



11. Rekursion, Komplexität von Algorithmen

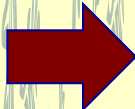
Teil 2

Java-Beispiele:

Power1.java

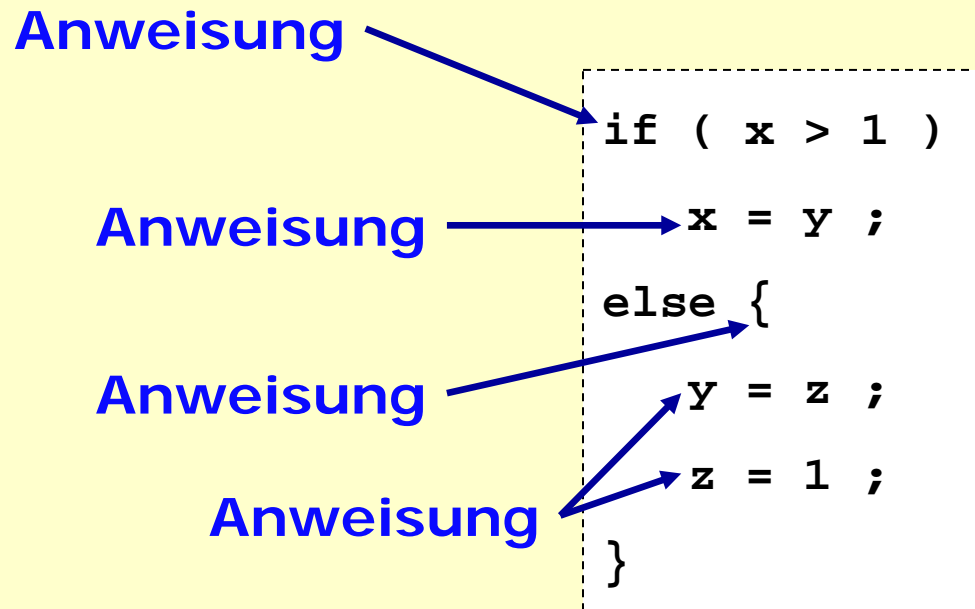
Hanoi.java

Anwendung der Rekursion

- Rekursiv definierte Funktionen
 - **Fibonacci-Funktion**
 - **Fakultät, Potenz**
 - ...
-  Rekursiver Aufbau der zu verarbeitenden Daten
 - **Programme (EBNF) -> Compiler**
 - **Bäume und Listen (Teil III)**
- Natürliche rekursive Problemlösungen
 - **Sortierverfahren**
 - **Türme von Hanoi**
 - ...

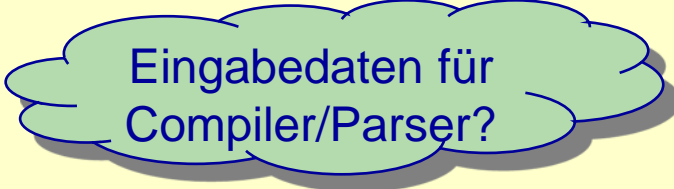
Rekursiver Aufbau von Daten: Syntax von Programmiersprachen

Programmstruktur: rekursiv definiert



Beispiel: rekursiver Aufbau der Daten

Compiler:
Parser (Syntaxanalyse)



Eingabedaten für
Compiler/Parser?

EBNF:

```
Anweisung ::=      Auswahlanweisung | Block | ...
```

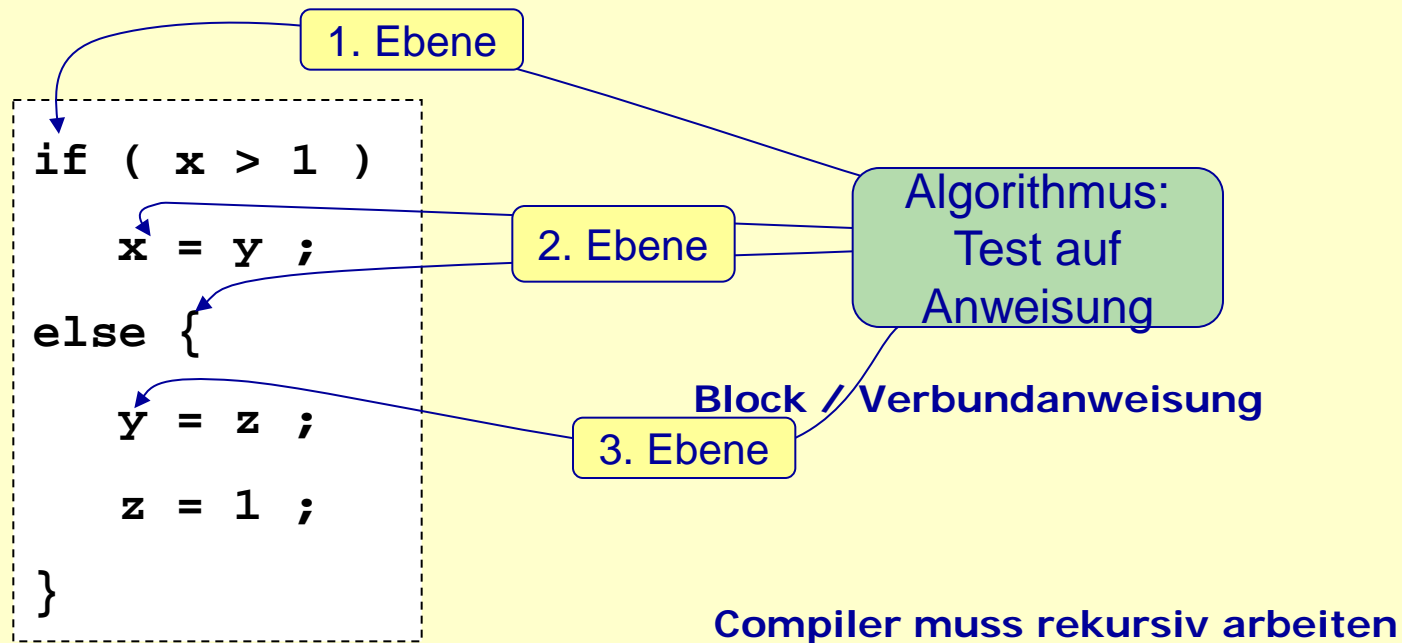
```
Auswahlanweisung ::=      if ( Ausdruck ) Anweisung  
                           [ else Anweisung ]
```

```
Block ::=      { Anweisung | ... }
```

