10. Die Hofstadter-Funktion ist rekursiv definiert (n natürliche Zahl):

    Rekursionsanfang: hof(1) = 1
    hof(2) = 1

    Rekursionsvorschrift: $hof(n) = hof[n - hof(n-1)] + hof[n - hof(n-2)],\ n>2$

    Formuliert man den Algorithmus zur Berechnung der Hofstadter-Funktion als Python-Programm mit rekursivem Funktionsaufruf, haben wir die Erfahrung gemacht, daß die Rechenzeit für große Werte von n sehr schnell wächst; der Grund ist die mit n sehr schnell wachsende Anzahl gleichzeitig aktiver Aufrufe der Funktion hof.

    Dieses ungünstige Laufzeitverhalten läßt sich umgehen, indem man den Algorithmus zur Berechnung der Hofstader-Funktion iterativ formuliert.

    Vorschlag zur iterativen Formulierung:
    Definiere ein array a mit den Komponenten a[0], a[1], a[2], ... und setze a[0] = hof(1) = 1, a[1] = hof(2) = 1.
    Den weiteren Komponenten a[2], a[3], . . . werden in dieser Reihenfolge die Werte hof(3), hof(4), ... zugewiesen.

    Konzipiere und teste das iterativ formulierte Python-Programm!

11. Zusatzaufgabe:

    Die Fibonacci-Folge $\{a_i\}$ ist wie folgt definiert:

    $$\mathbf{a_1 = a_2 = 1}$$

    $$\mathbf{a_n = a_{n-1} + a_{n-2}} \quad \text{für } n \geq 3$$

    Schreibe und teste ein Python-Programm zur Berechnung der Fibonacci-Folge.

12. Der als Python-Programm formulierte Algorithmus
    sorting_by_direct_selection.py.txt auf
    https://kalle2k.lima-city.de/computerscience/Informatik_12/sorting/
    sortiert ein array von Zufallszahlen aufsteigend, d. h. die sortierte Liste beginnt mit dem kleinsten Element.

    Modifiziere das Programm so, daß das Sortieren absteigend erfolgt.

Lösungen Aufgabenblatt Nr. 4 vom 26.01.2021

Nr. 10 (Hofstadter-Folge)

rekursiv:

```
def hof(x):
    global z
    z+=1
    if x == 1:
        return 1
    elif x == 2:
        return 1
    elif x > 2:
        return hof(x - hof(x - 1)) + hof(x - hof(x - 2))

endwert = int(input('Endwert n =  '))

n = 1

while n <= endwert:
    z = 0
    y = hof(n)
    print('hof(',n,') =',y)
    print('# Aufrufe =',z)
    print()
    n += 1
```

iterativ:

```
# Hofstadter iterativ

n = int(input('Endwert n =  '))
print()
y = list(range(1, n+2))

y[1]=1
y[2]=1

print ('hof(',1,') = ',y[1])
print ('hof(',2,') = ',y[2])

for i in range(3,n+1):
     y[i] = y[i-y[i-1]] + y[i - y[i-2]]
     print ('hof(',i,') = ',y[i])
```

Nr. 11 (Fibonacchi-Folge)

rekursiv:

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

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


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

iterativ:

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

a = list(range(0,n+1))

a[0] = 0
a[1] = 1

print('fibonacci(',0,') = ',a[0])
print('fibonacci(',1,') = ',a[1])

