SelectionSort

Aufgabenstellung:

Gegeben ist ein Array **a** mit den **n** Komponenten **a[0], a[2], ... , a[n-1]** als Datenelemente, für die die Ordnungsrelationen **<** , **>** , **≤**, **≥** erklärt sind (also Komponenten z. B. vom Typ `integer`, `char` oder `string`). Die Inhalte dieser Datenelemente sind aufsteigend so anzuordnen, daß gilt:

a[0] ≤ a[2] ≤ . . . . . ≤ a[n-1] .

In Python läßt sich ein Array **a** als Liste realisieren.

Sortieren durch direkte Auswahl („SelectionSort")

Bei diesem Verfahren handelt es um einen typischen Vertreter eines imperativ formulierten Algorithmus'.

Der Algorithmus **SelectionSort** bestimmt

- das kleinste Element (Minimum) der Liste a[0], a[1], ... , a[n-1] und weist dieses der Komponente a[0] zu, dabei wird der Inhalt von a[0] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- das kleinste Element (Minimum) der Liste a[1], ... , a[n-1] und weist dieses der Komponente a[1] zu, dabei wird der Inhalt von a[1] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- das kleinste Element (Minimum) der Liste a[2], ... , a[n-1] und weist dieses der Komponente a[2] zu, dabei wird der Inhalt von a[2] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- . . . . . .
- . . . . . .
- das kleinste Element (Minimum) der Liste a[n-2], a[n-1] und weist dieses der Komponente a[n-2] zu, dabei wird der Inhalt von a[n-2] der Komponente a[n-1] zugewiesen.

Nach dem Abarbeiten der vorgenannten **n-1** Schritte ist das Array **a** aufsteigend sortiert.

In Python lassen sich die ersten vier Schritte wie folgt formulieren:

```
min = a[0]
i = 0 + 1
while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[0]
          a[0] = min
     i = i + 1

min = a[1]
i = 1 + 1
while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[1]
          a[1] = min
     i = i + 1
```

```
min = a[2]
i = 2 + 1
while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[2]
          a[2] = min
     i = i + 1

min = a[3]
i = 3 + 1
while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[3]
          a[3] = min
     i = i + 1
```

Letzter Schritt:

```
min = a[n-2]
i = n-2 + 1
while i < n:
     if a[i] < min:
          min    = a[i]
          a[i]   = a[n-2]
          a[n-2] = min
     i = i + 1
```

Zusammenfassend gilt: Der Anweisungsblock

```
min = a[j]
i = j + 1
while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[j]
          a[j] = min
     i = i + 1
```

ist nacheinander für j = 0, 1, 2, ... , n-2 abzuarbeiten; folglich fassen wir diesen Block als Schleifenrumpf einer weiteren Schleife (hier: for-Schleife) mit Schleifenindex j auf:

```
for j in range(0,n-1):
   min = a[j]
   i = j + 1
   while i < n:
        if a[i] < min:
             min  = a[i]
             a[i] = a[j]
             a[j] = min
        i = i + 1
```

Alternativ können wir die äußere Schleife als while-Schleife formulieren:

```
j = 0
while j <= n - 2:
   min = a[j]
   i = j + 1
   while i < n:
        if a[i] < min:
             min  = a[i]
             a[i] = a[j]
             a[j] = min
        i = i + 1
   j = j + 1
```

Das folgende Python-Programm

- weist nach Eingabe von **n** den Komponenten der Liste **a** Zufallszahlen aus dem Bereich 1, . . . , 1000000 zu,
- sortiert diese Liste **a** aufsteigend,
- ermittelt den Zeitbedarf für das Sortieren der **n** Datenelemente,
- gibt jeweils einen Teil der Quelliste und der sortierten Liste sowie den Zeitaufwand für den Sortiervorgang (in s) aus.

```
# SelectionSort

n = int(input('Anzahl der Datenelemente = '))
r = int(input('Wieviele Elemente sollen angezeigt werden? '))

a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten der Liste a
for i in range(0,n):
    a[i]= randint(1,1000000)

# Ausgabe der Quelliste:
for i in range(0,r):
    print(a[i])

# Sortieren der Quelliste:

start = time.time()

j = 0
while j <= n-2:
   min = a[j]
   i = j + 1
   while i < n:
      if a[i] < min:
           min  = a[i]
           a[i] = a[j]
           a[j] = min
      i = i + 1
   j = j + 1

end = time.time()

# Ausgabe der sortierten Liste:
print()
print('sortierte Liste:')
for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f} s'.format(end-start))
```

Aufwandsbetrachtung

Wir untersuchen den Algorithmus **SelectionSort** hinsichtlich seiner zeitlichen Komplexität, d. h. wir untersuchen, wie der Zeitbedarf zur Laufzeit sich in Abhängigkeit von der Anzahl n der zu sortierenden Datensätze verhält. Den Aufwand hinsichtlich des Speicherplatzbedarfs können wir hier vernachlässigen, da der Algorithmus SelectionSort auf dem Array **a** operiert und keinen weiteren Speicherplatz zur Laufzeit benötigt.

Hierzu betrachten wir denjenigen Programmteil, der das Sortieren ausführt:

```
j = 0
while j <= n-2:
    min = a[j]
    i = j + 1
    while i < n:
        if a[i] < min:
            min  = a[i]
            a[i] = a[j]
            a[j] = min
        i = i + 1
    j = j + 1
```

```
j = 0
while j <= n-2:
    min = a[j]
    i = j + 1
    while i < n:
        A
    j = j + 1
```

Wir fassen die Anweisungen aus dem Schleifenrumpf der inneren Schleife (hier: rot markiert) dieses Programmauszugs gedanklich zum Anweisungsblock **A** zusammen.
Um den Aufwand zu ermitteln, ein aus **n** Komponenten bestehendes array zu sortieren, fragen wir, wie oft Block **A** in Abhängigkeit von **n** abgearbeitet wird.

In folgender Tabelle gibt **z(j)** jeweils an, wie oft Block **A** in Abhängigkeit von **j** abgearbeitet wird.

