Modul: B-PRG1:
Grundlagen der Programmierung 1 und Einführung in die Programmierung EPR

V20 Algorithmenkonstruktion (Algorithmen(entwurfs-)muster) &
Rekursion vs. Iteration

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Vorab

- Lehrveranstaltungsevaluation in der **nächsten Woche!**
- **Online in Präsenz**
- Bitte Smartphone oder Laptop mitbringen!

Rückblick

V01 Algorithmen: Begriff und Anwendung

V03 Kontrollstrukturen 1 (Schleifen) ➔ Iteration
V06 Kontrollstrukturen 2 (Unterprogramme) ➔ Rekursion

Schon sehr viel im Bereich "Software-Engineering" gemacht.

Aber: Wie finde ich "gute" Algorithmen?

Unser heutiges Lernziele

Reflektieren:

- *Welche Entwurfsmuster gibt es, um "gute" Algorithmen zu finden?*
- *Was bedeutet Rekursion in der Praxis?*

Übersicht

- Algorithmenkonstruktion = Entwurfsmuster für Algorithmen
  - Greedy-Methode
  - Divide & Conquer
  - Backtracking
- Rekursive Grundstrukturen
- Implementierungen in Python
- Zusammenfassung

Bedeutung effizienter Algorithmen und Programmen

"Software inefficiency can always outpace Moore's Law. Moore's Law isn't a match for our bad coding." – Jaron Lanier

Beispiel: Laufzeit für 2 Sortieralgorithmen bei sehr großen Eingaben (Sortierung von Listen mit >> 1.000.000 Elementen)

	Operationen pro Sekunde	Schlechter Algorithmus	Guter Algorithmus
Laptop	10^7	3 Jahrhunderte	3 Stunden
Supercomputer	10^{12}	2 Wochen	Interaktiv < 1 sec

➔ Qualität des Algorithmus entscheidender als eingesetzte Hardware oder "Geschwindigkeit der Programmiersprache"

Entwurf von Algorithmen

- Programmieren ist sowohl eine konstruktive wie auch kreative Tätigkeit.
- Erstellung eines optimalen Algorithmus ist i.a. nicht automatisierbar.
- Trotzdem existieren verschiedene **typische Muster** für Algorithmen, die in Praxis oft Anwendung finden (**Entwurfsmuster)**
  - Sind aber fallspezifisch anzupassen.

z.B. Algorithmen(entwurfs)muster für Optimierungsprobleme, u.a.

- **Greedy-Algorithmen**
  - effizient, liefern lokales Optimum
- **Divide and Conquer (*divide et impera*)**
  - Liefern oft bestes Zeitverhalten
- **Backtracking**
  - nicht so effizient, liefert aber globales Optimum

Entwurfsprinzip: Schrittweise Verfeinerung
➔ Top-Down-Entwurf

Beim Entwurf von Algorithmen beschreibe Grobverfahren in abstraktem Pseudocode dabei Zerlegung in Teilaufgaben

Verfeinere Pseudocode und ersetze diesen nach und nach durch detaillierten Pseudocode

wiederhole

bis letztlich Algorithmus in Programmiersprachencode geschrieben ist.

Top-Down-Entwurf

Entwurfsprinzip: Einsatz von Algorithmen(entwurfs-)mustern

Das (generische) Lösungsverfahren wird an einem möglichst einfachen Vertreter der Problemklasse vorgeführt und dokumentiert.

Der Entwickler versteht die Problemlösungsstrategie und überträgt diese auf sein Programm

➔ Konzept moderner Softwareentwicklung: **"Design Patterns"**
Eine Bibliothek von Mustern ("Design Patterns", "best-practice"-Strategien) wird genutzt, um einen abstrakten Programmrahmen zu generieren.

Abstraktion vom Begriff "Algorithmus".

Die "freien Stellen" dieses Programmrahmens werden dann problemspezifisch ausgefüllt. In modernen Programmiersprachen wird dies zum Teil schon unterstützt durch z.B. parametrisierte Algorithmen und – in objektorientierter Programmierung – Vererbung mit Überschreiben

Algorithmenmuster Greedy

1. Beispiel:
Wechselgeld (bis 1 Euro) herausgeben, mit minimaler Anzahl von Münzen à 50, 20, 10, 5, 2, und 1-Cent Münzen.

- Prinzip gieriger Algorithmen:
- Versuche in jedem Teilschritt so viel wie möglich zu erreichen, dem Ziel so nahe wie möglich zu kommen.

