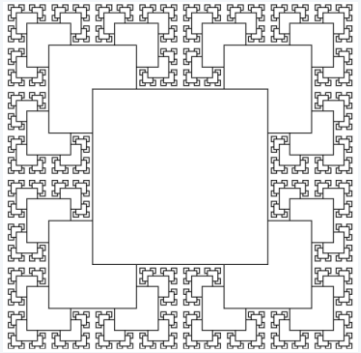
UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA
OBERÖSTERREICH

Studiengang Software Engineering
Fakultät für Informatik, Kommunikation, Medien
Campus Hagenberg

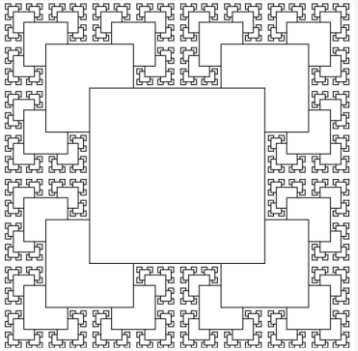
7 Rekursive Algorithmen



7.1 Begriff, Aufbau und Ablauf

Der Begriff Rekursion (auch Rekursivität oder Rekurrenz, engl. *recursion*) wird verwendet, wenn man ausdrücken will, dass sich etwas direkt oder indirekt auf sich selbst bezieht

$$n! = \begin{cases} 1 & n = 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$



"Um Rekursion verstehen zu können, muss man erst einmal Rekursion verstanden haben."

Rekursion ist ein wichtiges Konzept der Mathematik, das oft elegante und kurze Formulierungen ermöglicht

Prinzip und Anwendungsarten

Mathematische Funktionen können rekursiv definiert sein

Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

$$n! = \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1)}_{(n-1)!} \cdot n$$

Größter gemeinsamer Teiler nach Euklid

$$gcd(p,q) = \begin{cases} q & p \bmod q = 0 \\ gcd(q, p \bmod q) & p \bmod q \neq 0 \end{cases}$$

```
r := p mod q
while r ≠ 0 do
  p := q
  q := r
  r := p mod q
end
gcd := q
```

Fibonacci-Zahlen (0, 1, 1, 2, 3, 5, 8, 13, …)

$$fib(m) = \begin{cases} m & m = 0, m = 1 \\ fib(m-1) + fib(m-2) & m > 1 \end{cases}$$

Prinzip und Anwendungsarten

Algorithmen können sich selbst aufrufen (= Rekursiver Algorithmus)

Beispiel: Fakultätsfunktion für n ≥ 1 als rekursiver Algorithmus

```
Factorial(↓n: int): int
begin
  if n ≤ 1 then
    return 1
  else
    return n * Factorial(↓n - 1)
  end
end Factorial
```

laut Spez. n = 1

Rekursiver Aufruf von `Factorial` mit entsprechendem Parameterwert

$$n! = \begin{cases} 1 & n = 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Spezifikation (Überführungsschema)

Aufbau rekursiver Algorithmen

Jeder rekursive Algorithmus hat:

- Mindestens einen nicht-rekursiven Zweig (Rekursionsanker)

```
if n ≤ 1 then
  return 1
```

- Mindestens einen rekursiven Zweig

```
else
  return n * Factorial(↓n - 1)
end -- if
```

- Und der Ausdruck in der Bedingungsprüfung für die Auswahl des nicht-rekursiven oder rekursiven Zweiges muss bei endlicher Rekursionstiefe einen solchen Wert annehmen, dass der nicht-rekursive Zweig durchlaufen wird (sonst: Endlosrekursion!)

```
return n * Factorial(↓n - 1)
```

n wird irgendwann 1

Ablauf rekursiver Algorithmen

Beispiel: Aufruf der rekursiven Fakultätsfunktion

```
var
  fn: int
```

```
fn := Factorial(↓4)
```

24 4 * 6

1. Aufruf
n = 4

```
…
else      4                  4 - 1
  return n * Factorial(↓n - 1)
```

6 3 * 2

2. Aufruf
n = 3

```
…
else      3                  3 - 1
  return n * Factorial(↓n - 1)
```

2 2 * 1

3. Aufruf
n = 2

```
…
else      2                  2 - 1
  return n * Factorial(↓n - 1)
```

1

4. Aufruf
n = 1

```
…  1
if n ≤ 1 then
  return 1
```

Anwendungsbeispiele

Beispiel: Größter gemeinsamer Teiler nach Euklid

```
Gcd(↓p: int ↓q: int): int
begin
  if p mod q = 0 then
    return q
  else
    return Gcd(↓q ↓(p mod q))
  end
end Gcd
```

Aufgabe: Berechnung der Fibonacci-Zahlen (0, 1, 1, 2, 3, 5, 8, 13, …)

```
Fibonacci(↓n: int): int
begin




end Fibonacci
```

Anwendungsbeispiele

Auf wie viele Arten kann man eine ganze Zahl `m` in ganzzahlige Summanden $\leq$ `n` zerlegen?

