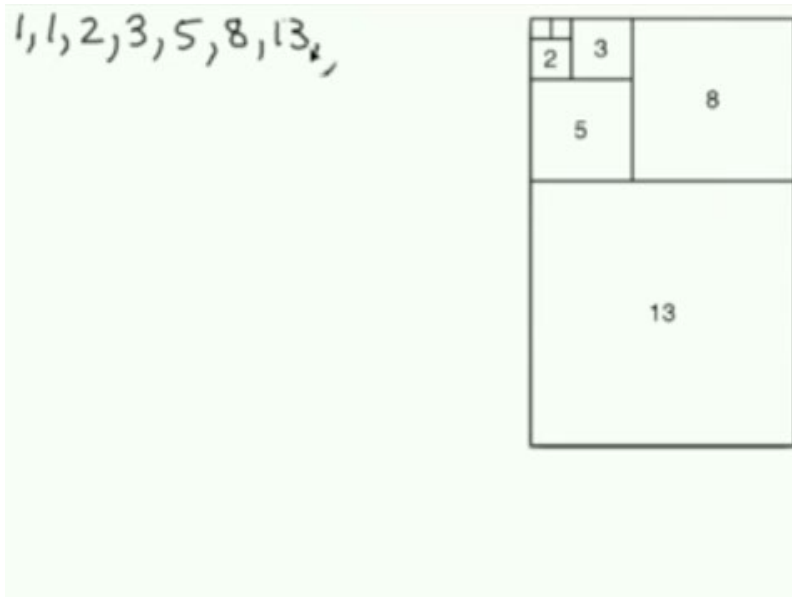
Monat 3: neugeborenes
Kaninchenpaar hat erste
Nachkommen (also jetzt 2 Paare),
Eltern haben ab jetzt monatlich
Nachwuchs usw.

...



führt zu folgender Zahlenfolge: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
(sog. **Fibonacci-Zahlen**) mit großer Bedeutung in der Informatik

Videos zu den Fibonacci-Zahlen (Auswahl)



<http://www.youtube.com/watch?v=c4Rgel0Et6s>



<http://www.youtube.com/watch?v=KpBfbzxS1l4>

Entwurf mittels Induktionsprinzip

- Basisfall:

- $n = 1: \text{fibonacci}(1) = 1$
- $n = 2: \text{fibonacci}(2) = 1$

fibonacci
ist (in AuD) nur
für $n \geq 1$ definiert.

- Rückführung auf kleinere Probleme:

- Teilproblem 1: $\text{fibonacci}(n - 2)$
(zweimonatige Kaninchen haben erste Nachkommen)
- Teilproblem 2: $\text{fibonacci}(n - 1)$
(Kaninchen aus dem vergangenem Monat bleiben da)
- Verknüpfung: Addition

- der Fall n wird also auf zwei kleinere Fälle (kaskadenartige Rekursion) und zwei Basisfälle zurückgeführt

- also (rekursive Definition):

$$\begin{aligned} \text{fibonacci}(1) &= 1 \text{ und } \text{fibonacci}(2) = 1, \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 2) + \text{fibonacci}(n - 1) \end{aligned}$$

Implementierung in Java

- Variante 1: mit **if-else**-Anweisung

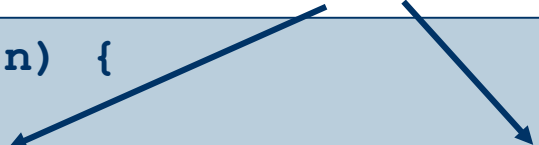
```
static int fibonacci1(int n) {  
    if (n <= 2) {  
        return 1;  
    } else {  
        return fibonacci1(n - 2) + fibonacci1(n - 1);  
    }  
}
```

kaskadenartige
Rekursion



- Variante 2: mit **?-Operator**

```
static int fibonacci2(int n) {  
    return (n <= 2) ? 1  
        : fibonacci2(n - 2) + fibonacci2(n - 1);  
}
```



4.1 Grundbegriffe

4.2 Lineare Rekursion und Endrekursion

4.3 Kaskadenartige Rekursion

→ 4.4 Verschränkte und verschachtelte Rekursion

纠缠和嵌套的递归

Verschränkte (auch: wechselseitige) Rekursion

- Zwei Methoden f und g heißen **verschränkt** (auch: **wechselseitig**) **rekursiv**, wenn die Methodendeklaration von f einen Aufruf der Methode g enthält und die Methodendeklaration von g einen Aufruf der Methode f enthält.
- Verschränkte Rekursion liegt auch vor, wenn sich mehr als zwei Methoden zyklisch gegenseitig aufrufen.
- Beispiel:
 - die Methoden **istGerade** und **istUngerade** sind verschränkt rekursiv.

Test, ob ganze Zahl n gerade bzw. ungerade ist

偶数

奇数

```
static boolean istGerade(int n) {  
    if (n == 0) {  
        return true;  
    } else {  
        return istUngerade(n - 1);  
    }  
}  
  
static boolean istUngerade(int n) {  
    if (n == 0) {  
        return false;  
    } else {  
        return istGerade(n - 1);  
    }  
}
```

Beispiel:

`istGerade(3)`

`istUngerade(2)`

`istGerade(1)`

`istUngerade(0)`

`false`

Verschachtelte Rekursion

- Eine Methode f heißt **verschachtelt rekursiv**, wenn die rekursiven Aufrufe von f nicht nur im Rumpf der Methodendeklaration von f sondern zusätzlich auch in einem Argumentausdruck eines rekursiven Aufrufs von f auftreten.
- Bei verschachtelt rekursiven Methoden sind Terminierungsbeweise i. d. R. schwer.
- Verschachtelt rekursive Methoden sind damit in der Praxis eher unbedeutend.
- Beispiel:
 - Die **ackermann**-Funktion ist verschachtelt rekursiv; sie ist in der Informatik von großer theoretischer Bedeutung.

Ackermann-Funktion

```
static int ackermann(int m, int n) {  
    if (m == 0) {  
        return n + 1;  
    } else {  
        if (n == 0) {  
            return ackermann(m - 1, 1);  
        } else {  
            return ackermann(m - 1, ackermann(m, n - 1));  
        }  
    }  
}
```

rekursiver Aufruf als
Argument eines
rekursiven Aufrufs



- Diese Funktion werden Sie in Lehrveranstaltungen zur theoretischen Informatik noch intensiv studieren (Stichwort: Berechenbarkeit):
 - Sie terminiert immer.
 - Ihre Werte wachsen jedoch bei Vergrößerung der Argumente außerordentlich schnell:

ackermann(3, 3) = 61

ackermann(2, 4) $\approx 10^{21000}$

ackermann(4, 4) $\approx 10^{10^{10^{21000}}}$