Parser:

1. testet, ob Anweisung vorliegt
2. findet if-Anweisung
3. testet im Innern der if-Anweisung:
liegen 1 (2) innere Anweisungen vor
→ **rekursiver Aufruf des Tests, ob *Anweisung* anliegt**

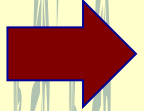
Parser : testet in jedem Zustand, ob bestimmte Syntaxeinheit anliegt



- | | |
|------------------|---|
| <u>1. Ebene:</u> | Einstieg in den Algorithmus |
| <u>2. Ebene:</u> | 1. rekursiver Aufruf (zwei mal) |
| <u>3. Ebene:</u> | 2. rekursiver Aufruf (in „Block“, zwei mal) |

Einzelheiten: Compilerbau

Anwendung der Rekursion

- Rekursiv definierte Funktionen
 - **Fibonacci-Funktion**
 - **Fakultät, Potenz**
 - ...
- Rekursiver Aufbau der zu verarbeitenden Daten
 - **Programme (EBNF)-> Compiler**
 - **Bäume und Listen (Teil III)**
-  • Natürliche rekursive Problemlösungen
 - **Sortierverfahren (II.12)**
 - **Türme von Hanoi**
 - ...

Rekursive Problemlösungen: Türme von Hanoi



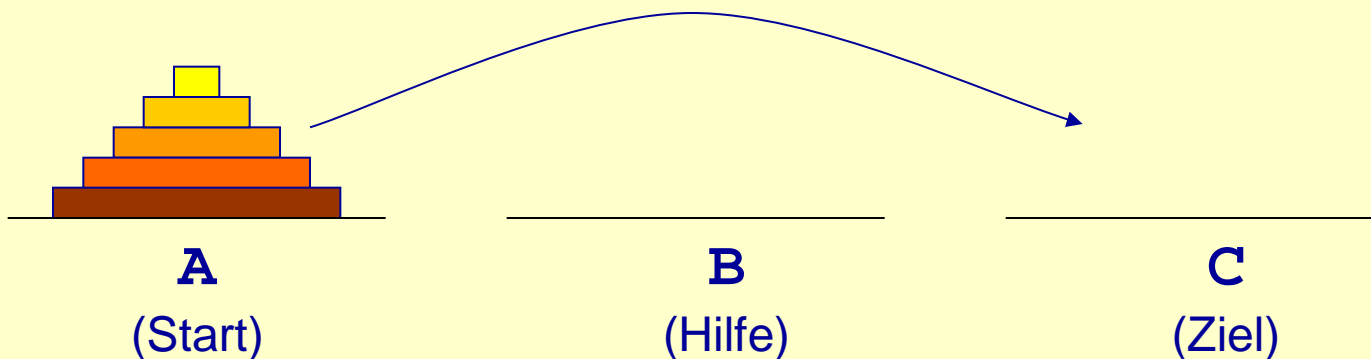
Rekursive Problemlösungen: Türme von Hanoi

Aufgabe:

Scheiben liegen der Größe nach geordnet auf einem Platz A und sollen auf einen Platz C unter Zuhilfenahme eines Platzes B transportiert werden.