Greedy-Strategie: Nimm jeweils **die größte Münze** unter Zielwert, und ziehe diese von Zielwert ab, solang, bis Zielwert = 0

- Bsp: 78 Cent ausgeben -> 50 + 20 + 5 + 2 +1, d.h. 5 Münzen
- Greedy berechnet bei dieser Aufgabe immer die optimale Lösung …

Was wäre wenn?

- Alles gut? – Immer optimale Lösung gefunden?
- … wäre es anders, wenn nur Münzen à 11, 5, und 1 Cent zu Verfügung stünden

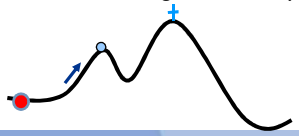
Ja, z.B. 15 Cent sind zu wechseln
nach Greedy: 15 = 11 + 4 • 1, d.h. 5 Münzen
optimal sind aber: 15 = 3 • 5, d.h. 3 Münzen

Greedy – Methode (engl. gierig, gefrässig)

- **Voraussetzung:** Explizite Bewertung des eigenen Zustands nötig.
- **Ziel**: Suche ein Optimum in mehreren Schritten.
- **Idee:** Mache jeden Schritt (**lokal, aktuell**) optimal.
- **Hoffnung**: Dann wird das Ergebnis optimal.

Beispiel *Bergbesteigung*
Mache jeden Schritt so, dass man ein Stück höher kommt.
Dann erreichst Du irgendwann die Bergspitze.

Problem der Hügel vorher (*lokales Optimum*)



2. Beispiel: Sortieren

Beschreibung des Sortierproblems

- **Gegeben**: Liste **v** mit *N* Elementen aus der Menge der natürlichen Zahlen
- **Gesucht**: Anordnung **v*** der Elemente von **v** in aufsteigender Reihenfolge, also

  $$\mathbf{v}[0] \leq \mathbf{v}[1] \leq \ldots \leq v[len(v)-1]$$

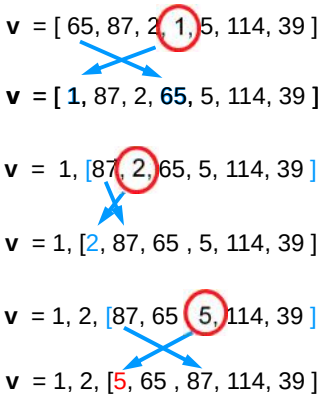
- **Beispiel**:

  Gegeben: **v** = [65, 87, 2, 1, 5, 114, 39]
  Ziel: **v*** = [1, 2, 5, 39, 65, 87, 114]

Selection_Sort (Ein einfaches Greedy-Verfahren zum Sortieren)

- Finde zuerst das kleinste Element in der Folge **v** und vertausche es mit dem Element in der ersten Position. Dieses Element steht dann schon einmal in der richtigen Position (das gierige daran).
- Finde das zweit-kleinste Element, indem unter den restlichen Elementen (2. bis n-te Position) wieder das kleinste gesucht wird, und vertausche es mit dem Element in der zweiten Position.
- Fahre in dieser Weise fort, bis die gesamte Folge sortiert ist.

Selection_sort



1.Min suchen, vertauschen.

2.Min suchen, vertauschen.

3.Min suchen, vertauschen.

Programm: selection_sort

```
def selection_sort(v):
    for i in range(len(source)):
    mini = min(source[i:]) #find minimum element
    min_index = source[i:].index(mini) #find index of minimum element
    source[i + min_index] = source[i]  #replace element at min_index
                                       #with first element
    source[i] = mini #replace first element with min element
    return v
```

Berechnungsaufwand Selection_Sort

- Um das kleinste Element zu finden, müssen maximal *N* Elemente betrachtet werden.
- Für das zweite eins weniger, also *N-1*, für das dritte *N-2* usw.
- Insgesamt werden maximal

  $N + (N-1) + (N-2) + \ldots + 2 + 1 = \sum_{i=1}^{N} i = (\quad + 1) \quad - \quad = \frac{N^2 + N}{2}$

  Elemente betrachtet.
- Für große *N* ist N^2 der dominierende Term;

  Wir sagen:

  „Die Laufzeit ist in der Größenordnung von N^2“ = $O(N^2)$. Zu dieser sogenannten (Groß-)O-Notation werden Sie in der Theorie noch viel hören.

