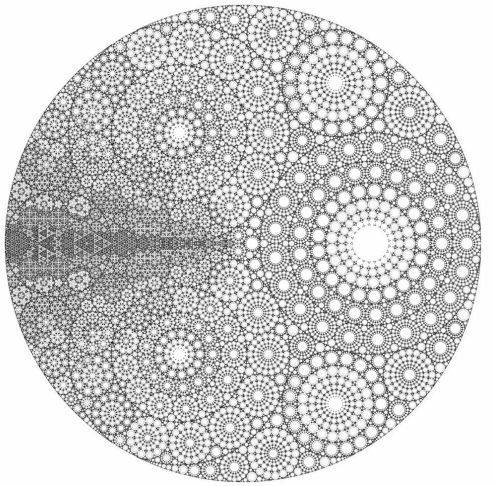
Rekursion +
Komplexität




Clean Code

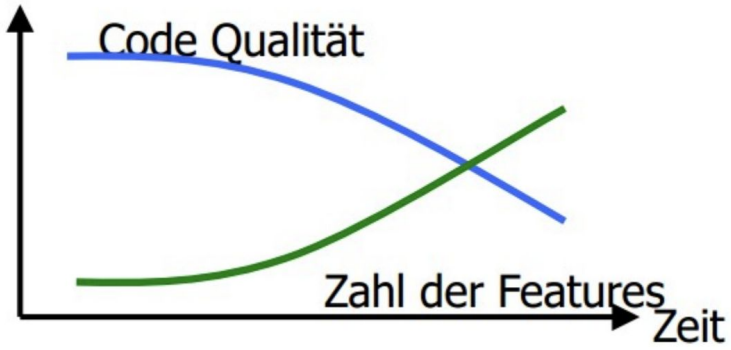
Clean Code

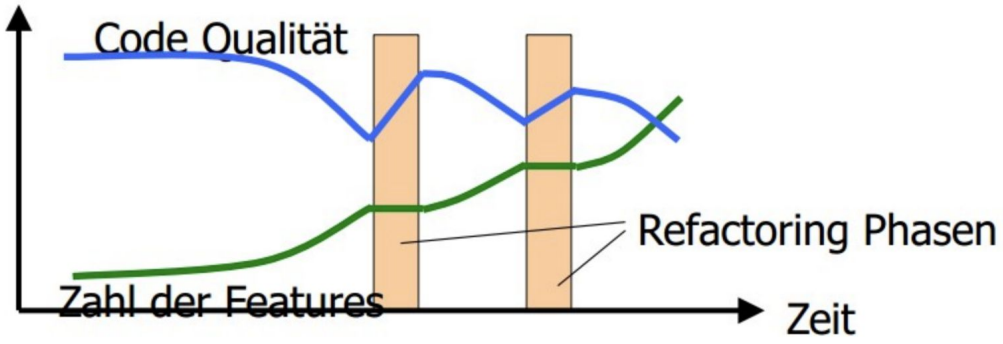

Me
My code works fine
Now time for
Some clean up
Also me
My code works fine
Push it now
Clean it later

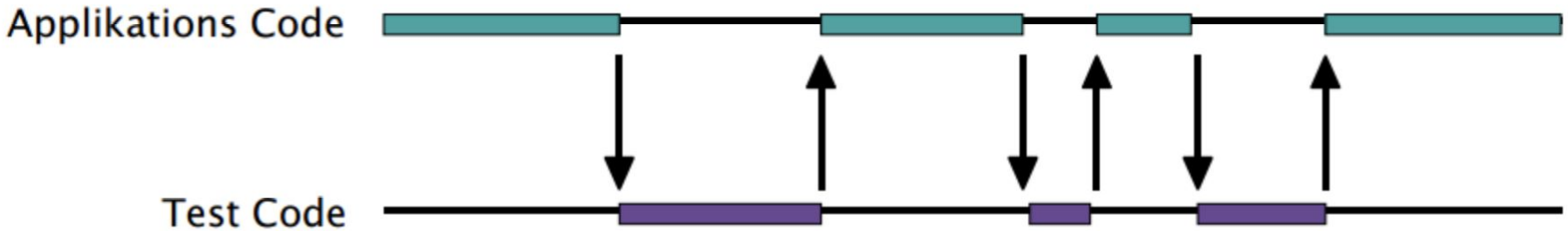
unser Mantra: „Code a little, test a little, refactor a litte!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.







Re•fac•to•ring

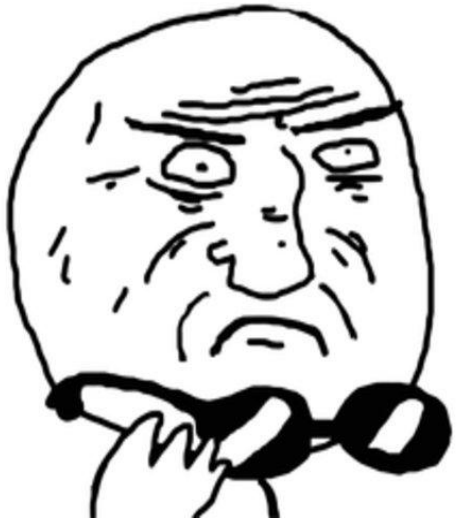
(noun)

„a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior„

-refactoring.com

was bedeutet das genau?

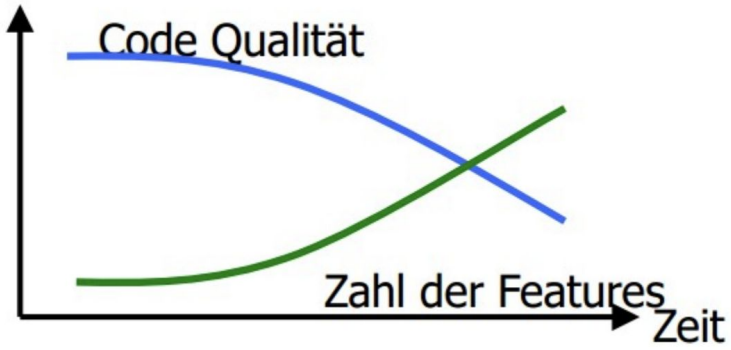
- man versucht den Code nicht direkt perfekt zu schreiben
- der Weg zum Clean Code ist durch Refactoring erreicht
- aber der Code muss immer richtig funktionieren
  - d.h. ohne Bugs ;)
  - testing is wichtig
- ein unendlicher Kreis

