

Informatik I: Einführung in die Programmierung

12. Rekursion, Endrekursion, Iteration

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Prof. Dr. Peter Thiemann

22. Dezember 2020



Rekursion verstehen

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Bäume sind induktiv definiert:
 - Ein Baum ist entweder leer \square oder
 - ein Knoten mit einer Markierung und einer Liste von Teilbäumen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Bäume sind induktiv definiert:
 - Ein Baum ist entweder leer \square oder
 - ein Knoten mit einer Markierung und einer Liste von Teilbäumen.
- Schema für Funktionen F auf Bäumen, die natürlich rekursiv sind:

- $F(\square) = A$

- $F \left(\begin{array}{c} \text{mark} \\ \swarrow \quad \downarrow \quad \searrow \\ t_0 \quad \dots \quad t_{n-1} \end{array} \right) = B(\text{mark}, F(t_0), \dots, F(t_{n-1}))$

- B ist ein Programmstück, das die Markierung der Wurzel, sowie die Ergebnisse der **Funktionsaufrufe von F auf den Teilbäumen** verwenden darf.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
@dataclass
class Tree:
    mark : Any
    children : list['Tree']

def tree_skeleton (tree : Optional[Tree]) -> Any:
    if tree is None:
        return "A" # result for empty tree
    else:
        # compute B from
        # - tree.mark
        # - tree_skeleton(tree.children[0])
        # - ...
        # - tree_skeleton(tree.children[n-1])
        # where n = len (tree.children)
        return "B"
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Binäre Suche

Rekursion
verstehen

**Binäre
Suche**

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Jede Rekursion folgt einer Baumstruktur



Binäre Suche

- Eingabe
 - aufsteigend sortierte Liste `lst : list[T]`
 - Suchbegriff `key : T`
- Ausgabe
 - falls `key` in `lst`: `i` sodass `lst[i] == key`
 - andernfalls: `None`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Jede Rekursion folgt einer Baumstruktur



Binäre Suche

- Eingabe
 - aufsteigend sortierte Liste `lst : list[T]`
 - Suchbegriff `key : T`
- Ausgabe
 - falls `key` in `lst`: `i` sodass `lst[i] == key`
 - andernfalls: `None`

Idee

- Betrachte die Liste wie einen binären Suchbaum
- Wähle ein Element als Wurzel: alle Elemente links davon sind kleiner, rechts davon größer
- Optimierte die Effizienz durch geschickte Wahl der Wurzel

Rekursion
verstehen

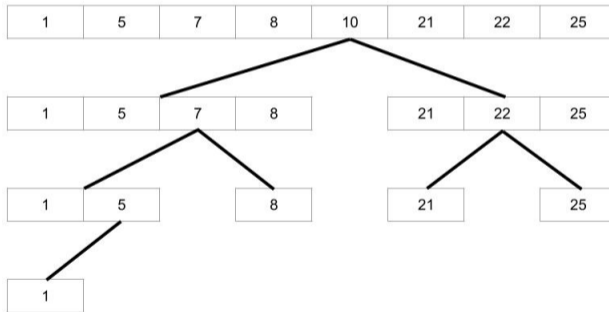
Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Rekursion
verstehen

Binäre
Suche

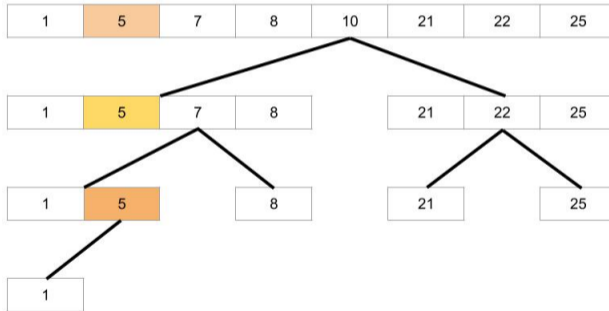
Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche (5) = 1



Rekursion
verstehen

Binäre
Suche

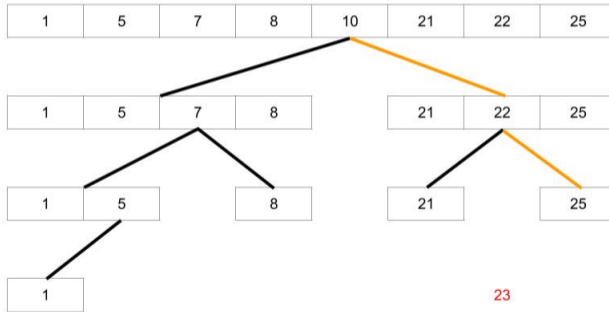
Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche (23) = None



Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche

Elementtyp int



```
def bsearch (lst : list[int], key : int) -> Optional[int]:
    n = len (lst)
    if n == 0:
        return None # key not in empty list
    m = n//2          # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch (lst[:m], key)
    else: # lst[m] < key
        r = bsearch (lst[m+1:], key)
        return None if r is None else r+m+1
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Funktioniert, aber `lst[:m]` und `lst[m+1:]` erzeugen jeweils **Kopien** der halben Liste

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Funktioniert, aber `lst[:m]` und `lst[m+1:]` erzeugen jeweils **Kopien** der halben Liste
- Alternative: Suche jeweils zwischen Startpunkt und Endpunkt in `lst`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Funktioniert, aber `lst[:m]` und `lst[m+1:]` erzeugen jeweils **Kopien** der halben Liste
- Alternative: Suche jeweils zwischen Startpunkt und Endpunkt in `lst`
- Für den rekursiven Aufruf muss dann nur der Start- bzw. Endpunkt verschoben werden

```
def bsearch (lst : list[int], key : int):  
    return bsearch2 (lst, key, 0, len (lst))  
  
def bsearch2 (lst : list[int], key : int,  
              low : int, high : int):  
    """ search for key in lst between low  
    (inclusive) and high (exclusive)  
    assumes low <= high """
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche ohne Kopieren



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    n = hi - lo      # length of list segment
    if n == 0:
        return None # key not in empty segment
    m = lo + n//2    # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch2 (lst, key, lo, m)
    else: # lst[m] < key
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche ohne Kopieren



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):  
    n = hi - lo      # length of list segment  
    if n == 0:  
        return None # key not in empty segment  
    m = lo + n//2   # position of root  
    if lst[m] == key:  
        return m  
    elif lst[m] > key:  
        return bsearch2 (lst, key, lo, m)  
    else: # lst[m] < key  
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    n = hi - lo      # length of list segment
    if n == 0:
        return None # key not in empty segment
    m = lo + n//2    # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch2 (lst, key, lo, m)
    else: # lst[m] < key
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen

- $n == 0$ entspricht $hi - lo == 0$ und damit $lo == hi$



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    n = hi - lo      # length of list segment
    if n == 0:
        return None # key not in empty segment
    m = lo + n//2    # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch2 (lst, key, lo, m)
    else: # lst[m] < key
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen

- $n == 0$ entspricht $hi - lo == 0$ und damit $lo == hi$
- $lo + (hi - lo)//2$ entspricht $(lo + hi)//2$

Binäre Suche ohne Kopieren, vereinfacht



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):  
    if lo == hi:  
        return None # key not in empty segment  
    m = (lo + hi)//2 # position of root  
    if lst[m] == key:  
        return m  
    elif lst[m] > key:  
        return bsearch2 (lst, key, lo, m)  
    else: # lst[m] < key  
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche ohne Kopieren, vereinfacht



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):  
    if lo == hi:  
        return None # key not in empty segment  
    m = (lo + hi)//2 # position of root  
    if lst[m] == key:  
        return m  
    elif lst[m] > key:  
        return bsearch2 (lst, key, lo, m)  
    else: # lst[m] < key  
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen

Binäre Suche ohne Kopieren, vereinfacht



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    if lo == hi:
        return None # key not in empty segment
    m = (lo + hi)//2 # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch2 (lst, key, lo, m)
    else: # lst[m] < key
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen

- Jeder rekursive Aufruf von bsearch2 erfolgt in return.

Binäre Suche ohne Kopieren, vereinfacht



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    if lo == hi:
        return None # key not in empty segment
    m = (lo + hi)//2 # position of root
    if lst[m] == key:
        return m
    elif lst[m] > key:
        return bsearch2 (lst, key, lo, m)
    else: # lst[m] < key
        return bsearch2 (lst, key, m+1, hi)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beobachtungen

- Jeder rekursive Aufruf von `bsearch2` erfolgt in `return`.
- Solche Aufrufe heißen **endrekursiv**.



Definition

Endrekursive Funktionen haben nur endrekursive Aufrufe.

Elimination von Endrekursion durch Iteration

- Endrekursive Funktionen können durch `while`-Schleifen (**Iteration**) implementiert werden.
- Die **Abbruchbedingung** der Rekursion wird **negiert zur Bedingung** der `while`-Schleife.
- Der Rest des Funktionsrumpfs wird zum Rumpf der `while`-Schleife.
- Die **endrekursiven Aufrufe** werden zu **Zuweisungen an die Parameter**.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



bsearch2 ist endrekursive Funktion

Abbruchbedingung der Rekursion:

```
if lo == hi:  
    return None
```

wird negiert zur Bedingung der while-Schleife

```
while lo != hi:  
    ...  
else:  
    return None
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



bsearch2 ist endrekursive Funktion

Endrekursive Aufrufe

```
return bsearch2 (lst, key, lo, m)
```

werden zu Zuweisungen an die Parameter

```
lst, key, lo, hi = lst, key, lo, m
```

bzw. hier reicht

```
hi = m
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Binäre Suche ohne Kopieren, iterativ



```
def bsearch2 (lst : list[int], key : int, lo:int, hi:int):
    while lo != hi:
        m = (lo + hi)//2
        if lst[m] == key:
            return m
        elif lst[m] > key:
            hi = m      # bsearch2 (lst, key, lo, m)
        else: # lst[m] < key
            lo = m+1   # bsearch2 (lst, key, m+1, hi)
    else:
        return None
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def search(tree : Optional[Node], item : Any):  
    if tree is None:  
        return False  
    elif tree.mark == item:  
        return True  
    elif tree.mark > item:  
        return search(tree.left, item)  
    else:  
        return search(tree.right, item)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

- Gleiches Muster ... nicht überraschend



