Sortieren durch Mischen ("MergeSort")

Aufgabe:

Gegeben ist eine Liste **L = {a[0], a[2], a[3], ... , a[n-1]}**
von n Datenelementen, für die die Ordnungsrelationen < , > , ≤, ≥ erklärt sind. Die Inhalte dieser Datenelemente sind so anzuordnen, daß gilt:

a[0] ≤ a[2] ≤. . . . . ≤ a[n-1] .

Wir fassen die Elemente der Liste auf als Komponenten eines arrays a.

Strategie: "Divide et impera"

Eine Liste, die nur ein einziges Element enthält, ist bereits sortiert.

Die Aufgabe, die n-elementige Liste (n > 1) zu sortieren, läßt sich in 4 Schritten bewältigen:

1. **Teile die n-elementige Liste in zwei etwa gleichlange Teillisten**
2. **Sortiere die erste Teilliste gemäß den Schritten 1). - 4).**
3. **Sortiere die zweite Teilliste gemäß den Schritten 1). - 4).**
4. **Mische die sortierten Teillisten zu einer sortierten Gesamtliste**

Falls **`left < right`** wahr ist, sortiert die rekursiv definierte Funktion

`sort(array, left, right)`

die Liste

`array[left], . . . . , array[right]`

unter Verwendung der Funktion **`merge`**.

Die Funktion

`merge(array, left, middle, right)`

mischt die sortierten Teillisten

`array[left], . . . . , array[middle]`

und

`array[middle+1], . . . . , array[right]`

zu der sortierten Gesamtliste

`array[left], . . . . , array[right]` .

Quellcode der Funktion **sort** in Python:

```
def sort(array, left, right):
     if left >= right:
          return
     middle = (left + right)//2
     sort(array, left, middle)
     sort(array, middle + 1, right)
     merge(array, left, middle, right)
```

Aufruf zum Sortieren der aus den **n** Komponenten

```
a[0], a[2], a[3], . . . . , a[n-1]
```

bestehenden Liste a:

```
sort(a, 0, len(a)-1)
```

Aufwandsbetrachtung:

Mit **A(n)** werde der Aufwand (die Anzahl elementarer Verarbeitungsschritte wie z. B. Additionen, Wertzuweisungen, Vergleichsoperationen) bezeichnet, eine aus n Komponenten bestehende Liste zu sortieren.

Dann gilt:

A(n) = 2 x Aufwand zum Sortieren einer Teilliste mit n/2 Elementen
+ Aufwand zum Mischen zweier sortierter Teillisten

A(n) = A(n/2) + A(n/2)
+ Aufwand zum Mischen zweier sortierter Teillisten

Der Aufwand zum Mischen zweier sortierter Teillisten zu einer sortierten Gesamtliste wächst linear mit der Anzahl **n** der zu sortierenden Datenelemente; somit erhalten wir für den Funktionsterm **A(n)** die Funktionalgleichung (c = Konstante = Proportionalitätsfaktor)

(*) **A(n) = A(n/2) + A(n/2) + c · n** mit der Bedingung
(**) **A(1) = 0** .

Behauptung: Die Funktion

$$\mathbf{A(n) = c \cdot n \cdot \log_2(n)}$$

ist Lösung der Funktionalgleichung (*) mit der Anfangsbedingung (**).

Beweis:

$$\begin{aligned} A(n/2) + A(n/2) + c \cdot n &= 2 \cdot A(n/2) + c \cdot n \\ &= 2 \cdot c \cdot n/2 \cdot \log_2(n/2) + c \cdot n \\ &= c \cdot n \cdot (\log_2(n) - \log_2(2)) + c \cdot n \\ &= c \cdot n \cdot (\log_2(n) - 1) + c \cdot n \\ &= c \cdot n \cdot \log_2(n) \\ &= A(n) \end{aligned}$$

Damit ist (*) erfüllt; wegen $\log_2(1) = 0$ genügt A(n) auch der Bedingung (**).

Bemerkung: Mit Methoden der Analysis läßt sich die Eindeutigkeit der Lösung des Problems (*), (**) zeigen, somit ist mit $A(n) = c \cdot n \cdot \log_2(n)$ die einzige Lösung der Funktionalgleichung gefunden.