Randbedingungen:

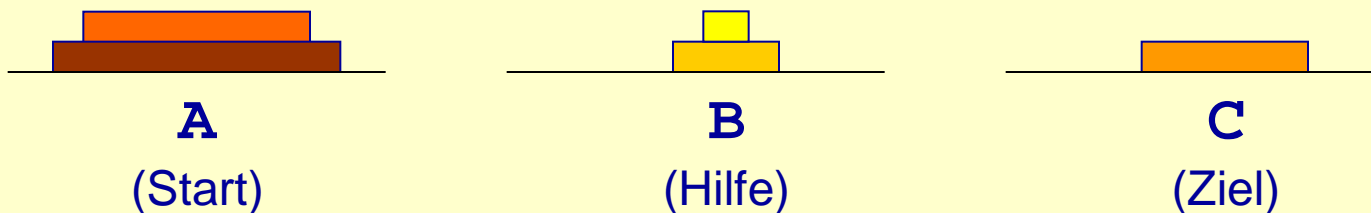
- immer nur eine Scheibe bewegen (die obere)
- niemals größere über einer kleineren Scheibe



Aufgabe (Fortsetzung)

Gesucht: Folge von Einzelbewegungen,
die zum Ziel führen

```
% java Hanoi  
Anzahl der Scheiben: 5  
Scheibenbewegungen:  
von A nach C  
von A nach B  
von C nach B  
von A nach C  
...
```



Lösungsalgorithmus: iterativ oder rekursiv?

Gesucht: **Folge** von Einzelbewegungen, die zum Ziel führen.

d. h. Algorithmus muss für wiederholte Berechnung von Einzelschritten sorgen.

→ Iterativer Algorithmus?

Möglich - aber: nicht naheliegend

<p>Solange (noch nicht am Ziel) Berechne nächste Bewegung</p>

→ Rekursiver Algorithmus?

Natürliche Lösung des Problems:

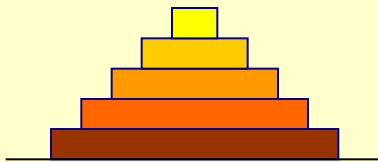
Zerlegung des Problems in einfachere

Teilprobleme, die rekursiv bearbeitet werden

bewege n Scheiben → bewege n-1 Scheiben

Rekursiver Lösungsalgorithmus

0



A
(Start)



B
(Hilfe)

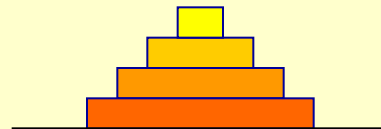


C
(Ziel)

1



A
(Start)

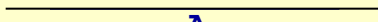


B
(Hilfe)

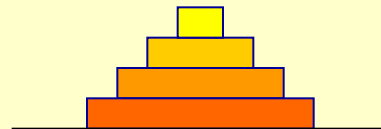


C
(Ziel)

2



A
(Start)



B
(Hilfe)



C
(Ziel)

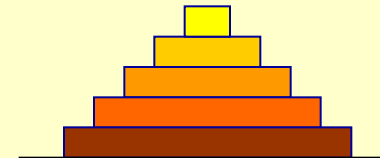
3



A
(Start)



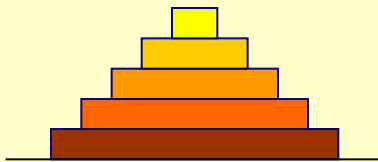
B
(Hilfe)



C
(Ziel)

Rekursiver Lösungsalgorithmus

0



A
(Start)



B
(Hilfe)



C
(Ziel)

Problem:
Bewege
n Scheiben

zurückgeführt
auf 3 Teilprobleme:
Bewege
n-1 Scheiben

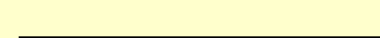
1



A
(Start)

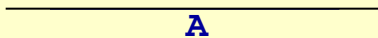


B
(Hilfe)



C
(Ziel)

2



A
(Start)



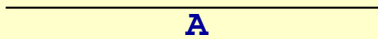
B
(Hilfe)



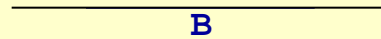
C
(Ziel)

Bewege
1 Scheibe

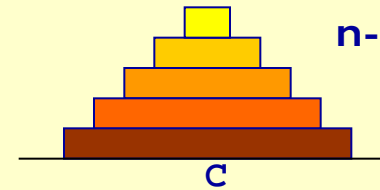
3



A
(Start)



B
(Hilfe)



C
(Ziel)

Bewege
n-1 Scheiben

Rekursiver Lösungsalgorithmus

Bewege n Scheiben von 'start' über 'hilfe' nach 'ziel'

Anfang: $n = 1$ (eine Scheibe)

Transportiere die Scheibe direkt von 'start' nach 'ziel'

Schritt: $n > 1$

1. Bewege $n - 1$ Scheiben^{*)} von 'start' über 'ziel' nach 'hilfe'
2. Transportiere eine (größte) Scheibe direkt von 'start' nach 'ziel'
3. Bewege $n - 1$ Scheiben^{*)} von 'hilfe' über 'start' nach 'ziel'

^{*)} „Bewege“: Kein gleichzeitiger Transport von $n - 1 > 1$ Scheiben!
Sondern: Anwendung des Algorithmus auf weniger als n Scheiben
(Rekursivität)

