



python

Sortieren
und Rekursion

Niels Lohmann
Stefanie Behrens

Lutz Hellmig
Karsten Wolf

Universität
Rostock



Traditio et Innovatio



python

Sortieren
und Rekursion

Organisatorisches

1

Niels Lohmann
Stefanie Behrens

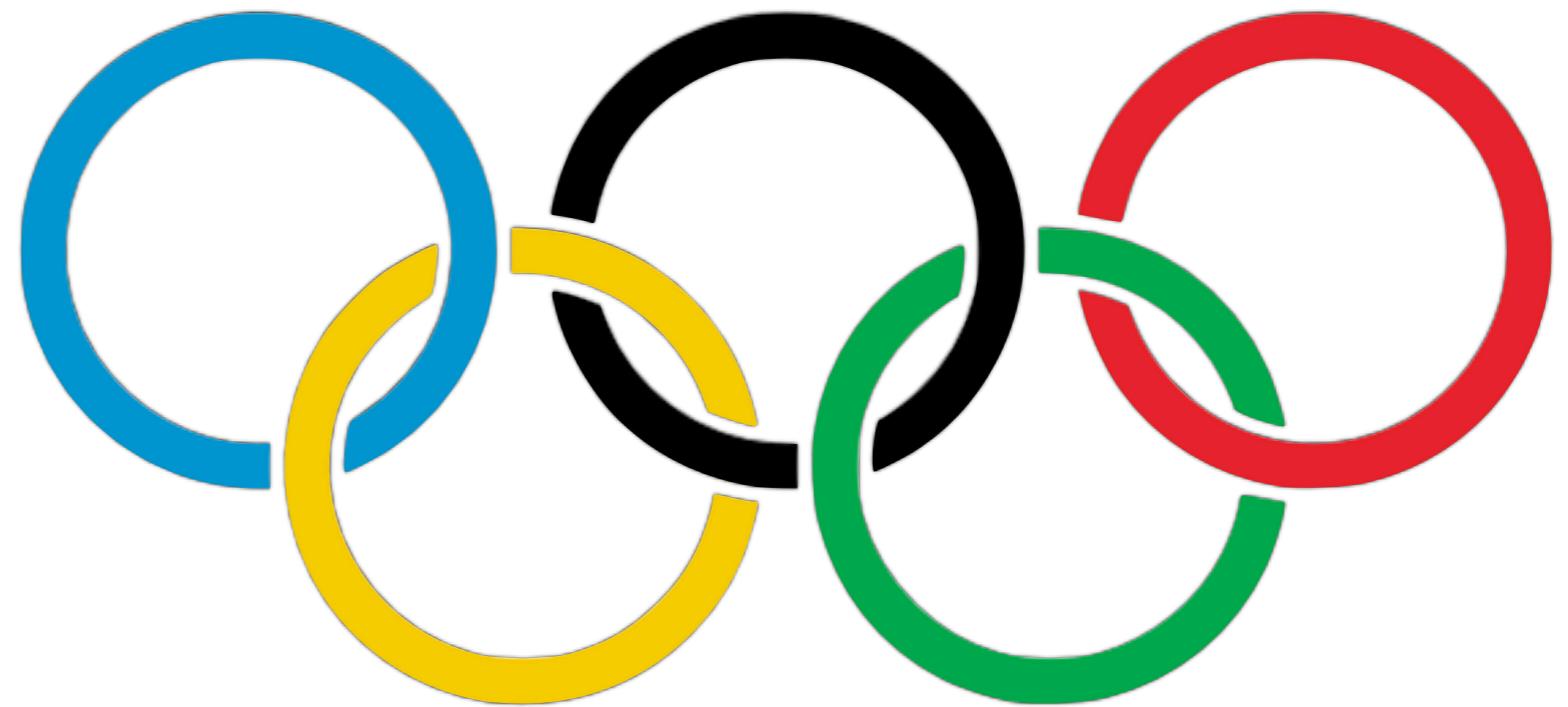
Lutz Hellmig
Karsten Wolf

Universität
Rostock



Traditio et Innovatio

ORGANISATORISCHES



TAGESABLAUF

Heute: Sortieralgorithmen und Rekursion

10:00-12:30 **Sortieralgorithmen**

13:00-15:15 **Rekursion**

15:30-18:00 **Dynamische Programmierung**

PROGRAMME ROBUST MACHEN

nicht alle Fehler können mit **if** umschiftt werden: falsche Eingaben

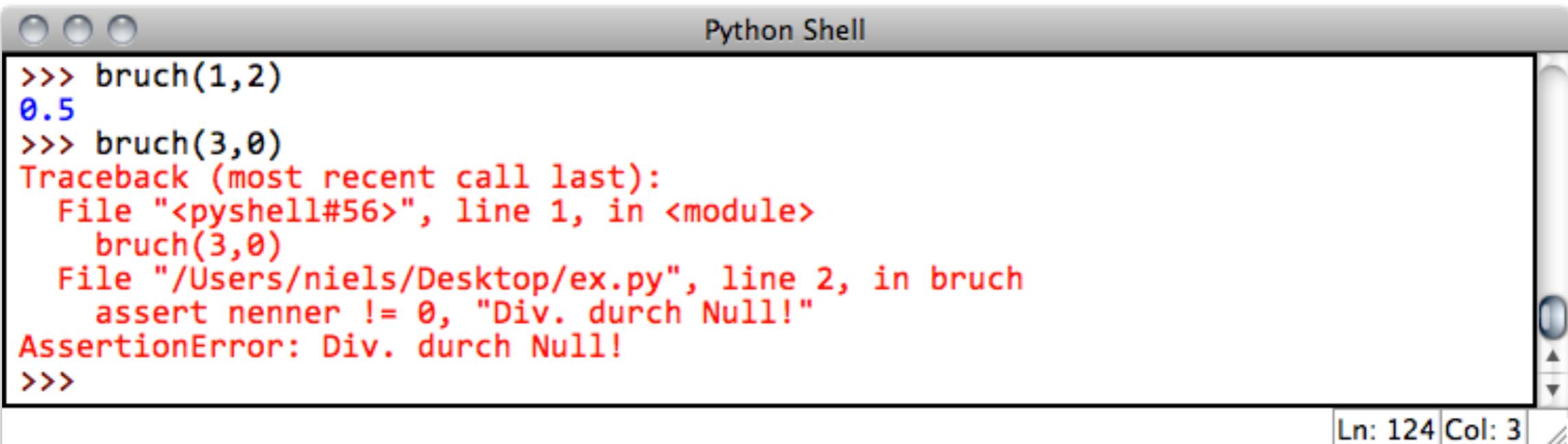
Lösung: Codeausführung nur "ausprobieren"
danach Fallunterscheidung

```
try:  
    y = int(input("Bitte Zahl eingeben! "))  
except:  
    print("Das war keine ganze Zahl!")  
else:  
    print("Das Doppelte ist", 2*y)
```

PROGRAMME ROBUST MACHEN

getroffene Annahmen verbindlich machen

```
def bruch(zahler, nenner):
    assert nenner != 0, "Div. durch Null!"
    return zahler / nenner
```



The screenshot shows a Python Shell window with the following interaction:

```
>>> bruch(1,2)
0.5
>>> bruch(3,0)
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    bruch(3,0)
  File "/Users/niels/Desktop/ex.py", line 2, in bruch
    assert nenner != 0, "Div. durch Null!"
AssertionError: Div. durch Null!
>>>
```

The window title is "Python Shell". The code defines a function `bruch` that takes two arguments: `zahler` and `nenner`. It includes an `assert` statement to check that `nenner` is not zero, printing a message "Div. durch Null!" if it is. The first call to `bruch(1,2)` returns `0.5`. The second call to `bruch(3,0)` triggers the assertion error, showing the traceback and the error message.

PROGRAMME ROBUST MACHEN

weitere Strategien:

```
print("vor myFunction")
myFunction()
print("nach myFunction")
```



```
myFunctionCalls = 0
```

```
def myFunction(x,y):
    global myFunctionCalls
    myFunctionCalls = myFunctionCalls+1
    ...
    ...
```

DATEIEIN- UND AUSGABE

Zeilen einer Datei in eine Liste **laden**:

```
name = "meineDatei.txt"
zeilen = open(name).read().splitlines()
```

Zeilen in eine Datei **speichern**:

```
name = "neueDatei.txt"
datei = open(name, "w")
datei.write("Eine Zeile\n")
datei.write("Noch eine Zeile\n")
datei.close()
```



python

Sortieren
und Rekursion

Sortieralgorithmen

2

Niels Lohmann
Stefanie Behrens

Lutz Hellmig
Karsten Wolf

Universität
Rostock



Traditio et Innovatio

SORTIEREN



Grundlage für viele andere Probleme:

Effizientes Suchen Duplikaterkennung Datenbanken

SORTIERALGORITHMEN

Eingabe: Liste

Ausgabe: sortierte Liste

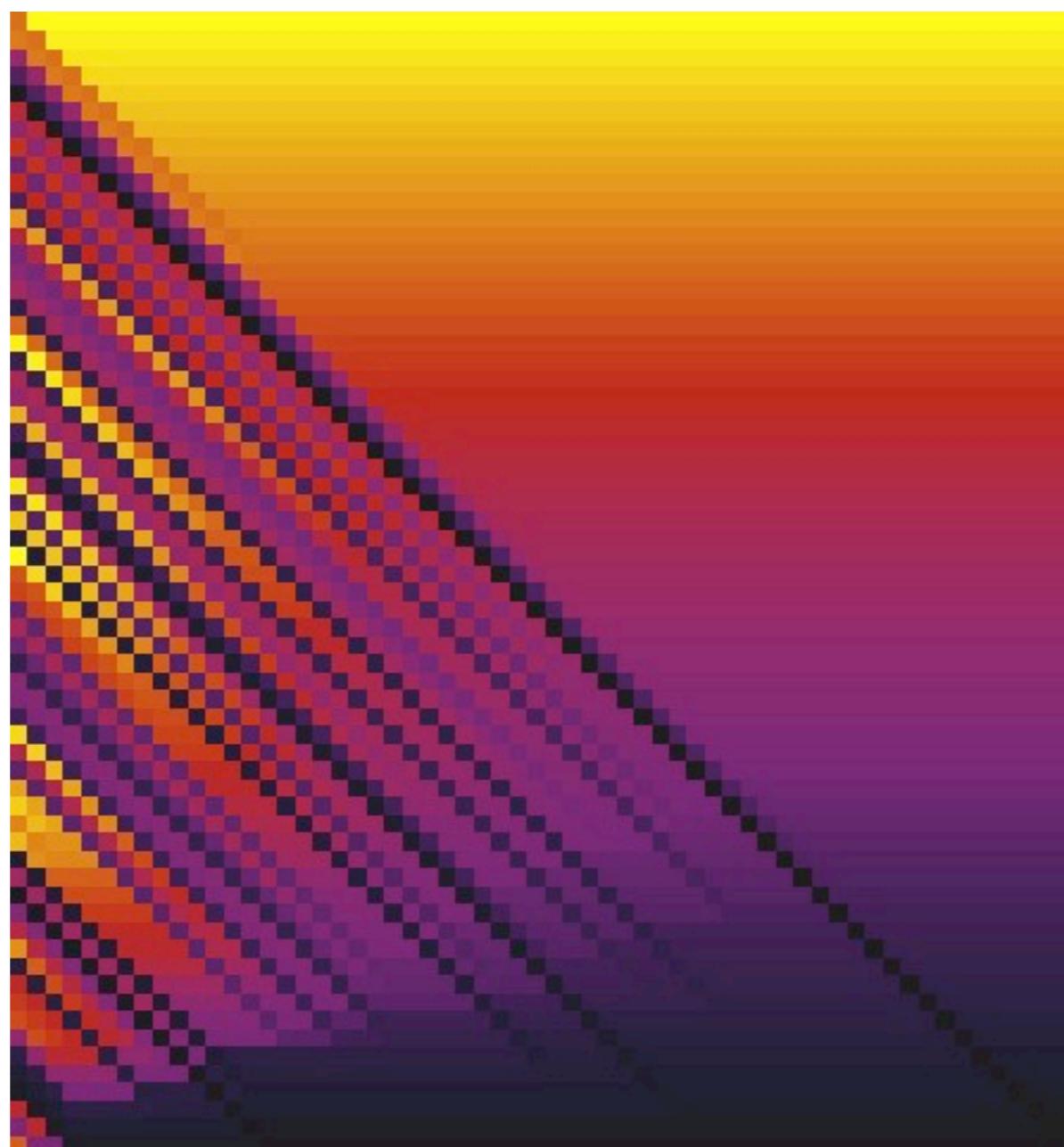
Parameter:

- Wie oft wird verglichen?
- Wie oft wird kopiert?
- Ist eine zweite Liste notwendig?