Sortieren

- Sortieren ist ein klassisches häufig zu lösendes Problem in der Informatik;
- Deshalb wird jeder vorgestellte Sortieralgorithmus bezüglich seines Berechnungsaufwandes (**Laufzeit**) abgeschätzt.
- In der Praxis gibt es weitere Punkte, die für die Wahl des optimalen Sortieralgorithmus wichtig sind:
  - **Welche** Daten sollen sortiert werden (Integer, Strings, …) ?
  - Wie **groß** ist der Datensatz?
  - Nach welcher **Vergleichsfunktion** soll sortiert werden?
  - In wieweit sind die Daten **vorsortiert**?
  - Wieviel **Speicher** steht zur Verfügung?
  - Sind **doppelte Einträge** vorhanden?

Zusammenfassung
Greedy (gefrässige, gierige) Algorithmen

- Greedy-Algorithmen besitzen einen hohen Stellenwert in der heutigen Informatik, da sieggf. schnelle, einfache Lösungen bieten:
- die Entscheidungen bleiben bestehen müssen nicht aufbewahrt werden. Einfach **iterativ** (➔ schnell) zu lösen.
- Trotz ihrer Eigenschaft, **nicht immer die optimale Lösung** zu finden, lassen sich eine große Menge an Problemen mit diesem Verfahren lösen.

➔ nach einer Möglichkeit suchen, die es leicht macht, Probleme, die durch ein Greedy-Verfahren optimal gelöst werden können, schnell und zuverlässig zu erkennen

➔ die sogenannten Matroiden: ***"Greedy-Algorithmen sind genau dann optimal, wenn das Problem als Matroid dargestellt werden kann."***
[➔ Theoretische Informatik]

Divide and Conquer – Methode

- **Teile (***Divide, divide***)**
  - Teile das Problem der Größe *N* in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn N > 1 ist.
  - Löse die Teilprobleme auf die selbe Art.
- **Herrsche (***Conquer, impera***)**
  - Ist ein Teilproblem hinreichend klein (z.B. N = 1), so löse es direkt und breche so die Rekursion im Lösungs-schema ab.
- **Vereinige (***Merge***)**
  - Füge die Lösungen für die Teilprobleme zur Gesamtlösung zusammen.

Sortier-Algorithmus 3: Merge-Sort

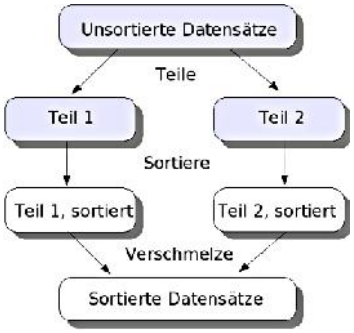
- Beispiel für Teile-und-Herrsche (Divide and Conquer) Methode

Idee: Sortieren ist einfacher/schneller für kurze als für lange Listen

- insbesondere sind Listen mit nur einem Element schon sortiert

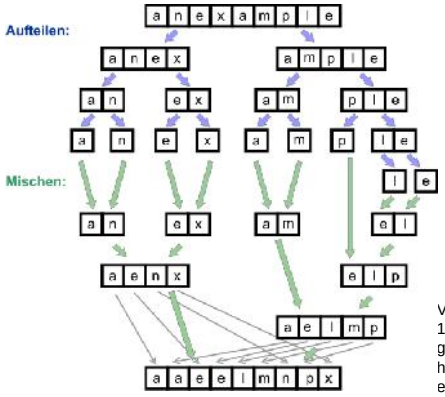
Konkret: Teile Liste in zwei kleinere Teil-Listen

- sortiere die Teil-Listen (evtl. durch weiteres Teilen)
- Verschmelze die sortierten Teil-Listen zum Endergebnis



- Von Matthias Kleine, 23.11.2004 - Erstellt mit OpenOffice und gimp., CC BY-SA 3.0, https://de.wikipedia.org/w/index.php?curid=448759

Funktionsweise Mergesort



Von Jkrieger 9. Jul 2005 16:01 (CEST) - selbst gezeichnet, Bild-frei, https://de.wikipedia.org/w/index.php?curid=793375

Pseudocode:

- **Recursively sort the first half of the input array.**
- **Recursively sort the second half of the input.**
- **Merge two sorted sub-lists into one list.**

Python Code for mergesort (Teil1)

```
def merge(a,b):
    """ Function to merge two lists. """
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c += b
    else:
        c += a
    return c
```

Python Code for mergesort (Teil2)

```
def merge_sort(x):
    """ Function to sort an array using merge sort algorithm """

    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = len(x)/2
        a = merge_sort(x[:middle])
        b = merge_sort(x[middle:])
        return merge(a,b)
```

Rekursion

Zeitkomplexität von mergesort

Es kann gezeigt werden:

- Die Ausführungszeit von merge_sort ist proportional zu n * log n = O(n * log n), mit n ist Größe der Eingabeliste.

