Suchalgorithmen

1 Einleitung

Neben dem Sortieren ist das Suchen eine der häufigsten Tätigkeiten, die ein Computer ausführt.

Miller und Ranum beschreiben dies in ihrem Buch wie folgt:

> Searching is the algorithmic process of finding a particular item in a collection of items.

Das Resultat einer Suche kann verschieden ausfallen:

- Wenn man nur wissen möchte, *ob* das gesuchte Objekt in der Menge vorhanden ist, genügt als Rückgabewert `True` oder `False`.
- Falls das gesuchte Objekt modifiziert werden soll, möchte man wissen, *wo* das Element innerhalb der Datenstruktur zu finden ist, sofern es überhaupt darin liegt. Möglicherweise tritt ein Wert auch mehrfach auf, so dass mehrere Positionen ermittelt werden müssen.

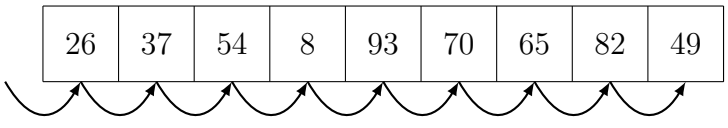
2 Sequentielle Suche

Voraussetzung

Die Daten sind in einer Datenstruktur abgelegt, in der – mit Ausnahme des ersten Elements – jedes Element Nachfolger von genau einem anderen Element ist.

Diese Art von Datenorganisation wird in Python durch Listen oder Tupel realisiert. Durch einen Index greift man auf die einzelnen Werte zu.

Der Algorithmus



Beginnend mit der ersten Position prüft man der Reihe nach jedes Element, bis man entweder gefunden hat, wonach man sucht oder bis man das Listenende erreicht hat, was bedeutet, dass das Element nicht vorhanden ist.

Implementation

```
def linearSearch(L, item):
    for i in range(0, len(L)):
        if L[i] == item:
            return True
    # Falls das Element nicht gefunden wurde:
    return False
```

Laufzeitanalyse der sequentiellen Suche

	Best Case	Average Case	Worst Case
Element $\in$ Liste	$O(1)$	$O(n)$	$O(n)$
Element $\notin$ Liste	$O(n)$	$O(n)$	$O(n)$

Tabelle 1: Laufzeitkomplexität in O-Notation

2.1 Binäre Suche

Voraussetzungen

Wieder sind die Daten in einer sequentiellen Datenstruktur abgelegt. Zusätzlich sind die Werte in aufsteigender Reihenfolge sortiert.

Beispiel

Suche in L = [8, 26, 37, 49, 54, 65, 70] nach 54:

1. Bestimme die mittlere Position: $m = \lfloor (6+0)/2 \rfloor = 3$
2. L[3] = 49 < 54 $\Rightarrow$ suche oberhalb von 49 weiter:
   [8, 26, 37, 49, *54*, *65*, *70*]
3. Bestimme die mittlere Position: $m = \lfloor (4+6)/2 \rfloor = 5$
4. L[5] = 65 > 54 $\Rightarrow$ suche unterhalb von 65 weiter:
   [8, 26, 37, 49, *54*, 65, 70]
5. Bestimme die mittlere Position: $m = \lfloor (4+4)/2 \rfloor = 4$
6. L[4] = 54 = 54 $\Rightarrow$ Das Element wurde gefunden und das Verfahren endet.

Iterative Implementation

```
def binarySearch(L, item):
    lower = 0
    upper = len(L)-1
    while (lower <= upper):
        mid = (lower+upper) // 2
        if item < L[mid]:
            upper = mid - 1
        elif item > L[mid]:
            lower = mid + 1
        else:
            return True
    return False
```

Laufzeitanalyse der binären Suche

Bei jedem Schritt wird die Menge der zu durchsuchendenn Elemente etwa halbiert. Im schlimmsten Fall ist nach k Schritten noch eine Liste mit einem Element übrig, die das gesuchte Objekt enthält oder nicht.