Hanoi-Programm: rekursiver Algorithmus

Hanoi.java

```
static void bewege
    (int n, char start, char hilfe, char ziel) {

    if (n == 1)
        System.out.println(" von " + start + " nach " + ziel);
    else {
        bewege(n - 1, start, ziel, hilfe);
        System.out.println(" von " + start + " nach " + ziel);
        bewege(n - 1, hilfe, start, ziel);
    }
}
```

Gesamtproblem →
3 Teilprobleme (mit 2 rekursiven Aufrufen)

Hanoi-Programm: Rahmen

Hanoi.java

```
public static void main (String argv[]) {
    int n;

    System.out.print("Anzahl der Scheiben: ");
    n = Keyboard.readInt();
    if (n > 0) {
        . . .
        bewege(n, 'A', 'B', 'C');
    }
    else
        System.out.println("Zahl nicht positiv");
}
```

Plätze durch Zeichen
bezeichnet

Hanoi-Programm: Rahmen

```
public static void main (String argv[]) {
    int n;

    System.out.print("Anzahl der Scheiben: ");
    n = Keyboard.readInt();
    if (n > 0) {
        . . .
        bewege(n, 'A', 'B', 'C');
    }
    else
        System.out.println("Zahl nicht positiv");
}
```

```
% java Hanoi
Anzahl der Scheiben: 5
Scheibenbewegungen:
von A nach C
von A nach B
von C nach B
von A nach C
. . .
```

```
% java Hanoi
Anzahl der Scheiben: 10
Scheibenbewegungen:
von A nach C
1023 Bewegungen .
```

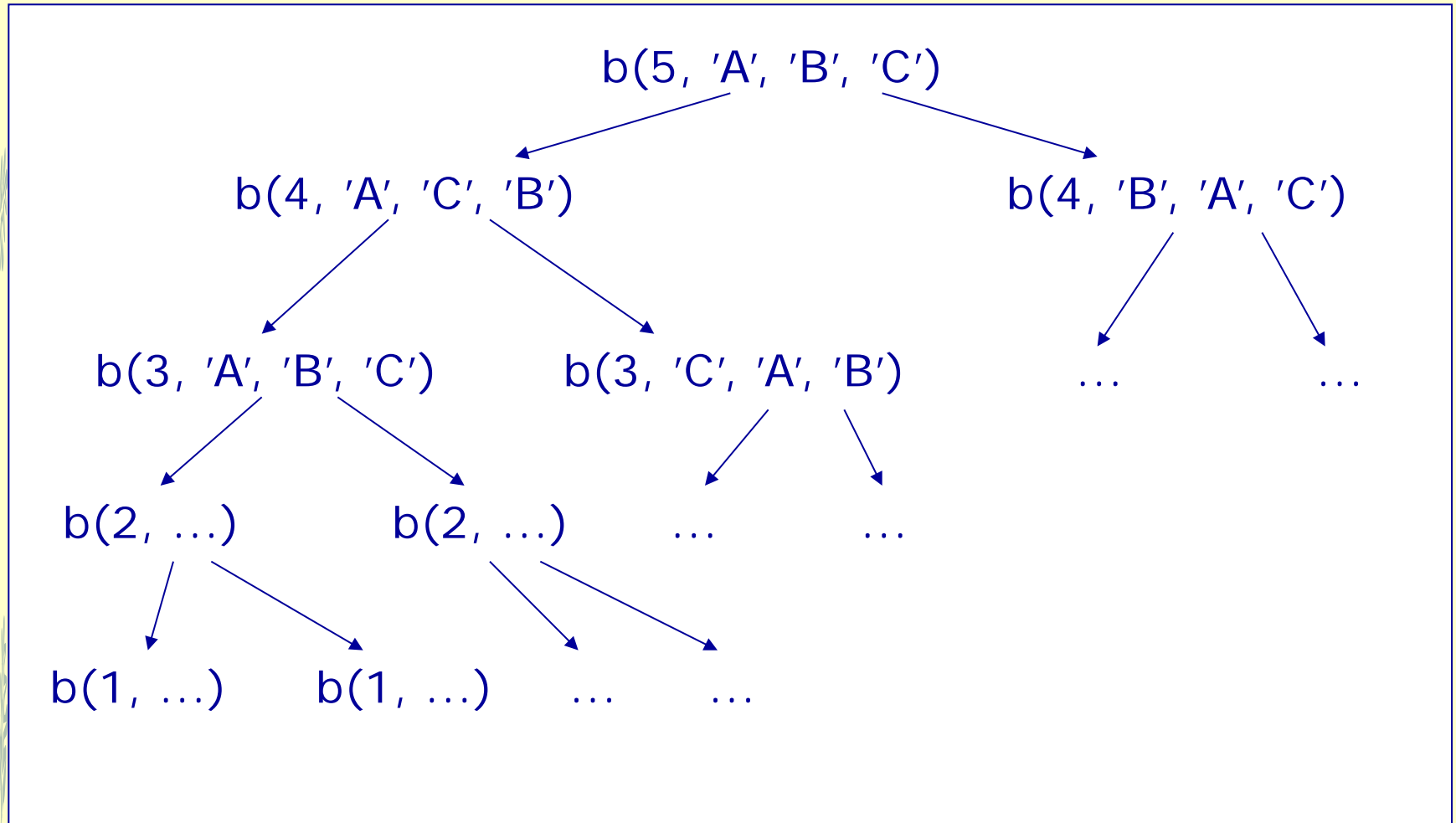
```
% java Hanoi
Anzahl der Scheiben: 100
ca. 10 Mrd.
Scheibenbewegungen:
. . .
```

Hanoi-Programm: Visualisierung

http://www.mathematik.ch/spiele/hanoi_mit_grafik/

Aufrufbeziehungen: Hanoi

Abk.: $\text{bewege}(5, 'A', 'B', 'C') = \text{b}(5, 'A', 'B', 'C')$



Überführung: Rekursion → Iteration



Warum?