```
def search(tree : Optional[Node], item : Any):
    while tree is not None:
        if tree.mark == item:
            return True
        elif tree.mark > item:
            tree = tree.left
        else:
            tree = tree.right
    else:
        return False
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Potenzieren

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Rekursion als Definitionstechnik: Potenzieren



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$

Rekursion
verstehen

Binäre
Suche

Potenzieren

**Rekursive
Definition**

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder "informatisch" hingeschrieben

```
power (x, 0)    == 1
power (x, n+1) == x * power (x, n)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0)    == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0)    == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?
- Erinnerung: Induktive Definition der natürlichen Zahlen

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0) == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?
- Erinnerung: Induktive Definition der natürlichen Zahlen
 - Eine natürliche Zahl ist entweder 0 oder

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0)    == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?
- Erinnerung: Induktive Definition der natürlichen Zahlen
 - Eine natürliche Zahl ist entweder 0 oder
 - der Nachfolger $1 + (n)$ einer natürlichen Zahl n .

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0)    == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?
- Erinnerung: Induktive Definition der natürlichen Zahlen
 - Eine natürliche Zahl ist entweder 0 oder
 - der Nachfolger $1 + (n)$ einer natürlichen Zahl n .

■ In Bäumen: 0 1+

 |

 n

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

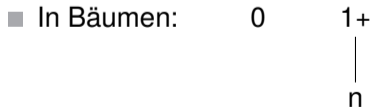
Lindenmayer
Systeme



- Mathematische Definition: $x^0 = 1$ $x^{n+1} = x \cdot x^n$
- Oder “informatisch” hingeschrieben

```
power (x, 0) == 1
power (x, n+1) == x * power (x, n)
```

- Wo ist da der Baum?
- Erinnerung: Induktive Definition der natürlichen Zahlen
 - Eine natürliche Zahl ist entweder 0 oder
 - der Nachfolger $1 + (n)$ einer natürlichen Zahl n .



- Daraus ergibt sich das Codegerüst.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power (x : complex, n : int):  
    """ x ** n for n >= 0 """  
    if n==0:  
        return 1  
    else: # n = 1+n'  
        return x * power (x, n-1)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

→ `power(2,3)` wählt else-Zweig und ruft auf:

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

→ `power(2,3)` wählt else-Zweig und ruft auf:

→ `power(2,2)` wählt else-Zweig und ruft auf:

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

- `power(2,3)` wählt else-Zweig und ruft auf:
 - `power(2,2)` wählt else-Zweig und ruft auf:
 - `power(2,1)` wählt else-Zweig und ruft auf:

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

- `power(2,3)` wählt else-Zweig und ruft auf:
 - `power(2,2)` wählt else-Zweig und ruft auf:
 - `power(2,1)` wählt else-Zweig und ruft auf:
 - `power(2,0)` wählt if-Zweig und:

Rekursion
verstehen

Binäre
Suche

Potenzieren
Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

- `power(2,3)` wählt else-Zweig und ruft auf:
 - `power(2,2)` wählt else-Zweig und ruft auf:
 - `power(2,1)` wählt else-Zweig und ruft auf:
 - `power(2,0)` wählt if-Zweig und:
 - ← `power(2,0)` gibt 1 zurück

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

- `power(2,3)` wählt else-Zweig und ruft auf:
 - `power(2,2)` wählt else-Zweig und ruft auf:
 - `power(2,1)` wählt else-Zweig und ruft auf:
 - `power(2,0)` wählt if-Zweig und:
 - ← `power(2,0)` gibt 1 zurück
 - ← `power(2,1)` gibt $(2 \times 1) = 2$ zurück

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

→ `power(2,3)` wählt else-Zweig und ruft auf:
 → `power(2,2)` wählt else-Zweig und ruft auf:
 → `power(2,1)` wählt else-Zweig und ruft auf:
 → `power(2,0)` wählt if-Zweig und:
 ← `power(2,0)` gibt 1 zurück
 ← `power(2,1)` gibt $(2 \times 1) = 2$ zurück
 ← `power(2,2)` gibt $(2 \times 2) = 4$ zurück

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

→ `power(2,3)` wählt else-Zweig und ruft auf:
 → `power(2,2)` wählt else-Zweig und ruft auf:
 → `power(2,1)` wählt else-Zweig und ruft auf:
 → `power(2,0)` wählt if-Zweig und:
 ← `power(2,0)` gibt 1 zurück
 ← `power(2,1)` gibt $(2 \times 1) = 2$ zurück
 ← `power(2,2)` gibt $(2 \times 2) = 4$ zurück
 ← `power(2,3)` gibt $(2 \times 4) = 8$ zurück

Rekursion
verstehen

Binäre
Suche

Potenzieren
Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Was passiert genau?

Aufrufsequenz

→ `power(2,3)` wählt else-Zweig und ruft auf:
 → `power(2,2)` wählt else-Zweig und ruft auf:
 → `power(2,1)` wählt else-Zweig und ruft auf:
 → `power(2,0)` wählt if-Zweig und:
 ← `power(2,0)` gibt 1 zurück
 ← `power(2,1)` gibt $(2 \times 1) = 2$ zurück
 ← `power(2,2)` gibt $(2 \times 2) = 4$ zurück
 ← `power(2,3)` gibt $(2 \times 4) = 8$ zurück

Rekursion
verstehen

Binäre
Suche

Potenzieren
Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Power ist **nicht** endrekursiv