Anzahl Schritte	ungefähre Anzahl Elemente
1	$n/2^1$
2	$n/2^2$
...	...
k	$n/2^k$

$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2(n) \Rightarrow T(n) \in O(\log n)$

Bemerkung

Damit das Verfahren der binären Suche angewendet werden kann, müssen die zu durchsuchenden Daten in geordneter Form vorliegen.

Ist dies nicht der Fall, müssen sie zuvor mit einem Sortierverfahren in die richtige Reihenfolge gebracht werden. Die Kosten dafür betragen mindestens $O(n \log n)$.

Da die Laufzeitkomplexität fürs Sortieren bereits grösser als die der sequentiellen Suche, lohnt es sich nicht, die Daten extra zu sortieren, nur um die schnellere binäre Suche anwenden zu können.

3 String-Matching

Eine weitere zentrale Suchaufgabe besteht darin, ein Textmuster (*pattern*) p in einer Zeichenkette (*string*) t zu finden.

Dabei sollen hier Algorithmen betrachtet werden, die nach *exakten* Übereinstimmungen (*matches*) suchen.

Zeichenketten und Muster werden als Listen repräsentiert, deren Elemente die einzelnen Zeichen sind.

Anwendungen

- Textverarbeitungsprogramme
- Untersuchung von DNA- und Proteinsequenzen in der Bioinformatik
- Erkennung von Plagiaten
- Virenscanner

3.1 Naive Methode (Brute Force)

Beispiel

Es soll untersucht werden, ob das Textmuster

`ABBA`

in der Zeichenkette

`ABABBCABBACB`

vorkommt.