Es kann auch gezeigt werden:

- minimale Ausführungszeit für beliebige Algorithmen zur Lösung des Problems "Sortieren" von unsortierten Listen der Länge n"
  ist proportional zu n log n

➔ **mergesort ist ein optimaler Sortieralgorithmus (bzgl. Zeitkomplexität)**

➔ **Devide and Conquer führt "natürlich" zu rekursiven Lösungen.**

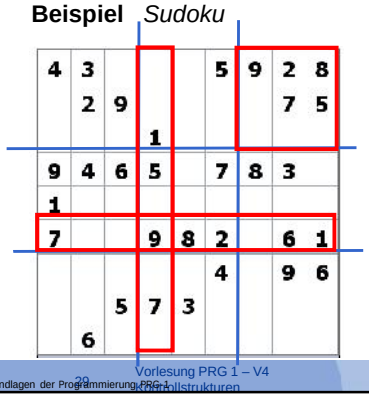
Zusammenfassung

- Bei großen Listen, z.B. ab 100 Elementen, spielt Zeitkomplexität der Algorithmen eine entscheidende Rolle.
- n log n Algorithmen (z.B. merge-sort) sind "theoretisch" immer besser als n^2-Algorithmen (selectionsort)
- Aber **Vorsicht**: auch bei gleicher Zeitkomplexität zweier Algorithmen bestehen manchmal deutliche Unterschiede bzgl. tatsächlicher Laufzeit.
- Für die meisten praktischen Fälle liefern die Sortierverfahren der Standardbibliotheken der jeweiligen Programmiersprachen sehr gute Ergebnisse.
- Diese Implementierungen sind i.d.R. hochgradig optimiert, z.B. Python: sort()-Methode.

Backtracking –Methode

- **Gegeben**: Suche ein Optimum in mehreren Schritten
- **Idee:** Suche **systematisch alle** Möglichkeiten ab
- **Gewissheit**: Dann wird das Ergebnis optimal.

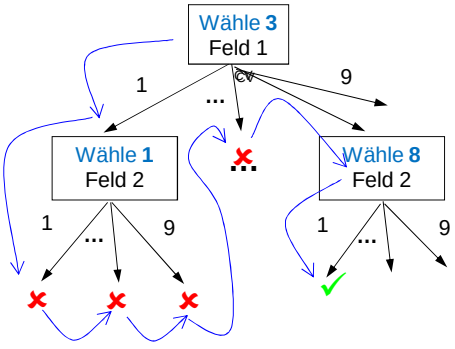
Beispiel *Sudoku*



Regeln

1. Jedes quadratische **Unterfeld** der Größe 3x3 enthält alle Zahlen von 1 bis 9. Hier dürfen also keine zwei Zahlen gleich sein.
2. In jeder **Spalte** dürfen keine zwei gleichen Zahlen vorkommen.
3. In jeder **Zeile** dürfen keine zwei gleichen Zahlen vorkommen.

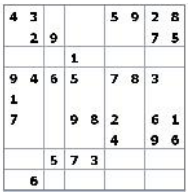
Beispiel Sudoku



Eine mögliche Belegung aller leeren Felder zusammen bildet eine Lösung.

🡐 **Lösung** *für 2 Felder*

🡒 Systematisches Absuchen des Entscheidungsbaumes



In order backtracking: Beispiel-Code

```
def track (Baum):          # Lösungssuche mit back-tracking

if Baum.Links != None :    # Existiert ein linker Unterbaum?
  if LsgOK(Baum.Links):    # und die Lösung ist ausreichend?
     track(Baum.Links)     # dann: in-order linker Unterbaum

if Test(Baum.Wurzel) :     # Lösung gefunden?
    print(„Lösung gefunden!“, Baum.Wurzel)

if Baum.Rechts != None :   # Existiert ein rechter Unterbaum?
  if LsgOK(Baum.Rechts):   # und die Lösung ist ausreichend?
     track(Baum.Rechts)    # dann: in-order rechter Unterbaum
```


Zusammenfassung Backtracking

Systematische Absuche **aller** Möglichkeiten („Suchraum“)