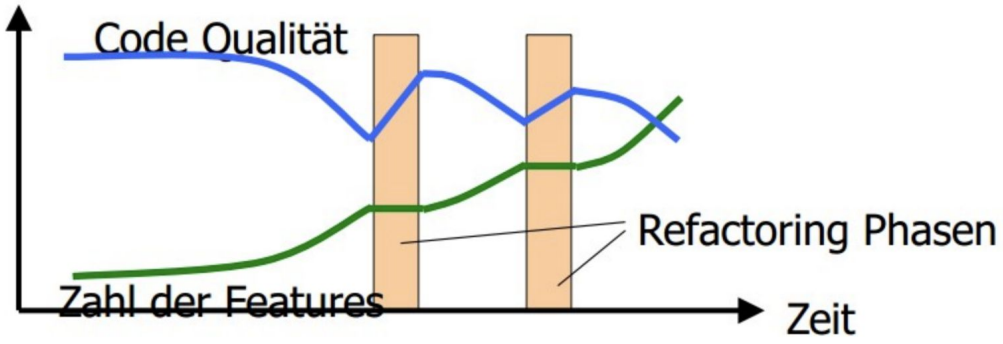


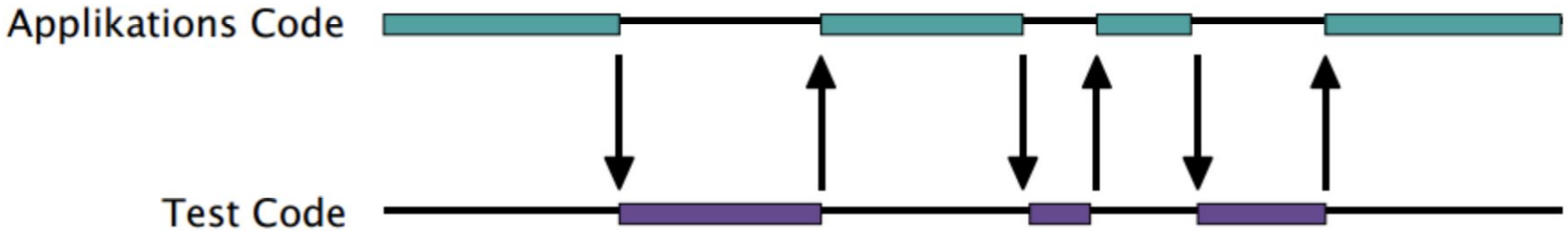
unser Mantra: „Code a little, test a little, refactor a litte!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.







4 Prinzipien

- man erstellt Code, den **alle** verstehen können
- die Personen, die den Code reviewen oder verwenden, sollten keine Annahme machen
- manchmal muss man Code löschen (nach dem Motto: *when less is more*)
- Es gibt kein perfekter Code
  - Es gibt immer Raum für Verbesserungen

Ziele des Refactoring

- Lesbarkeit des Codes erhöhen
  - Refactoring kann parallel zu einem Code Review erfolgen
- Design verbessern (sogenannte „Bad Smells“ beseitigen)
- Code so vorbereiten, dass neue Features implementiert werden können

not only good news...

- Refactoring ist riskant
  - Risiko minimieren durch gute Unit Test Abdeckung
  - Testing ist eine Voraussetzung

- Immer in kleinen Schritten:
  - Ein Refactoring Schritt
  - Testen
  - Nächster Refactoring Schritt
  - Testen

- Häufiger Wechsel zwischen Implementation eines neuen Features und Refactoring

Design verbessern/The return: Bad Smell

- ein Anti-Pattern: schlechte Codefragmente, die potenzielle Probleme anzeigen

- Duplizierter Code
  - Hoher Wartungsaufwand da Änderungen überall nachgeführt werden müssen

- Lange Methode / Grosse Klasse
  - Schwierig zu verstehen/hat viele Verantwortungen
  - Schlechte Wiederverwendbarkeit
  - Folge von Code Duplikationen
  - unser Controller: seminar 7-8

The return: Bad Smell

- Lange Parameter Liste
  - Schwierig zu lesen
  - Oft schlechte Wiederverwendbarkeit
  - Gefahr des Vertauschens bei Parametern des gleichen Typs

- Switch Statements bzw. if-else-if Ketten
  - Möglicherweise unflexibel für Erweiterungen
  - Gleichartige Switch Statements: Code Duplikationen
  - main-Methode Seminar 9: wir haben eine neue Methode erstellt

was können wir tun?

Methode extrahieren

- Ein Codefragment kann zusammengefasst werden
- Setze das Fragment in eine Methode, deren Name und Zweck bezeichnet

Motivation:

- Verbesserung der Lesbarkeit
- Codeduplikation: Verschiedene Codefragmente tun (fast) dasselbe.

Methode extrahieren

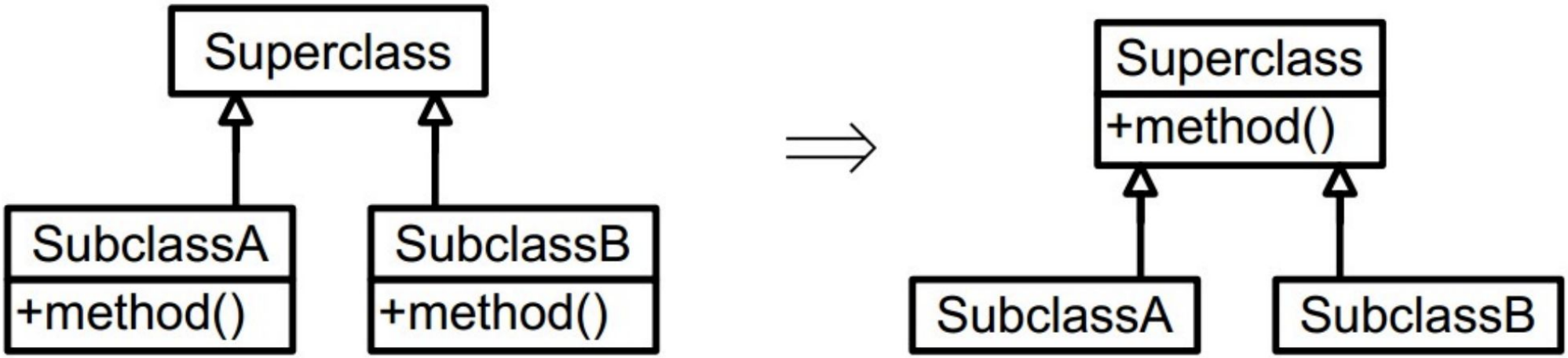
```
void foo()
{
    // berechne Kosten
    kosten = a * b + c;
    kosten -= discount;
}
```

```
void foo()
{
    berechneKosten();
}
```

```
void berechneKosten()
{
    kosten = a * b + c;
    kosten -= discount;
}
```

Methode hochziehen

- Es gibt Methoden mit identischen Ergebnissen in den Unterklassen
- Verschiebe die Methoden in die Oberklasse



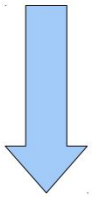
Methode umbenennen

- um die Lesbarkeit zu verbessern
- Der Name einer Methode macht die Absicht nicht deutlich
- Ändere den Name der Methode

Beschreibende Variable

- Es gibt einen komplizierten Ausdruck
- Setze den Ausdruck (oder Teile) in lokale Variable, deren Name den Zweck erklärt.

```
if  platform.toUpperCase().indexOf("MAC") > -1 and
browser.toUpperCase().indexOf("IE") > -1 and wasInitialized() and resize > 0: #stuff
```



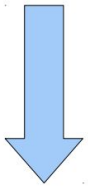
```
isMacOs = platform.toUpperCase().indexOf("MAC") > -1
isIEBrowser = browser.toUpperCase().indexOf("IE") > -1
wasResized = resize > 0;
if isMacOs and isIEBrowser and wasInitialized() and wasResized: #stuff
```

Replace Temp with Query

- typische Situation: eine temporäre Variable speichert das Ergebnis eines Ausdrucks
- man kann den Ausdruck in eine Abfrage-Methode stellen
- und die temporäre Variable durch Aufrufe der Methode ersetzen
- Vorteil: die neue Methode kann in anderen Methoden benutzt werden