`T[0]==P[0]`: True

A	B	A	B	B	C	A	B	B	A	C	B
A	B	B	A								

`T[1]==P[1]`: True

A	**B**	A	B	B	C	A	B	B	A	C	B
A	**B**	B	A								

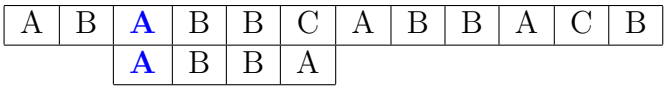
`T[2]==P[2]`: False

A	**B**	**A**	B	B	B	A	B	B	A	C	B
A	**B**	**B**	A								

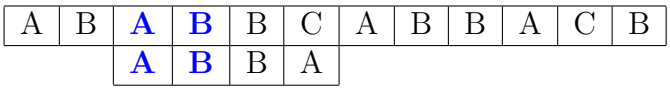
`T[1]==P[0]`: `False`

A	**B**	A	B	B	C	A	B	B	A	C	B
	A	B	B	A							

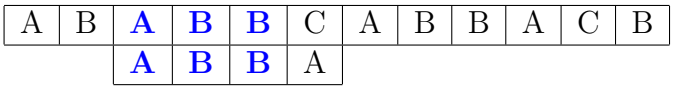
T[2]==P[0]: True



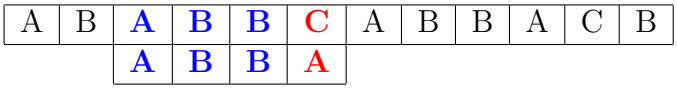
T[3]==P[1]: True



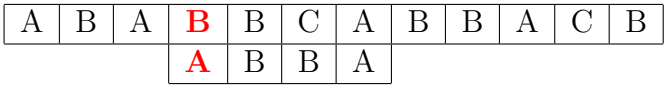
T[4]==P[2]: True



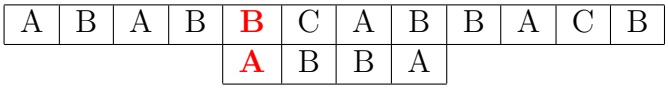
T[5]==P[3]: False



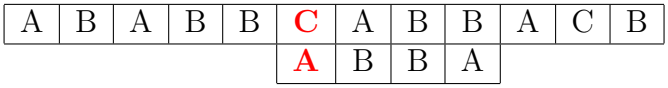
T[3]==P[0]: False



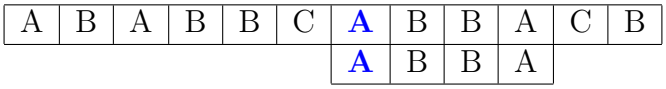
T[4]==P[0]: False



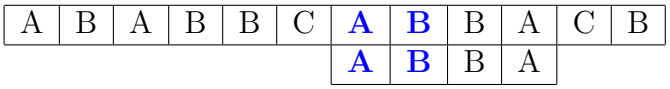
T[5]==P[0]: False



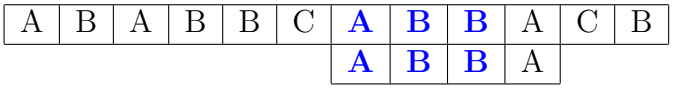
T[6]==P[0]: True



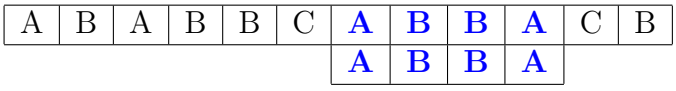
T[7]==P[1]: True



T[8]==P[2]: True



T[9]==P[3]: True



Analyse (Worst Case)

Text: `aaaaaaa` ($n = 7$ Zeichen)

Pattern: `aab` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	a	Vergleiche
a	a	b					3
	a	a	b				3
		a	a	b			3
			a	a	b		3
				a	a	b	3
							(7-3+1)*3=15

Allgemein:

$O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Python-Implementation

```
1  def naiveMatch(pattern, text):
2      n = len(text)
3      m = len(pattern)
4      for i in range(0, n-m+1):
5          j = 0
6          while j < m and text[i+j] == pattern[j]:
7              j += 1
8          if j==m:
9              return i # gefunden
10     return -1 # erfolglose Suche
```

3.2 Der Boyer-Moore-Horspool-Algorithmus

Übersicht

Beim Boyer-Moore-Horspool-Algorithmus wird vor der eigentlichen Mustersuche anhand des Textmusters eine Tabelle aufgebaut. Diese enthält bei Nichtübereinstimmung die maximale Zeichenzahl, um die das Muster verschoben werden kann, ohne dass man dadurch einen Treffer verpasst.

Damit dies funktioniert, werden die Vergleiche rückwärts, das heisst von rechts nach links durchgeführt.

Beispiel

Es soll wieder untersucht werden, ob das Textmuster

`ABBA`

in der Zeichenkette

`ABABBCABBACB`

vorkommt.

Bad Character Table

Text: $t =$ `ABABBCABBACB` der Länge $n = 12$

Pattern: $p =$ `ABBA` der Länge $m = 4$

Alphabet: $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ (jedes Symbol in $t \cup p$)

Für jedes Symbol $\sigma \in \Sigma$ wird $m = 4$ in die Tabelle eingetragen:

Symbol	A	B	C
Shift	4	4	4

Für $j = 0, 1, \ldots, m-2$ wird in der Tabelle der Wert für das Symbol $p[j]$ mit $m - j - 1$ überschrieben.

$j = 0$ $p[0] =$ `A`: Shift $= 4 - 0 - 1 = 3$

Symbol	A	B	C
Shift	3	4	4

$j = 1$ $p[1] =$ `B`: Shift $= 4 - 1 - 1 = 2$

Symbol	A	B	C
Shift	3	2	4

$j = 2$ $p[2] =$ `B`: Shift $= 4 - 2 - 1 = 1$

Symbol	A	B	C
Shift	3	1	4

(Für die letzte Position erfolgt keine Berechnung.)

