VIII. Rekursive Algorithmen

1. Struktur rekursiver Algorithmen
2. Verschiedene Rekursionsarten
3. Teilen-und-Herrschen
4. Türme von Hanoi

1. Struktur rekursiver Algorithmen

- Rekursive Probleme und Programme
- Struktur rekursiver Programme
- Rekursives Berechnungsformat und Termination

Rekursive Probleme und Programme

Motivation

- **Aufgabe**: Teile ein 1m langes Stück Holz in möglichst viele Stücke von jeweils genau 19 cm Länge

  Achtung: Man vergisst leicht, die Dicke des Sägeblatts zu beachten …

  - **Möglichkeit 1**: Miss die Dicke *d* des Sägeblatts und markiere die Holzlatte im Abstand

    $$p(k) = k * 19 \text{ cm} + (k - 1) * d$$

    vom linken Ende. Säge die Latte an den markierten Stellen durch, setze jeweils die linke Kante des Sägeblatts an die Markierung

  - **Möglichkeit 2**: Markiere die Latte im Abstand von 19 cm vom linken Ende und säge sie rechts der Markierung durch; wende das Verfahren auf das Reststück an, bis es kürzer als 19 cm ist

Die 2. Möglichkeit ist ein intuitiverer Lösungsweg!

(F.L. Bauer, G. Goos. Informatik – Eine einführende Übersicht, 4. Aufl. Springer, Berlin, 1992)

- In der **Mathematik** findet man häufig Funktionen, die sich auf sich selbst beziehen
  - Fakultät: $n\,! = \begin{cases} 1 & , \; n = 0 \\ n \cdot (n-1)! & , \; n \geq 1 \end{cases}$
  - Binomialkoeffizienten:

    Pascal'sches Dreieck – es gilt:

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

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

für $k > 0 \wedge k < n$

Bsp.: $\binom{6}{2} = \binom{5}{1} + \binom{5}{2}$

$$15 = 5 + 10$$

Werte der Binomialkoeffizienten $\binom{n}{k}$ (Pascal'sches Dreieck)

n													
0							1						
1						1		1					
2					1		2		1				
3				1		3		3		1			
4			1		4		6		4		1		
5		1		5		10		10		5		1	
6	1		6		15		20		15		6		1
:							...						

k = 0 → n

$$f(n,k) = \begin{cases} 1 & , \; k = 0 \vee k = n \\ f(n-1,k-1) + f(n-1,k) & , \; 0 < k < n \end{cases}$$

Struktur rekursiver Programme

Aufbau und Konstruktion rekursiver Algorithmen

- Typischer Aufbau einer rekursiven Methode

```
<result> recursiveProc(...) {

    ...
    if <Rekursionsende erreicht> {
        <Basisfall>
    }
    else {
        ...
        return recursiveProc(...);  // ein oder mehrere rekursive Aufrufe
        ...
    }
    ...
}
```

Hinweise:

- Die Bedingung <Rekursionsende erreicht> definiert die Termination der Rekursion
- Die Anweisung <Basisfall> legt das Ergebnis (Rückgabe) der Funktion für elementare Argumentwerte fest

Bsp.: Fakultäts-Funktion, fac(n) = n!

```
long fac(long n) {

    if (n == 0)
        return 1;
    else
        return n * fac(n-1);
}
```

Rekursionsende – Ergebnis bei Ende des Rekursions-Abstiegs

- Rekursion – rekursiver Abstieg
- Verknüpfung der Teilresultate

- **Erläuterungen**:
  - Zu Rekursion (*recurrere*): Ein "Zurückgehen" findet bei rekursiven Methoden in verschiedener Hinsicht statt:
    - Eine Methode ist rekursiv, wenn (im einfachsten Fall) in ihrem Rumpf ein Aufruf derselben Methode vorkommt – die Methode wird sozusagen durch **Rückverweis** auf sich selbst definiert
    - Der rekursive Aufruf im Rumpf setzt sich fort, bis eine Ende-Bedingung erfüllt ist. Das Ergebnis des Aufrufs kann im Allgemeinen erst nach Rückkehr vom rekursiven Aufruf berechnet werden
  - Die Rekursion stellt eine weitere Art von **Wiederholung** dar, ähnlich den Schleifen

- Rekursionen können direkt oder indirekt sein
  - **Direkte** Rekursion

```
<result> alpha() {

    ...
    alpha();
    ...
}
```

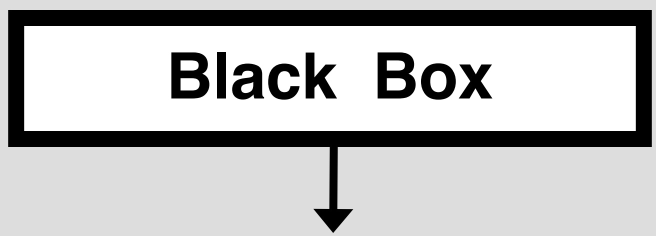
  - **Indirekte** Rekursion

```
<result> beta() {

    ...
    gamma();
    ...
}

<result> gamma() {

    ...
    beta();
    ...
}
```

- Allgemeine Hinweise zur Konstruktion rekursiver Algorithmen
  - Spezifikation der Aufgabe (wie bei anderen Projekten auch)

    „Schnittstellen-Beschreibung“: Identifikation der Rolle der Parameter

  - Konstruktion durch Fallunterscheidung
    - **Abbruchbedingung:** Berechnung des Wertes für den Fall der Beendigung der Rekursion (in diesem Abschnitt findet kein rekursiver Aufruf statt); diese Bedingung wird auch **Rekursionsverankerung** genannt
    - **Rekursiver Zweig** („Denkweise“): zu formulierende rekursive Prozedur existiert bereits als „*Black Box*“ wie eine Bibliothekslösung



Die Wirkung eindeutig durch Schnittstellen-Beschreibung gegeben !

- Allgemeine Überlegungen zur Vorgehensweise

Aufgabe: Berechne $n!$ ($n \geq 0$)

- *Wie sieht die Lösung des Problems für den einfachsten Fall aus?*

  Dies ist der **Basisfall** (**Rekursionsverankerung**)

Basisfall: $n = 0$

$0! = 1$

- *Wie kann das Problem so in gleichartige Teilprobleme zerlegt werden, dass die Lösung des Problems durch Kombination der Lösungen der einzelnen Teilprobleme konstruiert werden kann?*

  Dies ist der **Rekursionsfall**

Rekursionsfall: $n > 0$

$n! = n * (n - 1)!$

  Wie sind im Rekursionsfall die Teillösungen zu kombinieren?

- *Wie manifestiert sich die Reduktion auf „einfachere“ Teilprobleme?*

  Problem der **Termination**

```
long fac(long n) {
    if (n == 0)
        return 1;
    else
        return n * fac(n-1);
}
```

Phasen der Verarbeitung eines rekursiven Programms

- Beispiel der rekursiven Berechnung der Fakultäts-Funktion: Berechnung von 4!

„Black box"-Sicht

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

Abstieg

Rekursions-Ende

Abbruch-bedingung

Aufstieg

Ergebnis

- **Erläuterung**: Es gibt 2 Phasen – der **Abstieg** mit den rekursiven Aufrufen der Funktion (Methode) und der anschließende **Aufstieg** (die Phase des rekursiven Abstiegs wird durch das Rekursionsende (Termination) mit der Rekursionsverankerung abgeschlossen und damit der Aufstieg eingeleitet)

- Andere Darstellung …

```
long fac(long n) {
    if (n == 0)
        return 1;
    else
        return n * fac(n-1);
}
```

Sicht der Zustandsebenen

```
fac(4)
= 4 * fac(4-1)                                    4 * 6 → 24
        |
        fac(3)
        = 3 * fac(3-1)                        3 * 2 → 6
                |
                fac(2)
                = 2 * fac(2-1)            2 * 1 → 2
                        |
                        fac(1)
                        = 1 * fac(1-1)  1 * 1 → 1
                                |
                                fac(0)  ==   1
```

Programmzustands-Ebenen

Rekursives Berechnungsformat und Termination

Verarbeitung rekursiver Algorithmen mittels „Formular-Maschine“

Einordnung

Literatur: F.L. Bauer, G. Goos. Informatik – Eine einführende Übersicht, erster Teil, 4. Aufl. Springer-Verlag, Berlin, 1992

- Die Berechnungen eines Programms können in Form eines Berechnungs-Formulars notiert werden, das bei jedem Aufruf neu angelegt wird
- Für einen rekursiv definierten Algorithmus kann man ein derartiges Berechnungs-Formular aufstellen, das für jeden rekursiven Aufruf (**Inkarnation**) vervielfältigt wird
- Das Ergebnis der Berechnung wird durch Ein- und Rückübertragung von (Teil-) Lösungen bestimmt

Beobachtungen zum Ablauf eines rekursiven Programms

- Während des Berechnungsablaufs wird eine Methode immer wieder rekursiv **mit jeweils kleinerem Argument** aufgerufen

  Wichtig: Termination des Programms durch Reduzierung des Argument-Wertes
- Dabei werden beim **Abstieg** verschiedene **Inkarnationen** der jeweiligen Methode erzeugt – deren Anzahl hängt vom Algorithmus und vom Parameterwert ab

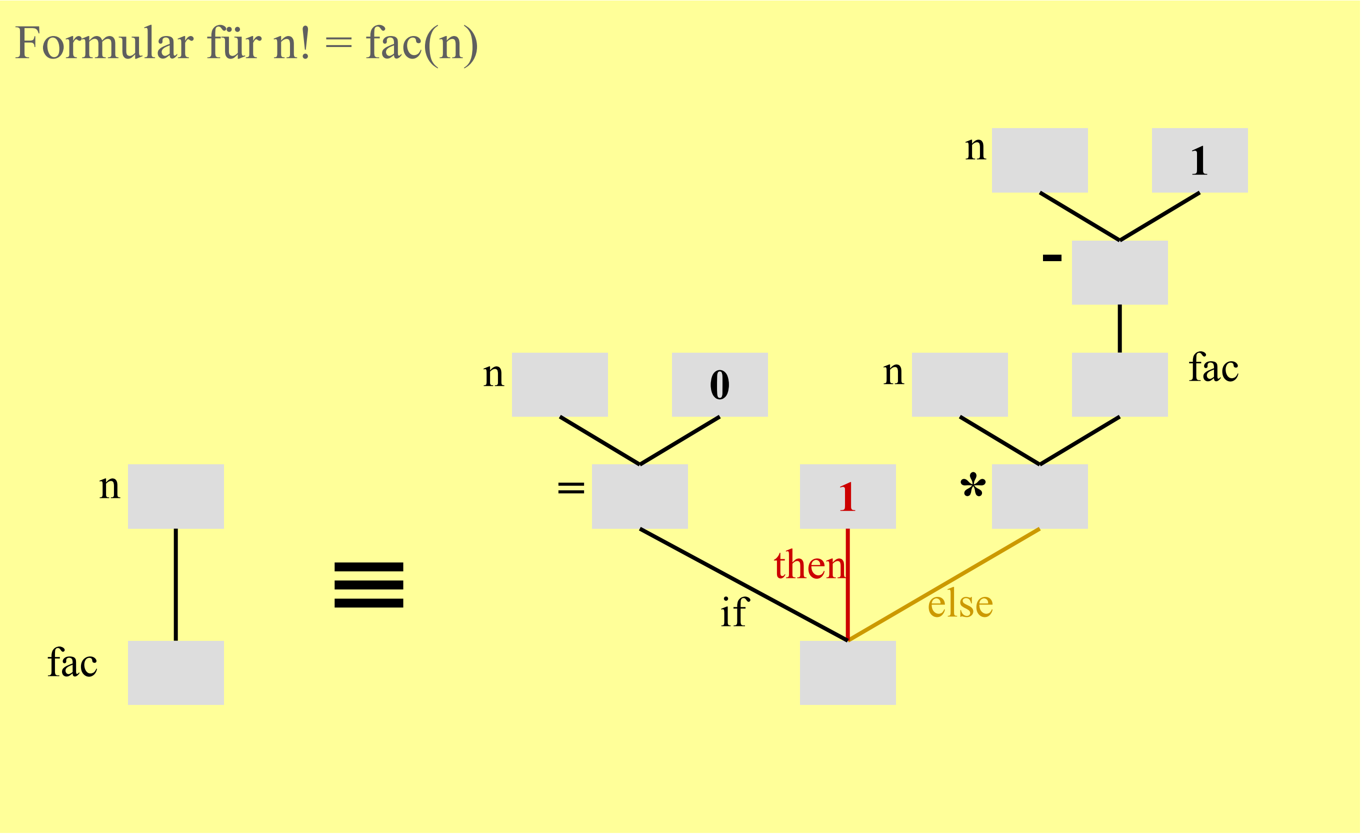
- Vorgehensweise bei der Berechnung mittels „Formularmaschine“:

1. Erstelle ein Berechnungs-**Formular** (**Algorithmus**) für das rekursive Problem
2. Übertrage die Argumentwerte in das Formular
3. Werte die Bedingung für die Fallunterscheidung aus
4. Wähle den zutreffenden Zweig der Fallunterscheidung
5. Werte nur den gewählten Zweig aus
6. Bei (nicht direkt auswertbaren) rekursiven Aufrufen: neues Formular über das aktuelle legen („stapeln“); weiter bei (2)
7. Sind die relevanten Formularteile vollständig ausgefüllt, so ist dieses zugleich wegzuwerfen und dessen Ergebnis auf das ggf. darunter liegende Formular zu übertragen; weiter bei (5), es sei denn: Das vollständig ausgefüllte Formular ist das letzte; eine Ergebnis-Übertragung ist dann nicht (mehr) möglich (**Termination**)

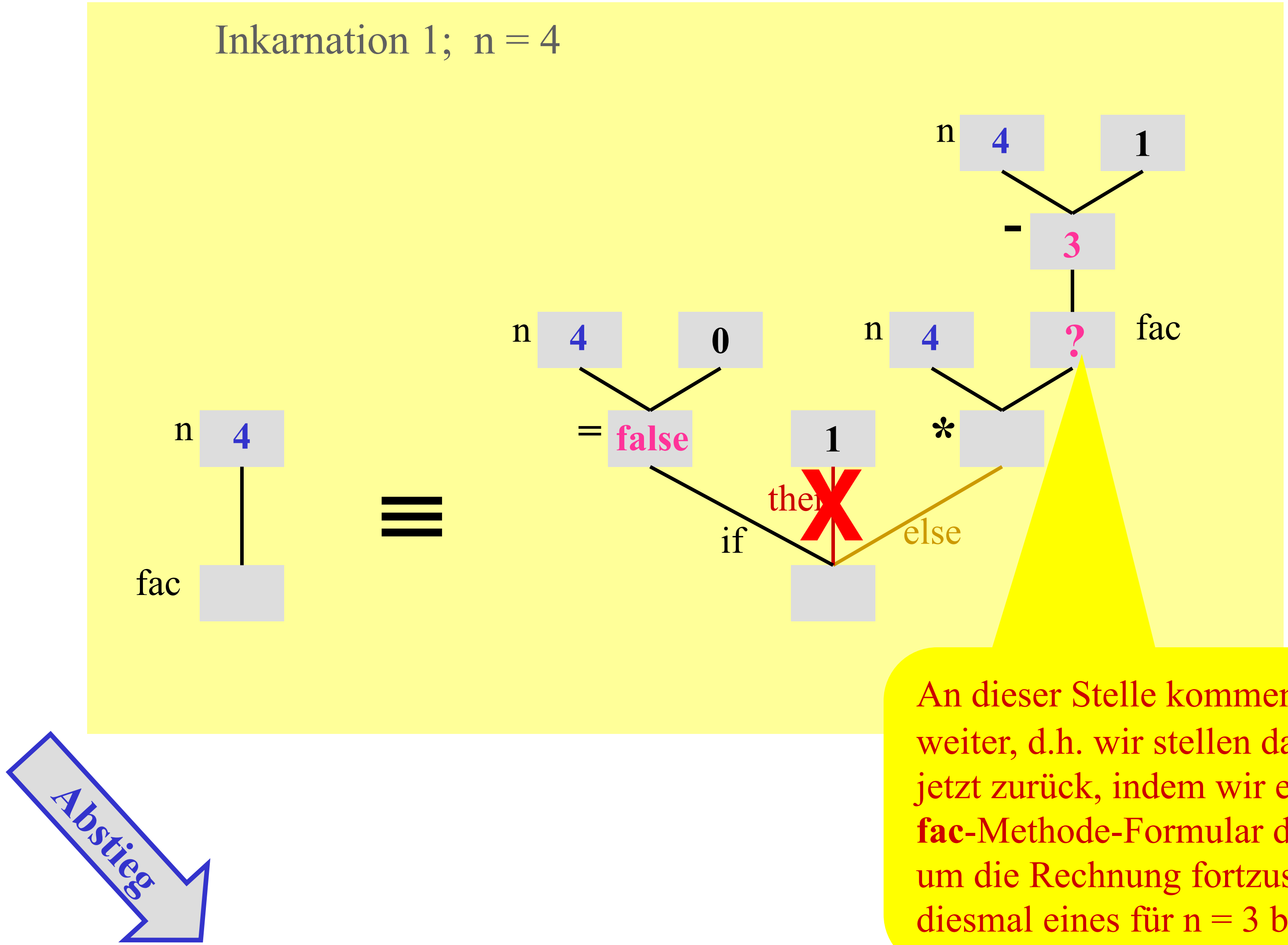
$\Rightarrow$ das Resultat ist das gesuchte Endergebnis

- **Ablauf – Beispiel Fakultäts-Funktion**:

  Schritt 1.: Erstelle ein **Formular** für das rekursive Problem

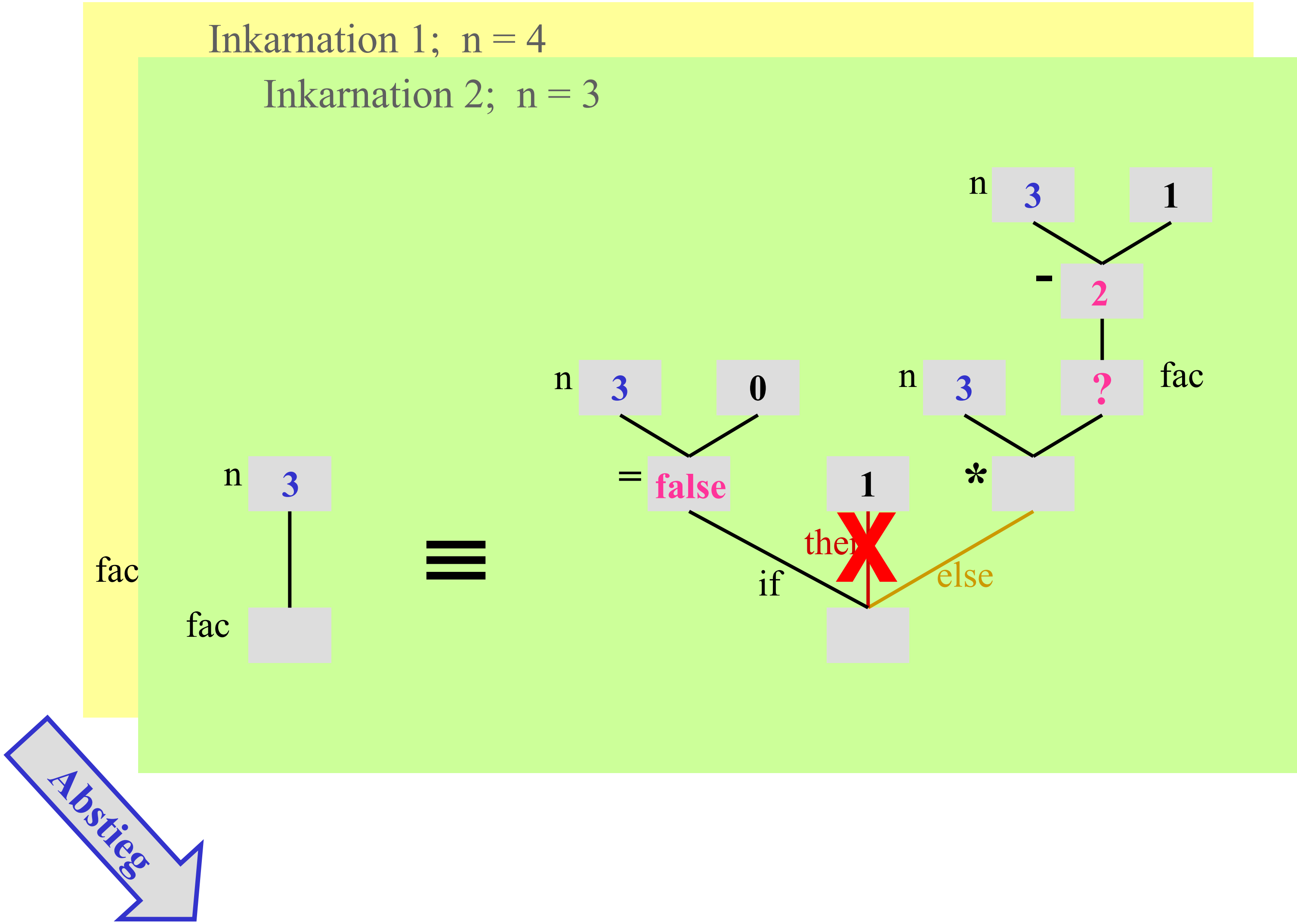


Schritte 2. bis 5.

