Teil II : Rekursive Algorithmen

- Rekursive Prozeduren und Funktionen
- „Teilen-und-Herrschen“
- Türme von Hanoi

1. Rekursive Prozeduren und Funktionen

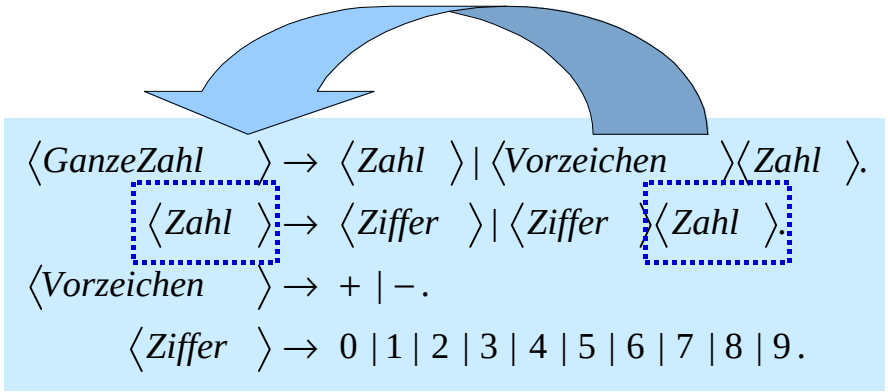
- Definition rekursiver Funktionen
- Struktur rekursiver Algorithmen
- Rekursion vs Iteration

Definition rekursiver Funktionen

Einordnung

- **Bisher :** **Listen** als Datenstrukturen, bei denen eine **Zeiger-Variable** eines Elements (Knoten) **wieder auf ein Element desselben Typs** verweist (→ rekursive Definition einer Datenstruktur);
  vgl. auch die rekursive Definition von **Syntaxdiagrammen** !

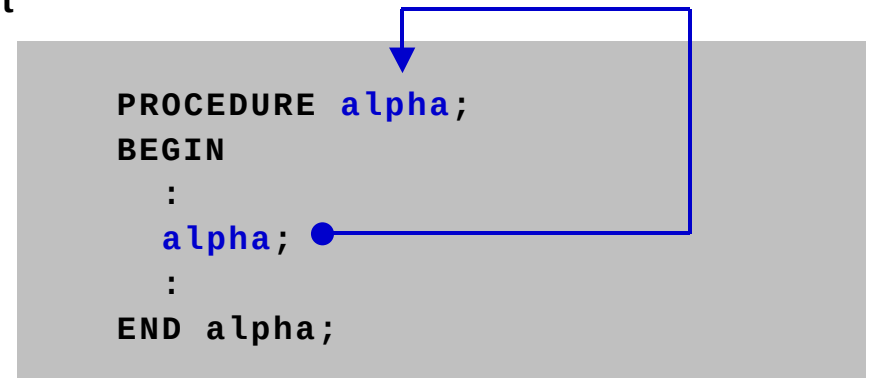
BNF : Syntax für ganze Dezimalzahlen



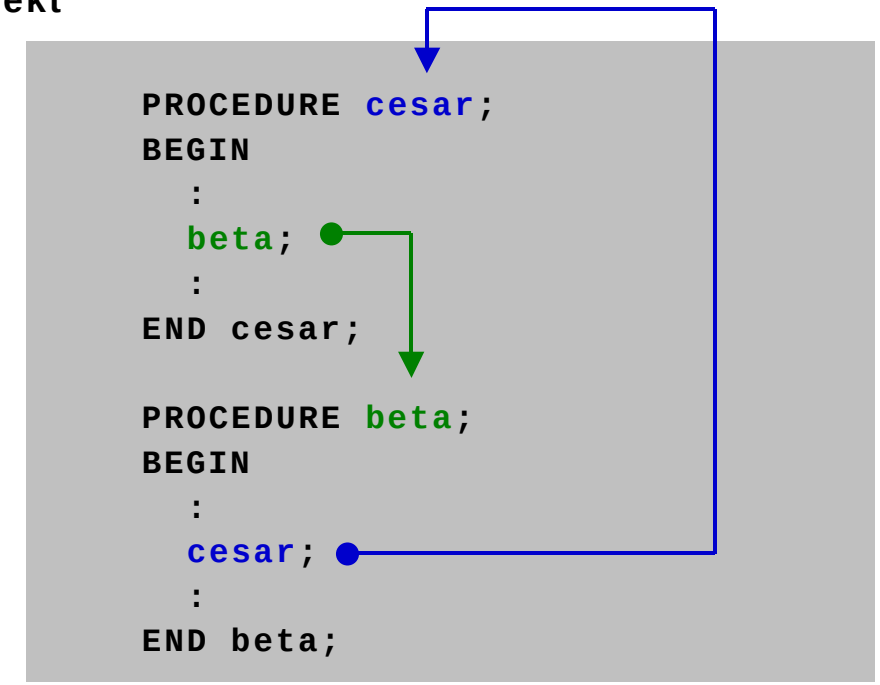
- **Neu :** Eine **Prozedur**, die **sich selbst aufruft** – möglicherweise indirekt über andere Prozeduren – heißt **rekursiv**

Struktur

1. Rekursivität **direkt**



2. Rekursivität **indirekt**



Mechanismen

- Durch verschiedene rekursive Aufrufe einer Prozedur entstehen jeweils zugehörige **Inkarnationen**

  → die **Anzahl** der Inkarnationen kann je nach Algorithmus **variieren** und hängt von der Verarbeitungsaufgabe ab !

- Die jeweils aktiven Inkarnationen besitzen **eigenständige Speicherbereiche** für
  - ihre **lokalen Variablen**
  - ihre **Parameter**

- **Parameterübergabe :**

  → Rekursionen und die Verwaltung eigener Speicherbereiche für die Inkarnationen setzt einen Parameterübergabe-Mechanismus für Prozeduren voraus, der einen „**pulsierenden Speicher**“ (~ „**Stack**“) verwendet !

  ⇒ es gibt Programmierspachen, die keine rekursiven Aufrufe erlauben ! (z.B. FORTRAN)

Beispielbetrachtungen – Mathematische Funktionen

1. **Fakultät**-Funktion

Def. :

$$0! := 1$$

$$n! := n \cdot (n-1)! \quad , \quad n \geq 1$$

$$\Rightarrow \quad fact(n) = \begin{cases} 1 & , n = 0 \\ n \cdot fact(n-1) & , n > 0 \end{cases}$$

Berechnung der Fakultät (rekursiver Algorithmus) :

```
PROCEDURE factorial(
  n :  INTEGER) :  INTEGER;

BEGIN
  IF n = 0 THEN
    RETURN 1
  ELSE
    RETURN n * factorial(n-1)
  END
END factorial;
```

Hinweis : Fakultät-Funktion ist für **n < 0 nicht definiert**;
der Aufruf

`factorial(-1)`

resultiert in einer Endlos-Schleife ! (~ undefiniert ≠ Endlos-Schleife)

⇒ bessere Lösung durch Verwendung von **CARDINAL** !

Übersetzung in eine <u>iterative</u> Darstellung :

```
PROCEDURE factorial(
  n :  CARDINAL) :  CARDINAL;

VAR
  fact, i :  CARDINAL;

BEGIN
  fact :=  1;
  FOR i := n TO 1 BY –1 DO
    fact :=  fact * i
  END;
  RETURN fact
END factorial;
```

Hinweis : Derart **<u>einfache</u>** Rekursionsbeziehungen können direkt in eine einfache **Iterationsform mittels FOR-Schleifen** gebracht werden !

2. **Fibonacci**-Zahlen

Ursprung : Beispiel zur mathematischen Populationsdynamik (→ Biomathematik)

„Das Weibchen eines Kaninchenpaars wirft von der Vollendung des 2. Lebensmonats an allmonatlich ein neues Kaninchenpaar. Man berechne die Anzahl F(n) der Kaninchenpaare im Monat n, wenn im [nach] Monat 0 genau ein neugeborenes Kaninchenpaar vorhanden ist.“

(Aufgabe von Leonardo von Pisa (= Fibonacci), ca. 1180 – ca. 1250; aus K. Jacobs. Einführung in die Kombinatorik, 1983)

Rekursion für die Folge F(0), F(1), ...

Def. :

$$F(0) = 0 \qquad \text{(manchmal auch = 1, siehe oben !)}$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad , \quad n \geq 2$$

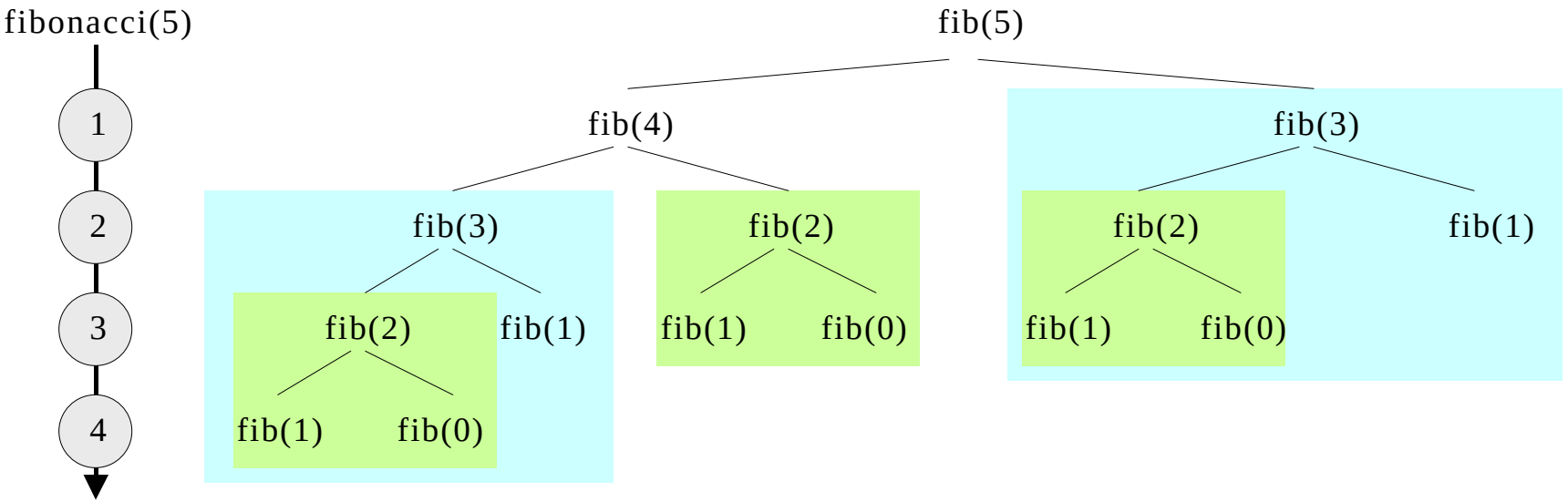
Generiert die Folge :

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

Berechnung mittels rekursiven Algorithmus‘ :

```
PROCEDURE fibonacci(
  n :  CARDINAL) :  CARDINAL;

BEGIN
  IF n <= 1 THEN
    RETURN n      (* Ergebnis: 0 oder 1 *)
  ELSE
    RETURN fibonacci(n–1) + fibonacci(n-2)
  END
END fibonacci;
```

Aufrufe der Prozedur :



Struktur :

$$
\begin{aligned}
F(5) &= F(4) + F(3) \\
&= F(3) + F(2) + F(2) + F(1) \\
&= F(2) + F(1) + F(1) + F(0) + F(1) + F(0) \\
&= F(1) + F(0) \\
&= F(1) + F(0) + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 5
\end{aligned}
$$

n		2	3	4	5	6	7	
F(n)		1	2	3	5	8	13	
Fibonacci	Aufrufe	2	4	8	14	24	...	
	Blätter	2	3	5	8	13	...	