Gegeben: $m > 0, n > 0$

Gesucht: `S(m, n)`: Anzahl der Zerlegungen von `m` mit Summanden $\leq$ `n`

Beispiel: Zahlenzerlegung für `m = 5, n = 5`

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

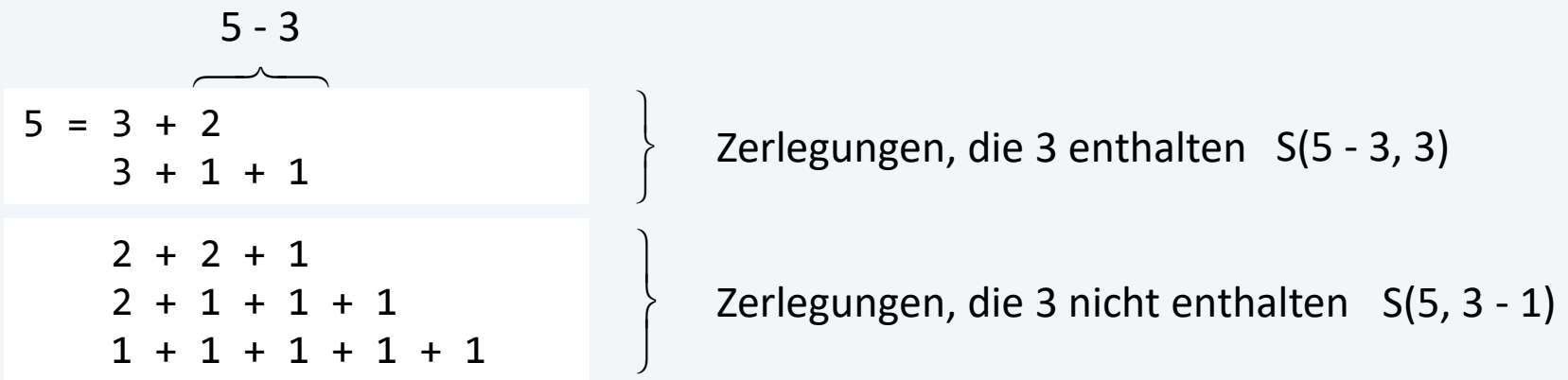
Insgesamt sieben Möglichkeiten: `S(5, 5) = 7`

Anwendungsbeispiele

Lösungsidee

Fall 1 (n = 1)	S(m, 1) =	1	für alle m
Fall 2 (m = 1)	S(1, n) =	1	für alle n
Fall 3 (m = n)	S(m, m) =	1 + S(m, m - 1)	für m > 1
Fall 4 (m < n)	S(m, n) =	S(m, m)	
Fall 5 (m > n)	S(m, n) =	S(m, n - 1) + S(m - n, n)	anschauen

Beispiel: S(5, 3):



Anwendungsbeispiele

Transformation der Lösungsidee in einen Algorithmus

```
S(↓m: int ↓n: int): int
begin
  if (m = 1) or (n = 1) then
    return 1
  elsif m ≤ n then
    return 1 + S(↓m ↓m - 1)
  else
    return S(↓m - n ↓n) + S(↓m ↓n - 1)
  end
end S
```

Fall 1 und Fall 2

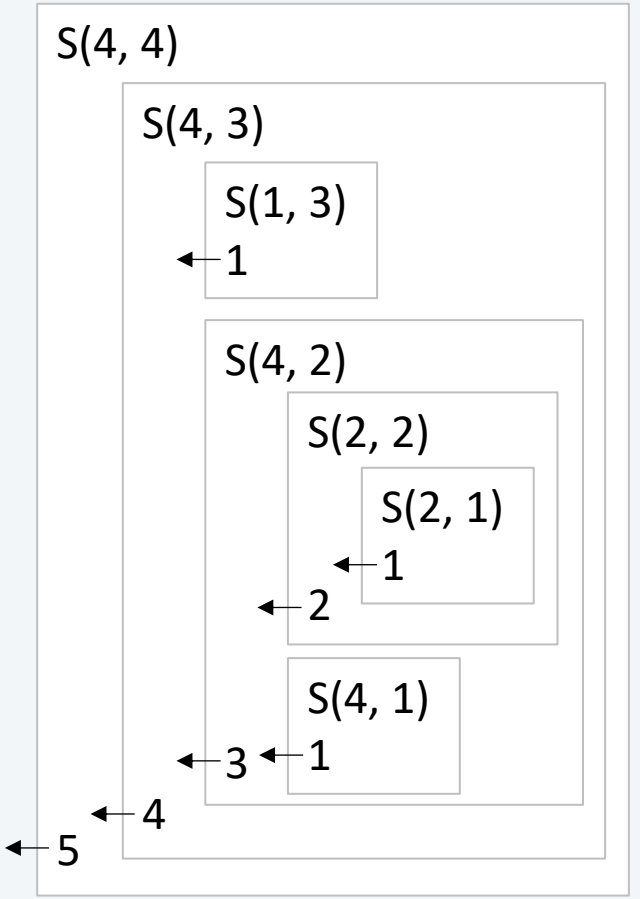
Fall 3 und Fall 4

Fall 5

Algorithmenablauf

Simulation mit m = 4 und n = 4

```
S(↓m: int ↓n: int): int
begin
  if (m = 1) or (n = 1) then
    return 1
  elsif m ≤ n then
    return 1 + S(↓m ↓m - 1)
  else
    return S(↓m - n ↓n) + S(↓m ↓n - 1)
  end
end S
```