Replace Temp with Query

```
basePrice = quantity * itemPrice;
if basePrice > 1000.00:  return basePrice * 0.95
else: return basePrice * 0.98
```



```
if basePrice() > 1000.00:
    return basePrice() * 0.95
else:
    return basePrice() * 0.98

def basePrice():
    return quantity * itemPrice
```

Schritte

- step0: Tests
- step1: alles zum Laufen zu bringen
- step2: Code Aufräumen (refactoring)
- step3: Code Reuse
- irgendwo mittendrin werden wir anfangen, *idiomatic python* zu schreiben


Wenn du einer der Non-Konformisten sein willst, dann musst du dich nur genau so anziehen wie wir und die gleiche Musik hören

Rekursion

- Neue Denkweise
- Wikipedia: “Als Rekursion bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst.”
- Rekursion ist eine Form der Wiederholung
- Rekursion ermöglicht
  - elegante Algorithmen
  - Komplexitätsanalyse

Rekursion

In der Mathematik ist es oft einfacher, eine Funktion rekursiv zu definieren

$$\text{ggt}(a, b) = \begin{cases} a & \text{falls } b = 0 \\ \text{ggt}(b, a \bmod b) & \text{sonst} \end{cases}$$

- **ggt** dient zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen
- Absicht deutlich aus eigener Definition

```
ggt(33, 12) = ggt(12, 33 mod 12) = ggt(12, 9)
            = ggt(9, 12 mod 9)   = ggt(9, 3)
            = ggt(3, 9 mod 3)    = ggt(3, 0)
            = 3
```

Definition

- eine Funktion nennt man **rekursiv**, wenn sie **sich selbst aufruft**
- rekursive Funktionen bestehen immer aus den folgenden
- Bestandteilen:
  - mindestens ein **Basisfall**, in dem die Rekursion abbricht und das Ergebnis entsteht
  - mindestens ein **rekursiver Fall**, in dem die Funktion sich selbst mit veränderten Argumenten aufruft

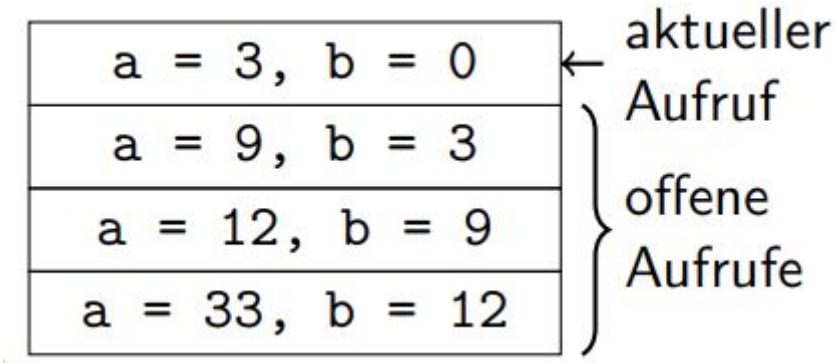
```
def ggt(a, b): //ggt = größter gemeinsamer Teiler
    if (b == 0)
        return a //Basisfall

    return ggt(b, a % b) // rekursiver Fall
```

Rekursion und der Stack

- jeder Aufruf legt einen neuen Stack Frame mit seinen Argumenten oben auf den Stack
- die aktuellen Argumentwerte stehen im obersten Frame
- bei einem return wird der Stack Frame geschlossen

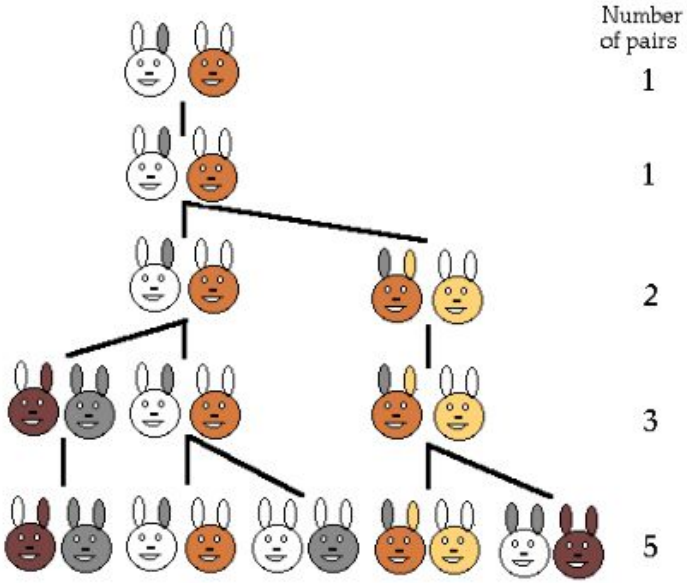
$$\begin{aligned} ggt(33, 12) &= ggt(12, 9) \\ &= ggt(9, 3) \\ &= ggt(3, 0) \\ &= 3 \end{aligned}$$



Rekursion

Beginn der Fibonacci-Folge

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...



- zu Beginn eines Jahres gibt es genau ein Paar neugeborener Kaninchen
- dieses Paar wirft nach 2 Monaten ein neues Kaninchenpaar
- und dann monatlich jeweils ein weiteres Paar
- jedes neugeborene Paar vermehrt sich auf die gleiche Weise