- ▶ Rekursion: zeit- und speicheraufwendig (Aufrufe von Methoden)
- ▶ Es gibt Programmiersprachen ohne Rekursion

Hanoi: rekursiv → iterativ

(mit Abspeicherung von zu lösenden Problemen: Stack)

Grundprinzip:

Wie?

- In einem (Zyklus-)Schritt:
Löse immer **das aktuelle** Problem und merke die später zu lösenden Probleme in einem Problemspeicher (Stack)
- Ein Problem:
Bewege n Scheiben von 'start' über 'hilfe' nach 'ziel'
- Im Problemspeicher (Stack / Stapel): das aktuelle Problem ist 'oben' gespeichert, die anderen Probleme liegen „darunter“

• Algorithmus:

Solange noch Probleme zu lösen sind

```
while (!isEmpty(problemStack)) {  
    // bearbeite das aktuelle (oberste) Problem  
}
```

$n = 1 \rightarrow$ drucken und
Problem streichen

$n > 1 \rightarrow$ Problem streichen + speichere
drei neue Probleme

Beispiel: 'Problem-Stack'

```
bewege (5, 'A', 'B', 'C');
```

Stackentwicklung (Speicher aller zu lösenden Probleme):

Anfang:

5 A B C

← aktuelles Problem

1. Zyklusschritt:

4	A	C	B
1	A	B	C
4	B	A	C

aktuelles Problem

alle zu lösenden Probleme

```
while (!isEmpty(problemStack)) {  
    // bearbeite das aktuelle (oberste) Problem  
}
```

$n = 1 \rightarrow$ drucken und Problem streichen

$n > 1 \rightarrow$ Problem streichen + speichere drei neue Probleme

Beispiel: 'Problem-Stack'

bewege (5, 'A', 'B', 'C');

Stackentwicklung:

2	A	C	B
1	A	B	C
2	B	A	C
1	A	C	B
3	C	A	B
1	A	B	C
4	B	A	C

1	A	B	C
1	A	C	B
1	C	A	B
1	A	B	C
2	B	A	C
1	A	C	B
3	C	A	B
1	A	B	C
4	B	A	C

Anfang:

5	A	B	C
---	---	---	---

3. Zyklusschritt:

4	A	C	B
1	A	B	C
4	B	A	C

1. Zyklusschritt:

3	A	B	C
1	A	C	B
3	C	A	B
1	A	B	C
4	B	A	C

4. Zyklusschritt:

2. Zyklusschritt:

Weitere Zyklusschritte?

Beispiel: 'Problem-Stack'

bewege (5, 'A', 'B', 'C');

Stackentwicklung:

Anfang:

5 A B C

3. Zyklusschritt:

4 A C B
1 A B C
4 B A C

1. Zyklusschritt:

3 A B C
1 A C B
3 C A B
1 A B C
4 B A C

2. Zyklusschritt:

4. Zyklusschritt:

5. 6. 7. 8. Zyklusschritt:
4 * drucken und
4 Probleme streichen:

2 A C B
1 A B C
2 B A C
1 A C B
3 C A B
1 A B C
4 B A C

~~1 A B C~~
~~1 A C B~~
~~1 C A B~~
~~1 A B C~~

2 B A C
1 A C B
3 C A B
1 A B C
4 B A C

Modifiziertes Problem: Türme von Hanoi

Modifizierte Aufgabe:

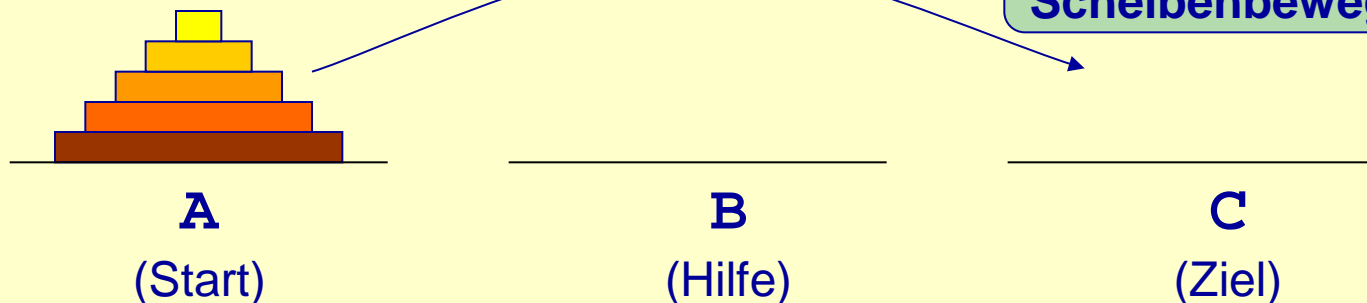
Scheiben liegen der Größe nach geordnet auf einem Platz A und sollen auf einen Platz C unter Zuhilfenahme eines Platzes B transportiert werden.

