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 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 7

```
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))
```