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.

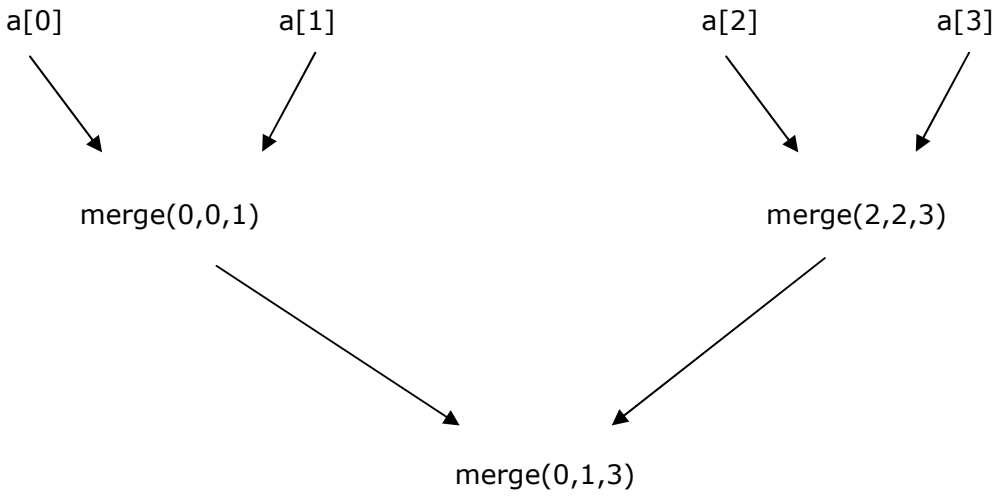
8. **MergeSort**

   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

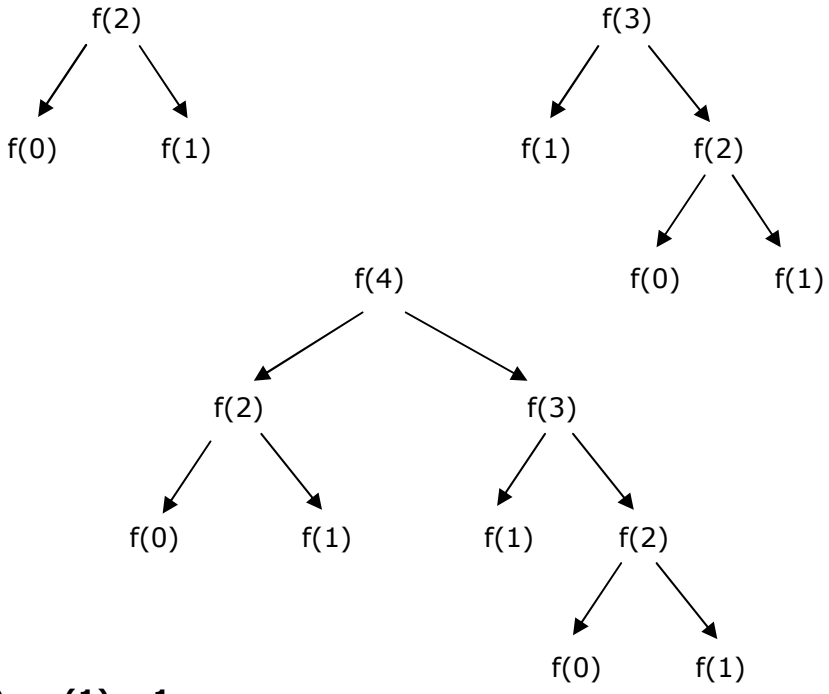
   Implementiere im Quelltext von **mergesort.py** eine weitere Zählvariable **y**, welche die Anzahl der Aufrufe von **`merge`** ermittelt.