Anzahl der gleichzeitigen Aktivierungssätze - relevant für speicher
anzahl der insgesamten Funktionsaufrufen - relevant für Laufzeit

Algorithmenablauf – Das Konzept des Aktivierungssatzes

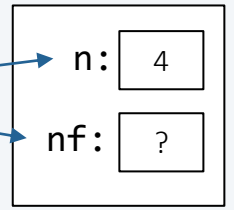
Beim Aufruf eines Algorithmus wird zur Verwaltung seiner Datenobjekte ein sogenannter Aktivierungssatz im Laufzeitkeller (*runtime stack*) angelegt

Beispiel: `Factorial(↓n: int): int` mit dem Aufruf `Factorial(↓4)`

```
Factorial(↓n: int): int
  var
    nf: int
begin
  if n ≤ 1 then
    nf := 1
  else
    nf := n * Factorial(↓n - 1)
  end
  return nf
end Factorial
```

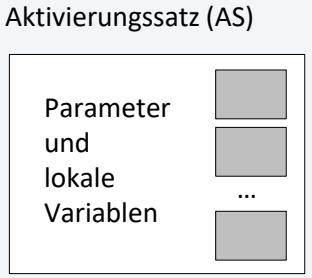
Parameter und lokale Variablen

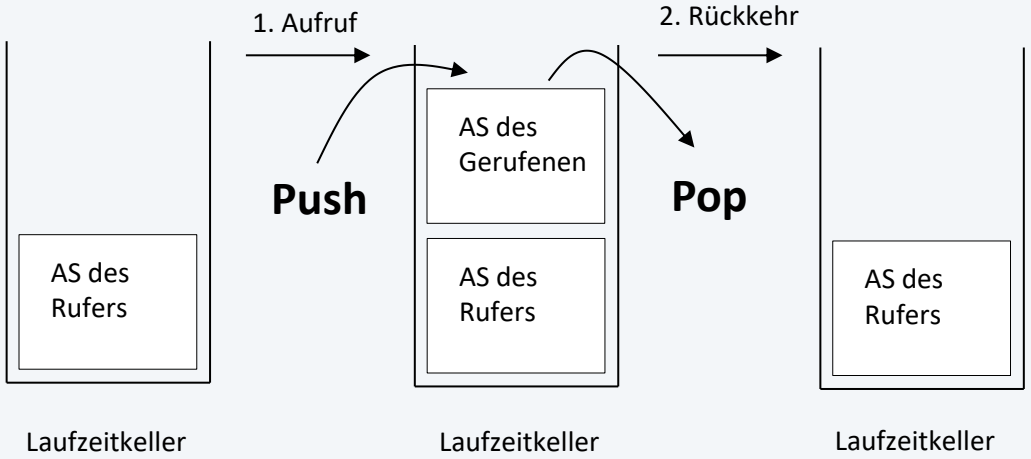
Aktivierungssatz:



Algorithmenablauf – Das Konzept des Aktivierungssatzes

Die Aktivierungssätze werden im Laufzeitkeller (engl. *runtime stack*) verwaltet





Zwei Schritte:

- Bei der Aktivierung eines Algorithmus wird automatisch ein neuer Aktivierungssatz erzeugt und in den Laufzeitkeller eingekellert (Push)
- Wenn die Ausführung eines Algorithmus endet, wird sein Aktivierungssatz automatisch aus dem Laufzeitkeller entfernt (*Pop*)

Bei rekursiven Algorithmen können mehrere Aktivierungssätze für ein und denselben Algorithmus gleichzeitig auf dem Laufzeitkeller liegen

Indirekt rekursive Algorithmen

Wir illustrieren die indirekte Rekursion anhand eines Beispiels

Beispiel: Gesucht sei ein Funktionsalgorithmus `IsPrim(↓x: int): bool`, der für ungerade $x \geq 3$ feststellt, ob x eine Primzahl ist

Lösungsidee: Division von x durch alle Primzahlen 3, 5, 7,11 … $\sqrt{x}$

```
IsPrim(↓x: int): bool
  var
    p: int
begin
  assert x ≥ 3 and x is odd
  p := 3
  while (p * p ≤ x) and (x mod p ≠ 0) do
    p := NextPrim(↓p)
  end
  assert (p * p ≤ x and x is not prim)
         or (p * p > x and x is prim)
  return p * p > x
end IsPrim
```

nur bis sqrt(x), weil:
sei a*b=x
wenn a < als sqrt(x), dann muss b größer sein

wenn z.B. 5 die Zahl x teilt, dann gibt es nach der wurzel eine zweite zahl die mit der zahl multipliziert x ergibt

man muss diese zahlen nur einmal prüfen.
also
z.b. 5 * 20
vs
20 * 5
(erfundene zahlen)

Indirekt rekursive Algorithmen

```
NextPrim(↓p: int): int
  var np: int
begin
  assert: p ≥ 3 and p ist prim
  np := p
  repeat
    np := np + 2
  until IsPrim(↓np)
  return np
end NextPrim
```

