# Suchen in einer Sequenz

## POS

Dr. Günter Kolousek

2016



## 1 Problematik

Betrachten wir die Aufgabenstellung, dass in einer Liste nach einem Eintrag gesucht werden soll.

## 2 Lineare Suche

Solange zwei Objekte miteinander auf Gleichheit verglichen werden können, kann man einfach über alle Einträge iterieren bis man den gesuchten Eintrag gefunden hat.

Eine Lösung dafür sieht folgendermaßen aus:

```
lst = ["abc", "def", "xy", "ghi", "jkl"]

search_elem = "ghi"

for e in lst:
    if e == search_elem:
        print("gefunden")
```

Damit kommt es zu folgender erwarteter Ausgabe:

```
gefunden
```

Etwas schwieriger wird dies allerdings, wenn man auch "nicht gefunden" ausgeben will. Der naive Ansatz funktioniert natürlich nicht:

```
lst = ["abc", "def", "xy", "ghi", "jkl"]

search_elem = "ghi"

for e in lst:
    if e == search_elem:
        print("gefunden")

print("nicht gefunden")
```

```
gefunden
nicht gefunden
```

Es gibt dafür einige Lösungen, die wir der Reihe nach behandeln wollen.

- Eine Lösung für dieses Teilproblem sieht so aus, dass man ein Flag einführt, das man setzt, wenn man das Element gefunden hat und die Ausgabe von diesem Flag abhängig macht:

```
lst = ["abc", "def", "xy", "ghi", "jkl"]

search_elem = "ghi"
found = False

for e in lst:
    if e == search_elem:
        found = True
        break

if found:
    print("gefunden")
else:
    print("nicht gefunden")
```

```
gefunden
```

- Diese Lösung funktioniert, aber es gibt in Python eine andere Lösung, die einfacher und übersichtlicher ist, aber nicht in anderen Programmiersprachen vorhanden ist:

```
lst = ["abc", "def", "xy", "ghi", "jkl"]

search_elem = "ghi"

for e in lst:
    if e == search_elem:
        print("gefunden")
        break
else:
    print("nicht gefunden")
```

```
gefunden
```

  Der `else`-Zweig einer `for`-Schleife wird nur ausgeführt, wenn die Schleife nicht über ein `break` Anweisung verlassen wird. Merken kann man sich das Verhalten so, dass dieses `else` wie ein `else` einer `if` Anweisung innerhalb der `for`-Schleife wäre.

- Die dritte Möglichkeit ist, die Suche wiederverwendbar zu machen und diese in eine Funktion zu verpacken:

```
def lin_search(seq, elem):
    for e in seq:
        if e == elem:
            return True
    return False

lst = ["abc", "def", "xy", "ghi", "jkl"]

print(lin_search(lst, "ghi"))
```

```
True
```

Wir sehen, dass diese Funktion `True` zurückliefert, wenn das Element in der Sequenz enthalten ist und `False` wenn es nicht enthalten ist. Daran ist prinzipiell nichts auszusetzen, wenn man mit diesem Verhalten zufrieden ist.

Will man allerdings auch die Position wissen, an der das Element steht, dann stellt sich die Frage was zurückgeliefert werden soll, wenn es nicht enthalten ist. Folgende Möglichkeiten gibt es:

- Man liefert weiterhin `False` zurück.
- Man liefert `None` zurück. Prinzipiell hat dies keinen besonderen Vorteil zu `False`, außer man möchte das Element selber zurückliefern nach dem gesucht worden ist. Wird in diesem Fall nach dem Wert `False` gesucht, könnte man nicht mehr unterscheiden ob das gefundene Element zurückgeliefert wurde oder ob es nicht vorhanden ist.
- Man liefert -1 zurück. Diese Lösung hat den Vorteil, dass `type(-1) == int` ist, also den gleichen Typ hat wie jede gültige Position in einer Sequenz. Das kann manchmal von Vorteil sein.

Was ist aber der Nachteil der linearen Suche? Ist das gesuchte Element nicht in der Sequenz vorhanden oder ist es das allerletzte Element, dann muss über die gesamte Sequenz iteriert werden. Das wird bei großen Sequenzen und mehrfachen Suchen sehr ineffektiv!

## 3 Binäre Suche

Die Idee der binären Suche ist die gleiche wie die sinnvolle Vorgehensweise beim Zahlenratespiel: Man tastet sich gezielt an das zu suchende Element an.

Dazu muss allerdings die vorliegende Sequenz sortiert sein. Das wiederum bedeutet, dass die Elemente der Sequenz miteinander bzgl. < und/oder > vergleichbar sein müssen (zusätzlich zu =).

Der Algorithmus lässt sich dann folgendermaßen formulieren:

```
def bin_search(seq, elem):
    l = 0
    r = len(seq) - 1
    while l <= r:
        m = (r + l) // 2
        if elem == seq[m]:
            return m
        elif elem < seq[m]:
            r = m - 1
        else:
            l = m + 1
    return None

lst = ["abc", "def", "xy", "ghi", "jkl"]

print(bin_search(lst, "ghi"))
```

```
None
```

Wieso funktioniert der Algorithmus nicht, obwohl ich behaupte, dass die Funktion richtig programmiert wurde? Überlege kurz **bevor** du weiterliest.

Das liegt daran, dass man Strings nicht per < vergleichen kann. Was sagst du dazu? So einfach ist das.