⇒ **Programm mit exponentiellem Zeitbedarf !**

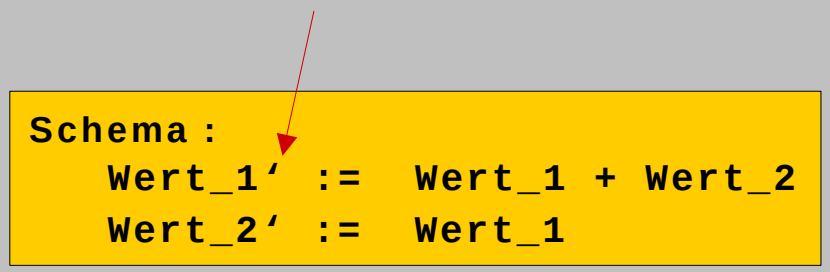
Iteratives Programm mit linearem Zeitbedarf :

```
CONST
  MAX =  100;

TYPE
  fibString =  ARRAY [0..MAX] OF CARDINAL;
  fibPtr    =  POINTER TO fibString;

PROCEDURE fibonacci() :  fibPtr;

VAR
  F :  fibPtr;
  i :  CARDINAL;

BEGIN
  NEW(F);
  F^[0] :=  0;
  F^[1] :=  1;

  FOR i := 2 TO MAX DO
    F^[i] :=  F^[i-1] + F^[i-2]
  END;
  RETURN F
END fibonacci;
```

Iteratives Programm, das als Funktion direkt – d.h. ohne lineares Feld zur Speicherung – den Wert F(n) bestimmt :

```
PROCEDURE fibonacci(
  n :  CARDINAL) :  CARDINAL;

VAR
  i, val1, val2, fib :  CARDINAL;

BEGIN
  IF n = 0 THEN
    fib :=  0
  ELSE
    val2 :=  0;
    val1 :=  1;

    FOR i := 1 TO n DO
      val1 :=  val1 + val2;
      val2 :=  val1 – val2   (* ergibt das alte ‘val1‘ ! *)
    END;
    fib :=  val1
  END;
  RETURN fib
END fibonacci;
```

neuer Wert (nach Zuweisung)



Ergebnisberechnung :

n	val1	val2	fib
0			0
1	1	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	5	3	5
6	8	5	8
7	13	8	13
8	21	13	21

Beispiel-Implementierungen – Vergleich der Laufzeiten

Rekursive Lösung

```
MODULE FibonacciRek;
  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

VAR
 n :  CARDINAL;


PROCEDURE fibonacci(
  n :  CARDINAL) :  CARDINAL;

BEGIN
  IF n <= 1 THEN
    RETURN n      (* Ergebnis: 0 oder 1 *)
  ELSE
    RETURN fibonacci(n-1) + fibonacci(n-2)
  END
END fibonacci;


BEGIN (* -- Hauptprogramm -- *)
  WriteString("Fibonacci-Fkt rekursiv (n >= 0):");
  WriteLn;
  WriteString("  n =  "); ReadCard(n);
  WriteLn;

  WriteString("Ergebnis :  "); WriteCard(fibonacci(n),15); WriteLn
END FibonacciRek.
```

Iterative Lösung

```
MODULE FibonacciIter;
  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;


VAR
  n :  CARDINAL;


PROCEDURE fibonacci(
  n :  CARDINAL) :  CARDINAL;

VAR
  i, a, b, fib :  CARDINAL;

BEGIN
  IF n = 0 THEN
    fib := 0      (* Ergebnis: 0 *)
  ELSE
    a := 0;
    b := 1;
    FOR i := 1 TO n DO
      a := a + b;
      b := a - b      (* ergibt das alte 'a' ! *)
    END;
    fib := a
  END;
  RETURN fib
END fibonacci;


BEGIN (* -- Hauptprogramm -- *)
  WriteString("Fibonacci-Fkt iterativ (0 <= n <= 47):");
  WriteLn;
  WriteString("  n =  "); ReadCard(n);
  WriteLn;

  WriteString("Ergebnis :  "); WriteCard(fibonacci(n),15); WriteLn
END FibonacciIter.
```

Parameterbeispiele : n = 35, 36, 37, ..., 40

Struktur rekursiver Algorithmen

Typischer Aufbau einer rekursiven Prozedur

- **Generell :**

  Viele Aufgabenstellungen, Funktions-Definitionen, etc. sind **von „Natur“ aus rekursiv**, so daß die Formulierung einer **zugehörigen rekursiven Prozedur** oft die **eleganteste und klarste Lösung** der Aufgabe darstellt !

  → häufig stellen rekursive Prozeduren nicht die effizientesten Lösungen dar ... !

```
PROCEDURE recProc(...);

  :

BEGIN
  IF < Rekursionsende erreicht > THEN
    < nicht-rekursiver Teil >
  ELSE
    :
    recProc(...);   (* ein oder mehrere rekursive Aufrufe *)
    :
  END;
  :
END recProc;
```

- **Präzisierung :**

  Häufig ist der Test nach Erreichen des Rekursionsendes an einen **Parameter** der (rekursiven) Prozedur geknüpft !

```
PROCEDURE recProc(
  n :  CARDINAL;
  ...);

  :

BEGIN
  IF n = 0 THEN
    < nicht-rekursiver Teil = Rekursionsende >
  ELSE
    :
    recProc(n-1, ...);   (* ein oder mehrere rekursive Aufrufe
                            von 'recProc' mit Parameterwert(en) n' < n *)
    :
  END;
  :
END recProc;
```

Eigenschaft : Für die Berechnung der **Zwischenergebnisse** bei der rekursiven Berechnung werden keine (zusätzlichen) lokalen Variablen benötigt

⇒ **Direkte Rechnung** mit den jeweiligen Parameterwerten verschiedener Inkarnationen

zur Erinnerung ...

Berechnung der Fakultät :

```
PROCEDURE factorial(
  n :  INTEGER) :  INTEGER;

BEGIN
  IF n = 0 THEN
    RETURN 1
  ELSE
    RETURN n * factorial(n-1)
  END
END factorial;
```

Terminations-Bedingung – Rekursionsende

Rekursionsende : Ergebnis bei Ende des Rekursions-Abstiegs

- Rekursion – rekursiver Abstieg
- Verknüpfung der Teilresultate

Konstruktion rekursiver Algorithmen (allgemeine Hinweise)

1. Spezifikation der Aufgabe

→ wie bei anderen Vorhaben !

„Schnittstellen-Beschreibung“ : **Identifikation der Rolle der Parameter** !

2. Konstruktion

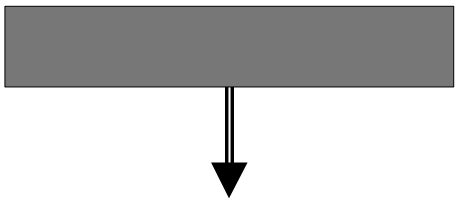
a) Anfang : IF-Abfrage zur Bestimmung des Rekursionsendes
(~ Beginn des Rekursions-Aufstiegs zur Berechnung des Ergebnisses)
→ in diesem Teil findet **kein** rekursiver Aufruf statt !

b) Rekursiver Zweig :

→ „Denkweise“ :

Annahme : zu formulierenden rekursive Prozedur existiert bereits (wie ein Programm in einer Bibliothek)

⟶ „Black Box“



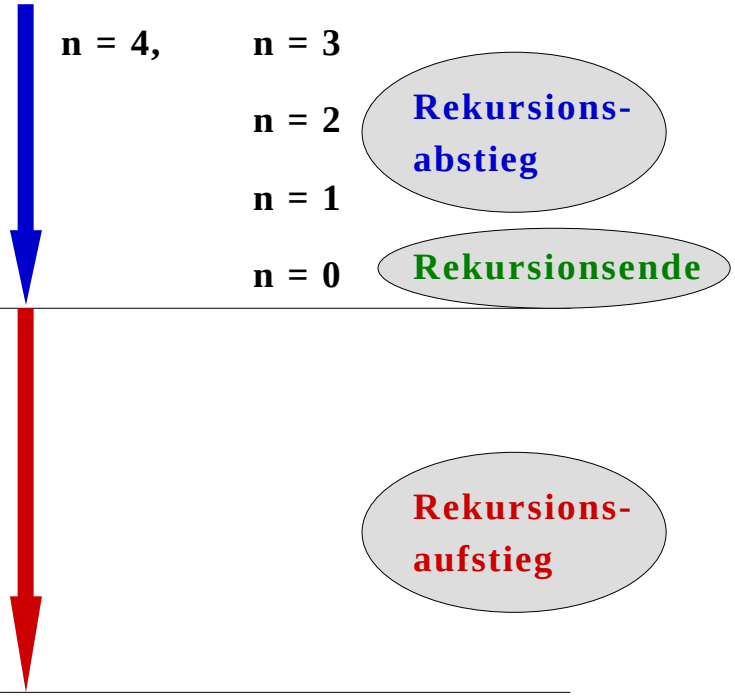
Wirkung eindeutig durch
Schnittstellen-Beschreibung gegeben !

Phasen eines rekursiven Programms

- Erzeugung der verschiedenen **Inkarnationen** (~ **Rekursionsabstieg**)
- **Rekursionsende**
- Verknüpfung der Zwischenergebnisse (~ **Rekursionsaufstieg**)

Beispiel : Berechnung von **4!**

$$
\begin{aligned}
factorial\ (4) &= 4 \cdot factorial\ (3) \\
&= 4 \cdot (3 \cdot factorial\ (2)) \\
&= 4 \cdot (3 \cdot (2 \cdot factorial\ (1))) \\
&= 4 \cdot (3 \cdot (2 \cdot (1 \cdot factorial\ (0)))) \\
&= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) \\
&= 4 \cdot (3 \cdot (2 \cdot 1)) \\
&= 4 \cdot (3 \cdot 2) \\
&= 4 \cdot 6 \\
&= 24
\end{aligned}
$$



Verlauf einer Rekursion :

```
MODULE ZeigeRekursion;
  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

CONST
  EINRUECKEN =  "    |";

VAR
  maxAnzahl :  CARDINAL;


PROCEDURE rekursiv(
  rest, max :  CARDINAL);

VAR
  tiefe, i :  CARDINAL;

BEGIN
  tiefe := max - rest;

  WriteLn; WriteLn;
  FOR i := 1 TO tiefe DO
    WriteString(EINRUECKEN);
  END;

  WriteString("Inkarnation Nr. :  "); WriteCard(tiefe, 0);
  IF (rest = 0) THEN
    WriteString(" --> ENDE DER REKURSION ")
  ELSE
    WriteString(" :: Aufruf rekursiv(");
    WriteCard(tiefe + 1, 0); WriteString(")");
    rekursiv(rest - 1, max)
  END;

  WriteLn; WriteLn;
  FOR i := 1 TO tiefe DO
    WriteString(EINRUECKEN)
  END;
  WriteString("Abschluss der Inkarnation Nr. :  "); WriteCard(tiefe, 0)
END rekursiv;


BEGIN (* -- Hauptprogramm -- *)
  WriteLn; WriteString(" Maximale Anzahl der Rekursionen (> 0) :  ");
  ReadCard(maxAnzahl);

  IF (maxAnzahl > 0) THEN
    WriteLn; WriteString(" Start :  "); WriteLn;
    rekursiv(maxAnzahl, maxAnzahl);
    WriteLn
  END
END ZeigeRekursion.
```

Terminationsbedingung

Rekursiver Aufruf

Aufstieg

Ergebnis – Beispielausgabe für maxAnzahl = 7

```
Maximale Anzahl der Rekursionen (> 0) :
Start :

Inkarnation Nr. :  0 :: Aufruf rekursiv(1)
   |Inkarnation Nr. :  1 :: Aufruf rekursiv(2)
   |   |Inkarnation Nr. :  2 :: Aufruf rekursiv(3)
   |   |   |Inkarnation Nr. :  3 :: Aufruf rekursiv(4)
   |   |   |   |Inkarnation Nr. :  4 :: Aufruf rekursiv(5)
   |   |   |   |   |Inkarnation Nr. :  5 :: Aufruf rekursiv(6)
   |   |   |   |   |   |Inkarnation Nr. :  6 :: Aufruf rekursiv(7)
   |   |   |   |   |   |   |Inkarnation Nr. :  7 --> ENDE DER REKURSION
   |   |   |   |   |   |   |Abschluss der Inkarnation Nr. :  7
   |   |   |   |   |   |Abschluss der Inkarnation Nr. :  6
   |   |   |   |   |Abschluss der Inkarnation Nr. :  5
   |   |   |   |Abschluss der Inkarnation Nr. :  4
   |   |   |Abschluss der Inkarnation Nr. :  3
   |   |Abschluss der Inkarnation Nr. :  2
   |Abschluss der Inkarnation Nr. :  1
Abschluss der Inkarnation Nr. :  0
```

Rekursion vs Iteration

Fundamentalaussage

Rekursive Prozeduren lassen sich stets in eine **nicht-rekursive**, d.h. iterative, Formulierung überführen, indem der Programmierer den **„pulsierenden Speicher“** für die Speicherung der dynamischen Variablen und Parameter (und deren Verwaltung) **selbst programmiert** !

⇒ Dieser „pulsierende Speicher“ wird mittels eines oder mehrerer **„Stack(s)“** realisiert !

a) bei **rekursiven** Prozeduren ⟶ **System-“Stack“**

b) bei **iterativen** Lösungen ⟶ **benutzerdefinierter „Stack“**

2. „Teilen-und-Herrschen“

- Prinzip „Teilen-und-Herrschen“ („Divide-and-conquer“)
- Binäre Bäume
- Nicht-rekursive Lösung zur Markierung eines Lineals

Prinzip „Teilen-und-Herrschen“ („Divide-and-conquer“)

Konzept

→ Häufig bieten sich rekursive Lösungen für ein Problem an, in denen die Eingabemenge in **2 (etwa gleich große) Teile** zerlegt wird, die **jeweils** durch **rekursive Aufrufe** des Lösungs-Algorithmus bearbeitet werden ! Je nach Aufgabenstellung kann die Zerlegung auch in mehr als 2 Teile erfolgen !