Allgemein läßt sich beweisen, daß der Aufwand zum Sortieren von n Datensätzen grundsätzlich mindestens von der Ordnung $n \cdot \log_2(n)$ wächst. In diesem Sinne kann das Sortierverfahren „MergeSort" als optimales Vefahren gelten.

Ergänzende Betrachtung zum Speicherplatzbedarf:

Nachdem wir festgestellt haben, daß der Aufwand zum Sortieren von n Datenelementen von der Ordnung $n \cdot \log_2(n)$ wächst und damit ein Optimum erreicht ist, erhebt sich die Frage, ob dieser Vorteil durch die zwar elegante, aber rekursive Formulierung des Sortieralgorithmus nicht aufgehoben wird; denn rekursive Algorithmen haben grundsätzlich den Nachteil, daß sie während der Laufzeit mehr Arbeitsspeicher beanspruchen als iterative. Daß dieser Effekt bei MergeSort nicht oder nur unwesentlich ins Gewicht fällt, zeigt folgende Überlegung:

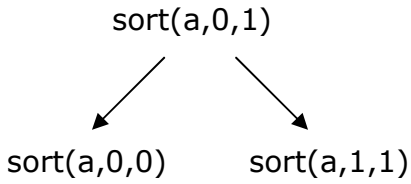
Mit **f(n)** bezeichnen wir die Anzahl der gleichzeitig aktiven Aufrufe der Funktion **`sort`**, wenn eine Liste mit **`n`** Datenelementen zu sortieren ist.

O. B. d. A. sei n eine Zweierpotenz, d. h. $n=2^k$, $k \in \{0, 1, 2, 3, \ldots\ldots\}$.

Bemerkung: Der Pfeil ⟶ *bedeutet: „ruft auf"*

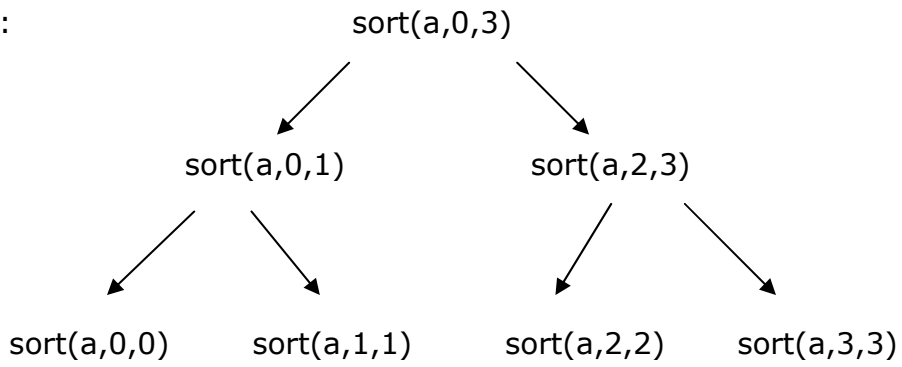
n = 1: sort(a,0,0) 1 Aufruf

n = 2:



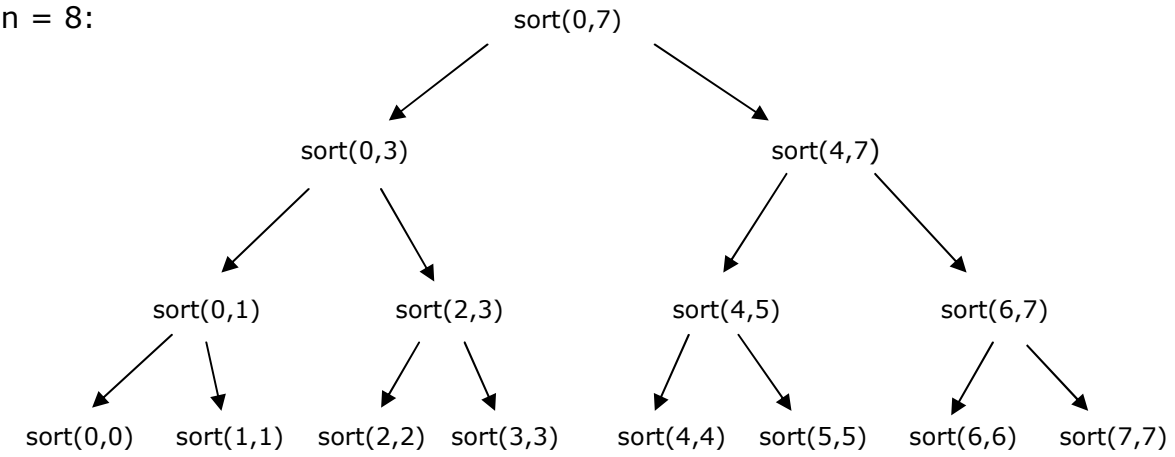
$1 + 2 \cdot 1 = 3$ Aufrufe

n = 4:



