PM1/PT Ruby: Rekursion

Einführung

- Rekursion: eine Technik um komplexe Probleme durch Zurückführen auf einfachere Probleme zu lösen.

- Kennzeichen:
  - ein Funktion ruft sich selber erneut ein oder mehrmals auf
  - beim Aufruf wird das Problem „reduziert“

- Wir unterscheiden
  - Einfachrekursion <-> Mehrfachrekursion
  - Endrekursion (Tailrecursion) als Spezialform der Einfachrekursion: nur der rekursive Aufruf steht am Ende der Funktion
  - Funktionale <-> Objektrekursion

- Rekursion findet sich in Computeranwendungen, z.B. für die kombinatorische Suche über Bäumen oder die Konstruktion / das Zeichnen von Fraktalen.

Inhalt

- *Einfache Rekursion / Endrekursion – Harmonische Zahlen / ggt*
- Mehrfachrekursion: Towers of Hanoi, Drachenkurve
- Rekursive Grafiken: HTree, Sierpinski Dreieck
- Rekursion mit Speicher: Umformen in Endrekursion
  - Harmonische Zahlen und Fakultaet
  - Fibonacci Zahlen
- Objektrekursion

Einfache Rekursion: Harmonische Zahlen

- Gegeben die Formel: $$h(n) = \sum_{i=1}^{n} \frac{1}{i}$$
- durch Umformung erhalten wir:

$$h(n) = \sum_{i=1}^{n} \frac{1}{i} = \underbrace{1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{(n-1)}} + \frac{1}{n}$$

$$= \sum_{i=1}^{(n-1)} \frac{1}{i} + \frac{1}{n} = h(n-1) + \frac{1}{n}$$

- also $h(n) = \begin{cases} h(n-1) + \frac{1}{n}; \text{für } n > 1 \\ 1; \text{für } n = 1 \end{cases}$

Das rekursive Programm für harmonische Zahlen

$$h(n) = \begin{cases} h(n-1) + \dfrac{1}{n}; f\ddot{u}r\ n > 1 \\ 1; f\ddot{u}r\ n = 1 \end{cases}$$

```ruby
def harm(n)
  if n < 1   # nicht definiert
    -1
  elsif n==1   # Terminierungsbedingung
    1
  else
    harm(n-1) + 1.0/n  # Rekursionsschritt
end
```

harm(n): Verarbeitungstrace

```ruby
def harm(n)
  if n < 1    # nicht definiert
    -1
  elsif n==1    # Terminierungsbedingung
    1
  else
    harm(n-1) +1.0/n   # Rekursionsschritt
end
```

```
=> harm(5)
   => harm(4)
      => harm(3)
         => harm(2)
            => harm(1)
            <= harm(1) r==1
         <= harm(2) r==1.5
      <= harm(3) r==1.8333333333333333
   <= harm(4) r==2.083333333333333
<= harm(5) r==2.283333333333333
```

=> rekursiver Abstieg = rekursiver Aufruf der Funktion

<= rekursiver Aufstieg = Berechnen des Ergebnisses im i‘ten Schritt durch Addition des Ergebnisses im (i-1)‘ten Schritt mit dem Summanden des i‘ten Schrittes

Harmonische Zahlen ein Beispiel für einfache Rekursion

```ruby
def harm(n)
  if n < 1   # nicht definiert
    -1
  elsif n==1   # Terminierungsbedingung
    1
  else
    harm(n-1) +1.0/n  # Rekursionsschritt
end
```

- **Kennzeichen einfacher Rekursion:**
  - Terminierungsbedingung
  - Rekursionsschritt = rekursiver Aufruf der Methode durch sich selbst
  - die Methode ruft sich **nur genau einmal** rekursiv auf (im Code, zur Laufzeit höchstens einmal)
  - das Ergebnis des rekursiven Aufrufs kann mit weiteren Berechnungen verknüpft sein

Endrekursion: größter gemeinsamer Teiler (ggt)

- Problembeschreibung: Der ggt zweier Zahlen n und m ist wie folgt definiert:
  - ggt(n,m) ist m, wenn n ein Vielfaches von m ist (Rest: n%m=0)
  - ggt(n,m) ist ggt(m,n%m), wenn n kein Vielfaches von m ist
  - **Abbruch**: ist immer gegeben für positive Zahlen, da der Rest immer kleiner und schließlich 0 wird.
- rekursive Formulierung:

$$ggt(n,m) = \begin{cases} m; wenn\ n\%m = 0 \\ ggt(m, n\%m); wenn\ n\%m \neq 0 \end{cases}$$

Das rekursive Programm für den ggt

$$ggt(n,m) = \begin{cases} m; wenn\ n\%m = 0 \\ ggt(m, n\%m); wenn\ n\%m \neq 0 \end{cases}$$

```
def ggt(n,m)
  if m==0
   n    # Terminierungsbedingung
  else
    ggt(m,n%m) # Rekursionschritt
end
```

ggt - ein Beispiel für Endrekursion

```
def ggt(n,m)
  if m==0
    n    # Terminierungsbedingung
  else
    ggt(m,n%m) # Rekursionschritt
  end
end
```

- **Kennzeichen von Endrekursion**:
  - Terminierungsbedingung
  - Rekursionsschritt = rekursiver Aufruf der Methode durch sich selbst
  - beim rekursiven Aufruf wird **nur** die Methode aufgerufen (keine Verknüpfung mit anderen Operationen)

ggt(n,m)-Verarbeitungstrace

```
=> ggt(25,40)
   => ggt(40,25)
      => ggt(25,15)
         => ggt(15,10)
            => ggt(10,5)
               => ggt(5,0)
               <= ggt(5,0) r==5
            <= ggt(10,5) r==5
         <= ggt(15,10) r==5
      <= ggt(25,15) r==5
   <= ggt(40,25) r==5
<= ggt(25,40) r==5
```

```ruby
def ggt(n,m)
  if m==0
   n     # Terminierungsbedingung
  else
    ggt(m,n%m) # Rekursionschritt
  end
end
```

$\Longrightarrow$ rekursiver Abstieg = rekursiver Aufruf der Funktion mit Reduktion über beide Parameter

<= rekursiver Aufstieg = Rückgabe des Ergebnisses, das bei der Terminierung im Parameter n steht.

Ü-9-b-1

- Berechnen Sie bitte die Formel sum(x,n) rekursiv

$$sum(x,n) = \sum_{i=1}^{n} \frac{(x-1)^i}{i * x^i}, für\ x > 0.5$$

- Entwickeln Sie eine rekursive Formulierung für die Formel.

- Schreiben Sie eine Methode in Ruby, die die Formel berechnet.
- Was wird hier berechnet?