Eigenschaften :

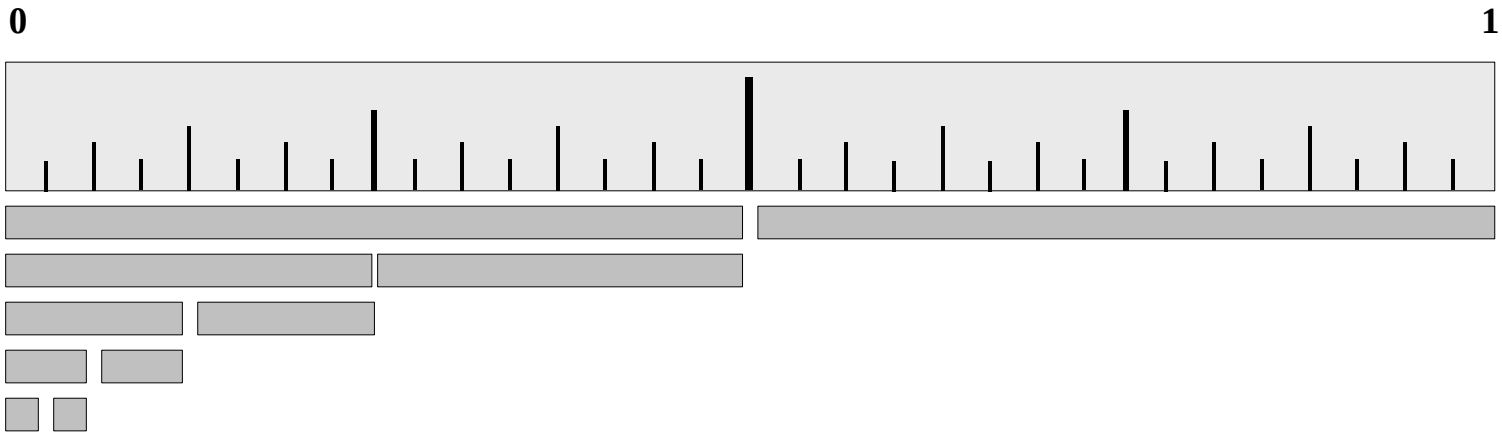
- Lösungen (~ Algorithmen) nach dem „Teile-und-Herrsche“-Prinzip lassen sich **nicht direkt** in **iterative** Verfahren abbilden, wg. mehrfachen rekursiven Aufrufs
  (→ vgl. Berechnung der Fakultät in „**`factorial`**“)

- Normalerweise finden bei „Teilen-und-Herrschen“-Verfahren keine redundanten Berechnungen statt, da die Eingabedatenmenge in **nicht**-überlappende Teilmengen zerlegt wird :

$$card\ (A \cup B) = card\ (A) + card\ (B) \qquad \text{oder}$$

$$A \cap B = \varnothing$$

Beispiel : Markieren eines Lineals



Aufgabe :

N Markierung bei $\underbrace{\frac{1}{2}[0:1]}_{1/2}$

N-1 Markierung bei $\underbrace{\frac{1}{2}\left[0:\frac{1}{2}\right]}_{1/4}$ und $\underbrace{\frac{1}{2}\left[\frac{1}{2}:1\right]}_{3/4}$

N-2 Markierung bei $\underbrace{\frac{1}{2}\left[0:\frac{1}{4}\right]}_{1/8}$ und $\underbrace{\frac{1}{2}\left[\frac{1}{4}:\frac{1}{2}\right]}_{3/8}$ und $\underbrace{\frac{1}{2}\left[\frac{1}{2}:\frac{3}{4}\right]}_{5/8}$ und $\underbrace{\frac{1}{2}\left[\frac{3}{4}:1\right]}_{7/8}$

N-3

Die „Ebene“ bestimmt die Länge der Markierung !

- **Strategie der rekursiven Lösung**

  1. Das Lineal wird sukzessive **halbiert**
  2. Die **Mitte** wird **markiert**
     (Länge der Markierung = Anzahl der bisherigen Zerlegungen)
  3. Für jede (aktuelle) Lineal-Hälfte werden die Schritte

     Halbierung + Markierung

     wiederholt
  4. **Ende** : Länge der Markierung = 0 <u>oder</u> gleichbedeutend
     Länge des aktuellen Teil-Lineals = 0

- **Algorithmus – Entwurfsentscheidungen**

  1. Das **Lineal** ist als **lineares Feld** aus INTEGER / CARDINAL-Elementen realisiert
  2. Die **Werte der Elemente** (des Feldes) bezeichnen die **Längen der Markierungen**
     Initialisierung (angenommen) : $(\forall i, 0 \leq i \leq N-1): \boxed{A[i] = 0}$

```
  :
CONST
  N =  < cardinal >;

VAR
  ruler                 :  ARRAY [0..N-1] OF CARDINAL;
  begin, end, mheight   :  CARDINAL;

PROCEDURE mark(
  index, height :  CARDINAL);
BEGIN
  ruler[index] :=  height
END mark;

PROCEDURE rule(
  i_begin, i_end, mheight :  CARDINAL);
VAR
  i_middle :  CARDINAL;

BEGIN
  IF mheight > 0 AND (i_end - i_begin) > 0 THEN
    i_middle :=  (i_begin + i_end) DIV 2;
    mark(i_middle, mheight);
    rule(i_begin, i_middle, mheight-1);
    rule(i_middle, i_end, mheight-1)
  END
END rule;
```

Zahl, z.B. 64

Rekursionsende hier mit „leerer“ Aktion !

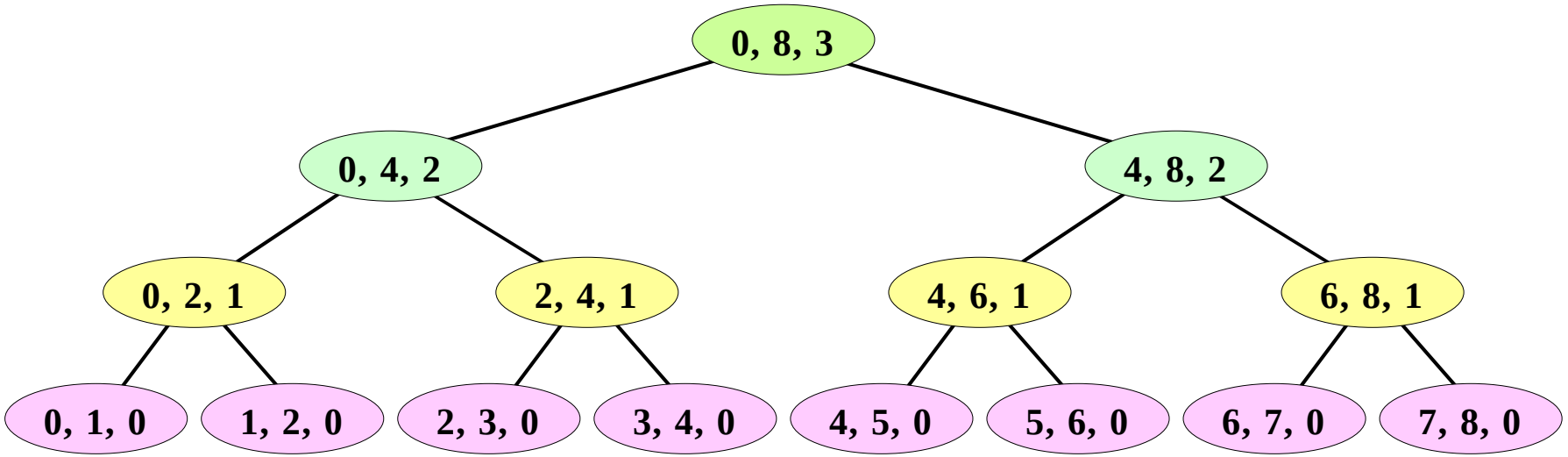
```
BEGIN   (* -- Hauptprogramm *)
  :
  mheight :=  ...;
  begin   :=  0;
  end     :=  N - 1;

END ...
```

Binäre Bäume

Beispiel – Markierung des Lineals

Bsp. : **begin = 0 mheight = 3**
end = 8

Darstellung der Zerlegung als (binärer) Baum :



Prinzipien – Aufbau und Traversierung von Bäumen

→ vgl. Beispiel-Implementierung (s.o.)

1) Pre-Order

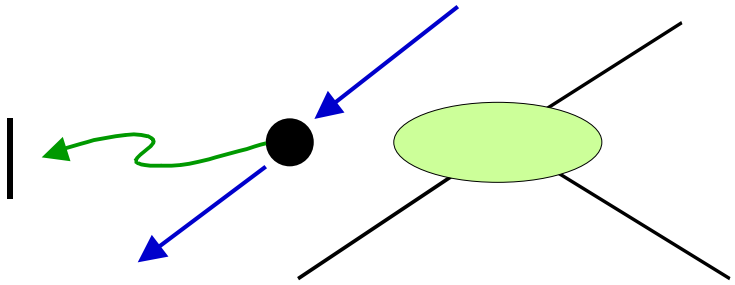
```
mark(i_middle, mheight);
rule(i_begin, i_middle, mheight-1);
rule(i_middle, i_end, mheight-1)
```

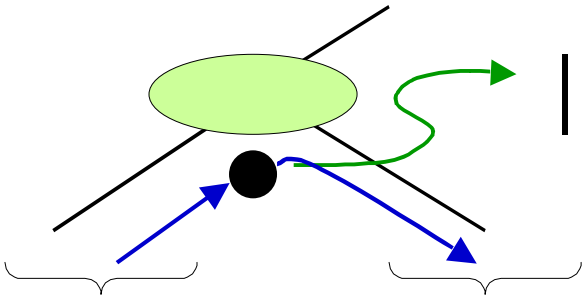
Markierung, wenn Knoten (mit **`mheight > 0`**) angetroffen wird (→ „top-down“)

2) In-Order

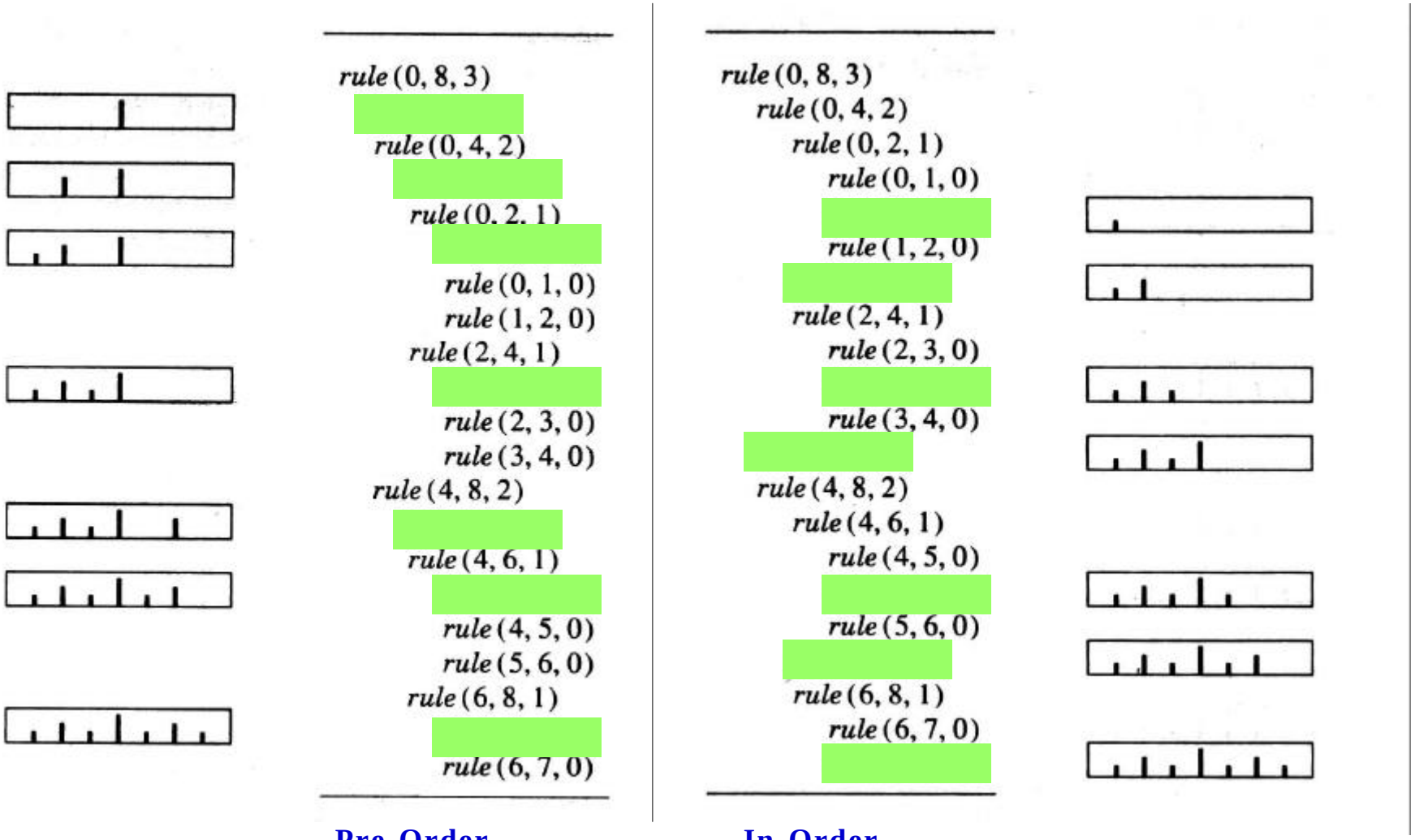
```
rule(i_begin, i_middle, mheight-1);
mark(i_middle, mheight);
rule(i_middle, i_end, mheight-1)
```

