Binäre Suche

Informatik 12 Februar 2024

Gegeben: Ein aufsteigend sortiertes Array **a** mit den **n** Komponenten **a[0], ... , a[n-1]**

Aufgabe: Entscheide, ob ein für die Variable **value** eingegebener Suchwert mit dem Wert einer Komponente des Arrays **a** übereinstimmt.

Wir durchlaufen den Algorithmus schrittweise anhand des folgenden Beispiels.

Gegeben: Array **a** mit den Komponenten **a[0], ... , a[9]**; n = len(a) = 10
value = **13**

Die rekursiv formulierte Boolesche Funktion **binarysearch** liefert den Wert **`True`**, falls **value** mit dem Wert irgendeiner Komponente von **a** übereinstimmt, andernfalls liefert sie den Wert **`False`**.

Wir übergeben **value** und die Liste **a[0], . . . , a[9]**

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

der Funktion **binarysearch**,
welche **a[0], . . . , a[9]** als lokale Liste **array[0], . . . , array[9]** fortführt:

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:
Bestimme den mittleren Index **middle** des Arrays **array**: **middle** = len(array)//2 = 10//2 = 5

2. Schritt:
midvalue = array[middle] = array[5] = **8**
Vergleiche **value** mit **midvalue**:
Falls **`value == midvalue`**: **binarysearch** gibt den Wert **`True`** zurück; gefunden!
Falls **`value < midvalue`**: suche in der Liste array[0], . . . , array[4] links von array[5]
Falls **`value > midvalue`**: suche in der Liste array[6], . . . , array[9] rechts von array[5]

hier: wegen **13 > 8** suche in der Liste array[6], . . . , array[9] .

binarysearch übergibt **value** und die Liste **array[6], . . . , array[9]**

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

der Funktion **binarysearch**,
welche **array [6], . . . , array [9]** als lokale Liste **array[0], . . . , array[3]** fortführt:

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

1. Schritt:
Bestimme den mittleren Index **middle** des Arrays **array**: **middle** = len(array)//2 = 4//2 = 2

2. Schritt:
midvalue = array[middle] = array[2] = 19
Vergleiche **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** suche in der Liste array[0], . . . , array[1] .

binarysearch übergibt **value** und die Liste **array[0], . . . , array[1]**

array[0]	array[1]
11	13

der Funktion **binarysearch**,
welche **array [0], . . . , array [1]** als lokale Liste **array[0], . . . , array[1]** fortführt:

array[0]	array[1]
11	13

1. Schritt:
Bestimme den mittleren Index **middle** des Arrays **array**: **middle** = len(array)//2 = 2//2 = 1

2. Schritt:
midvalue = array[middle] = array[1] = 13
Vergleiche **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`**: die Liste [] rechts von array[1] ist leer;
dann gibt **binarysearch** den Wert **`False`** zurück: nicht gefunden!

Hier: da der Suchwert **value** und **array[middle]** übereinstimmen, hat der Boolesche Term **`value == midvalue`** den Wert **`True`**; folglich liefert **binarysearch** den Wert **`True`**: gefunden!

Wäre 12 der Suchwert, erhielte man wegen 12<13 im 2. Schritt: suche in der Liste array[0] links von array[1]

binarysearch übergibt **value** und die Liste **array[0]**

array[0]
11

der Funktion **binarysearch**, welche den Wert **`False`** liefert, da array[0]≠value und array die Länge 1 hat.
Zusammengefaßt: **binarysearch** liefert den Wert **`False`**, falls

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

den Wert **`True`** annimmt.

Formulierungen der Booleschen Funktion **binarysearch** in Python:

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

def binarysearch(value, a):
     if a == [] or (a[0] !=  value and len(a) == 1): return False
     else:
         if value == a[len(a)//2]:  return True
         elif value < a[len(a)//2]: return binarysearch(value, a[:len(a)//2])
         else:                      return binarysearch(value, a[len(a)//2 + 1:])
```

Funktionsaufruf zum Suchen von **value** im sortierten Feld **a**: **binarysearch(value,a)**

Mittels einer Zählvariablen **z** ermitteln wir die Anzahl der Aufrufe von **binarysearch**, die Anweisung **print(array)** gibt die jeweilige Teilliste aus, auf der **binarysearch** operiert:

```
. . . . . .
z = 0
. . . . . .
def binarysearch(array,value):
     global z
     z += 1
     print(array)
     if array == [] or (len(array) == 1 and array[0] !=  value): return False
     . . . . . .
. . . . .
```

Komplexität des Algorithmus **`binarysearch`**:

Die Komplexität zur Laufzeit und damit der Rechenaufwand **A(n)** wird wesentlich bestimmt durch die Anzahl der Vergleiche von **`value`** mit **`midvalue`**, somit durch die Anzahl **z** der Aufrufe der rekursiv definierten Funktion **`binarysearch`**; o. B. d. A. sei **n** eine Potenz von **2**, d. h. $\mathbf{n = 2^k}$ mit k = 0, 1, 2, 3, 4, ...
Beachte: die maximale Anzahl von Aufrufen (worst case) tritt ein, falls die Suche ergebnislos ist.

k = 0 ⇔ n = 1

```
# Aufrufe binarysearch = 1
```

k = 3 ⇔ n = 8

```
gesuchte Zahl: 79
[14, 50, 52, 70, 74, 80, 89, 97]
[80, 89, 97]
[80]
79 wurde nicht gefunden
# Aufrufe binarysearch = 3
```

k = 4 ⇔ n = 16

```
gesuchte Zahl: 80
[13, 33, 42, 42, 44, 44, 45, 45, 47, 52, 57, 59, 62, 72, 92, 94]
[52, 57, 59, 62, 72, 92, 94]
[72, 92, 94]
[72]
80 wurde nicht gefunden
# Aufrufe binarysearch = 4
```

k = 5 ⇔ n = 32

```
zu suchende Zahl: 33
[6, 9, 9, 11, 13, 18, 19, 23, 26, 28, 32, 34, 37, 37, 44, 44, 48, 50, 58, 58, 59, 66, 76, 79, 81, 84, 86, 94, 95, 97, 97, 98]
[6, 9, 9, 11, 13, 18, 19, 23, 26, 28, 32, 34, 37, 37, 44, 44]
[28, 32, 34, 37, 37, 44, 44]
[28, 32, 34]
[34]
33 wurde nicht gefunden
# Aufrufe binarysearch = 5
```

Eine Verdopplung der „Problemgröße" **n** impliziert höchstens einen weiteren Aufruf von **binarysearch**!

Offensichtlich gilt:

$$\mathbf{z = k}$$

Wegen $\mathbf{n = 2^k \Leftrightarrow k = \log_2(n)}$ folgt:

$$\mathbf{z = \log_2(n)}$$

Somit hat der Algorithmus **binarysearch** logarithmische Komplexität:

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

Bemerkung:
Bei rekursiver Formulierung des Algorithmus BinarySearch ist die Speicherkomplexität ebenfalls von der Ordnung $O(\log_2(n))$, da bei jedem Aufruf der Funktion binarysearch neuer Speicherplatz für die Variablen bereitgestellt wird. Die weniger elegante iterative Formulierung von BinarySearch hätte zur Laufzeit einen vernachlässigbar geringeren Speicherbedarf zur Folge als die rekursive Formulierung.