Aufgaben 6.a), 6.c):

```
n = int(input('n = '))

def fib(n):
    global z
    z +=1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

for i in range(n+1):
    z = 0
    print('fib(',i,') = ',fib(i))
    print('# Aufrufe: ',z)
    print()
```

```
from functools import lru_cache

n = int(input('n = '))

@lru_cache(maxsize=1000)

def fib(n):
    global z
    z +=1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

for i in range(n+1):
    z = 0
    print('fib(',i,') = ',fib(i))
    print('# Aufrufe: ',z)
    print()
```

Aufgabe 6.b): f(n) = fibo(n); „ ⟶ " bedeutet: „ruft auf"



z(0) = z(1) = 1

z(2) = 1 + z(0) + z(1) = 1 + 1 + 1 = 3
z(3) = 1 + z(1) + z(2) = 1 + 1 + 3 = 5
z(4) = 1 + z(2) + z(3) = 1 + 3 + 5 = 9
z(5) = 1 + z(3) + z(4) = 1 + 5 + 9 = 15

allgemein:

z(n) = 1 + z(n-1) + z(n-2) falls n > 1

Aufgabe 6.d):

```
n = int(input('n = '))
a = list(range(0,2))
a[0] = 0
a[1] = 1

if n > 1:
   i = 2
   while i <= n:
       a.append(a[i-1] + a[i-2])
       i += 1

for i in range(n+1):
    print('fib(',i,') = ',a[i])
```

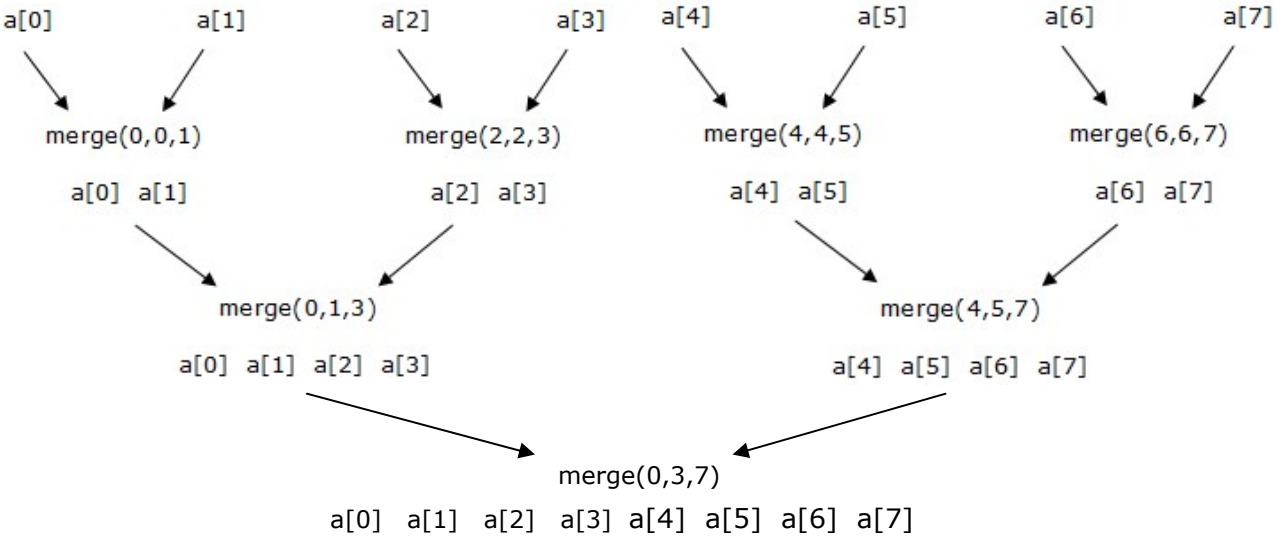
Aufgabe 7.:

Die Funktion **`min(x,j)`** ermittelt in dem array **`x[j], . . . . , x[n-1]`** das kleinste Element und weist es der Komponente **`x[j]`** zu. In der ursprünglichen Version von **`min(x,j)`** werden die Inhalte der Komponenten **`x[j]`** und **`x[i]`** immer dann unter Verwendung des Zwischenspeichers **`temp`** ausgetauscht, sobald **`x[i]`** kleiner als **`x[j]`** ist (**`j < i ≤ n-1`**); folglich finden u. U. sehr viele solcher swap-Operationen statt, die unnötig viel Rechenzeit beanspruchen.
Patriks Vorschlag: Nachdem man durch sukzessives Vergleichen den Index **`k`** des kleinsten Elements bestimmt hat, wird der swap-Vorgang nur einmal ausgeführt. (Praktische Versuche zeigen, daß man durch diese Optimierung mit einer Halbierung des Zeitbedarfs zum Sortieren eines arrays rechnen kann.)

```
def min(x,j):
    for i in range(j+1,len(x)):
        if x[i] < x[j]:
            temp = x[j]
            x[j] = x[i]
            x[i] = temp
```

```
def min(x,j):
    k = j
    minimum = x[k]
    for i in range(j+1,len(x)):
        if x[i] < minimum:
            minimum = x[i]
            k = i
    x[k] = x[j]
    x[j] = minimum
```

Aufgabe 8.: Baumdiagramm für n=8



g(1) = 0
g(n) = 1 + 2 · g(n/2) falls $n = 2^k$, $k > 1$
g(n) = n – 1

Eigenschaften der rekursiv definierten Fibonacci-Folge $\{f(n)\}_{n=0}^{\infty}$

(1) **f(0) = 0 , f(1) = 1**
(2) **f(n) = f(n–1) + f(n–2)** falls n > 1

1. Die Folge $\{f(n)\}$ ist streng monoton wachsend für n > 1.
   Beweis:
   f(n+1) – f(n) = f(n-1) > 0 falls n > 1

2. Behauptung: **$f(n) < 2^{n-1} = ½ \cdot 2^n$** falls n > 1
   Beweis: $f(n) = f(n-1) + f(n-2) < 2 \cdot f(n-1)$ wegen der Monotonie
   $< 2 \cdot 2 \cdot f(n–2) = 2^2 \cdot f(n–2)$
   $< 2^3 \cdot f(n–3)$
   . . . . . .
   $< 2^{n-1} \cdot f(n–(n-1)) = 2^{n-1} \cdot f(1) = 2^{n-1}$

3. Behauptung: **$f(n) > ½ \cdot (\sqrt{2})^n$** falls n > 2
   Beweis: n sei gerade mit n = 2·m, m > 1
   $f(2m) = f(2m–1) + f(2m–2)$
   $> 2 \cdot f(2m–2) = 2^1 \cdot f(2(m–1))$ wegen der Monotonie
   $> 2 \cdot 2 \cdot f(2m–4) = 2^2 \cdot f(2(m–2))$
   $> 2^3 \cdot f(2(m–3))$
   . . . . . .
   $> 2^{m-1} \cdot f(2(m–(m-1))) = 2^{m-1} \cdot f(2) = 2^{m–1}$
   mit m = n/2 folgt:
   $f(n) > 2^{n/2-1} = ½ \cdot 2^{n/2} = ½ \cdot (\sqrt{2})^n$

4. Folglich erhalten wir für **f(n)** die Abschätzung

   $½ \cdot (\sqrt{2})^n < f(n) < ½ \cdot 2^n$ falls n > 2

   Die Fibonacci-Folge wächst exponentiell mit n.

5. Das exponentielle Wachstum läßt sich auch an der für die Fibonacci-Folge geltenden Formel von Moivre-Binet ablesen:

   $$f(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

   Für große Werte von n kann man den Subtrahend gegenüber dem Minuend vernachlässigen.

6. Berechnet man die Fibonacci-Folge mit der rekursiv formulierten Funktion fib, erhält man für die Anzahl z(n) der Aufrufe von fib:

   z(0) = z(1) = 1

   z(n) = 1 + z(n-1) + z(n-2) , n > 1

   Wegen $z(n) \geq f(n)$ wächst auch z(n) exponentiell mit n.