Markierung, wenn Knoten nach Abtrieg des jeweils **linken** Teilbaums zur Bearbeitung des (zugehörigen) **rechten** Teilbaums angetroffen wird





Verlauf der Pre-Order bzw. In-Order Rekursion :



(aus R. Sedgewick. Algorithms, 2nd edition. Addison-Wesley Publ. Co., 1988)

Nicht-rekursive Lösung zur Markierung eines Lineals

„Direkte“ Lösung

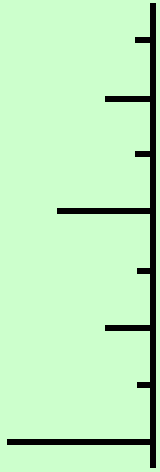
Die einzelnen Markierungen werden in der Folge ihres Auftretens (links → rechts) dargestellt :

```
FOR i := 1 TO N-1 DO
  mark(i, height(i))
END;
```

mit **height(i)** : Höhe = **Anzahl** der „anhängenden“ 0-Bits in der binären Repräsentation von **i**

Bsp. : **N = 8** :
i = 1, .., 7

0000 0001	0
0010	1
0011	0
0100	2
0101	0
0110	1
0111	0
1000	3



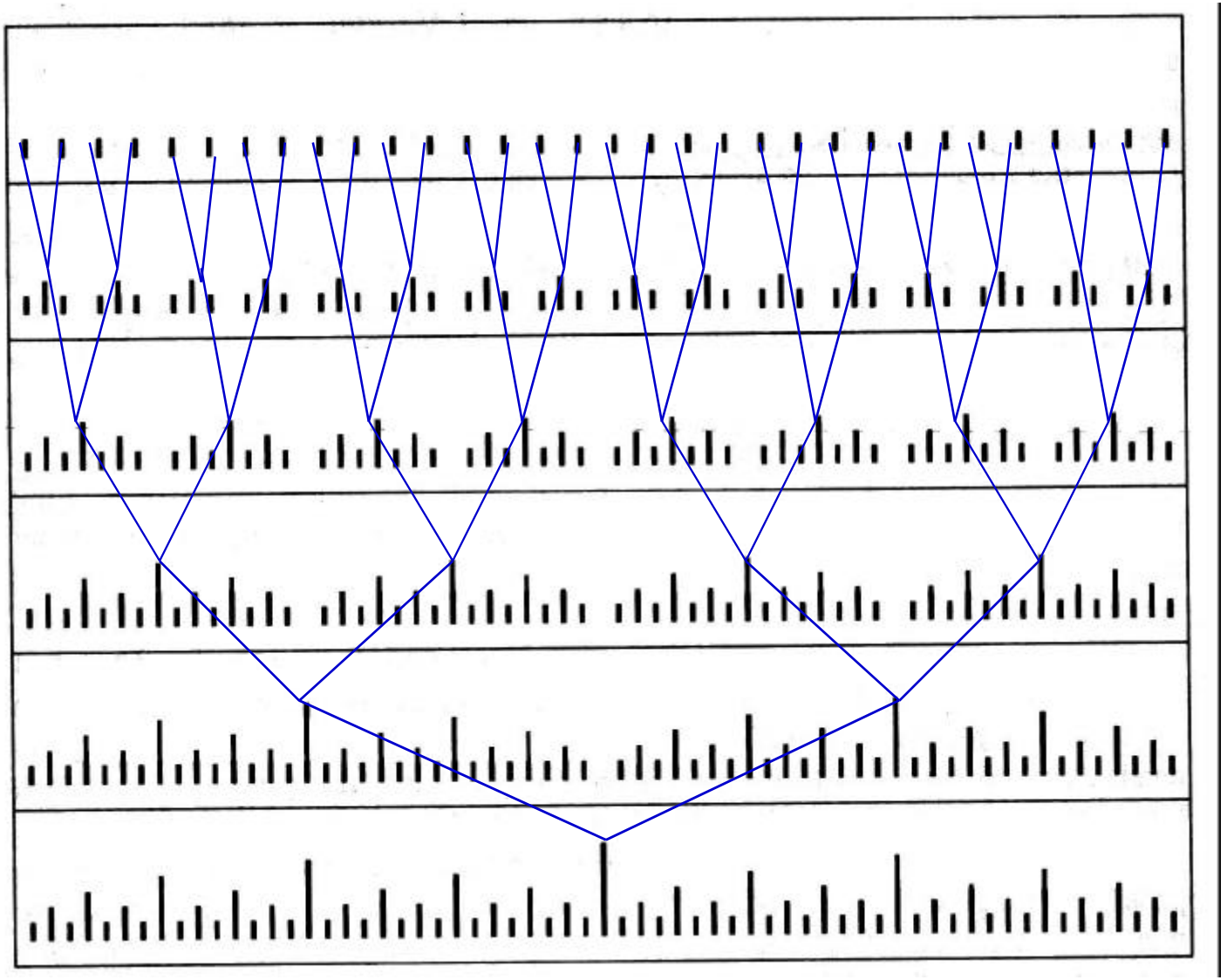
„Abtastung“ mit unterschiedlicher Auflösung

Schema :

Zeichnen der kürzesten Markierungen
2.-kürzesten Markierungen
3.-kürzesten Markierungen
:

→ Erfordert aufwendigere Adreß- / Positionsberechnungen

```
PROCEDURE rule(
  i_begin, i_end, mheight :  CARDINAL);

VAR
  i, j, x, pos :  CARDINAL;

BEGIN
  j :=  1;

  FOR i := 1 TO mheight DO
    FOR x := 0 TO (i_begin + i_end) DIV j DO
      pos := i_begin + j + 2*j*x;
      mark(pos, i)
    END;
    j :=  2 * j
  END
END rule;
```



(aus R. Sedgewick. Algorithms, 2nd edition. Addison-Wesley Publ. Co., 1988)

3. Türme von Hanoi

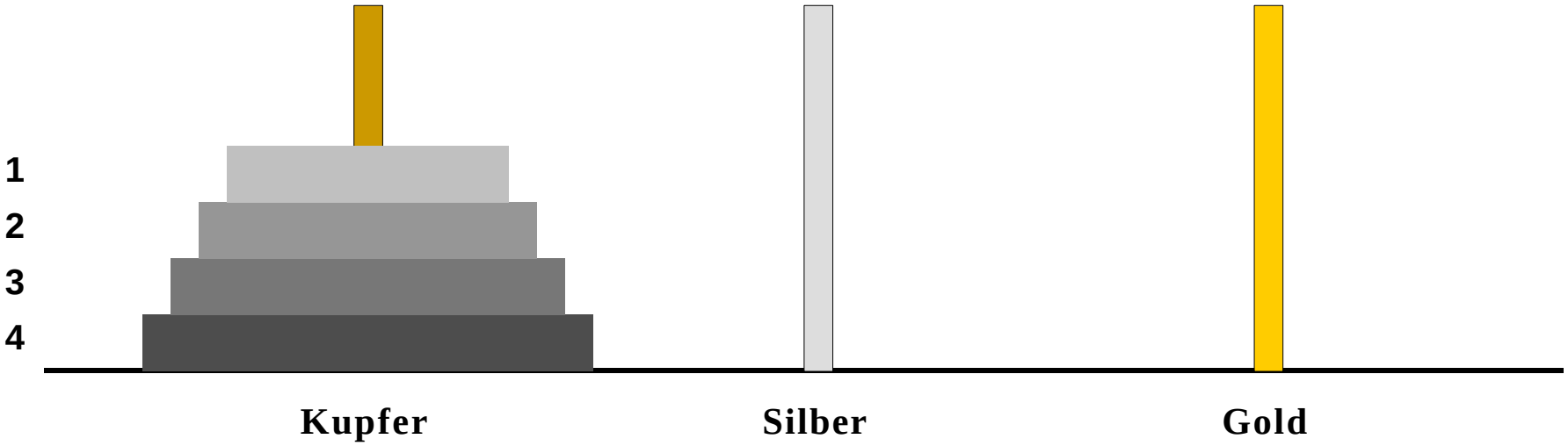
- Problem
- Strategie
- (Rekursiver) Algorithmus
- Beweis der Gültigkeit
- Beispielhafte Realisierungen
- Aufwandsabschätzungen (rekursive Lösung)
- Einfacher nicht-rekursiver Algorithmus

Problem

Nach einer alten Legende standen vor langer Zeit vor einem Tempel in Hanoi drei Säulen :

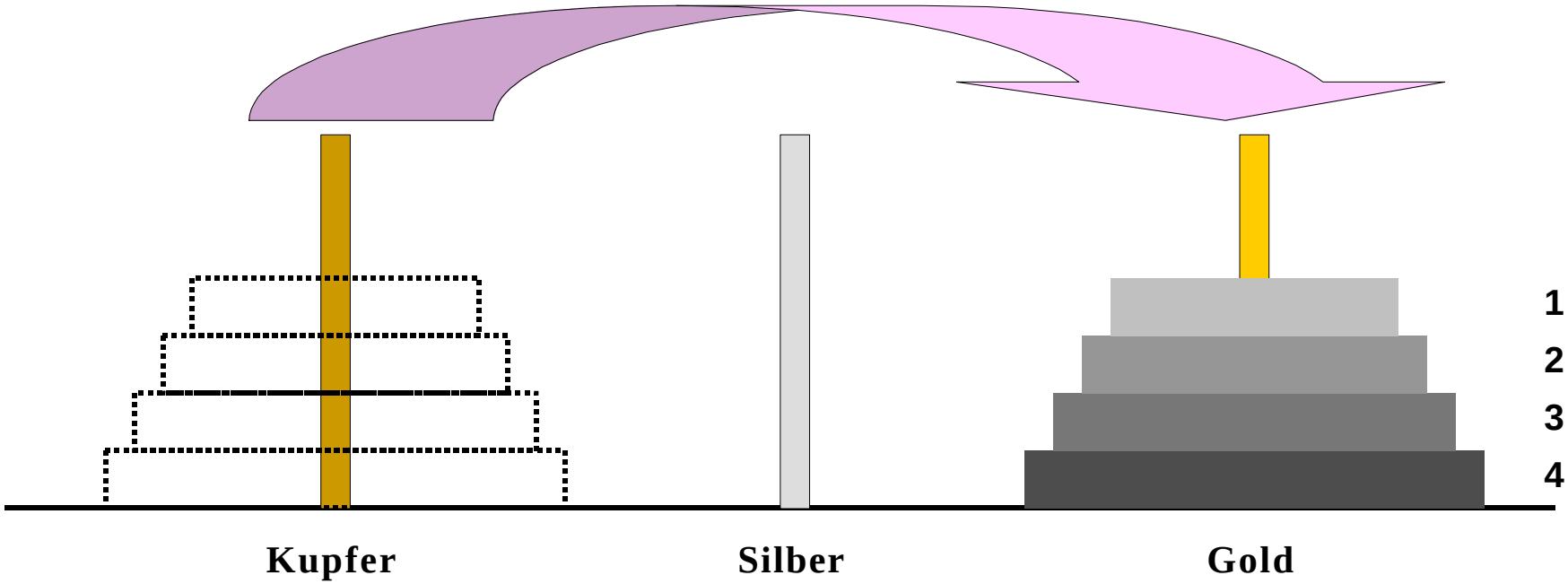
- eine aus **Kupfer**,
- eine aus **Silber** und
- eine aus **Gold**.