Inhalt

- Einfache Rekursion - Endrekursion – Harmonische Zahlen, ggt
- **Mehrfachrekursion: Towers of Hanoi, Drachenkurve**
- Rekursive Grafiken: HTree, Sierpinski Dreieck
- Rekursion mit Speicher: Umformen in Endrekursion
  - Harmonische Zahlen und Fakultät
  - Fibonacci Zahlen
- Objektrekursion

Mehrfachrekursion

- Mehrfacher rekursiver Aufruf einer Funktion durch sich selbst

  → Es entstehen Rekursionsbäume

    - jeder Knoten entspricht einem Funktionsaufruf
    - von jedem Knoten gehen mehrere Funktionsaufrufe aus

  → Exponentieller Aufwand für die Berechnung (Vorlesung AD und/oder LB)

- Reduktion des Problems über Parameter des Aufrufs

Mehrfachrekursion: Towers Of Hanoi

- Problembeschreibung:
  - Gegeben ein Stapel von n Scheiben unterschiedlicher Größe, sowie 3 Stapel
  - Ziel: n Scheiben von einem Stapel auf den anderen zu verschieben
  - Randbedingung:
    - bei jedem Verschieben darf immer nur eine kleiner Scheibe auf der Größeren liegen.
    - es darf pro Schritt immer nur eine Scheibe bewegt werden.

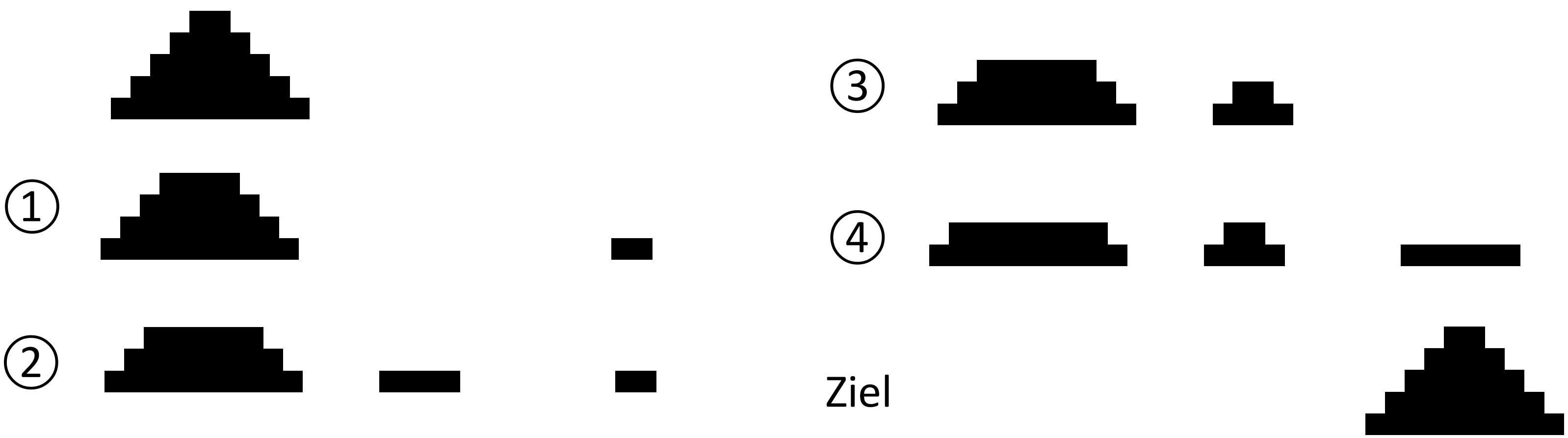
Mehrfachrekursion: Towers Of Hanoi

- **Lösungsidee**:
  - Wir führen die Lösung bis auf das Verschieben der kleinsten Scheibe zurück.
  - Wenn die kleinste Scheibe erreicht ist, dann verschieben wir diese nach links.
  - Dann nehmen wir die nächst größere Scheibe und verschieben diese nach rechts (! links).
  - Anschließend können wir die kleinste Scheibe nehmen und diese nach links verschieben.
  - Dann liegt die kleinste Scheibe auf der nächst größeren
  - Im nächsten Schritt nehmen wiederum die nächst größere und verschieben diese nach links (der Platz ist jetzt frei)
  - u.s.w.

Mehrfachrekursion: Towers Of Hanoi

- **Rekursive Formulierung:**
  - Methode zum Verschieben: move(left, n)

$$move(left, n) = \begin{cases} move(!left, n-1), versetze(left, n), move(!left, n-1); n > 0 \\ fertig; n = 0 \end{cases}$$



Towers Of Hanoi – das Programm

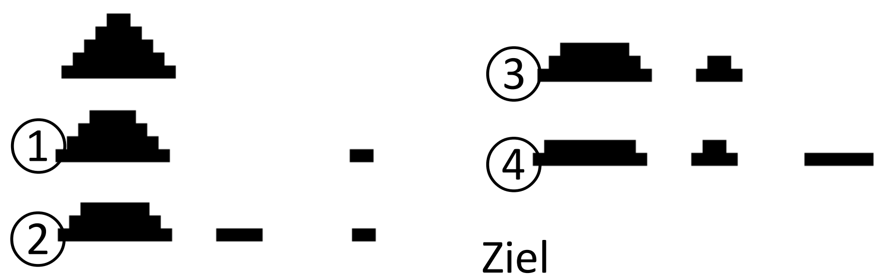
```ruby
class TowersOfHanoi

  def move(left,n,depth=0)

    if (n==0)
      return nil
    end

    move(!left,n-1,depth+1)

    if left
      puts "#{"\t"*depth}t(#{depth}):: Scheibe #{n} <= "
    else
      puts "#{"\t"*depth}t(#{depth}):: Scheibe #{n} => "
    end

    move(!left,n-1,depth+1)
  end
end
```

Rekursionsbaum für Towers Of Hanoi mit 5 Scheiben

```
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
        t(2):: Scheibe 3 <=
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
    t(1):: Scheibe 4 =>
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
        t(2):: Scheibe 3 <=
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
t(0):: Scheibe 5 <=
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
        t(2):: Scheibe 3 <=
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
    t(1):: Scheibe 4 =>
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
        t(2):: Scheibe 3 <=
                t(4):: Scheibe 1 <=
            t(3):: Scheibe 2 =>
                t(4):: Scheibe 1 <=
```