```
def power (x : complex, n : int):  
    if n==0:  
        return 1  
    else:  
        return x * power (x, n-1)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Power ist **nicht** endrekursiv



```
def power (x : complex, n : int):  
    if n==0:  
        return 1  
    else:  
        return x * power (x, n-1)
```

- Aber wir könnten das Ergebnis auch *rückwärts* in einem **akkumulierenden Argument** berechnen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Power ist **nicht** endrekursiv



```
def power (x : complex, n : int):  
    if n==0:  
        return 1  
    else:  
        return x * power (x, n-1)
```

- Aber wir könnten das Ergebnis auch *rückwärts* in einem **akkumulierenden Argument** berechnen.

```
def power_acc (x : complex, n : int, acc : complex):  
    if n==0:  
        return acc  
    else:  
        return power_acc (x, n-1, acc * x)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Power ist **nicht** endrekursiv



```
def power (x : complex, n : int):  
    if n==0:  
        return 1  
    else:  
        return x * power (x, n-1)
```

- Aber wir könnten das Ergebnis auch *rückwärts* in einem **akkumulierenden Argument** berechnen.

```
def power_acc (x : complex, n : int, acc : complex):  
    if n==0:  
        return acc  
    else:  
        return power_acc (x, n-1, acc * x)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Power ist **nicht** endrekursiv



```
def power (x : complex, n : int):  
    if n==0:  
        return 1  
    else:  
        return x * power (x, n-1)
```

- Aber wir könnten das Ergebnis auch *rückwärts* in einem **akkumulierenden Argument** berechnen.

```
def power_acc (x : complex, n : int, acc : complex):  
    if n==0:  
        return acc  
    else:  
        return power_acc (x, n-1, acc * x)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



■ Schematische Transformation in Iteration

```
def power_it (x : complex, n : int, acc : complex = 1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Schematische Transformation in Iteration

```
def power_it (x : complex, n : int, acc : complex = 1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

- Startwert $acc = 1$ im Funktionskopf definiert.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



■ Schematische Transformation in Iteration

```
def power_it (x : complex, n : int, acc : complex = 1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

- Startwert $acc = 1$ im Funktionskopf definiert.
- Jeder Aufruf `power_it (x, n)` verwendet $acc=1$.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



■ Schematische Transformation in Iteration

```
def power_it (x : complex, n : int, acc : complex = 1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

- Startwert `acc = 1` im Funktionskopf definiert.
- Jeder Aufruf `power_it (x, n)` verwendet `acc=1`.
- Ein Aufruf (z.B.) `power_it (x, n, 42)` startet mit `acc=42`.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Rekursive
Definition

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Schneller Potenzieren

Rekursion
verstehen

Binäre
Suche

Potenzieren

**Schneller
Potenzieren**

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)`?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

■ `power (x, 0)?` **0**

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` 0
- `power (x, 1)?`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` **0**
- `power (x, 1)?` **1**

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` 0
- `power (x, 1)?` 1
- `power (x, 2)?`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` **0**
- `power (x, 1)?` **1**
- `power (x, 2)?` **2**

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` 0
- `power (x, 1)?` 1
- `power (x, 2)?` 2
- `power (x, n)?`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` 0
- `power (x, 1)?` 1
- `power (x, 2)?` 2
- `power (x, n)?` n

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def power_it (x : complex, n : int, acc : complex=1):  
    while n != 0:  
        n, acc = n-1, acc*x  
    else:  
        return acc
```

Wieviele Multiplikationen benötigen wir zur Berechnung von

- `power (x, 0)?` **0**
- `power (x, 1)?` **1**
- `power (x, 2)?` **2**
- `power (x, n)?` **n**

Mehr Multiplikationen als unbedingt notwendig!