Auf der **kupfernen Säule** befanden sich **hundert** verschieden große **Scheiben** aus Porphyr (vulkanisches Gestein), wobei die Scheiben in **ihrer Größe nach oben hin immer kleiner** wurden :



Ein alter Mönch hatte sich die Aufgabe gestellt, **alle Scheiben von der kupfernen zur goldenen Säule** zu tragen. Da die Porphyrscheiben sehr schwer waren, konnte der Mönch **immer nur eine Scheibe** gleichzeitig transportieren. Da die Säule ihm gleichzeitig als Treppe diente, durfte bei der Umschichtung **nie eine größere auf eine kleine Scheibe** gelegt werden.

Wenn der Mönch – so die Legende – seine Aufgabe erfüllt habe, so werde das Ende der Welt kommen.



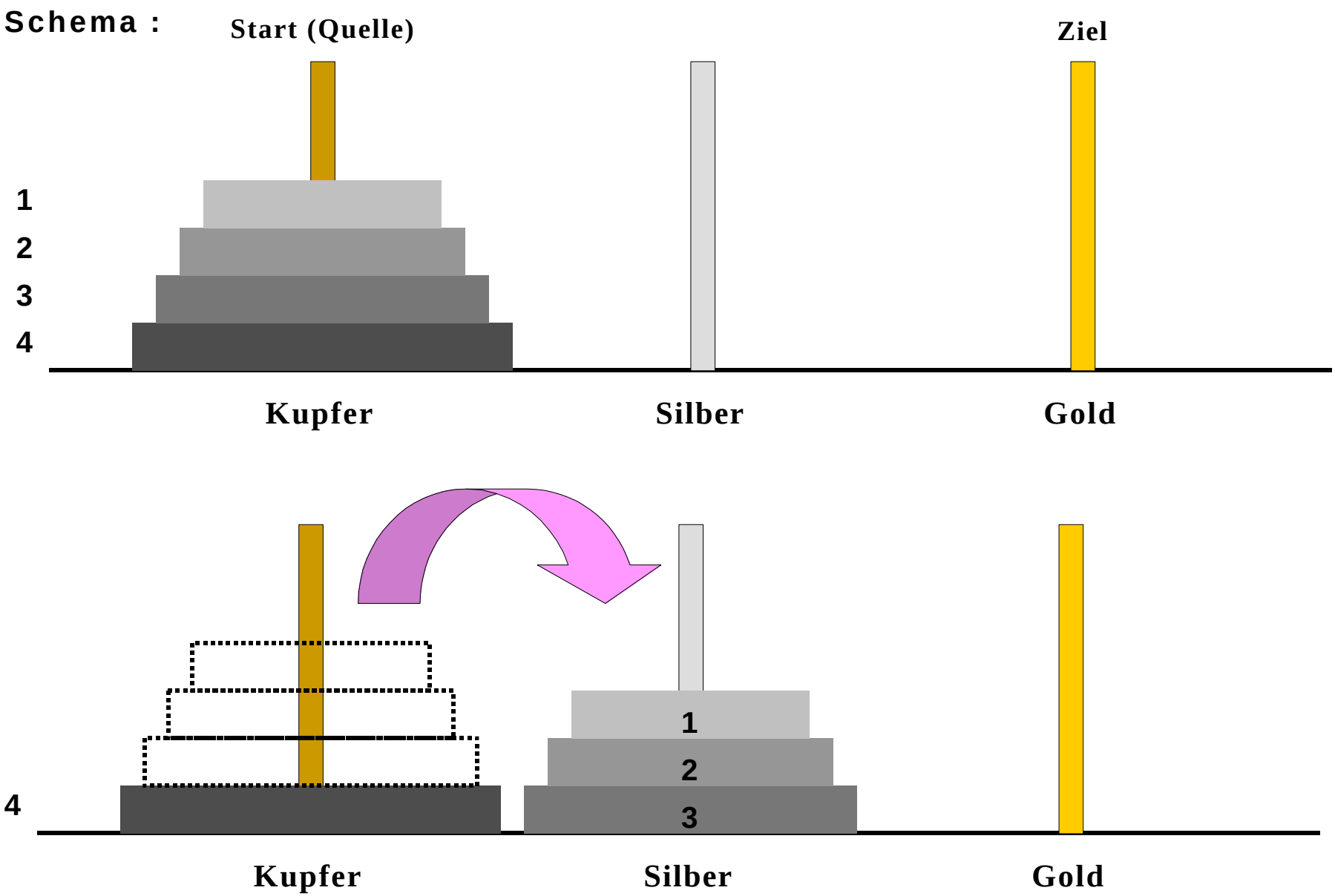
Strategie

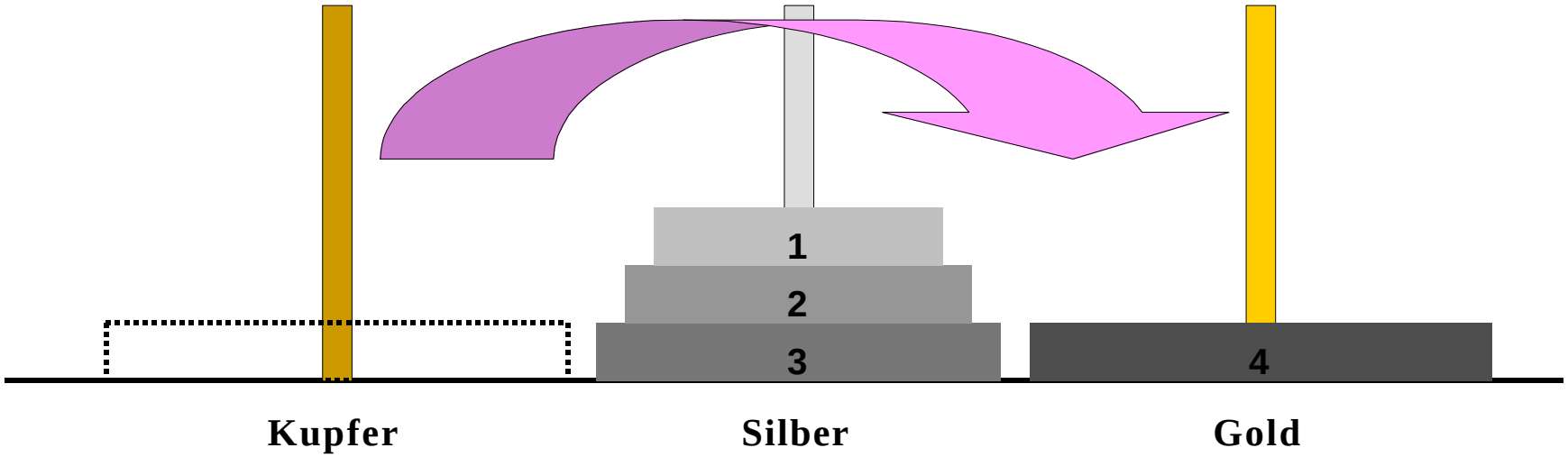
Der Mönch bemerkte sehr schnell, daß er beim Transport der Scheiben **auch die silberne Säule benötigte**, da er ja **immer nur eine Scheibe gleichzeitig tragen** konnte. Nach einigen Tagen Meditation bekam er auf einmal die Erleuchtung:

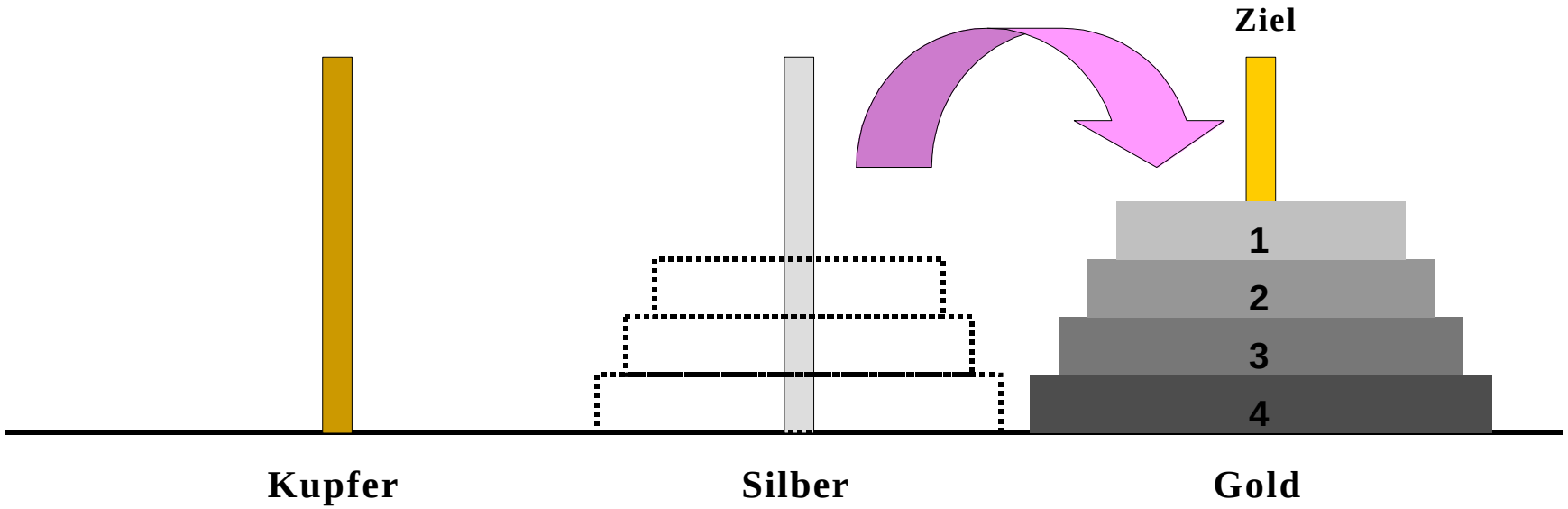
Die Aufgabe kann in drei Teilaufgaben zerlegt werden:

Teil 1 :	Transportiere den Turm – bestehend aus 99 **oberen** Scheiben von der **kupfernen zur silbernen** Säule
Teil 2 :	Transportiere die **übriggebliebene** 100ste Scheibe (ganz unten) von der **kupfernen zur goldenen** Säule
Teil 3 :	Transportiere den Turm mit den 99 Scheiben von der **silbernen zur goldenen** Säule

Beim Betrachten dieses Schemas bemerkte der Mönch, daß Teil 1 und Teil 3 außerordentlich mühsam sein würden. Da er nicht nur ein alter, sondern auch ein weiser Mönch war, entschloß er sich, **Teil 1 von seinem ältesten Schüler ausführen** zu lassen. Wenn dieser mit der Arbeit fertig wäre, würde der **Mönch selbst die große Scheibe von der kupfernen zur goldenen Säule tragen**; und **dann nochmals die Dienste seines ältesten Schülers** in Anspruch nehmen.


Schema :
Start (Quelle)
Ziel
1
2
3
4
Kupfer
Silber
Gold
1
2
3
4
Kupfer
Silber
Gold


1
2
3
4
Kupfer
Silber
Gold


Ziel
1
2
3
4
Kupfer
Silber
Gold

Um seinem ältesten Schüler, der selbst schon in den Jahren war, nicht zu viel Arbeit zu machen, wollte er ihm diesen **Plan** mitteilen, damit auch er es sich leicht machen könnte.

Der **Algorithmus**, den der Mönch am nächsten Tag an die Tempeltür nagelte, ist aus dem Alt-Vietnamesischen übersetzt :

Anleitung, um einen Turm von n Scheiben von der **einen zu der anderen** Säule – unter Verwendung einer weiteren (dritten) Säule – zu transportieren :

wenn der Turm aus mehr als einer Scheibe besteht, dann bitte Deinen ältesten Schüler, einen Turm von n-1 Scheiben von der **ersten zur dritten Säule** zu transportieren;

trage selbst eine Scheibe von der **ersten zur anderen** Säule;

wenn der Turm aus mehr als einer Scheibe besteht, dann bitte Deinen ältesten Schüler, einen Turm von n-1 Scheiben von der **dritten zur anderen** Säule zu transportieren.

Als der Mönch dieses Dokument festgenagelt hatte, fragte er sich, was jetzt zu tun sei – er mußte einen Turm von 100 Scheiben von der kupfernen zur goldenen Säule transportieren. Weil er nach der schweren Denkarbeit der vorherigen Tage etwas zerstreut war, wußte er nicht mehr genau, wie er das machen sollte; als er vor der Tempeltür eine Traube von Menschen sah, die offenbar etwas lasen, erinnerte er sich wieder. Entschlossen drängelte er sich zur Tempeltür und las, wie zu verfahren sei.