Berechnungsaufwand T(n) für n

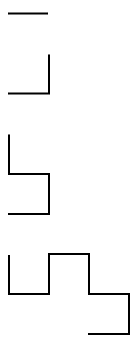
$T(n) = 2^n - 1$



Mehrfachrekursion: Drachenkurve

- **Problembeschreibung**: Zeichne eine Drachenkurve der Ordnung n nach folgender Konstruktionsvorschrift:
  - zeichne eine Drachenkurve der Ordnung (n-1)
  - drehe nach links (gibt „L“ aus)
  - zeichne eine umgekehrte Drachenkurve der Ordnung (n-1)
    - eine umgekehrte Drachenkurve zeichnet eine Drachenkurve mit Rechtsdrehung (Rechtsdrehung gibt „R“ aus)
- Abbruchbedingung: n=0 zeichne eine gerade Linie (gib „F“ aus)

Muster der Drachenkurve für n=0,1,2,3



```
Drachenkurve Ordnung 0: F
Drachenkurve Ordnung 1: FLF
Drachenkurve Ordnung 2: FLFLFRF
Drachenkurve Ordnung 3: FLFLFRFLFLFRFRF
```

Mehrfachrekursion: Drachenkurve

- **Rekursive Formulierung:**

$$z(n) = \begin{cases} geradeaus(); n = 0 \\ z(n-1), links\_drehen(), z\_rechts(n-1); n > 0 \end{cases}$$

$$z\_rechts(n) = \begin{cases} geradeaus(); n = 0 \\ z(n-1), rechts\_drehen(), z\_rechts(n-1) \end{cases}$$

$$links\_drehen() : gibt\ L\ aus$$

$$rechts\_drehen() : gibt\ R\ aus$$

$$geradeaus() : gibt\ F\ aus$$

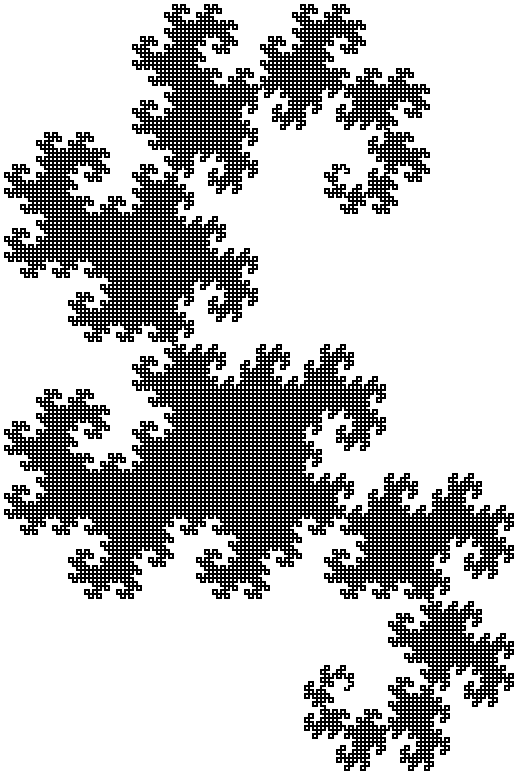
Rekursionsbaum der Drachenkurve für n=3

(Projekt V9-b Rekursion)

```
                        t(3): dk(0) - geradeaus
                t(2): dk(1) links drehen
                        t(3): dk(0) - geradeaus
        t(1): dk(2) links drehen
                        t(3): dk(0) - geradeaus
                t(2): dk(1) rechts drehen
                        t(3): dk(0) - geradeaus
t(0): dk(3) links drehen
                        t(3): dk(0) - geradeaus
                t(2): dk(1) links drehen
                        t(3): dk(0) - geradeaus
        t(1): dk(2) rechts drehen
                        t(3): dk(0) - geradeaus
                t(2): dk(1) rechts drehen
                        t(3): dk(0) - geradeaus
```

DrachenkurveMitRekursionsBaum.rb

Drachenkurve der Ordnung (n=14)
(-> Projekt V9-b RekursiveGrafik)



Inhalt

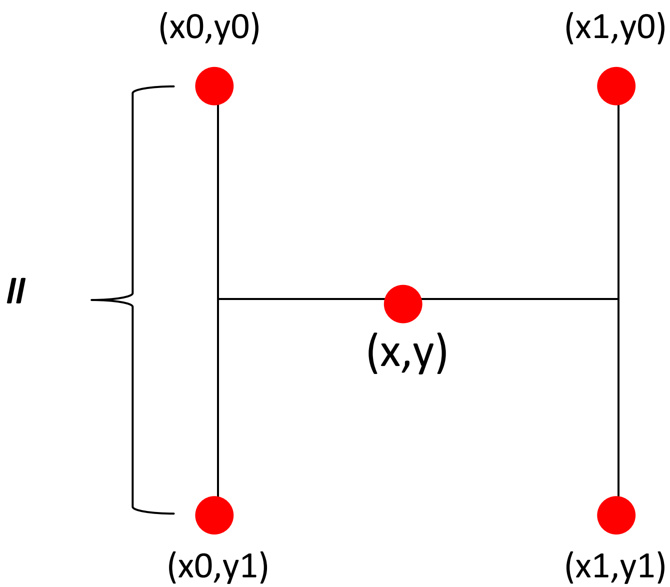
- Einfache Rekursion: Endrekursion – Harmonische Zahlen, ggt
- Mehrfachrekursion: Towers of Hanoi, Drachenkurve
- **Rekursive Grafiken: HTree, Sierpinski Dreieck**
- Rekursion mit Speicher: Umformen in Endrekursion
  - Harmonische Zahlen und Fakultät
  - Fibonacci Zahlen
- Objektrekursion

Rekursive Grafiken: HTree

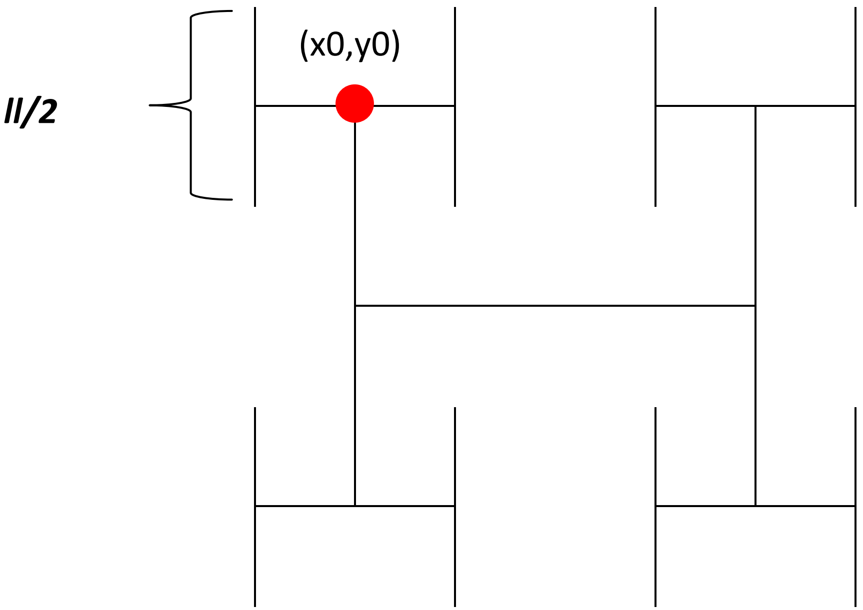
- Problembeschreibung:
  - Zeichne ein ***H*** mit Mittelpunkt ***(x,y)*** und Linienlänge ***ll***
  - Alle Linien des H haben gleiche Längen
  - Wiederhole die folgende Konstruktionsvorschrift n-mal
    - Bestimme die äußeren Punkte (x0,y0),(x0,y1)(x1,y1)(x1,y0) des H
    - Wähle die Punkte als neue Mittelpunkte und zeichne ein H der halben Länge ***ll/2*** des ursprünglichen H‘s
  - Abbruch bei n=1

