6. **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 ermittelt.

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) Zeige: Für die Anzahl **z(n)** der Funktionsaufrufe gilt

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

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

Hinweis: Erstelle für fibo(2), fibo(3), fibo(4) jeweils ein Baumdiagramm, so wie es für die Aufrufe von sort in dem paper „mergesort_update.pdf“ gemacht wurde.

c) 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 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):
     . . . . . . .
     . . . . . . .
```

d) Schreibe und teste ein iterativ 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).

7. **SelectionSort**

Der Algorithmus **sorting_by_direct_selection.py** (enthalten im zip-Archiv MergeSort_update.zip) hat noch Optimierungspotential hinsichtlich des Zeitbedarfs zum Sortieren einer als array gegebenen Liste. Hierzu läßt sich die Funktion **`min(x,j)`** in geeigneter Weise modifizieren; ergreife diese Möglichkeit!

Allerdings ändert diese Optimierung nichts an der quadratischen Komplexität des Algorithmus.

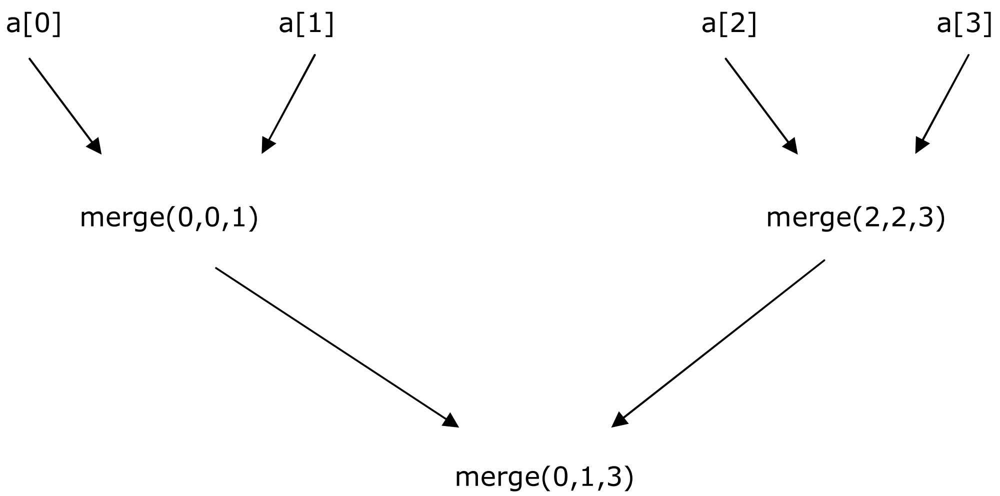
8. **MergeSort**

In dem paper **mergesort_update.pdf** (zip-Archiv MergeSort_update.zip) wurde die Funktion **f(n)** ermittelt, welche die Anzahl der Aufrufe der Funktion **`sort`** angibt.

Finde in entsprechender Weise einen Funktionsterm und eine Funktionalgleichung für die Funktion **g(n)**, welche die Anzahl der Aufrufe der Funktion **`merge`** angibt.

Hinweis:
Erstelle Baumdiagramme für n = 2, n = 4, n = 8

Baum-Diagramm für n = 4:



g(4) = 3

Implementiere im Quelltext von **mergesort.py** eine weitere Zählvariable **y**, welche die Anzahl der Aufrufe von **`merge`** ermittelt.