BUBBLE-SORT: SORTIEREN DURCH AUSTAUSCHEN

Idee: Sind benachbarte Elemente in der falschen Reihenfolge, werden sie vertauscht.



BUBBLE-SORT

[4, 2, 1, 6, 3, 5]

[2, 4, 1, 6, 3, 5]

[2, 1, 4, 6, 3, 5]

[2, 1, 4, 3, 6, 5]

[1, 2, 4, 3, 5, 6]

[1, 2, 3, 4, 5, 6]

BUBBLE-SORT

Idee: Sind benachbarte Elemente in der falschen Reihenfolge, werden sie vertauscht.

```
def bubblesort(liste):
    fertig = False

    while not fertig:
        fertig = True
        for i in range(0, len(liste) - 1):
            if liste[i] > liste[i+1]:
                temp = liste[i]
                liste[i] = liste[i+1]
                liste[i+1] = temp
                fertig = False

    return liste
```

BUBBLE-SORT

Wie gut ist Bubble-Sort?

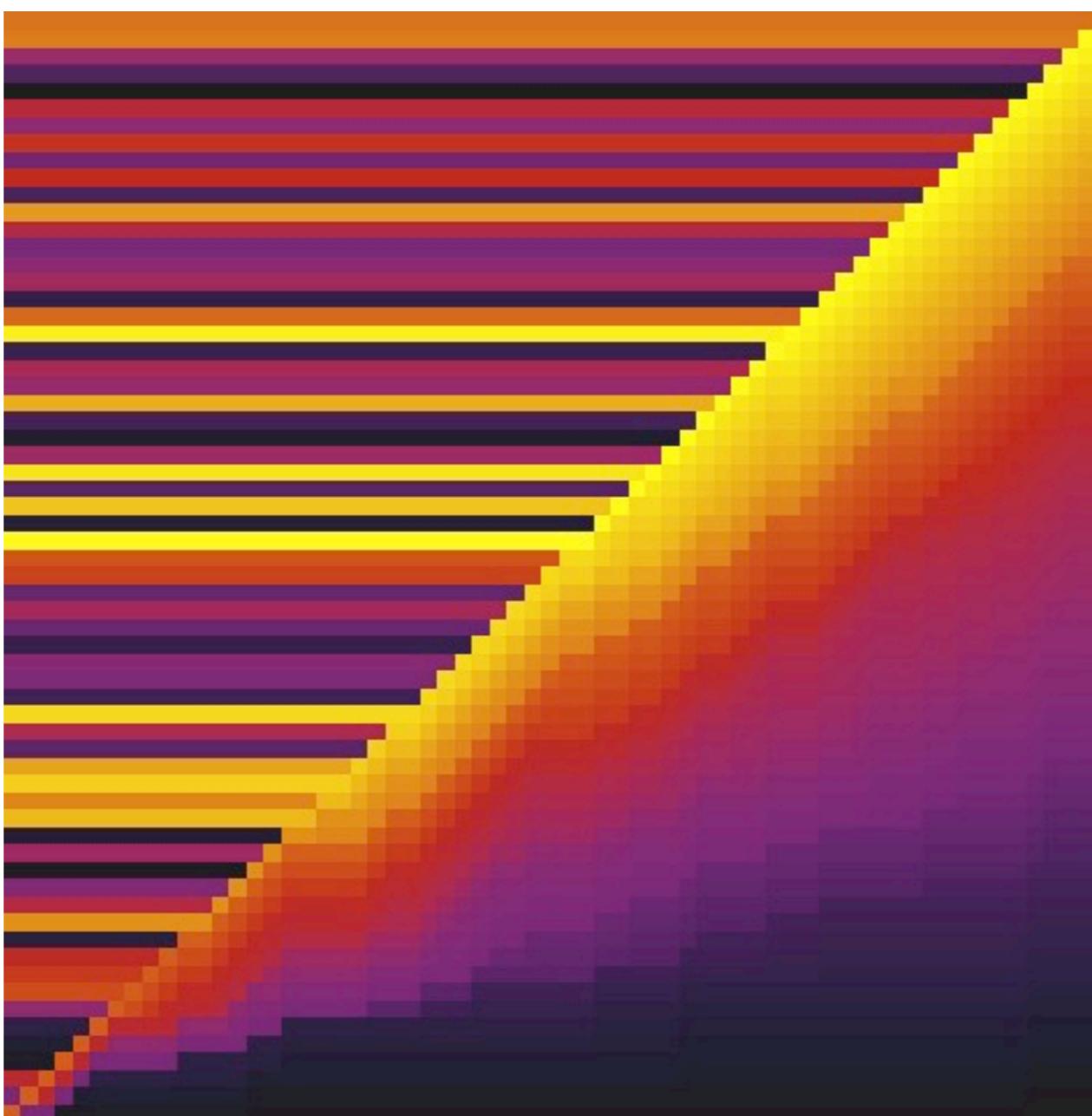
Anzahl Vergleiche?

Anzahl Kopiervorgänge?

```
>>> a = zufallsliste(10000)
>>> b = bubble_annotated(a)
97980201 Vergleiche
74300031 Kopieroperationen
56.7522950172 Sekunden
```

INSERTION-SORT: SORTIEREN DURCH EINFÜGEN

Idee: Nimm Element und sortiere es an der richtigen Stelle ein.



INSERTION-SORT

[4, 2, 1, 6, 3, 5]

[**4**, 2, 1, 6, 3, 5]

[**2**, 4, 1, 6, 3, 5]

[**1**, 2, **4**, 6, 3, 5]

[1, 2, **4**, **6**, 3, 5]

[1, 2, **3**, **4**, 6, 5]

[1, 2, **3**, **4**, **5**, **6**]

INSERTION-SORT

Idee: Nimm Element und sortiere es an der richtigen Stelle ein.

```
def insertionsort(liste):
    for i in range(0, len(liste) - 1):
        wert = liste[i]
        j = i
        while j > 0 and liste[j-1] > wert:
            liste[j] = liste[j-1]
            j = j-1
        liste[j] = wert

    return liste
```

INSERTION-SORT

Wie gut ist Insertion-Sort?

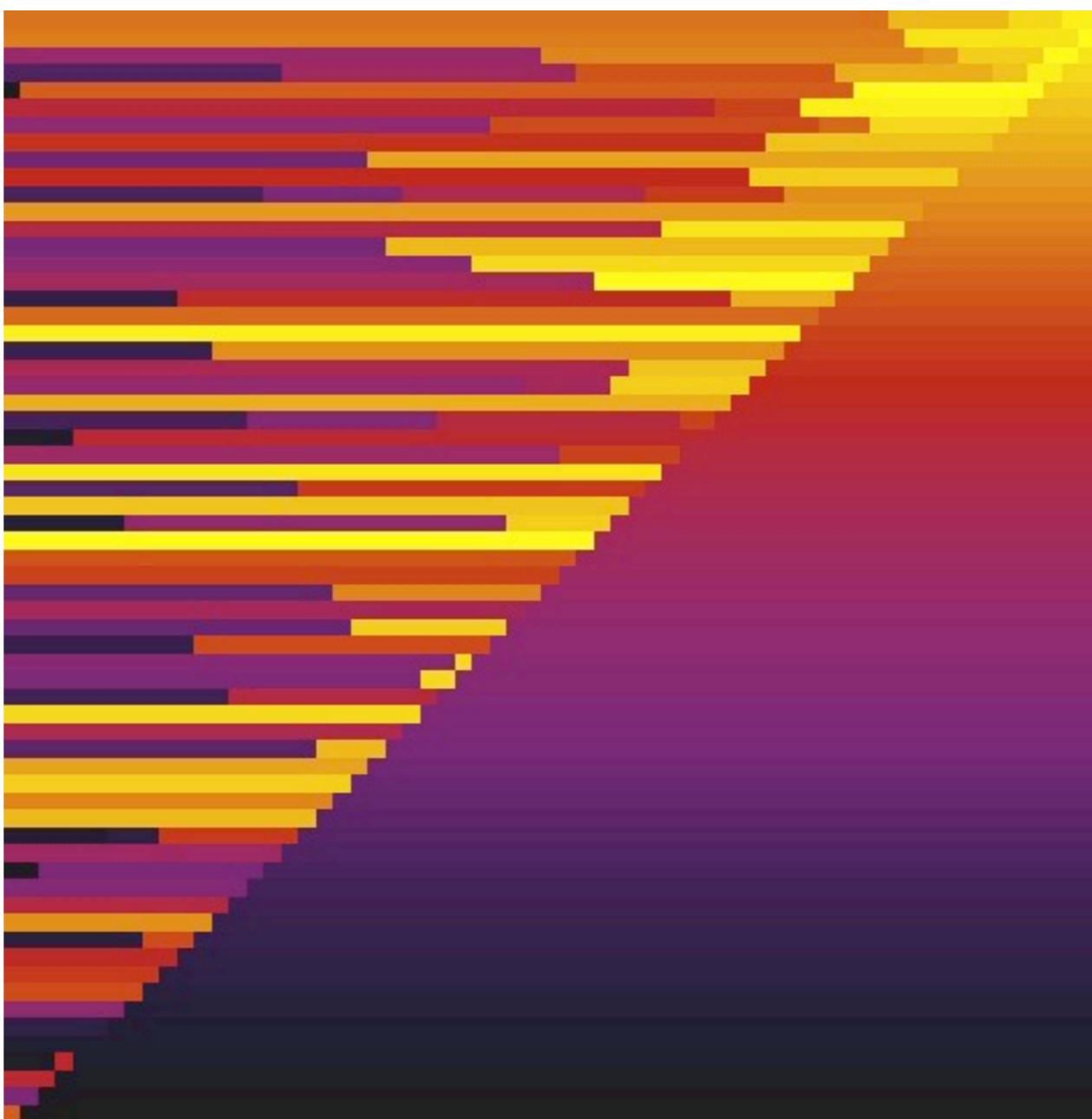
Anzahl Vergleiche?

Anzahl Kopiervorgänge?

```
>>> a = zufallsliste(10000)
>>> b = insertionsort_annotated(a)
24896481 Vergleiche
24906480 Kopieroperationen
22.4507021904 Sekunden
```

SELECTION-SORT: SORTIEREN DURCH AUSWAHL

Idee: Sortiere jeweils kleinstes Element nach vorne.