Fibonacci-Zahlen

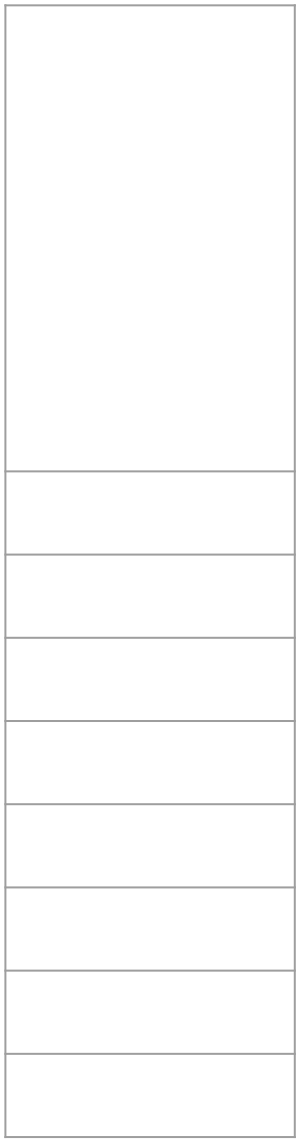
- Rekursive Definition der Fibonacci-Folge:
  a. `fib(n) = 0`, falls `n = 0`
  b. `fib(n) = 1`, falls `n = 1`
  c. `fib(n) = fib(n - 1)+fib(n - 2)`, falls `n ≥ 2`
- Die Rekursion in (c) stoppt, wenn `n = 0` oder `n = 1`
- Abbruchbedingung?

```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

Fibonacci-Zahlen

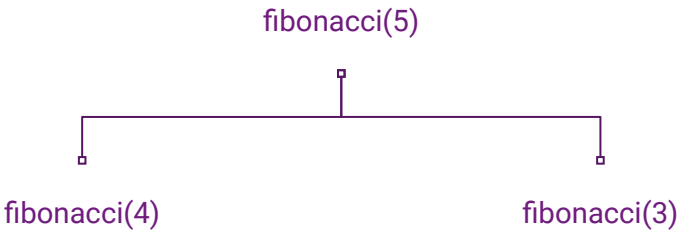
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

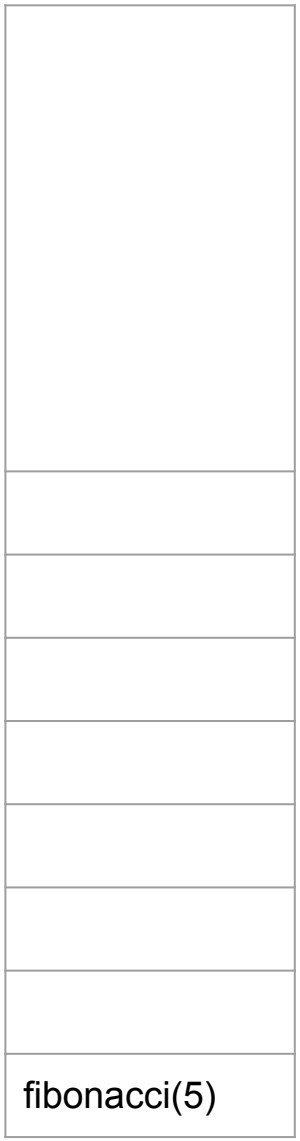
fibonacci(5)



Fibonacci-Zahlen

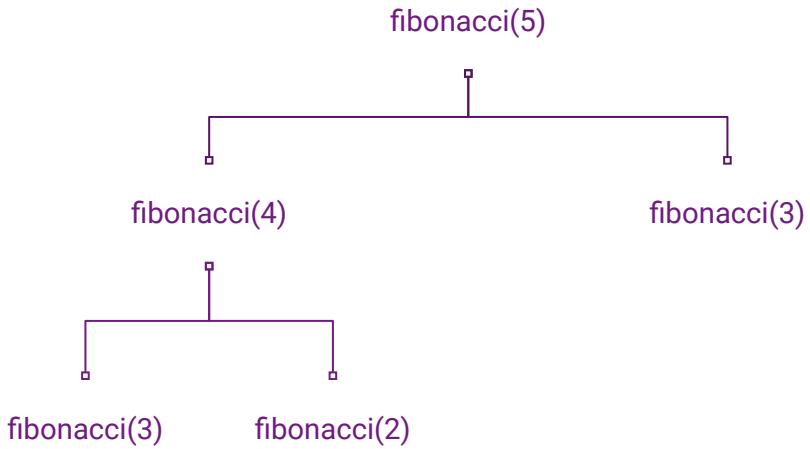
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