Index j	Index i	z(j)
j = 0	1 ≤ i ≤ n-1	n - 1
j = 1	2 ≤ i ≤ n-1	n - 2
j = 2	3 ≤ i ≤ n-1	n - 3
j = 3	4 ≤ i ≤ n-1	n - 4
....	....	....
j = n-3	n-2 ≤ i ≤ n-1	2
j = n-2	n-1 ≤ i ≤ n-1	1

Für die Gesamtanzahl **z** der Abarbeitungen von Block **A** erhalten wir:

$$
\begin{aligned}
\mathbf{z} &= z(0) + z(1) + z(2) + z(3) + \ldots\ldots + z(n-3) + z(n-2) \\
&= (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots + 2 + 1 \\
&= 1 + 2 + \ldots\ldots\ldots + n-1 \\
&= \tfrac{1}{2} \cdot (n-1) \cdot n \quad \textit{(beachte untenstehenden Hinweis)} \\
&= \tfrac{1}{2} \cdot (n^2 - n) \\
&= \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n
\end{aligned}
$$

Für große Werte von n können wir den Summand ½ · n gegenüber dem Summand ½ · n^2 vernachlässigen; somit folgt:

z $\approx \frac{1}{2} \cdot n^2$

z $\sim$ **n^2**

Bei SelectionSort wächst der Zeitbedarf proportional zum Quadrat der Anzahl n der zu sortierenden Datenelemente.

SelectionSort ist von polynomialer (hier: quadratischer) Komplexität.

Hinweis:
Für die Summe der ersten n natürlichen Zahlen gilt:

$$1 + 2 + \ldots\ldots + n = \frac{1}{2} \cdot n \cdot (n + 1)$$

Aufgaben:

1. Bestätige die quadratische Komplexität von SelectionSort experimentell anhand geeigneter Testläufe.
2. Modifiziere den Quelltext so, daß SelectionSort absteigend sortiert.
3. Sobald in der Teilliste `a[j], . . . , a[n-1],` `0 ≤ j ≤ n-2,` ein Element gefunden wird, welches kleiner ist als das jeweils aktuelle Minimum, werden die Wertzuweisungen innerhalb des Blocks **`A`** ausgeführt, was für ein bestimmtes `j` ggf. auch mehrmals erfolgt. Optimiere den Algorithmus so, daß die Wertzuweisungen jeweils höchstens ein Mal für jeden Wert von `j` vorgenommen werden.
   Bestimme experimentell die Laufzeit und bestätige die (insgesamt bescheidene) Optimierung.
   Hinweis: Ermittle zunächst den Index derjenigen Komponente, welche den kleinsten Inhalt innerhalb der Liste ***`a[j], . . , a[n-1]`*** *hat, und führe anschließend einmalig die Wertzuweisungen des Blocks* ***`A`*** *aus.*

Komplexität von Algorithmen
A(n) bezeichne den Aufwand und damit den Zeitbedarf zur Laufzeit in Abhängigkeit von **n** (z. B. n = Anzahl der zu verarbeitenden Datenelemente).

Algorithmus	Aufwand	Art der Komplexität
sequentielle oder lineare Suche	**A(n) ~ n**	linear
binäre Suche	**A(n) ~ $\log_2$(n)**	logarithmisch
SelectionSort	**A(n) ~ n^2**	polynomial (hier: quadratisch)
MergeSort	**A(n) ~ n · $\log_2$(n)**	linear-logarithmisch
Fibonacci-Folge (rekursiv)	**A(n) ~ 2^n**	exponentiell
Ackermann-Funktion	**A(3,n) ~ $2^{n+3} - 3$** **A(3,n) ~ 2↑(n+3) – 3** **A(4,n) ~ 2↑↑(n+3) – 3** **A(5,n) ~ 2↑↑↑(n+3) – 3**	exponentiell hyper-exponentiell

Algorithmen mit exponentieller Komplexität erweisen sich in der Praxis als unbrauchbar; selbst Algorithmen mit polynomialer Komplexität zeigen häufig ein ungünstiges Laufzeitverhalten.

03.07.2023

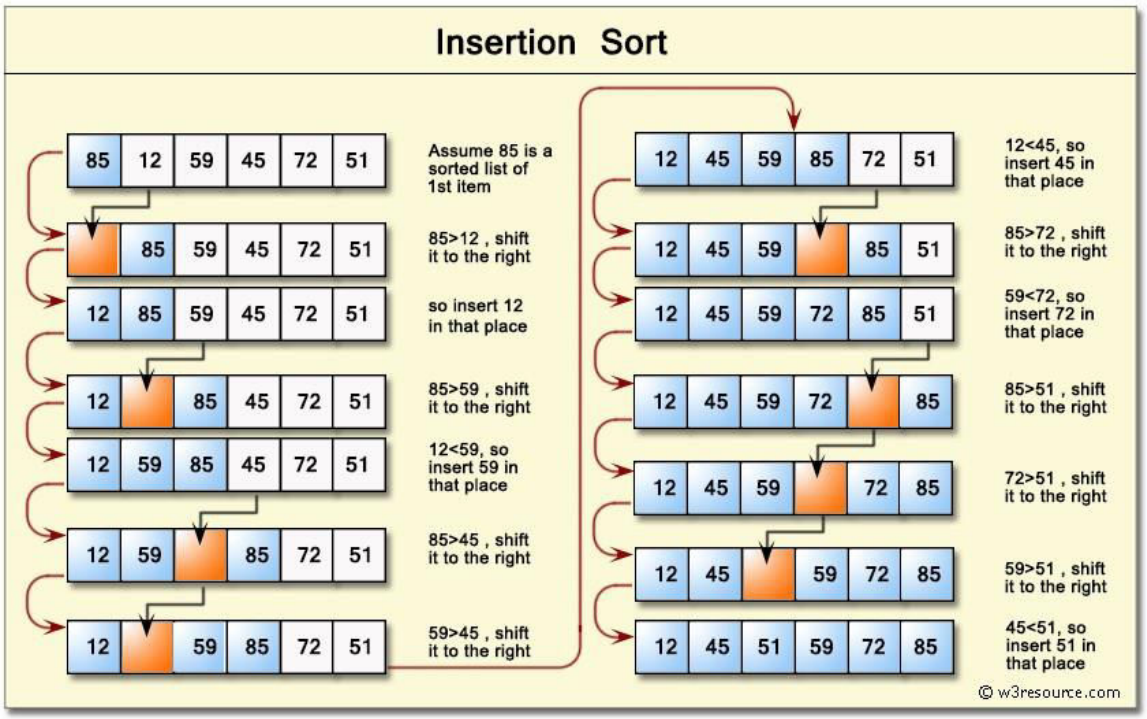
Entwicklung eines Algorithmus InsertionSort

