Informatik

Rekursion

Eine Funktion darf sich im Rumpf selbst wieder aufrufen. Man nennt das einen rekursiven Aufruf.

Eine Funktion darf sich im Rumpf selbst wieder aufrufen. Man nennt das einen rekursiven Aufruf. Jede Rekursion braucht eine Bremse, damit es nicht zu unendlich vielen Aufrufen kommt.

Eine Funktion darf sich im Rumpf selbst wieder aufrufen. Man nennt das einen rekursiven Aufruf. Jede Rekursion braucht eine Bremse, damit es nicht zu unendlich vielen Aufrufen kommt.

Iterative Definition der Fakultät:

Eine Funktion darf sich im Rumpf selbst wieder aufrufen. Man nennt das einen rekursiven Aufruf. Jede Rekursion braucht eine Bremse, damit es nicht zu unendlich vielen Aufrufen kommt.

Iterative Definition der Fakultät:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \cdot 2 \cdot 3 \cdot ... \cdot n & n > 0 \end{cases}$$

Eine Funktion darf sich im Rumpf selbst wieder aufrufen. Man nennt das einen rekursiven Aufruf. Jede Rekursion braucht eine Bremse, damit es nicht zu unendlich vielen Aufrufen kommt.

Iterative Definition der Fakultät:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n & n > 0 \end{cases}$$

Rekursive Definition der Fakultät:

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

```python
# iterative Implementation der Fakultät
def fakultaet(n):
    if n == 0: return 1
    temp = 1
    for i in range(n):
        temp = temp * (i+1)
    return temp

# rekursive Implementation der Fakultät
def fakultaet(n):
    if n == 0: return 1
    return n * fakultaet(n-1)

#Aufruf:
print(fakultaet(5))
```

Iterative Definition der Zweierpotenz:

$$2^n = \begin{cases} 1 & n = 0 \\ 2 \cdot 2 \cdot 2 \cdot ... \cdot 2 \text{ (n-mal)} & n > 0 \end{cases}$$

Iterative Definition der Zweierpotenz:

$$2^n = \begin{cases} 1 & n = 0 \\ 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 \text{ (n-mal)} & n > 0 \end{cases}$$

Rekursive Definition der Zweierpotenz:

$$2^n = \begin{cases} 1, & n = 0 \\ 2^{n-1} \cdot 2 & n > 0 \end{cases}$$

Eine rekursive Methode darf die Lösung für *kleinere Problemgrößen* in ihrem Rumpf benutzen.

Eine rekursive Methode darf die Lösung für *kleinere Problemgrößen* in ihrem Rumpf benutzen.

Die Methode dreheUm dreht die Reihenfolge der Zeichen eines Strings um.

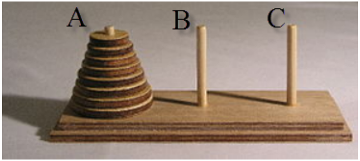
Eine rekursive Methode darf die Lösung für *kleinere Problemgrößen* in ihrem Rumpf benutzen.

Die Methode dreheUm dreht die Reihenfolge der Zeichen eines Strings um.

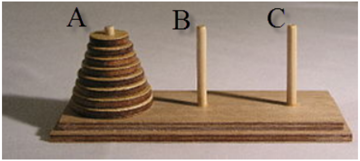
```
def dreheUm(s):
    if len(s) == 0:
        return ''
    letztes = s[-1]
    bisVorletztes = s[:-1]
    return letztes + dreheUm(bisVorletztes)

Aufruf:
print(dreheUm("Python"))
```

Türme von Hanoi



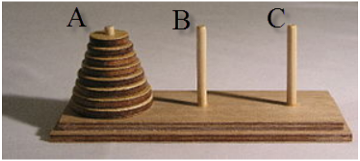
Türme von Hanoi



Der Turm soll von Position A nach Position C. Die Scheiben dürfen nur einzeln bewegt werden und nie darf eine größere auf eine kleinere Scheibe gelegt werden. Eine Zwischenposition B steht zur Verfügung.

https://www.youtube.com/watch?v=w9LgLiW9YHU

Türme von Hanoi



Der Turm soll von Position A nach Position C. Die Scheiben dürfen nur einzeln bewegt werden und nie darf eine größere auf eine kleinere Scheibe gelegt werden. Eine Zwischenposition B steht zur Verfügung.