- **Findet "theoretisch" die optimale Lösung!**
- **Problem**: Bei *n* Elementen mit jeweils *m* möglichen Werten haben wir m^n Möglichkeiten – „kombinatorische Explosion“, sehr großer Suchraum ➔ lange Laufzeit
- Bei unserem Sudoku: m = 9, n = (81-32) Felder = 49 … also 9^{49}
- **Nur** anzuwenden, **wenn** sonst keine Informationen existieren, die Zahl der möglichen Lösungen einzuschränken („Beschränkung auf eine Teilmenge des Suchraums“)
- **Backtracking führt "natürlich" zu rekursiven Lösungen.**

Zusammenfassung: Algorithmenmuster (1)

- **Greedy**
  - berechnet **lokales** Optimum
  - polynomiale Zeitkomplexität z.B. n^2
- **Backtracking**
- berechnet **globales** Optimum
- Zeitkomplexität entsprechend Größe des (problemspezifischen) Konfigurationsraums, z.B. exponentiell (proportional 2^n) oder auch – noch schlimmer – proportional zu n!
- vollständiges Durchsuchen des Konfigurationsraums per Backtracking daher nur für kleinere Probleminstanzen praktisch durchführbar

Divide-and-Conquer (D&C)

- Wenn D&C Algorithmen möglich sind, dann ist die Problemstellung oft schon so, dass bereits der brute-force Algorithmus eine polynomielle Laufzeit hat. Durch D & C wird diese Laufzeit dann weiter reduziert, z.B. O(n*logn).

Weitere Algorithmenmuster

- dynamisches Programmieren (➔ Theorie)

Rekursive Grundstrukturen

Rekursion, auch *Rekurrenz* oder *Rekursivität*, bedeutet Selbstbezüglichkeit (von lateinisch *recurrere* = zurücklaufen; engl. *recursion*).

tritt immer dann auf, wenn etwas auf sich selbst verweist.

muss nicht immer **direkt** auf sich selbst verweisen (**direkte Rekursion**), eine Rekursion kann auch über mehrere Zwischenschritte entstehen.

Rekursion kann dazu führen, dass merkwürdige Schleifen entstehen.

So ist z.B. der Satz „Dieser Satz ist unwahr“ **rekursiv**, da er von sich selbst spricht.

Beispiele

Eine etwas subtilere Form der Rekursion (**indirekte Rekursion**) kann auftreten, wenn zwei Dinge gegenseitig aufeinander verweisen.

„Der folgende Satz ist wahr.“ - „Der vorhergehende Satz ist nicht wahr.“.

„Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen.“

„Kürzeste Definition für Rekursion: siehe Rekursion.“

Definition (Informatik)

Als **Rekursion** bezeichnet man in der Informatik den **Aufruf** eines Unterprogramms **durch sich selbst**.

Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen so genannten infiniten Regress. (wie eine Endlosschleife)

Grundidee der rekursiven Definition

Der Funktionswert *f*(n+1) einer Funktion *f*: $\mathbf{N_0} \rightarrow \mathbf{N_0}$ ergibt sich durch Verknüpfung bereits vorher berechneter Werte *f*(n), *f*(n-1), ...

Falls außerdem die Funktionswerte von *f* für hinreichend viele Startargumente bekannt sind, kann jeder Funktionswert von *f* berechnet werden.

Bei einer rekursiven Definition einer Funktion *f* ruft sich die Funktion so oft selbst auf, bis ein vorgegebenes Argument erreicht ist, so dass die Funktion terminiert (abbricht).

Vorgehensweise in der funktionalen Programmierung (siehe PRG 2)

Beispiel

Die Funktion *sum (n)* berechnet die Summe der ersten *n* Zahlen.

also: sum (n): $\mathbf{N_0} \rightarrow \mathbf{N_0}$: *sum*(n) = 0 + 1 + 2 +...+ n

Anders ausgedrückt: *sum*(n) = *sum*(n-1) + n (**Rekursionsschritt**)

Das heißt also, die Summe der ersten *n* Zahlen lässt sich berechnen, indem man die Summe der ersten *n* - 1 Zahlen berechnet und dazu die Zahl *n* addiert.

Damit die Funktion terminiert, legt man hier für *sum*(0) = 0 (**Rekursionsanfang, base case**) fest.