Zu einer natürlichen Zahl n ist ein Array **a** mit den **n** Komponenten **a[0], . . . , a[n-1]** gegeben (in Python läßt sich ein Array als Liste definieren), für die die Operationen **=** , **<** und **>** definiert sind.

Ziel: Die Inhalte der Komponenten sind gemäß dem Algorithmus „Sortieren durch direktes Einfügen" (InsertionSort) so anzuordnen, daß gilt:

a[0] ≤ a[1] ≤ . . . ≤ a[n-1]

Beispiel (n = 6):



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
85	12	59	45	72	51

Die aus der Komponente **a[0]** bestehende 1-elementige Teilliste gilt als sortiert, die aus den Komponenten **a[1] , . . . , a[n-1]** bestehende Teilliste ist zu Anfang unsortiert.

Wir verwenden die Variable `current` als temporäre Variable.

1. Schritt:

```
current = a[1]
i = 1 - 1
if current < a[i]:
        a[i+1] = a[i]
        a[i]   = current
```

Ergebnis des 1. Schritts:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
12	85	59	45	72	51

Die aus den Komponenten **a[0], a[1]** bestehende Teilliste ist sortiert, der Bereich **a[2] , . . . , a[5]** unsortiert.

2. Schritt:

```
current = a[2]
i = 2 - 1
while i >= 0:
   if current < a[i]:
         a[i+1] = a[i]
         a[i]   = current
   i = i - 1
```

Ergebnis des 2. Schritts:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
12	59	85	45	72	51

Die aus den Komponenten **a[0], a[1], a[2]** bestehende Teilliste ist sortiert, der Bereich **a[3] , . . . , a[5]** unsortiert.

3. Schritt:

```
current = a[3]
i = 3 - 1
while i >= 0:
   if current < a[i]:
         a[i+1] = a[i]
         a[i]   = current
   i = i - 1
```

Ergebnis des 3. Schritts:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
12	45	59	85	72	51

Die aus den Komponenten **a[0], . . . , a[3]** bestehende Teilliste ist sortiert, der Bereich **a[4] , a[5]** unsortiert.

4. Schritt:

```
current = a[4]
i = 4 - 1
while i >= 0:
   if current < a[i]:
         a[i+1] = a[i]
         a[i]   = current
   i = i - 1
```

Ergebnis des 4. Schritts:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
12	45	59	72	85	51

Die aus den Komponenten **a[0], . . . , a[4]** bestehende sortierte Teilliste ist mit der Komponente **a[5]** zu einer sortierten Gesamtliste zu verschmelzen.

5. Schritt:

```
current = a[5]
i = 5 - 1
while i >= 0:
    if current < a[i]:
          a[i+1] = a[i]
          a[i]   = current
    i = i - 1
```

Ergebnis des 5. Schritts:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
12	45	51	59	72	85

Bei dem gewählten Beispiel (n = 6) ist der Anweisungsblock

```
current = a[j]
i = j - 1
while i >= 0:
    if current < a[i]:
          a[i+1] = a[i]
          a[i]   = current
    i = i - 1
```

für j = 1, 2, . . . , 5 zu wiederholen.

Allgemein halten wir fest:

Die zu sortierende Gesamtliste besteht vor jedem Schritt aus einer bereits sortierten Teilliste und einer unsortierten Teilleiste; vor dem ersten Schritt ist die aus dem einen Element a[0] bestehende Liste sortiert und die Liste a[1] , . . , a[n-1] unsortiert. Nachfolgend wird das jeweils erste Element der unsortierten Teilliste an der richtigen Stelle in die sortierte Teilliste eingefügt, so daß der sortierte Bereich mit jedem Schritt wächst, bis die gesamte Liste sortiert ist.

Falls das Array **a** aus den **n** Komponenten **a[0] , . . . , a[n-1]** besteht, ist der Anweisungsblock

```
current = a[j]
i = j - 1
while i >= 0:
    if current < a[i]:
          a[i+1] = a[i]
          a[i]   = current
    i = i - 1
```

nacheinander für j = 1, . . . , n–1 zu wiederholen. Folglich implementieren wir diesen Anweisungsblock als Schleifenrumpf einer geeignet initialisierten `for`- oder `while`-Schleife.

Vorbemerkung:
Gegeben ist ein Array **a** mit den **n** Komponenten **a[0], a[2], ... , a[n-1]** als Datenelemente, für die die Ordnungsrelationen **<** , **>** , **=** erklärt sind (also Komponenten z. B. vom Typ integer, char oder string). Die Inhalte dieser Datenelemente sind aufsteigend so anzuordnen, daß gilt:

$$a[0] \leq a[2] \leq \ldots\ldots \leq a[n-1].$$

In Python läßt sich ein Array **a** als Liste realisieren.