for i in range(2,n+1):
    a[i] = a[i-1] + a[i-2]
    print('fibonacci(',i,') = ',a[i])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368]
```

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:

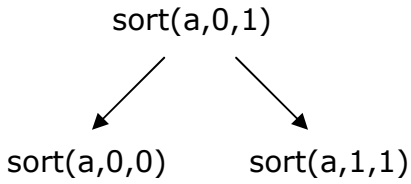
Mit **f(n)** bezeichnen wir die Anzahl der gleichzeitig aktiven Aufrufe der Funktion **`sort`**, wenn eine Liste mit **`n`** Datenelementen zu sortieren ist.

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"*

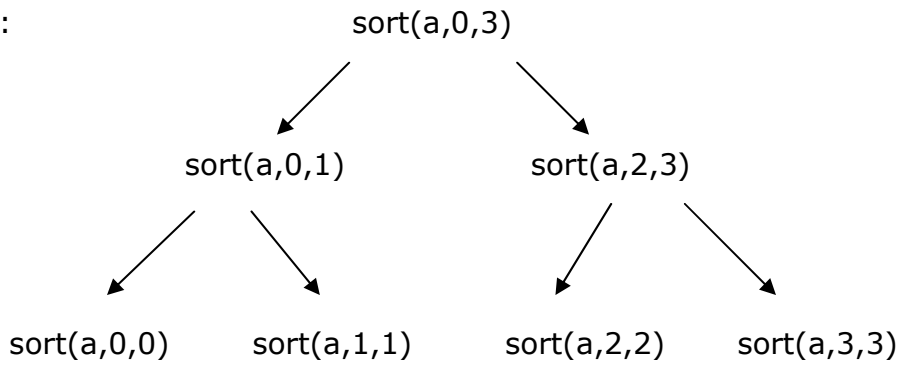
n = 1: sort(a,0,0) 1 Aufruf

n = 2:



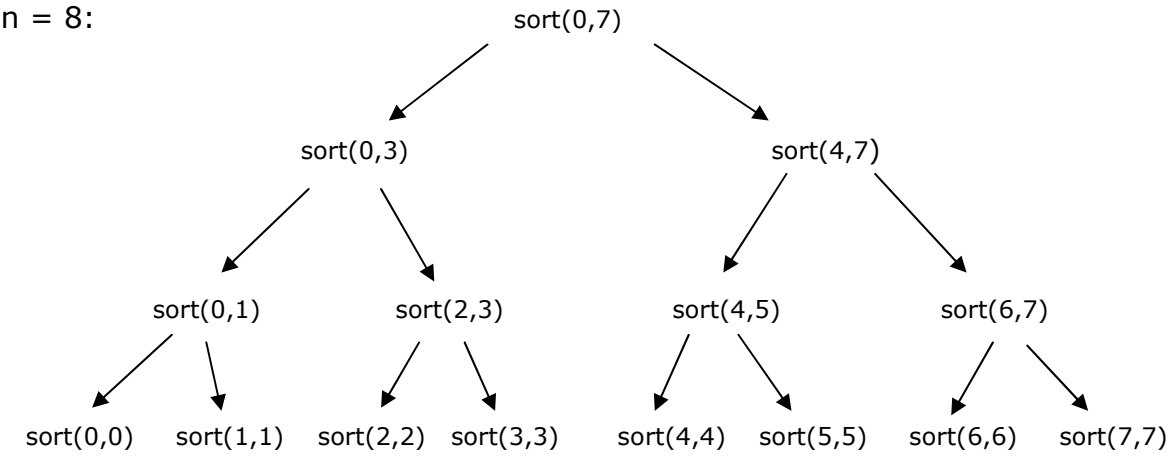
$1 + 2 \cdot 1 = 3$ Aufrufe

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

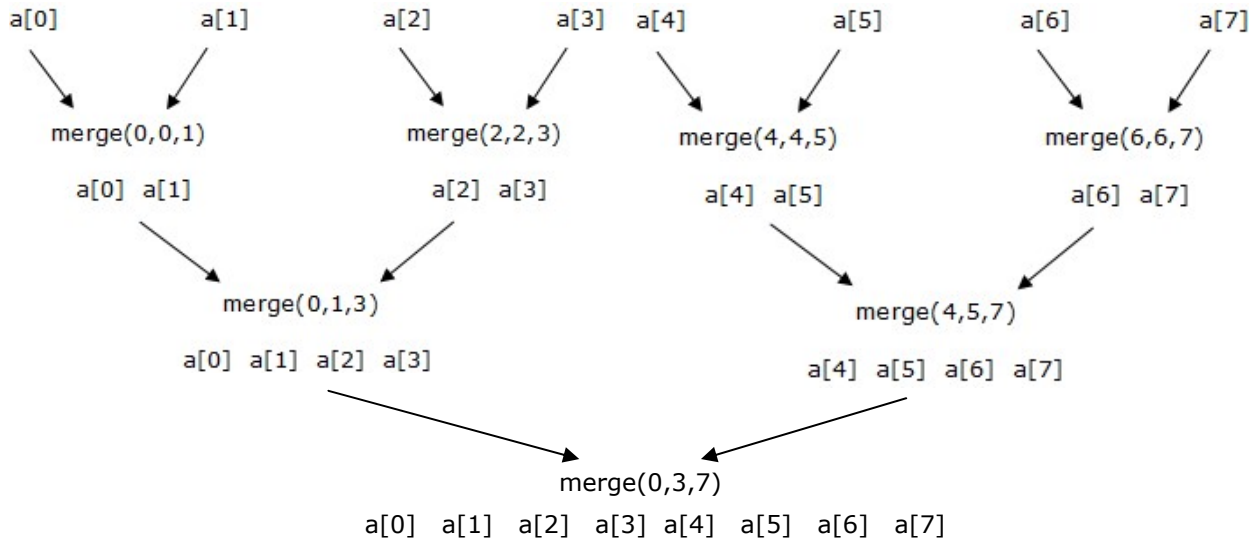
$f(n) = 1 + 2 \cdot f(n/2)$

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

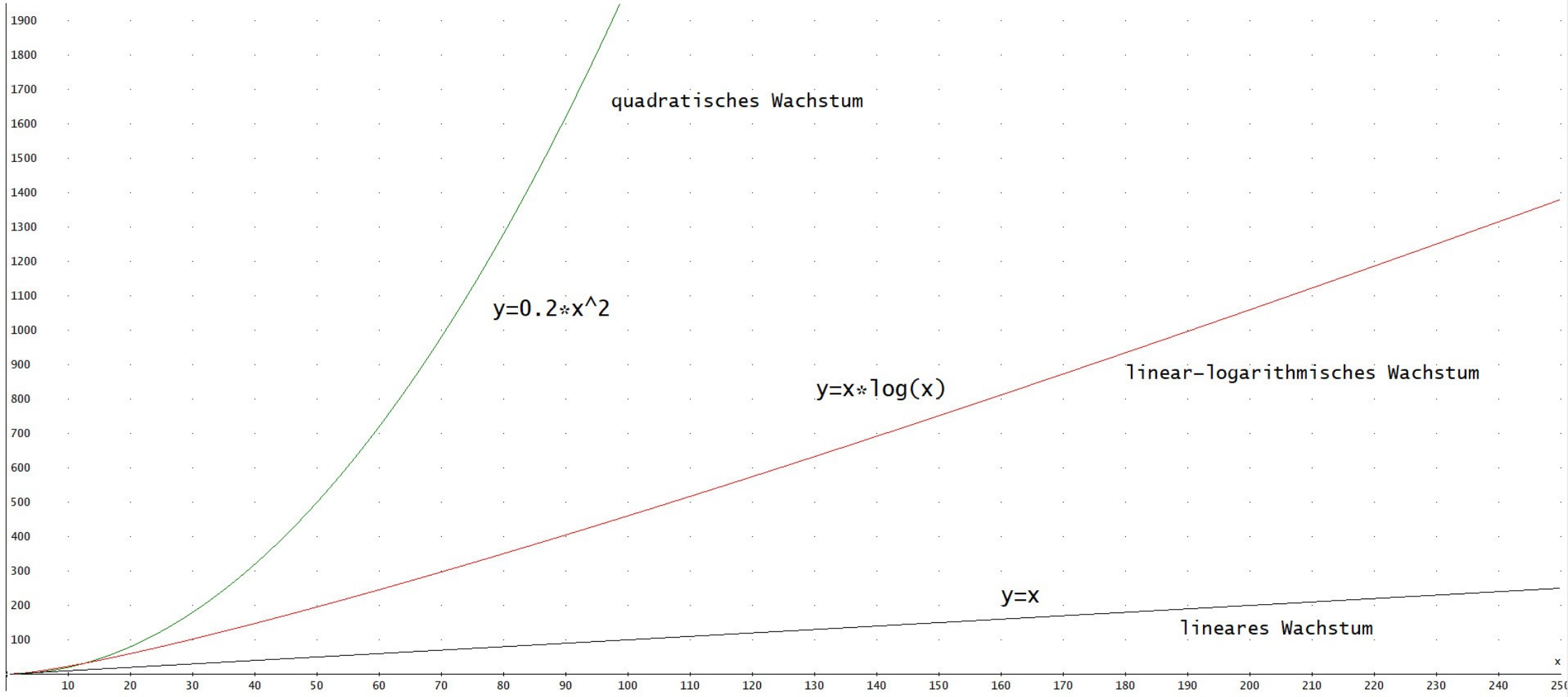
Bemerkung:
Für den Aufwand A(n) und folglich den Zeitbedarf zur Laufzeit des Algorithmus erhalten wir bei

- 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.

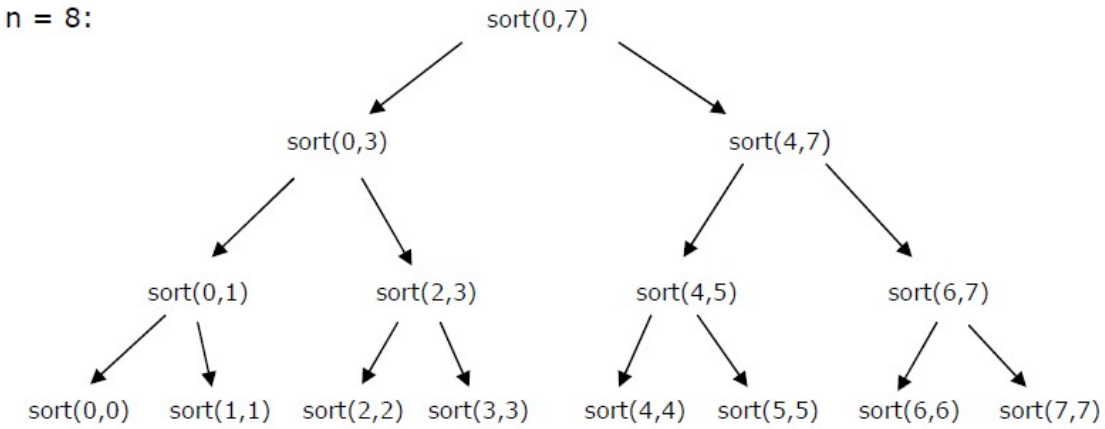

quadratisches Wachstum
y=0.2*x^2
y=x*log(x)
linear-logarithmisches Wachstum
y=x
lineares Wachstum
x
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210
220
230
240

13. **MergeSort**

Gegeben ist das aus den acht Komponenten **a[0], a[1], ... , a[7]** bestehende array **a**, die gemäß beigefügtem Arbeitsblatt mit ganzen Zahlen belegt sind; das array soll schrittweise gemäß dem Algorithmus MergeSort aufsteigend sortiert werden.

Bemerkung:
Im folgenden schreiben wir sort(left,right) statt sort(a,left,right) und merge(left,middle,right) statt merge(a,left,middle,right).

Mit dem Aufruf **sort(a,0,7)** bzw. **sort(0,7)** wird der Vorgang zum Sortieren des aus 8 Komponenten bestehenden arrays **a** eingeleitet; dabei veranlaßt die rekursiv formulierte Funktion **sort** weitere Aufrufe von sich selbst gemäß folgendem Baumdiagramm:



Diese Baumstruktur ist auf der Seite 1 des beigefügten Arbeitsblatts nachempfunden. Nachdem das array a in Teillisten jeweils der Länge 1 zerlegt wurde (eine aus 1 Element bestehende Liste ist bereits sortiert), werden jeweils 2 sortierten Teillisten mit merge zu 1 sortierten Liste gemischt (Seite 2).

Aufgabe: In der beigefügten Übersicht MergeSort_Arbeitsblatt.doc wird der Sortiervorgang zum Sortieren eines aus 8 Komponenten bestehenden arrays schrittweise vollzogen; ergänze alle fehlenden Einträge in MergeSort_Arbeitsblatt.doc (oder handschriftlich in der ausgedruckten Version MergeSort_Arbeitsblatt.pdf).

14. **SelectionSort**

Der Algorithmus **sorting_by_direct_selection.py**
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!

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)					sort(4,7)			