$1 + 2 \cdot 3 = 7$ Aufrufe

n = 8:



$1 + 2 \cdot 7 = 15$ Aufrufe

$f(1) = 1 = 1 = 2 \cdot 1 - 1$

$f(2) = 1 + 2 \cdot 1 = 3 = 2 \cdot 2 - 1$

$f(4) = 1 + 2 \cdot 3 = 7 = 2 \cdot 4 - 1$

$f(8) = 1 + 2 \cdot 7 = 15 = 2 \cdot 8 - 1$

$f(16) = 1 + 2 \cdot 15 = 31 = 2 \cdot 16 - 1$

$f(32) = 1 + 2 \cdot 31 = 63 = 2 \cdot 32 - 1$

allgemein:

$f(n) = 2 \cdot n - 1$

Offensichtlich ist f(n) Lösung der rekursiv definierten Funktionalgleichung

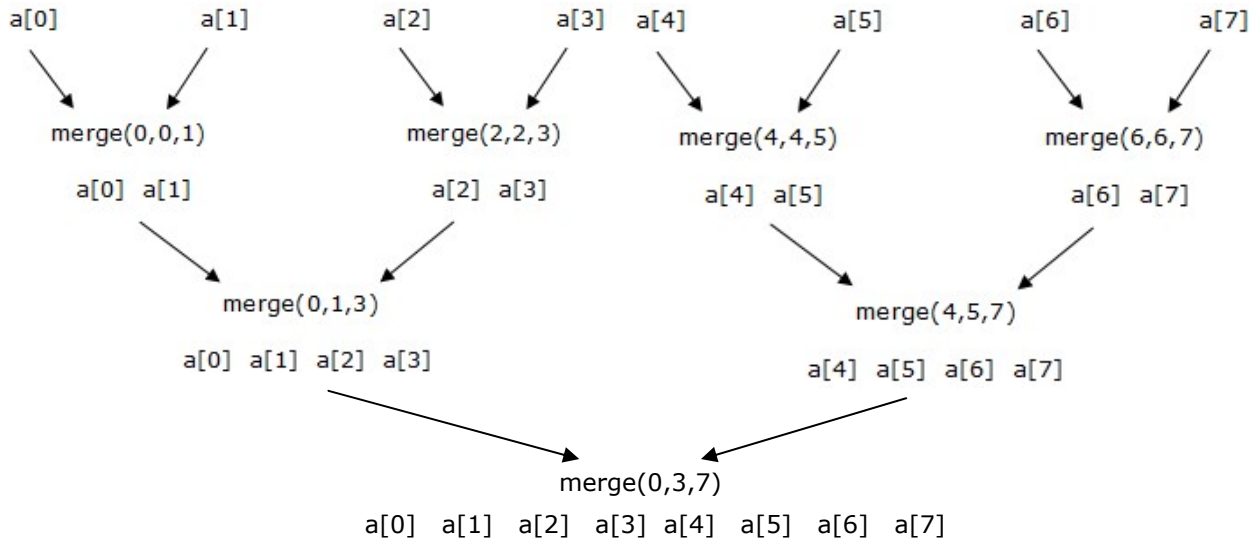
$f(n) = 1 + 2 \cdot f(n/2)$

mit der Anfangsbedingung **$f(1) = 1$** .

Die Anzahl **f(n)** der gleichzeitig aktiven Aufrufe von MergeSort und damit der Speicherplatzbedarf während der Laufzeit wächst somit linear mit **n**, also wesentlich schwächer als die Anzahl **A(n)** elementarer Rechenoperationen.