1. Aufgabe
Optimierung des Algorithmus SelectionSort (Sortieren durch direkte Auswahl)

Der in Python geschriebene Quelltext *SelectionSort_for-loop_while-loop_time.py* enthält eine Uhr, um den Zeitbedarf (in s) zum Sortieren eines aus n Komponenten bestehenden Arrays zu ermitteln.
Der folgende Programmteil führt den Sortiervorgang aus:

```
for j in range(0,n-1):
  min = a[j]
  i = j + 1
  while i < n:
     if a[i] < min:
          min  = a[i]
          a[i] = a[j]
          a[j] = min
     i = i + 1
```

Sobald in der Teilliste **a[j], . . . , a[n-1],** $0 \leq j \leq n-2$, ein Element gefunden wird, welches kleiner ist als das jeweils aktuell definierte Minimum **min**, wird der „swap" (Tausch der Werte zweier Variabler; hier: die Wertzuweisungen innerhalb des Schleifenrumpfs der inneren Schleife) ausgeführt, was für ein bestimmtes **j** ggf. auch mehrmals erfolgt. Optimiere den Algorithmus so, daß die Wertzuweisungen jeweils höchstens ein Mal für jeden Wert von **j** vorgenommen werden.
Bestimme experimentell die Laufzeit und bestätige die (insgesamt bescheidene) Optimierung. Zeige ferner empirisch, daß die Zeitkomplexität von der Ordnung n^2 ist (d. h.: Der Zeitbedarf wächst mit dem Quadrat der Anzahl n der zu sortierenden Datenelemente; Schreibweise: **$O(n^2)$**).
*Hinweis: Ermittle zunächst den Index derjenigen Komponente, welche den kleinsten Inhalt innerhalb der Liste **a[j], . . , a[n-1]** hat, und führe anschließend den swap höchstens einmalig aus.*

2. Aufgabe
Algorithmus InsertionSort (Sortieren durch direktes Einfügen)

a) Erstelle, ausgehend von dem Skript *InsertionSort_11-09-2023.pdf*, einen in Python geschriebenen Quelltext zu InsertionSort.

b) Teste das Programm anhand diverser Eingaben.

c) Implementiere eine Uhr im Quelltext (wegen der Syntax orientiere man sich an *SelectionSort_for-loop_while-loop_time.py*) und bestätige empirisch, daß die Zeitkomplexität von der Ordnung $O(n^2)$ ist.

3. Aufgabe
Optimierungen des Algorithmus InsertionSort (Sortieren durch direktes Einfügen)

a) Der folgende Programmauszug veranlaßt das Sortieren des Arrays **a**, indem das jeweils erste Element `a[j]`, $j \in \{1, \ldots, n-1\}$, der noch unsortierten Teilliste an der richtigen Stelle der bereits sortierten Teilliste eingefügt wird:

```
j = 1

while j <= n - 1:
  current = a[j]
  i = j - 1
  while i >= 0:
     if current < a[i]:
           a[i+1] = a[i]
           a[i]   = current
     i = i - 1
  j = j + 1
```

Begründe anhand eines Beispiels, daß dieser Programmcode nicht optimal ist. Modifiziere den Quelltext und zeige empirisch die Effizienzverbesserung, wobei die quadratischen Zeitkomplexität allerdings erhalten bleibt.

Zeige ferner, daß der modifizierte Algorithmus Vorteile bitte, falls das Array **a** bereits in Teilen oder vollständig vorsortiert ist.

Bemerkung:
*Tatsächlich ist die Zeitkomplexität von der Ordnung O(n) (d. h., die Laufzeit wächst im wesentlichen linear mit der Anzahl n), falls die Liste **a** bereits sortiert ist; diesen Vorteil beobachtet man bei SelectionSort nicht.*

b) Delegiere das Einfügen des jeweils ersten Elements `a[j]`, $j = 1, \ldots, n-1$, der noch unsortierten Teilliste an die richtige Stelle der bereits sortierten Teilliste an ein Unterprogramm (Prozedur; in Python: definiere in geeigneter Weise eine Funktion).

Zeige, daß sich mit dieser Modifikation eine weitere Verbesserung der Laufzeit erzielen läßt (zumindest in Python); versuche, eine Erklärung zu geben.

Bemerkung: Da hier das Merkmal der Rekursion fehlt, bleibt der Algorithmus auch nach der Implementierung der Funktion „insert" imperativ formuliert.

Lösung zu a):

```
j = 1
while j <= n - 1:
  current = a[j]
  i = j - 1
  while i >= 0 and current < a[i]:
          a[i+1] = a[i]
          a[i]   = current
          i = i - 1
  j = j + 1
```

Lösung zu b):

```
def insert(x):
   current = a[x]
   i = x - 1
   while i >= 0 and current < a[i]:
           a[i+1] = a[i]
           a[i]   = current
           i -= 1

j = 1
while j <= n - 1:
  insert(j)
  j += 1
```

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:

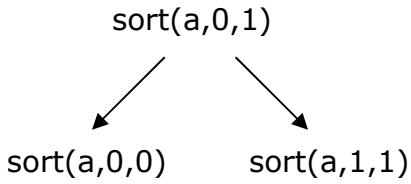
Mit **f(n)** bezeichnen wir die Anzahl der gleichzeitig aktiven Aufrufe der Funktion **`sort`**, wenn eine Liste mit **`n`** Datenelementen zu sortieren ist.

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"*

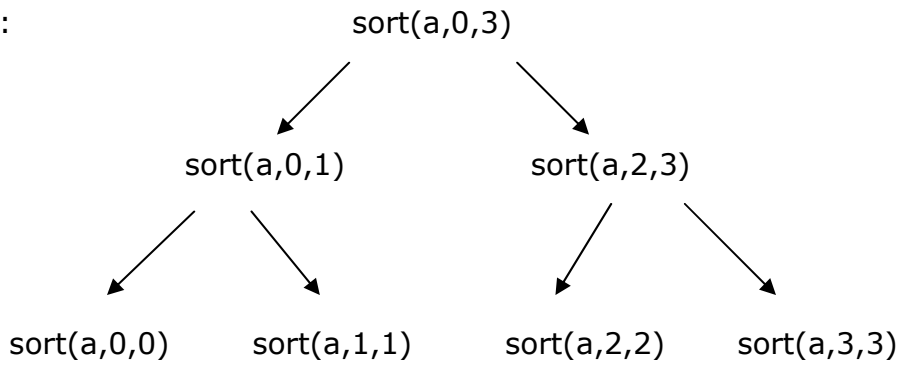
n = 1: sort(a,0,0) 1 Aufruf

n = 2:



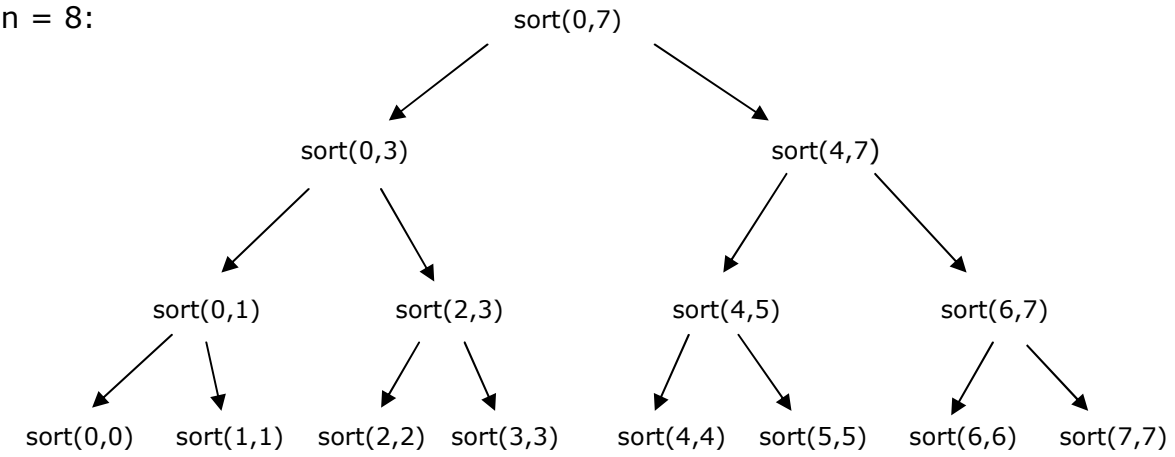
$1 + 2 \cdot 1 = 3$ Aufrufe

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

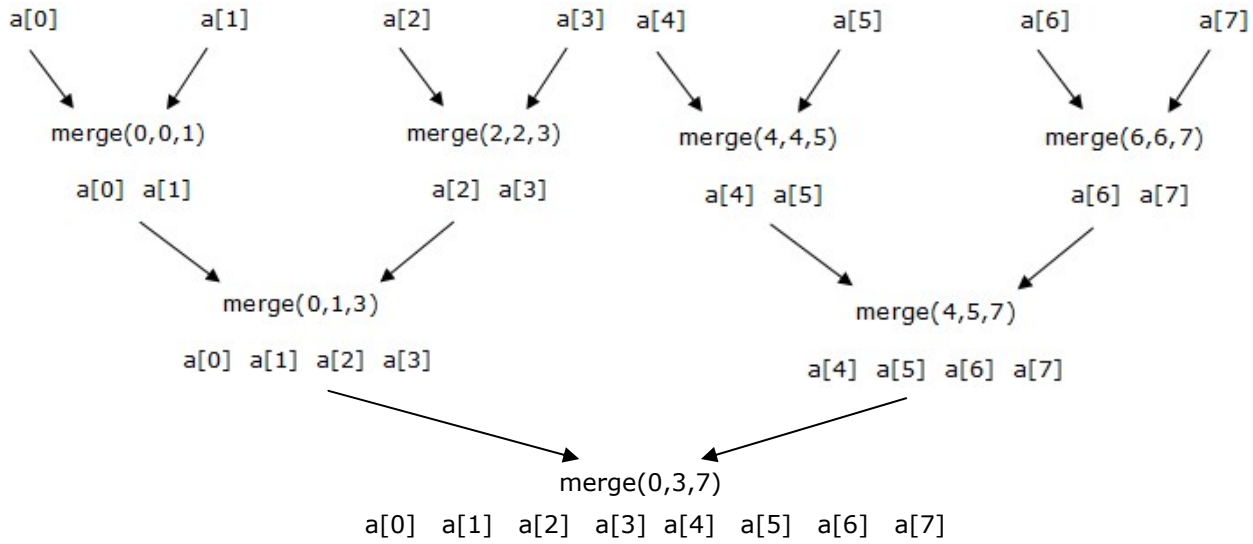
$f(n) = 1 + 2 \cdot f(n/2)$

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

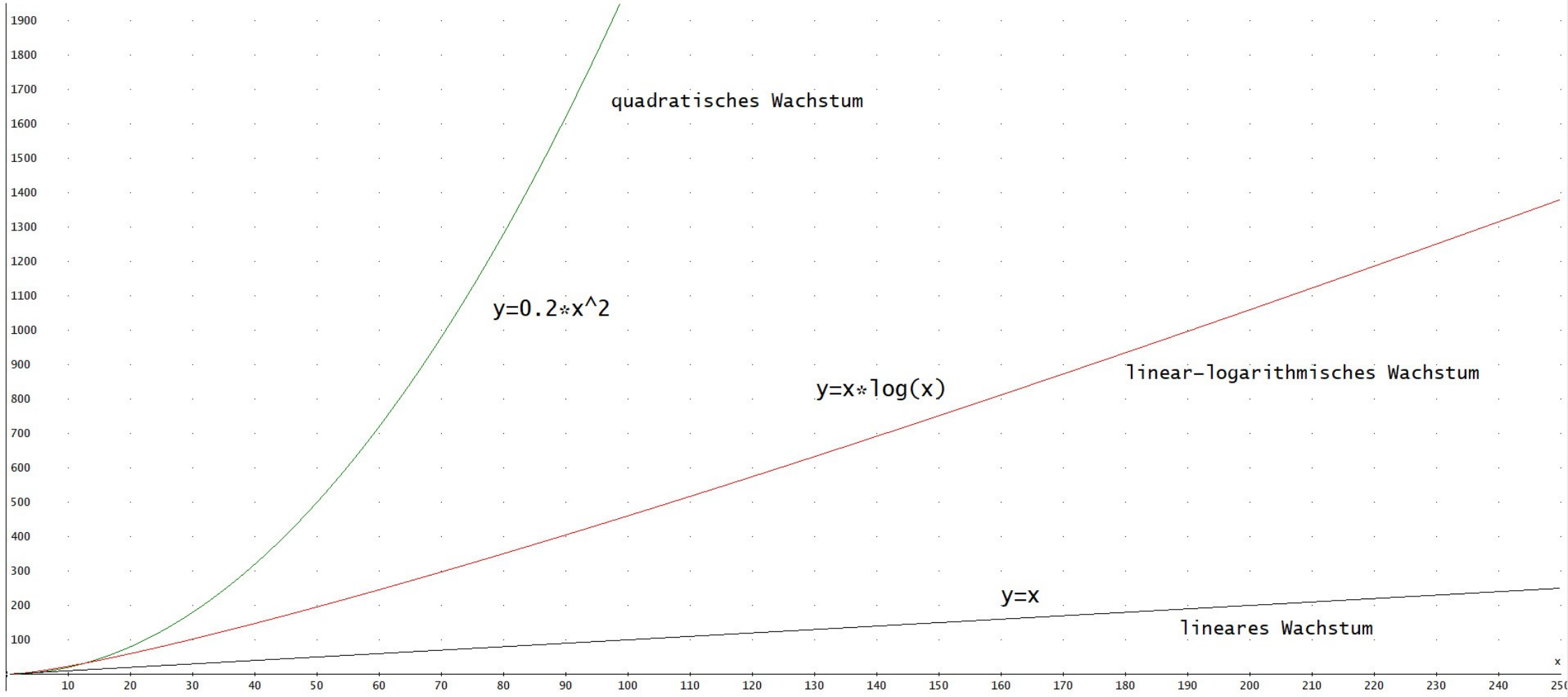
Bemerkung:
Für den Aufwand A(n) und folglich den Zeitbedarf zur Laufzeit des Algorithmus erhalten wir bei