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(0,1)			sort(2,3)			sort(4,5)			sort(6,7)	
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(0,0)		sort(1,1)		sort(2,2)		sort(3,3)		sort(4,4)		sort(5,5)		sort(6,6)		sort(7,7)
a[0]		a[1]		a[2]		a[3]		a[4]		a[5]		a[6]		a[7]
7		6		8		2		9		3		8		5

a[0]		a[1]		a[2]		a[3]		a[4]		a[5]		a[6]		a[7]
7		6		8		2		9		3		8		5
merge(0,0,1)				merge(2,2,3)				merge(4,4,5)				merge(6,6,7)		

a[0]	a[1]		a[2]	a[3]		a[4]	a[5]		a[6]	a[7]
6	7		2	8		3	9		5	8
merge(0,1,3)						merge(4,5,7)				

a[0]	a[1]	a[2]	a[3]		a[4]	a[5]	a[6]	a[7]
2	6	7	8		3	5	8	9
merge(0,3,7)								

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
2	3	5	6	7	8	8	9

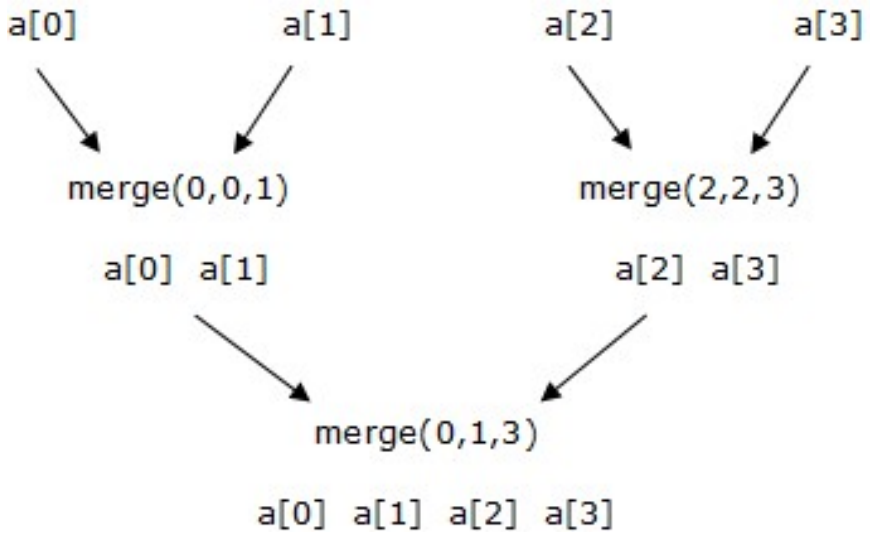
Lukas Pleis

15. **MergeSort**

In dem paper **MergeSort_final.pdf** (08.03.2021) wurde die Funktion **f(n)** hergeleitet, welche in Abhängigkeit von n die Anzahl der Aufrufe der Funktion **`sort`** angibt; wegen f(n) = 2n – 1 wächst f(n) linear mit n und daher erheblich schwächer als der Aufwand A(n).

Aufgabe: Finde in entsprechender Weise einen Funktionsterm für die Funktion **g(n)**, welche die Anzahl der Aufrufe der Funktion **`merge`** in Abhängigkeit von n bestimmt.

Hinweis: Auch hier beschränke man sich auf Werte von n, die sich als Zweierpotenz schreiben lassen (n = 1, 2, 4, 8, ...). Fertige für n = 2 und n = 8 jeweils eine Baumstruktur an gemäß folgendem Beispiel (n = 4):



Die Pfeile bedeuten hier: „wird gemischt"; z. B. werden die sortierten Teillisten {a[0], a[1]} und {a[2], a[3]} vermöge merge(0,1,3) zur sortierten Liste {a[0], a[1], a[2], a[3]} gemischt.

16. Implementiere in dem in Python geschriebenen Quelltext **mergesort.py** Zählvariablen z und y, welche zur Laufzeit des Algorithmus die Anzahl der Aufrufe der Funktion **`sort`** und der Funktion **`merge`** ermitteln; bestätige auf diese Weise die Ergebnisse, die für f(n) und g(n) gefunden wurden.

Bemerkung:
Bei MergeSort hat der Rechenaufwand A(n), um eine Liste mit n Komponenten zu sortieren, wegen $A(n) \sim n \cdot \log_2(n)$ eine linear-logarithmische Komplexität; da die Anzahl der rekursiven Funktionsaufrufe linear mit n wächst, hat der zur Laufzeit des Algorithmus benötigte Speicher lineare Komplexität.

Binäre Suche

Gegeben: Ein sortiertes Array **a** mit **n** Komponenten **a[0], ... , a[n-1]**
Aufgabe: Entscheide, ob ein zur Laufzeit für die Variable **value** eingegebener Wert im Array **a** vorkommt.

Beispiel

value = 13
n = len(a) = 10

Suche **value** in der Liste **a[0], . . . , a[9]**; diese Liste und **value** übergeben wir der Booleschen Funktion **binarysearch**, welche **a[0], . . . , a[9]** als lokale Liste **array[0], . . . , array[9]** verarbeitet.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