Die rekursiv veranlaßten Aufrufe der Funktion **`sort`** zerlegen die zu sortierende Liste in Teillisten jeweils der Länge 1, die als ein-elementige Listen bereits sortiert sind. Die Funktion **`merge`** mischt je zwei sortierte Teillisten zu jeweils einer sortierten Liste gemäß folgendem Diagramm:

Bemerkung: Der Pfeil ⟶ *bedeutet: „wird gemischt"*



Für die Anzahl g(n) der Aufrufe von merge verifiziert man unmittelbar:

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

Lösung der vorstehenden Funktionalgleichung:

g(n) = n – 1

Die Anzahl f(n) der Aufrufe der Funktion **`sort`** und die Anzahl g(n) der Aufrufe der Funktion **`merge`** wachsen jeweils linear mit n.

Februar 2021

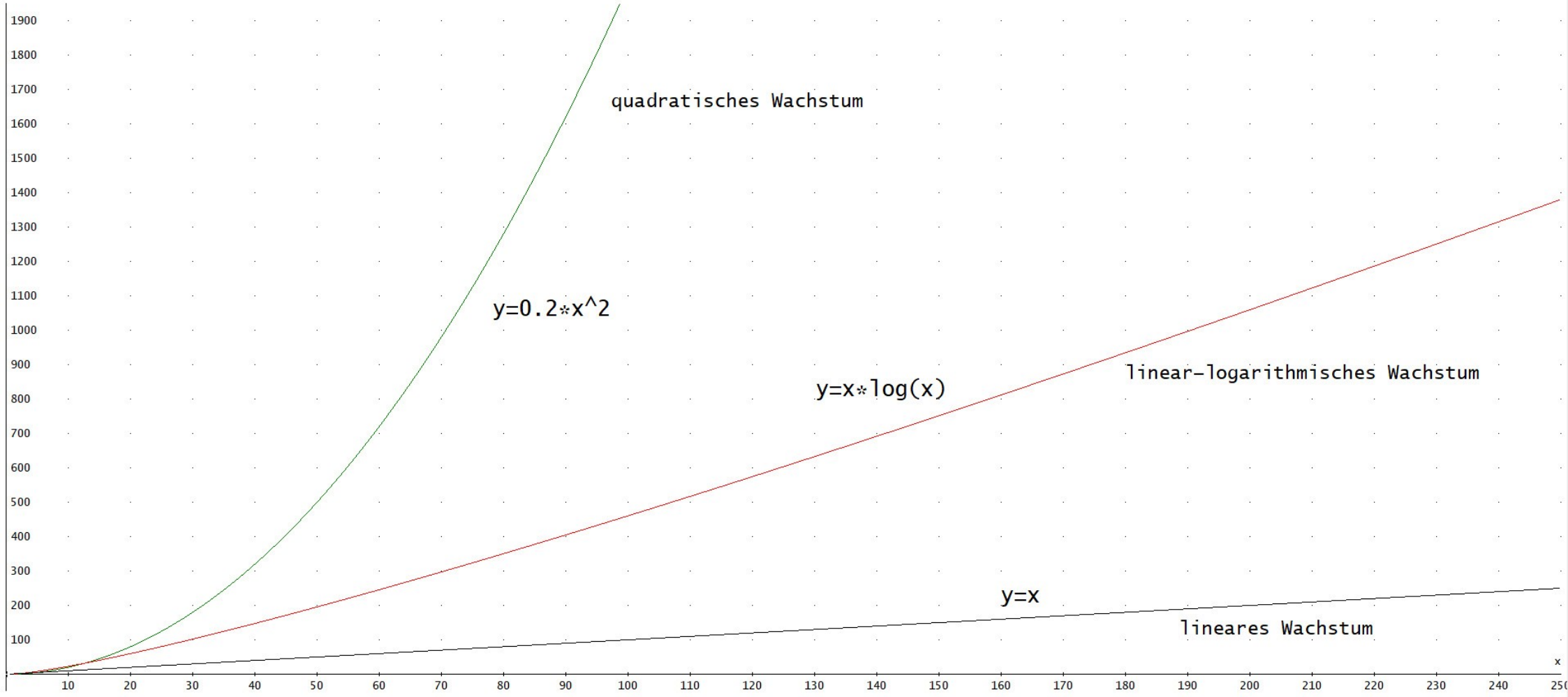
Bemerkung:
Für den Aufwand A(n) und folglich den Zeitbedarf zur Laufzeit des Algorithmus erhalten wir bei

- SelectionSort: $A(n) \sim n^2$
- MergeSort: $A(n) \sim n \cdot \log_2(n)$
- Fibonacchi-Folge: $A(n) \sim 2^n$ (bei rekursiver Berechnung)
- BinarySearch: $A(n) \sim \log_2(n)$

Entsprechend haben

- SelectionSort quadratische Komplexität,
- MergeSort linear-logarithmische Komplexität,
- die rekursive Berechnung der Fibonacchi-Folge exponentielle Komplexität,
- BinarySearch logarithmische Komplexität.