HTree Konstruktion

n=1



n=2



Rekursive Grafiken: HTree

- Rekursive Formulierung:

$$z(n,x,y,ll)=\begin{cases} fertig\,;n=1 \\ h(x,y,ll),z(n-1,x_0,y_0,\frac{ll}{2}),z(n-1,x_0,y_1,\frac{ll}{2}),z(n-1,x_1,y_1,\frac{ll}{2}),z(n-1,x_1,y_0,\frac{ll}{2});n>1 \end{cases}$$

$h(x,y,ll)$ *zeichnet das H*

$$x_0=x-\frac{ll}{2}$$

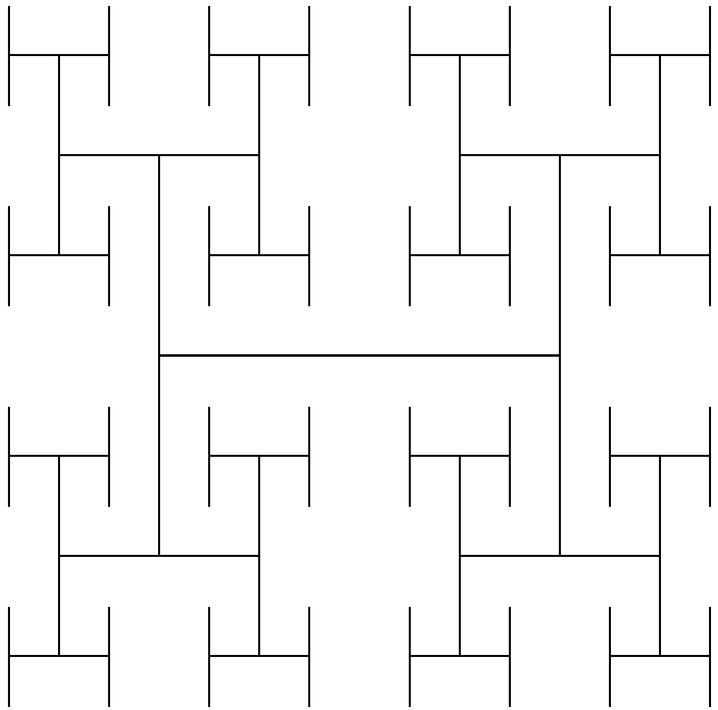
$$y_0=y+\frac{ll}{2}$$

$$x_1=x+\frac{ll}{2}$$

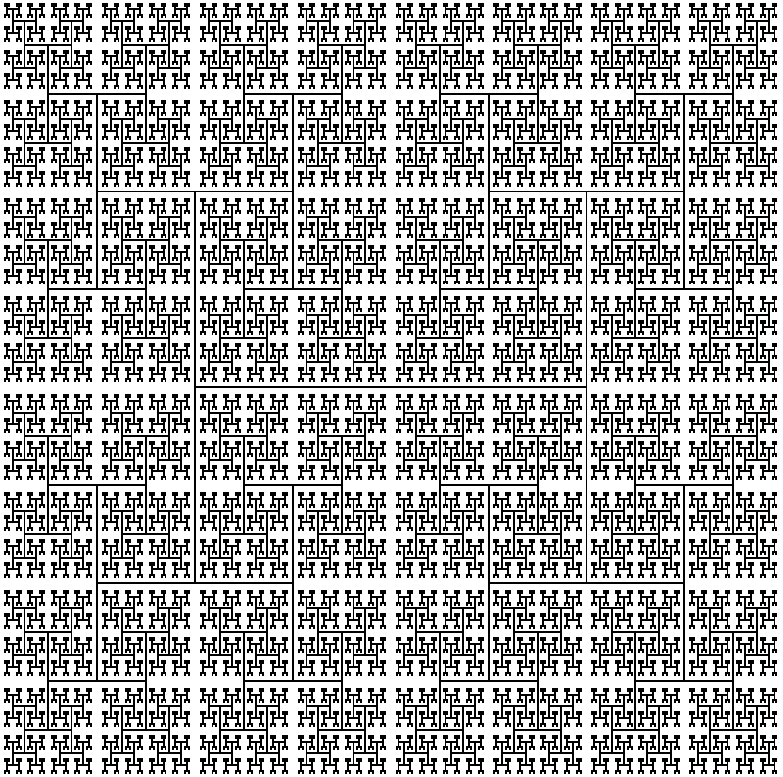
$$y_1=y-\frac{ll}{2}$$

HTree Beispiele

n=3



n=8



HTree – Programm (Auszug)

```
class HTree
  # Initialisierung für die grafische Animation
  def zeichnen(n=2, x=300,y=300, ll=200) ... end # Vorbereitung für grafische Animation

  def _zeichnen(n,x,y,ll)
    x0 = x-ll/2
    x1 = x+ll/2
    y0 = y+ll/2
    y1 = y-ll/2

    zeichne_h([x0,y0,x0,y1,x0,y,x1,y,x1,y0,x1,y1])

    return if (n==1)

    _zeichnen(n-1,x0,y0,ll/2)
    _zeichnen(n-1,x0,y1,ll/2)
    _zeichnen(n-1,x1,y1,ll/2)
    _zeichnen(n-1,x1,y0,ll/2)
  end

  def zeichne_h(xys,frequenz=20) ...  end # zeichnen des H‘s mit Zeitverzögerung
end
```