SELECTION-SORT

[4, 2, 1, 6, 3, 5]

[1, 2, 4, 6, 3, 5]

[1, 2, 4, 6, 3, 5]

[1, 2, 3, 6, 4, 5]

[1, 2, 3, 4, 6, 5]

[1, 2, 3, 4, 5, 6]

SELECTION-SORT

Idee: Sortiere jeweils kleinstes Element nach vorne.

```
def selectionsort(liste):
    for i in range(0, len(liste) - 1):
        minimum_index = i
        for j in range(i+1, len(liste) - 1):
            if liste[minimum_index] > liste[j]:
                minimum_index = j

        temp = liste[i]
        liste[i] = liste[minimum_index]
        liste[minimum_index] = temp

    return liste
```

SELECTION-SORT

Wie gut ist Selection-Sort?

Anzahl Vergleiche?

Anzahl Kopiervorgänge?

```
>>> a = zufallsliste(10000)
>>> b = selectionsort_annotated(a)
49995000 Vergleiche
29997 Kopieroperationen
17.5488231182 Sekunden
```



python

Sortieren
und Rekursion

Rekursion

3

Niels Lohmann
Stefanie Behrens

Lutz Hellmig
Karsten Wolf

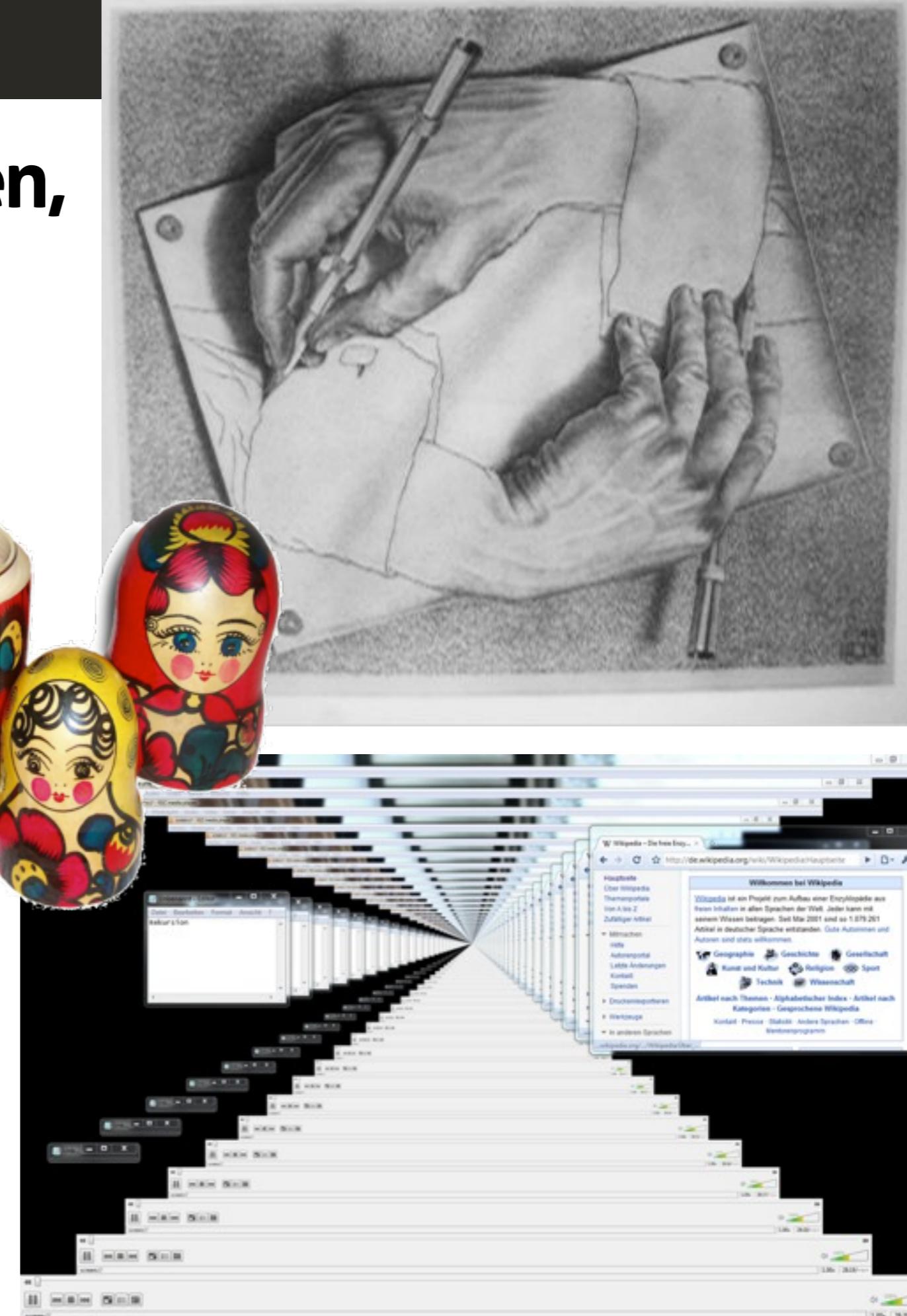
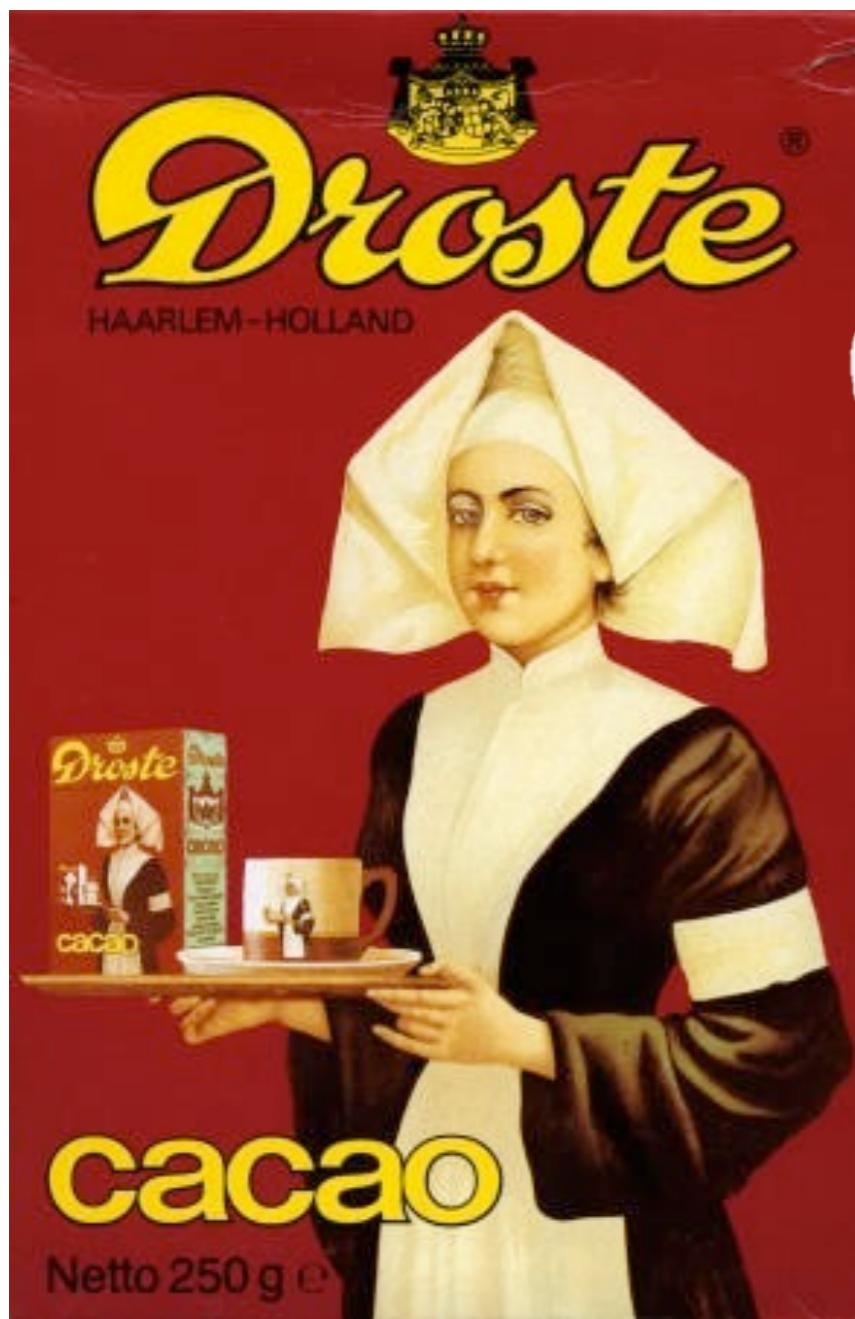
Universität
Rostock