+2 weil zu einer primzahl addiert immer eine gerade zahl gibt

- Indirektion ist hier an den Haaren herbeigezogen, weil es bessere Lösungsideen gibt; das Beispiel wurde nur zur Illustration herangezogen.
- Indirekte Rekursion tritt in der Praxis häufig auf, z.B. im Compilerbau

7.2 Klassifikation rekursiver Algorithmen

Rekursive Algorithmen können entsprechend ihres Aufbaus in Klassen eingeteilt werden

Klassifikation nach Broy

- Linear rekursive Algorithmen
- Endrekursive Algorithmen
- Nichtlinear rekursive Algorithmen
- Geschachtelt rekursive Algorithmen
- Indirekt rekursive Algorithmen

Lineare rekursive Algorithmen

Der Aufruf solcher Algorithmen löst nur einen rekursiven Aufruf aus, mehrere Aufrufstellen sind aber möglich, z.B.

```
A(…)
begin
  …
  if … then
    A(…)
  else if … then
    A(…)
  …
end A
```

pro Funktionsaufruf nur ein rekursiver Aufruf
anzahl der rekursiven Aufrufe in Abhängigkeit von n ist linear
also wenn z.B. A(n+1) = A(n)+1
oder A(n+1) = A(n) + x
linear halt

Beispiel: Fakultätsberechnung, GGT nach Euklid

```
Factorial(↓n: int): int
  …
  else
    return n * Factorial(↓n - 1)
  …
```

Endrekursive Algorithmen

wichtig

- Sonderfall linear rekursiver Algorithmen
- Der rekursive Aufruf erfolgt „am Ende“, danach keine weiteren Berechnungen mehr
- Können auf einfache Weise in äquivalente iterative Algorithmen transformiert werden

Beispiel: Größter gemeinsamer Teiler nach Euklid

Rekursiv:

```
Gcd(↓p: int ↓q: int): int
begin
  if p mod q = 0 then
    return q
  else
    return Gcd(↓q ↓(p mod q))
  end
end Gcd
```

Iterativ:

```
Gcd(↓p: int ↓q: int): int
begin
  while p mod q ≠ 0 do
    r := p mod q
    p := q
    q := r
  end
  return q
end Gcd
```

Nichtlinear rekursive Algorithmen

Solche Algorithmen enthalten mindestens einen Zweig, der mehrere rekursive Aufrufe umfasst

Beispiele: Zahlenzerlegung, Fibonacci-Zahlen

```
…
else
  return S(↓m - n ↓n) + S(↓m ↓n - 1)
…
```

```
…
else
  return Fibonacci(↓n - 1) + Fibonacci(↓n - 2)
…
```

steigen stärker als linear - z.B. quadratisch oder exponentiell

Geschachtelt rekursive Algorithmen

Solche Algorithmen haben einen Parameter, der einen rekursiven Aufruf repräsentiert

```
F(↓F(↓x))
```

Können in nichtlinear rekursive Algorithmen umgewandelt werden:

```
a := F(↓F(↓x))
```

```
h := F(↓x)
a := F(↓h)
```

Beispiel: Ackermann-Funktion

A(0, n) =	n + 1
A(m, 0) =	A(m - 1, 1)
A(m, n) =	A(m - 1, A(m, n - 1))

Ackermann(↓2 ↓2)

```
Ackermann(2, 2)
  Ackermann(2, 1)
    Ackermann(2, 0)
      Ackermann(1, 1)
        Ackermann(1, 0)
          Ackermann(0, 1)
        Ackermann(0, 2)
    Ackermann(1, 3)
      Ackermann(1, 2)
        Ackermann(1, 1)
          Ackermann(1, 0)
            Ackermann(0, 1)
          Ackermann(0, 2)
        Ackermann(0, 3)
      Ackermann(0, 4)
  Ackermann(1, 5)
    Ackermann(1, 4)
      Ackermann(1, 3)
        Ackermann(1, 2)
          Ackermann(1, 1)
            Ackermann(1, 0)
              Ackermann(0, 1)
            Ackermann(0, 2)
          Ackermann(0, 3)
        Ackermann(0, 4)
      Ackermann(0, 5)
    Ackermann(0, 6)
```

z.B. für Benchmarks interessant, ansonsten kommt lt prof in der Praxis nicht vor

7.2.5 Indirekt rekursive Algorithmen

Von indirekt rekursiven Algorithmen sprechen wir, wenn sich zwei oder mehr Algorithmen gegenseitig aufrufen

Beispiel: Division von x durch alle Primzahlen 3, 5, 7,11 … $\sqrt{x}$

```
IsPrim(↓x: int): bool
  var
    p: int
begin
  p := 3
  while (p * p ≤ x) and (x mod p ≠ 0) do
    p := NextPrim(↓p)
  end
  return p * p > x
end IsPrim
```

```
NextPrim(↓p: int): int
  var np: int
begin
  np := p
  repeat
    np := np + 2
  until IsPrim(↓np)
  return np
end NextPrim
```