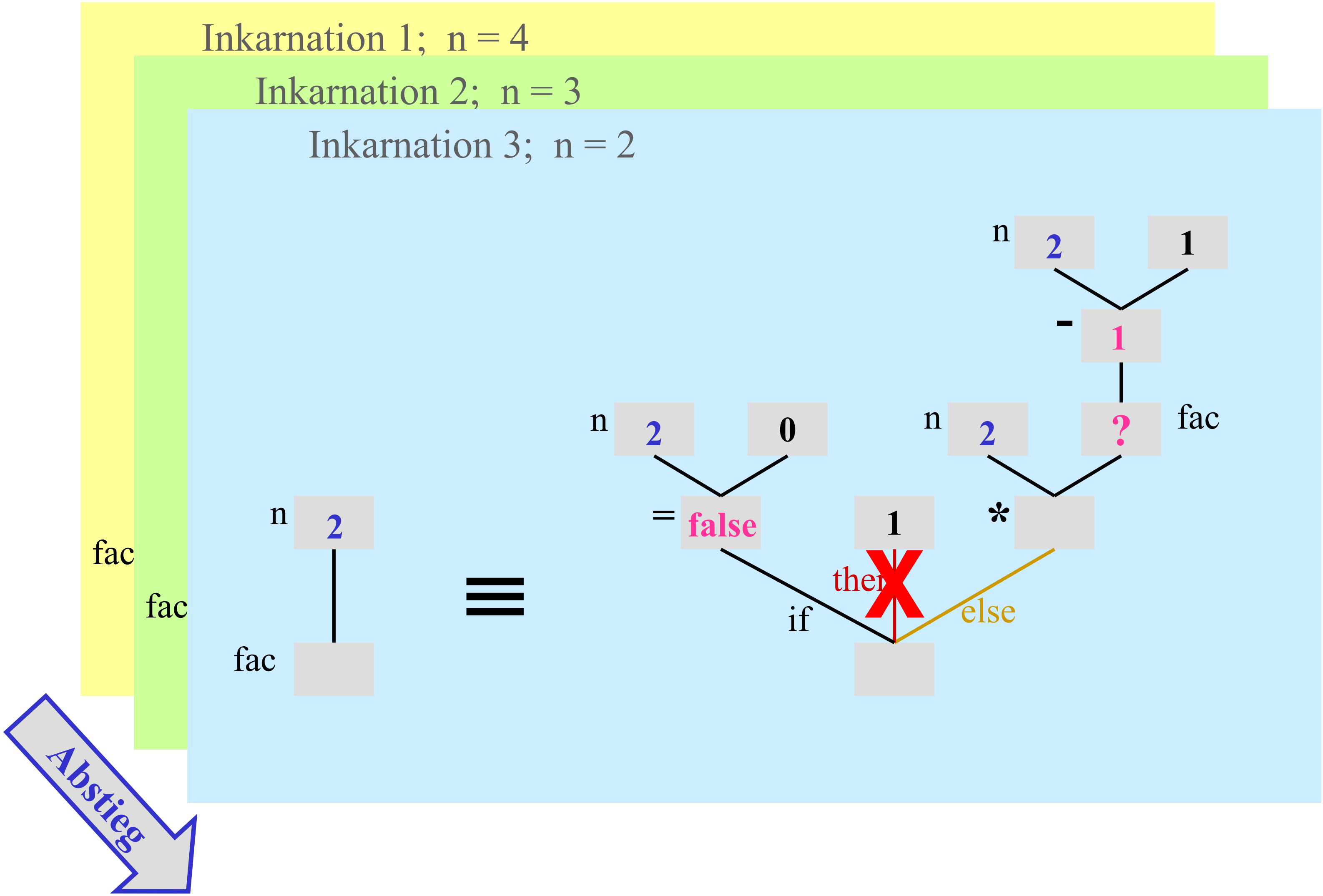


An dieser Stelle kommen wir nicht weiter, d.h. wir stellen das Formular jetzt zurück, indem wir ein weiteres **fac**-Methode-Formular darüber legen, um die Rechnung fortzusetzen; diesmal eines für n = 3 bzw. fac(3)

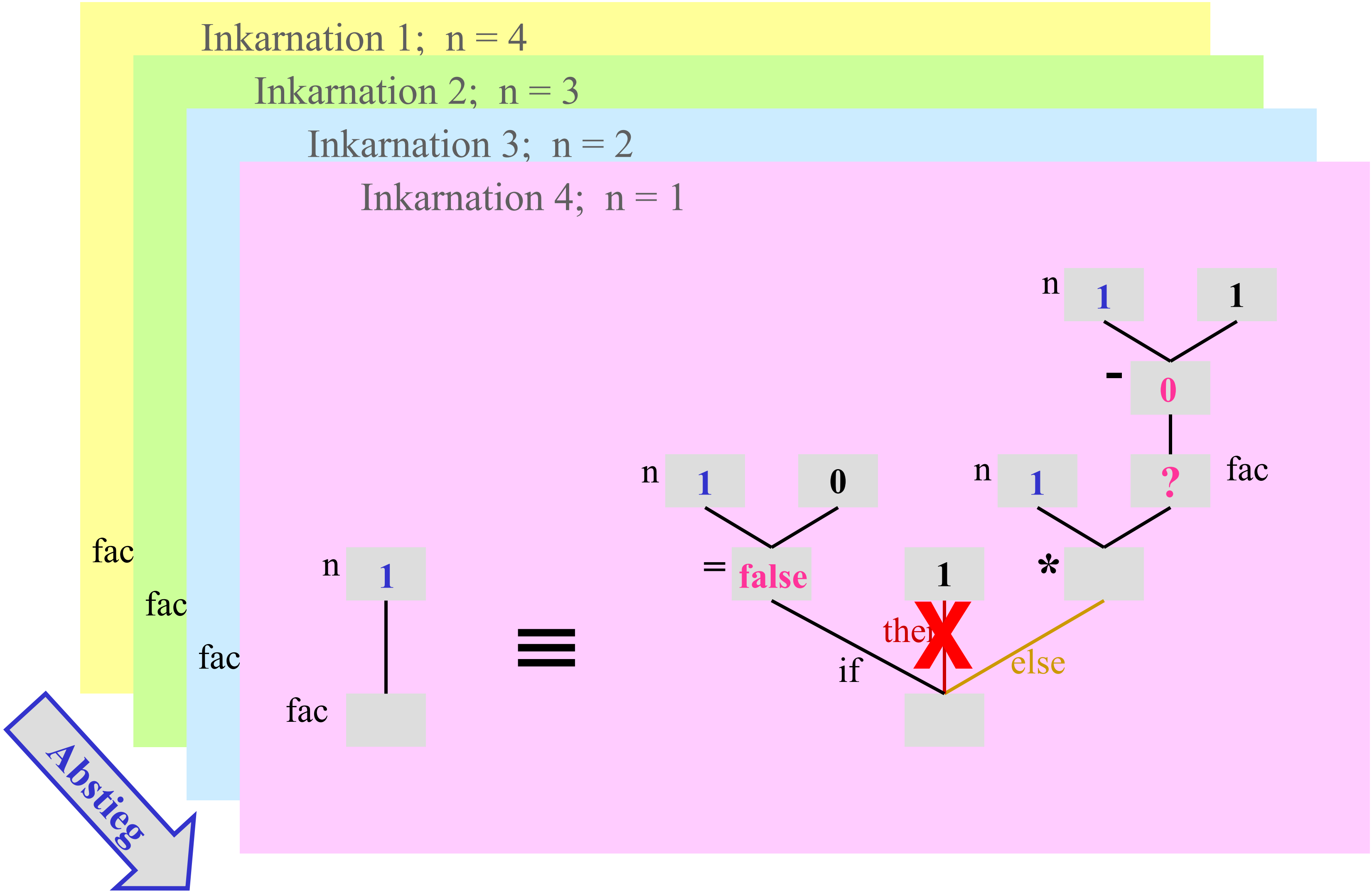
Schritte 6., 2. bis 5.



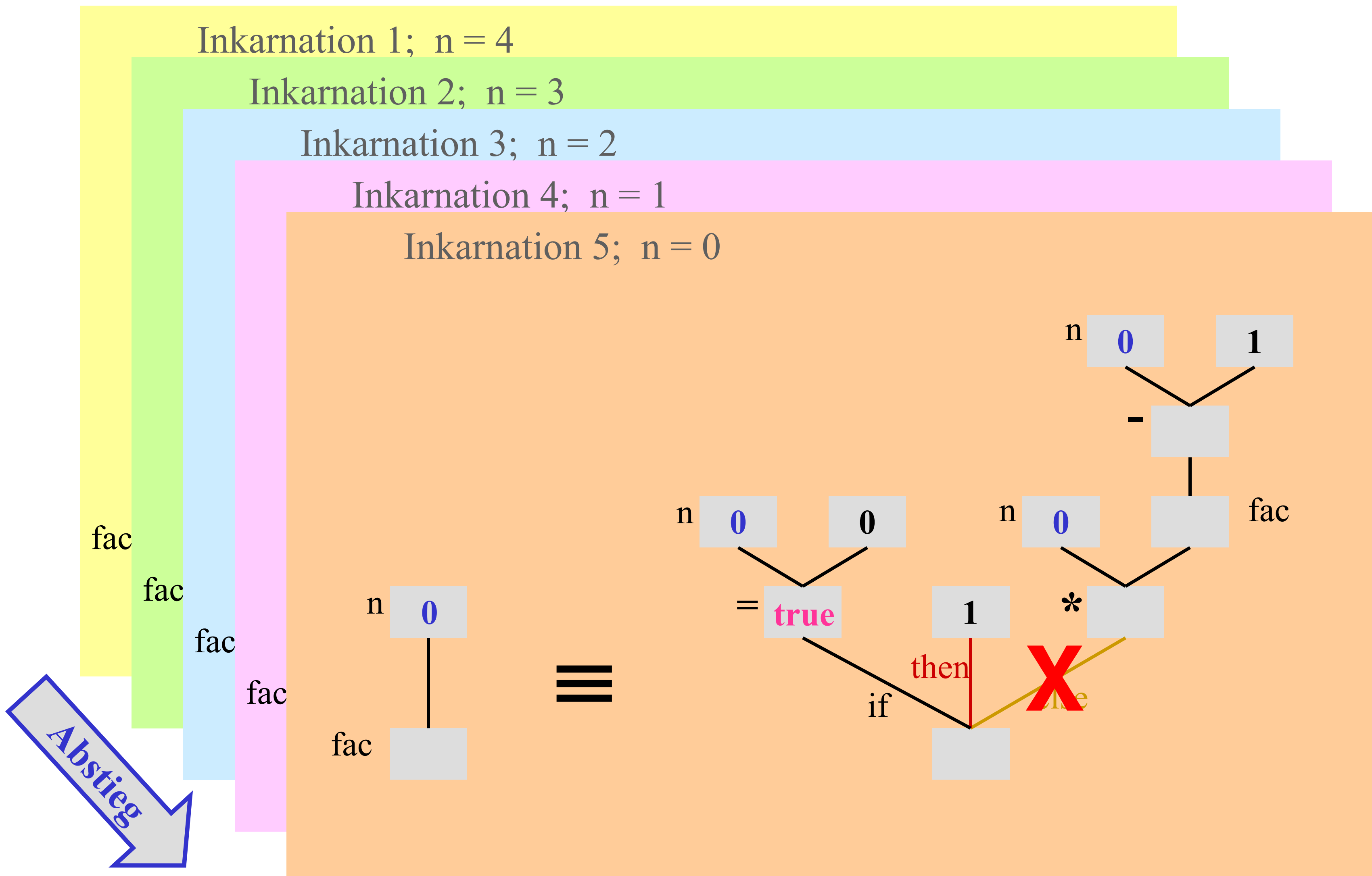
Schritte 6., 2. bis 5.



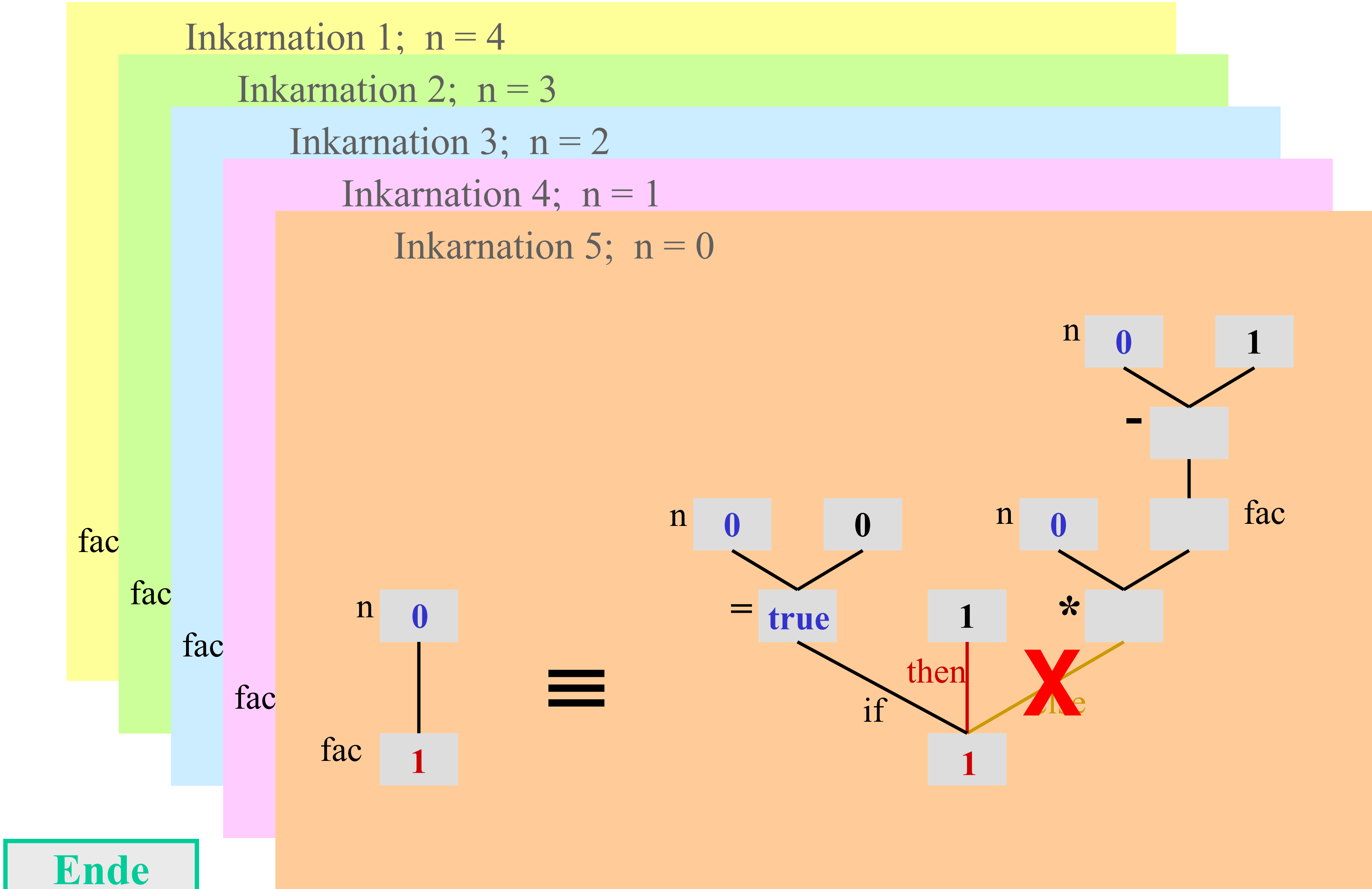
Schritte 6., 2. bis 5.