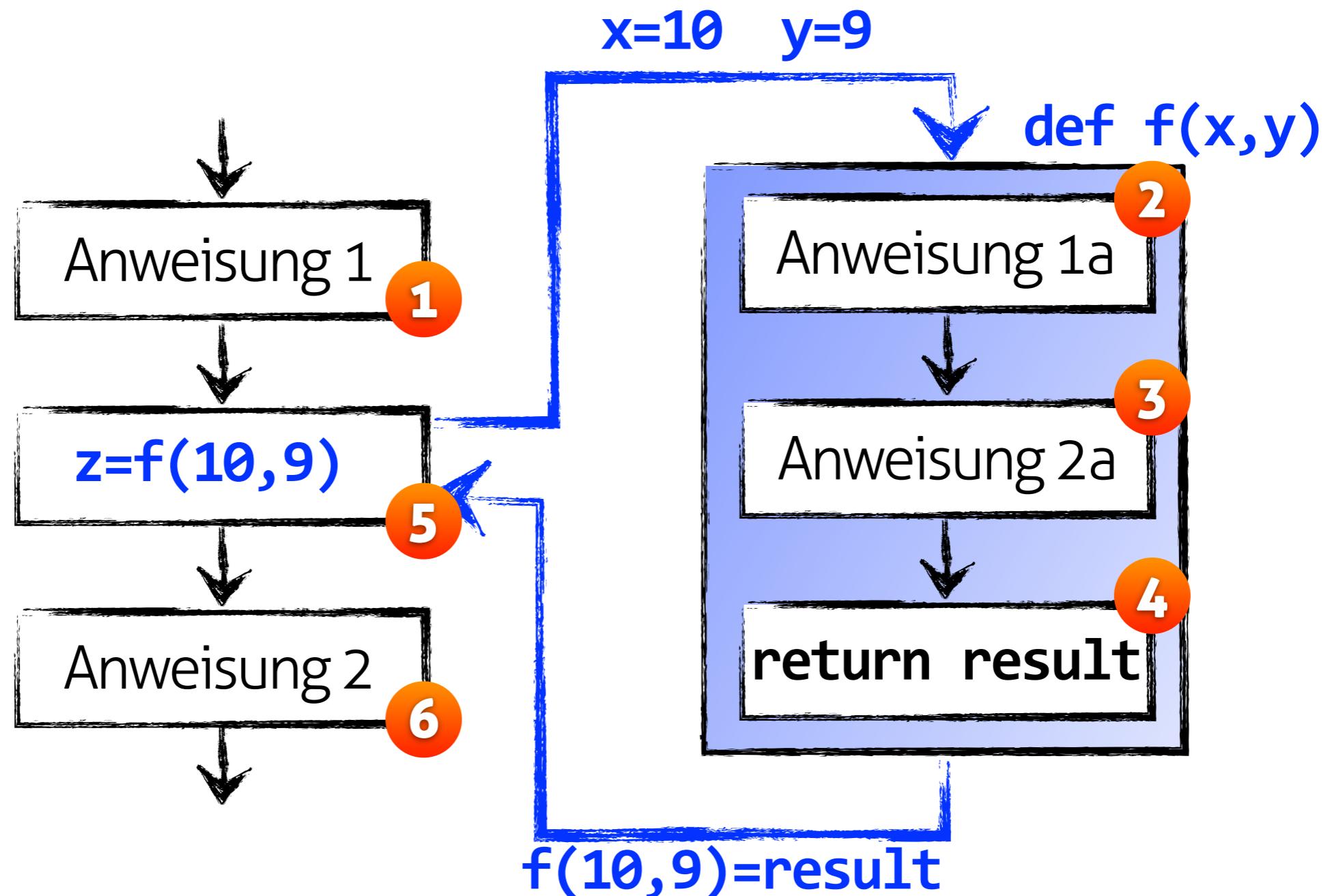
Traditio et Innovatio

REKURSION

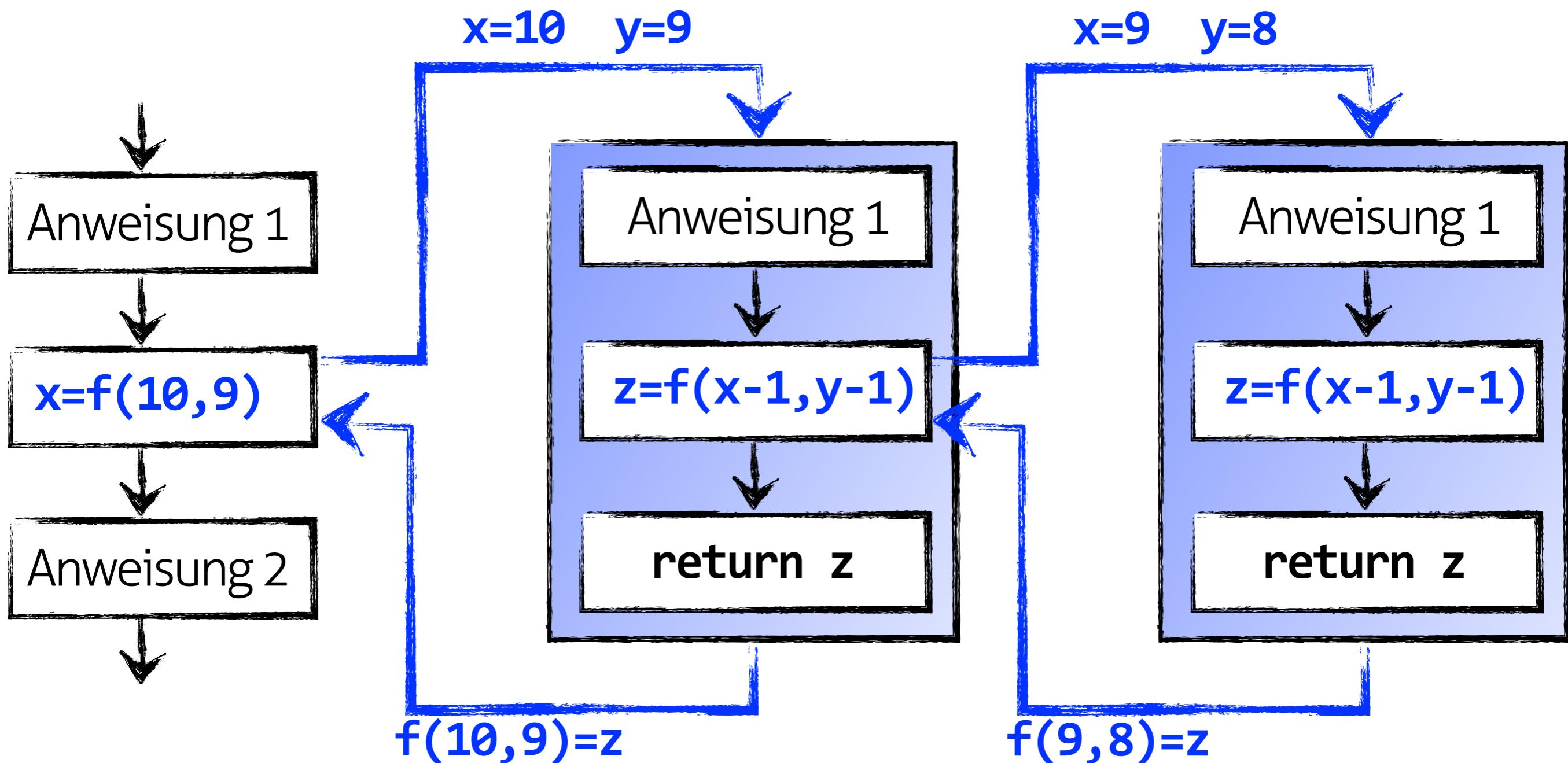
Um Rekursion zu verstehen,
muss man eigentlich nur
Rekursion verstehen!



FUNKTIONSAUFRUF

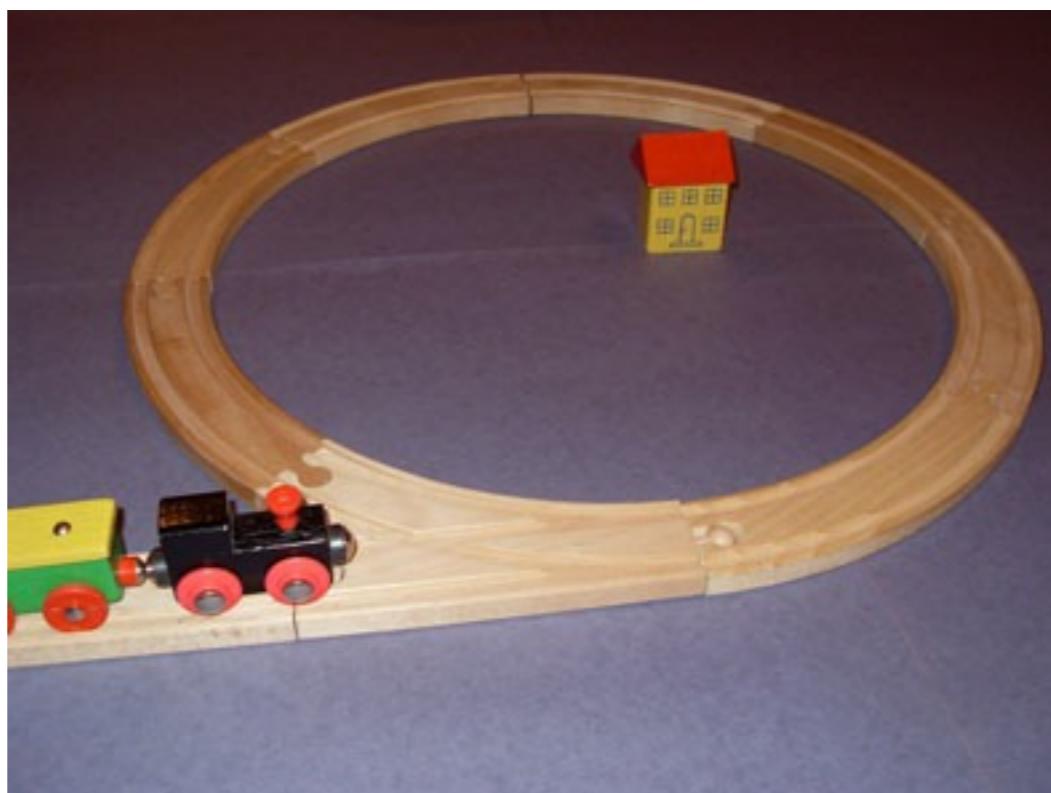


REKURSIVER FUNKTIONSAUFRUF



REKURSIVER FUNKTIONSAUFRUF

Rekursion = Endlosschleife?



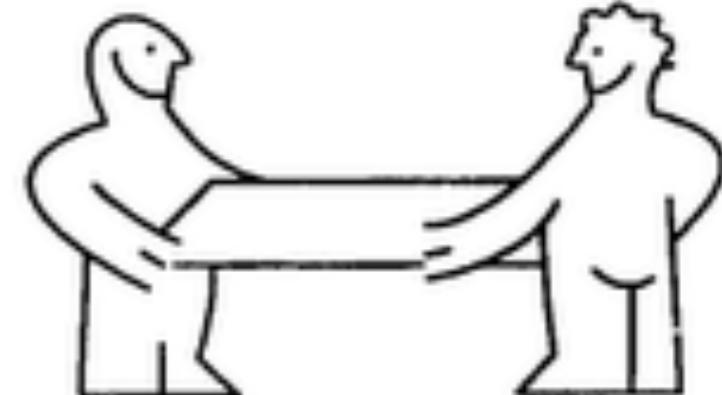
REKURSION = TEILEN UND HERRSCHEN

kleine Probleme sind einfacher als große

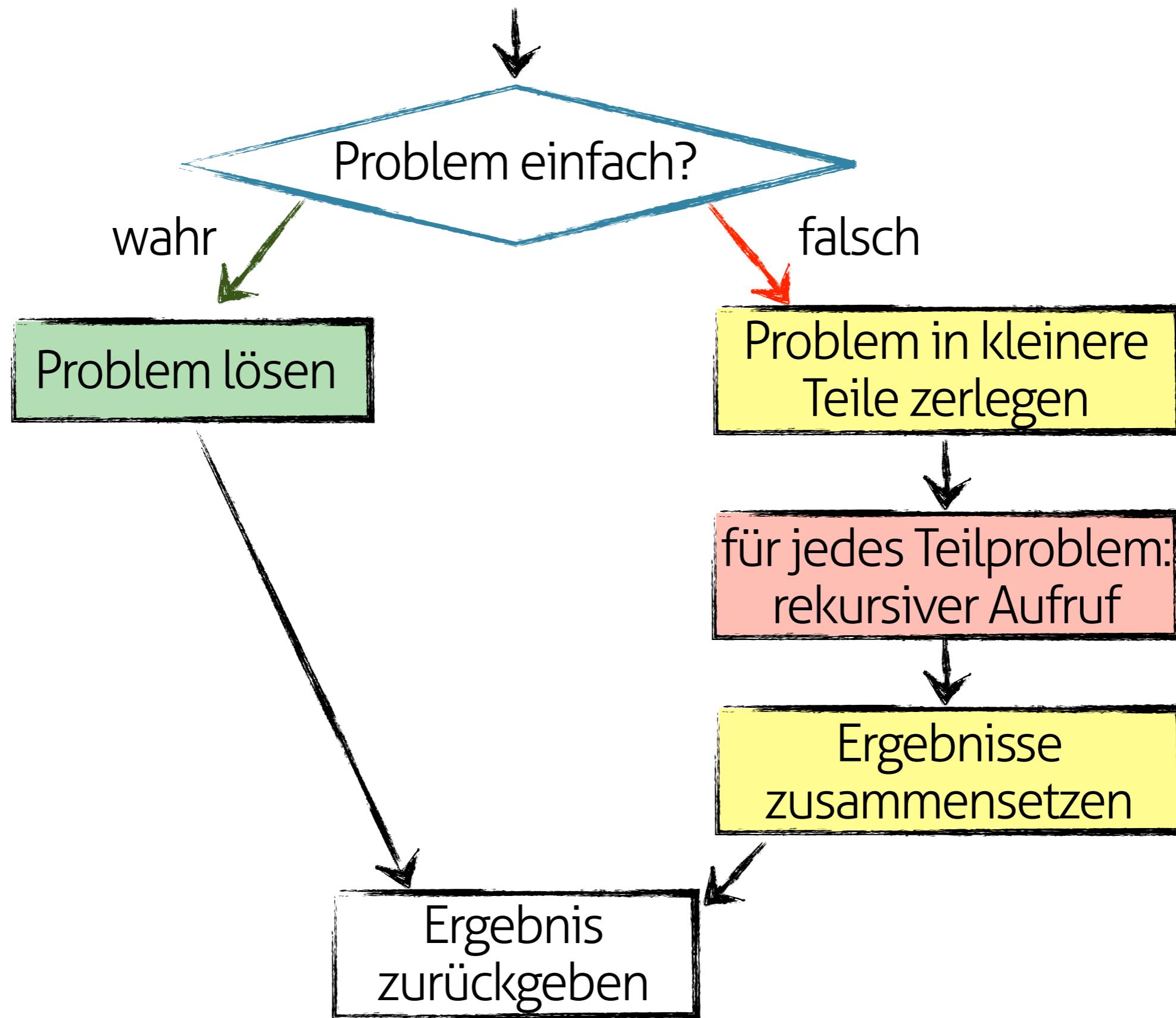
Teilen und Herrschen:

große Probleme in kleine aufteilen

bei jedem rekursiven Aufruf Problem **kleiner machen...**
... und irgendwann auch **lösen**



REKURSION = TEILEN UND HERRSCHEN



REKURSION

Jeder rekursive Algorithmus kann prinzipiell auch imperativ realisiert werden und umgekehrt.