Einer der Algorithmen muss nicht-rekursiven Zweig enthalten

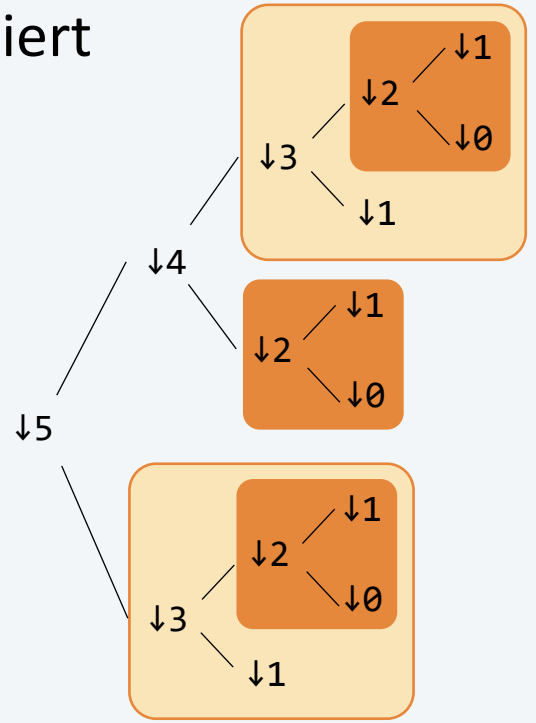
7.3 Typische Anwendungsgebiete

Mathematische Funktionen

- Mathematische Funktionen werden oft rekursiv definiert
- Beispiel: Fibonacci-Zahlen (0, 1, 1, 2, 3, 5, 8, 13, …)

```
Fibonacci(↓n: int): int
begin
  if n ≤ 1 then
    return n
  else
    return Fibonacci(↓n - 1) + Fibonacci(↓n - 2)
  end
end Fibonacci
```

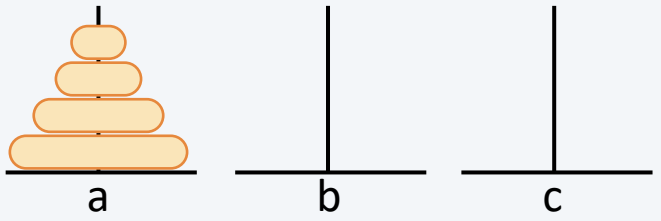
mathematisch sinnvoll, aber softwaretechnisch katastrophal ineffizient



- Viele Berechnungen doppelt und mehrfach, Anzahl der Aufrufe wächst exponentiell, deshalb aus Effizienzgründen meist iterativ implementiert

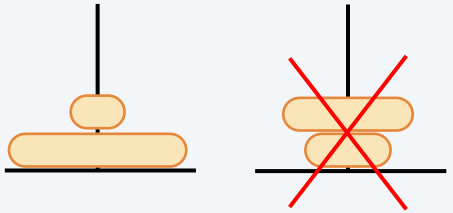
Typische Anwendungsgebiete – Beispiel Türme von Hanoi

Rekursive Problemstellungen wie z. B.



Das Problem der **Türme von Hanoi**:

- Auf einem Stapel *a* liegen *n* Scheiben, die auf einen Stapel *c* (unter Zuhilfenahme eines Stapels *b*) umgeschichtet werden sollen
- Nebenbedingungen:
  - Die Scheiben dürfen nur einzeln bewegt werden
  - Es dürfen nur kleinere Scheiben auf größere Scheiben gelegt werden



- Schnittstelle

```
Move(↓n: int ↓source: char ↓dest: char ↓aux: char)
```

- Aufruf (Beispiel)

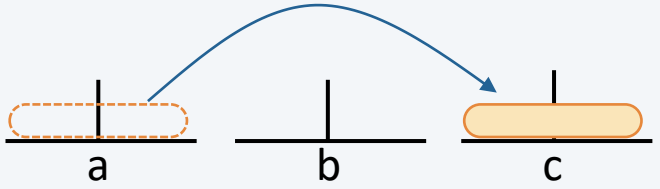
```
Move(↓64 ↓'a' ↓'c' ↓'b')
```

Typische Anwendungsgebiete – Beispiel Türme von Hanoi

Überlegungen zur Lösungsidee

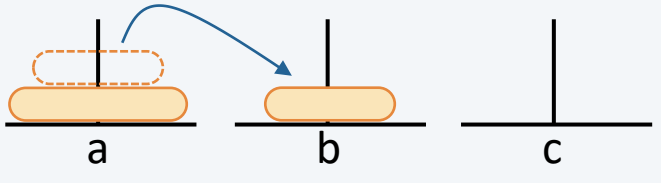
- Fall $n = 1$ (trivial)