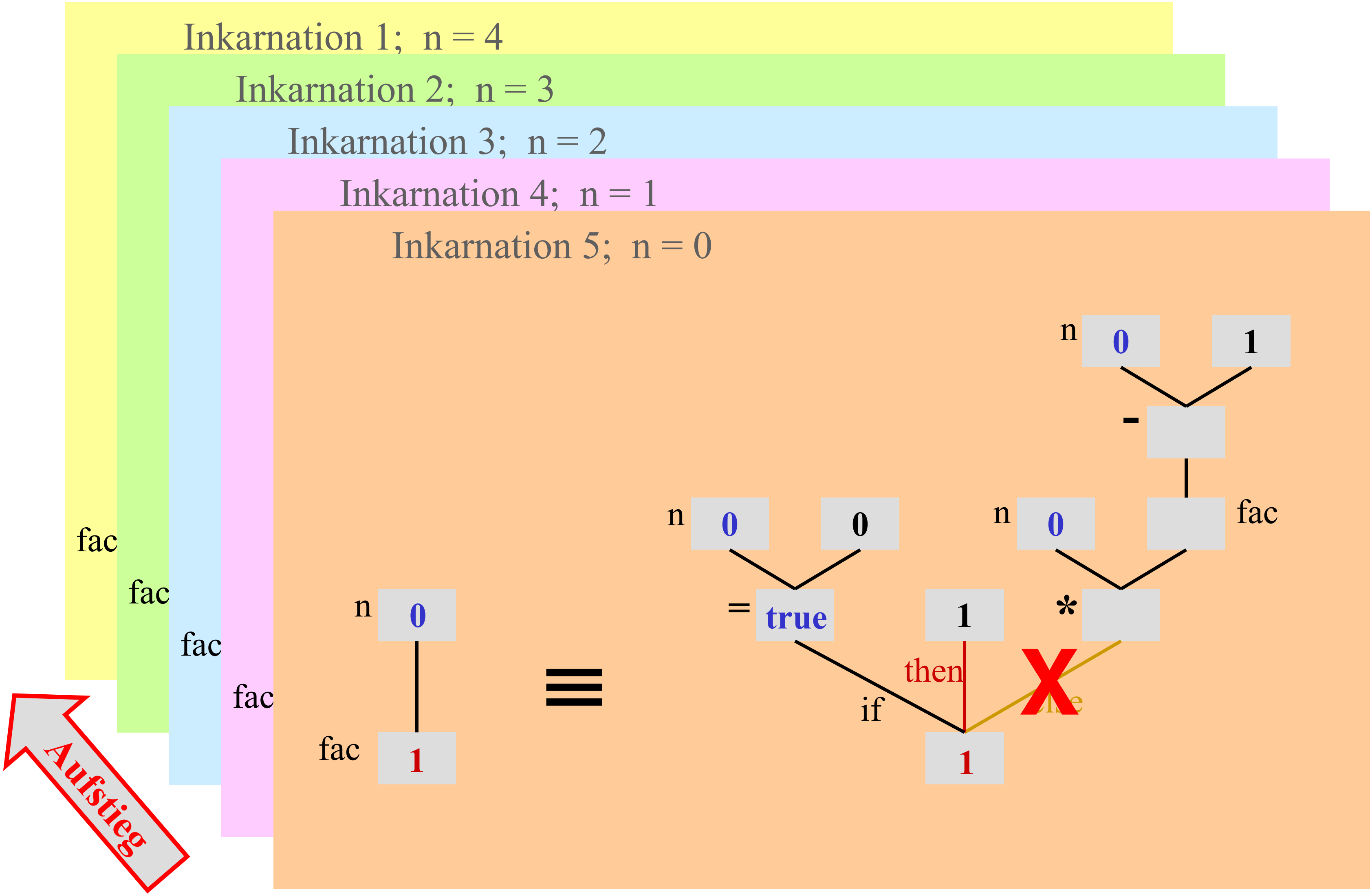
Schritte 6., 2. bis 5.



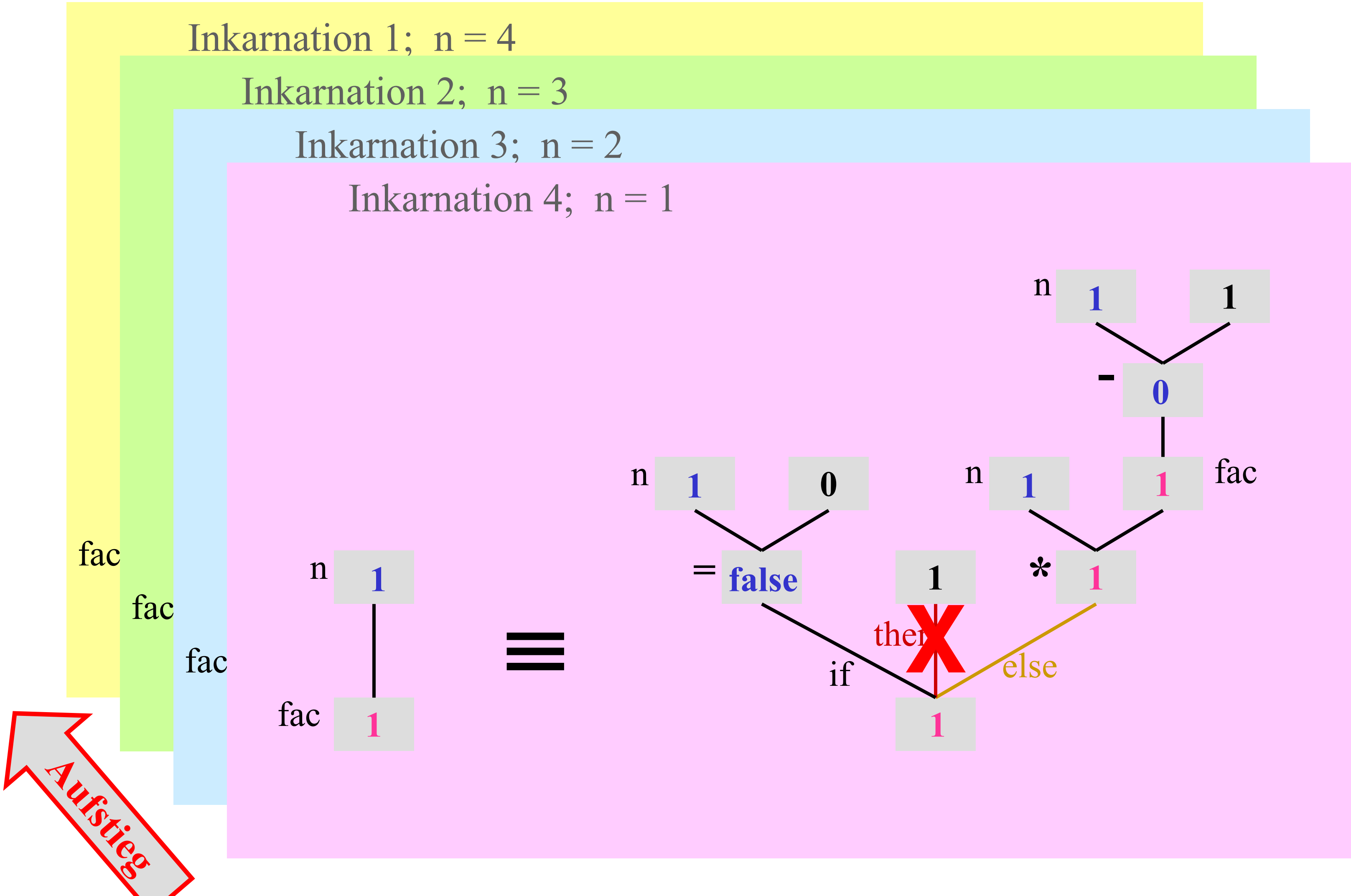
Ende des Abstiegs …



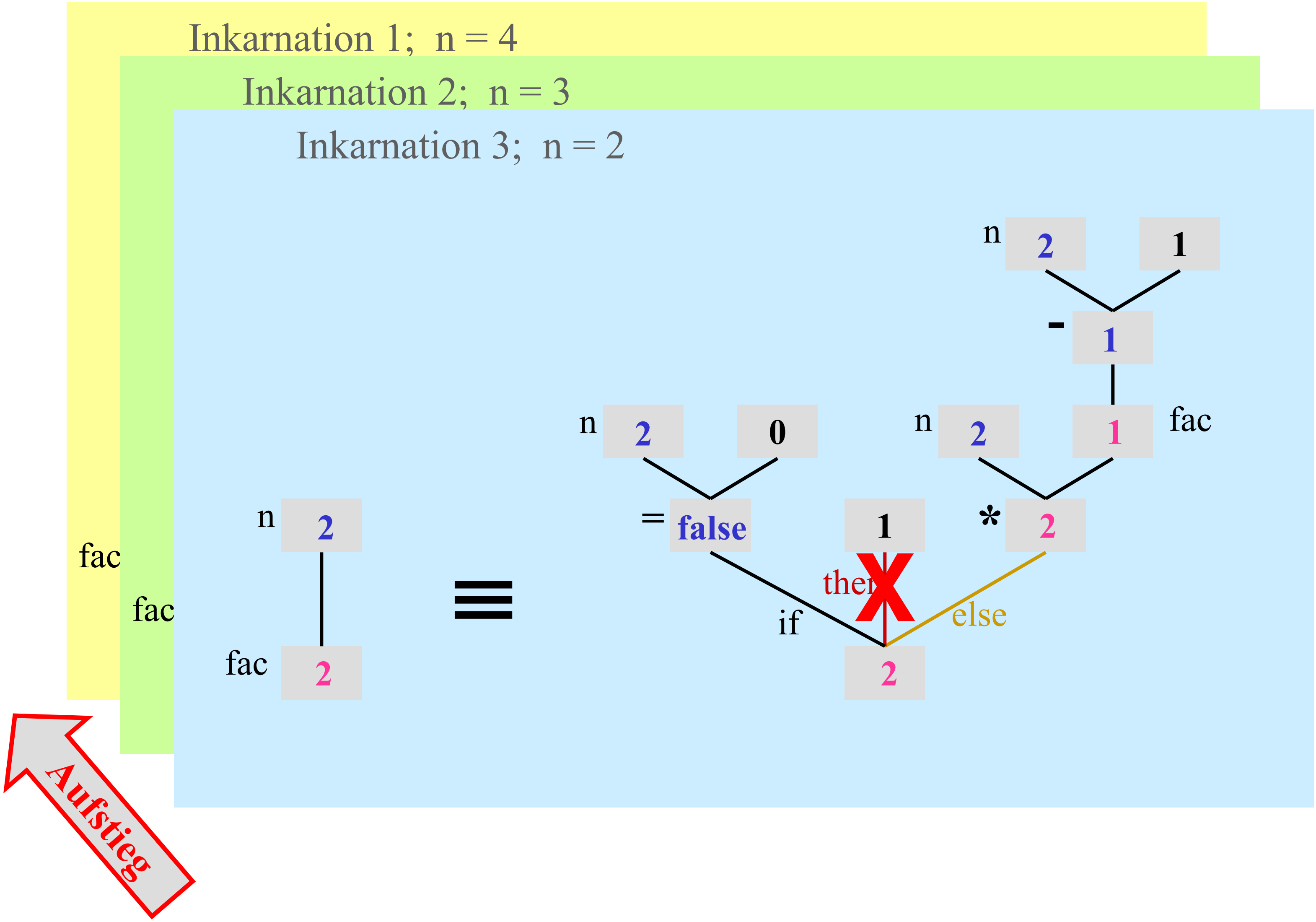
Schritt 7.


Inkarnation 1; n = 4
Inkarnation 2; n = 3
Inkarnation 3; n = 2
Inkarnation 4; n = 1
Inkarnation 5; n = 0
fac
fac
fac
fac
Aufstieg
n
0
fac
1
≡
n
0
0
=
true
1
then
if
1
n
0
1
-
n
0
fac
*
X

Schritte 5., 7.

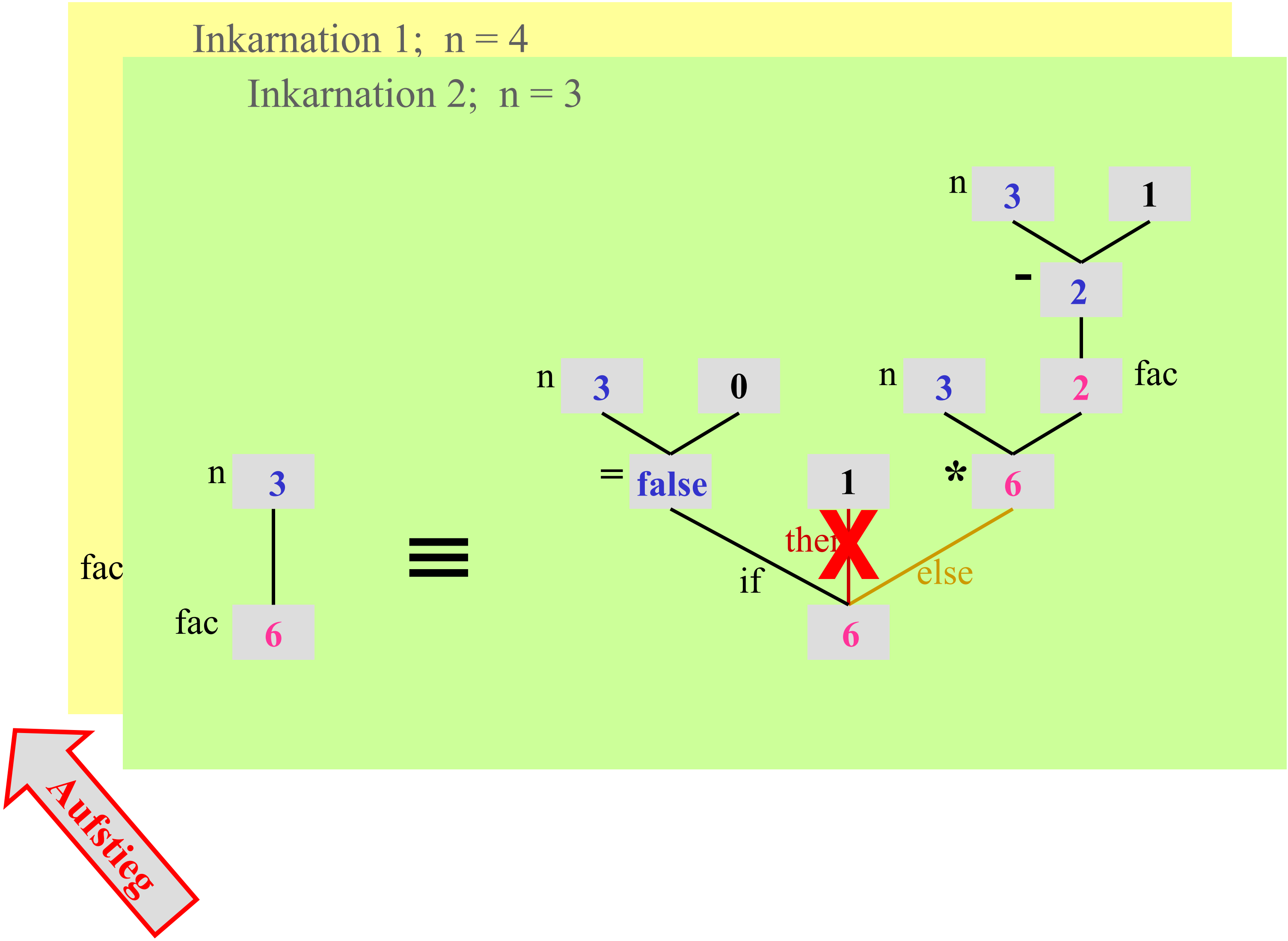

Inkarnation 1; n = 4
Inkarnation 2; n = 3
Inkarnation 3; n = 2
Inkarnation 4; n = 1
fac
fac
fac
n
1
fac
1
≡
n
1
0
=
false
1
then
else
if
1
n
1
1
-
0
n
1
fac
1
*
1
Aufstieg

Schritte 5., 7.

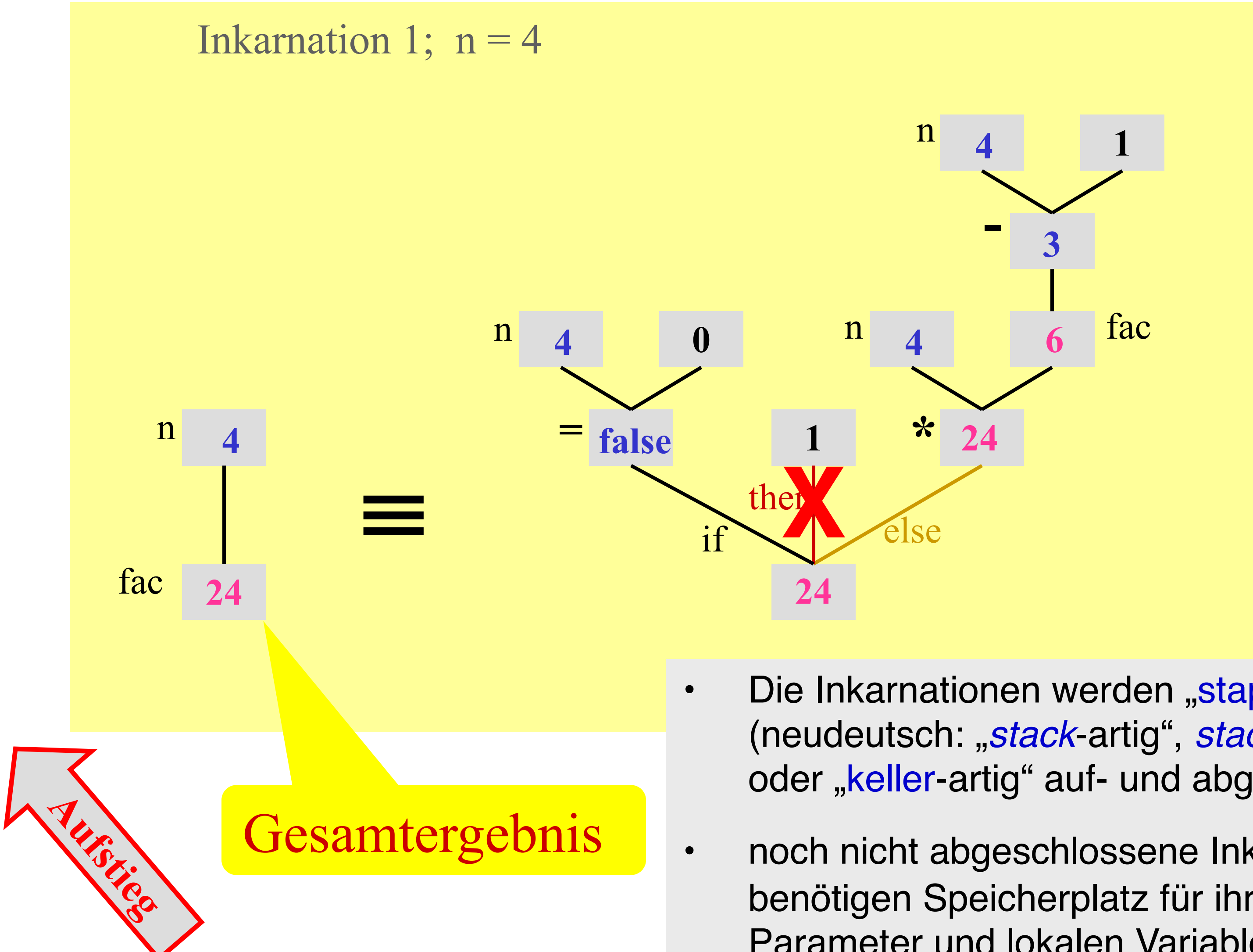


Inkarnation 1; n = 4
Inkarnation 2; n = 3
Inkarnation 3; n = 2
n
2
1
-
1
fac
1
n
2
0
n
2
=
false
1
*
2
then
else
if
2
n
2
fac
2
fac
fac
≡
Aufstieg

Schritte 5., 7.


Inkarnation 1; n = 4
Inkarnation 2; n = 3
n
3
1
-
2
fac
n
3
0
n
3
2
=
false
1
*
6
n
3
then
else
if
fac
fac
6
6
Aufstieg

Schritte 5., 7.