3	4	5	5	7	8	11	13	19	21

array[0]	array [1]	array [2]	array [3]	array [4]	array [5]	array [6]	array [7]	array [8]	array [9]
3	4	5	5	7	8	11	13	19	21

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 5

2. Schritt:
midvalue = array[len(array)//2] = array[10//2] = array[5] = 8

Wir vergleichen value mit midvalue:
Falls value == midvalue: binarysearch gibt den Wert True zurück; gefunden!
Falls value < midvalue: suche in der Liste a[0], . . . , a[4] links von a[5]
Falls value > midvalue: suche in der Liste a[6], . . . , a[9] rechts von a[5]

hier: wegen 13 > 8 suchen wir in der Liste a[6], . . . , a[9]

Suche **value** in der Liste **a[6], . . . , a[9]**

a[6]	a[7]	a[8]	a[9]
11	13	19	21

Diese Liste **a[6], . . , a[9]** und **value** übergeben wir der Booleschen Funktion **binarysearch**, welche **a[6], . . , a[9]** als lokale Liste **array[0], . . . , array[3]** verarbeitet.

array[0]	array[1]	array[2]	array[3]
11	13	19	21

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 4//2 = 2

2. Schritt:
midvalue = array[len(array)//2] = array[4//2] = array[2] = 19

Wir vergleichen value mit midvalue:
Falls value == midvalue: binarysearch gibt den Wert True zurück; gefunden!
Falls value < midvalue: suche in der Liste array[0], . . . , array[1] links von array[2]
Falls value > midvalue: suche in der Liste array[3] rechts von array[2]

hier: wegen 13 < 19 suchen wir in der Liste array[0], . . . , array[1]

Suche **value** in der Liste **array[0], . . . , array[1]**

array[0]	array[1]
11	13

Diese Liste **array[0], . . . , array[1]** und **value** übergeben wir der Booleschen Funktion **binarysearch**, welche **array[0], . . . , array[1]** als lokale Liste **array[0], . . . , array[1]** verarbeitet.

1. Schritt:
Wir bestimmen den mittleren Index des Arrays array: len(array)//2 = 2//2 = 1

2. Schritt:
midvalue = array[len(array)//2] = array[2//2] = array[1] = 13
Wir vergleichen value mit midvalue:
Falls value == midvalue: binarysearch gibt den Wert True zurück; gefunden!
Falls value < midvalue: suche in der Liste array[0] links von array[1]
Falls value > midvalue: suche in der leeren Liste [] rechts von array[1], dann: binarysearch gibt den Wert False zurück; nicht gefunden!

hier: wegen 13 = value = midvalue = 13: gefunden!

Listen in Python 13.04.2021

Definition: Unter einem Array verstehen wir eine Folge von Variablen gleichen Typs.

Bemerkung: *In Python läßt sich ein Array z. B. als Liste realisieren; es gibt Programmiersprachen (z. B. Pascal), bei denen ‚array' Schlüsselwort für eine Datenstruktur ist.*