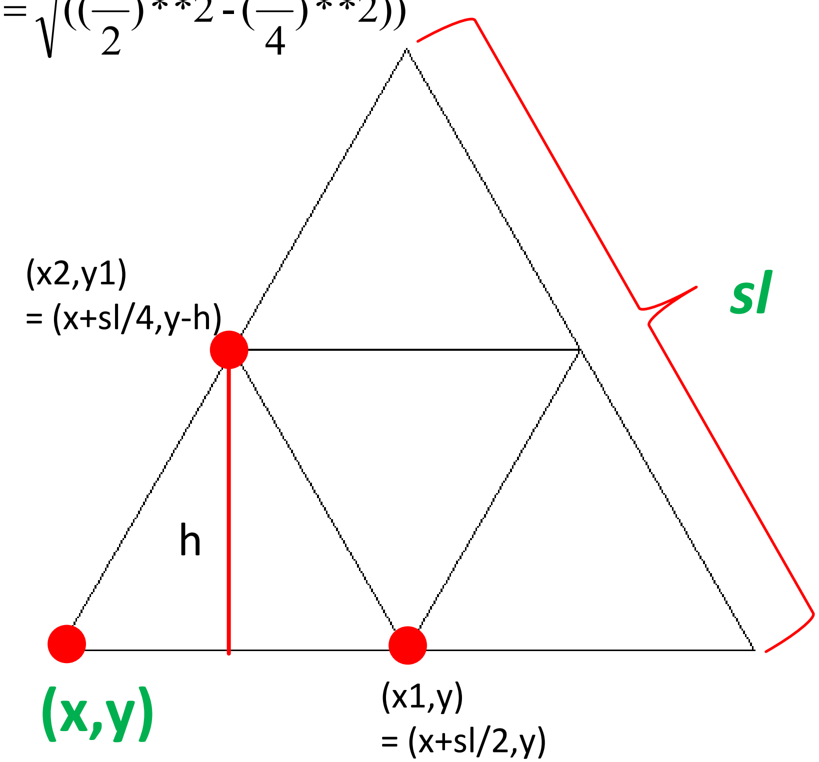
Rekursive Grafiken: Sierpinski Dreieck

- **Problembeschreibung**:
  - Zeichne ein gleichseitiges x-achsenparalleles Sierpinksi Dreieck der Seitenlänge ***sl*** mit dem äußerst linken Punkt ***(x,y)*** der Ordnung ***n*** nach folgender Konstruktionsvorschrift
    - zeichne eine Dreieck für ***(x,y)*** mit Kantenlänge n
    - halbiere jede Seite
    - Zeichne mit den sich ergebenen Punkten 3 Sierpinski Dreiecke der Ordnung ***(n-1)***
  - Abbruch: für ***n=0***

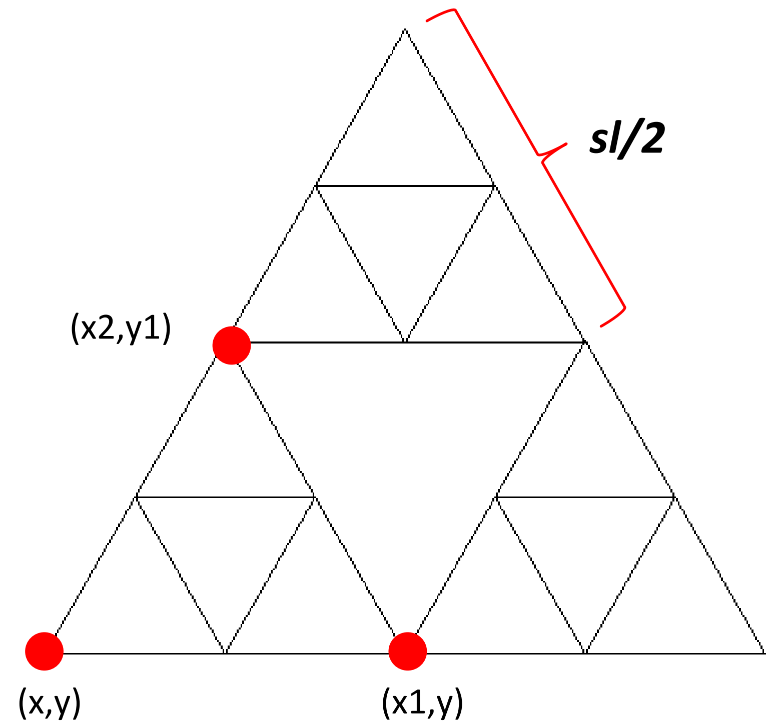
Sierpinski Dreieck: Konstruktion

n=1

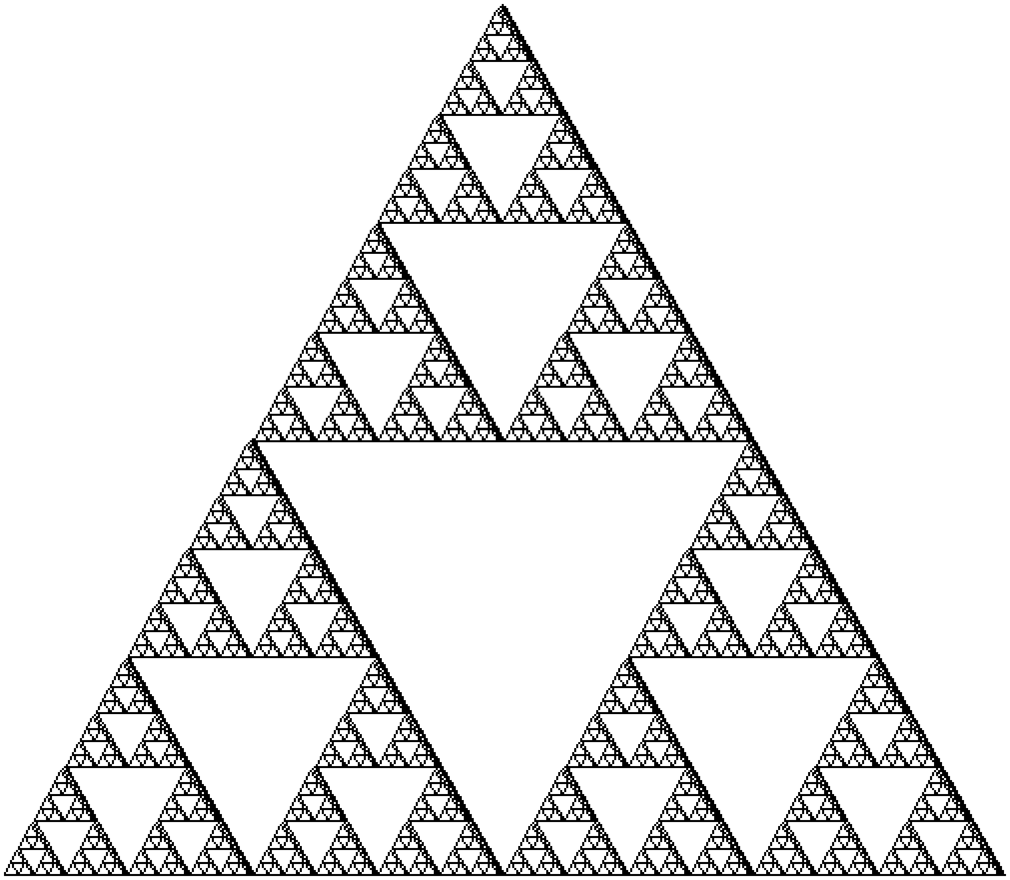
$$h = \sqrt{((\frac{sl}{2})**2 - (\frac{sl}{4})**2))}$$



n=2



Sierpinski Dreieck der Ordnung 6



Ü-9-b-2

- Entwickeln Sie eine rekursive Formulierung für die Lösung des Sierpinski Dreiecks n‘ter Ordnung.
- Nehmen Sie dafür an, dass Sie eine Methode ***d(x,y,sl)*** haben, die ein gleichseitiges Dreieck zeichnet.