- Die Inkarnationen werden „stapel-artig“ (neudeutsch: „*stack*-artig“, *stack* = Stapel) oder „keller-artig“ auf- und abgebaut
- noch nicht abgeschlossene Inkarnationen benötigen Speicherplatz für ihre Parameter und lokalen Variablen

Beobachtungen zum Ablauf der rekursiven Fakultäts-Berechnung

- Während des Berechnungsablaufs wird die Methode `fac` immer wieder (rekursiv) **mit jeweils kleinerem Argument** (**Parameter**) aufgerufen

  Wichtig:
  - Die Reduzierung des Argument-Wertes ist wichtig für die Termination des rekursiven Programms
  - Diese Ablaufstruktur ähnelt der einer Wiederholungsschleife

- Dabei werden beim **Abstieg** verschiedene **Inkarnationen** der jeweiligen Methode erzeugt – deren Anzahl (hier: 5) hängt vom Algorithmus und vom ursprünglichen Parameterwert ab

- Das **Rekursionsende** wird durch die Abbruchbedingung `fac(0) == 1` definiert

- Während des **Aufstiegs** werden die Zwischenergebnisse verknüpft und die Inkarnationen geschlossen und zerstört

- Bei jeder Inkarnation werden neue lokale Variablen erzeugt und mit eigenen Werten belegt; beim Aufstieg werden diese Variablen beim Abschluss der Ausführung der jeweils aufgerufenen Methode wieder abgebaut

- Der momentane Zustand der Methode wird auf einem Systemspeicherbereich (System-*Stack*) gesichert und später wieder geladen

Details zu den Inkarnationen einer Funktion und lokalen Variablen

- Bei der Abarbeitung rekursiver Methoden existieren zu einem Zeitpunkt in der Regel mehrere verschiedene Inkarnationen dieser Methode
- Jeder rekursive Aufruf einer Methode hat seinen eigenen Speicherbereich und damit seinen eigenen Satz von lokalen Variablen und Parametern
  - Jede Inkarnation kann jeweils nur genau die zu ihr gehörenden Parameter und Variablen sehen – das entspricht dem aktuellen „Formular“ des rekursiven Aufrufs
  - Nach der Rückkehr aus diesem Aufruf stehen die alten Werte der lokalen Variablen und Parameter wieder zur Verfügung (die alten Werte werden beim neuen Aufruf der Methode (Funktion) in einem speziellen Speicherbereich, dem sog. System-*Stack* gespeichert und beim Rücksprung von dort wieder geladen; Technische Informatik) – *Stacks* sind dynamische Datenstrukturen; Details später
- natürlich: Alle rekursiven Aufrufe einer Methode teilen sich die Klassenvariablen der umgebenden Klasse

Bsp.: Rekursiver Aufruf der Fakultäts-Funktion mit globalen Klassenvariablen zur Visualisierung der Inkarnation (`long i`) und der Zwischenwerte in einem Feld `a`

Algorithmus

```
static long[] a = new long[4];
static long   i = 0;

public long fac(long n) {

 long f = 1;
  if (n > 0)
    f = n * fac(n-1);

  a[i++] = f;
  return f;
}
```

Inkarnationen

Aufruf: `fac(3) = 6`

```
a:      [1, 1, 2, 6]
i: 4
  fac(3):
    n: 3
    f: 6
      fac(2):
        n: 2
        f: 2
          fac(1):
            n: 1
            f: 1
              fac(0):
                n: 0
                f: 1
```

2. Verschiedene Rekursionsarten

- Primitiv-rekursive Funktionen und lineare Rekursion
- Baumartige (kaskadenartige) Rekursion
- Geschachtelte und verschränkte Rekursion

Primitiv-rekursive Funktionen und lineare Rekursion

Eigenschaften

- **Primitiv-rekursive Funktionen**:
  - Die Grundfunktion G hat die Rekursionstiefe 0
  - Die Funktionen sind für alle $x \in \mathbf{IN}_0$ definiert (totale Definition)
  - Sie beruhen auf der induktiven Definition natürlicher Zahlen

  Bsp.: Primitiv-rekursive Addition, add: $\mathbf{IN}_0 \times \mathbf{IN}_0 \rightarrow \mathbf{IN}_0$

G	:	add(x, 0) = x			
Rekursion	:	add(x, y+1)	= x + y + 1	= add(x, y) + 1	oder
		add(x, y)	= x + y – 1 + 1	= add(x, y–1) + 1	

- **Lineare Rekursion**: Häufigste Rekursionsform, bei der höchstens ein weiterer rekursiver Aufruf vorkommen darf; die Berechnung erfolgt entlang einer Kette von Aufrufen

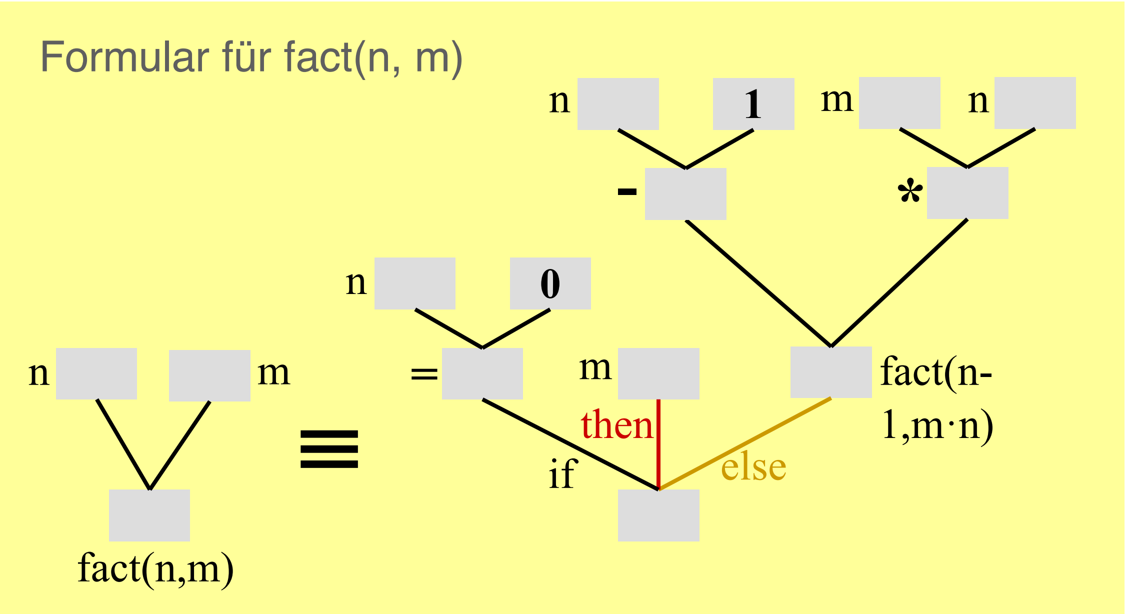
- **Kopf- und End-Rekursion**
  - **Kopf-Rekursion** (*head recursion*): rekursiver Aufruf am Anfang der Funktion
  - **End-Rekursion** (*tail recursion*): rekursiver Aufruf am Ende der Funktion

- Die **Termination der End-Rekursion** liefert bereits das Gesamtergebnis, d.h. de facto findet in diesem Fall nur ein rekursiver Abstieg statt; der Aufstieg besteht lediglich in der Schließung der Inkarnationen und dem Übertrag der (unveränderten) Ergebnisse

<u>Bsp</u>.: *fac* ist nicht end-rekursiv, aber folgende Verallgemeinerung liefert einen end-rekursiven Algorithmus:

$$fact(n,m) = \begin{cases} m & , \; n = 0 \\ fact(n-1, m \cdot n) & , \; n \geq 1 \end{cases}$$

Spezialfall: *fact*(*n*, 1) = *n*! – allgemein gilt: *fact*(*n*, *m*) = *m* □ *n*!

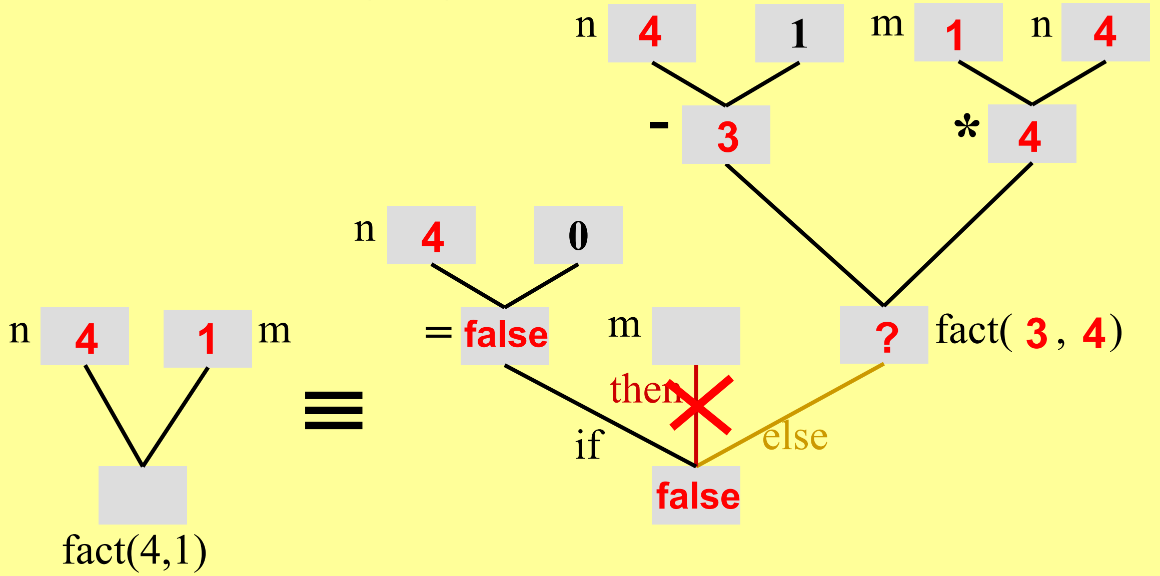



```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```

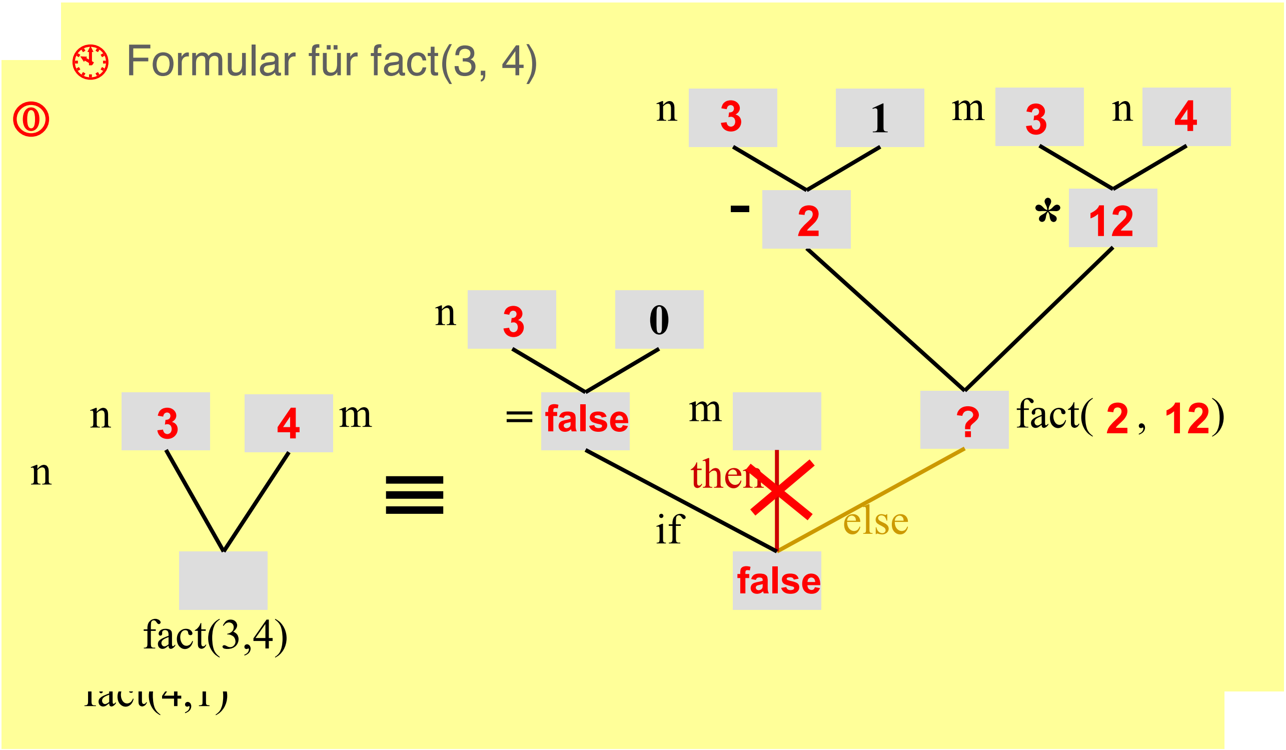
fact(4, 1) = fact(3, 4)
= fact(2, 12)
= fact(1, 24)
= fact(0, 24) = **24**

```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```

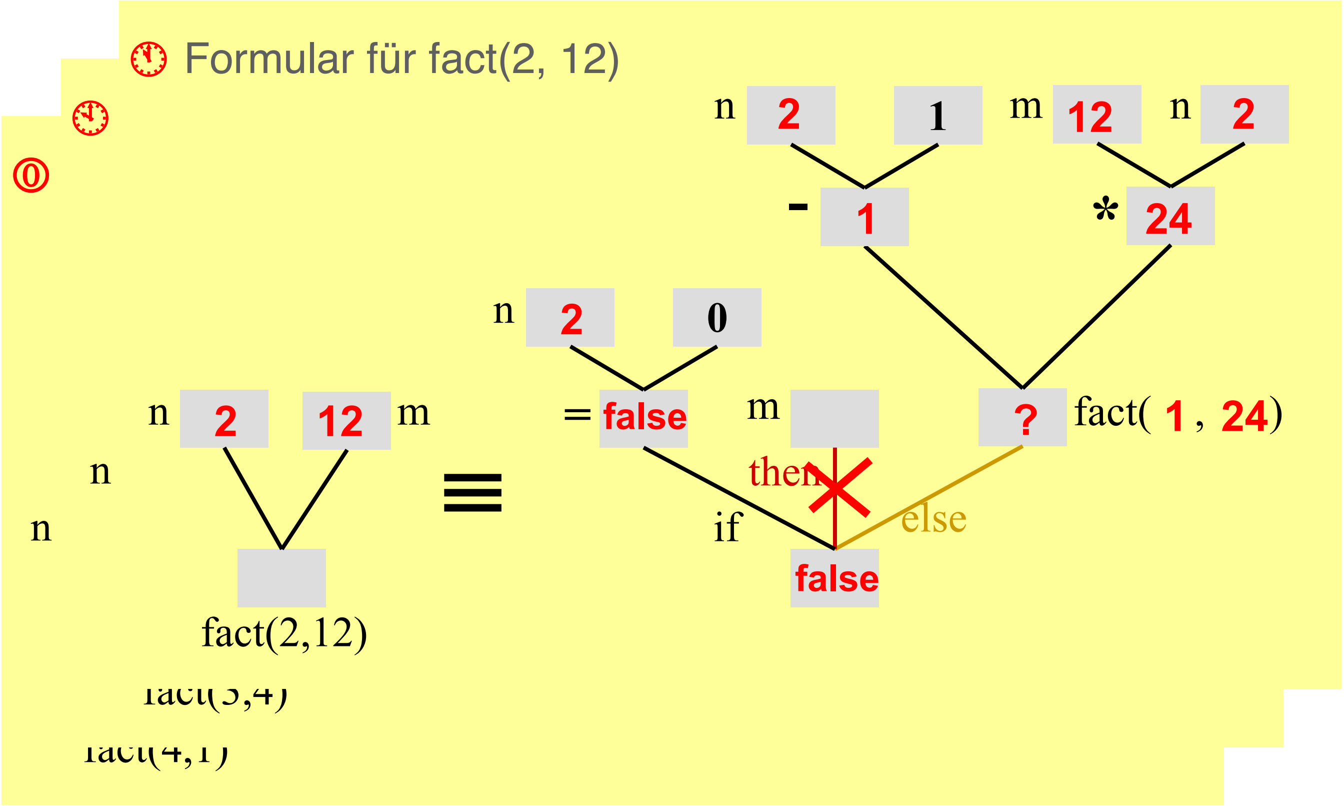
⓪ Formular für fact(4, 1)



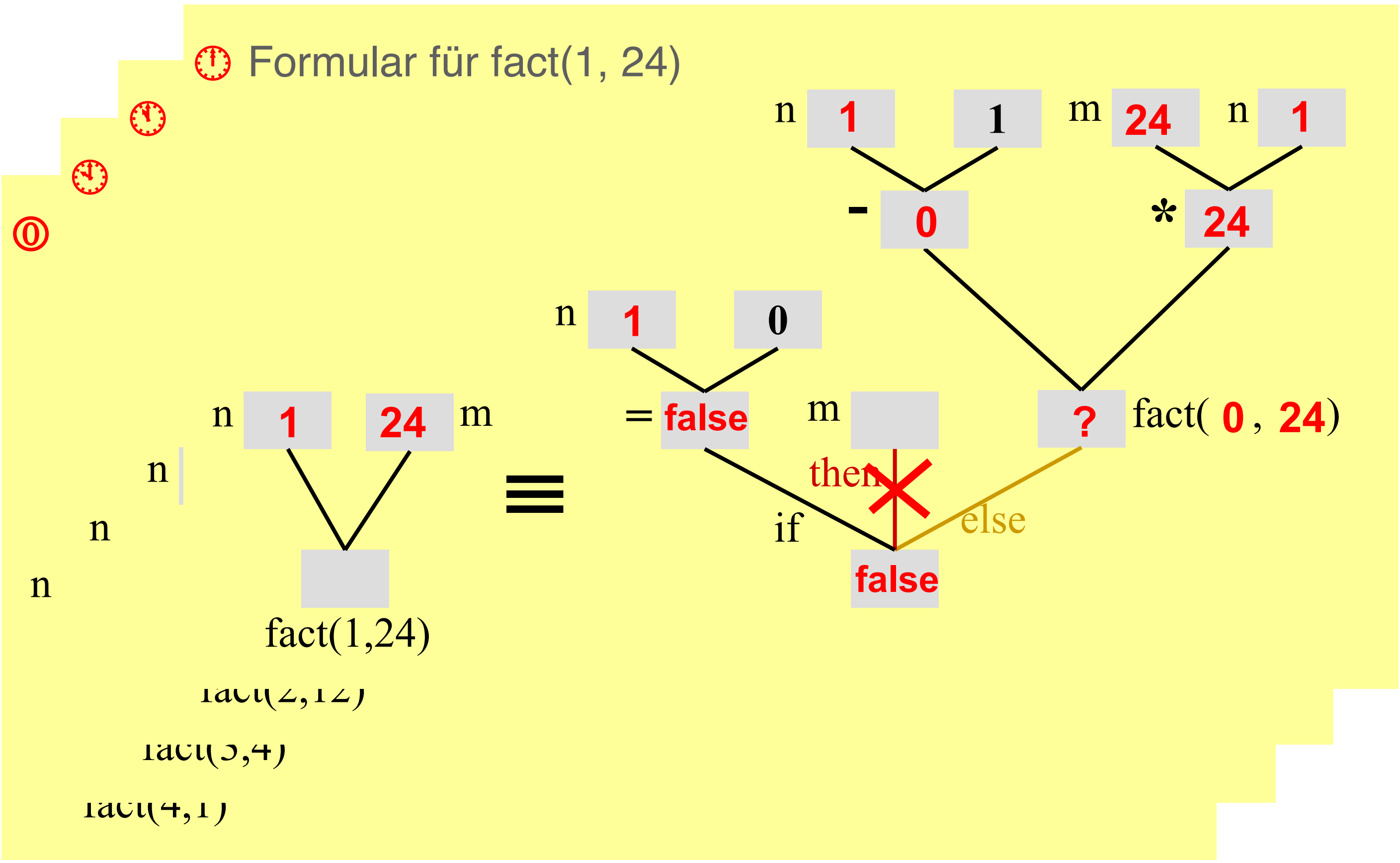
```
long fact(long n, long m) {

   if (n == 0)
      return m;
   else
      return fact(n-1, m*n);
}
```