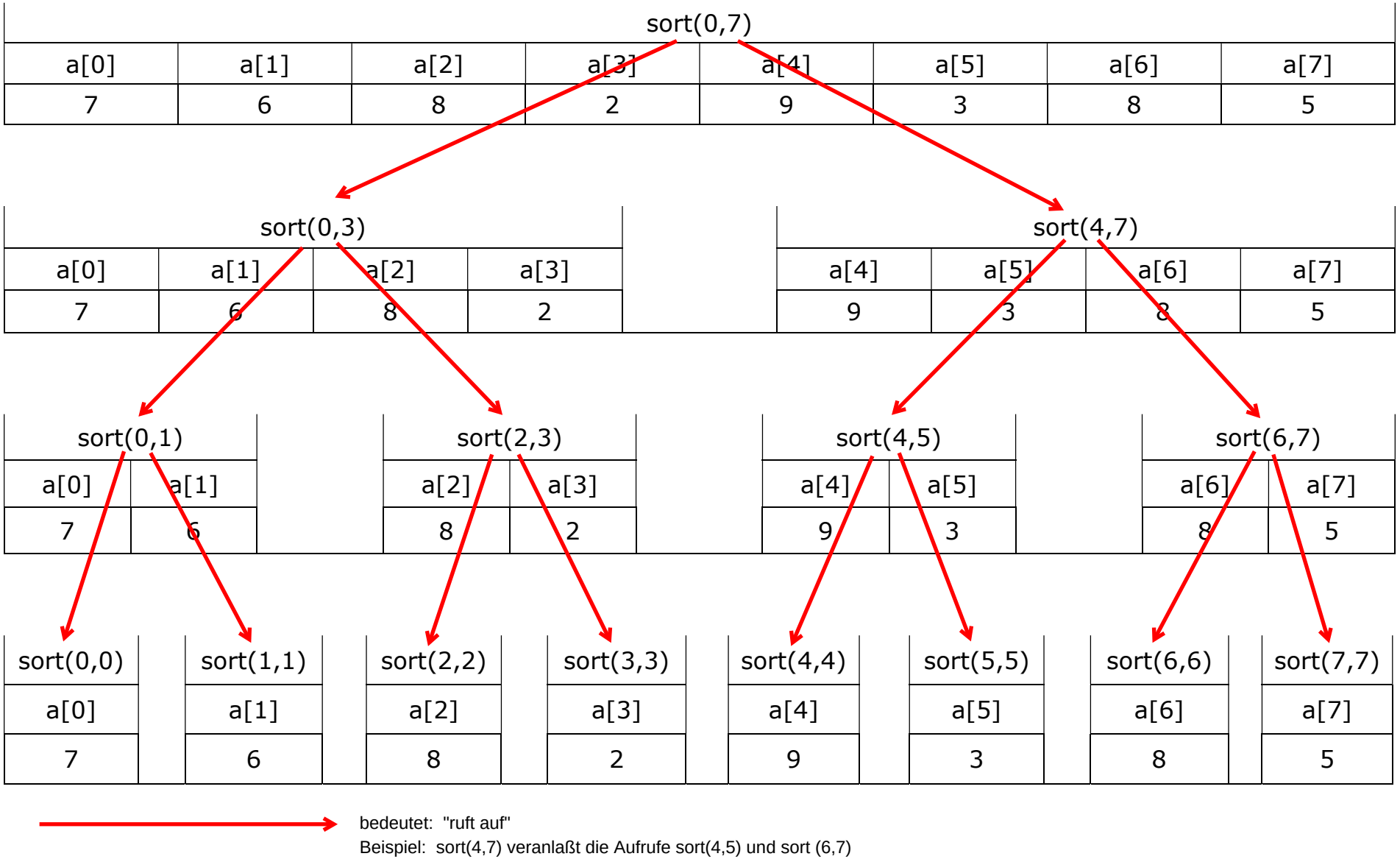
- SelectionSort: $A(n) \sim n^2$
- MergeSort: $A(n) \sim n \cdot \log_2(n)$
- Fibonacchi-Folge: $A(n) \sim 2^n$ (bei rekursiver Berechnung)
- BinarySearch: $A(n) \sim \log_2(n)$