Rekursive Grafiken: Sierpinski Dreieck

- **Rekursive Formulierung**:

$$z(n,x,y,sl) = \begin{cases} fertig\,; n = 1 \\ d(x,y,sl), z(n-1,x,y,\frac{sl}{2}), z(n-1,x_1,y,\frac{sl}{2}), z(n-1,x_2,y_1,\frac{sl}{2}); n > 1 \end{cases}$$

$$d(x,y,sl)\ zeichnet\ das\ Dreieck$$

$$x_1 = x + \frac{sl}{2}$$

$$y_1 = (\mathrm{y} - \sqrt{((\frac{sl}{2})**2 - (\frac{sl}{4})**2)})$$

$$x_2 = x + \frac{sl}{4}$$

Sierpinski Dreieck – das Programm (Auszug)

```
class SierpinskiDreieck
 # Initialisierung für die grafische Animation
  def zeichnen(n=3,x=100,y=400,sl=400) ... end  # Vorbereitung für grafische Animation

  def _zeichnen(n,x,y,sl)
    x1 = x +sl/2
    x2 = x + sl/4
    y1 = (y - Math.sqrt((sl/2)**2 -(sl/4)**2)).round()

    zeichne_dreieck(x,y,sl)

    # Rekursions Abbruch
    return if n ==0

    # Rekursion
    _zeichnen(n-1,x,y,sl/2)
    _zeichnen(n-1,x1,y,sl/2)
    _zeichnen(n-1,x2, y1,sl/2)
  end

  def zeichne_dreieck(x,y,sl,f=50) #coords,frequenz=50) ...end
            # zeichnen des Dreiecks mit Zeitverzögerung
end
```

Eigenschaften von Mehrfach-Rekursionen

- Mehrmaliger Aufruf (anzahl = ***k***) einer Funktion durch sich selbst.
- Reduktion auf ein kleineres Problem über eine Größe. In den Beispielen über die Schritte ***n.***
- Abbruchbedingung: Kriterium für das Beenden des rekursiven Aufrufs.
- Darstellung der Verarbeitung nur durch einen Rekursionsbaum möglich.
- Berechnungsaufwand exponentiell abhängig von ***k*** und ***n***: $k^{n-c1}-c2$

Probleme mit Rekursion

- Fehlende Abbruchbedingung → Stack Overflow (stack level too deep)
- Fehlende Konvergenz durch fehlende Reduktion → Stack Overflow
- Exponentieller Aufwand → für große ***n*** ist das Problem mit einem Rechner nicht lösbar (Stack Overflow)

Inhalt

- Einfache Rekursion: Endrekursion – Harmonische Zahlen, ggt
- Mehrfachrekursion: Towers of Hanoi, Drachenkurve
- Rekursive Grafiken: HTree, Sierpinski Dreieck
- **Rekursion mit Speicher: Umformen in Endrekursion**
  - **Harmonische Zahlen und Fakultät (mathematisch, nicht Hochschulorganisation)**
  - **Fibonacci Zahlen**
- Objektrekursion

Rekursion mit Speicher / Umformen in Endrekursion

- **Ausgangspunkt**: eine einfach rekursive Funktion, die nicht endrekursiv ist
- **Ziel**: Umformen in eine endrekursive Funktion
- **Idee**: Berechnung mit dem Ergebnis des rekursiven Aufruf, in den rekursiven Aufruf „hineinziehen“
- **Lösung**: Verwenden eines Speichers, mit dem die Berechnung bereits beim rekursiven Abstieg erfolgt

Rekursion mit Speicher für die harmonische Reihe

```ruby
# Harmonische Reihe mit Speicher/Akkumulator
def harm_acc(n, speicher = 0)
  return -1 if n < 0                          # nicht definiert
  return speicher  if n == 0               # Terminierungsbedingung
  return harm_acc(n-1,speicher+1.0/n)  # Rekursionsschritt
end
```

Die Addition des Summanden für das aktuelle n erfolgt beim rekursiven Aufruf als Operation auf dem Speicher.
Der Speicher muss zu Beginn mit dem neutralen Element für die jeweilige Operation initialisiert werden (für Addition ist das die 0).

Im Speicher stehen also nacheinander die Werte
1/n,
1/n+1/(n-1), …
1/n+1/(n-1) + … + 1/3 + 1/2 +1

harm_acc(n) - Verarbeitungstrace

```
def harm_acc(n,speicher=0)
  return -1 if n < 0                    # nicht defini
  return speicher if n == 0             # Terminierung
  return harm_acc(n-1,speicher+1.0/n)   # Rekursionssc
end
=> harm_acc(4,0)
   => harm_acc(3,0.25)
      => harm_acc(2,0.5833333333333333)
         => harm_acc(1,1.0833333333333333)
            => harm_acc(0,2.083333333333333)
            <= harm_acc(0,2.083333333333333) r==2.083333333333333
         <= harm_acc(1,1.0833333333333333) r==2.083333333333333
      <= harm_acc(2,0.5833333333333333) r==2.083333333333333
   <= harm_acc(3,0.25) r==2.083333333333333
<= harm_acc(4,0) r==2.083333333333333
```

- Das Ergebnis steht im *speicher*, wenn die Terminierungsbedingung erfüllt ist.
- *harm_acc* ist endrekursiv.

Rekursion mit Speicher für die harmonische Reihe Variante

```ruby
# Harmonische Reihe mit Speicher Variante
def harm_acc_v1(n,i=1,speicher=0)
  return speicher if i > n
  return harm_acc_v1(n,i+1,speicher+1.0/i)
end
```

Statt n in jedem Rekursionsschritt zu reduzieren, wird eine weiterer Parameter i von 1 bis n hochgezählt und der Summand 1.0/i auf den Speicher addiert.
Die Terminierungsbedingung ist hier, dass i > n wird. Dann steht das Ergebnis im Speicher.