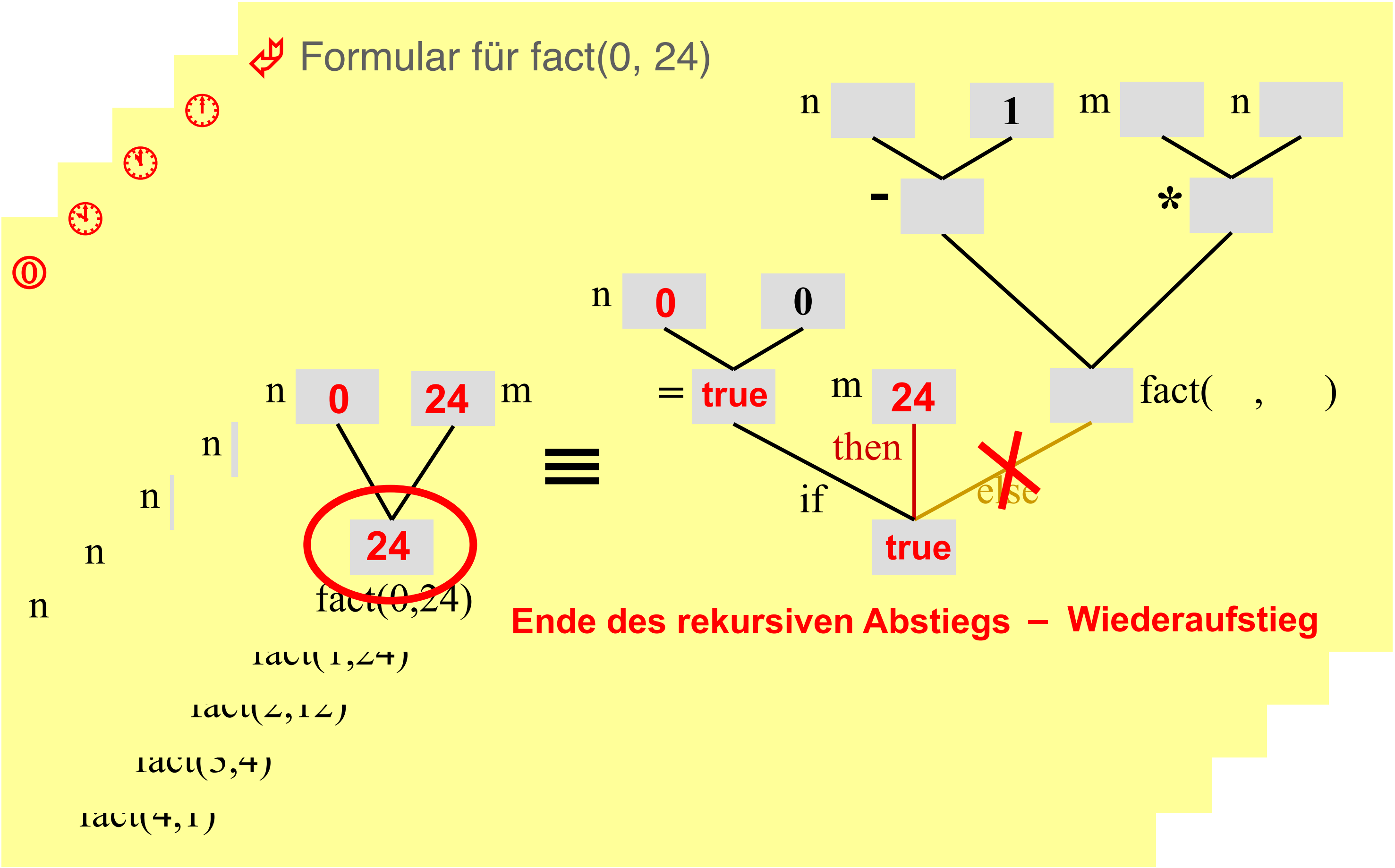
```
long fact(long n, long m) {

   if (n == 0)
      return m;
   else
      return fact(n-1, m*n);
}
```



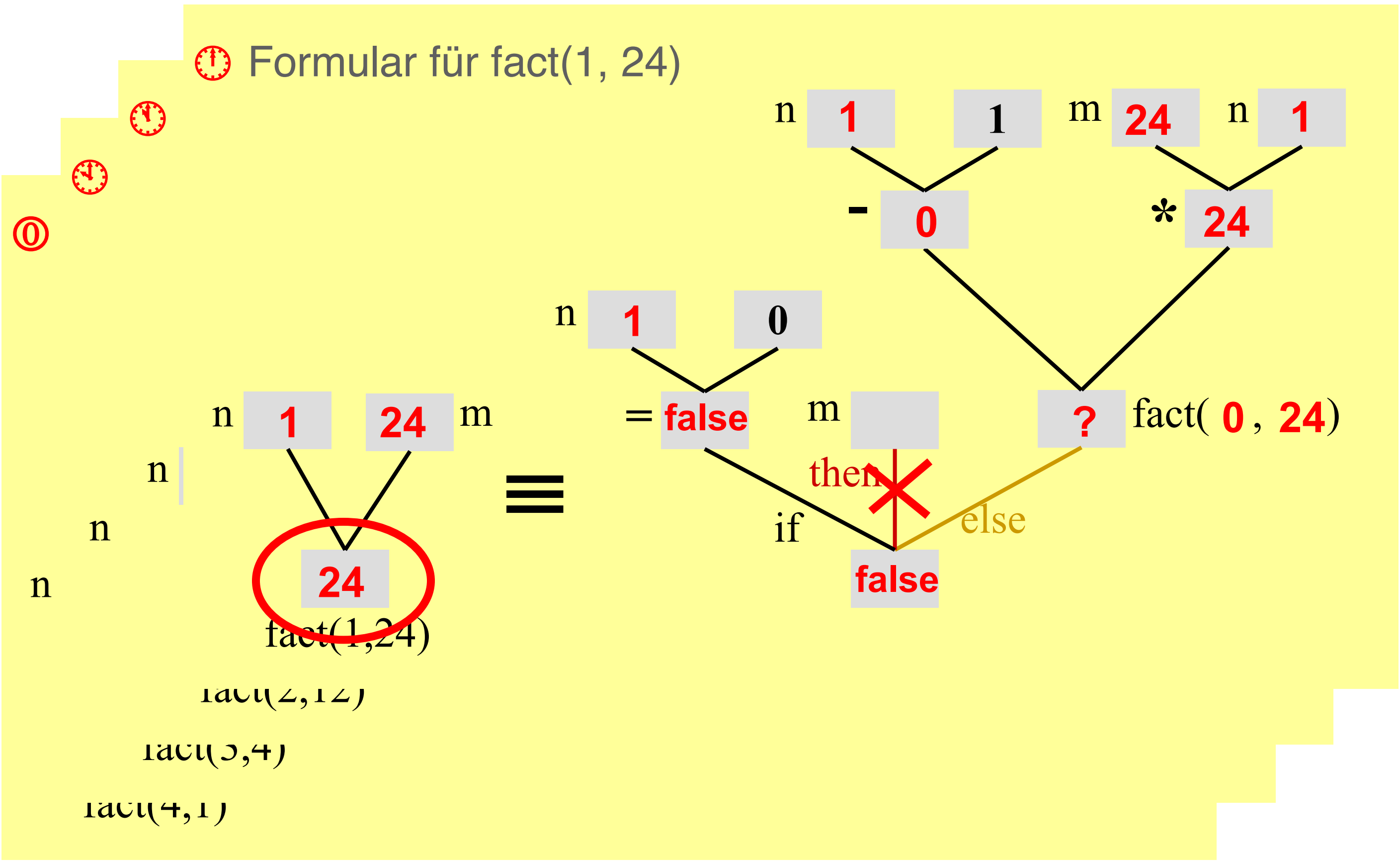
```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```



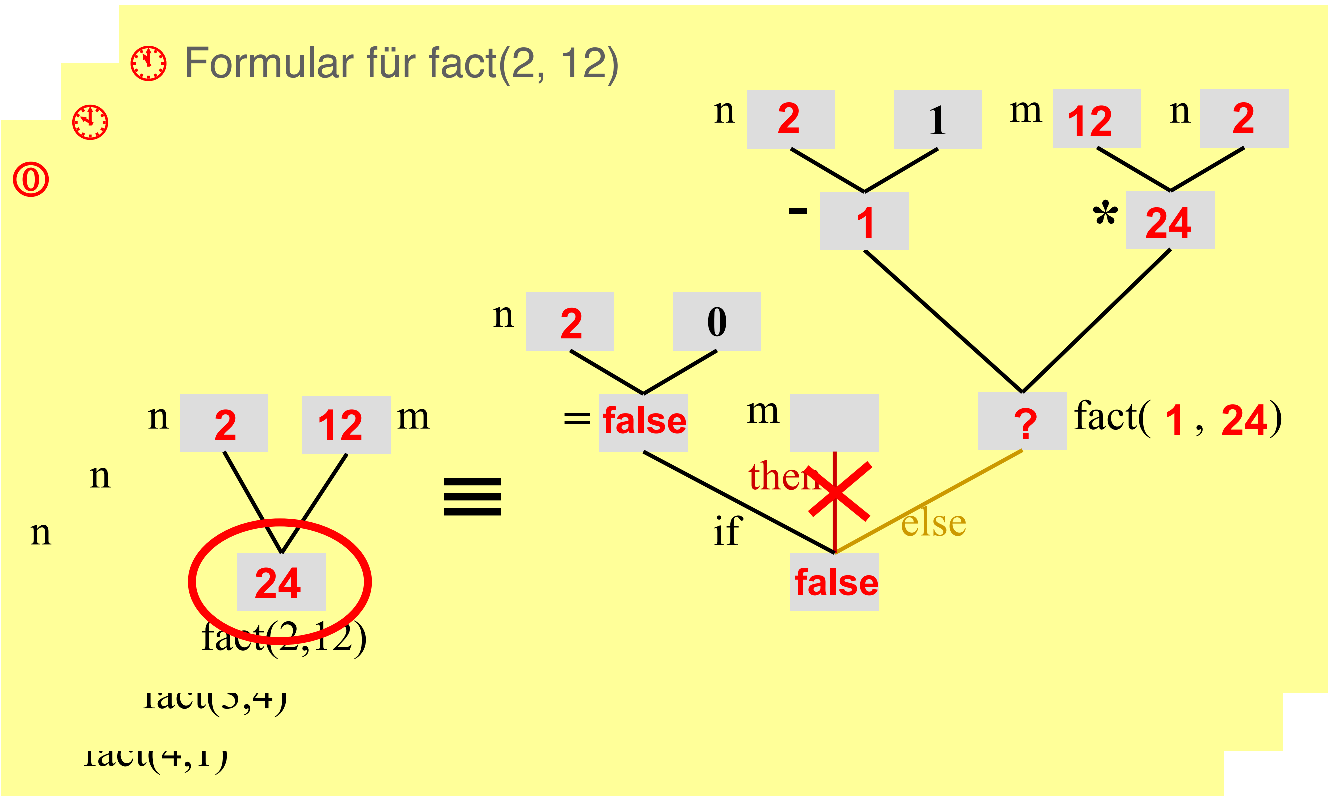
```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```



```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```

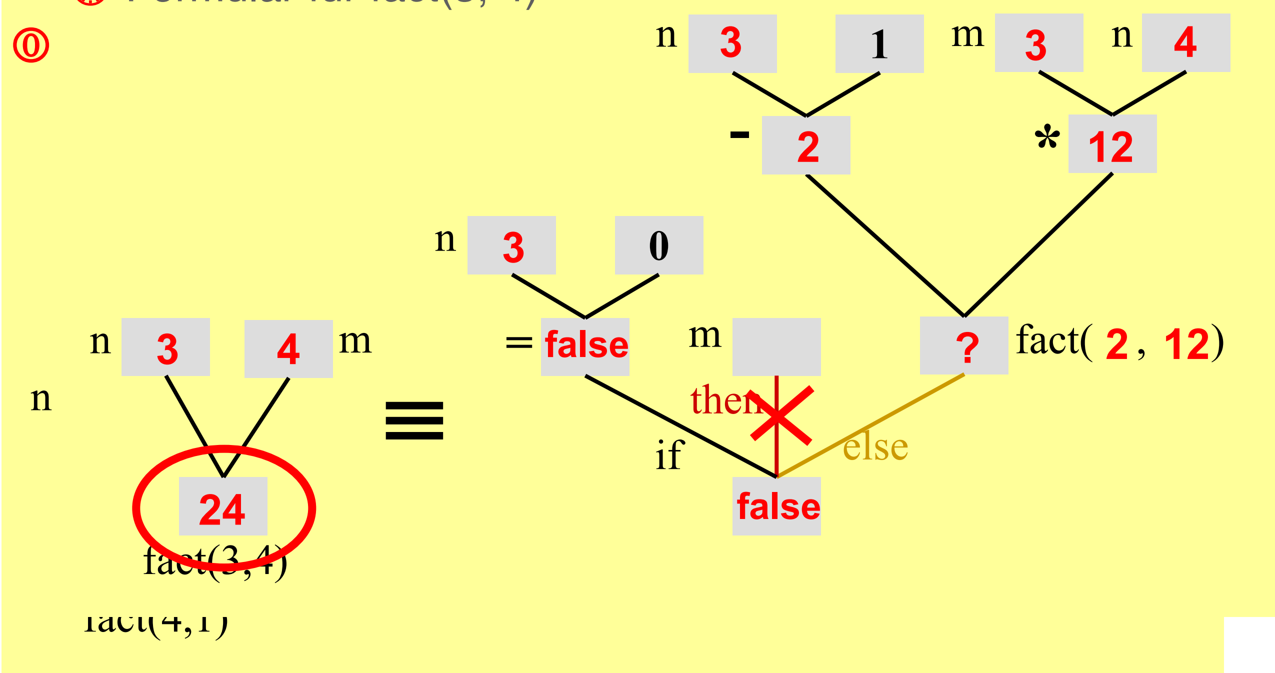


```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```



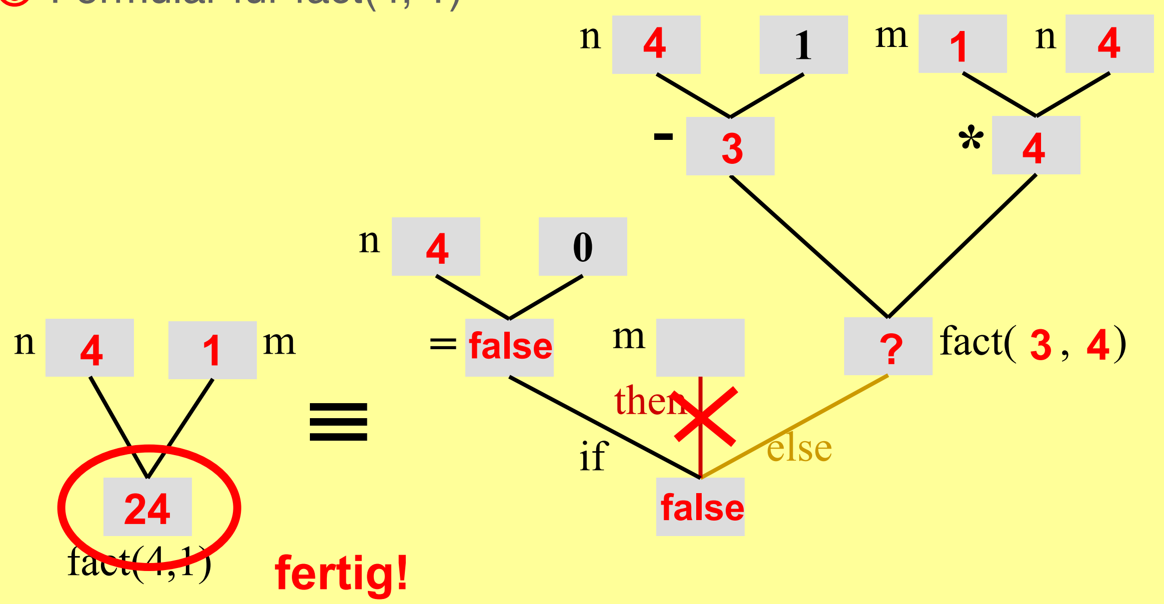
```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```

Formular für fact(3, 4)



```
long fact(long n, long m) {

    if (n == 0)
        return m;
    else
        return fact(n-1, m*n);
}
```

⓪ Formular für fact(4, 1)



Rekursion vs Schleifen

- End-Rekursionen lassen sich unmittelbar durch **`while`**-Schleifen ersetzen – und umgekehrt

  Hinweis: Jede primitiv-rekursive Funktion kann unter Zuhilfenahme eines Stapels (*stack*) durch eine **`for`**-Schleife ersetzt werden (siehe später)

- Bei der Auflösung der End-Rekursion werden die noch nicht beendeten Inkarnationen nicht mehr gebraucht und müssen nur noch geschlossen werden, wobei nur die jeweiligen Ergebnisse zurückgegeben werden

- Struktur bei **Transformationen von Rekursionen** in **Schleifen**:
  - Die Abbruchbedingung der Rekursion (**`if`**-Anweisung) wird in die Bedingung der Wiederholungsschleife und deren Prüfung überführt
  - Der Rumpf der rekursiven Methode (mit den Anweisungen des Berechnungsschemas) wird zum Schleifenrumpf (Block)
  - In manchen Fällen werden zusätzliche Daten oder Datenstrukturen notwendig

Bsp.: Berechnung der fact(n, m)-Funktion

```
long fact(long n, long m) {
    while (n > 0) {
        m = m * n;
        n = n - 1;
    }
    return m;
}
```

Bsp.: Berechnung des „größten gemeinsamen Teilers“ (ggT)

```
public int ggT(int a, int b) {

    if (b == 0)
        return a;
    else
        return ggt(b, a % b);
}
```

```
public int ggT(int a, int b) {

    while (b != 0) {
        int temp = b;

        b = a % b;
        a = temp;
    }
    return a;
}
```

Hinweis: Hier sind die zusätzlichen benötigten Daten in Form der temporären Variable `temp` innerhalb des Blocks der Wiederholungsschleife notwendig; bei komplexeren Problemen kann dies z.B. auch ein *Array* oder eine entsprechende Struktur sein

- Struktur bei **Transformationen von Schleifen** in (lineare) **Rekursionen**:
  - Der Schleifenrumpf (Block aus Anweisungen) wird Bestandteil des Rumpfes der rekursiven Methode (der Abschnitt, der nicht die Terminations-Bedingung beschreibt)
  - Die Prüfung der Bedingung der Wiederholungsschleife wird durch eine entsprechende **`if`**-Anweisung zur Abbruchbedingung der Rekursion

Bsp.: Berechnung einer Produktfolge (aus einer **`while`**- bzw. **`for`**-Schleife)