`T[3]==P[3]`: False

A	B	A	**B**	B	C	A	B	B	A	C	B
A	B	B	**A**								

Wegen Nichtübereinstimmung wird um 1 Zeichen verschoben:

`T[4]==P[3]`: False

A	B	A	B	**B**	C	A	B	B	A	C	B
	A	B	B	**A**							

Wegen Nichtübereinstimmung wird um 1 Zeichen verschoben:

`T[5]==P[3]`: False

A	B	A	B	B	**C**	A	B	B	A	C	B
		A	B	B	**A**						

Wegen Nichtübereinstimmung wird um 4 Zeichen verschoben:

`T[9]==P[3]`: True

A	B	A	B	B	C	A	B	B	**A**	C	B
						A	B	B	**A**		

`T[8]==P[2]`: True

A	B	A	B	B	C	A	B	**B**	**A**	C	B
						A	B	**B**	**A**		

`T[7]==P[1]`: True

A	B	A	B	B	C	A	**B**	**B**	**A**	C	B
						A	**B**	**B**	**A**		

`T[6]==P[0]`: True

A	B	A	B	B	C	**A**	**B**	**B**	**A**	C	B
						A	**B**	**B**	**A**		

Implementation

```
def badCharacterTable(pattern, symbols):
    m = len(pattern)
    D = dict()
    for i in range(0, len(symbols)):
        D[symbols[i]] = m
    for i in range(0, m-1):
        D[pattern[i]] = m-i-1
    return D
```

```
def bmhMatch(pattern, text, symbols):
    bct = badCharacterTable(pattern, symbols)
    n = len(text)
    m = len(pattern)
    i = 0 # Position im Text
    while (i < n-m+1):
        j = m-1 # letzte Position im Muster
        while (j>-1 and pattern[j] == text[i+j]):
            j = j-1
        if j == -1: # alle Zeichen matchen
            return i
        i = i + bct[text[i+m-1]] # shift aufgrund BCT
    return -1 # erfolglose Suche
```

Abgesehen vom Aufwand für den Aufbau der Tabelle $O(|\Sigma|+m)$ waren $3+4 = 7$ Vergleiche nötig.

Worst Case-Analyse

Text: `aaaaaa` ($n = 6$ Zeichen)

Pattern: `baa` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	a	a				3
	b	a	a			3
		b	a	a		3
			b	a	a	3
						(6-3+1)*3=12

Allgemein:

$O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Solche Text-Muster-Strukturen sind jedoch eher die Ausnahme!

Best Case-Analyse

Text: `aaaaaaa` ($n = 6$ Zeichen)

Pattern: `bbb` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	b	b				1
			b	b	b	1
						(6//3)*1=2

Allgemein: $O(n/m)$

Auch solche Text-Muster-Strukturen sind eher die Ausnahme.

Average Case-Analyse

Ricardo Baeza-Yates und Mireill Régnier haben in ihrem Artikel in der Fachzeitschrift *Theoretical Computer Science* 1992 gezeigt, dass die Komplexität des Boyer-Moore-Horspool-Algorithmus im Mittel $O(n)$ ist.

Bemerkung

Der hier vorgestellten Boyer-Moore-Horspool-Algorithmus (BMH) ist eine Vereinfachung des Boyer-Moore-Algorithmus' (BM), der zusätzlich zur Bad Character Table allfällige Übereinstimmungen am Ende des Musters einbezieht, um es beim ersten Mismatch eventuell noch weiter nach rechts zu verschieben.

Wenn man den Fachartikeln im Internet Glauben schenkt, so ist für natürliche Sprachen der BMH-Algorithmus dem BM-Algorithmus überlegen. Dies liegt offenbar daran, dass der BM-Algorithmus mehr Aufwand zur Verschiebung des Suchmusters betreibt, was sich in einer grösseren Anzahl von Anweisungen niederschlägt. Siehe z. B.:

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61442/