  $a \rightarrow c$



- Fall $n = 2$

  $a \rightarrow b, \quad a \rightarrow c, \quad b \rightarrow c$

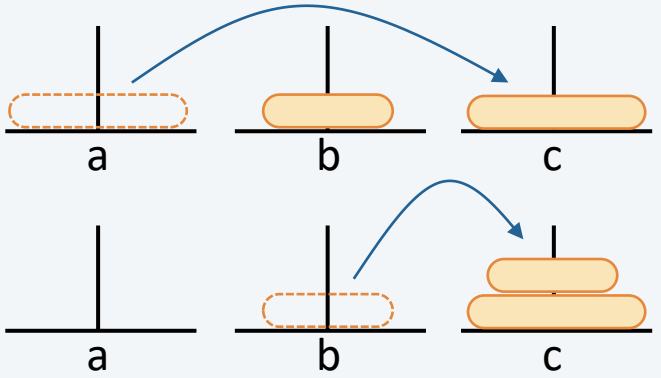


- Fall $n = 3$

  $a \rightarrow c, \quad a \rightarrow b, \quad c \rightarrow b$

  $a \rightarrow c$

  $b \rightarrow a, \quad b \rightarrow c, \quad a \rightarrow c$



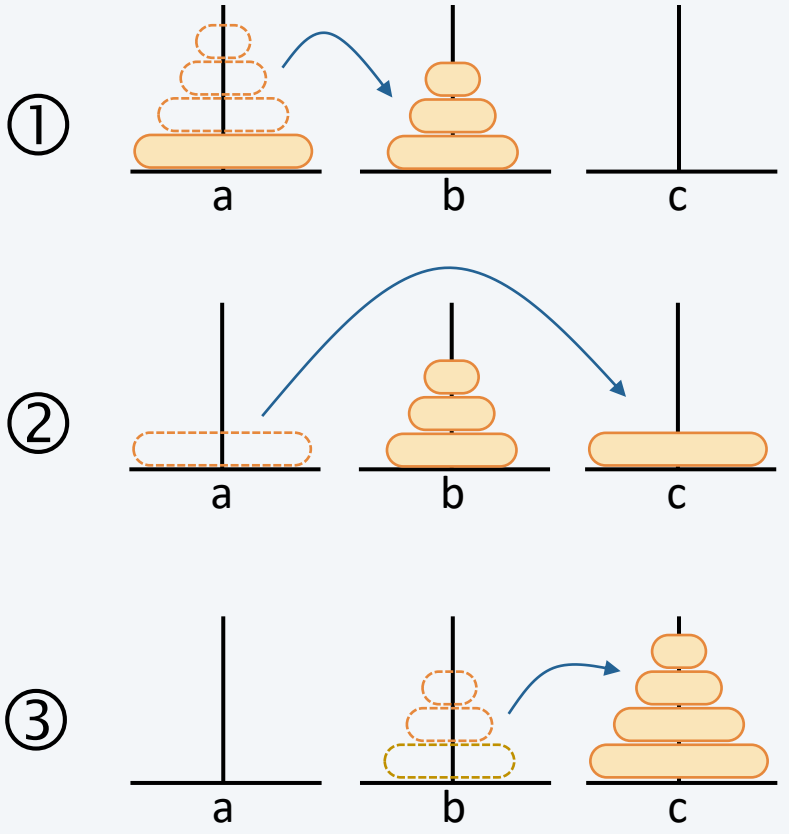
Typische Anwendungsgebiete – Beispiel Türme von Hanoi

Rekursiver Lösungsansatz:

1. Bewege *n* - 1 Scheiben von *a* nach *b*
2. Bewege 1 Scheibe von *a* nach *c*
3. Bewege *n* - 1 Scheiben von *b* nach *c*

Algorithmus

```
Move(↓n: int ↓source ↓dest ↓aux: char)
begin
  if n > 0 then
    Move(↓n - 1 ↓source ↓aux ↓dest)   ①
    Write(↓source ↓"->" ↓dest)        ②
    Move(↓n - 1 ↓aux ↓dest ↓source)   ③
  end
end Move
```



Nichtlineare Rekursion mit leerem Rekursionsanker

Typische Anwendungsgebiete

Manipulation rekursiv definierter vernetzter Datenstrukturen

- Verkettete Listen, Bäume, Graphen
- Rekursive Algorithmen für rekursive Datenstrukturen meist einfach

Beispiel: Ausgabe von Listen-Komponenten in unterschiedl. Richtungen

```
                    ↓
Forward(↓list: ListPtr)
begin
  if list ≠ null then
    WriteLn(↓list→data)
    Forward(↓list→next)
  end
end Forward
```

```
Backward(↓list: ListPtr)
begin
  if list ≠ null then
    Backward(↓list→next)
    WriteLn(↓list→data)
  end
end Backward
```

Backward ist nicht endrekursiv, daher iterative Lösung nicht so einfach

Typische Anwendungsgebiete

Beispiel: Eine ganze Zahl in eine Binärzahl umwandeln

```
PrintBinary(↓n: int)
begin
  if n = 1 then
    Write(↓1)
  else
    PrintBinary(↓n div 2)
    Write(↓n mod 2)
  end
end PrintBinary
```