```
long product(long n) {
    long prod = 1;

    while (n > 1) {
        prod = prod * n;
        n    = n - 1;
    }
    return prod;
}
```

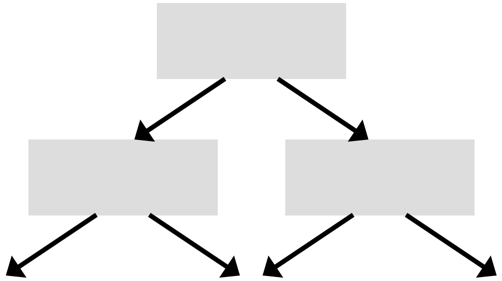
```
long product(long n) {
    long prod = 1;

    for (long i = n; i > 1; i--)
        prod = prod * i;

    return prod;
}
```

```
long product(long n) {

    if (n == 1)
        return 1;
    else
        return n * product(n-1);
}
```

Baumartige (kaskadenartige) Rekursion

Allgemeine Betrachtungen

- Jeder nicht-terminierende Aufruf eines Algorithmus mit baumartiger Rekursion führt während des Abstiegs zu mindestens zwei weiteren Aufrufen der rekursiven Methode
- Beispiele sind:
  - Berechnung der Fibonacci-Zahlen
  - Berechnung des Binomial-Koeffizienten (s.S. 5)
  - Bestimmung der kürzesten Wege auf einem Schachbrett („Kansas City“ Problem)
  - ...
- Die Baumartigkeit (bzw. Kaskadenform) wird dadurch deutlich, dass der rekursive Abstieg mit seinen Aufrufen als Baumstruktur dargestellt werden kann und der eine kaskadenartige Struktur aufweist



Fibonacci-Zahlen

Aus dieser Mehrdeutigkeit ergibt sich, dass der Wert für *F*(0) häufig auch mit 1 definiert wird

- Ursprung: Beispiel zur mathematischen Populationsdynamik (→ Biomathematik)

  „Das Weibchen jedes 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)

- Die Berechnungsvorschrift ist wie folgt definiert:

$$F(0) = 0$$

$$F(1) = 1$$

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

Anzahl der Kaninchenpaare des letzten Monats

Anzahl neugeborener Paare = Anzahl Kaninchenpaare, die ≥ 2 Monate alt sind

... generiert die Folge

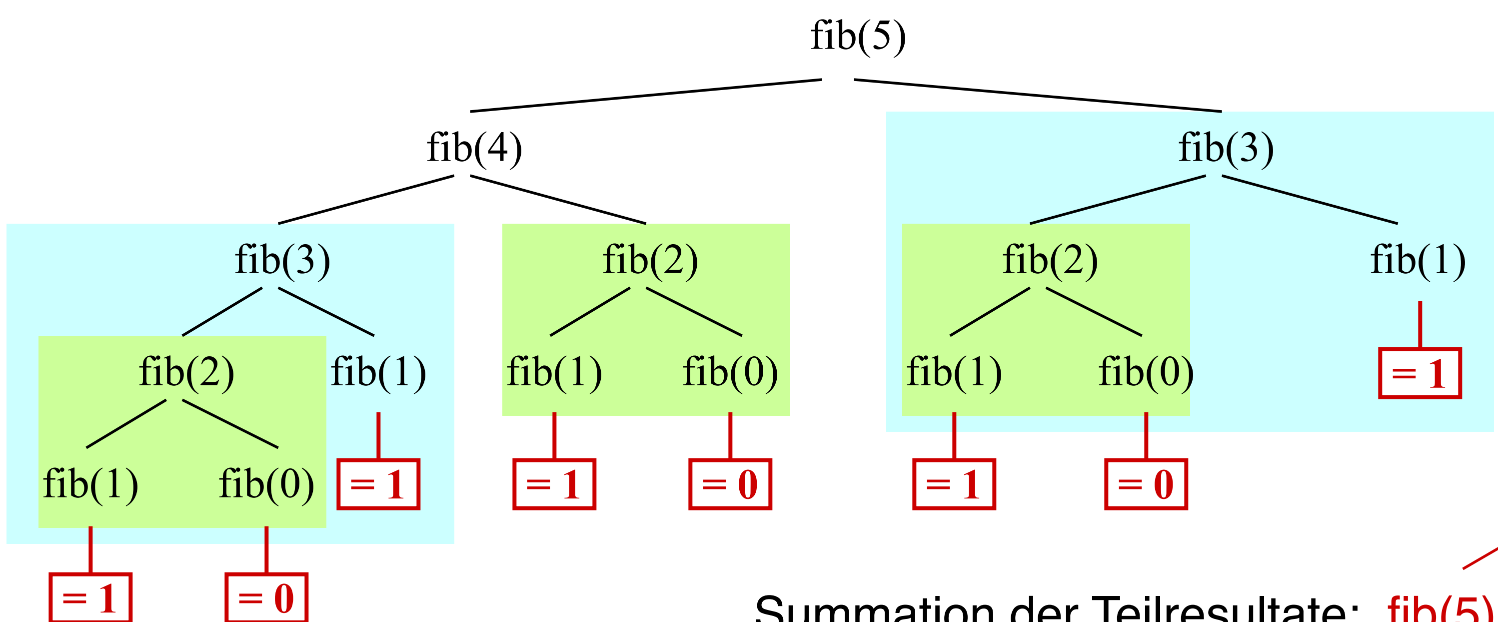
n	0	1	2	3	4	5	6	7	8	9	10	11	12
F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

- Berechnung durch rekursiven Algorithmus (mit fib(0) = 0)

```
long fib(long n) {

    if ((n == 0) || (n == 1))
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

- 2-facher Aufruf von fib(...)
- Ergebnisse werden addiert

- Aufruf der Methode und die kaskadenförmige Rekursion (Beispiel n = 5)



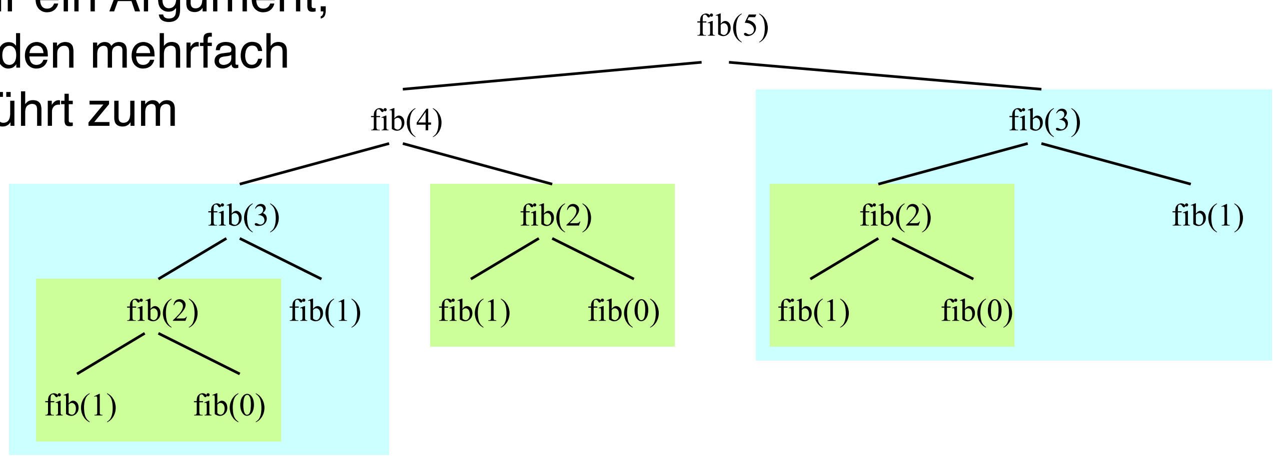
Summation der Teilresultate: fib(5) = 5

Eigenschaften

- Die rekursive Berechnung erfolgt kaskadenförmig (Ausschnitt für fib(3))

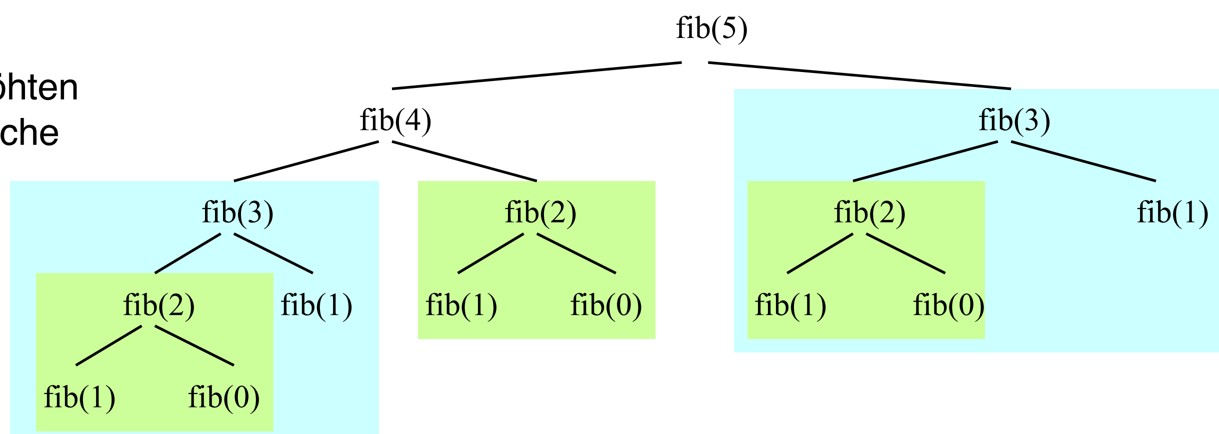
```
fib(3) = fib(2) +           fib(1)
           |
       = fib(1) + fib(0) + fib(1)
           |
       = 1      + fib(0) + fib(1)
                    |
       = 1      + 0      + fib(1)
                             |
       = 1      + 0      + 1
       = 2
```

- Die Berechnungen für ein Argument, z.B. $n = 2$, $n = 3$, werden mehrfach durchgeführt – dies führt zum exponentiellen Wachstum des Zeitbedarfs bei der Berechnung



- Struktur der rekursiven Aufruf-Kaskade

	n	**2**	**3**	**4**	**5**	**6**	**7**
	$F(n) \equiv fib(n)$	**1**	**2**	**3**	**5**	**8**	**13**
Fibonacci	neue Aufrufe(*)	**2**	**4**	**8**	**14**	**24**	**...**
	Blätter(**)	**2**	**3**	**5**	**8**	**13**	**...**



<u>Bemerkungen</u>:

(*)für einen um einen Zähler erhöhten Argumentwert werden zusätzliche Aufrufe der rekursiven Funktion notwendig

(**)es entstehen jeweils in dem Baum der rekursiven Aufrufe am untersten Ende eine bestimmte Anzahl von Blättern, die die Werte für den Rekursionsbeginn repräsentieren

Iterativer Algorithmus zur direkten Bestimmung des Fibonacci-Werts

- Algorithmus:

```
long fibIter(long n) {

    if (n == 0)
        return 0;
    else {
        long a    = 0,
             b    = 1,
             fibo = 1;

        for (long i = 2; i <= n; i++) {
            fibo = a + b;
            a    = b;
            b    = fibo;
        }
        return fibo;
    }
}
```

- Berechnungsschema (**`for`**-Schleife):

$\mathtt{fibo}_{\mathtt{neu}} = \mathtt{a}_{\mathtt{alt}} + \mathtt{b}_{\mathtt{alt}};$ 1

$\mathtt{a}_{\mathtt{neu}} = \mathtt{b}_{\mathtt{alt}};$ 2

$\mathtt{b}_{\mathtt{neu}} = \mathtt{fibo}_{\mathtt{neu}}$ 3

$= \mathtt{a}_{\mathtt{alt}} + \mathtt{b}_{\mathtt{alt}};$

Es ergibt sich folgende Struktur: **`b`** „läuft **`fibo`** hinterher“ und **`a`** „läuft **`b`** hinterher“

- Ergebnisberechnung im iterativen Fall

`n`	`a`	`b`	`fibo = fibonacci(n)`
0			0
1	0	1	1
2	1	1	1
3	1	2	2
4	2 +	3	3
5	3	5	5
6	5	8	8
7	8	13	13
8	13	21	21

1 2 3

Zu Rekursion und Iteration – weitere Anmerkungen

- *Warum werden Rekursionen in Iterationen umgewandelt?*
  - Rekursive Unterprogramme setzen rekursive Definitionen und Aufrufe auf natürliche Weise um
  - Sie sind gegenüber iterativen Lösungen jedoch meist weniger speicher- und zeiteffizient (für jede Inkarnation muss der Zustand des (Unter-) Programms gespeichert werden)
- *Wie werden Rekursionen in Iterationen umgewandelt?*
  - Die Umwandlung end-rekursiver (linearer), primitiv-rekursiver Funktionen ist im allgemeinen einfach, da nur je ein rekursiver Aufruf erfolgt und somit nur jeweils ein Zwischenergebnis verrechnet werden muss
  - Im Falle mehrerer rekursiver Aufrufe (bei kaskadenartiger Rekursion) müssen mehrere Zwischenergebnisse verwaltet sowie Zwischenresultate berechnet und weitergereicht werden; hierfür werden Hilfsvariablen oder dynamische Speicherbereiche (*stack*; **Teil XI**) verwendet

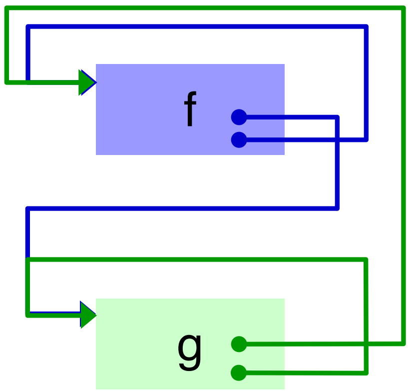
Geschachtelte und verschränkte Rekursion

Einordnung

- Bei geschachtelten Rekursionen (*compound recursion*) ist das Argument (Parameter) eines rekursiven Aufrufs selbst wieder ein rekursiver Aufruf

  Hinweis: Die Wirkungsweise derartiger Berechnungen ist häufig nur schwer zu durchschauen

- Bei verschränkten (oder wechselseitigen) Rekursionen verwenden sich mehrere Funktionen (Methoden) gegenseitig:
  - Methode f ruft Methode g (und sich selbst) auf
  - Methode g ruft Methode f (und sich selbst) auf



- Algorithmen mit verschränkten Rekursionen lassen sich nicht ohne Weiteres mittels einfacher Schleifen iterativ programmieren

  Hinweis: Beliebige Rekursionsarten lassen sich immer dadurch in ein iteratives Format bringen, indem die kellerartige Verarbeitung der Rekursion durch Speicherung von Zwischenergebnissen ausprogrammiert wird

Beispiele für geschachtelte Rekursionen

- Berechnung der Modulo-Funktion

```
int modulo(int a, int b) {
    if (a < b)
        return a;
    else if (a < 2*b)
        return a - b;
    else
        return modulo(modulo(a, 2*b), b)
}
```

- Berechnung der Ackermann-Funktion

<u>Def</u>.:

$$A(x, y) = \begin{cases} y + 1 & \text{falls } x = 0 \\ A(x-1, 1) & \text{falls } y = 0 \\ \boxed{A(x-1, \boxed{A(x, y-1)})} & \text{sonst} \end{cases}$$

- <u>**Einordnung**</u>: Die Ackermann-Funktion ist hochgradig rekursiv, A(4, 2) hat bereits über 19000 Dezimalstellen; diese und andere mehrfach rekursive (*compound recursive*) Funktionen gehören zur Klasse der **μ-rekursiven** (nicht primitiv-rekursive) Funktionen (**Theoretische Informatik**)

3. Teilen-und-Herrschen

- Prinzip Teilen-und-Herrschen (*Divide-and-Conquer*)
- Beispielaufgabe – Markieren eines Lineals

Prinzip Teilen-und-Herrschen (*Divide-and-Conquer*)

Einordnung

- Häufig bieten sich rekursive Lösungen für ein Problem an, in denen die Eingabemenge in zwei – etwa gleich große – Teile zerlegt wird
- Diese Teile werden wiederum **jeweils** durch rekursive Aufrufe des Lösungsalgorithmus bearbeitet
- Schema von Teile-und-herrsche (*divide-and-conquer*) Verfahren:

Gegeben: ein Problem p

Ist p trivial lösbar?
Wenn ja:
  dann löse das Problem p
Wenn nein:
  dann zerteile (*divide*) das Problem p in (etwa gleich große) Teilprobleme p_A, p_B;
  wende den Algorithmus auf die Teilprobleme p_A und p_B an (Rekursion);
  setze die Lösung für p aus den Lösungen der Teilprobleme zusammen (*conquer*).

- Die Teilprobleme aus den *divide*-Schritten werden wiederum in gleiche Teile zerlegt, so dass sich eine baumartige Rekursion ergibt

Eigenschaften und Analyse

- Man kann *Divide-and-conquer* Verfahren meist recht genau hinsichtlich ihrer Komplexität analysieren
- Es sei T(n) die Anzahl der Schritte, die ein *Divide-and-conquer* Verfahren benötigt, welches eine Datenmenge der Größe n in genau zwei gleich große Teile zerlegt (für die Berechnung der Lösung entstehen (normalerweise) keine redundanten Berechnungen, da die Menge der Eingabedaten in nicht-überlappende Teilmengen A, B zerlegt wird, d.h. $|A \cup B| = |A| + |B|$)
- Wenn die Anzahl der benötigten Schritte
  - beim Zerlegen (*divide*) Schritt und
  - beim *Conquer*-Schritt

  jeweils proportional zu n ist, dann ergibt sich folgende Gleichung für T(n) mit einer Konstanten c:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

  Die Gleichung hat die Lösung:

$$T(n) = c \cdot n \cdot \log_2(n) + d \cdot n \quad \text{mit} \quad d = T(1)$$

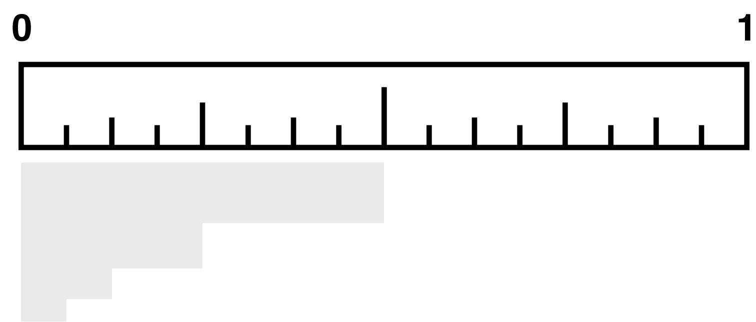
  Solch ein *Divide-and-conquer* Algorithmus hat also eine Laufzeit von $O(n \cdot \log_2 n)$

Beispielaufgabe – Markieren eines Lineals

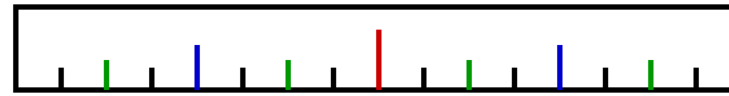
Aufgabe

Literatur: R. Sedgewick (1988) Algorithms, 2nd Edition. Addison-Wesley, Reading, MA (USA)

- Es soll ein Lineal mit Markierungen unterschiedlicher Länge versehen werden



- Das Problem kann rekursiv gelöst werden:
  - Das Lineal wird sukzessive halbiert
  - Die Mitte wird markiert; die Länge der Markierung (= Anzahl der bisherigen Zerlegungen) entspricht der Tiefe der Rekursion
  - Für jede Lineal-Hälfte wird der Halbierungs- und Markierungs-Schritt wiederholt
  - Rekursionsende:
    - Länge der Markierung = 0 oder
    - Länge des aktuellen Linealabschnitts = 0



- Vorgehensweise

Ebene: N — Markierung bei $\frac{1}{2}[0:1]$

N – 1 — Markierung bei $\frac{1}{2}\left[0:\frac{1}{2}\right]$ u. $\frac{1}{2}\left[\frac{1}{2}:1\right]$