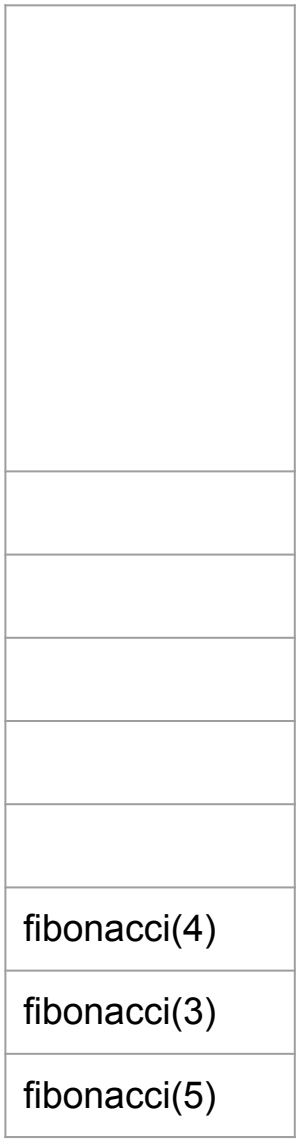




Fibonacci-Zahlen

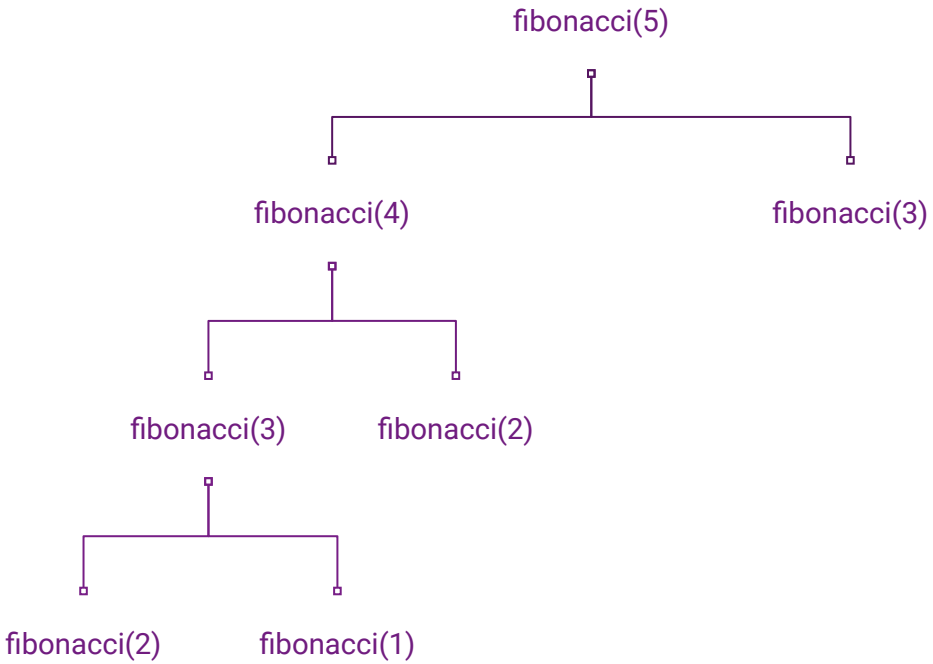
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

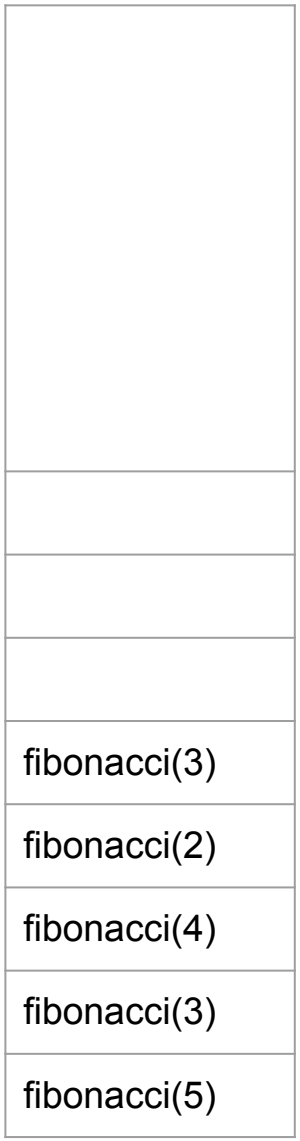




Fibonacci-Zahlen

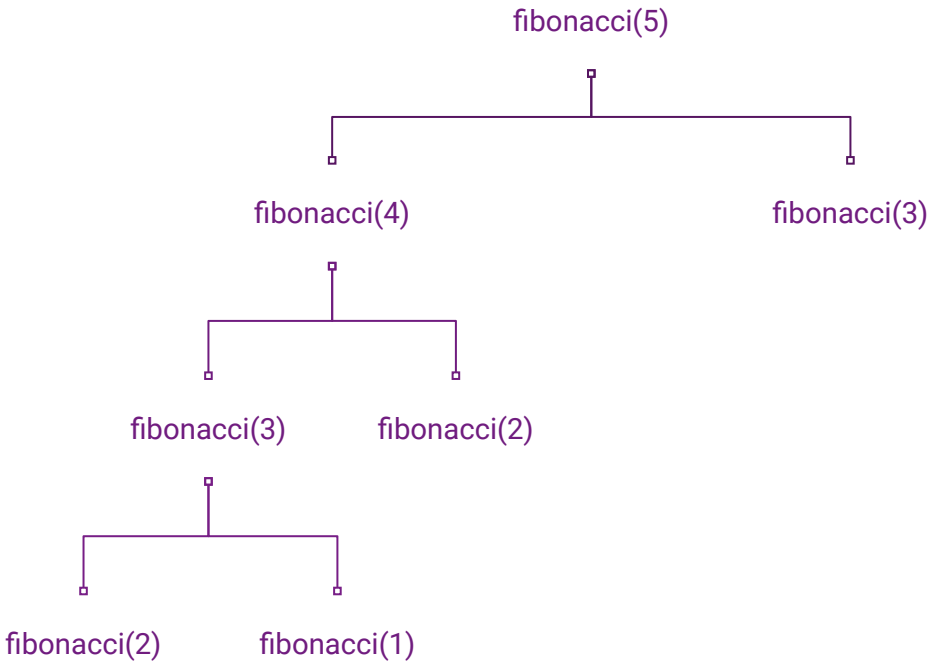
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

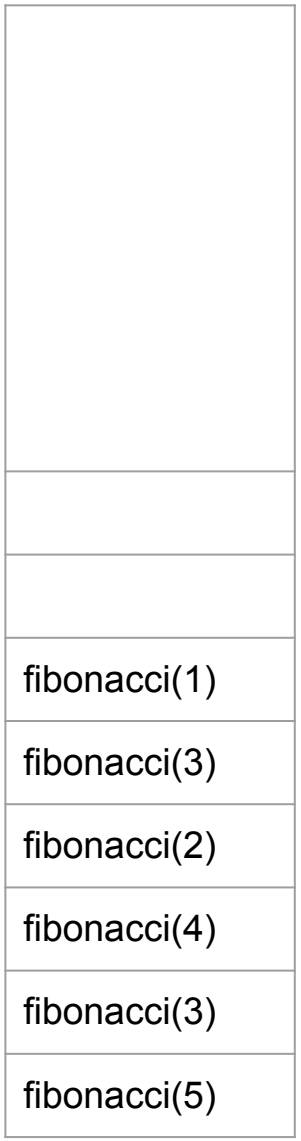




Fibonacci-Zahlen

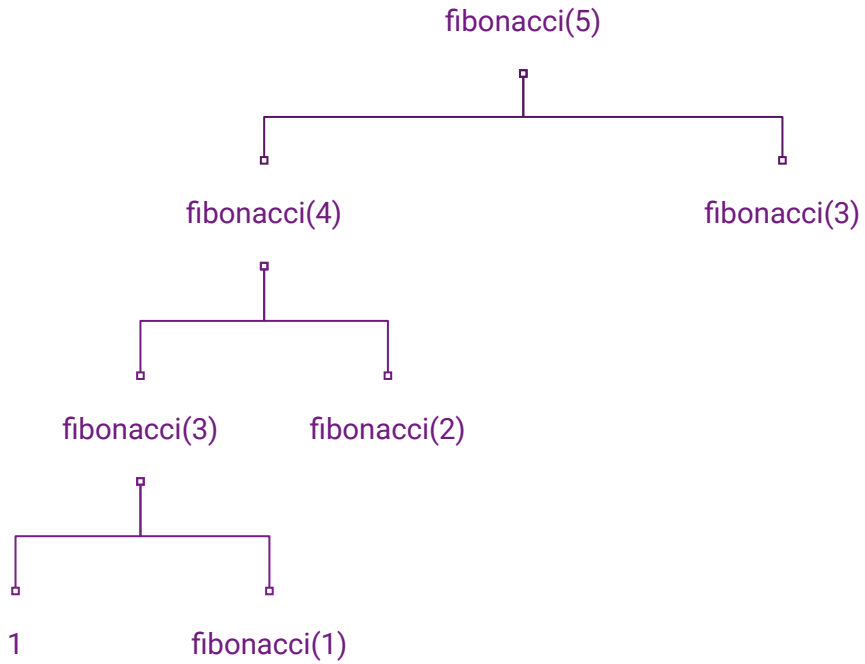
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

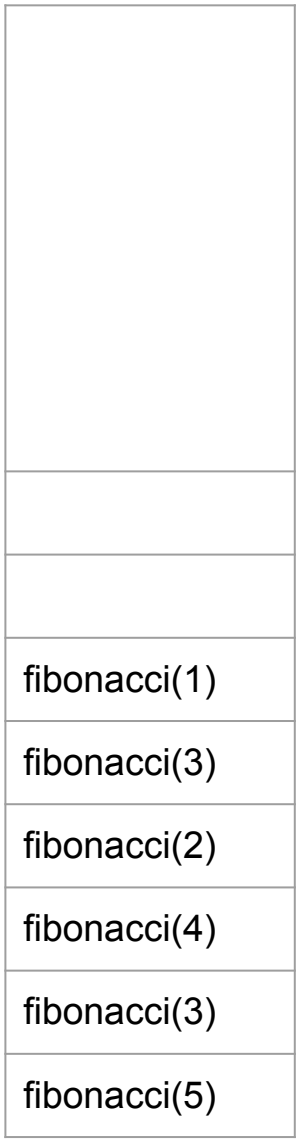




Fibonacci-Zahlen

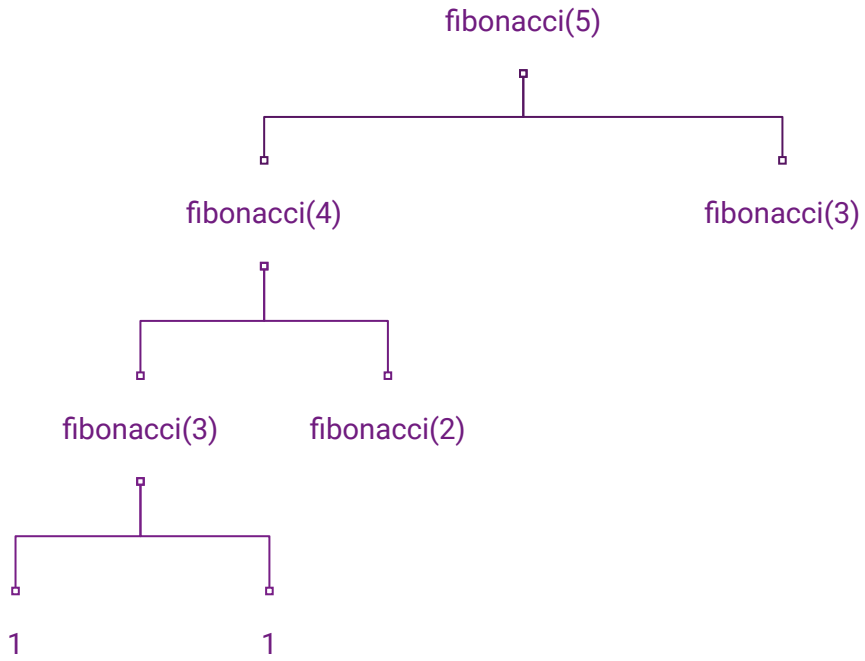
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

