25. **Fibonacci-Folge**

Für n ∈ {0, 1, 2, 3, ... } läßt sich die Fibonacci-Folge rekursiv definieren:

Rekursionsanfang: **fibo(0) = 0**
fibo(1) = 1

Rekursionsvorschrift: **fibo(n) = fibo(n–1) + fibo(n–2)** falls n > 1

(In Worten: für n > 1 erhält man das n-te Folgenglied als Summe der beiden vorangehenden Folgenglieder.)

a) Schreibe und teste ein Python-Programm mit rekursivem Funktionsaufruf, welches nach Eingabe von n den Wert fibo(n) ausgibt (oder: alle Werte fibo(0), . . . , fibo(n)); implementiere auch eine Variable z, welche die Anzahl der Funktionsaufrufe ausgibt.
Ermittle den Zeitbedarf, den die Berechnung von fibo(n) erfordert.

Bemerkung: Hier handelt es sich um einen Algorithmus mit exponentieller Komplexität, denn die Anzahl z der Funktionsaufrufe wächst exponentiell mit n; bei n = 38, 39, 40, . . nimmt die Berechnung bereits sehr viel Zeit in Anspruch.

b) Wenn man **lru_cache** des Python-Moduls **functools** nutzt, läßt sich die Laufzeit erheblich verbessern (hier werden bereits berechnete Werte in einem cache zwischengespeichert); allerdings kommt man mit **lru_cache** bei der Berechnung der Ackermann-Funktion (Aufgabe 26) wegen derer ungeheuren Rekursionstiefe kaum weiter: acker(3,9) läßt sich noch berechnen, bei acker(3,10) oder acker(4,n), n>0, ist Schluß.

```
from functools import lru_cache

n = int(input('n = '))
z = 0

@lru_cache(maxsize=64)
def fibo(n):
      . . . . . . .
      . . . . . . .
```

c) Schreibe und teste ein imperativ formuliertes Python-Programm, z. B. indem die Werte der Fibonacci-Folge in einem array mit den Komponenten a[0], a[1], ... , a[n] abgelegt werden (setze a[0] = 0 und a[1] = 1). Vergleiche die Laufzeit mit dem funktional formulierten Algorithmus aus a).

26. Die **Ackermann-Funktion**

Für m, n ∈ {0, 1, 2, 3, ... } ist die Ackermann-Funktion **f** wie folgt definiert:

Rekursionsanfang: (1) **f(0,n) = n+1**

Rekursionsvorschrift: (2) **f(m,0) = f(m-1, 1)**
(3) **f(m,n) = f(m-1, f(m,n-1))**

a) Man erhält: f(0,0) = 1, f(0,1) = 2, f(0,2) = 3, f(1,0) = f(0,1) = 2
Berechne f(2,0); f(1,1); f(1,2); f(3,0) .

b) Schreibe den Algorithmus zur Berechnung der Ackermann-Funktion als Python-Programm mit rekursivem Funktionsaufruf.
Implementiere eine Zählvariable z, um die Anzahl der Funktionsaufrufe bestimmen; ermittle den Zeitbedarf zur Laufzeit.

Berechne f(3,7); f(3,8); f(4,0); f(3,8); f(3,9); f(4,1); f(4,2)

Bemerkung: Die Ackermann-Funktion ist eine berechenbare Funktion, allerdings übersteigt deren ungeheure Rekursionstiefe sehr schnell die Möglichkeiten jedes auch noch so leistungsfähigen Computers!