Algorithmen mit exponentieller Komplexität erweisen sich in der Praxis als unbrauchbar.


quadratisches Wachstum
y=0.2*x^2
y=x*log(x)
linear-logarithmisches Wachstum
y=x
lineares Wachstum
x
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210
220
230
240

MergeSort

Anzahl und Reihenfolge der Aufrufe der Funktionen **`sort`** und **`merge`**

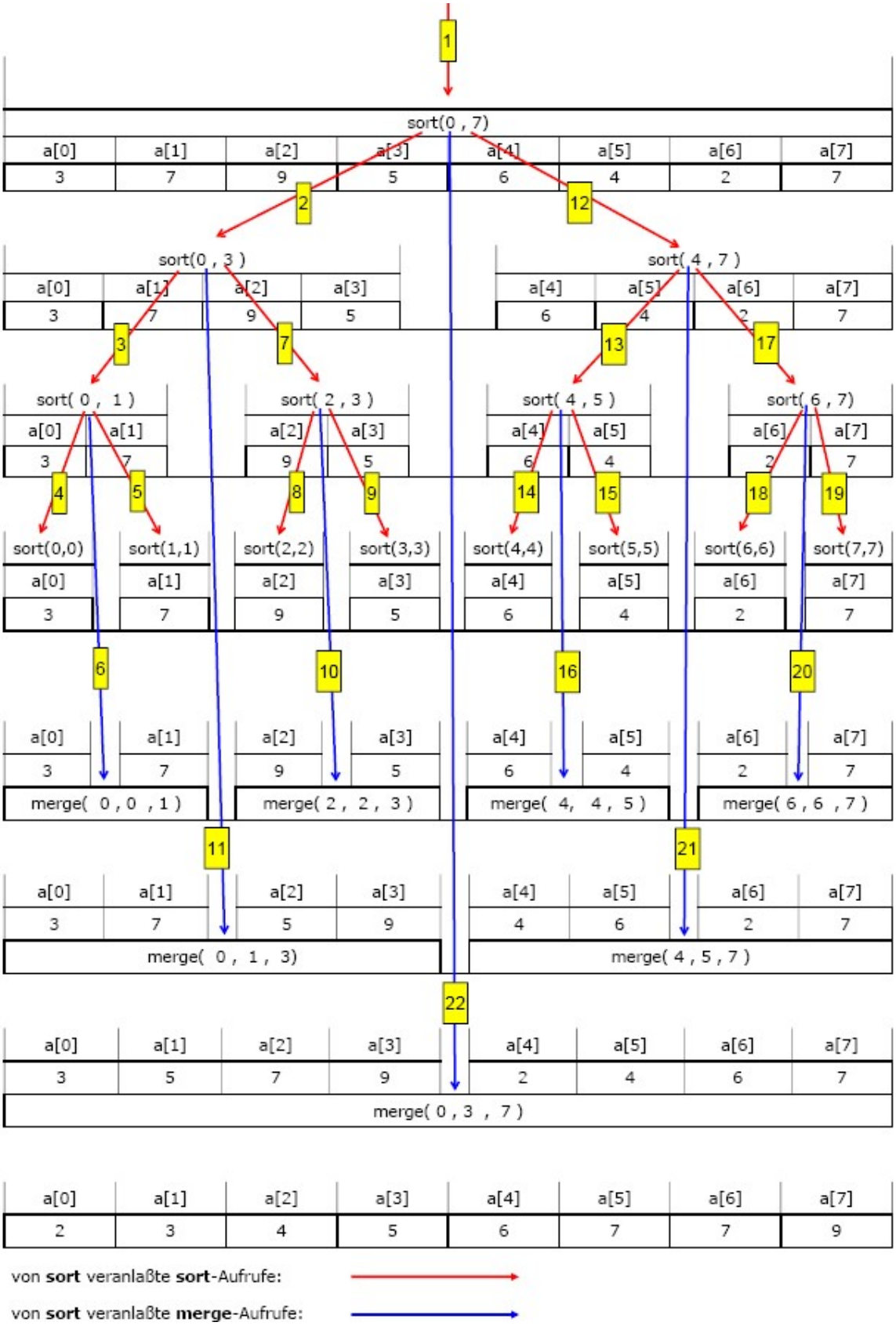
Quelltext

```
# MergeSort
# Ausgabe der Reihenfolge der Funktionsaufrufe
from random import randint
z = 0
y = 0
n = int(input('Laenge des arrays: '))
print()

# Erzeugen des arrays mit dem Namen a
# und den n Komponenten a[0], . . . , a[n-1]
a = list(range(0,n))

# Zuweisung von Zufallszahlen an die Komponenten des arrays a
for i in range(0,n): a[i] = randint(0,99)

# Ausgabe der Quelliste
anzahl = int(input('Wieviele Elemente sollen angezeigt werden? '))
print()
for i in range(0,anzahl): print(a[i])
print()

def merge(array, left, middle, right):
    global y
    y += 1
    left_sublist = array[left:middle + 1]
    right_sublist = array[middle+1:right+1]
    left_sublist_index = 0
    right_sublist_index = 0
    sorted_index = left
    while left_sublist_index < len(left_sublist) and right_sublist_index < len(right_sublist):
        if left_sublist[left_sublist_index] <= right_sublist[right_sublist_index]:
            array[sorted_index] = left_sublist[left_sublist_index]
            left_sublist_index = left_sublist_index + 1
        else:
            array[sorted_index] = right_sublist[right_sublist_index]
            right_sublist_index = right_sublist_index + 1
        sorted_index = sorted_index + 1
    while left_sublist_index < len(left_sublist):