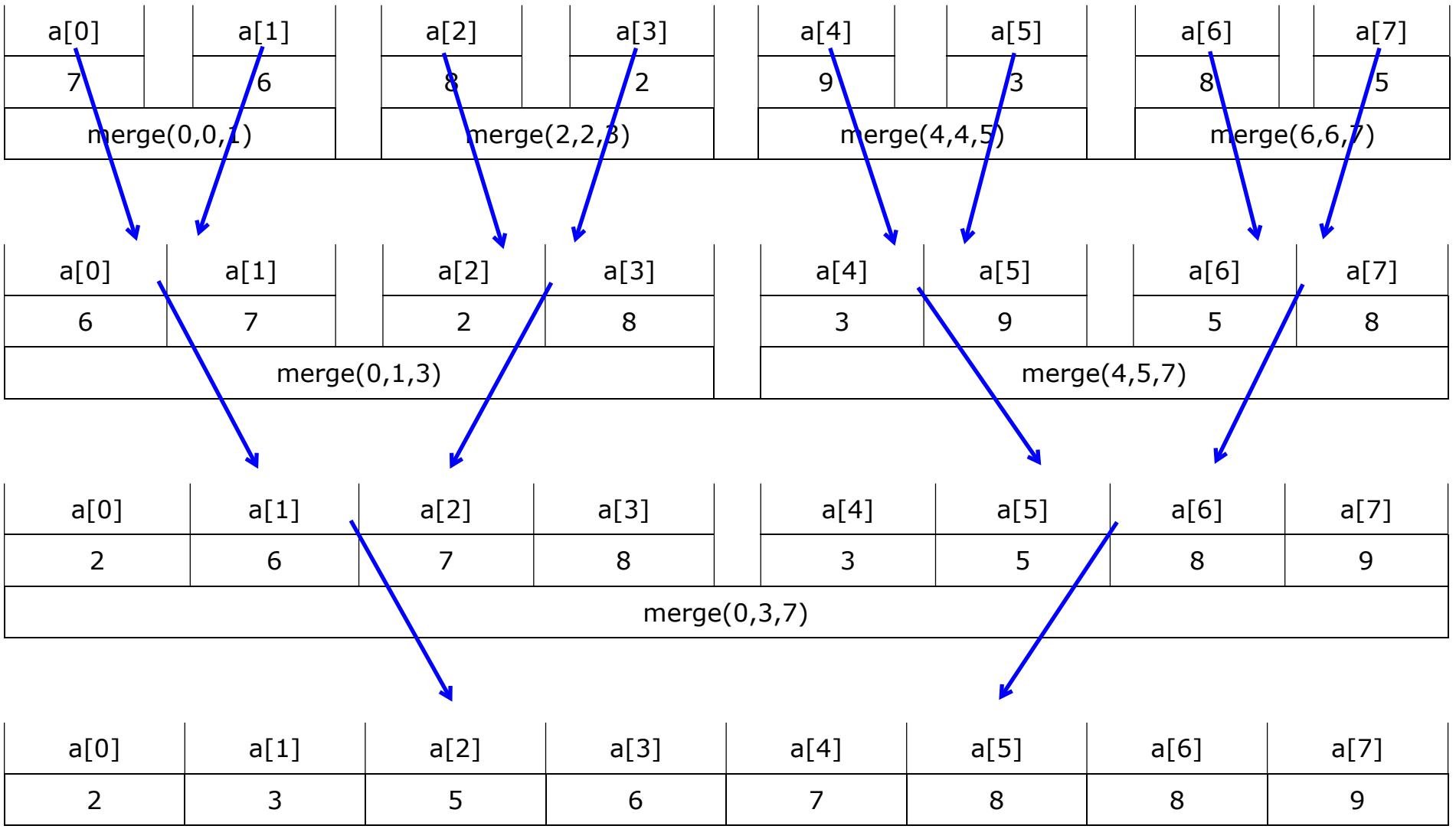
Entsprechend haben

- 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


sort(0,7)
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
7 6 8 2 9 3 8 5
sort(0,3)
a[0] a[1] a[2] a[3]
7 6 8 2
sort(4,7)
a[4] a[5] a[6] a[7]
9 3 8 5
sort(0,1)
a[0] a[1]
7 6
sort(2,3)
a[2] a[3]
8 2
sort(4,5)
a[4] a[5]
9 3
sort(6,7)
a[6] a[7]
8 5
sort(0,0) a[0] 7
sort(1,1) a[1] 6
sort(2,2) a[2] 8
sort(3,3) a[3] 2
sort(4,4) a[4] 9
sort(5,5) a[5] 3
sort(6,6) a[6] 8
sort(7,7) a[7] 5
bedeutet: "ruft auf"
Beispiel: sort(4,7) veranlaßt die Aufrufe sort(4,5) und sort (6,7)



bedeutet: "wird gemischt"

Beispiel: merge(0,3,7) mischt die sortierten Listen a[0], . . , a[3] und a[4], . . , a[7] zu der sortierten Liste a[0], . . . , a[7]

Zeitkomplexität von Algorithmen

Verdeutlichung der O-Notation anhand eines Beispiels
Der Zeitbedarf A(n) von SelectionSort in Abhängigkeit von der Anzahl n der zu verarbeitenden Datenelemente („Problemgröße") wächst quadratisch für große Werte von n:
$A(n) \sim n^2$ für große n.
Man sagt auch: Die Zeitkomplexität von SelectionSort ist von der Ordnung $O(n^2)$.

Algorithmus	**lineare Suche** **Fakultät (rekursiv oder iterativ)**	**binäre Suche auf einer sortierten Menge**	**Selection-Sort** **Insertion-Sort**	**MergeSort**	**Fibonacci-Folge (rekursiv)** **Türme von Hanoi**	**Ackermann-Funktion (rekursiv)**
Komplexität	**$O(n)$**	**$O(\log_2 n)$**	**$O(n^2)$**	**$O(n \cdot \log_2 n)$**	**$O(2^n)$**	
Art des Wachstums	linear	logarithmisch	polynomial hier: quadratisch	linear-logarithmisch	exponentiell	hyper-exponentiell

Algorithmen mit polynomialer Komplexität sind bedingt brauchbar, Algorithmen mit exponentieller Komplexität erweisen sich in der Praxis als unbrauchbar.