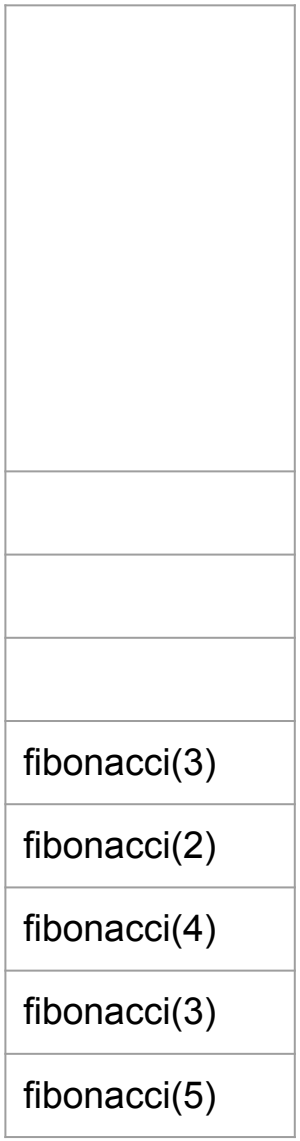




Fibonacci-Zahlen

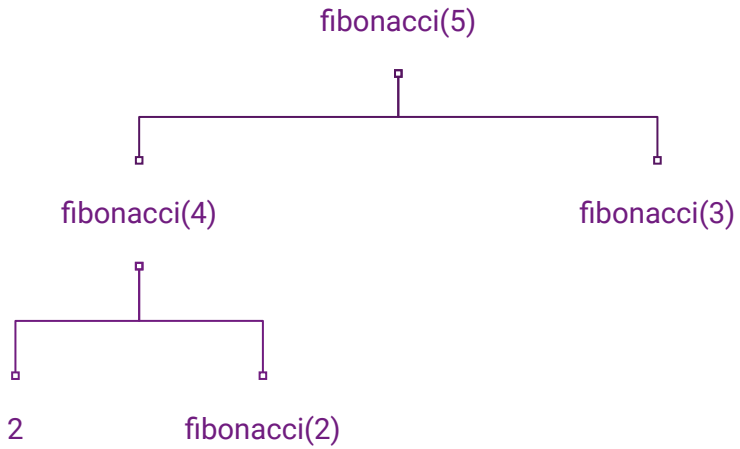
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

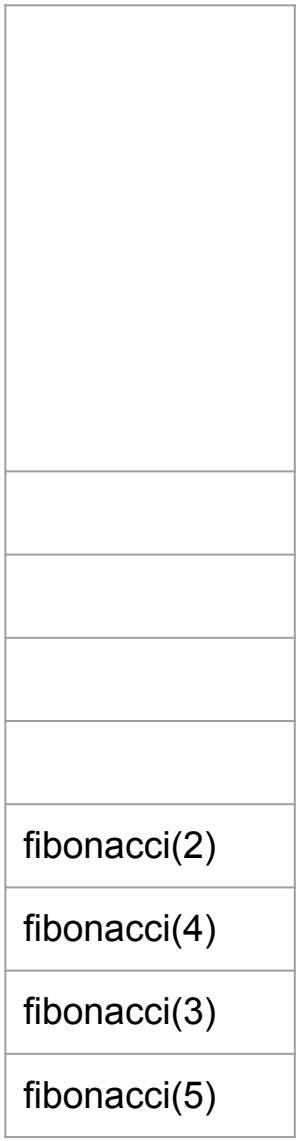




Fibonacci-Zahlen

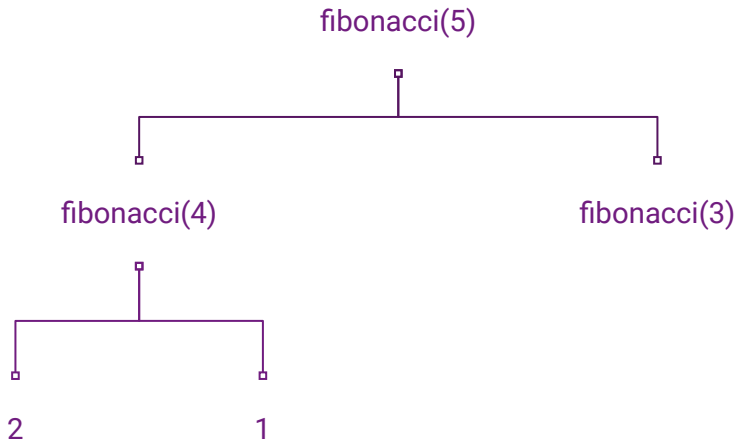
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

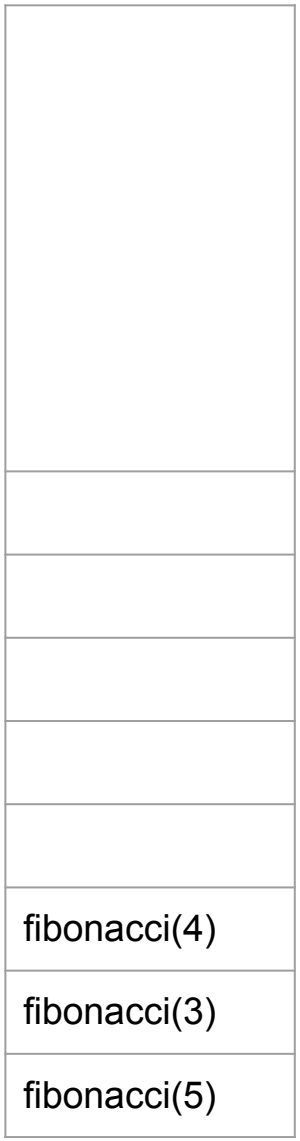




Fibonacci-Zahlen

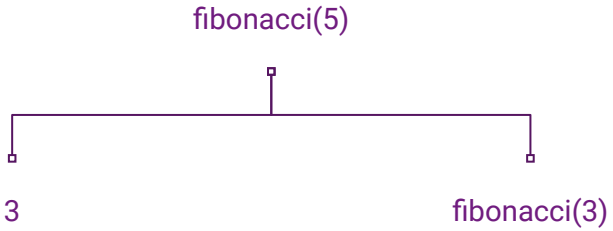
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

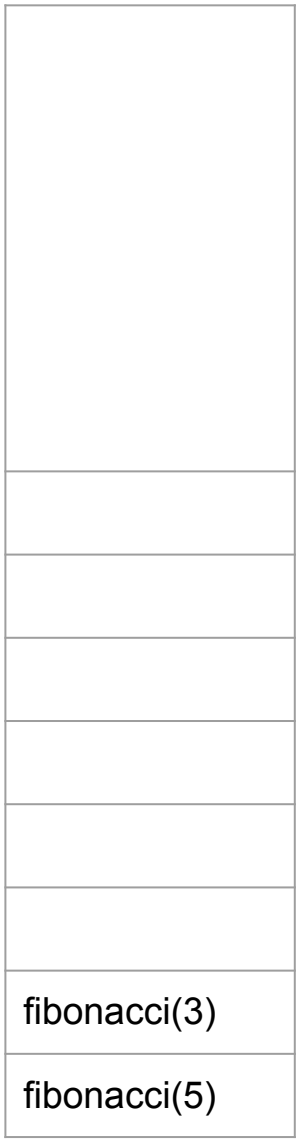




Fibonacci-Zahlen

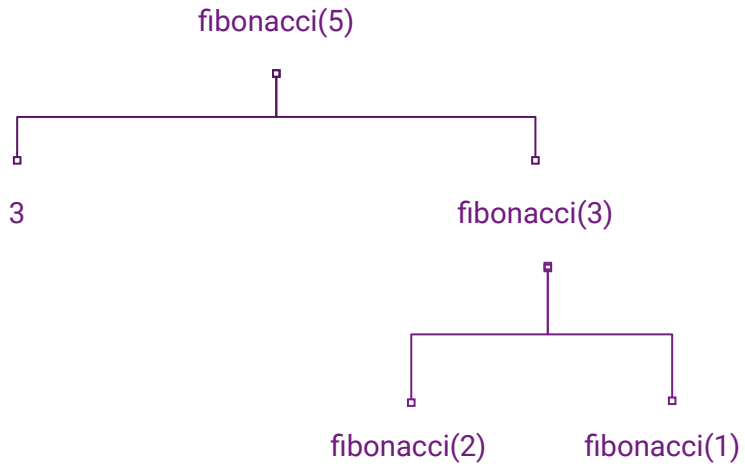
```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