Rekursive Funktionen

- sind oft durch **wenig Quelltext** darstellbar,
- sind manchmal leichter zu formulieren als imperative Lösungen,

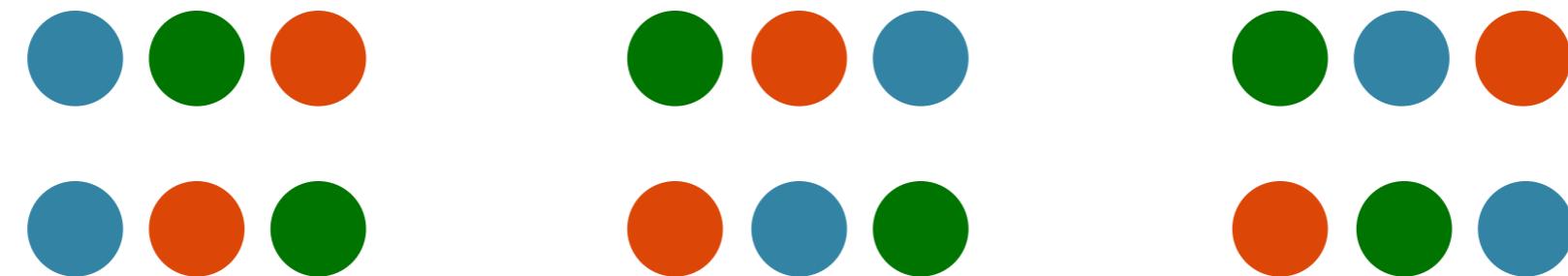
aber...

- benötigen z.T. enorme **Ressourcen** an Speicherplatz und Rechenzeit
- erfordern viel **Übung**

EINFACHES BEISPIEL: FAKULTÄT

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n - 1)! & \text{falls } n > 0 \end{cases}$$

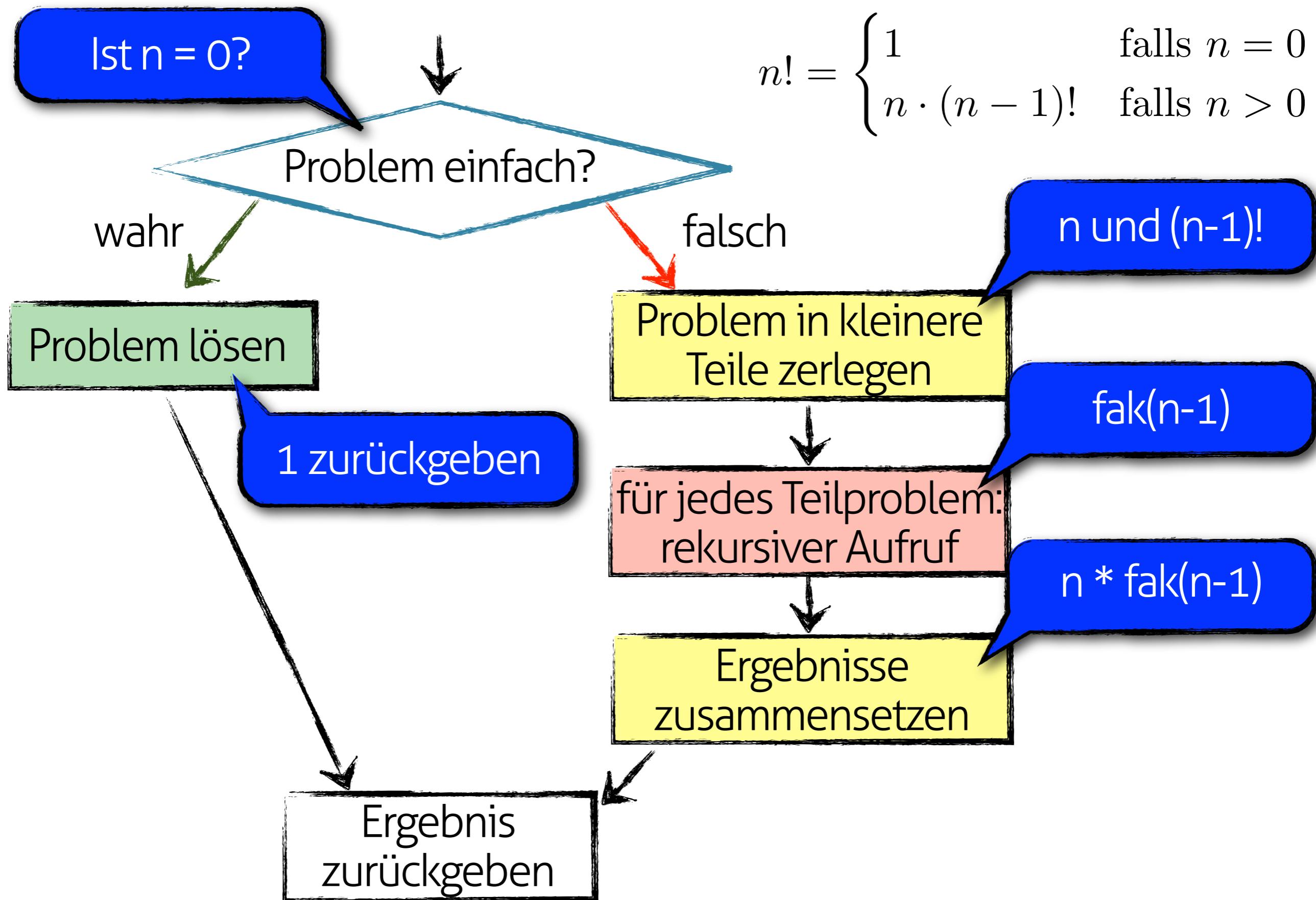
n Dinge können auf **n!** Weisen geordnet werden



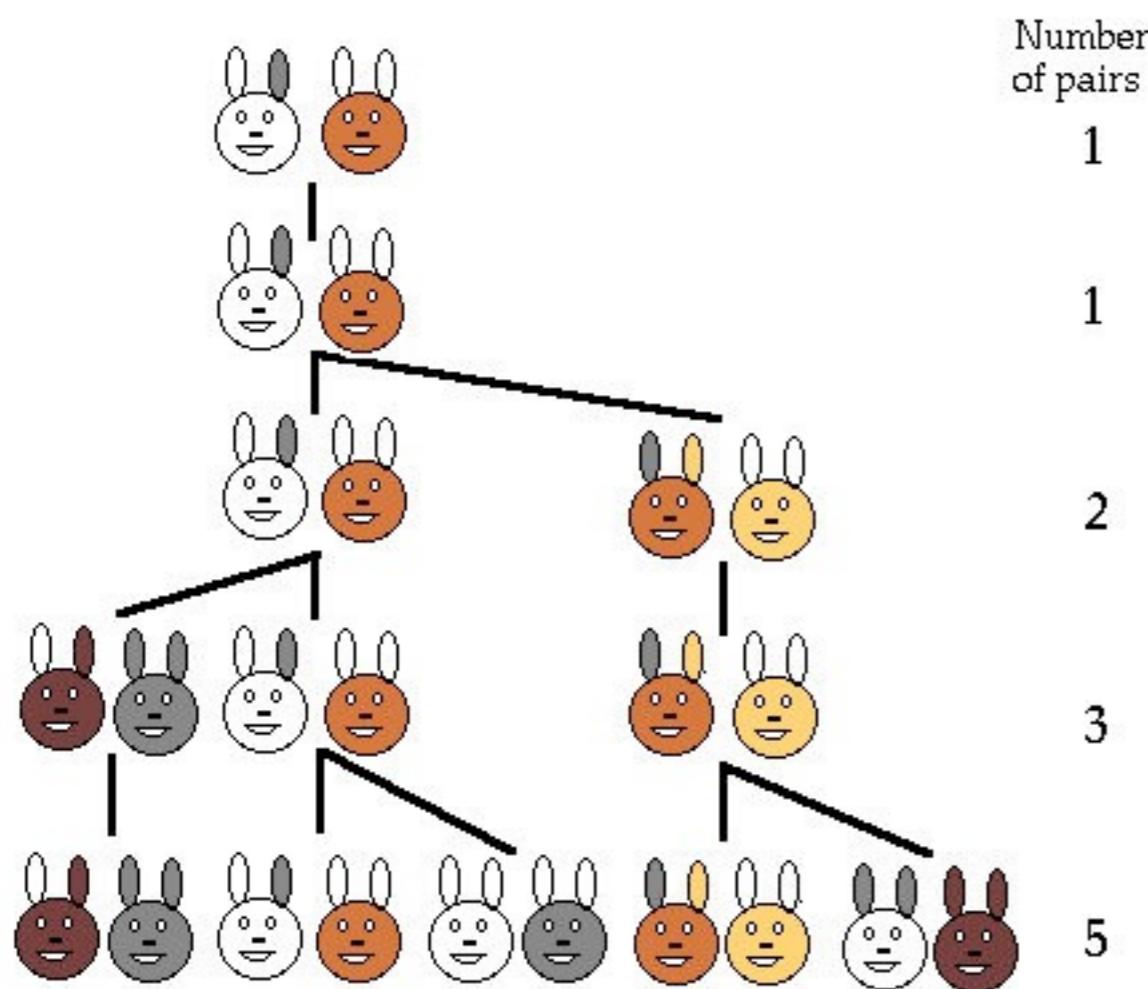
$$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 \cdot 0! = 3 \cdot 2 \cdot 1 \cdot 1 = 6$$