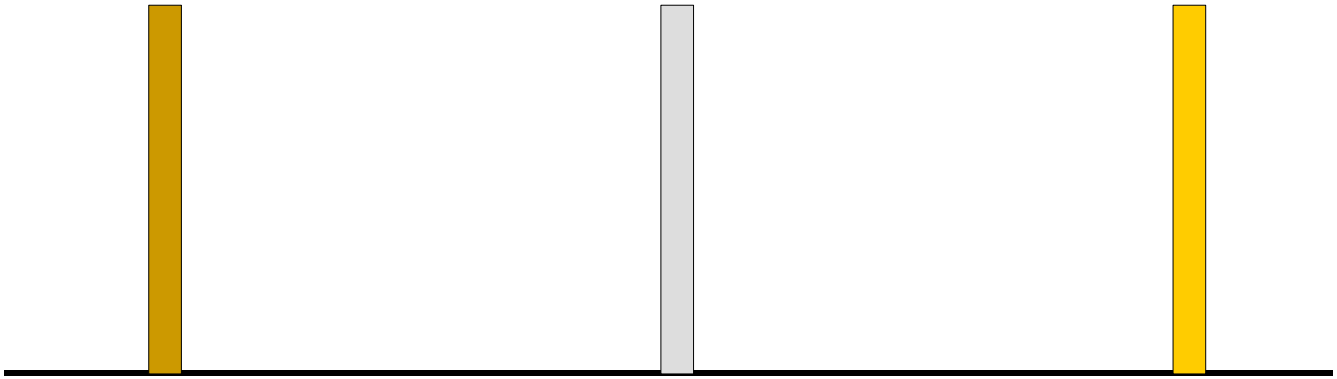
Und so rief er seinen ältesten Schüler zu sich und bat ihn, einen Turm von 99 Scheiben von der kupfernen zur silbernen Säule (unter Verwendung der goldenen) zu transportieren – und sich danach wieder bei ihm zu melden ...

(nach G. Hommel, C.A.H. Koster. Algorithmen, TU Berlin, 1977/78)

(Rekursiver) Algorithmus

Grobstruktur

geg. :	Turm	:	**TOWER** (Höhe : n)
	Ausgangssäule	:	**A (src)**
	Zielsäule	:	**C (dst)**



Numerierung **1 = A** **2 = B** **3 = C**

Rekursiver Algorithmus „Türme von Hanoi“ – erste Skizze

```
TYPE
  COLUMN   =  (1, 2, 3);   (* Saeulen *)
  pntTOWER =  POINTER TO TOWER;
  TOWER    =  < Struktur zur Repräsentation eines Turms >;
  pntELEMS =  POINTER TO ELEMS;
  ELEMS    =  ARRAY [1..2] OF pntTOWER;

PROCEDURE towersOfHanoi(
  t        :  pntTOWER;
  src, dst :  COLUMN);

VAR
  t1, t2 :  pntTOWER;
  pElems :  pntELEMS;

BEGIN
  IF height(t) = 0 THEN
    EXIT     (* Termination *)
  ELSE
    pElems :=  splitTower(t);
    t1     :=  pElems^[1];
    t2     :=  pElems^[2];
    towersOfHanoi(t1, src, servCol(src, dst));
    moveDisk(t2, src, dst);
    towersOfHanoi(t1, servCol(src, dst), dst)
  END
END towersOfHanoi;
```

Rekursionsende:
Beendigung der Zerlegung und Aufstieg !

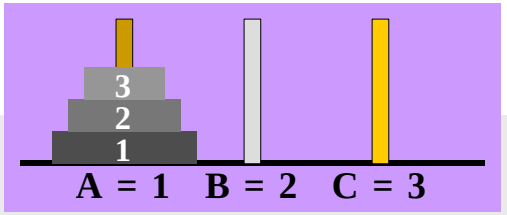
Generiert t1: height(t1) = n – 1
t2: height(t2) = 1

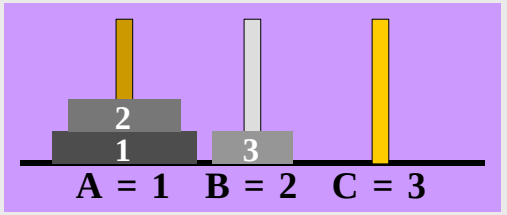
Rekursion

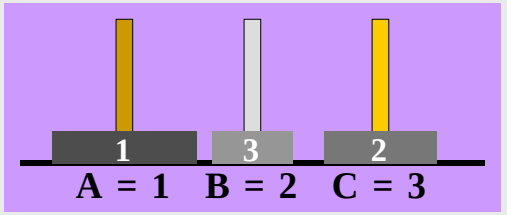
```
PROCEDURE servCol(
  src, dst :  COLUMN): COLUMN;

CONST
  SUM =  6;

BEGIN
  RETURN SUM – (src + dst)
END servCol;
```

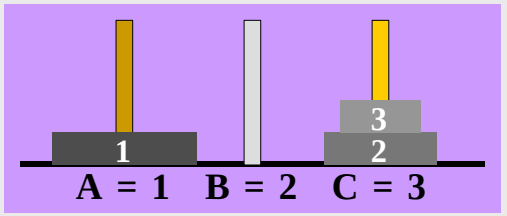
„Handsimulation“

```
towersOfHanoi(t={1-2-3}, A, B)
  height(t) = 3 > 0:
    splitTower(t) ⇒ t1 = {2-3}
                    t2 = {1}
    towersOfHanoi(t1={2-3}, A, servCol(A, B)=<3>=C)
      height(t) = 2 > 0:
        splitTower(t) ⇒ t1 = {3}
                        t2 = {2}
        towersOfHanoi(t1={3}, A, servCol(A, C)=<2>=B)
          height(t) = 1 > 0:
            splitTower(t) ⇒ t1 = {/}
                            t2 = {3}
            towersOfHanoi(t1={/}, A, servCol(A, B)=<3>=C)
              height(t) = 0 ⇒ EXIT
            moveDisk(t2={3}, A, B)
            towersOfHanoi(t1={/}, C, B)
              height(t) = 0 ⇒ EXIT
        moveDisk(t2={2}, A, C)
        towersOfHanoi(t1={3}, B, C)
          height(t) = 1 > 0:
            splitTower(t) ⇒ t1 = {/}
                            t2 = {3}
            towersOfHanoi(t1={/}, B, servCol(B, C)=<1>=A)
              height(t) = 0 ⇒ EXIT
            moveDisk(t2={3}, B, C)
            towersOfHanoi(t1={/}, A, C)
```

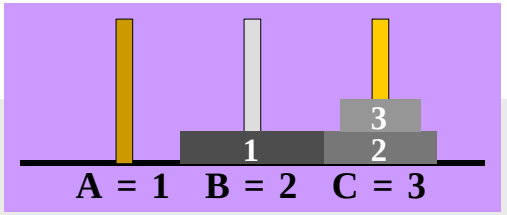


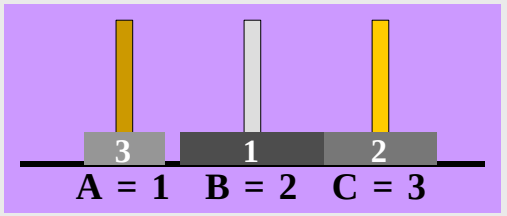


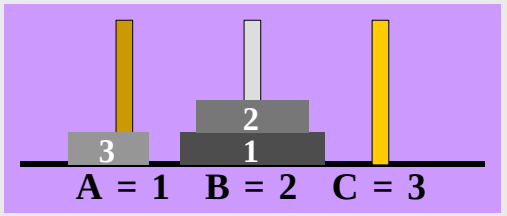


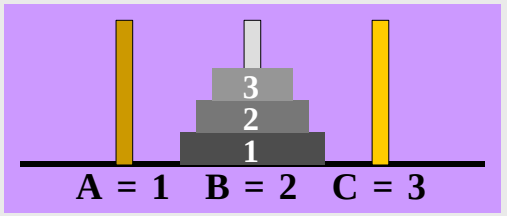


```
          height(t) = 0 ⇒ EXIT
moveDisk(t2={1}, A, B)
towersOfHanoi(t1={2-3}, C, B)
  height(t) = 2 > 0:
    splitTower(t) ⇒ t1 = {3}
                    t2 = {2}
    towersOfHanoi(t1={3}, C, servCol(C, B)=<1>=A)
      height(t) = 1 > 0:
        splitTower(t) ⇒ t1 = {/}
                        t2 = {3}
        towersOfHanoi(t1={/}, C, servCol(C, A)=<2>=B)
          height(t) = 0 ⇒ EXIT
        moveDisk(t2={3}, C, A)
        towersOfHanoi(t1={/}, B, A)
          height(t) = 0 ⇒ EXIT
    moveDisk(t2={2}, C, B)
    towersOfHanoi(t1={3}, A, B)
      height(t) = 1 > 0:
        splitTower(t) ⇒ t1 = {/}
                        t2 = {3}
        towersOfHanoi(t1={/}, A, servCol(A, B)=<3>=C)
          height(t) = 0 ⇒ EXIT
        moveDisk(t2={3}, A, B)
        towersOfHanoi(t1={/}, C, B)
          height(t) = 0 ⇒ EXIT
```









Programmende

Beweis der Gültigkeit

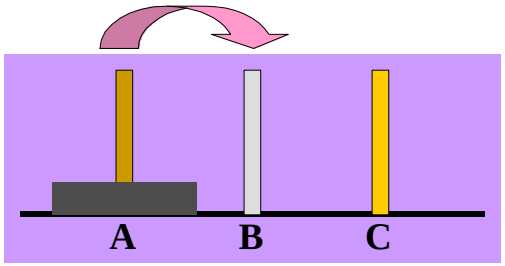
Liefert der Algorithmus stets die korrekte Lösung ?

→ Vollständige Induktion

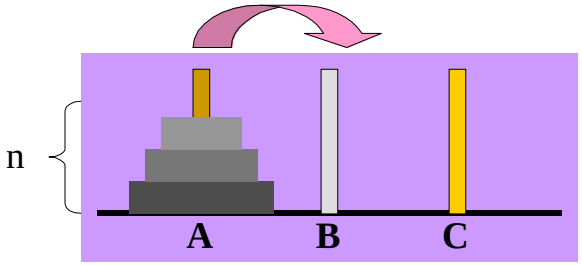
A. Induktionsbeginn : n = 1

Bewege 1 Scheibe von A nach B

```
moveDisk(t, A, B)
```



B. Induktionsannahme : Der Algorithmus arbeitet für n Scheiben korrekt



C. Induktionsschritt : Arbeitet der Algorithmus auch für n + 1 Scheiben korrekt ?

Bewege n + 1 Scheiben von A nach B (Hilfssäule ist C)

```
1.   splitTower(t) ⇒          t1 der Höhe n
                              t2 der Höhe 1 (= 1 Scheibe)
2.   towersOfHanoi(t1, A, C)
3.   moveDisk(t2, A, B)
4.   towersOfHanoi(t1, C, B)
```

---

2. gilt laut **Induktionsannahme**, da **height(t1) = n** und damit

Bewege n Scheiben von A nach C (Hilfssäule : B)

3. gilt laut **Induktionsbeginn**, mit

Bewege 1 Scheibe von A nach B

4. gilt laut **Induktionsannahme**, da **height(t1) = n** und damit

Bewege n Scheiben von C nach B (Hilfssäule : A)

außerdem gilt die Transitivität der Transportoperation :

$$transport\ (A \to C) \wedge transport\ (C \to B) \Rightarrow transport\ (A \to B)$$

⇒ **Induktionsschluß** : Der Algorithmus arbeitet auch für n + 1 Scheiben !