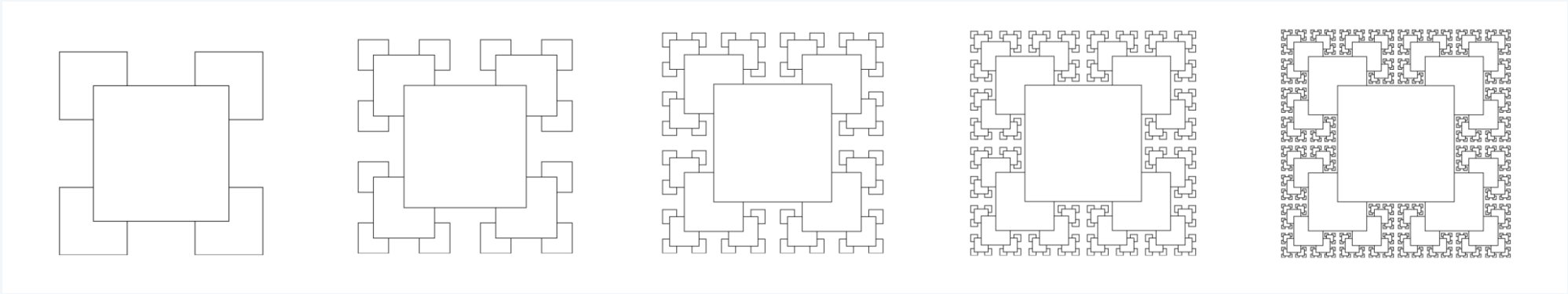
```
var
  n: int
```

```
Read(↑n)
PrintBinary(↓n)
```

n	Ausgabe
1	1
2	10
5	101
50	110010
255	11111111
256	100000000
5461	101010101010101

Typische Anwendungsgebiete: Fraktal

Beispiel: Magische Quadrate



Lösungsidee für Quadrate der Größe `size`

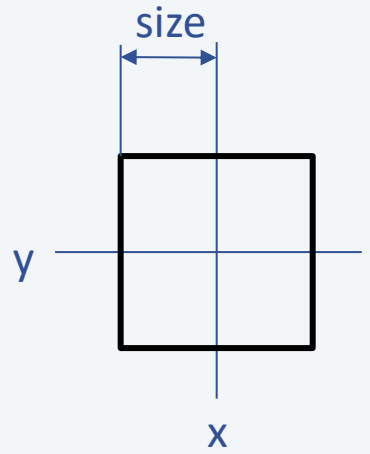
Wenn `size` ≥ 1:

1. zeichne vier Quadrate der Größe `size div 2` zentriert an den Ecken des Quadrats
2. zeichne Quadrat der Größe `size`

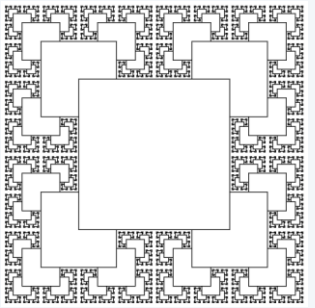
Typische Anwendungsgebiete: Fraktal

Rekursiver Algorithmus

```
Square(↓x: int ↓y: int ↓size: int)
begin
  if size ≥ 1 then
    Square(↓x-size ↓y+size ↓size div 2)
    Square(↓x+size ↓y+size ↓size div 2)
    Square(↓x-size ↓y-size ↓size div 2)
    Square(↓x+size ↓y-size ↓size div 2)
    DrawSquare(↓x ↓y ↓size)
  end
end Square
```




```
Square(↓200 ↓200 ↓100)
```



7.4 Rekursion und Iteration

Grundsätzlich gilt,

- dass man jeden rekursiven in einen iterativen Algorithmus und jeden iterativen in einen rekursiven Algorithmus transformieren kann, und

- dass sich für jede Aufgabenstellung auf mehr oder weniger natürliche Weise eine iterative und eine rekursive Lösungsidee entwickeln lässt

7.4.1 Rekursion → Iteration

- Ggf. zur Verbesserung der Laufzeiteffizienz
- Wenn eine Programmiersprache keine Rekursion bereitstellt
- Ggf. zur Verringerung des Speicherbedarfs

- Im allgemeinen Fall, d. h. wenn wir einen beliebigen rekursiven in einen iterativen Algorithmus transformieren, ist ein eigener Kellerspeicher (Stack) notwendig, der die Rolle des vom Laufzeitsystem verwalteten Laufzeitkellers bei rekursiven Algorithmen übernimmt

Rekursion → Iteration

Ein allgemeines Schema für die Transformation Rekursion → Iteration

- Einführen von Zuständen 0, 1, 2 und 3
- Jeder rekursive Aufruf führt zu neuem Schleifendurchlauf

Rekursion

```
R(↓e)
begin
①   if cond then
      X
②     R(↓e1)
      Y
③     R(↓e2)
      Z
⓪   end
end R
```



```
R(↓e)
begin
  while state ≠ 0 do
    case state of
      1: if cond then X …  state := {0,1}
      2: … Y …             state := {1}
      3: … Z …             state := {0,2,3}
    end
  end
end R
```

Rekursion → Iteration

Ein allgemeines Schema für die Transformation Rekursion → Iteration

```
R(↓e)
begin
  if cond then
    X
    R(↓e1)
    Y
    R(↓e2)
    Z
  end
end R
```

①

```
R(↓e)
begin
  InitStack()
  state := 1
  while state ≠ 0 do
    case state of
    1: if not cond then
         state := 0
       else
         X
         Push(↓lokale Daten)
         Push(↓2)
         e := e1
         state := 1
       end
    2: …
```

damit man nach allen rekursiven aufrufen von R(e1) irgendwann in den Zustand 2 kommen

kommt nicht zur Klausur

Rekursion → Iteration

Ein allgemeines Schema für die Transformation Rekursion → Iteration

```
R(↓e)
begin
  if cond then
    X
    R(↓e1)
    Y
    R(↓e2)
    Z
  end
end R
```