Im Speicher stehen also nacheinander die Werte

1,

1+1/2, ...

1+1/2+ ... + 1/(n-1) +1/n

harm_acc_v1(n) - Verarbeitungstrace

```ruby
def harm_acc_v1(n,i=1,speicher=0)
  return speicher if i > n
  return harm_acc_v1(n,i+1,speicher+1.0/i)
end
```

```
=> harm_acc_v1(4,1,0)
   => harm_acc_v1(4,2,1.0)
      => harm_acc_v1(4,3,1.5)
         => harm_acc_v1(4,4,1.8333333333333333)
            => harm_acc_v1(4,5,2.083333333333333)
            <= harm_acc_v1(4,5,2.083333333333333) r==2.083333333333333
         <= harm_acc_v1(4,4,1.8333333333333333) r==2.083333333333333
      <= harm_acc_v1(4,3,1.5) r==2.083333333333333
   <= harm_acc_v1(4,2,1.0) r==2.083333333333333
<= harm_acc_v1(4,1,0) r==2.083333333333333
```

- *harm_acc_v1* ist endrekursiv.
- Die Werte entwickeln sich in umgekehrter Reihenfolge zu der ersten Lösung.

Von der Endrekursion zur Iteration

- Die Variante ***harm_acc_v1*** der Rekursion mit Speicher lässt sich schematisch in eine iterative Lösung verwandeln
  - ***speicher*** ist die Variable, in der die Ergebnisse akkumuliert werden. Der Startwert des Speichers wird für die Initialisierung einer lokalen Variable übernommen. (im Beispiel ***speicher=0***)
  - Das ***i*** ist der Laufindex des for-Iterators.
  - Der Startwert für ***i*** ist die untere Grenze des Intervalls (im Beispiel ***i=1***)
  - Der Wert, gegen den in der Terminierung geprüft wird, ist die obere Grenze des Intervalls (im Beispiel ***i > n***)
  - Die Operation mit dem Speicher beim rekursiven Aufruf ist die Berechnung für jeden Iterationsschritt (im Beispiel: ***speicher+1.0/i***)
  - Das Berechnungsergebnis muss in jedem Iterationsschritt dem Speicher zugewiesen.

Endrekursive Lösung *harm_acc_v1* versus iterative Lösung *harm_iter*

```ruby
def harm_acc_v1(n,i=1,speicher=0)
  return speicher if i > n
  return harm_acc_v1(n,i+1,speicher+1.0/i)
end
```

```ruby
# iterative Lösung
def harm_iter(n)
  speicher = 0
  for i in (1..n)
    speicher = speicher + 1.0/i
  end
  speicher
end
```

Da die Umformung schematisch ist, gibt es Programmiersprachen, die diese Umformung automatisch vornehmen (Tail recursion optimization - TRO).

Die rekursive Formulierung von Problemen ist häufig leichter / kürzer. Die iterative Lösung ist schneller und benötigt weniger Platz.

Best of both worlds bekommen Sie durch Umformen in Endrekursion bei Programmiersprachen mit TRO.

Rekursion mit Speicher für Fibonacci Zahlen

- **Problembeschreibung**:
  - Eine Fibonacci Folge ist eine Folge von Zahlen, in der sich die n'te Fibonacci-Zahl aus der Summe der (n-1)'ten und der (n-2)'ten Fibonacci Zahlen berechnet.

- **Erste Lösungsidee**: rekursive Formulierung

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

Fibonacci Zahlen - erste Lösung - das Programm

```
#
# rekursive Berechnung der Fibonacci-Zahlen
# fib(n) = fib(n-1)+fib(n-2)
#
def fib(n)
  return 0 if n==0 # Terminierung
  return 1 if n==1 # Terminierung
  return fib(n-1) + fib(n-2) # Rekursionsschritt
end
```

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

fib(4) Verarbeitungstrace

```
=> fib(4)
   => fib(3)
      => fib(2)
         => fib(1)
         <= fib(1) r==1
         => fib(0)
         <= fib(0) r==0
      <= fib(2) r==1
      => fib(1)
      <= fib(1) r==1
   <= fib(3) r==2
   => fib(2)
      => fib(1)
      <= fib(1) r==1
      => fib(0)
      <= fib(0) r==0
   <= fib(2) r==1
<= fib(4) r==3
```

```ruby
def fib(n)
  return 0 if n==0
  return 1 if n==1
  return fib(n-1) + fib(n-2)
end
```

Wir sehen schon für n=4 (der Trace für n=5 passt nicht auf eine Folie) das Drama der Lösung:

fib(2), fib(1) und fib(0) werden mehrfach berechnet.

Versuchen Sie mit der ersten naiven Lösung einmal fib(100) zu berechnen ☺

Rekursion mit Speicher: Fibonacci Zahlen

- **Bewertung der ersten Lösung:**

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

- exponentieller Berechnungsaufwand $\approx 2^n$ (genauer $1{,}618034^n$)

- wiederholte Mehrfachberechnung: *fib(n-2)* wird in *fib(n-1)* und *fib(n)* berechnet. *fib(n-3)* in *fib(n-2)*, *fib(n-1), fib(n)* usw.

Rekursion mit Speicher: Fibonacci Zahlen

- **zweite Lösung**:
  - wir speichern die Vorgänger Zahlen bei jedem Aufruf und addieren nur noch die beiden Vorgänger zur neuen Zahl
  - gestartet wird die Berechnung mit 0 für fib_{n_2} und 1 für fib_{n_1}
  - wenn n==0/n==1 geben wir die akkumulierte Zahl in der Variablen fib_{n_2} / fib_{n_1} zurück

$$fib(n, fib_{n\_1}, fib_{n\_2}) = \begin{cases} fib(n-1, fib_{n\_1} + fib_{n\_2}, fib_{n\_1}); n > 1 \\ fib_{n\_1}; n = 1, \\ fib_{n\_2}; n = 0 \end{cases}$$