Rechenzeiten in Abhängigkeit von der Zeitkomplexität des Algorithmus

Annahme:
Für die Verarbeitung des jeweiligen Problems mit minimaler Problemgröße (n = 1) werde ein Zeitbedarf von 1 μs = 10^{-6} s angesetzt.

Komplexität	**n = 1**	**n = 100**	**n = 10^3**	**n = 10^4**	**n = 10^6**	**n = 10^9**
$O(n)$	10^{-6} s	10^{-4} s	10^{-3} s	10^{-2} s	1 s	10^3 s ≈ 17 min
$O(\log_2 n)$	10^{-6} s	$7 \cdot 10^{-6}$ s	$10 \cdot 10^{-6}$ s	$13 \cdot 10^{-6}$ s	$20 \cdot 10^{-6}$ s	$30 \cdot 10^{-6}$ s
$O(n \cdot \log_2 n)$	10^{-6} s	$7 \cdot 10^{-4}$ s	10^{-2} s	0,13 s	20 s	30 000 s ≈ 8 h
$O(n^2)$	10^{-6} s	10^{-2} s	1 s	100 s	10^6 s ≈ 12 d	10^{12} s ≈ 31 700 a
$O(2^n)$	10^{-6} s	$1{,}3 \cdot 10^{24}$ s ≈ $4 \cdot 10^{16}$ a	10^{295} s ≈ $3{,}4 \cdot 10^{287}$ a	$2 \cdot 10^{3004}$ s		

Rechenzeit bei exponentieller Zeitkomplexität:

Komplexität	**n = 1**	**n = 10**	**n = 20**	**n = 40**	**n = 50**	**n = 60**
$O(2^n)$	10^{-6} s	0,001 s	1,05 s	$1{,}1 \cdot 10^6$ s ≈ 12,7 d	$1{,}1 \cdot 10^9$ s ≈ 35,7 a	$1{,}2 \cdot 10^{12}$ s ≈ 36 600 a

Alter des Universums: 13,8 Milliarden Jahre = $13{,}8 \cdot 10^9$ a = $4{,}35 \cdot 10^{17}$ s

MergeSort

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

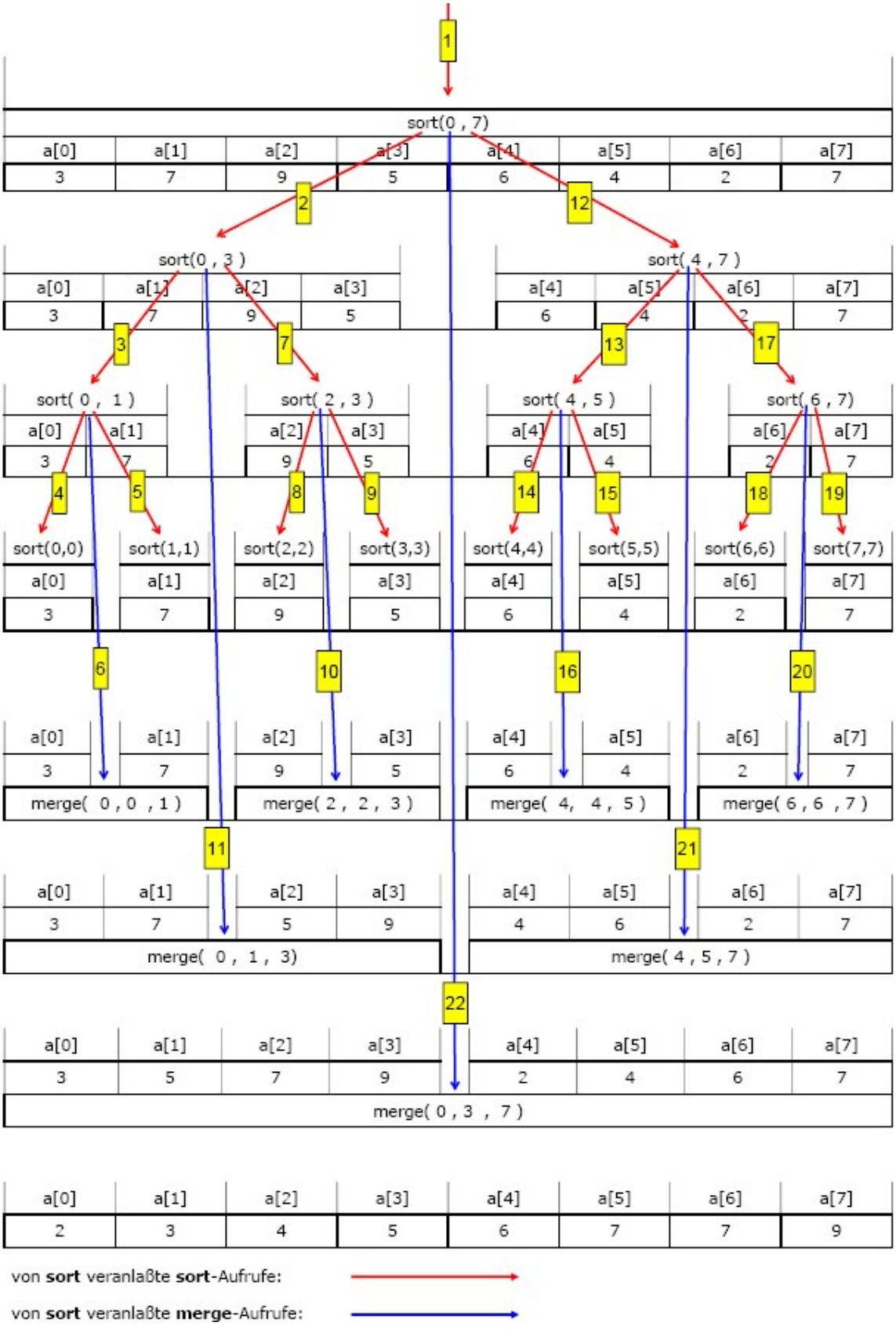
Quelltext

```
# 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