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Alternative Definition von Power



```
power(x, 0)      == 1
power(x, 2*n)    == power(x*x, n)      # n > 0
power(x, 2*n+1) == x * power(x*x, n)  # n >= 0
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
power(x, 0)      == 1
power(x, 2*n)    == power(x*x, n)      # n > 0
power(x, 2*n+1) == x * power(x*x, n)  # n >= 0
```

- Alternative Aufteilung der natürlichen Zahlen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
power(x, 0)      == 1
power(x, 2*n)    == power(x*x, n)      # n > 0
power(x, 2*n+1) == x * power(x*x, n)  # n >= 0
```

- Alternative Aufteilung der natürlichen Zahlen.
- Jede natürliche Zahl ungleich 0 ist entweder gerade oder ungerade.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
power(x, 0)      == 1
power(x, 2*n)    == power(x*x, n)      # n > 0
power(x, 2*n+1) == x * power(x*x, n)  # n >= 0
```

- Alternative Aufteilung der natürlichen Zahlen.
- Jede natürliche Zahl ungleich 0 ist entweder gerade oder ungerade.
- In jedem Fall können wir die Berechnung von `power` entweder sofort abbrechen oder auf die `power` mit einem echt kleineren Exponenten `n` zurückführen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Schnelle Exponentiation



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_power (x*x, n//2)
    else: # n % 2 == 1
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4
- Multiplikationen für $n = 2^k$?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4
- Multiplikationen für $n = 2^k$? $k+2$

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4
- Multiplikationen für $n = 2^k$? $k+2$
- Multiplikationen für $n < 2^k$: höchstens $2k \approx 2 \log_2 n$.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_power (x*x, n//2)
    else: # n % 2 == 1
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4
- Multiplikationen für $n = 2^k$? $k+2$
- Multiplikationen für $n < 2^k$: höchstens $2k \approx 2 \log_2 n$.
- Also schneller: logarithmisch viele Multiplikationen!

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def fast_power (x : complex, n : int):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return fast_power (x*x, n//2)  
    else: # n % 2 == 1  
        return x * fast_power (x*x, n//2)
```

- Multiplikationen für $n = 1$? 2
- Multiplikationen für $n = 2$? 3
- Multiplikationen für $n = 4$? 4
- Multiplikationen für $n = 2^k$? $k+2$
- Multiplikationen für $n < 2^k$: höchstens $2k \approx 2 \log_2 n$.
- Also schneller: logarithmisch viele Multiplikationen!
- Berechnung von $n//2$ und $n\%2$ ist billig. Warum?

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Schnelle Exponentiation, iterativ?



```
def fast_power (x : complex, n : int):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_power (x*x, n//2)
    else: # n % 2 == 1
        return x * fast_power (x*x, n//2)
```

- Nicht endrekursiv!
- Aber es kann wieder ein akkumulierender Parameter eingeführt werden, der die äußeren Multiplikationen mit dem x durchführt.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Schnelle Exponentiation, endrekursiv!