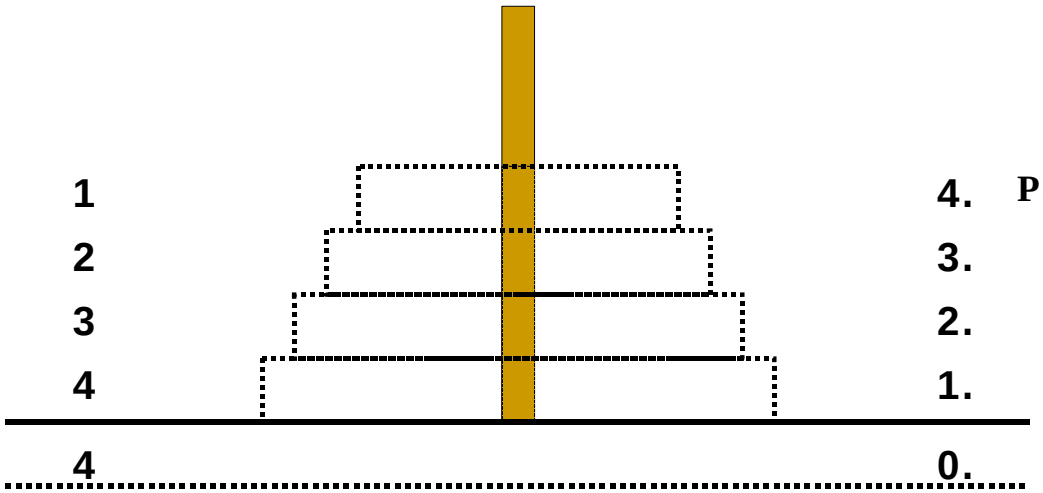
Beispielhafte Realisierungen

Abstrakte Repräsentation

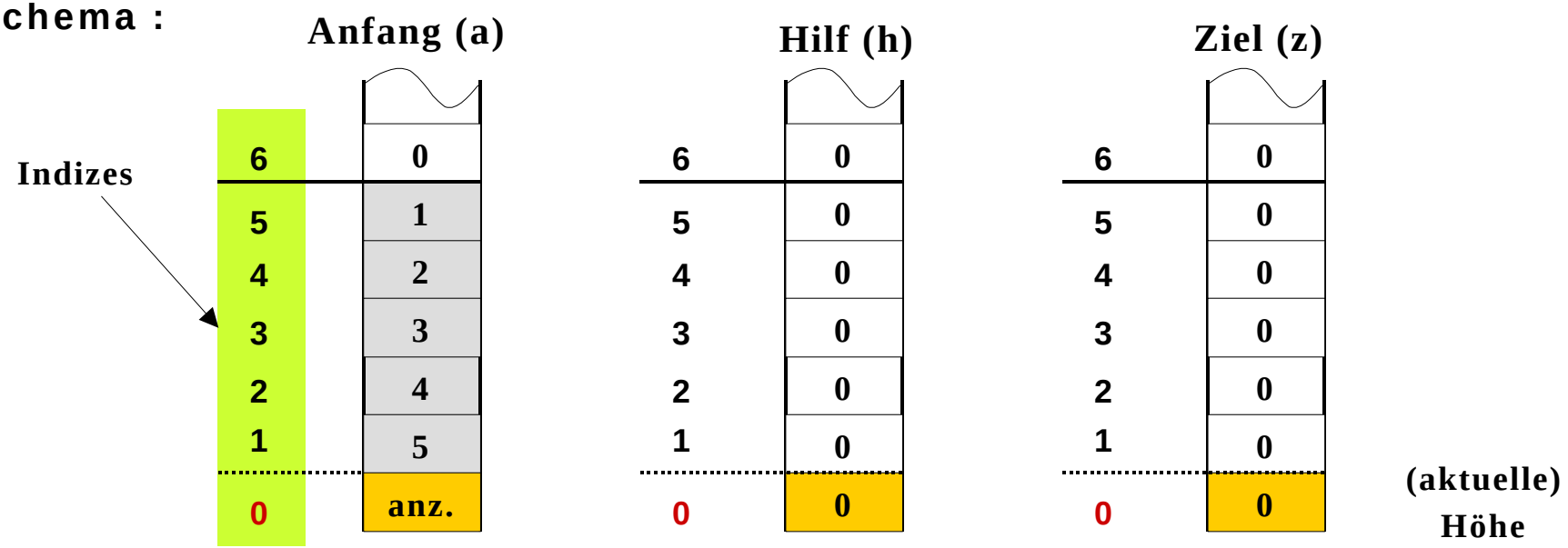
- Die Scheiben werden **implizit** durch Zahlen repräsentiert
- Ein Turm stellt sich als **Anzahl n ≥ 1 von Scheiben-Nummern** dar, wobei die oberste (kleinste) Scheibe durch n bezeichnet wird

  oder

  ein Element zeigt die **aktuelle Höhe des Turms** an einer Säule, die **Scheibennummern** werden an den jeweils durch die **n-te Position** bezeichneten Stellen gelesen



Schema :

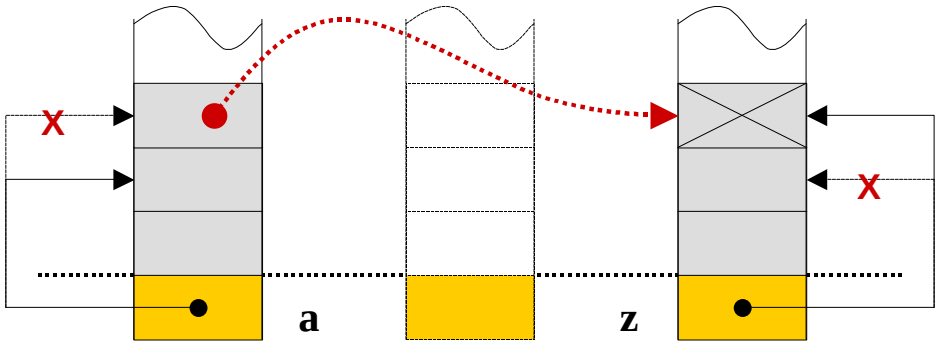


Init

```
anz = 5 :

    1 :   5 - 1 + 1 = 5
    2 :   5 - 2 + 1 = 4
    3 :   5 - 3 + 1 = 3
    4 :   5 - 4 + 1 = 2
    5 :   5 - 5 + 1 = 1
```

Transfer

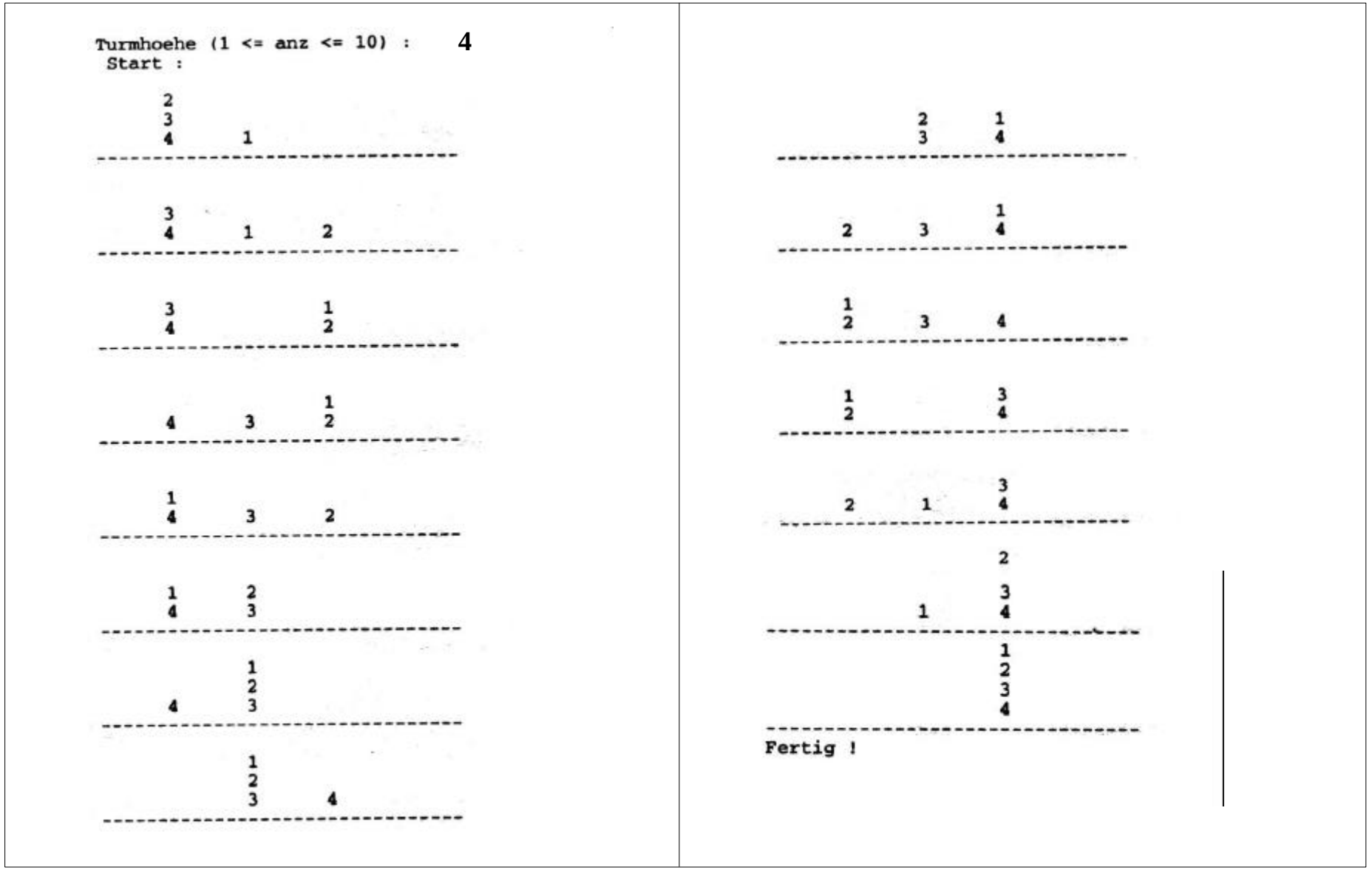
```
n = colA[0] = 5

z[z[0] + 1] :=  a[a[0]]   ( abstraktes moveDisk )
```



Beispiel-Implementierung :

```
MODULE TowersOfHanoi;
  FROM InOut   IMPORT ReadCard, WriteCard, WriteString, WriteLn;
  FROM SysExit IMPORT Exit;

CONST
  Max =  10;

TYPE
  PLATZ =  ARRAY [0..Max] OF CARDINAL;

VAR
  colA, colH, colZ :  PLATZ;     (* colAnfang, colHilf, colZiel *)
  anz, n           :  CARDINAL;


PROCEDURE transfer(
      n       :  CARDINAL;
  VAR a, h, z :  PLATZ;
      anz     :  CARDINAL);

BEGIN
  IF n = 0 THEN
    (* fertig ... Ende des rekursiven Abstiegs *)
  ELSE
    (* es sind noch Scheiben zu transportieren ... *)
    transfer(n-1, a, z, h, anz);

    z[z[0]+1] :=  a[a[0]];      (* abstraktes move( a --> z ) *)
    a[a[0]]   :=  0;
    z[0]      :=  z[0] + 1;
    a[0]      :=  a[0] - 1;

    print(colA, colH, colZ, anz);

    transfer(n-1, h, a, z, anz)
  END
END transfer;


PROCEDURE print(
  a, h, z :  ARRAY OF CARDINAL;
  anz     :  CARDINAL);

CONST
  abstand =  "      ";
  trenner =  "----------------------------------";
  blank   =  " ";
VAR
  i :  CARDINAL;

BEGIN
  (* -- ACHTUNG: Felder muessen vollstaendig initialisiert sein; *)
  (* --          es muss  anz <= HIGH(a)  sein !                 *)
  FOR i := anz TO 1 BY -1 DO

    WriteString(abstand);
    IF (a[i] = 0) THEN
      WriteString(blank)
    ELSE
      WriteCard(a[i], 0)
    END;

    WriteString(abstand);
    IF (h[i] = 0) THEN
      WriteString(blank)
    ELSE
      WriteCard(h[i], 0)
    END;

    WriteString(abstand);
    IF (z[i] = 0) THEN
      WriteString(blank)
    ELSE
      WriteCard(z[i], 0)
    END;
    WriteLn
  END;
  WriteString(trenner); WriteLn
END print;


PROCEDURE init(
  VAR a, h, z :  ARRAY OF CARDINAL;
  VAR anz     :  CARDINAL) :  BOOLEAN;

VAR
  i :  CARDINAL;

BEGIN
  WriteString("Turmhoehe (1 <= anz <= "); WriteCard(Max, 0);
  WriteString(") :  "); ReadCard(anz);
  IF (anz >= 1) & (anz <= Max) THEN
    a[0] :=  anz;
    h[0] :=  0;
    z[0] :=  0;
    FOR i := 1 TO HIGH(a) DO
      IF (i <= anz) THEN
        a[i] := anz - i + 1
      ELSE
        a[i] := 0
      END;
      h[i] :=  0;
      z[i] :=  0
    END;
    RETURN TRUE
  ELSE
    RETURN FALSE
  END
END init;


BEGIN (* -- Hauptprogramm -- *)
  IF (NOT init(colA, colH, colZ, anz)) THEN
    Exit(1)
  ELSE
    n :=  colA[0];
    WriteLn; WriteString(" Start :  "); WriteLn;
    transfer(n, colA, colH, colZ, anz);
    WriteString("Fertig !"); WriteLn; WriteLn
  END
END TowersOfHanoi.
```

Ausgabe (Scheibenbewegungen)



Explizite Repräsentation von Objekten (~ Scheiben) – Vorschlag

```
TYPE
  elemSize       =  CARDINAL;
  pntDISK        =  POINTER TO DISK;
  DISK           =  RECORD
                      size       :  elemSize;
                      nextHigher :  pntDISK
                    END;
  pntTOWER       =  POINTER TO TOWER;
  TOWER          =  RECORD
                      height :  CARDINAL;
                      bottom :  pntDISK
                    END;
  COLUMN         =  (copper, silver, gold);  (* oder :  (1, 2, 3) *)
  pntSplitTower  =  POINTER TO SplitTOWER;
  SplitTOWER     =  RECORD
                      disk, rest : pntTOWER
                    END;
```