EINFACHES BEISPIEL: FAKULTÄT

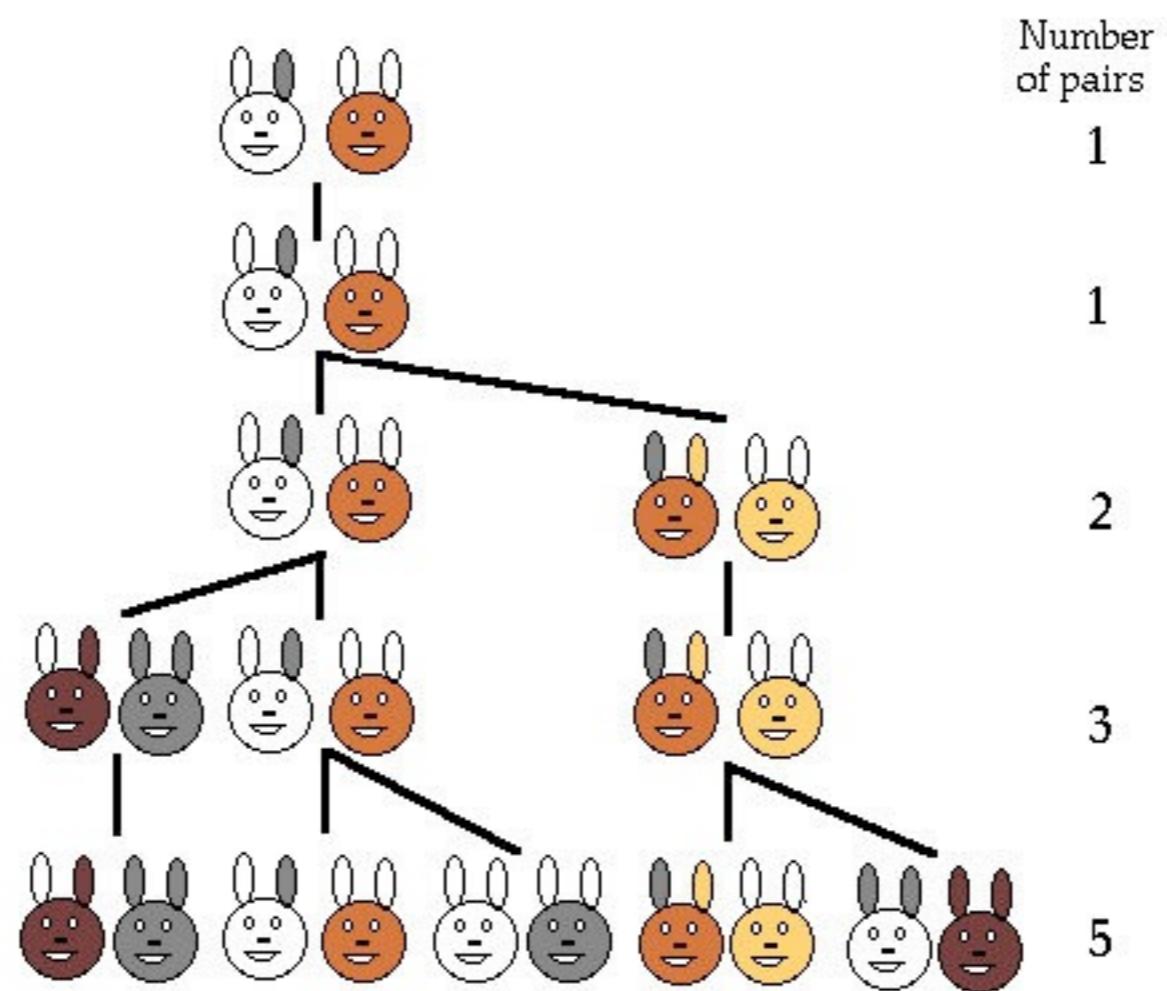


FIBONACCI-FOLGE



FIBONACCI-FOLGE

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n - 1) + fib(n - 2) & \text{falls } n > 1 \end{cases}$$



FIBONACCI-FOLGE

Ist $n=0$ oder $n=1$?



Problem einfach?

wahr

Problem lösen

0 oder 1 zurückgeben

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n - 1) + fib(n - 2) & \text{falls } n > 1 \end{cases}$$

falsch

Problem in kleinere Teile zerlegen

$fib(n-1), fib(n-2)$

für jedes Teilproblem:
rekursiver Aufruf

$fib(n-1), fib(n-2)$

Ergebnisse zusammensetzen

$fib(n-1)+fib(n-2)$

Ergebnis zurückgeben

PALINDROME

Ein **Palindrom** ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

- OTTO
- LAGERREGAL
- REITTIER
- RELIEFPFEILER
- RENTNER
- ROTOR
- EBBE
- MAOAM
- RADAR
- TOT

PALINDROME

Ein **Palindrom** ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

Rekursive Definition:

1. Worte mit **einem Buchstaben** sind Palindrome.
2. Worte sind Palindrome, falls:
 - der **erste und letzte Buchstabe gleich** sind, und
 - das **Wort ohne das erste und letzte Zeichen** ein Palindrom ist.

PALINDROME

Ein **Palindrom** ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

Rekursive Definition:

1. Worte mit **einem Buchstaben** sind Palindrome.
2. Worte sind Palindrome, falls:
 - der **erste und letzte Buchstabe gleich** sind, und
 - das **Wort ohne das erste und letzte Zeichen** ein Palindrom ist.

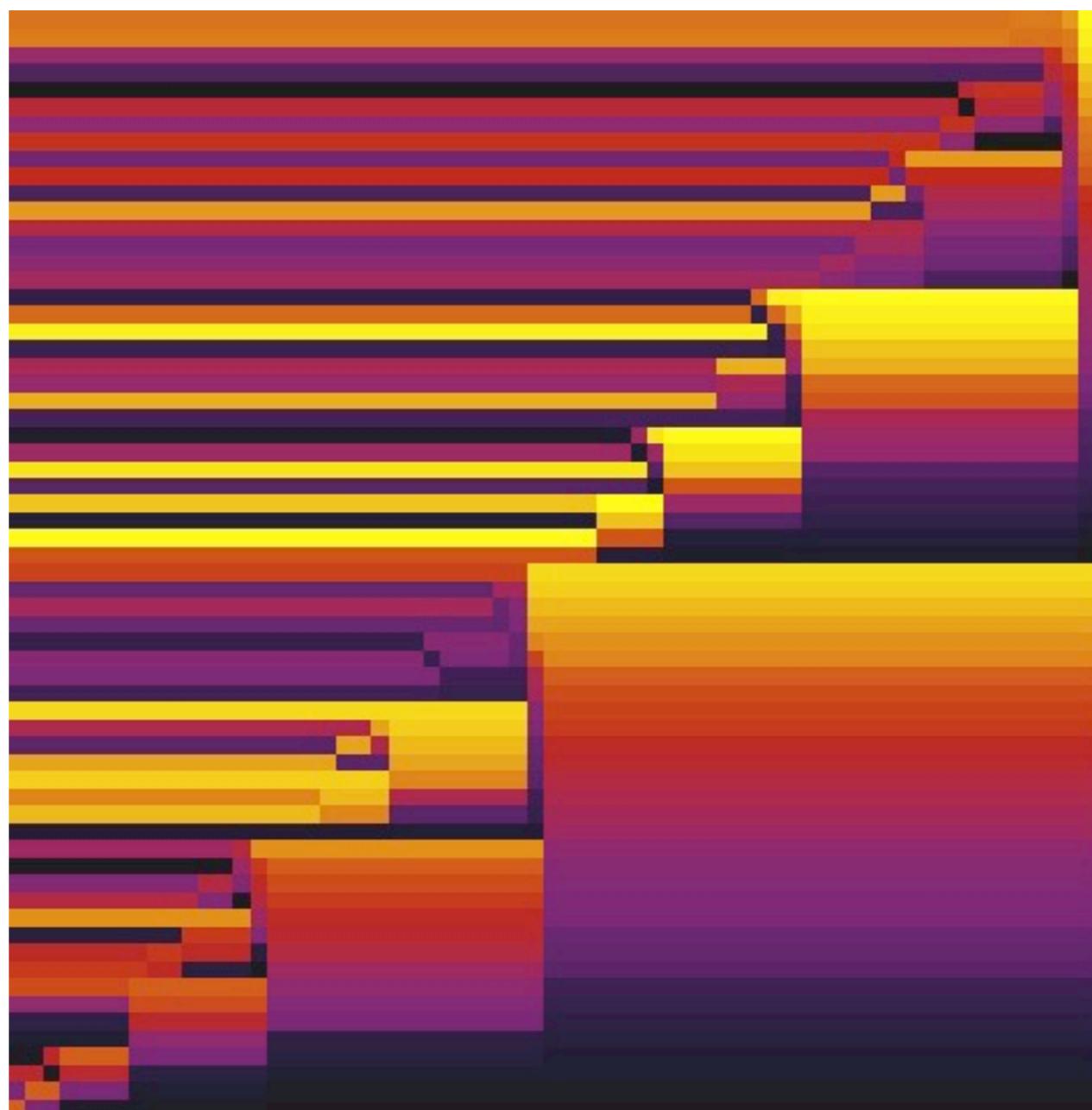
PALINDROME

- 1.** Worte mit **einem Buchstaben** sind Palindrome.
- 2.** Worte sind Palindrome, falls:
 - der **erste und letzte Buchstabe gleich** sind, und
 - das **Wort ohne das erste und letzte Zeichen** ein Palindrom ist.

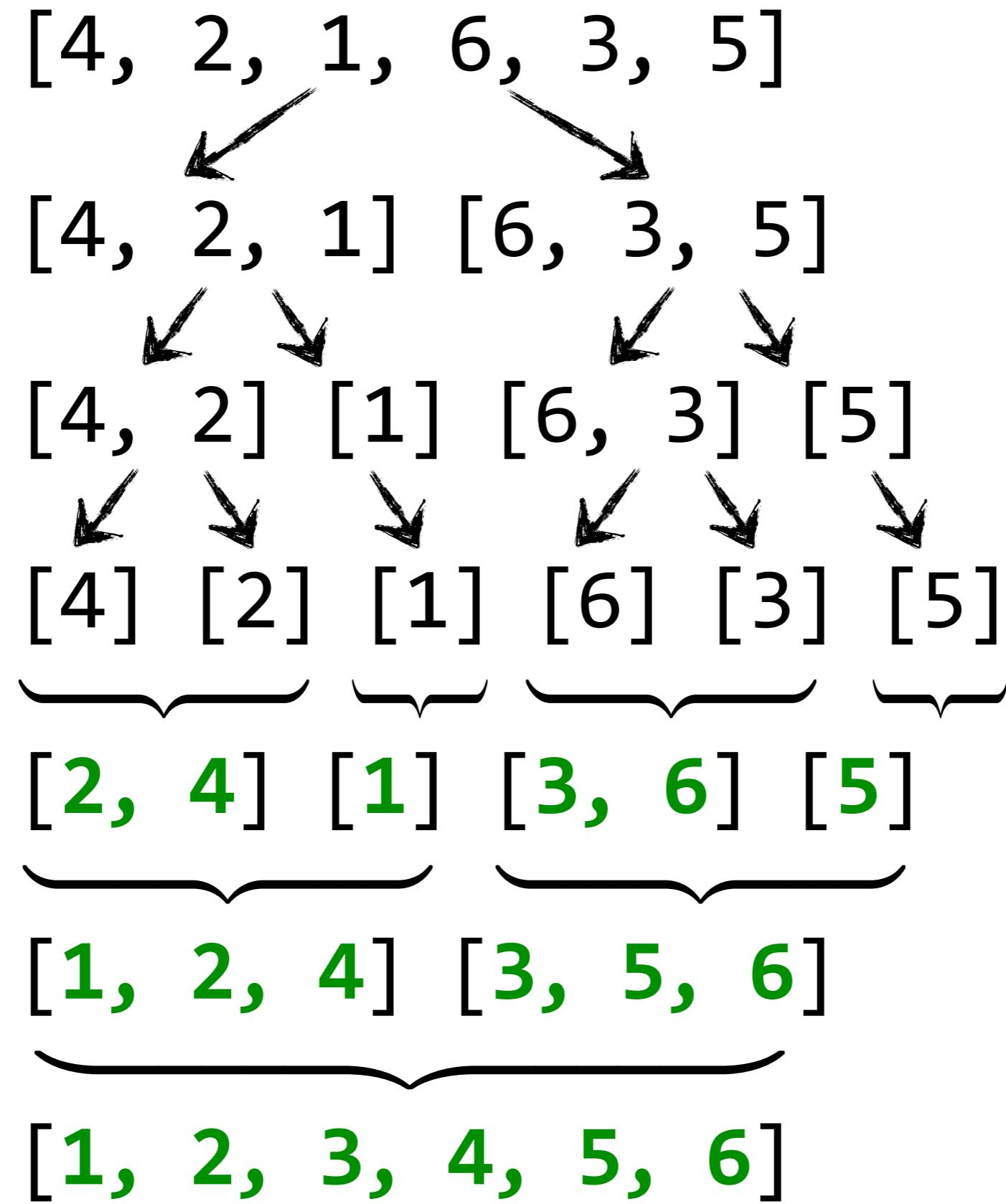
```
def pal(wort):  
    if len(wort) == 0 or len(wort) == 1:  
        return True  
    else:  
        return (wort[0]==wort[-1]) and pal(wort[1:-1])
```

MERGE-SORT: SORTIEREN DURCH VERSCHMELZEN

Idee: Teile Liste in zwei kleine, sortiere diese und verschmelze sie wieder zu einer Liste.