https://www.youtube.com/watch?v=w9LgLiW9YHU

Rekursive Idee: verlagere den Turm ohne die unterste Scheibe rekursiv nach B (kleinere Problemgrößen dürfen wir als gelöst annehmen). Verlege dann die unterste Scheibe von A nach C und verlagere dann den kleineren Turm rekursiv von B nach C.

```
def hanoi(n, start, ziel, zwischen):
    if n == 0: return
    hanoi(n-1,start,zwischen,ziel)
    print("Scheibe",n," von ",start," nach ",ziel)
    hanoi(n-1,zwischen,ziel,start)

Aufruf:
hanoi(5,"A","C","B")
```

Scheiben	Verlegeoperationen
1	1

Scheiben	Verlegeoperationen
1	1
2	3
3	7
4	15
5	31
6	63
n	

Scheiben	Verlegeoperationen
1	1
2	3
3	7
4	15
5	31
6	63
n	$2^n - 1$

Scheiben	Verlegeoperationen
1	1
2	3
3	7
4	15
5	31
6	63
n	$2^n - 1$

1 Verlegeoperation $=$
1 Sekunde

Anzahl Scheiben	Benötigte Zeit
5	31 Sekunden
10	17,1 Minuten
20	12 Tage
30	34 Jahre
40	348 Jahrhunderte
60	36,6 Milliarden Jahre
64	585 Milliarden Jahre

Fibonacci-Zahlen

n	1	2	3	4	5	6	7	8	...
fib(n)	1	1	2	3	5	8	13	21	...

Fibonacci-Zahlen

n	1	2	3	4	5	6	7	8	...
fib(n)	1	1	2	3	5	8	13	21	...

Rekursive Definition der Fibonacci-Zahlen:

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

Fibonacci-Zahlen

n	1	2	3	4	5	6	7	8	...
fib(n)	1	1	2	3	5	8	13	21	...

Rekursive Definition der Fibonacci-Zahlen:

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

```python
# Fibonacci-Zahlen rekursiv
def fib(n):
    if n <= 2: return 1
    return fib(n-2) + fib(n-1)
```

Fibonacci-Zahlen

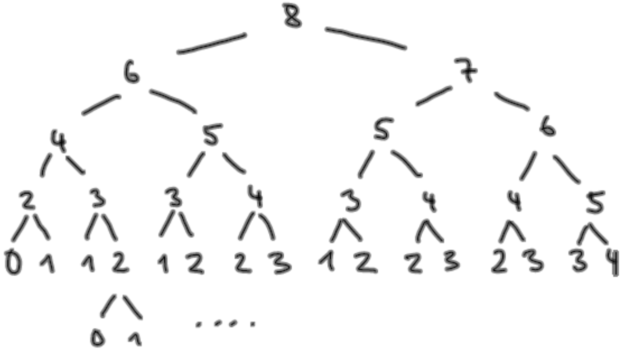
n	1	2	3	4	5	6	7	8	...
fib(n)	1	1	2	3	5	8	13	21	...

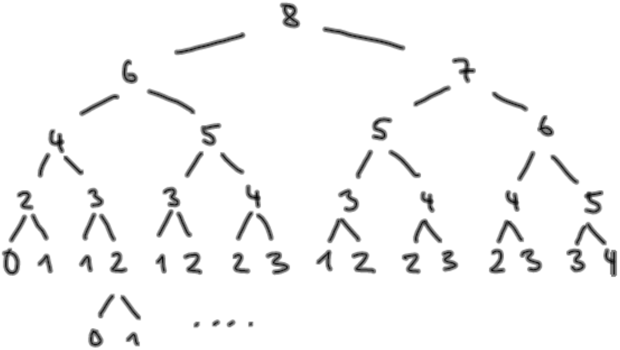
Rekursive Definition der Fibonacci-Zahlen:

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

```python
# Fibonacci-Zahlen rekursiv
def fib(n):
    if n <= 2: return 1
    return fib(n-2) + fib(n-1)
```

Schon fib(40) dauert ziemlich lange.


8
6
7
4
5
5
6
2
3
3
4
3
4
4
5
0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4
0 1
....



Rekursive Implementation ist sehr unwirtschaftlich, da schnell anwachsende Zahl von fib-Aufrufen.

Besser: *dynamische Programmierung* = Tabellen mit Teillösungen aufbauen.

Besser: *dynamische Programmierung* = Tabellen mit Teillösungen aufbauen.

```
def fib(n):
    if n <= 2: return 1
    a,b = 1,1
    for i in range(3,n+1):
        c = a+b
        a,b = b,c
    return c
```