Jede Scheibe ist ein Eintrag/Objekt mit seiner Größe

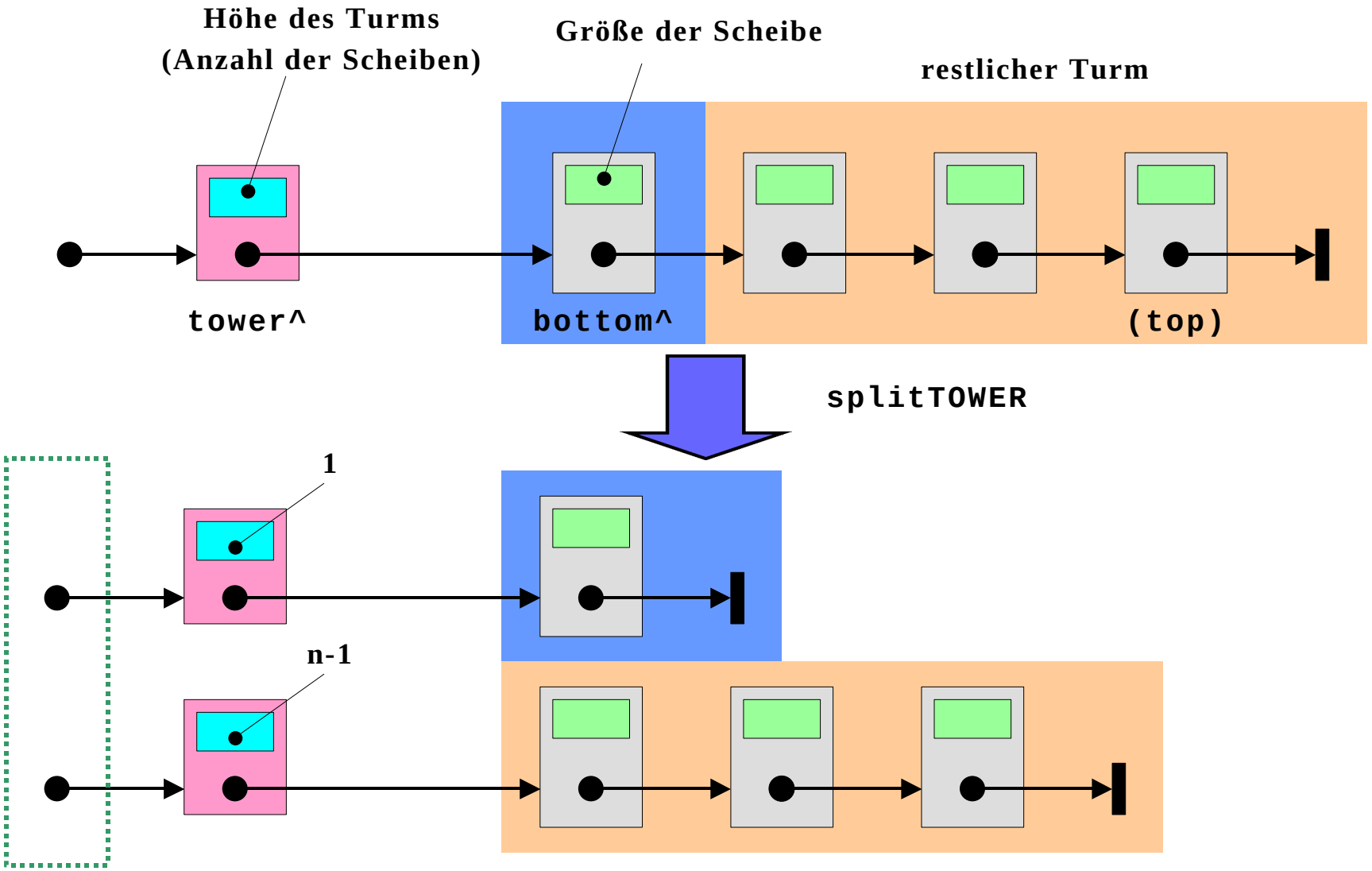
Turm mit Scheiben (Scheiben-Anzahl = Höhe)

Aufspaltung :

- **unterste Scheibe**
- **Turm mit Höhe - 1**

Zerlegung eines Turms in

- untere (= größte) Scheibe und
- restlicher Turm



Aufwandsabschätzung (rekursive Lösung)

Frage : *Wie oft müssen Scheiben hin- und hergetragen werden ?*

Ausgangssituation : Auf der kupfernen Säule befinden sich n Scheiben

Scheiben n	Trageoperationen
1	1
2	1 + 2 · 1 = 3
3	1 + 2 · 3 = 7
4	1 + 2 · 7 = 15
:	:
k	1 + 2 · Trageoperationen bei (k-1)

Vermutung : Bei Betrachtung der Zahlenfolge 1, 3, 7, 15, ... liegt die Vermutung nahe, daß bei k Scheiben $2^k - 1$ Trageoperationen notwendig sind !

Überprüfung : Trageoperationen bei k = 1 + 2 · Trageoperationen bei (k – 1)

$$2^k - 1 \stackrel{?}{=} 1 + 2 \cdot \left(2^{k-1} - 1\right)$$
$$\stackrel{?}{=} 1 + 2^k - 2$$
$$= 2^k - 1 \quad \blacksquare$$

⇒ Die Anzahl der Trageoperationen nimmt **exponentiell** mit n zu !

Zurück zum Anfang ...

Wenn der Mönch – so die Legende – seine Aufgabe erfüllt habe, so werde das Ende der Welt kommen.

Wenn alle Mönche für den Transport der **100 Scheiben** sehr fleißig arbeiten und **jede Minute eine Scheibe** transportieren, dann benötigen sie für den Transport des Turms

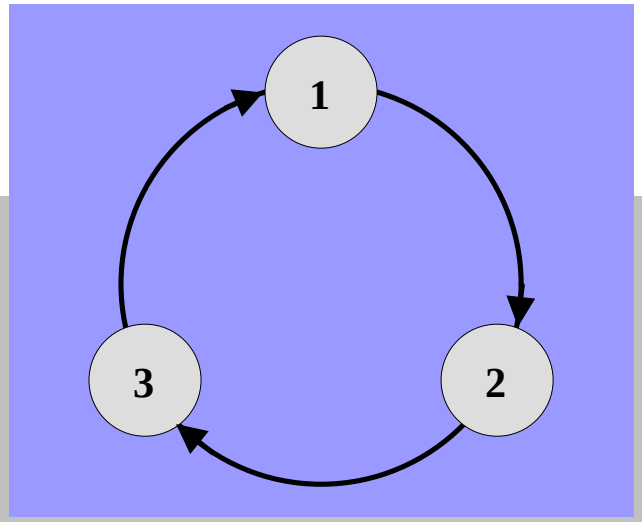
$$2^{100} - 1\,\text{min} \;.\approx 1.26765 \cdot 10^{30}\;\text{min}\;. \qquad (1\ \text{Jahr} = 525600\ \text{min.})$$

$$\approx 2.4 \cdot 10^{24}\ \textit{Jahre}\ (!)$$

... das Ende der Welt ist dann wahrscheinlich (in der Tat) nicht mehr fern !

Einfacher nicht-rekursiver Algorithmus

Struktur



→ Es wird **keine** „Ziel-Säule“ angegeben !

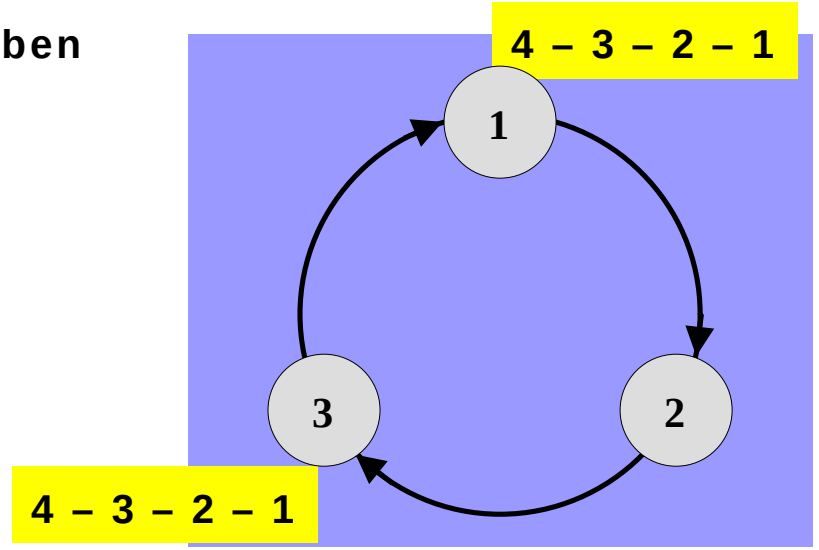
```
:
anfangsSaeule := 1;

WHILE < es sind noch nicht alle Scheiben korrekt
        auf einer anderen Säule gestapelt > DO

  < bewege die kleinste Scheibe (= oberste Scheibe der 'anfangsSaeule')
    um eine Position „im Uhrzeigersinn“ von Säule k → k' = (k mod 3) + 1
    (also 1 → 2 oder 2 → 3 oder 3 → 1) >;

  < bewege die nächst kleinere Scheibe (≠ kleinste Scheibe auf Säule k') von einer
    der verbleibenden Säulen c ≠ k' ∈ {1, 2, 3} „im Uhrzeigersinn“ auf die
    nächste mögliche Säule >
END;
:
```

(nach D. Harel. The science of computing. Addison-Wesley, 1989, p.107)

Ablauf

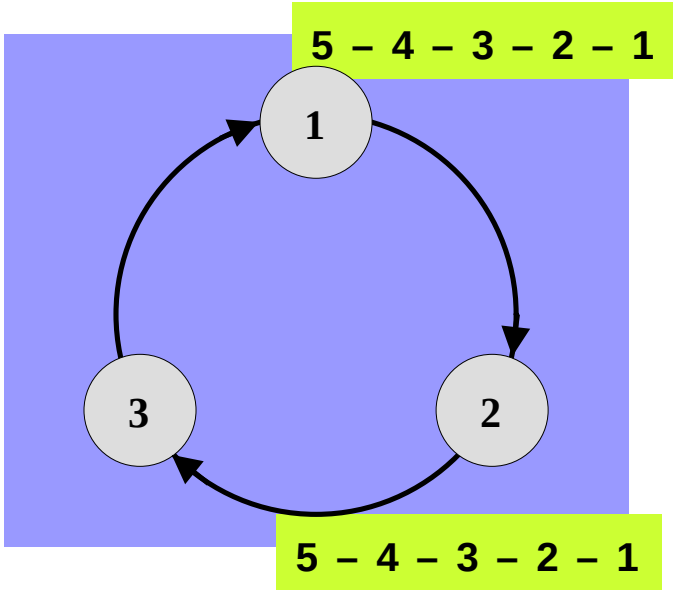
Transport einer <u>geraden</u> Anzahl von Scheiben

1
2
3
4



Transport einer <u>ungeraden</u> Anzahl von Scheiben

1
2
3
4
5



- **Rekursive** Funktionen / Prozeduren **rufen sich selbst auf** – direkt oder indirekt
- Zur Verwaltung der bei jedem Aufruf (**Inkarnation**) angelegten Variablen und übergebenen Parameter wird (vom System) ein **pulsierender Speicher** („**Stack**“) verwendet
- Rekursive Prozeduren lassen sich in nicht-rekursive Versionen überführen – der pulsierende Speicher muß dann explizit (oder implizit) selbst realisiert und verwaltet werden
- **Phasen** eines rekursiven Programms :
  Abstieg – **Ende** (Terminationsbedingung) – **Aufstieg**
  – beim Rekursionsaufstieg werden die Teilergebnisse zusammengefügt;
  die Programme rekursiver Prozeduren sind entsprechend dieser Phasen aufgebaut
- Verfahren nach dem Prinzip „**Teilen-und-Herrschen**“ zerlegen die **Datenmenge** in **etwa gleich große Teile** und wenden den **Algorithmus jeweils auf den Teilmengen** an;
  rekursive Aufrufe und zugehörige Operationen lassen sich als **Bäume** darstellen
- Das „**Türme-von-Hanoi**“-Problem stellt ein klassisches Beispiel dar, eine **Aufgabe rekursiv zu lösen**, eine gleichwertige iterative Lösung ist dagegen sehr viel schwerer durchschaubar;
  rekursive Lösungen dieser Art benötigen **exponentiellen Aufwand**