N – 2 — Markierung bei $\frac{1}{2}\left[0:\frac{1}{4}\right]$ u. $\frac{1}{2}\left[\frac{1}{4}:\frac{1}{2}\right]$ sowie $\frac{1}{2}\left[\frac{1}{2}:\frac{3}{4}\right]$ u. $\frac{1}{2}\left[\frac{3}{4}:1\right]$

N – 3 — Markierung bei **...**

:

<u>Anmerkung</u>: Die „Ebene“ bestimmt die Länge der Markierung
(längste Markierung auf der obersten Ebene N)

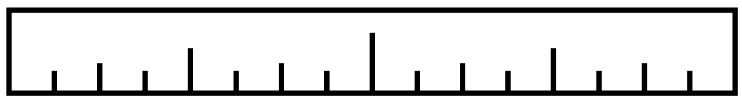
- Es bleibt festzulegen, in welcher Reihenfolge die Markierungen rekursiv aufgetragen werden

Algorithmus und Implementierung

Rekursiver Algorithmus

- Das Lineal ist als lineares *Array* (Feld) mit **`int`**-Elementen realisiert (Länge N)
- Die Werte der Einträge bezeichnen jeweils die Länge der Markierung an der betreffenden Position

Lineal

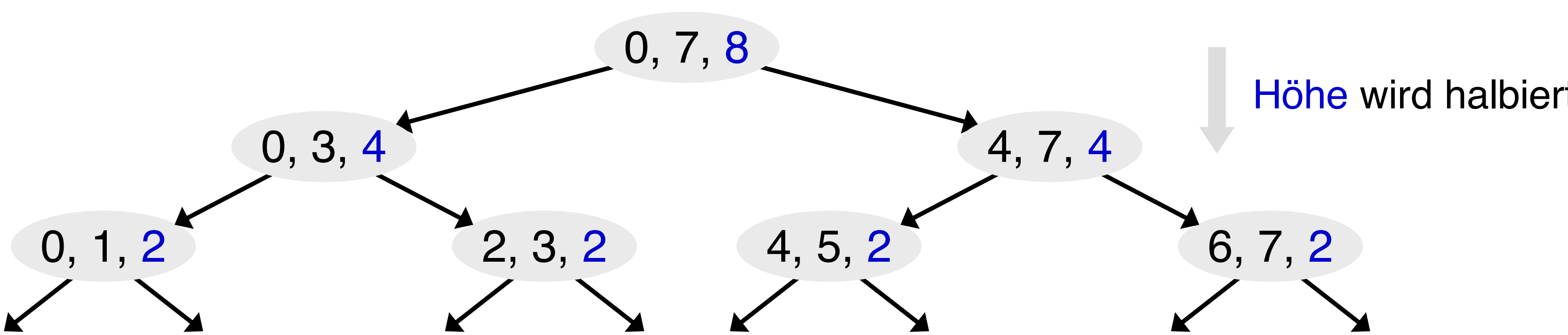


Array `ruler`

0 1 2 N-1

Initialisierung: $\forall i,\ 0 \leq i \leq N-1$: `ruler[i] = 0`

- Vorgehensweise: Markierungen für `beg = 0` und `end = 7` (`height = 8`)



Implementierung in Java (Demo: MarkRuler.java)

```
public class MarkRuler {

    public static void main(String[] args) {
        final int N = 16;  // Laenge des Lineals
        int[]     ruler = new int[N];  // initialisiert mit 0
        int       beg = 0,
                  end = N - 1;

        markRuler(ruler, beg, end, N);

        System.out.println("Markierungen des Lineals: ");
        for (int i = 0; i < ruler.length; i++)
            System.out.print(ruler[i] + " ");
    }

    static void markRuler(int[] ruler, int beg, int end, int height) {
        if ((height > 0) && (end - beg > 0)) {
            int middle = (beg + end) / 2;

            setMark(ruler, middle, height);              // Markierung des Lineals
            markRuler(ruler, beg, middle, height/2);     // Rekursion fuer linke Haelfte
            markRuler(ruler, middle+1, end, height/2);   // Rekursion fuer rechte Haelfte
        }
    }  // end markRuler

    static void setMark(int[] ruler, int index, int height) {
        ruler[index] = height;
    }  // end setMark
}  // end class MarkRuler
```

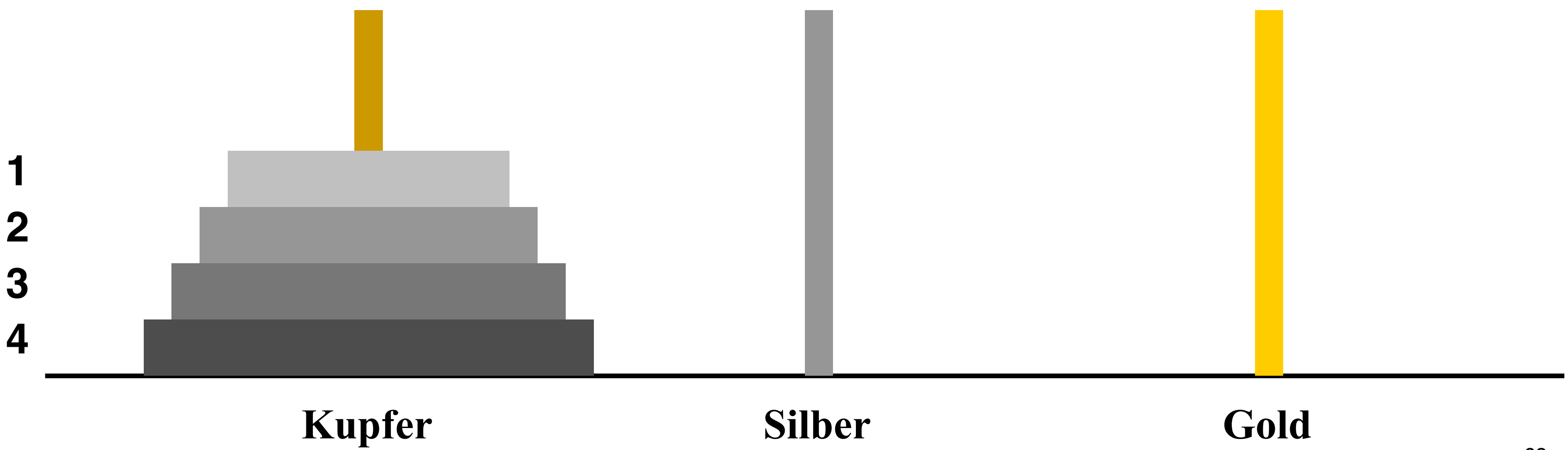
4. Türme von Hanoi

- Türme von Hanoi – Aufgabe und Lösungsstrategie
- Rekursiver Algorithmus
- Aufwandsabschätzung für die rekursive Lösung
- Nicht-rekursive Lösung

Türme von Hanoi – Aufgabe und Lösungsstrategie

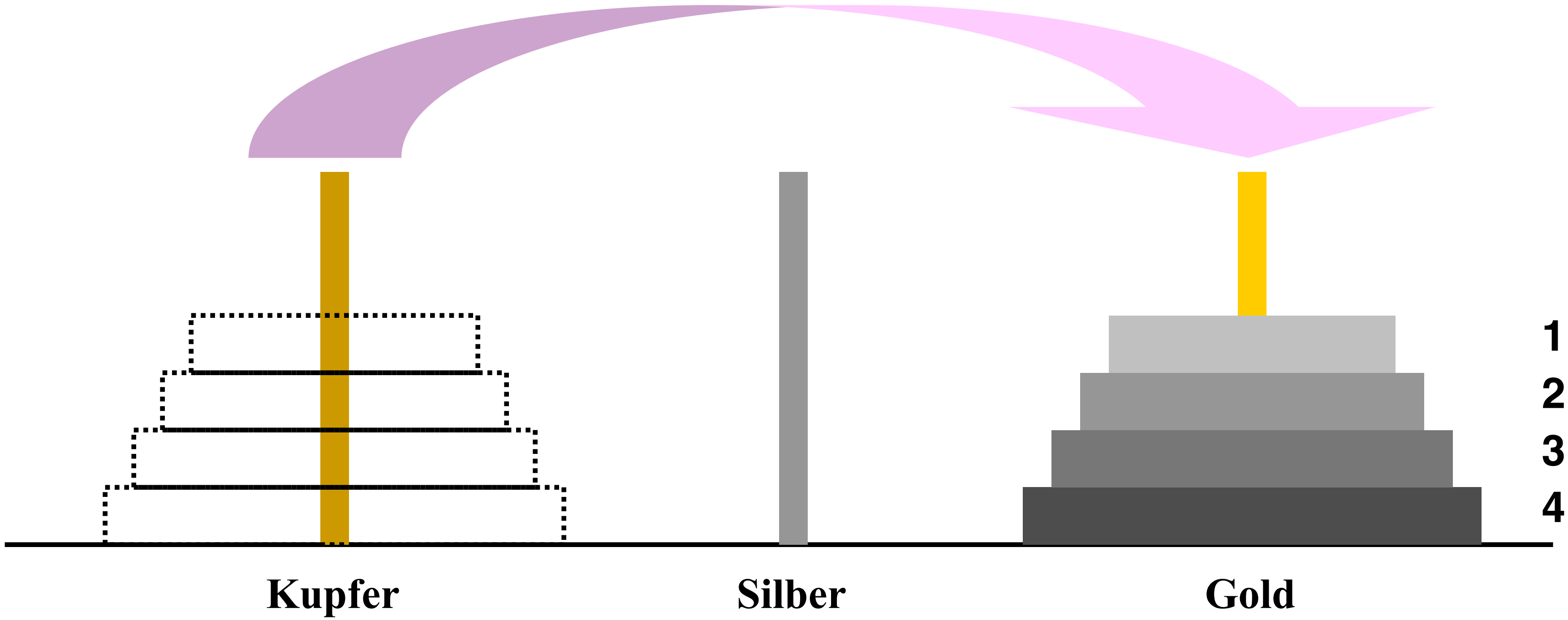
Einordnung

- Nach einer alten Legende standen vor langer Zeit vor einem Tempel in Hanoi drei Säulen:
  - eine aus **Kupfer**,
  - eine aus **Silber**,
  - 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



Lösungsstrategie

- 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 …

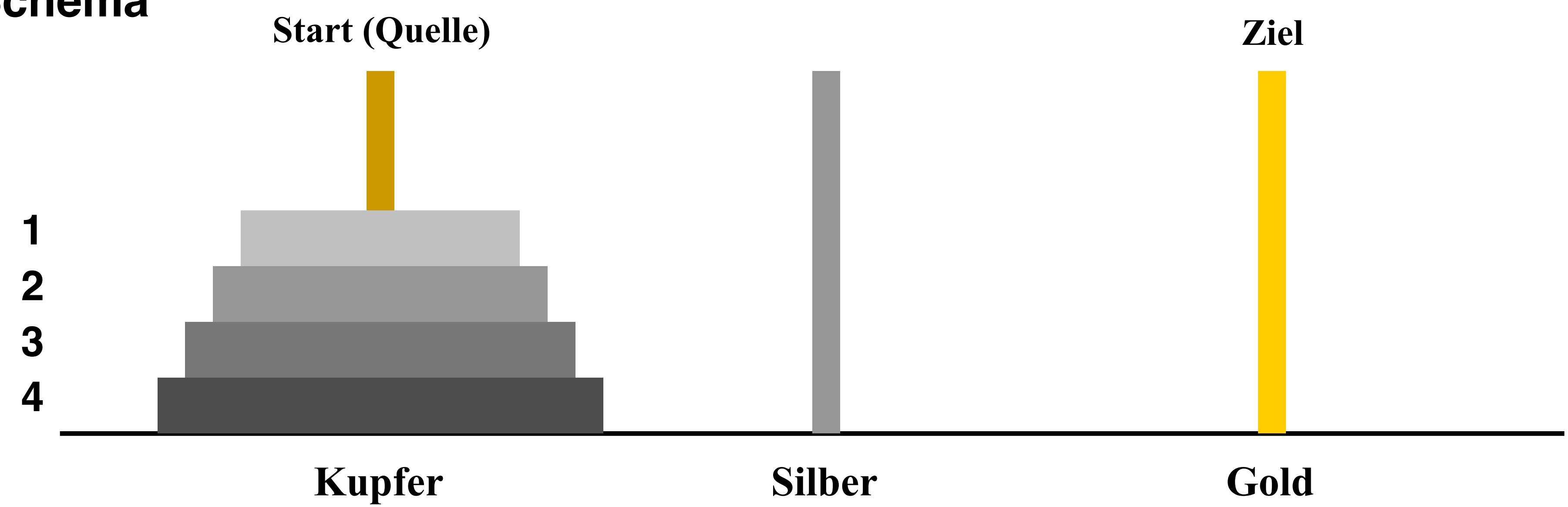
- Die Aufgabe kann in **drei Teilaufgaben** zerlegt werden:

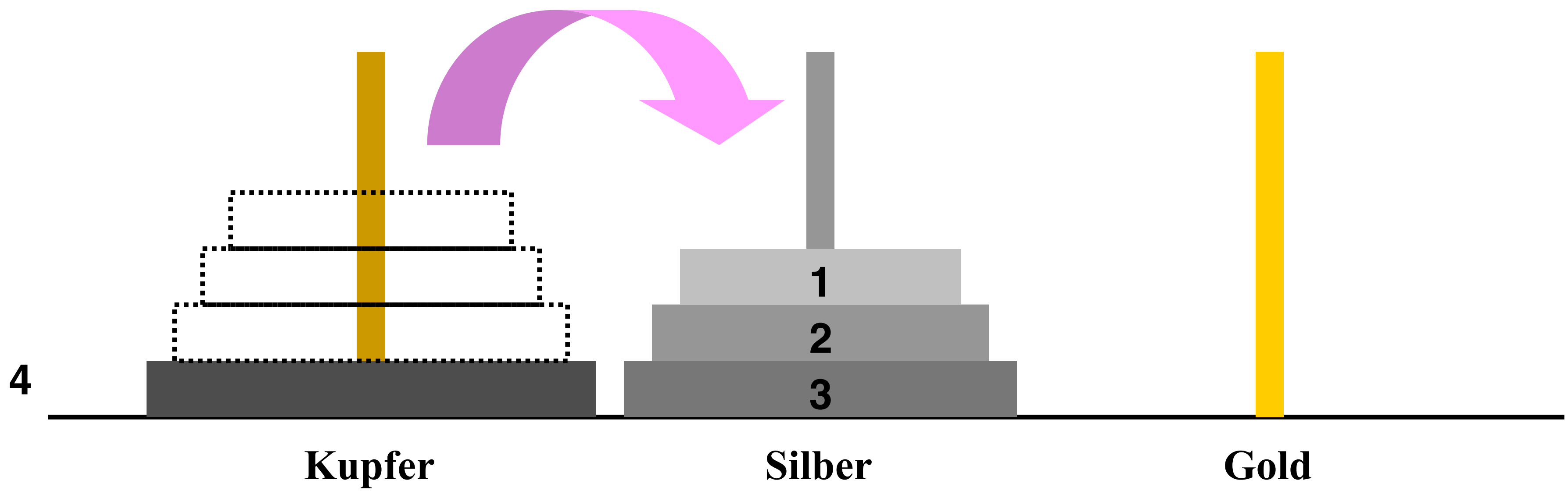
Teil 1: Transportiere den Turm – bestehend aus 99 **oberen** Scheiben von der **kupfernen** zur **silbernen** Säule
Teil 2: Transportiere die **übriggebliebene** 100ste Scheibe (ganz unten) von der **kupfernen** zur **goldenen** Säule
Teil 3: Transportiere den Turm mit den 99 Scheiben von der **silbernen**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

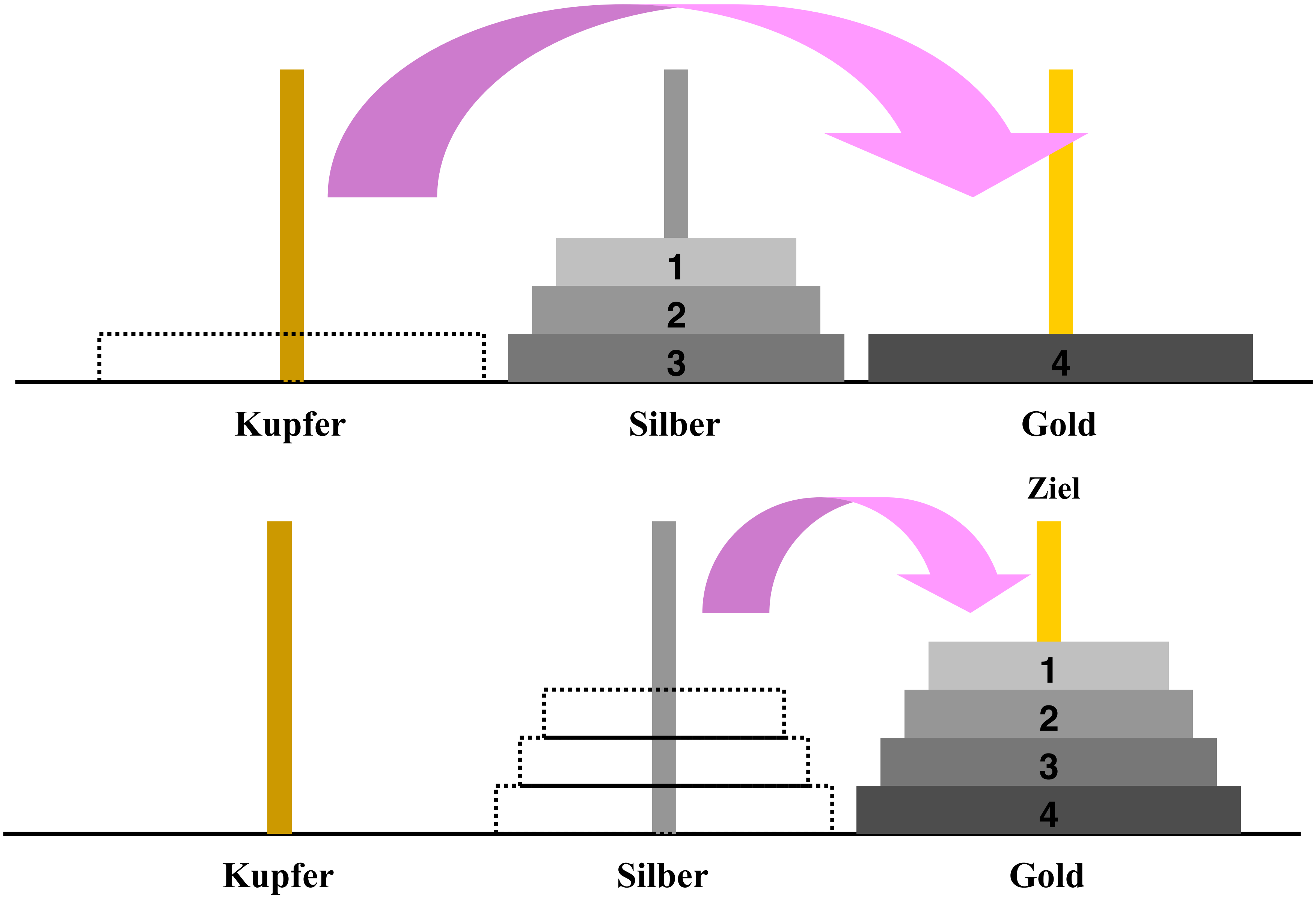
- Beim Betrachten dieses Schemas bemerkte der Mönch, dass Teil 1 und Teil 3 außerordentlich mühsam sein würden; da er nicht nur ein alter, sondern auch ein weiser Mönch war, entschloss er sich, **Teil 1 von seinem ältesten Schüler ausführen** zu lassen