```
>>> a=list(range(4,14))
>>> print(a)
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> print(a[0])
4
>>> print(a[9])
13
>>> print(a[3])
7
>>> print(a[:3])
[4, 5, 6]
>>> print(a[3:])
[7, 8, 9, 10, 11, 12, 13]
>>> a.append(87)
>>> print(a)
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 87]
>>> print(a[10])
87
>>>
```

Bemerkung: Der Operator , : ' heißt auch slice-Operator.

Die folgende Liste ist kein array im engeren Sinne, da Komponenten unterschiedlichen Typs (integer und string) vorkommen:

```
>>> a[10]='fritz'
>>> print(a)
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 'fritz']
```

Bei dem zweiten der beiden folgenden Python-Programme wird die aus 7 Komponenten bestehende Teilliste a[3], ... , a[9] von a der Funktion test übergeben, welche diese Komponenten quadriert; innerhalb der Funktion test wird die aus 7 Komponenten bestehende Liste b mit 0, 1, ... , 6 indiziert.

```
a = list(range(4,14))
print('Quelliste a:')
print(a)

def test(b):
     for i in range(0,len(b)):
         b[i] = b[i]*b[i]
     print(b)

print()
# Aufruf der Funktion test
test(a)
test(a[3:])
test(a[:3])
```

```
Quelliste a:
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

[16, 25, 36, 49, 64, 81, 100, 121, 144, 169]
[2401, 4096, 6561, 10000, 14641, 20736, 28561]
[256, 625, 1296]
```