Randbedingungen:

- immer nur eine Scheibe bewegen (die obere)
- niemals größere über einer kleineren Scheibe
- Transport nur zwischen Nachbarplätzen, d.h. verboten $A \rightarrow C$, $C \rightarrow A$

Idee?

Anzahl der
Scheibenbewegungen?



Wertung: Rekursion

- Alternative *iterative* Lösung immer möglich
 - weil alles in Maschinensprache ablaufen muss – ohne Rekursion
 - Idee für iterative Lösung: natürliche rekursive Lösung
- Iterative Variante: oft schneller (Methodenaufrufe: zeitintensiv)
- Rekursive Lösung: oft lesbarer, eleganter – aber zeit- und speicheraufwendig
- Rekursion: nicht um jeden Preis

Nur wenn Lesbarkeit erhöht, anwenden auf :

- rekursive Funktionen
- rekursive Daten
- rekursive Problemlösungen

→ Rekursive Lösung dort meist die einzige sinnvolle!

Komplexität von Algorithmen: Rechenaufwand

Nicht: Komplexität von Programmen

Unterschied?

- **Programm: Größe/Länge des Programms**
- **Algorithmus: Laufzeit/Berechnungsaufwand**

Komplexitätsbetrachtungen von Algorithmen – Warum?

Oft wichtig:

- Vor Entwicklung von Programmen:
 - Lohnt sich die Entwicklung eines Programms überhaupt: rechnet 100000 Jahre
- Vor Nutzung von Programmen:
 - Welche Eingaben verkraftet das Programm bzgl. der Laufzeit?

Beispiel: Welche Eingaben verkraftet das Hanoi-Programm?

```
% java Hanoi  
Anzahl der Scheiben: 1000  
Scheibenbewegungen:  
  von A nach C  
  . . .
```

Wie viele Ausgaben
(welche Zeit)?

Scheiben	1	5	30	1000
Bewegungen	1	31	~ 1 Mrd.	~ 10^{100}

Zahl mit 100
Nullen

Dauer aller Scheibenbewegungen

Bei einer Scheibe pro Sekunde

Anzahl Scheiben	Benötigte Zeit
5	31 Sekunden
10	17,1 Minuten
20	12 Tage
30	34 Jahre
40	348 Jahrhunderte
60	36,6 Milliarden Jahre
64	585 Milliarden Jahre

**Falls schneller Rechner:
1 Mia Bewegungen pro Sekunde
→585 Jahre für 64 Scheiben!
→ 1000 Scheiben? 😊**

Komplexitätsbetrachtungen: Hanoi

n	1	2	6	10	1000
Anzahl Bewegungen	1	3	63	1023	$\sim 10^{100}$

Allgemein: $\text{anz}(n) = 2^n - 1$

Beweis: Vollständige Induktion

Anfang: $\text{anz}(1) = 2 - 1 = 1$ (gilt nach dem Algorithmus)

Schritt:

$$\begin{aligned}\text{anz}(n+1) &= 2^{(n+1)} - 1 \\ &= 2 * 2^n - 1 \\ &= 2 * (2^n - 1) + 1 \\ &= 2 * \text{anz}(n) + 1 \quad (\text{nach Vor.})\end{aligned}$$

→ Damit gilt:

für (n+1) wird die bisher ermittelte Anzahl für n verdoppelt + 1

→ Gilt nach: Hanoi-Algorithmus -> siehe Beschreibung des Algorithmus

Komplexitätsbetrachtungen: Hanoi

n	1	2	6	10	1000
Anzahl Bewegungen	1	3	63	1023	$\sim 10^{100}$

Allgemein: $\text{anz}(n) = 2^n - 1$

Beweis: Vollständige Induktion

Anfang: $\text{anz}(1) = 2 - 1 = 1$ (gilt nach dem Algorithmus)

Schritt: $\text{anz}(n+1) = 2^{(n+1)} - 1$
 $= 2 * 2^n - 1$
 $= 2 * (2^n - 1) + 1$
 $= 2 * \text{anz}(n) + 1$

```
bewege(n - 1, start, ziel, hilfe);  
Transportiere eine Scheibe(start " nach " ziel);  
bewege(n - 1, hilfe, start, ziel);
```