```
def fast_power_acc (x : complex, n : int, acc : complex = 1):  
    if n == 0:  
        return acc  
    elif n % 2 == 0:  
        return fast_power_acc(x*x, n//2, acc)  
    else: # n % 2 == 1  
        return fast_power_acc(x*x, n//2, acc*x)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Schematische Transformation liefert

```
def fast_power_it (x : complex, n : int, acc : complex = 1):
    while n != 0:
        if n % 2 == 0:
            x, n, acc = (x*x, n//2, acc)
        else: # n % 2 == 1
            x, n, acc = (x*x, n//2, acc*x)
    else:
        return acc
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Sortieren

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Sortieren

- Eingabe
 - Liste `lst` : `list[T]`
 - (Ordnung \leq auf den Listenelementen vom Typ `T`)
- Ausgabe
 - aufsteigend sortierte Liste (gemäß \leq)
 - jedes Element muss in der Ausgabe genauso oft vorkommen wie in der Eingabe

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Sortieren

- Eingabe
 - Liste `lst : list [T]`
 - (Ordnung \leq auf den Listenelementen vom Typ `T`)
- Ausgabe
 - aufsteigend sortierte Liste (gemäß \leq)
 - jedes Element muss in der Ausgabe genauso oft vorkommen wie in der Eingabe

Sortieren durch Partitionieren

- Quicksort
- Erdacht von Sir C.A.R. Hoare um 1960
- Lange Zeit einer der schnellsten Sortieralgorithmen

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Vorgehensweise

- Falls lst leer ist, so ist die Ausgabe die leere Liste.
- Sonst wähle ein Element p aus lst .
- Sei lst_{lo} die Liste der Elemente aus lst , die $\leq p$ sind.
- Sei lst_{hi} die Liste der Elemente aus lst , die nicht $\leq p$ sind.
- Sortiere lst_{lo} und lst_{hi} mit Ergebnissen $sort_{lo}$ und $sort_{hi}$.
- Dann ist $sort_{lo} + [p] + sort_{hi}$ eine sortierte Version von lst .

Rekursion
verstehen

Binäre
Suche

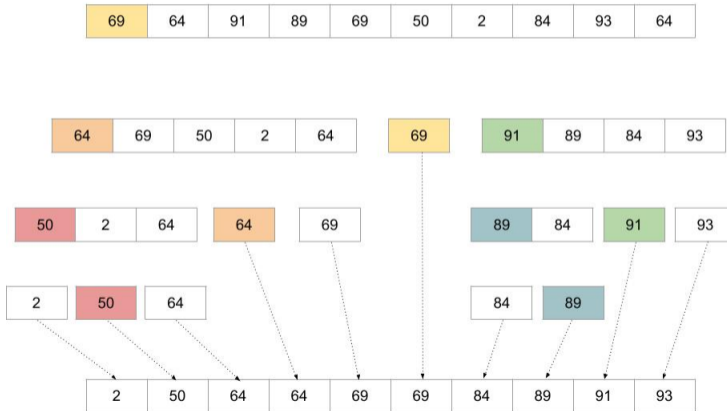
Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Quicksort Beispiel



Rekursion verstehen

Binäre Suche

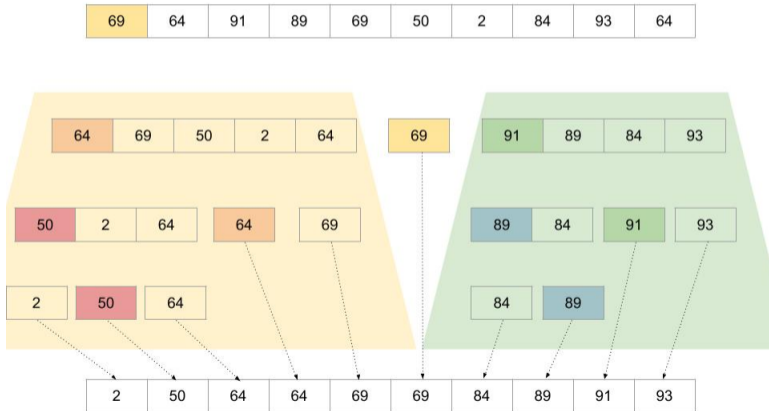
Potenzieren

Schneller Potenzieren

Sortieren

Lindenmayer Systeme

Quicksort Beispiel



Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def quicksort (lst : list[int]):  
    if len (lst) <= 1:  
        return lst  
    else:  
        p, lst_lo, lst_hi = partition (lst)  
        return (quicksort (lst_lo)  
                + [p]  
                + quicksort (lst_hi))
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def quicksort (lst : list[int]):  
    if len (lst) <= 1:  
        return lst  
    else:  
        p, lst_lo, lst_hi = partition (lst)  
        return (quicksort (lst_lo)  
                + [p]  
                + quicksort (lst_hi))
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Wunschdenken

- Wir nehmen an, dass `partition (lst)` für `len (lst) >= 1` ein 3-Tupel liefert, wobei
 - `p` ist ein Element von `lst`
 - `lst_lo` enthält die Elemente `<= p`
 - `lst_hi` enthält die Elemente `> p`



