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.