View on GitHub

# Praktikum am Gymnasium Lichtenstein

## Rekursion in Python

Link zum Arbeitsblatt

Rekursion ist eine fortgeschrittene Programmiertechnik. Weil Python mächtige und flexible Schleifenkonstruktionen zur Verfügung stellt, werden in Python rekursive Lösungen vor allem verwendet, um über Strukturen mit unvorhersehbaren Formen und Tiefen zu iterieren. Daher ist die Rekursion eine nützliche Technik, die man kennen muss.

Weil rekursiv implementierte Funktionen eine fortgeschrittenere Lösung für Probleme darstellen, und vor allem für anspruchsvollere Aufgaben ein nützliches Hilfsmittel sind, lohnt es sich, Fehlvorstellungen im Zusammenhang mit der rekursiven Implementierung von Funktionen aufzudecken und richtig zu stellen.

## Was heist *Rekursion*?

Rekursion bedeutet auf sich selber Bezug zu nehmen. Im Programmieren ist dies dann der Fall, wenn eine Funktion sich selber aufruft.

## Was charakterisiert eine rekursive Funktion?

Gegeben ist das folgende Beispiel:

```
def shining():

    print("All work and no play"
          "makes Jack a dull boy.")

    shining()
```

Was geschieht, wenn die Funktion aufgerufen wird?

Es wird grundsätzlich

```
All work and no play makes Jack a dull boy.
```

endlos ausgegeben.
Damit Rekursion zielführend eingesetzt werden kann, braucht es eine Abbruchbedinung.

Wie muss das Beispiel angepasst werden, damit die Funktion nach fünf ausgaben stoppt?

```
def shining_u(stop=0):
    if stop >= 5:
        return
    print("All work and no play"
          "makes Jack a dull boy.")
    stop += 1
    shining_u(stop)
```

*Zeitbedarf: 5'*

## Wie kommt man zu einer rekursiven Funktion?

Ausgangslage bietet der kleine Gauss:

$$1+2+3+4+\cdots+n=\sum_{k=1}^{n}k=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$

Kann diese Aufgabenstellung auch rekursiv definiert werden?

$$\sum_{k=1}^{n}k=\begin{cases}1, & n=1 \quad \text{Basisfall}\\ n+\sum_{k=1}^{n-1}k, & \forall n>1 \quad \text{Rekursionsfall}\end{cases}$$

Dies bietet die Möglichkeit, ein grosses Problem solange in kleinere Probleme zu zerlegen, bis ein lösbares Problem übrig bleibt.

*Zeitbedarf: 10'*

# Rekursion sichtbar machen

Wie muss die Funktion gauss() angepasst werden, damit die Rekursion sichtbar gemacht werden kann?

```
gauss(5)
    gauss(4)
        gauss(3)
            gauss(2)
                gauss(1)
                return 1
            return 2 + 1
        return 3 + 3
    return 4 + 6
return 5 + 10
```

*Zeitbedarf: 5'*

# Anwendungsbeispiel $n!$

Kann $n!$ rekursiv formuliert werden?

$$n!=\begin{cases}1, & n=0 \vee n=1 \quad Basisfall\\ n\cdot(n-1)! & \forall n>1 \quad Rekursionsfall\end{cases}$$

*Zeitbedarf: 5'*

# Rekursion ist nicht alternativlos

Rekursive Funktionen können auch iterativ geschrieben werden.
Wie sieht der kleine Gauss als schleife aus?

```
def gauss_loop(n):
    total = 0
    while n >= 1:
        total += n
        n -= 1
    return total
```

*Zeitbedarf: 5'*

# Beurteilung von rekursiven Lösungen

- Funktionen sind oft einfach zu schreiben
- die inneren Abläufe der Funktionen sind schwerer nachzuvollziehen

- braucht viel Speicherplatz

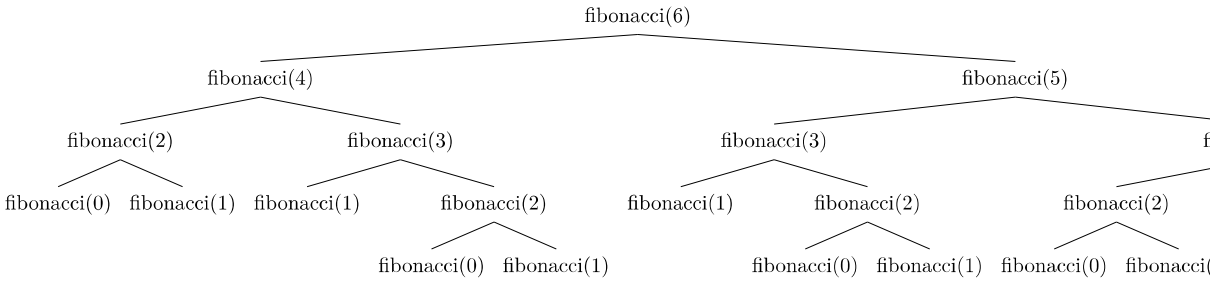
*Zeitbedarf: 5'*

## Darstellung der Stärken und Schwächen am Beispiel der Fibonacci Zahlen

Die Fibonacci Zahlen können rekursiv folgendermassen dargestellt werden:

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & \forall n > 1 \end{cases}$$

aber




*Zeitbedarf: 10'*