

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

- *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

```
def f(n):  
    if n == 1:  
        return 7  
    else:  
        return n + f(n-1)  
  
print(f(4))
```

$$\begin{aligned} f(4) &= 4 + f(3) = 4 + (3 + f(2)) = 4 + (3 + (2 + f(1))) \\ &= 4 + (3 + (2 + 7)) = 4 + (3 + 9) = 4 + 12 = 16 \end{aligned}$$

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

```
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))
```

$$\begin{aligned} f(5) &= 2 + f(4) = 2 + (2 \cdot f(3)) = 2 + (2 \cdot (2 + f(2))) \\ &= 2 + (2 \cdot (2 + 2)) = 2 + (2 \cdot 4) = 2 + 8 = 10 \end{aligned}$$

Aufgabe 6

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

```
def g(n):  
    summe = 0  
    for k in range(1, n+1):  
        summe = summe + k  
    return summe
```

```
for i in range(1, 20):  
    print(f(i), g(i))
```

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

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

# Alternative:
# for k in range(1, n+1):
#     f = f * k

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

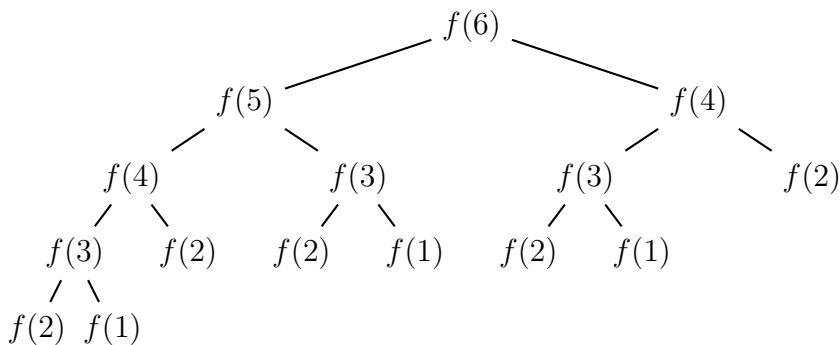
Aufgabe 9

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

# Testcode
print(*[fibo(i) for i in range(1, 10)])
```

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

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

# Testcode
print(*[fibo(i) for i in range(1, 10)])
```