```
a = list(range(4,14))
print('Quelliste a:')
print(a)

def test(b):
     for i in range(0,len(b)):
         b[i] = b[i]*b[i]
     print(b)
     print(b[0])
     print(b[6])

print()
# Aufruf der Funktion test
test(a[3:])
```

```
Quelliste a:
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

[49, 64, 81, 100, 121, 144, 169]
49
169
```

Binäre Suche (BinarySearch)

Als Datenstruktur legen wir das aus den **n** Komponenten **a[0], ... , a[n-1]** bestehende Array **a** zugrunde, für dessen Komponenten die Ordnungsrelationen <, >, ≤, ≥, = definiert sind.
Nach Zuweisung eines Wertes an die Variable **value** werden das sortierte Array **a** und **value** der Funktion **binarysearch** übergeben; **binarysearch** entscheidet, ob es in der sortierten Liste mit **a[0] ≤ . . . . ≤ a[n-1]** einen Index **i** gibt mit **a[i] = value**; falls dies zutrifft, liefert **binarysearch** den Booleschen Wert **True**, andernfalls den Wert **False**.

Python-Quelltext:

```
from random import randint
z = 0

n = int(input('Anzahl der Datenelemente = '))

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

for i in range(0,n):
    a[i]= randint(1,100)

print('Quelliste: ')
print(a)

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

print('sortierte Liste: ')
print(a)
print()
value = int(input('gesuchte Zahl: '))

# binarysearch liefert den Wert True, falls value als Inhalt einer Komponente
# des Arrays array vorkommt,andernfalls liefert binarysearch den Wert False.

def binarysearch(array,value):
     global z
     z += 1
     print(array)
     if array == [] or (len(array) == 1 and array[0] !=  value):
         return False
     else:
         midvalue = array[len(array)//2]
         if midvalue == value:
             return True
         elif value < midvalue:
             return binarysearch(array[:len(array)//2],value)
         else:
             return binarysearch(array[len(array)//2 + 1:],value)


