Rekursion

Prüfungsvorbereitung

Aufgabe 1

Beschreibe den Begriff *Rekursion* in der Programmierung.

Aufgabe 1

Bei der Rekursion enthält die Definition einer Funktion einen Teil, in dem sie sich selber aufruft. Damit bei der Ausführung der Funktion kein endloser Prozess entsteht, muss die Definition der rekursiven Funktion eine Abbruchbedingung (*Base Case*) enthalten. werden.

Aufgabe 2

Beschreibe je einen Vor- und einen Nachteil des Programmierens mit einer Rekursion.

Aufgabe 2

- *Vorteil(e):* Rekursiv definierte Funktionen haben oft einen kurzen Quellcode und sind (für erfahrene Anwender) manchmal übersichtlicher als die entsprechende iterative Version.
- *Nachteil:* Jeder rekursive Aufruf benötigt zusätzlichen Speicherplatz für die Zwischenresultate, was dazu führen kann, dass die Rekursion aufgrund von Speichermangel „abstürzt“.

Aufgabe 3

Welche Ausgabe macht die folgende Python-Funktion?

```
def f(n):
    if n == 1:
        return 7
    else:
        return n + f(n-1)

print(f(4))
```

Aufgabe 3

```
def f(n):
    if n == 1:
        return 7
    else:
        return n + f(n-1)

print(f(4))
```

$$f(4) = 4 + f(3) = 4 + (3 + f(2)) = 4 + (3 + (2 + f(1)))$$
$$= 4 + (3 + (2 + 7)) = 4 + (3 + 9) = 4 + 12 = 16$$

Aufgabe 4

Welche Ausgabe macht die folgende Python-Funktion?

```
def f(n):
    if n < 2:
        return 3
    else:
        return 2*f(n-1) + 1

print(f(4))
```

Aufgabe 4

```
def f(n):
    if n < 2:
        return 3
    else:
        return 2*f(n-1) + 1

print(f(4))
```

$$
\begin{aligned}
f(4) &= 2 \cdot f(3) + 1 = 2 \cdot (2 \cdot f(2) + 1) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot f(1) + 1) + 1) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot 3 + 1) + 1) + 1 \\
&= 2 \cdot (2 \cdot 7 + 1) + 1 = 2 \cdot 15 + 1 = 31
\end{aligned}
$$

Aufgabe 5

Welche Ausgabe macht die folgende Python-Funktion?

```
def f(n):
    if n < 3:
        return n
    elif n % 2 == 0:
        return 2*f(n-1)
    else:
        return 2+f(n-1)

print(f(5))
```

Aufgabe 5

```
def f(n):
    if n < 3:
        return n
    elif n % 2 == 0:
        return 2*f(n-1)
    else:
        return 2+f(n-1)

print(f(5))
```

$$f(5) = 2 + f(4) = 2 + (2 \cdot f(3)) = 2 + (2 \cdot (2 + f(2)))$$
$$= 2 + (2 \cdot (2 + 2)) = 2 + (2 \cdot 4) = 2 + 8 = 10$$

Aufgabe 6

Berechne die Werte der unten abgebildeten rekursiven Funktion $f(n)$ für $n = 1, 2, 4, 5$. Schreibe danach eine Funktion $g(n)$, die mit einer `for`-Schleife und ohne Rekursion die gleichen Werte wie $f(n)$ berechnet und als Werte zurückgibt.

Funktionen, welche eine Aufgabe mittels Schleifen und ohne Rekursion lösen, werden *iterativ* genannt.

```
def f(n):
    if n == 1:
        return n
    else:
        return n + f(n-1)
```

Aufgabe 6

$$f(1) = 1$$

$$f(2) = 2 + 1 = 3$$

$$f(3) = 3 + 2 + 1 = 6$$

$$f(4) = 4 + 3 + 2 + 1 = 10$$

$$f(5) = 5 + 4 + 3 + 2 + 1 = 15$$

```
def f(n): # rekursive Funktion (wie in Aufgabe)
    if n == 1:
        return n
    else:
        return n + f(n-1)

def g(n): # iterative Funktion
    s = 0
    for k in range(1, n+1):
        s = s + k
    return s

# Test, ob f(n) == g(n) gilt (n = 1
```

Aufgabe 7

Definiere eine Python-Funktion `factorial(n)`, welche die Werte $n!$ für $n = 0, 1, 2, \ldots$ rekursiv berechnet.