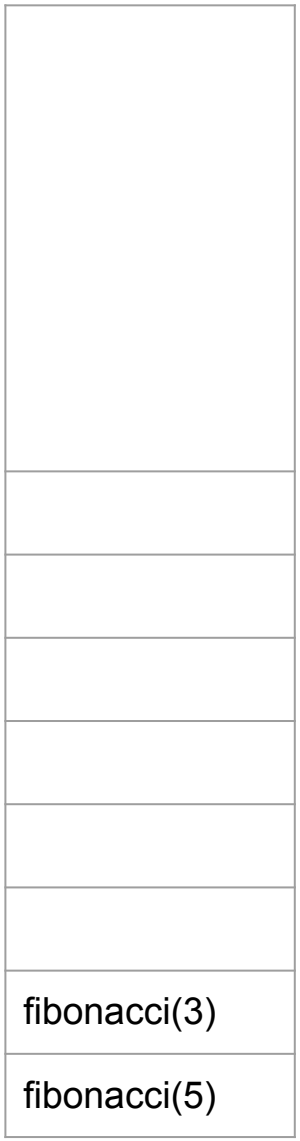




Fibonacci-Zahlen

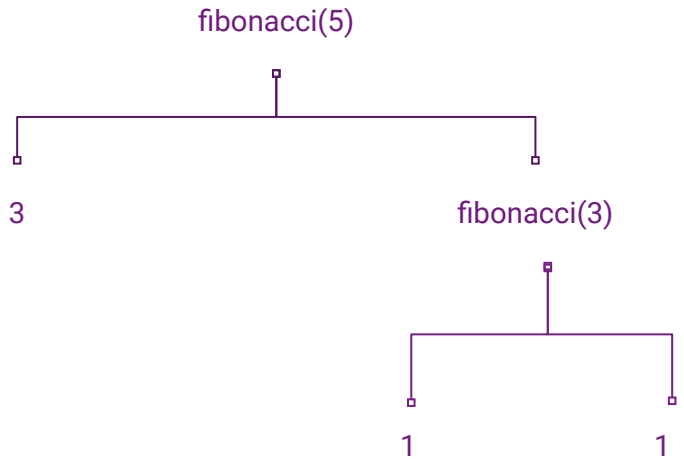
```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

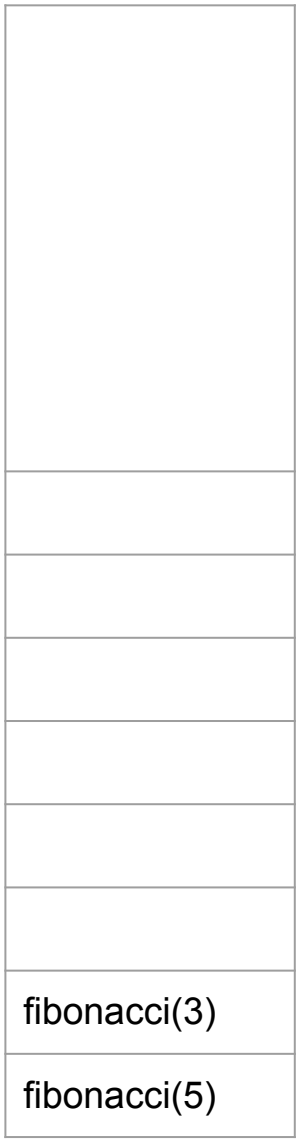




Fibonacci-Zahlen

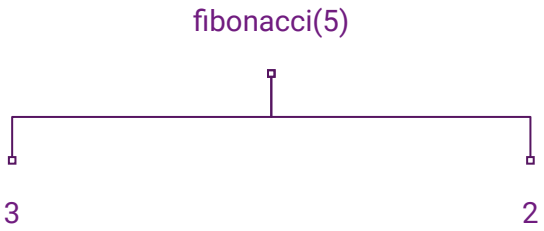
```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

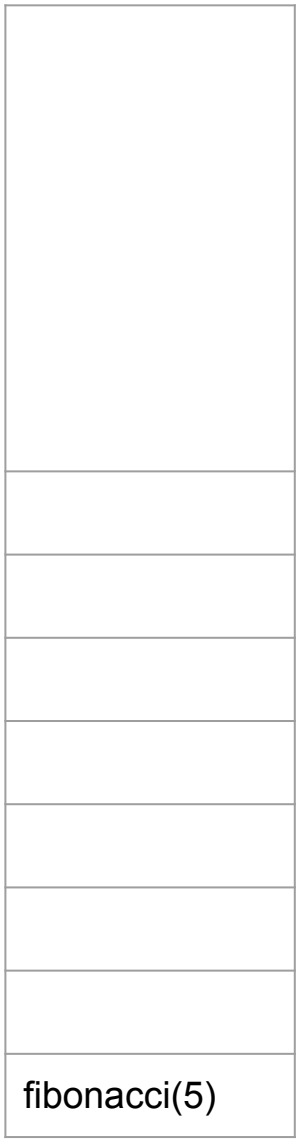




Fibonacci-Zahlen

```python
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

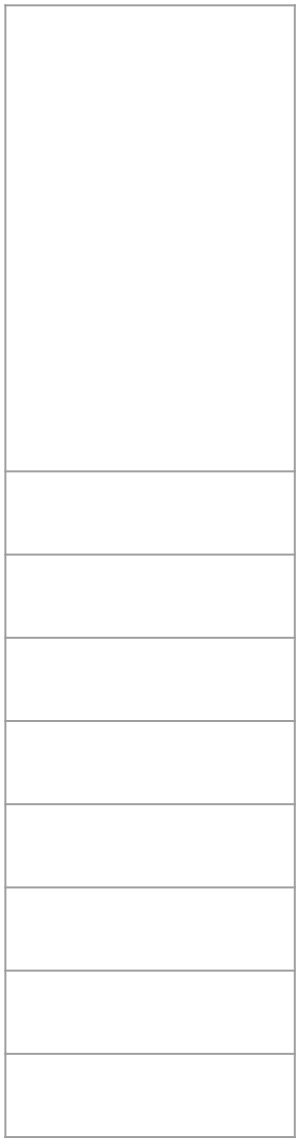




Fibonacci-Zahlen

```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

fibonacci(5) = 5



Fakultätsfunktion

die Fakultätsfunktion:

4! = 4 * 3 * 2 * 1

rekursive Definition der Fakultätsfunktion:

a. `faku(n) = 1`, falls `n = 0`

b. `faku(n) = n * faku(n-1)`, falls `n > 0`

```
def factorial(n):
    """
      compute the factorial
      n  is a positive integer
      return n!
    """
    if n== 0:
        return 1
    return  factorial(n-1)*n
```

Rekursion

Abbruchbedingung?