# Aufruf der Funktion binarysearch zur Suche von value im Array a

if binarysearch(a,value) == True:
    print(value,'wurde gefunden')
else:
    print(value,'wurde nicht gefunden')
print('Anzahl Aufrufe binarysearch =',z)
```

Durchführung des Algorithmus für ein aus 32 Zufallszahlen bestehendes Array **a**:

n = **len(a)** = Anzahl der Datenelemente = 32

Quelliste:

[77, 26, 19, 54, 29, 20, 38, 38, 1, 94, 83, 53, 90, 17, 66, 79, 43, 36, 11, 57, 52, 99, 68, 20, 32, 27, 7, 46, 91, 75, 54, 78]

a[0]	a[1]	a[2]	a[3]	....	....	....	....	....	....	a[30]	a[31]
77	26	19	54	....	....	....	....	....	....	54	78

sortierte Liste **a**:

[1, 7, 11, 17, 19, 20, 20, 26, 27, 29, 32, 36, 38, 38, 43, 46, 52, 53, 54, 54, 57, 66, 68, 75, 77, 78, 79, 83, 90, 91, 94, 99]

a[0]	a[1]	a[2]	....	....	a[15]	a[16]	a[17]	a[18]	....	a[30]	a[31]
1	7	11	....	....	46	52	53	54	....	94	99

Bemerkung: Jede Teilliste der Liste **a** ist ebenfalls sortiert.

gesuchter Wert: **value** = 76

1. Aufruf der Funktion **binarysearch**

Die sortierte Liste **a** und **value** werden mit dem Aufruf **binarysearch(a,value)** der Funktion **binarysearch** übergeben; **binarysearch** übernimmt das Array **a** als lokales Array **array**.

1. Schritt:
BinarySearch bestimmt den mittleren Index des Arrays: **len(a)//2 = 32//2 = 16**

2. Schritt:
Wert der Komponente in der Mitte des Arrays: **midvalue = a[len(a)//2] = a[16] = 52**
Wegen 76 > 52 nimmt der Boolesche Term **value < midvalue** den Wert **False** an; folglich ist die Suche in der aus 15 Komponenten bestehenden Teilliste „rechts" von a[16] fortzusetzen, also in der Liste

[53, 54, 54, 57, 66, 68, 75, 77, 78, 79, 83, 90, 91, 94, 99]

a[17]	a[18]	....	....	....	a[30]	a[31]
53	54	....	....	....	94	99

2. Aufruf der Funktion **binarysearch**

Mit dem rekursiven Aufruf
binarysearch(array[len(array)//2 + 1:],value)
(beachte: **(value < midvalue)** hat den Wert **False**) werden die vorstehende Teilliste und **value** der Funktion **binarysearch** übergeben; **binarysearch** verarbeitet diese Teilliste als lokales array mit den Indices 0, 1, ... , 14:

a[0]	a[1]	....	a[6]	a[7]	a[8]	....	a[13]	a[14]
53	54	....	75	77	78	....	94	99

1. Schritt:
BinarySearch bestimmt den mittleren Index des Arrays: **len(a)//2 = 15//2 = 7**

2. Schritt:
Wert der Komponente in der Mitte des Arrays: **midvalue = a[len(a)//2] = a[7] = 77**
Wegen 76 < 77 nimmt der Boolesche Term **value < midvalue** den Wert **True** an; folglich ist die Suche in der aus 7 Komponenten bestehenden Teilliste „links" von a[7] fortzusetzen, also in der Liste

[53, 54, 54, 57, 66, 68, 75]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
53	54	54	57	66	68	75

3. Aufruf der Funktion **binarysearch**

Mit dem rekursiven Aufruf
binarysearch(array[:len(array)//2],value)

(beachte: **(value < midvalue)** hat den Wert **True**) werden die vorstehende Teilliste und **value** der Funktion **binarysearch** übergeben; **binarysearch** verarbeitet diese Teilliste als lokales array mit den Indices 0, 1, ... , 6:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
53	54	54	57	66	68	75

1. Schritt:
BinarySearch bestimmt den mittleren Index des Arrays: **len(a)//2 = 7//2 = 3**

2. Schritt:
Wert der Komponente in der Mitte des Arrays: **midvalue = a[len(a)//2] = a[3] = 57**
Wegen 76 > 57 nimmt der Boolesche Term **value < midvalue** den Wert **False** an; folglich ist die Suche in der aus 3 Komponenten bestehenden Teilliste „rechts" von a[3] fortzusetzen, also in der Liste

[66, 68, 75]

a[4]	a[5]	a[6]
66	68	75

4. Aufruf der Funktion **binarysearch**

Mit dem rekursiven Aufruf
binarysearch(array[len(array)//2 + 1:],value)
(beachte: **(value < midvalue)** hat den Wert **False**) werden die vorstehende Teilliste und **value** der Funktion **binarysearch** übergeben; **binarysearch** verarbeitet diese Teilliste als lokales array mit den Indices 0, 1, 2:

a[0]	a[1]	a[2]
66	68	75

1. Schritt:
BinarySearch bestimmt den mittleren Index des Arrays: **len(a)//2 = 3//2 = 1**

2. Schritt:
Wert der Komponente in der Mitte des Arrays: **midvalue = a[len(a)//2] = a[1] = 68**
Wegen 76 > 68 nimmt der Boolesche Term **value < midvalue** den Wert **False** an; folglich ist die Suche in der aus 1 Komponente bestehenden Teilliste „rechts" von a[1] fortzusetzen, also in der Liste

[75]

a[2]
75

5. Aufruf der Funktion **binarysearch**

Mit dem rekursiven Aufruf
binarysearch(array[len(array)//2 + 1:],value)
(beachte: **(value < midvalue)** hat den Wert **False**) werden die vorstehende Teilliste und **value** der Funktion **binarysearch** übergeben; **binarysearch** verarbeitet diese Teilliste als lokales array mit dem Index 0:

a[0]
75

Da wegen 75 ≠ 76 der Boolesche Term **array[0] != value** den Wert **True** annimmt und da die Länge des übergebenen Arrays den Wert 1 hat, erhält die Boolesche Konjunktion

len(array) == 1 and array[0] != value

den Wert **True**; folglich liefert die Funktion **binarysearch** den Wert **False**, und der Algorithmus bricht ab mit der Ausgabe: „76 wurde nicht gefunden".

Aufwandsbetrachtung:

Die erfolglose Suche (wie im oben durchgeführten Beispiel) in einem aus n Komponenten bestehenden Array erfordert eine maximale Anzahl von Aufrufen der Funktion **`binarysearch`**; dagegen endet eine erfolgreiche Suche, sobald der Boolesche Term **`midvalue == value`** den Wert **`True`** annimmt.

O. B. d. A. nehmen wir an, daß n eine Potenz von 2 ist, d. h. es gibt eine ganze nicht negative Zahl k mit $n = 2^k$.

Wir überlegen, wie viele Teilungen und damit wie viele Aufrufe von **`binarysearch`** im „worst case" benötigt werden, bis man zu einem Array mit 1 Komponente gelangt:

n	k	Maximale Anzahl der Aufrufe **`binarysearch`**
1	0	1
2	1	2
4	2	3
8	3	4
16	4	5
32	5	6
64	6	7
n	$\log_2(n)$	$1 + \log_2(n)$

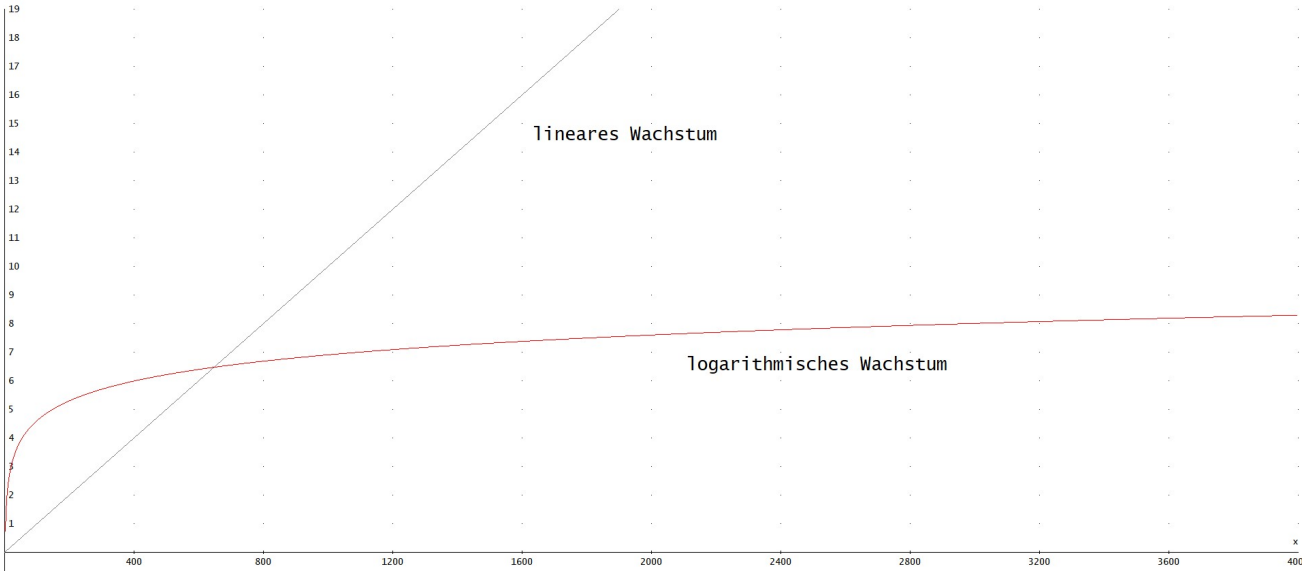
Wegen $n = 2^k$ gilt $k = \log_2(n)$; damit folgt für die maximale Anzahl **A** der Aufrufe von **`binarysearch`**:

$$\mathbf{A = 1 + \log_2(n)}$$

Für große Werte von n kann man den Summand 1 vernachlässigen, so daß in guter Näherung gilt:

$$\mathbf{A \approx \log_2(n)}$$

Da die Rechenzeit der Anzahl der benötigten Aufrufe der rekursiv formulierten Funktion **`binarysearch`** folgt, hat der Algorithmus „Binäre Suche" logarithmische Komplexität.



binarysearch

Quelliste: Liste von 50 Zufallszahlen

gesuchtes Element: 80

Quelliste:
[67, 54, 59, 16, 60, 81, 47, 63, 31, 71, 20, 97, 31, 27, 86, 22, 92, 78, 75, 95, 14, 87, 16, 88, 63, 72, 44, 21, 59, 55, 67, 60, 34, 27, 54, 7, 58, 87, 21, 17, 14, 31, 67, 44, 75, 51, 47, 90, 68, 44]

sortierte Liste:
[7, 14, 14, 16, 16, 17, 20, 21, 21, 22, 27, 27, 31, 31, 31, 34, 44, 44, 44, 47, 47, 51, 54, 54, 55, 58, 59, 59, 60, 60, 63, 63, 67, 67, 67, 68, 71, 72, 75, 75, 78, 81, 86, 87, 87, 88, 90, 92, 95, 97]

an binarysearch uebergebene Liste nach dem 1 . Aufruf:

[7, 14, 14, 16, 16, 17, 20, 21, 21, 22, 27, 27, 31, 31, 31, 34, 44, 44, 44, 47, 47, 51, 54, 54, 55, 58, 59, 59, 60, 60, 63, 63, 67, 67, 67, 68, 71, 72, 75, 75, 78, 81, 86, 87, 87, 88, 90, 92, 95, 97]

an binarysearch uebergebene Liste nach dem 2 . Aufruf:

[59, 59, 60, 60, 63, 63, 67, 67, 67, 68, 71, 72, 75, 75, 78, 81, 86, 87, 87, 88, 90, 92, 95, 97]

an binarysearch uebergebene Liste nach dem 3 . Aufruf:

[75, 78, 81, 86, 87, 87, 88, 90, 92, 95, 97]

an binarysearch uebergebene Liste nach dem 4 . Aufruf:

[75, 78, 81, 86, 87]

an binarysearch uebergebene Liste nach dem 5 . Aufruf:

[75, 78]

an binarysearch uebergebene Liste nach dem 6 . Aufruf:

[]

80 wurde nicht gefunden

Anzahl der Aufrufe der Routine binarysearch: 6