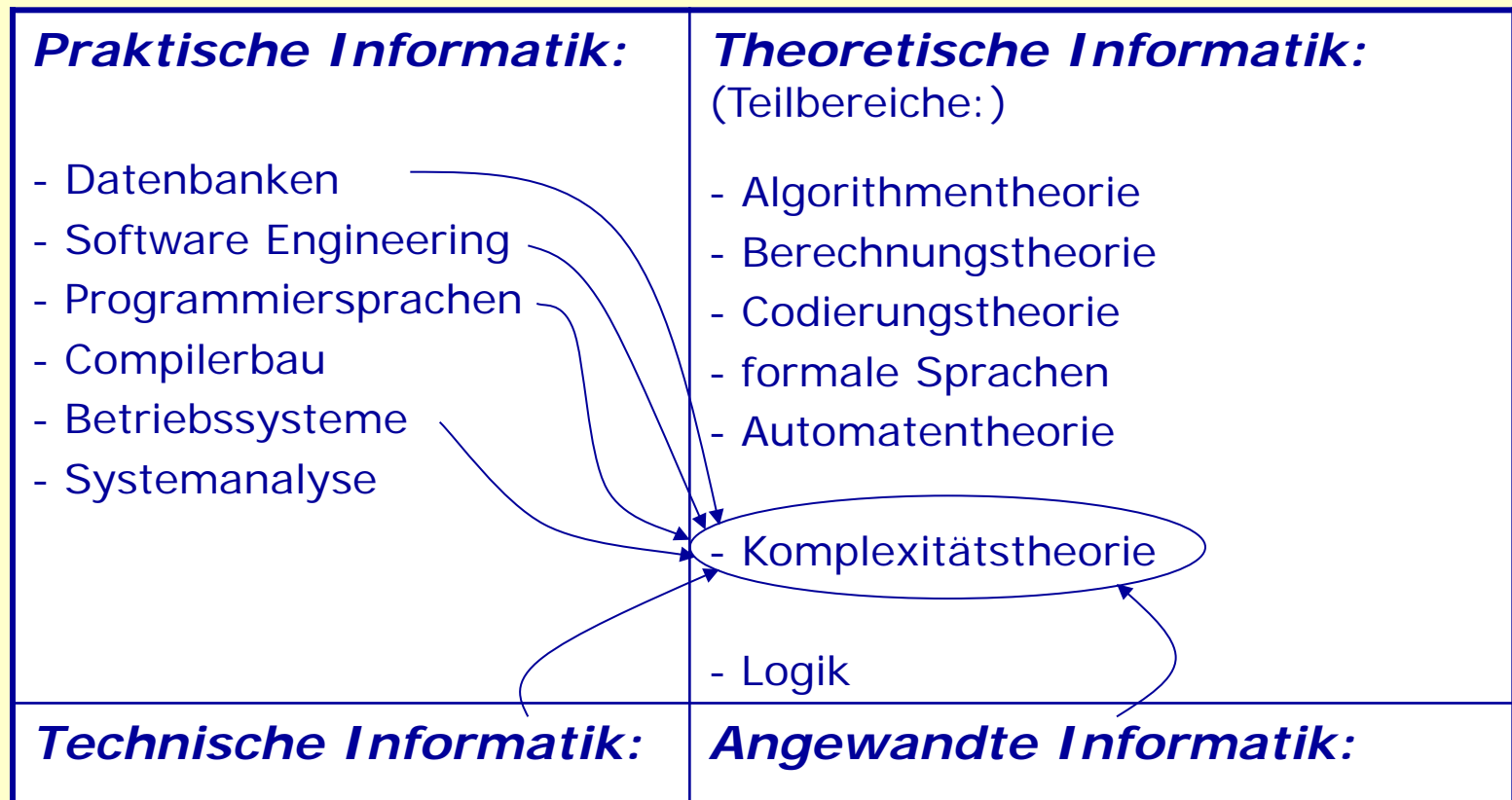
→ Damit gilt:

für (n+1) wird die bisher ermittelte Anzahl für n verdoppelt + 1

→ Gilt nach: Hanoi-Algorithmus -> siehe Beschreibung des Algorithmus

Teilgebiete der Informatik – und ihre Beziehungen

**Komplexitätstheorie:
Theorie des Berechnungsaufwands von Algorithmen**



Komplexität von Algorithmen

Zusammenhang zwischen

Größe der Eingabe $n \rightarrow$ Berechnungsaufwand $O(n)$

Beispiele:

- Exponent n bei 'power'

$n \rightarrow n$ (power)
 $n \rightarrow \log_2 n$ (power1)

Multiplikationen

- Anzahl n der Scheiben bei Hanoi

$n \rightarrow 2^n - 1$

Ausgabeoperationen
(Scheibenbewegungen)

- Anzahl n der Elemente beim Sortieren

Anzahl: Vergleiche, Vertauschungen

Komplexitätsklassen von Algorithmen

Algorithmen heißen ... , falls der Zusammenhang zwischen n und dem Berechnungsaufwand O in dem angegebenen Verhältnis steht.