        array[sorted_index] = left_sublist[left_sublist_index]
        left_sublist_index = left_sublist_index + 1
        sorted_index = sorted_index + 1
    while right_sublist_index < len(right_sublist):
        array[sorted_index] = right_sublist[right_sublist_index]
        right_sublist_index = right_sublist_index + 1
        sorted_index = sorted_index + 1

def sort(array, left, right):
    global z
    z +=1
    if left == right: return
    middle = (left + right)//2
    print('sort(',left,',',middle,')')
    sort(array, left, middle)
    print('sort(',middle + 1,',',right,')')
    sort(array, middle + 1, right)
    print('merge(',left,',',middle,',',right,')')
    merge(array, left, middle, right)

l = 0
r = len(a)-1
print('sort(',l,',',r,')')
sort(a, l, r)

print()
print('Sortierte Liste:')
print()
for i in range(0,anzahl): print(a[i])

print()
print('# Aufrufe sort:  ',z)
print('# Aufrufe merge: ',y)
```

Durchführung von MergeSort und Auflistung der Aufrufe der Funktionen **`sort`** und **`merge`** für eine aus den 8 Komponenten **`a[0], . . . , a[7]`** bestehende Liste **`a`**:

```
Wieviele Elemente sollen angezeigt werden? 8

89
37
31
0
19
86
33
1

sort( 0 , 7 )
sort( 0 , 3 )
sort( 0 , 1 )
sort( 0 , 0 )
sort( 1 , 1 )
merge( 0 , 0 , 1 )
sort( 2 , 3 )
sort( 2 , 2 )
sort( 3 , 3 )
merge( 2 , 2 , 3 )
merge( 0 , 1 , 3 )
sort( 4 , 7 )
sort( 4 , 5 )
sort( 4 , 4 )
sort( 5 , 5 )
merge( 4 , 4 , 5 )
sort( 6 , 7 )
sort( 6 , 6 )
sort( 7 , 7 )
merge( 6 , 6 , 7 )
merge( 4 , 5 , 7 )
merge( 0 , 3 , 7 )

Sortierte Liste:

0
1
19
31
33
37
86
89

# Aufrufe sort:   15
# Aufrufe merge:  7
```

Baumstruktur mit Reihenfolge für die Funktionsaufrufe:



08.12.2023