  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 mindestens 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;
> - 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 … 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)

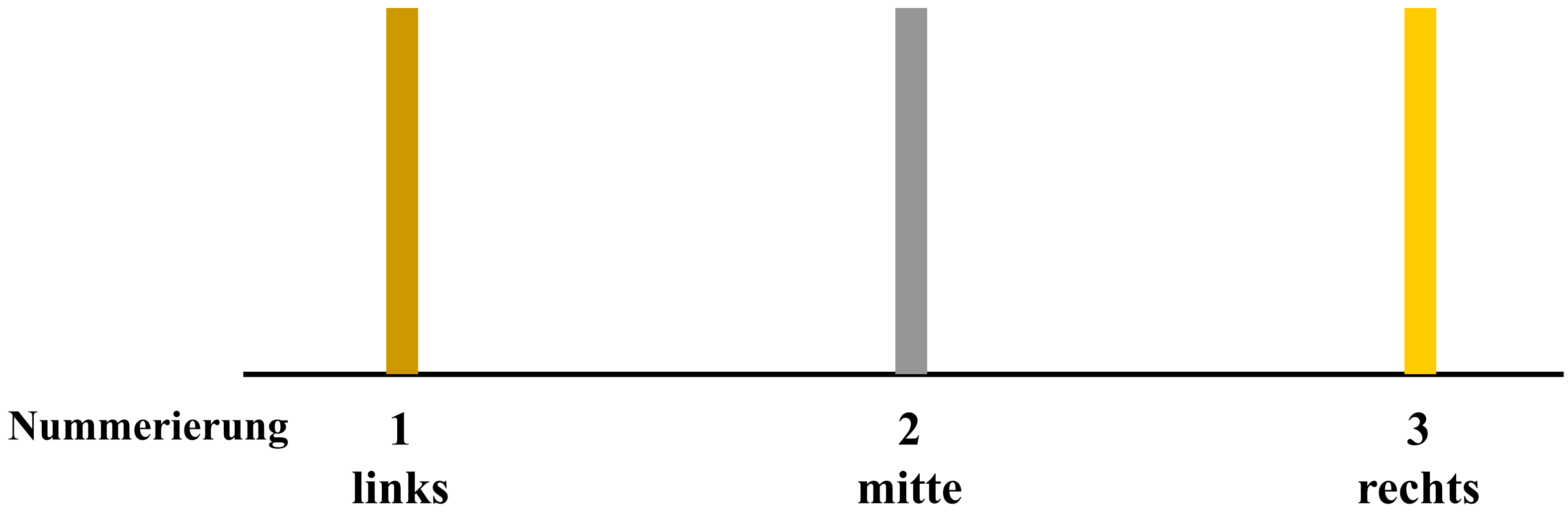
Rekursiver Algorithmus

Grobstruktur und Implementierung

Struktur und Parametrisierung

Eingabe und festgesetzte Größen:

- Turmhöhe (Parameter) : **`heightOfTower`** (eingelesene Anzahl der Scheiben)
- Ausgangssäule : **`1 (links)`**
- Zielsäule : **`3 (rechts)`**



Implementierung (Demo: TowersOfHanoi.java, TowersOfHanoiS.java mit Kurzausgabe)

```
public class TowersOfHanoiS {
    public static void main(String[] args) {
        int heightOfTower;  // Hoehe des Turms = Anzahl der Scheiben
        final int POLE_START = 1;
        final int POLE_GOAL  = 3;

        TextIO.put("Anzahl der Scheiben (positive Zahl): ");
        heightOfTower = TextIO.getlnInt();
        while (heightOfTower < 0) {
            TextIO.put("Parameter positiv oder 0, noch einmal ...");
            heightOfTower = TextIO.getlnInt();
        }  // Wert fuer heightOfTower >= 0 ...

        moveTower(heightOfTower, POLE_START, POLE_GOAL);
    }  // end main

    static void moveTower(int height, int startPole, int goalPole) {
        if (height > 0) {  // Rekursionsende bei height == 0
            int otherPole = 6 - startPole - goalPole;
            moveTower(height-1, startPole, otherPole);
            System.out.println("Scheibe von " + startPole +
                               " nach "       + goalPole);
            moveTower(height-1, otherPole, goalPole);
        }
    }  // end moveTower
}  // end class TowersOfHanoiS
```

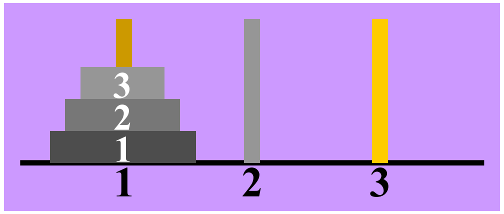
Rekursion

transportDisk(startPole, goalPole)

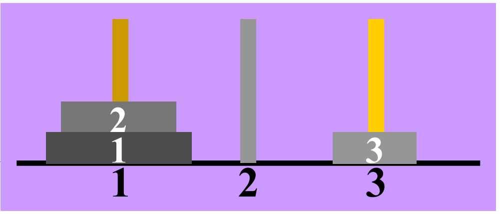
`start` = Position *Ausgangssäule*
`goal` = Position *Zielsäule*
`6-start-goal` = Position *Hilfssäule*

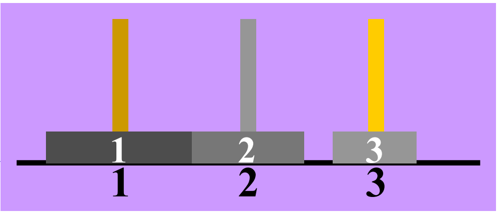
„Handsimulation“ von `TowersOfHanoi`

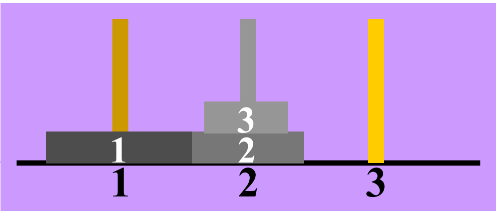
Start :

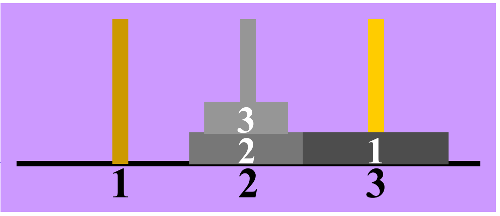



```
height = 3
moveTower(3, 1, 3)
.........................................................
   height = 3 > 0:
   moveTower(2, 1, 6-1-3=2)
      height = 2 > 0:
      moveTower(1, 1, 6-1-2=3)
         height = 1 > 0:
         moveTower(0, 1, 6-1-3=2)
            height = 0 ⇒ EXIT
         transportDisk(1, 3)
         moveTower(0, 3, 6-1-3=2)
            height = 0 ⇒ EXIT
      transportDisk(1, 2)
      moveTower(1, 6-1-2=3, 2)
         height = 1 > 0:
         moveTower(0, 3, 6-3-2=1)
            height = 0 ⇒ EXIT
         transportDisk(3, 2)
         moveTower(0, 6-3-2=1, 2)
   transportDisk(1, 3)
   moveTower(2, 6-1-3=2, 3)
   : <→ nächste Seite >
```



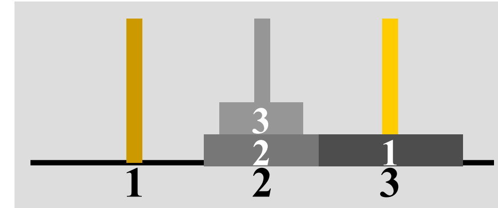


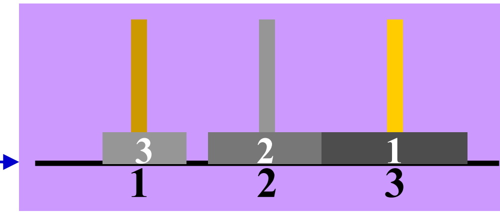


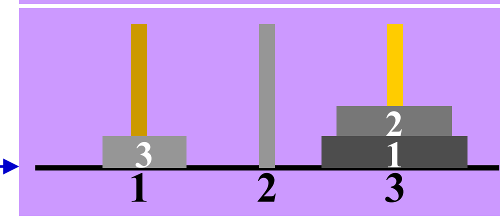


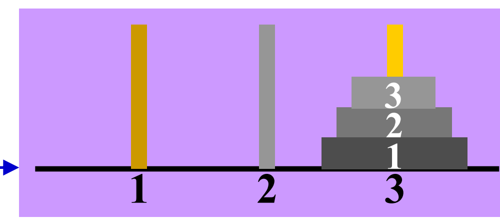
```
   :
   transportDisk(1, 3)
   moveTower(2, 6-1-3=2, 3)
➡ height = 2 > 0:
➡ moveTower(1, 2, 6-2-3=1)
    ➡ height = 1 > 0:
    ➡ moveTower(0, 2, 6-2-1=3)
        ➡ height = 0 ⇒ EXIT
    ➡ transportDisk(2, 1)
    ➡ moveTower(0, 6-2-1=3, 1)
        ➡ height = 0 ⇒ EXIT
➡ transportDisk(2, 3)
➡ moveTower(1, 6-2-3=1, 3)
    ➡ height = 1 > 0:
    ➡ moveTower(0, 1, 6-1-3=2)
        ➡ height = 0 ⇒ EXIT
    ➡ transportDisk(1, 3)
    ➡ moveTower(0, 6-3-2=1, 2)
        ➡ height = 0 ⇒ EXIT
TERMINATION
```

Ende letzte Folie



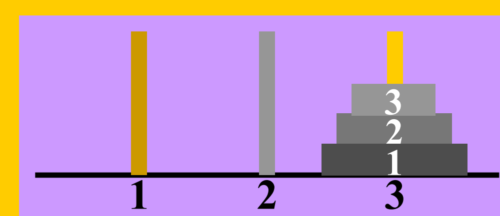






→ **Programmende**

Finale:



Aufwandsabschätzung für die rekursive Lösung

Problemstellung

- Frage: *Wie oft müssen Scheiben hin- und hergetragen werden?*
- Ausgangssituation: Auf der kupfernen Säule befinden sich $n \geq 0$ Scheiben

Scheibenanzahl	Anzahl der Trageoperationen
1	**1**
2	**1 + 1 + 1 = 3**
3	**3 + 1 + 3 = 7**
4	**7 + 1 + 7 = 15**
:	**:**
n	**1 + 2 · Trageoperationen bei (n-1)**

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

Überprüfung

- Trageoperationen bei n = 1 + 2·Trageoperationen bei (n – 1) (n: Anzahl Scheiben)

Kontrolle:

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

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

Zurück zur Ausgangssituation …

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

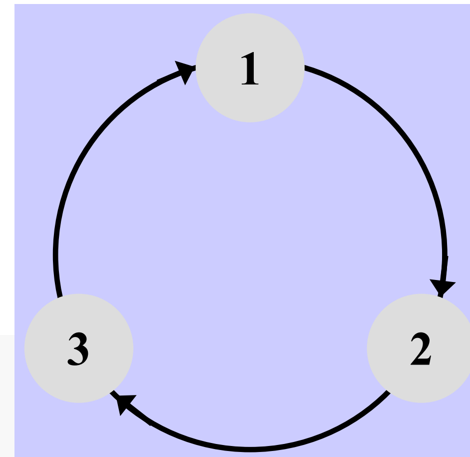
$$2^{100} - 1\,\text{min.} \approx 1.26765 \cdot 10^{30}\,\text{min.} \qquad (1\ \text{Jahr} = 525600\ \text{min.})$$
$$\approx 2.4 \cdot 10^{24}\,Jahre(!)$$

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

Nicht-rekursive Lösung

Repräsentation und Algorithmus

Literatur: D. Harel. The Science of Computing. Addison-Wesley Publ.Comp., Reading, MA, USA, 1989



- Die drei Säulen sind auf einem Ring angeordnet
- Alle n Scheiben sind zu Beginn auf einer Säule gestapelt (z.B. auf Säule 1, oben)
- Algorithmus

```
...

while ( true ) {
    bewege die kleinste Scheibe (= oberste Scheibe der Start-Saeule)
       um eine Position im Uhrzeigersinn (1 → 2 oder 2 → 3 oder 3 → 1);

    if ( alle Scheiben korrekt auf einer
         anderen Saeule (= Ziel-Saeule) gestapelt )
        break;
    else
        bewege nicht die kleinste Scheibe,
           mache den einzig momentan verbleibenden Zug;
}
...
```

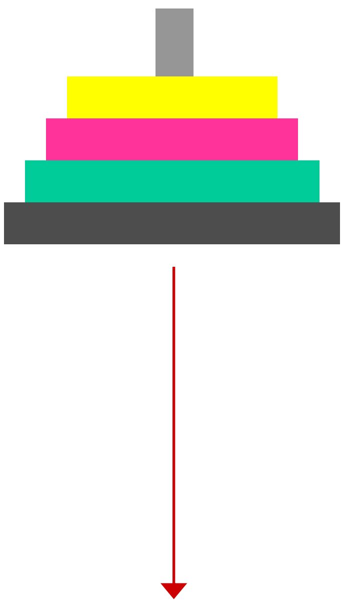
so dass nur eine kleinere Scheibe auf einer größeren liegt …

(nach D. Harel. The Science of Computing. Addison-Wesley, 1989, p.107)

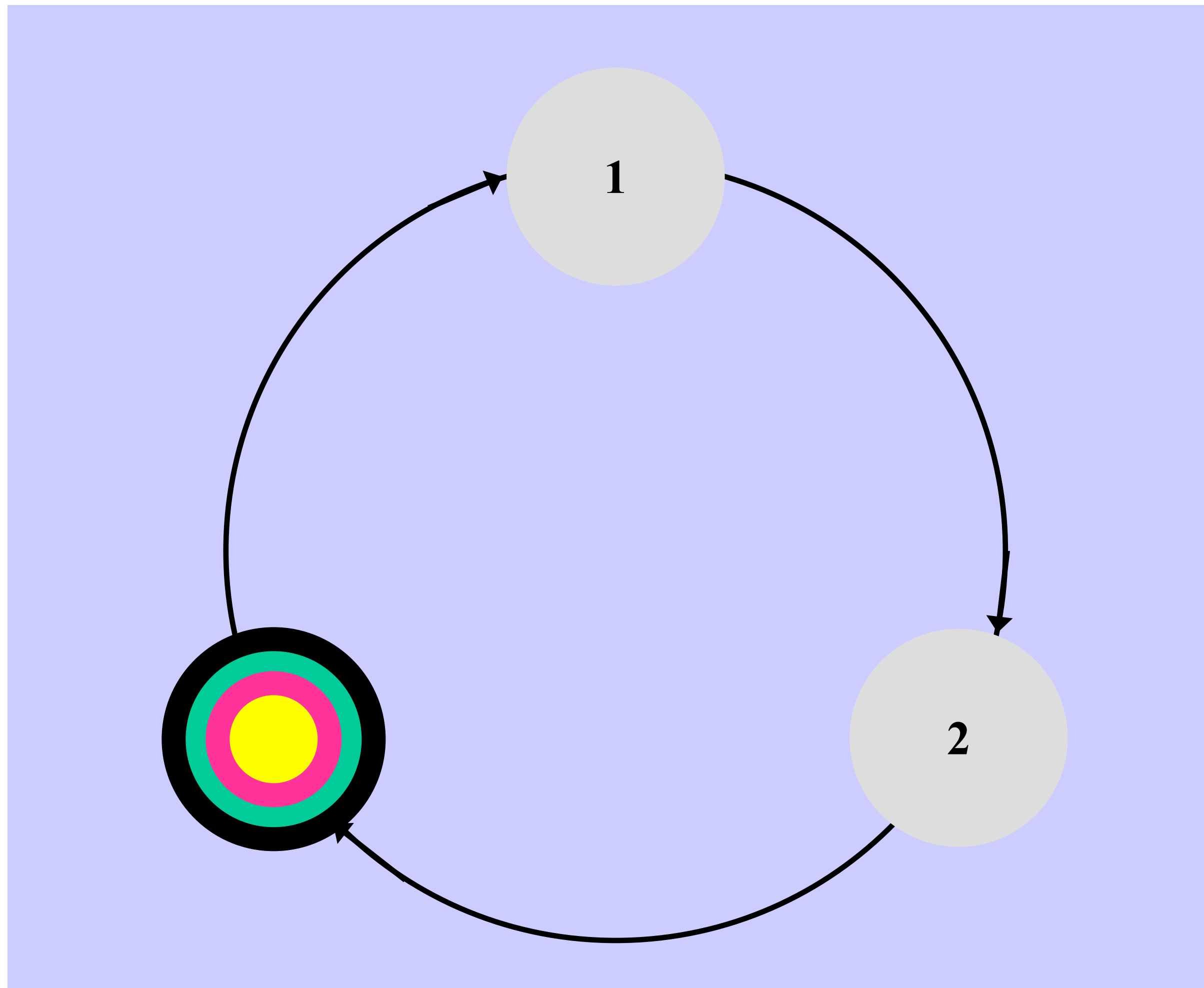
Ablauf

- **<u>Hinweis</u>**: Es kann bei einer gegebenen Start-Säule **keine** Ziel-Säule explizit gewählt werden, diese ergibt sich aus der Anzahl der zu transportierenden Scheiben

- Ablauf für 4 Scheiben
  - Start = 1
  - Transport im Uhrzeigersinn

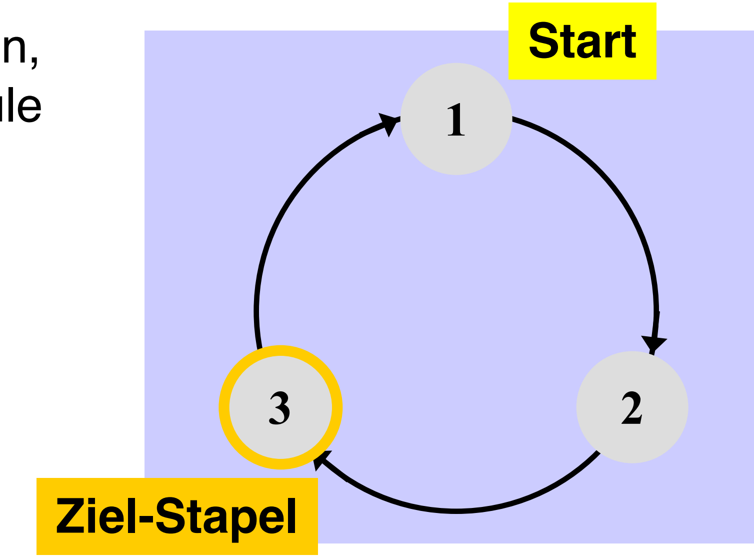


Fertig !



Ergebnisse

- Transport einer **geraden** Anzahl von Scheiben, dann ist der Ziel-Stapel auf der nächsten Säule **gegen den** Uhrzeigersinn



- Transport einer **ungeraden** Anzahl von Scheiben, dann ist der Ziel-Stapel auf der nächsten Säule **im** Uhrzeigersinn