Analog zu unendlichen Schleifen mit for- und while- Schleifen

```
def gogogo(x):
    print x

    if x % 2 == 0:
        return gogogo(x / 2)
    else:
        return gogogo(3 * x + 1)
```

Rekursion vs. Iteration

- rekursive Algorithmen sind oft **natürlicher** und **einfacher** zu finden
- die **Korrektheit** rekursiver Algorithmen ist oft einfacher zu prüfen
- rekursive Lösungen sind i.d.R. statisch kürzer und auch, weil verständlicher, änderung freundlicher

Rekursion vs. Iteration

jeder rekursive Algorithmus kann in einen iterativen transformiert werden

```
TailRecursiveAlgorithm (…) {
    if condition {
        A
    } else {
        B
        TailRecursiveAlgorithm(…);
    }
}
```

```
IterativeAlgorithm (…) {
    while not condition {
        B
    }
    A
}
```

Rekursion vs. Iteration

von Iteration zu Rekursion

```
IterativeAlgorithm (…) {
    A
    while condition {
      B
    }
    C
}
```

```
Algorithm (…) {
    A
    RecursiveAlgorithm(…);
    C
}

RecursiveAlgorithm (…) {
    if condition {
      B
      RecursiveAlgorithm(…);
    }
}
```

Übung

Schreiben Sie eine Funktion, die die Summe aller Ziffern einer Zahl berechnet.

Übung

Schreiben Sie eine Funktion, die a**b berechnet.

Übung

Schreiben Sie für einen String eine Funktion, die true zurückgibt, wenn die angegebene Zeichenkette palindrom ist, andernfalls false.

Übung

Schreiben Sie eine Funktion, die alle Großbuchstaben eines Strings ermittelt.

Übung

Schreiben Sie eine Funktion, die prüft, ob eine Liste sortiert ist.

Übung

Schreiben Sie eine Funktion, die einen String umkehrt.

Übung

Schreiben Sie ein Programm, um einen Stapel mit hilfe von Rekursion umzukehren.

Sie dürfen keine Schleifenkonstrukte wie while, for verwenden, und Sie können nur die folgenden Funktionen auf `Stack S` verwenden:

- `is_empty(S)`
- `push(S)`
- `pop(S)`

Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren
- Welche Algorithmen sind besser?
- nicht-funktionaler Eigenschaften:
  - Zeiteffizienz: Wie lange dauert die Ausführung?
  - Speichereffizienz: Wie viel Speicher wird zur Ausführung benötigt?
  - Benötigte Netzwerkbandbreite
  - Einfachheit des Algorithmus
  - Aufwand für die Programmierung

Ressourcenbedarf

- Prozesse verbrauchen:
  - Rechenzeit
  - Speicherplatz

- Die Ausführungszeit hängt ab von:
  - der konkreten Programmierung
  - Prozessorgeschwindigkeit
  - Programmiersprache
  - Qualität des Compilers

Beispiel

```
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

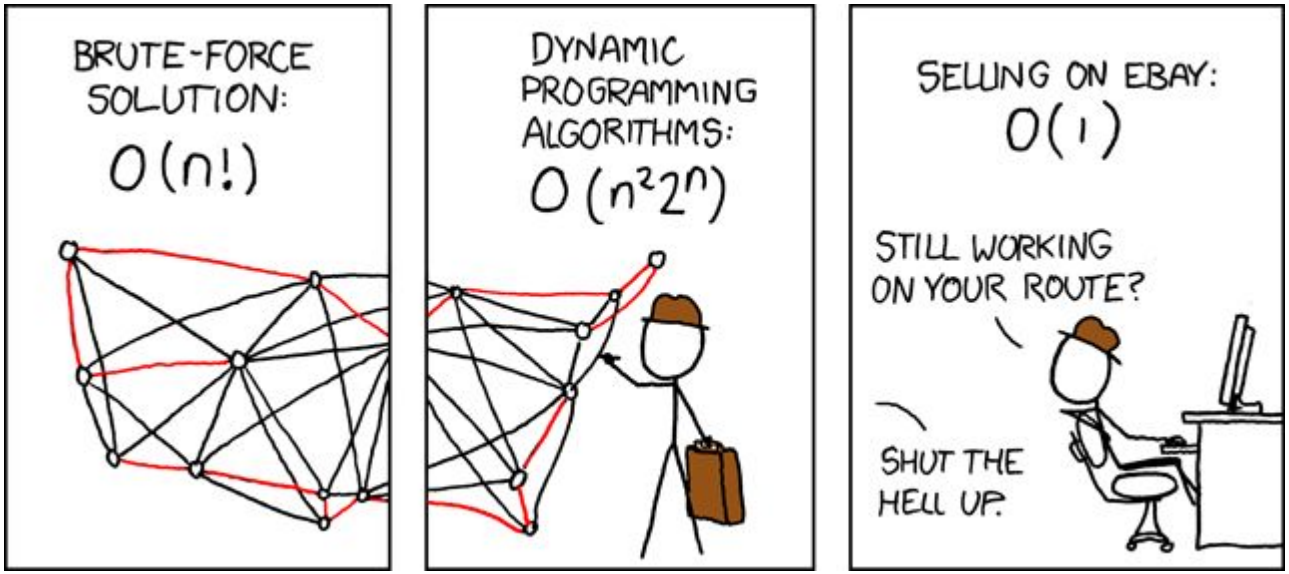
Leistungsverhalten

- **Speicherplatzkomplexität:**
  - Wird primärer & sekundärer Speicherplatz effizient genutzt?
- **Laufzeitkomplexität:**
  - Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- **Theorie:**
  - liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst
- Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems

Laufzeit

Die Laufzeit **T(x)** eines Algorithmus **A** bei Eingabe **x** ist definiert als die Anzahl von Basisoperationen, die Algorithmus **A** zur Berechnung der Lösung bei Eingabe **x** benötigt

Ziel: Laufzeit = Funktion der Größe der Eingabe



Laufzeit

- Sei **P** ein gegebenes Programm und **x** Eingabe für **P**, **|x|** Länge von **x**, und **T(x)** die Laufzeit von **P** auf **x**

- **Ziel:** beschreibe den Aufwand eines Algorithmus als Funktion *der Größe des Inputs*

- **Der beste Fall:**

  $T(n) = inf \{T(x) \mid |x| = n, x \text{ Eingabe für } P\}$

- **Der schlechteste Fall:**

  $T(n) = sup \{T(x) \mid |x| = n, x \text{ Eingabe für } P\}$

Minimum-Suche

Eingabe: Array von `n` Zahlen

Ausgabe: index `i`, so dass `a[i]<a[j]`, für alle `j`

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

Minimum-Suche

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

Kosten:	Max Anzahl:
c1	1
c2	n-1
c3	n-1
c4	n-1

Zeit:

$T(n) = c1 + (n-1)\ (c2+c3+c4) < c5n + c1$

n = Größe des Arrays