MERGE-SORT



MERGE-SORT

Idee: Teile Liste in zwei kleine, sortiere diese und verschmelze sie wieder zu einer Liste.

```
def mergesort(liste):
    if len(liste) <= 1:
        return liste
    else:
        liste1 = []
        liste2 = []
        for i in range(0, len(liste)):
            if i < (len(liste)/2):
                liste1.append(liste[i])
            else:
                liste2.append(liste[i])

    return merge(mergesort(liste1),mergesort(liste2))
```

MERGE-SORT

Idee: Teile Liste in zwei kleine, sortiere diese und verschmelze sie wieder zu einer Liste.

```
def merge(liste1, liste2):
    ergebnis = []
    i1 = 0
    i2 = 0

    while i1 < len(liste1) and i2 < len(liste2):
        if liste1[i1] < liste2[i2]:
            ergebnis.append(liste1[i1])
            i1 = i1 + 1
        else:
            ergebnis.append(liste2[i2])
            i2 = i2 + 1

    while i1 < len(liste1):
        ergebnis.append(liste1[i1])
        i1 = i1 + 1
    while i2 < len(liste2):
        ergebnis.append(liste2[i2])
        i2 = i2 + 1

    return ergebnis
```

MERGE-SORT

Wie gut ist Merge-Sort?

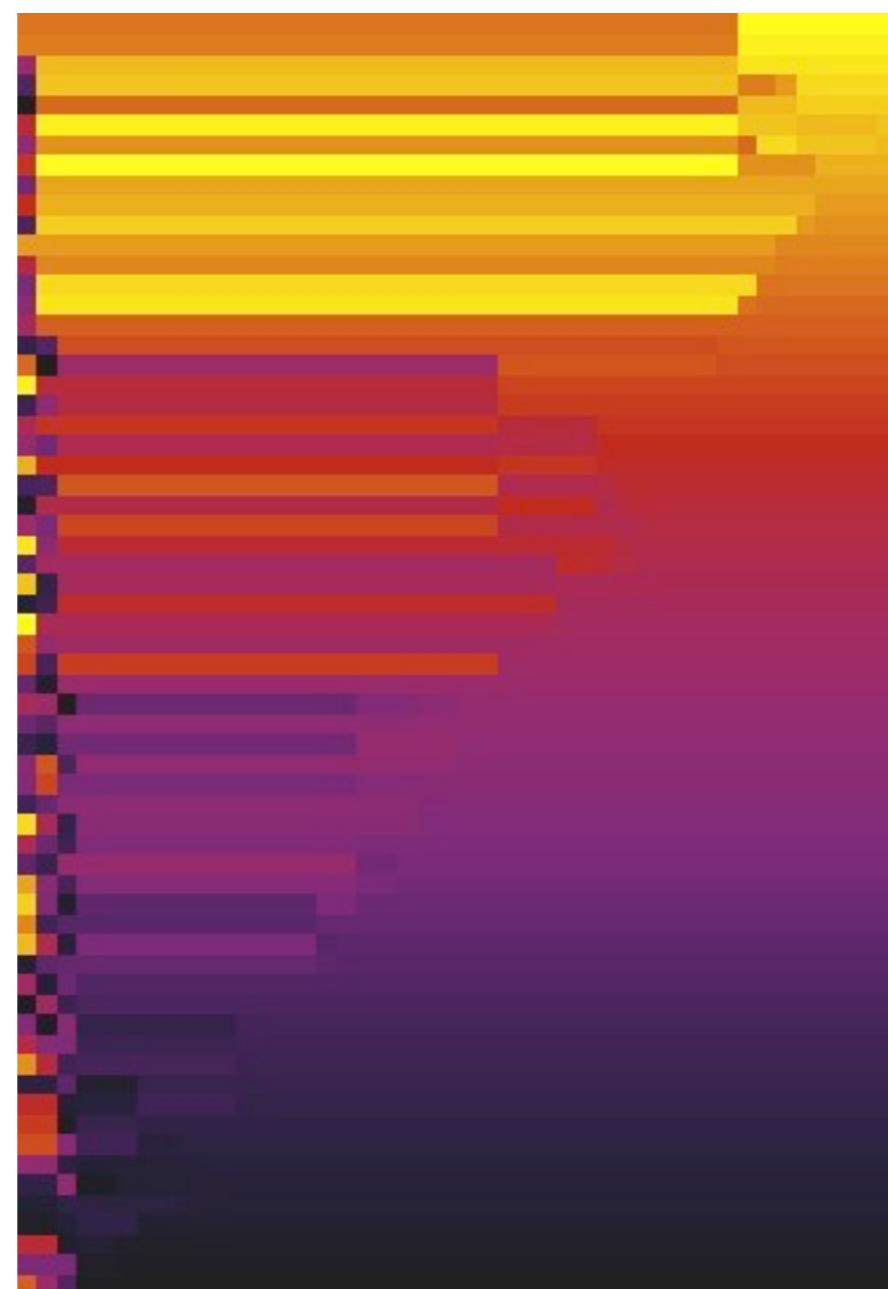
Anzahl Vergleiche?

Anzahl Kopiervorgänge?

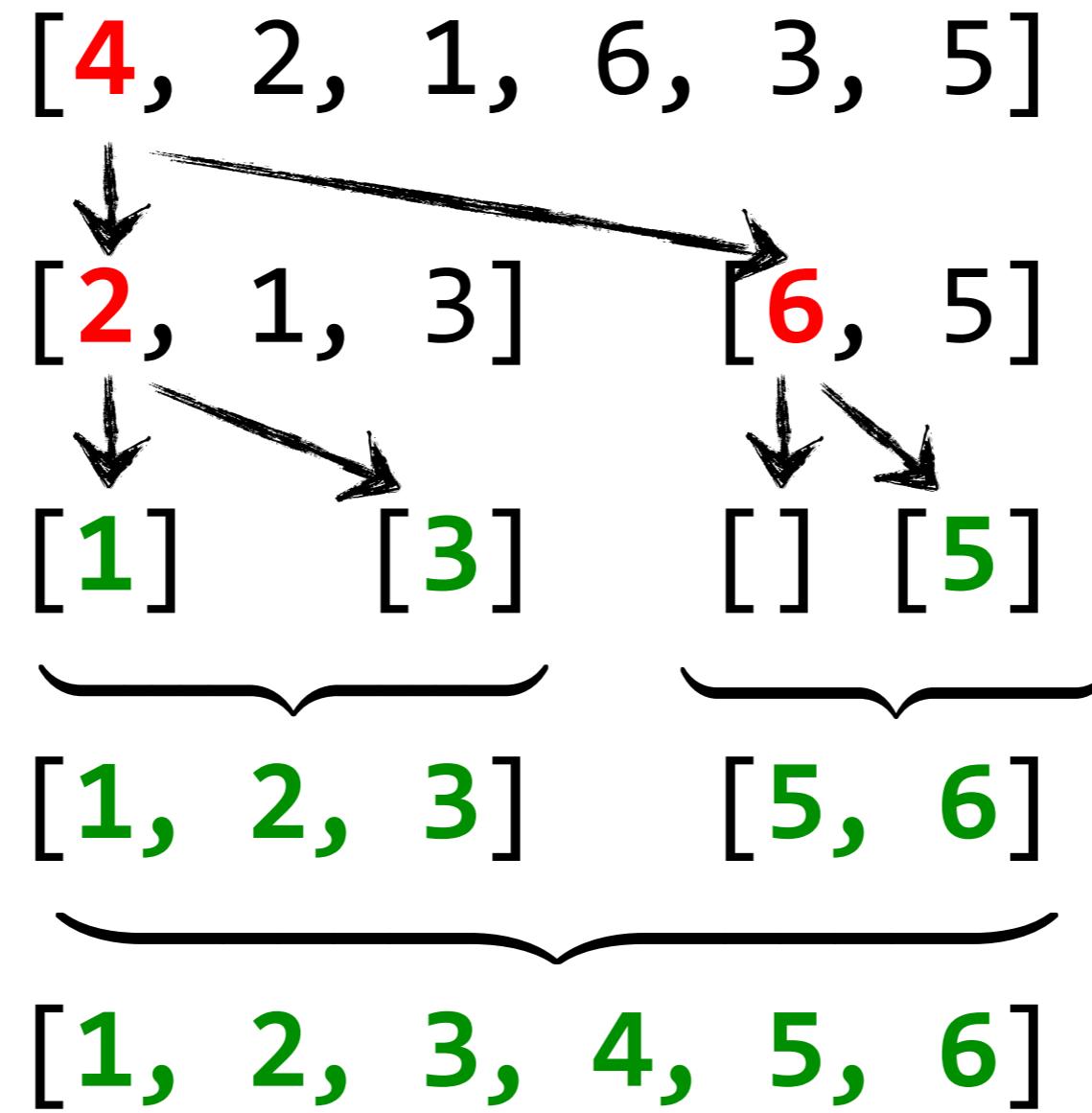
```
>>> a = zufallsliste(10000)
>>> b = mergesort_annotated(a)
120232 Vergleiche
267232 Kopieroperationen
0.403515815735 Sekunden
```

QUICKSORT: SORTIEREN DURCH PARTITIONIEREN

Idee: Nimm ein Element und sortiere kleinere Elemente links und größere Elemente rechts ein.



QUICKSORT



QUICKSORT

Idee: Nimm ein Element und sortiere kleinere Elemente links und größere Elemente rechts ein.

```
def quicksort(liste):
    if len(liste) <= 1:
        return liste
    else:
        pivot = liste[0]
        lesser = []
        greater = []
        for i in range(1, len(liste)):
            if liste[i] < pivot:
                lesser.append(liste[i])
            else:
                greater.append(liste[i])
    return quicksort(lesser)
+ [pivot] + quicksort(greater)
```

QUICKSORT

Wie gut ist Quicksort?

Anzahl Vergleiche?