```
def partition (lst):  
    """ assume len (lst) >= 1 """  
    p, rest = lst[0], lst[1:]  
    lst_lo = []  
    lst_hi = []  
    for x in rest:  
        if x <= p:  
            lst_lo = lst_lo + [x]  
        else:  
            lst_hi = lst_hi + [x]  
    return p, lst_lo, lst_hi
```

- Codegerüst für Listenverarbeitung
- Zwei Akkumulatoren `lst_lo` und `lst_hi`

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Der rekursive Algorithmus ist die einfachste Beschreibung von Quicksort.
- Eine iterative Implementierung ist möglich.
- Diese ist aber deutlich schwieriger zu verstehen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Lindenmayer Systeme

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Wikipedia

Bei den Lindenmayer- oder L-Systemen handelt es sich um einen mathematischen Formalismus, der 1968 von dem ungarischen theoretischen Biologen Aristid Lindenmayer als Grundlage einer axiomatischen Theorie biologischer Entwicklung vorgeschlagen wurde. In jüngerer Zeit fanden L-Systeme Anwendung in der Computergrafik bei der Erzeugung von Fraktalen und in der realitätsnahen Modellierung von Pflanzen.

[Rekursion verstehen](#)

[Binäre Suche](#)

[Potenzieren](#)

[Schneller Potenzieren](#)

[Sortieren](#)

[Lindenmayer Systeme](#)



Wikipedia

Bei den Lindenmayer- oder L-Systemen handelt es sich um einen mathematischen Formalismus, der 1968 von dem ungarischen theoretischen Biologen Aristid Lindenmayer als Grundlage einer axiomatischen Theorie biologischer Entwicklung vorgeschlagen wurde. In jüngerer Zeit fanden L-Systeme **Anwendung in der Computergrafik bei der Erzeugung von Fraktalen** und in der realitätsnahen Modellierung von Pflanzen.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Definition

Ein 0L-System ist ein Tupel $G = (V, \omega, P)$, wobei

- V eine Menge von Symbolen (Alphabet),
- $\omega \in V^*$ ein String von Symbolen und
- $P \subseteq V \times V^*$ eine Menge von **Produktionen** ist, wobei zu jedem $A \in V$ mindestens eine Produktion $(A, w) \in P$ existieren muss.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Definition

Ein 0L-System ist ein Tupel $G = (V, \omega, P)$, wobei

- V eine Menge von Symbolen (Alphabet),
- $\omega \in V^*$ ein String von Symbolen und
- $P \subseteq V \times V^*$ eine Menge von **Produktionen** ist, wobei zu jedem $A \in V$ mindestens eine Produktion $(A, w) \in P$ existieren muss.

Beispiel (Lindenmayer): 0L-System für Algenwachstum

- $V = \{A, B\}$
- $\omega = A$
- $P = \{A \rightarrow BA, B \rightarrow A\}$

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Definition

Sei $G = (V, \omega, P)$ ein 0L-System.

Sei $A_1A_2 \dots A_n$ ein String über Symbolen aus V (also $A_i \in V$).

Ein Schritt von G ersetzt **jedes** Symbol durch eine zugehörige rechte Produktionsseite:

$$A_1A_2 \dots A_n \Rightarrow w_1w_2 \dots w_n$$

wobei $(A_i, w_i) \in P$, für $1 \leq i \leq n$.

Die **Sprache von G** besteht aus allen Strings, die aus ω durch endlich viele \Rightarrow -Schritte erzeugt werden können.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

1 A

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

1 A

2 BA

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 A
- 2 BA
- 3 ABA

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 A
- 2 BA
- 3 ABA
- 4 $BAABA$

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 *A*
- 2 *BA*
- 3 *ABA*
- 4 *BAABA*
- 5 *ABABAABA*

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 *A*
- 2 *BA*
- 3 *ABA*
- 4 *BAABA*
- 5 *ABABAABA*
- 6 *BAABAABABAABA*

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 *A*
- 2 *BA*
- 3 *ABA*
- 4 *BAABA*
- 5 *ABABAABA*
- 6 *BAABAABABAABA*
- 7 *ABABAABABAABAABABAABA*

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Beispiel Algenwachstum



$$P = \{A \rightarrow BA, B \rightarrow A\}$$

- 1 *A*
- 2 *BA*
- 3 *ABA*
- 4 *BAABA*
- 5 *ABABAABA*
- 6 *BAABAABABAABA*
- 7 *ABABAABABAABAABABAABA*
- 8 USW

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Die Kochkurve ist ein Fraktal.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Die **Kochkurve** ist ein **Fraktal**.
- D.h. eine selbstähnliche Kurve mit rekursiver Beschreibung und weiteren spannenden Eigenschaften.

Rekursion
verstehen

Binäre
Suche

Potenzieren

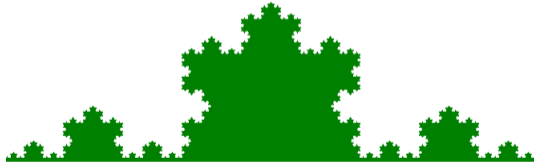
Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Die **Kochkurve** ist ein **Fraktal**.
- D.h. eine selbstähnliche Kurve mit rekursiver Beschreibung und weiteren spannenden Eigenschaften.



<https://commons.wikimedia.org/wiki/File:Kochkurve.png>

Rekursion
verstehen

Binäre
Suche

Potenzieren

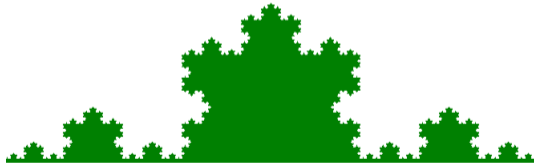
Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- Die **Kochkurve** ist ein **Fraktal**.
- D.h. eine selbstähnliche Kurve mit rekursiver Beschreibung und weiteren spannenden Eigenschaften.



<https://commons.wikimedia.org/wiki/File:Kochkurve.png>

- Sie kann durch ein 0L-System beschrieben werden.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



0L-System für die Kochkurve

- $V = \{F, +, -\}$
- $\omega = F$
- $P = \{F \mapsto F + F - F + F\}$ sowie $+ \mapsto +$ und $- \mapsto -$

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



0L-System für die Kochkurve

- $V = \{F, +, -\}$
- $\omega = F$
- $P = \{F \mapsto F + F - F + F\}$ sowie $+ \mapsto +$ und $- \mapsto -$

Interpretation

- F Strecke vorwärts zeichnen
- $+$ um 60° nach links abbiegen
- $-$ um 120° nach rechts abbiegen

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Idee der “Schildkrötengrafik”

Eine Schildkröte sitzt auf einer Zeichenfläche. Sie kann eine bestimmte Strecke geradeaus gehen oder abbiegen. Wenn ihr Hintern dabei über den Boden schleift, hinterläßt sie einen geraden Strich.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Idee der “Schildkrötengrafik”

Eine Schildkröte sitzt auf einer Zeichenfläche. Sie kann eine bestimmte Strecke geradeaus gehen oder abbiegen. Wenn ihr Hintern dabei über den Boden schleift, hinterläßt sie einen geraden Strich.

Befehle an die Schildkröte