Aufgabe 7

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

# Testcode (optional)
for i in range(0, 10):
    print(i, factorial(i))
```

Aufgabe 8

Definiere eine Python-Funktion `factorial(n)`, welche die Werte $n!$ für $n = 0, 1, 2, \ldots$ iterativ, also mit einer Schleife berechnet. Verwende zum Platzsparen eine Einrückung von 2 statt 4 Zeichen.

Aufgabe 8

```
def factorial(n):
    f = 1                       # Alternative:
    for k in range(0,n):        # for k in range(1, n+1)
        f = f * (k+1)           #     f = f * k
    return f

# Testcode (optional)
for i in range(0, 10):
    print(i, factorial(i))
```

Aufgabe 9

Definiere eine Python-Funktion `fib(n)`, welche die Werte

`fib(1)=1, fib(2)=1, fib(3)=2, fib(4)=3, fib(5)=5, fib(6)=8, ...`

rekursiv berechnet. Verwende eine Einrückungstiefe von 2 statt 4 Zeichen.

Aufgabe 9

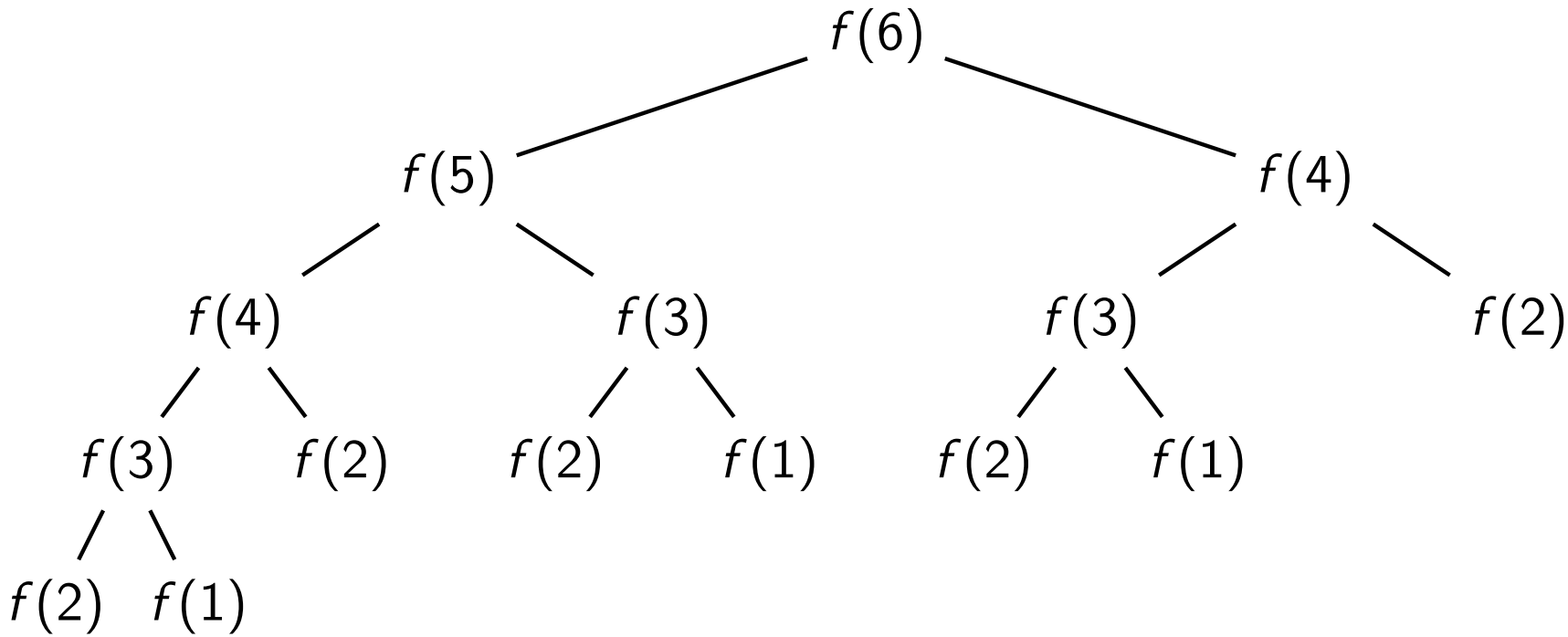
```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1)+fib(n-2)

# Testcode
for k in range(1, 36):
    print(k, fib(k))
```

Aufgabe 10

Warum verlangsamt sich die Ausführung der rekursiven Funktion zur Berechnung der Werte der Fibonacci-Folge immer stärker?

Aufgabe 10

Der Rekursionsbaum der Fibonacci-Folge zeigt, dass für $n > 1$ der Aufruf von $f(n)$ umso häufiger vorkommt, je kleiner die Zahl n ist. Daher verbringt die rekursive Funktion immer mehr Zeit damit, Werte die sie bereits berechnet hat, in weiteren Rekursionsaufrufen noch weitere Male zu berechnen.



Aufgabe 11

Definiere eine Python-Funktion `fib(n)`, welche die Werte

`fib(1)=1, fib(2)=1, fib(3)=2, fib(4)=3, fib(5)=5, fib(6)=8, ...`

iterativ berechnet. Verwende eine Einrückungstiefe von 2 statt 4 Zeichen.

Aufgabe 11

```
def fib(n):
    a1 = 0
    a2 = 1
    for i in range(1, n):
        a2, a1 = a1 + a2, a2
    return a2

# Testcode
for k in range(0, 36):
    print(k, fib(k))
```