②

```
…          Pfeil umdrehen! muss nach oben
    2: Pop(↓lokale Daten)
       Y
       Push(↓lokale Daten)
       Push(↓3)
       e := e2
       state := 1
    3: Pop(↓lokale Daten)
       Z
       state := 0
    end -- case

    if state = 0 and Stack not empty then
      Pop(↑state)
    end
  end -- while
end R
```

Rekursion → Iteration

Ein allgemeines Schema für die Transformation Rekursion → Iteration

```
R(↓e)
begin
  if cond then
    X
    R(↓e1)
    Y
    R(↓e2)
    Z
  end
end R
```

③ →

```
    …
    2: Pop(↓lokale Daten)
       Y
       Push(↓lokale Daten)
       Push(↓3)
       e := e2
       state := 1
    3: Pop(↓lokale Daten)
       Z
       state := 0
    end -- case

    if state = 0 and Stack not empty then
      Pop(↑state)
    end
  end -- while
end R
```

Rekursion → Iteration

Sonderfall Endrekursion

Rekursion:

```
R(↓e)
begin
  if cond then
    X
    R(↓e1)
  end
end R
```

Iteration:

```
R(↓e)
begin
  while cond do
    X
    e := e1
  end
end R
```

- Nur ein rekursiver Aufruf am Ende des Algorithmus
- Da es nur eine Aufrufstelle gibt und die lokalen Daten daher nach der Aufrufstelle nicht mehr benötigt werden, ist kein Stack erforderlich

7.4.2 Iteration → Rekursion

Jeder iterative Algorithmus lässt sich durch einen äquivalenten rekursiven ersetzen

Schleifen lassen sich (wie folgt) durch rekursive Algorithmen ersetzen:

- Abweisschleife (`while`-Schleife) siehe Beispiel Endrekursion oben
- Durchlaufschleife (`repeat-Schleife`)

Beispiel: Durchlaufschleife

```
R(…)
begin
  repeat
    X
  until cond
end R
```



```
R(…)
begin
  X
  if not cond then
    R(…)
  end
end R
```

Iteration → Rekursion

Beispiel: Zählschleife (`for`-Schleife)

```
R(↓n: int)
  var
    i: int
begin
  for i := 1 to n do
    X
  end
end R
```



```
R(↓n: int ↓i: int)
begin
  if i ≤ n then
    X
    R(↓n ↓i + 1)
  end
end R
```

Lokale Variable `i` wird zu Parameter

```
                  i1  i2  i3  i4
R(↓3 ↓1)          1
R(↓3 ↓2)          1   2
R(↓3 ↓3)          1   2   3
R(↓3 ↓4)          1   2   3   4
```

7.5 Funktionale Programmierung

Jeder „gewöhnliche“ Algorithmus lässt sich durch Funktionsalgorithmen ersetzen

```
P(↓x ↑y ↑z)
```



```
F1(↓x)
```

Funktionsalgorithmus der als Funktionswert y liefert

```
F2(↓x)
```

Funktionsalgorithmus der als Funktionswert z liefert

Jede Verzweigung/`if`-Anweisung lässt sich durch einen Funktionsalgorithmus *If* ersetzen:

```
if expr then
  G(↓x)
else
  H(↓y)
end
```



```
If(↓expr ↓G(↓x) ↓H(↓y))
```

```
If(↓true ↓G(↓x) ↓H(↓y)) = a und
If(↓false ↓G(↓x) ↓H(↓y)) = b

mit G(↓x) liefert a und H(↓y) liefert b
```

Jede Schleife lässt sich durch einen rekursiven Algorithmus ersetzen

Funktionale Programmierung

Die Folge davon ist:

- Keine Zuweisungen und damit keine Variable (im bisher verwendeten Sinn) und kein sequentieller Ablauf mehr
- Algorithmen werden vollständig auf mathematische Funktionen zurückgeführt

Funktionale Programmierung

- Es gibt nur noch Eingangsparameter
- Die Ausgangsparameter (genau einer!) der Algorithmen sind das Funktionsergebnis
- Zuweisungen sind in der Parameterübergabe bei den rekursiven Aufrufen versteckt
- Funktionale Programmiersprachen: LISP, Haskell, Miranda, ML, …

Funktionale Programmierung

Beispiele:

- Absolutbetrag

```
Abs(x) := If(x ≥ 0, x, -x)
```

- Summe der Zahlen 1..n:

```
Sum(n) := If(n = 0, 0, Sum(n - 1) + n)
```

- Euklid

```
Euklid(p, q) := If(p mod q = 0, q, Euklid(q, p mod q))
```

Beispiele

- Binäre Suche in Feldern

```
BinarySearch(a, n, x) := BS(a, 1, n, x)
```

```
BS(a, min, max, x) :=
  If(min > max,
     0,
     If(x = a[(min + max) div 2],
        (min + max) div 2,
        If(x < a[(min + max) div 2],
           BS(a, min, (min + max) div 2 - 1, x)
           BS(a,      (min + max) div 2 + 1, max, x)
        )
     )
  )
```