Ich habe dich doch darum gebeten, dir das zu überlegen, **bevor** du weiterliest und nicht gleich jeder Falschmeldung auf dem Leim zu gehen. Also was ist der Fehler, denn man kann Strings **sicher** per < als auch per > vergleichen.

Der Fehler liegt daran, dass die Liste nicht sortiert ist!

```
def bin_search(seq, elem):
    l = 0
    r = len(seq) - 1
    while l <= r:
        m = (r + l) // 2
        if elem == seq[m]:
            return m
        elif elem < seq[m]:
            r = m - 1
        else:
            l = m + 1
    return None

lst = ["abc", "def", "xy", "ghi", "jkl"]
sorted_lst = sorted(lst)

print(sorted_lst)
print(bin_search(sorted_lst, "ghi"))
```

```
['abc', 'def', 'ghi', 'jkl', 'xy']
2
```

Zwei Aufgabenstellungen lasse ich dir:

- Damit du den Algorithmus besser verstehst, gehe jetzt folgendermaßen vor, dass du geeignete `print` Anweisungen in die `while` Schleife einbaust, damit du für die Fälle "gefunden" und auch "nicht gefunden" nachvollziehen kannst, welche Zweige der `if` Anweisung innerhalb der Iterationen genommen werden. Probiere das auch an Hand einer größeren Liste aus.
- Wieviele Schleifendurchgänge werden maximal bei einer Liste mit 1, 2, 3,…, 10 Elementen benötigt?

  Schreibe ein Programm, das für jede Antwort in einer Zeile jeweils die Anzahl der Elemente $n$, die Anzahl der Schleifendurchgänge und den $\log_2(n)$ ausgibt. Damit sicher die Maximalanzahl bestimmt wird muss einfach nach einer nicht existierenden Zahl gesucht werden.

  Als Listenelemente kannst du einfach die natürlichen Zahlen von 1 bis zur jeweiligen Obergrenze nehmen und damit einfach eine Schleife programmieren. Es gibt leichte Abweichungen ob nach unten oder nach oben gesucht wird. Daher schreibe das Programm so, dass es einmal alle Ermittlungen mit der Zahl 0 (suchen nach unten) und einmal mit der Zahl 100 (suchen nach oben) durchführt.

  So, und nun probiere es selber aus! **Nur** wenn du absolut nicht weißt wie es geht (nach 10 Minuten Nachdenken), **dann** schaue auf der nächsten Seite weiter!

```python
import math

cnt = 0

# binary search also counting the number of iteration
# using the global variable cnt
def bin_search(seq, elem):
    l = 0
    r = len(seq) - 1
    global cnt
    cnt = 0
    while l <= r:
        cnt += 1
        m = (r + l) // 2
        if elem == seq[m]:
            return m
        elif elem < seq[m]:
            r = m - 1
        else:
            l = m + 1
    return None

# search to the lower bound
for i in range(1, 11):
     bin_search(list(range(1, i + 1)), 0)
     print(i, cnt, math.log2(i))

print()

# search to the upper bound
for i in range(1, 11):
     bin_search(list(range(1, i + 1)), 100)
     print(i, cnt, math.log2(i))
```

```
1 1 0.0
2 1 1.0
3 2 1.584962500721156
4 2 2.0
5 2 2.321928094887362
6 2 2.584962500721156
7 3 2.807354922057604
8 3 3.0
9 3 3.169925001442312
10 3 3.321928094887362

1 1 0.0
2 2 1.0
3 2 1.584962500721156
4 3 2.0
5 3 2.321928094887362
6 3 2.584962500721156
7 3 2.807354922057604
```

```
8 4 3.0
9 4 3.169925001442312
10 4 3.321928094887362
```

Zum Schluss interpretiere die Ausgabe.

Kannst du daraus eine allgemeine Formel finden, die die maximale Anzahl der Schleifendurchgänge errechnet? Denken, denken, denken!

Ok, auf der nächsten Seite folgt wieder die Lösung!

$[\log_2(x)]+1$, wobei es sich bei $[x]$ um die sogenannte Gaußklammer handelt, die auch als $\text{floor}(x)$ geschrieben wird und bedeutet, dass die nächstliegende nicht größere ganze Zahl von $x$ als Ergebnis erhalten wird.

For die mathematische Funktion $\text{floor}(x)$ gibt es im Mathematikmodul von Python eine entsprechende Funktion. Diese kannst du gerne verwenden. Wie sieht dies allerdings aus, wenn du die Funktionalität selber programmieren willst? Für den Fall, dass man floor(x) nur für positive $x$ verwendet, ist es relativ einfach, da…

Ok, wenn du wieder eine Lösung brauchst, dann schaue auf der nächsten Seite weiter, aber eigentlich könntest du schon selber drauf kommen!

Gut, hier ein kleines Programm, das dies demonstriert:

```
print(int(2.0))
print(int(2.3))
print(int(2.9))

def floor(x):
    if x < 0:
        return min(int(x), int(x) - 1)
    else:
        return int(x)

print()

print(floor(2.0))
print(floor(2.3))
print(floor(2.9))
print(floor(-2.0))
print(floor(-2.3))
```

```
2
2
2

2
2
2
-3
-3
```

Natürlich könntest du jetzt einwerfen, dass die ganze Geschichte mit der Gaußklammer oder meinentwegen floor für unsere Aufgabenstellung nicht notwendig ist, aber was tut man nicht alles, um in neuen Bahnen zu denken, nicht wahr?