Beispiel (2)

$$sum(n) = \begin{cases} 0 & \text{falls } n = 0 \text{ (Rekursionsanfang)} \\ sum(n-1) + n & \text{falls } n \geq 1 \text{ (Rekursionsschritt)} \end{cases}$$

$$\begin{aligned} sum(3) &= sum(2) + 3 \\ &= sum(1) + 2 + 3 \\ &= sum(0) + 1 + 2 + 3 \\ &= 0 + 1 + 2 + 3 \\ &= 6 \end{aligned}$$

Beispiel: Fakultät einer Zahl

Für alle natürlichen Zahlen

$$n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

also:

$$3! = 1 \cdot 2 \cdot 3 = 6$$

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

häufig definiert man zusätzlich **0! = 1** (hier liegt das leere Produkt vor)

Realisierung (rekursiv)

```
def factorial_recursive(n):
    if n <= 1: return 1
    else: return (n * factorial_recursive(n-1)) #Rekursion
```

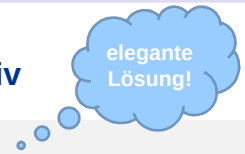
```
== RESTART: C:/Users/kroemker/Desktop/PRG1-EPR-2015/ \
factorial_rekursive.py ==
>>> factorial_recursive(3)
6
>>> factorial_recursive(5)
120
```


Realisierung (iterativ)

```
def factorial_iterative(n):
    factorial = 1
    factor = 2
    while factor <= n:
        factorial *= factor
        factor += 1
    return factorial
```

```
== RESTART: C:/Users/kroemker/Desktop/PRG1-EPR-2015/ \
factorilal_iterative.py ==
>>> factorial_iterative (3)
6
>>> factorial_iterative(5)
120
```

Gegenüberstellung rekursiv vs. iterativ




```
def factorial_recursive(n):
    if n <= 1: return 1
    else: return (n * factorial_recursive(n-1)) #Rekursion
```

```
def factorial_iterative(n):
    factorial = 1
    factor = 2
    while factor <= n:
        factorial *= factor
        factor += 1
    return factorial
```


Ein Versuch!

```
>>> def rekuriere():
        rekuriere()

>>> rekuriere()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    rekuriere()
  File "<pyshell#13>", line 2, in rekuriere
    rekuriere()
  File "<pyshell#13>", line 2, in rekuriere
    rekuriere()
  File "<pyshell#13>", line 2, in rekuriere
    rekuriere()
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
>>>
```

Ein Versuch: Teil 2

- `"maximum recursion depth"`
  ist eine (System-)Variable im modul `sys`
- kann man auslesen und setzen:

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```


Ein Versuch: Teil 3

```
>>> import sys
>>> sys.setrecursionlimit(10000)
>>> def rekuriere():
        rekuriere()


>>> rekuriere()

============================== RESTART: Shell ========
```

Eine Rekursion ist oft eine elegante Lösung, aber

- Ist teuer bezüglich
  - Laufzeit: Wechsel des Namensraumes … ➔ Stack
  - Speicher: Stack benötigt ggf. viele Platz
- Iteration ist fast immer die effizientere Implementierung.

Primitive Rekursion

Ist stets durch eine Iteration ersetzbar.

Kennzeichen einer primitiven Rekursion:
der Aufruf-Baum enthält keine Verzweigungen, das heißt er ist eigentlich eine Aufruf-Kette

das ist immer dann der Fall, wenn eine rekursive Funktion sich selbst jeweils nur **einmal** aufruft,

insbesondere am Anfang (***Head Recursion***) oder nur am Ende (***Tail Recursion***) der Funktion.

Sogar mehr:

Mittels eines Stacks ist jede Rekursion in eine Iteration wandelbar!

(so macht es ja der Compiler/Interpreter)

Üben Sie das wandeln **Rekursion ← → Iteration**

Zusammenfassung Rekursion

- **Rekursion** bedeutet Selbstbezüglichkeit
- Sie tritt immer dann auf, wenn etwas auf sich selbst verweist.
- muss nicht immer **direkt** auf sich selbst verweisen (**direkte Rekursion**).
- Als Algorithmus ist **die Rekursion oft der elegantere Weg, z.B:** kürzer oder folgt direkt aus dem *Entwurfsmuster.*
- **Implementiert wird die Rekursion in Programmen dadurch,** dass sich Unterprogramme (Routinen, Subroutinen, Funktionen) selbst wieder aufrufen. Dies ist **relativ aufwändig!**

Fragen

Fragen ?

und (hoffentlich) Antworten

Ausblick ... nächsten Freitag

Daten – Information – Wissen

... und, danke für Ihre Aufmerksamkeit!