- Berechnungsaufwand: n

Endrekursion – Fibonacci mit Speicher

```ruby
#
# fib mit 2 Speichern fib_n_1 speichert fib(n-1),
# fib_n_2 speicher fib(n-2)
# In jedem rekursiven Aufruf wird aus der Summe fib_n_1 + fib_n_2 das
# neue fib_n_1
# und fib_n_2 wird das alte fib_n_1
#
# Terminierung ist erreicht, wenn n==0, oder n==1
#

def fib_acc(n,fib_n_1=1,fib_n_2=0)
  if n == 0
    return fib_n_2  # Terminierung
  end
  if n== 1
    return fib_n_1  # Terminierung
  end
  fib_acc(n-1,fib_n_1+fib_n_2,fib_n_1) # Rekursionsschritt
end
```

fib_acc - Verarbeitungstrace für n = 7

```ruby
def fib_acc(n,fib_n_1=1,fib_n_2=0)
  if n == 0
    return fib_n_2
  end
  if n== 1
    return fib_n_1
  end
  fib_acc(n-1,fib_n_1+fib_n_2,fib_n_1)
end
```

```
=> fib(7,1,0)
   => fib(6,1,1)
      => fib(5,2,1)
         => fib(4,3,2)
            => fib(3,5,3)
               => fib(2,8,5)
                  => fib(1,13,8)
                  <= fib(1,13,8) r==13
               <= fib(2,8,5) r==13
            <= fib(3,5,3) r==13
         <= fib(4,3,2) r==13
      <= fib(5,2,1) r==13
   <= fib(6,1,1) r==13
<= fib(7,1,0) r==13
```

Mit dieser Lösung, die nur n Berechnungsschritte benötigt, lässt sich jetzt fib_acc(500) effizient berechnen.

fib_acc(500)
=1394232245616978801397243828704072839500702565876973072641089629483255716228632906915576588762225212941 25

Fibonacci Zahlen: Iterative Berechnung

- Aus der zweiten rekursiven Lösung lässt sich die iterative Lösung ableiten
  - Wir merken uns in den Variablen fib_{n_2}, fib_{n_1} die Ergebnisse der vorausgehenden Iteration
  - gestartet wird die Berechnung mit 0 für fib_{n_2} und 1 für fib_{n_1}
  - wenn n==0/n==1 wird fib_{n_2} / fib_{n_1} zurückgegeben
  - wenn n > 1, berechnen wir das neue fib_{n_1} aus fib_{n_2} + fib_{n_1} und fib_{n_2} wird zu fib_{n_1}

$$fib(n, fib_{n\_1}, fib_{n\_2}) = \begin{cases} fib(n-1, fib_{n\_1} + fib_{n\_2}, fib_{n\_1}); n > 1 \\ fib_{n\_1}; n = 1, \\ fib_{n\_2}; n = 0 \end{cases}$$

- Berechnungsaufwand: n

Fibonacci iterativ

```ruby
# Fibonacci iterativ
def fib_iter(n)
  fib_n_1 =1
  fib_n_2 =0
  return fib_n_1 if n == 1
  return fib_n_2 if n == 0
  for i in (2..n)
    old_fib_n_1 = fib_n_1
    fib_n_1 = fib_n_2 + fib_n_1
    fib_n_2 = old_fib_n_1
  end
  return fib_n_1
end
```

Übungen

- **Ü-9-b-3**: Schreiben Sie bitte eine rekursive Lösung für die Fakultät! Formen Sie die Lösung in einer Endrekursion um und leiten Sie daraus die iterative Lösung ab.

- **Ü-9-b-4** : Gegeben ist die rekursive Definition:

$$f(n) = \begin{cases} f(n-1) + \frac{1}{(4*n-1)(4*n+1)}\ ; \textit{für}\ n > 1 \\ \frac{1}{15}\ ; \textit{für}\ n = 1 \end{cases}$$

Schreiben Sie bitte eine äquivalente rekursive Methode mit Speicher! Wandeln Sie anschließend die Rekursion in eine Iteration um!

Inhalt

- Einfache Rekursion: Endrekursion – Harmonische Zahlen, ggt
- Mehrfachrekursion: Towers of Hanoi, Drachenkurve
- Rekursive Grafiken: HTree, Sierpinski Dreieck
- Rekursion mit Speicher (Memoisation): Fibonacci Zahlen
- **Objektrekursion**

Objektrekursion

- Objekte enthalten selbst wieder Objekte des eigenen Typs

- **Typische Lösungen**: rekursiver Abstieg über die enthaltenen Strukturen gleichen Typs. (siehe z.B. GKA, Literatur über Compiler)

- **Abbruchbedingung**: Es sind keine Objekte des eigenen Typs mehr enthalten.

- **Beispiel**: geschachtelte Arrays etc., Bäume, Stücklisten

Objektrekursion: Erweitern der Klasse Arrays

- **Problemstellung**:

  - Gegeben ein beliebig geschachteltes Array
  - Zählen Sie für dieses Array alle in dem Array enthaltenen Arrays.

- **Lösungsskizze**:

  - Iterieren über das Array
  - Prüfen, ob ein Element von Typ Array ist
  - Inkrementieren des Zählers
  - Rekursiver Aufruf für das enthaltene Array und Addieren des Ergebnisses

Zählen der enthaltenen Arrays

```
def count_arys(ary)
  count = 0
  ary.each() do |elem|
    if elem.is_a?(Array)
      count += count_arys(elem) +1
    end
 end
  return count
end
```

Iterieren über das Array
Prüfen, ob ein Element von Typ Array ist
Inkrementieren des Zählers
Rekursiver Aufruf für das enthaltene Array und Addieren des Ergebnisses

Objektrekursive Methoden der Klasse Array

- **Problemstellung**:
  - Gegeben ein beliebig geschachteltes Array. Schreiben Sie die Methode ***flatten*** der Klasse Array, die das Array in ein flaches Array überführt. Die Verwendung der Methoden ***flatten*** und ***flatten!*** ist nicht erlaubt.

- **Lösungsskizze**:
  - Iterieren über das Array.
  - Prüfen, ob ein Element von Typ Array ist.
  - Wenn ja, das Element „flatten“ und dem Ergebnis anhängen.
  - Wenn nein, das Element in das Ergebnis übertragen.

Objektrekursive Methoden der Klasse Array

```
class Array
  def flatten()
    flat_ary = Array.new()
    each() do |elem|
      if elem.is_a?(Array)
        flat_ary = flat_ary + elem.flatten()
      else
         flat_ary << elem
      end
     end
   return flat_ary
  end
end
```

Iterieren über das Array.
Prüfen, ob ein Element von Typ Array ist.
Wenn ja, das Element „flatten“ und dem Ergebnis anhängen.
Wenn nein das Element in das Ergebnis übertragen.

Übungen

- **Ü-9-b-6**: Gegeben ein beliebig geschachteltes Array. Zählen Sie bitte für dieses Array die Anzahl aller enthaltenen geraden und ungeraden Zahlen! Die Methode gibt ein 2-elementiges Array zurück. Auf Position 0 steht die Anzahl der geraden, auf Position 1 die Anzahl der ungeraden Zahlen. Das Benutzen von ***flatten*** ist verboten. Lösen Sie dies rekursiv!

- **Ü-9-b-7**: Schreiben Sie bitte eine rekursive nicht destruktive Methode ***deep_reverse*** für die Klasse Array ohne / mit Speicher! Sie dürfen die Methode ***reverse*** von Array verwenden.