Anzahl Kopiervorgänge?

```
>>> a = zufallsliste(10000)
>>> b = quicksort_annotated(a)
589024 Vergleiche
1197848 Kopieroperationen
0.576086044312 Sekunden
```



python

Sortieren
und Rekursion

Dynamische Programmierung

4

Niels Lohmann
Stefanie Behrens

Lutz Hellmig
Karsten Wolf

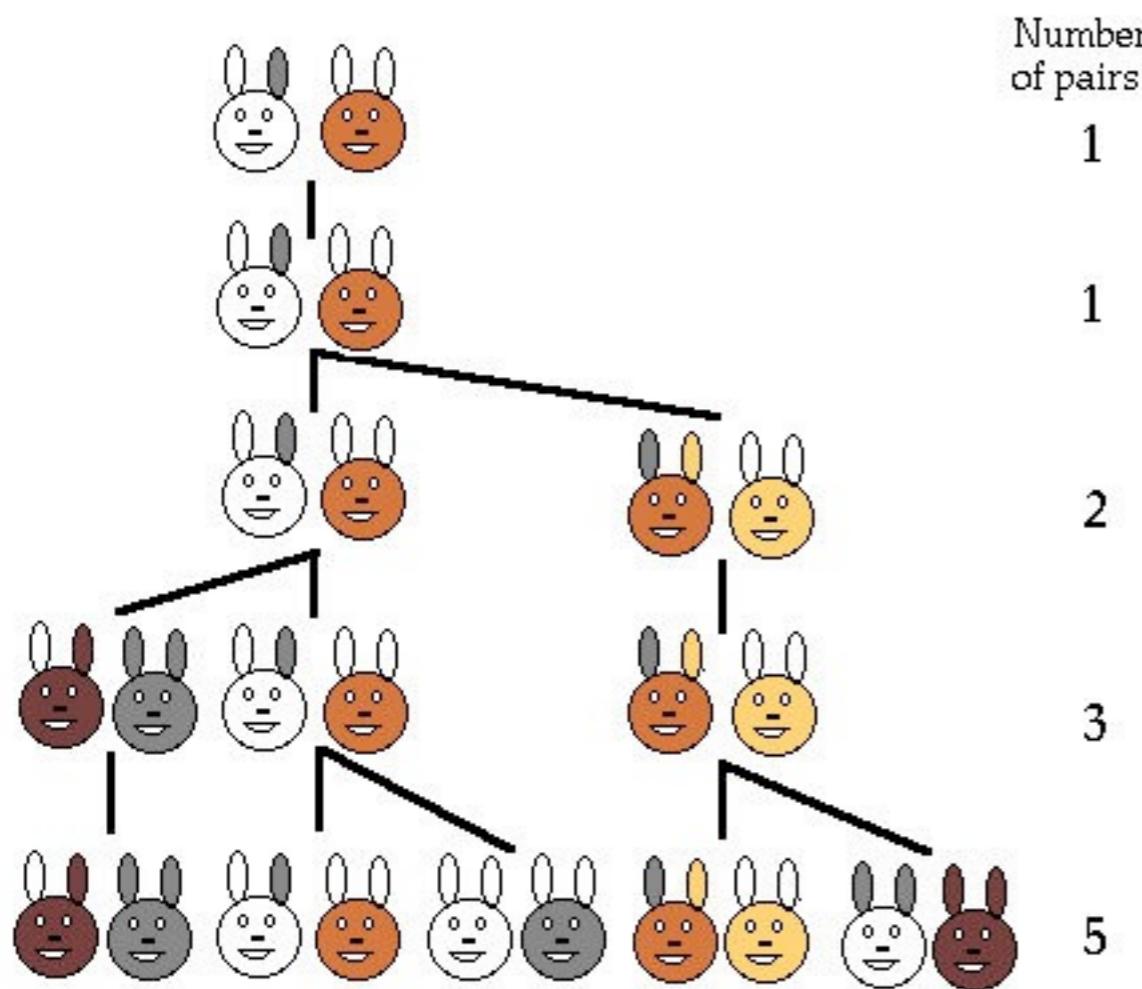
Universität
Rostock



Traditio et Innovatio

FIBONACCI-FOLGE

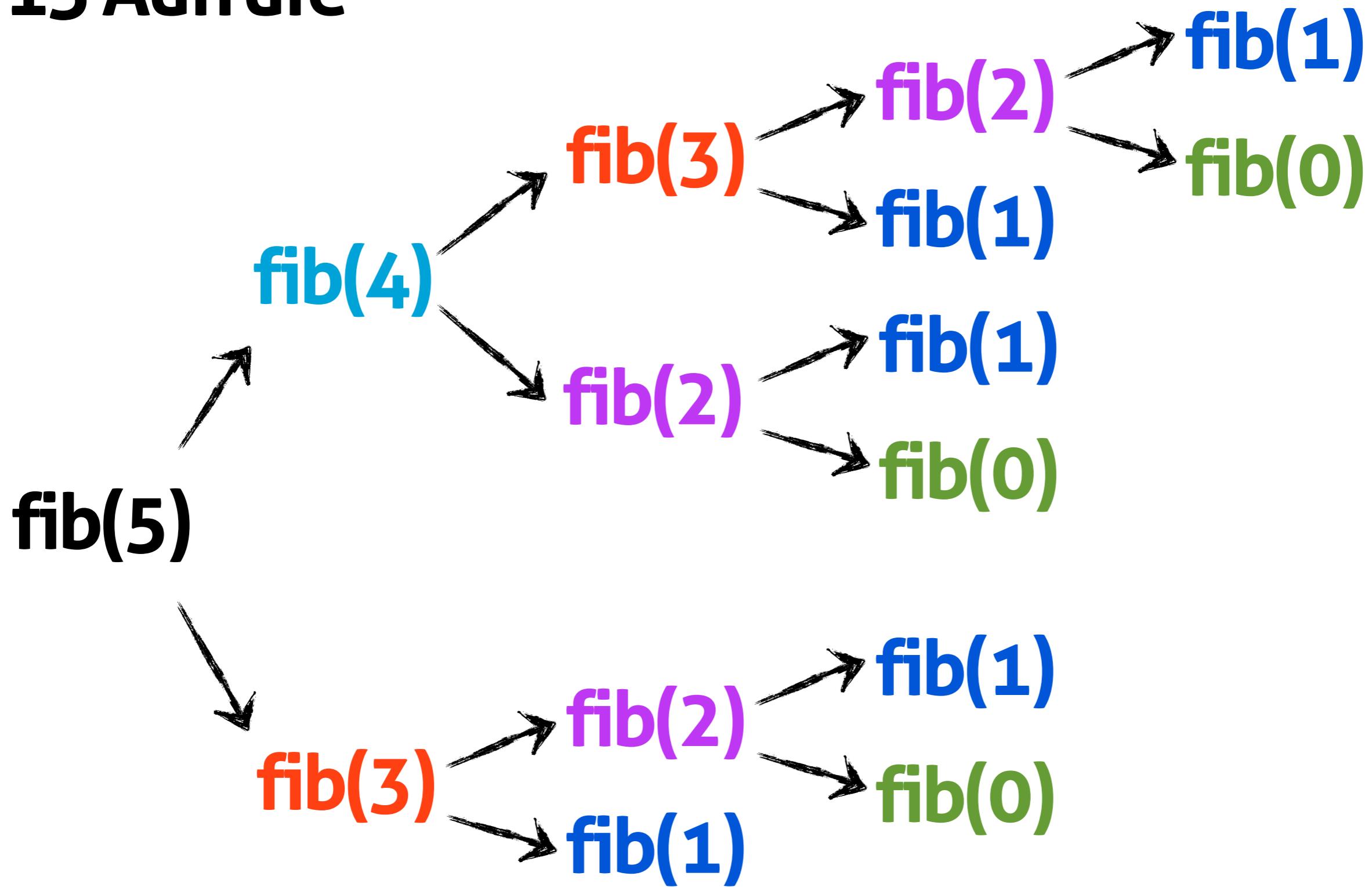
$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n - 1) + fib(n - 2) & \text{falls } n > 1 \end{cases}$$



fib(100) = ???

FIBONACCI-FOLGE

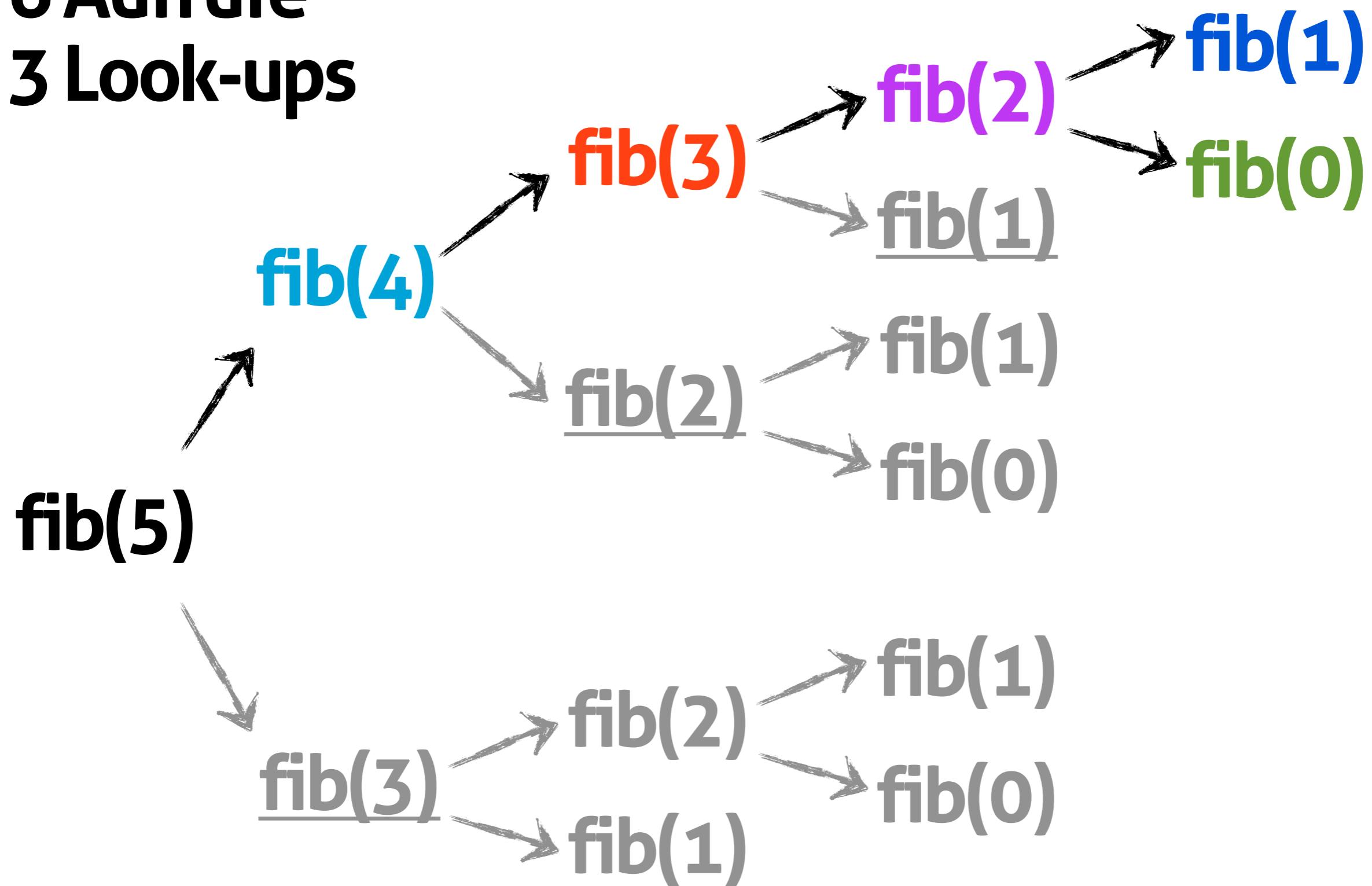
15 Aufrufe



FIBONACCI-FOLGE

6 Aufrufe

3 Look-ups



UMSETZUNG: DICTIONARIES

```
cache = dict() <-- leeres Dictionary anlegen
```

```
def fib_cached(n):  
    global cache <-- auf Dictionary verweisen
```

```
if n not in cache: <-- Test auf Eintrag an Stelle n
```

```
    if n == 0:
```

```
        cache[n] = 0 <-- Speichern von Wert an Stelle n
```

```
    if n == 1:
```

```
        cache[n] = 1
```

```
    if n > 1:
```

```
        cache[n] = fib_cached(n-1) + fib_cached(n-2)
```

```
return cache[n] <-- Lesen von Wert an Stelle n
```



python

Sortieren
und Rekursion

Niels Lohmann
Stefanie Behrens

Lutz Hellmig
Karsten Wolf

Universität
Rostock



Traditio et Innovatio