```
from turtle import *
pencolor('black') #use the force
pendown()          #let it all hang out
forward(100)
left(120)
forward(100)
left(120)
forward(100)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

Schildkröten-Interpretation



Die Operationen

- F forward (size)
- + left (60)
- - right (120)

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Die Operationen

- F forward (size)
- + left (60)
- - right (120)

Die Produktion $F \mapsto F + F - F + F$

```
def koch(size:float, n:int):  
    #...  
    koch(size/3, n-1) #F  
    left(60)          #+  
    koch(size/3, n-1) #F  
    right(120)        #-  
    koch(size/3, n-1) #F  
    left(60)          #+  
    koch(size/3, n-1) #F
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def koch (size:float, n:int):  
    if n == 0:  
        forward(size)  
    else:  
        koch (size/3, n-1)  
        left(60)  
        koch (size/3, n-1)  
        right(120)  
        koch (size/3, n-1)  
        left(60)  
        koch (size/3, n-1)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



0L-System für fraktale Binärbäume

- $V = \{0, 1, [,]\}$
- $\omega = 0$
- $P = \{1 \mapsto 11, 0 \mapsto 1[0]0\}$

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



0L-System für fraktale Binärbäume

- $V = \{0, 1, [,]\}$
- $\omega = 0$
- $P = \{1 \mapsto 11, 0 \mapsto 1[0]0\}$

Interpretation

- 0 Strecke vorwärts zeichnen mit Blatt am Ende
- 1 Strecke vorwärts zeichnen
- [Position und Richtung merken und um 45° nach links abbiegen
-] Position und Richtung von zugehöriger öffnender Klammer wiederherstellen und um 45° nach rechts abbiegen

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



```
def btree_1 (size:float , n:int):  
    if n == 0:  
        forward (size)  
    else:  
        n = n - 1  
        btree_1 (size/3, n)  
        btree_1 (size/3, n)
```

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme

- $n=0$: letzte Generation erreicht
- Faktor $1/3$ willkürlich gewählt

Turtle-Graphics Implementierung Teil 0



```
def btree_0 (size:float, n:int):
    if n == 0:
        forward(size)           # line segment
        dot (2, 'green')       # draw leaf
    else:
        n = n - 1
        btree_1 (size/3, n)    # "1"
        pos = position()      # "["
        ang = heading()
        left(45)
        btree_0 (size/3, n)    # "0"
        penup()               # "]"
        setposition (pos)
        setheading (ang)
        pendown()
        right (45)
        btree_0 (size/3, n)    # "0"
```

Rekursion
verstehen

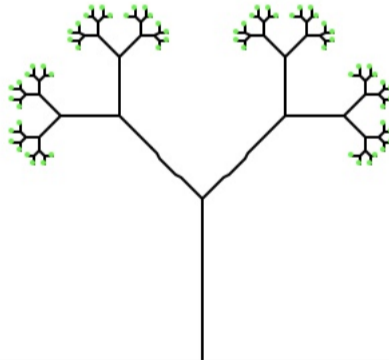
Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- **Induktion** ist eine Definitionstechnik aus der Mathematik.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- **Induktion** ist eine Definitionstechnik aus der Mathematik.
- Induktiv definierte Funktionen können meist kurz und elegant rekursiv implementiert werden.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- **Induktion** ist eine Definitionstechnik aus der Mathematik.
- Induktiv definierte Funktionen können meist kurz und elegant rekursiv implementiert werden.
- Funktionen auf induktiv definierten Daten (d.h. baumartigen Strukturen) sind generell rekursiv.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- **Induktion** ist eine Definitionstechnik aus der Mathematik.
- Induktiv definierte Funktionen können meist kurz und elegant rekursiv implementiert werden.
- Funktionen auf induktiv definierten Daten (d.h. baumartigen Strukturen) sind generell rekursiv.
- In Python ist Rekursion ist nicht immer die **effizienteste** Implementierung einer Funktion!

[Rekursion verstehen](#)

[Binäre Suche](#)

[Potenzieren](#)

[Schneller Potenzieren](#)

[Sortieren](#)

[Lindenmayer Systeme](#)



- **Induktion** ist eine Definitionstechnik aus der Mathematik.
- Induktiv definierte Funktionen können meist kurz und elegant rekursiv implementiert werden.
- Funktionen auf induktiv definierten Daten (d.h. baumartigen Strukturen) sind generell rekursiv.
- In Python ist Rekursion ist nicht immer die **effizienteste** Implementierung einer Funktion!
- **Endrekursion** kann schematisch in effiziente **Iteration** umgewandelt werden.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme



- **Induktion** ist eine Definitionstechnik aus der Mathematik.
- Induktiv definierte Funktionen können meist kurz und elegant rekursiv implementiert werden.
- Funktionen auf induktiv definierten Daten (d.h. baumartigen Strukturen) sind generell rekursiv.
- In Python ist Rekursion ist nicht immer die **effizienteste** Implementierung einer Funktion!
- **Endrekursion** kann schematisch in effiziente **Iteration** umgewandelt werden.
- **Allgemeine Rekursion** ist komplizierter umzuwandeln.

Rekursion
verstehen

Binäre
Suche

Potenzieren

Schneller
Potenzieren

Sortieren

Lindenmayer
Systeme