konstant: $O(n) = \text{Konstante } k$

logarithmisch:
(power1) $O(n) = k * \log_2 n$

linear:
(power) $O(n) = k * n$

$n \log_2 n$: $O(n) = k * n \log_2 n$

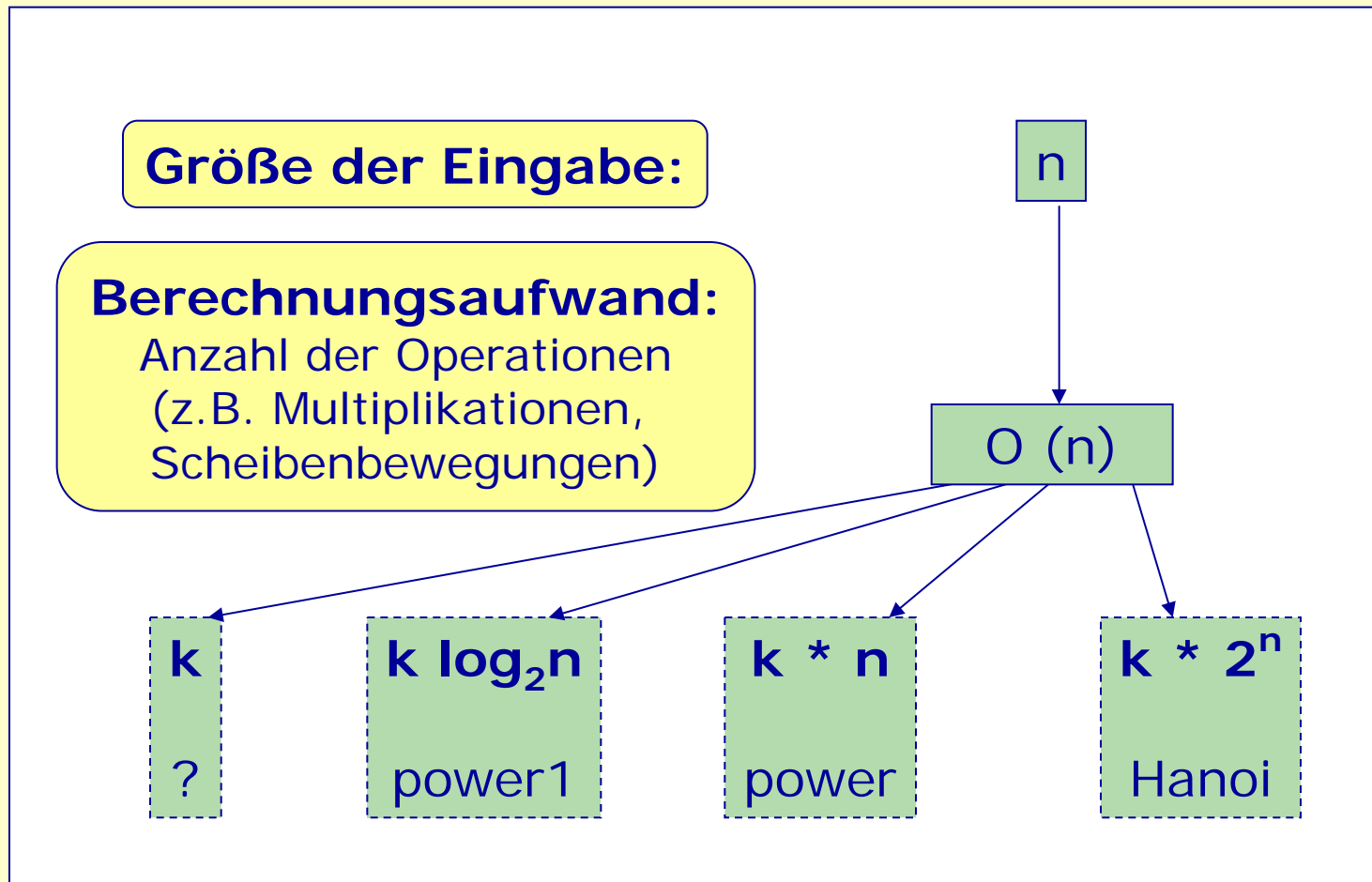
quadratisch: $O(n) = k * n^2$

polynomiell: $O(n) = k * n^m \quad (m > 1)$

exponentiell:
(Hanoi) $O(n) = k * 2^n$

genauer Wert bei Hanoi: $2^n - 1$
kleinere Dimension bei $O(n)$ weggelassen
– es kommt auf Größenordnung an

Komplexitätsklassen: Zusammenhänge



Komplexitätsklassen: ausgewählte Werte

n	2	8	10	100	1000
konstant	1	1	1	1	1
logarithmisch (power1)	1	3	4	7	10
linear (power)	2	8	10	100	1000
quadratisch	4	64	100	10.000	1.000.000
exponentiell (Hanoi)	4	256	1024	~10 Mrd.	~10 ¹⁰⁰

(Größere Dimension entscheidet, kleinere Dimension weggelassen)

Komplexitätsbetrachtungen

Programmierung:

Suche nach effizienten Algorithmen

– nicht jeder Algorithmus ist geeignet

- Lohnt sich die Entwicklung eines Programms überhaupt (rechnet 100000 Jahre)?
- Welche Eingaben verkraftet das Programm bzgl. der Laufzeit (vgl. Hanoi-Programm)?