Sortieralgorithmen

Inhaltsverzeichnis

1 Insertion Sort

Beschreibung

Die erste Zahl links bildet eine „sortierte“ Liste der Länge 1.

Diese Liste wird um das nächste Element erweitert. Wir sortieren dieses Element in die vorangehende Liste ein, indem wir es mit dem links stehenden Element vergleichen. Ist es grösser, vertauschen wir die Plätze dieser beiden Elemente, sofern das Element nicht an der vordersten Position steht. Andernfalls ist es am richtigen Platz.

Nun fahren wir mit den folgenden Elementen in gleicher Weise fort, bis wir das letzte Element der Liste korrekt einsortiert haben.

Diese Methode wird Sortieren durch Einfügen (*Insertion sort*) genannt.

Beispiel

13	**5**	2	21	8	Vergleiche	Vertauschungen
5	**13**	**2**	21	8	1	1
2	**5**	**13**	**21**	8	2	2
2	**5**	**13**	**21**	**8**	1	0
2	**5**	**8**	**13**	**21**	3	2
					7	**5**

Python-Implementation

```
def insertionsort(L):
    '''Sortiert die Liste L aufsteigend/inplace '''
    n = len(L)
    # äussere Schleife: Beginne mit L[1]
    for i in range(1, n):
        cand = L[i] # das einzusortierende Element
        j = i # aktuelle Position von 'cand'
        # innere Schleife: Gibt es ein vor dem
        # Kandidaten liegendes grösseres Element,
        # vertausche es mit dem Kandidaten und rücke
        # j um eine Position nach links:
        while(j > 0 and L[j-1] > cand):
            (L[j], L[j-1]) = (L[j-1], L[j])
            j = j-1
```

2 Selection Sort

Beschreibung

Ein anderes Rezept zum Sortieren besteht darin, zuerst das kleinste Element in der gesamten Liste zu suchen und mit dem Element an der ersten Stelle zu tauschen (sofern das kleinste Element nicht schon an erster Stelle steht. Danach sucht man in der Teilliste, die beim zweiten Element beginnt das kleinste Element und tauscht es, sofern nötig, mit dem Element an der zweiten Stelle. Danach suchen wir in der Teilliste vom dritten bis zum letzten Element wieder das kleinste Element und vertauschen es gegebenenfalls mit dem dritten usw.

Diese Methode wird Sortieren durch Auswählen (*selection sort*) genannt.

Beispiel

13	5	**2**	21	8	Vergleiche	Vertauschungen
2	**5**	13	21	8	4	1
2	**5**	13	21	**8**	3	„1“
2	**5**	**8**	21	**13**	2	1
2	**5**	**8**	**13**	**21**	1	1
					10	**4**

Beachte

Bei Selectionsort wird im i-ten Schritt grundsätzlich das i-te Element mit dem kleinsten Element der bei i beginnenden Teilliste vertauscht, selbst wenn das i-te Element schon das kleinste Element ist und daher mit sich selbst vertauscht wird. Diese Vertauschung liesse sich nur durch einem zusätzlichen Vergleich vermeiden, was die Zeitersparnis jedoch wieder zunichte machen würde.

Python-Implementation

```
def selectionsort(L):
    '''Sortiert die Liste L aufsteigend/inplace'''
    n = len(L)
    # nach dem i-ten Durchlauf sind die Elemente an
    # den Positionen 0, 1, ..., i sortiert.
    for i in range(0, n-1):
        minpos = i # Positionskandidat
        # Bestimme den Index des kleinsten Elements
        # aus L[i], L[i+1], ..., L[n-1]:
        for j in range(i+1, n):
            if L[j] < L[minpos]:
                minpos = j
        # Vertausche L[i] mit dem kleinsten Element:
        (L[minpos], L[i]) = (L[i], L[minpos])
```

3 Bubble Sort

Beschreibung

Zuerst vergleicht man die ersten beiden Elemente und bringt sie, falls nötig in die richtige Reihenfolge. Dann vergleicht man das zweite und dritte Element und bringt sie, falls nötig, in die richtige Reihenfolge. Auf diese Weise „treibt“ man durch fortgesetzte Vertauschungen das grösste Element ans Ende der Liste. Danach beginnt man wieder von vorne und treibt durch Vertauschungen das zweitgrösste Element an die zweitletzte Stelle. Auf diese Weise fährt man fort, bis man am Schluss noch die ersten beiden Elemente in die richtige Reihenfolge bringen muss.

Die Bezeichnung Bubblesort rührt daher, weil die grösseren Elemente, wie Luftblasen im Wasser, ans Ende der Liste „aufsteigen“.

Beispiel

13	**5**	2	21	8	Vergleiche	Vertauschungen
5	**13**	**2**	21	8	1	1
5	2	**13**	**21**	8	1	1
5	2	13	**21**	**8**	1	0
5	**2**	13	8	**21**	1	1
2	**5**	**13**	8	**21**	1	1
2	5	**13**	**8**	**21**	1	0
2	**5**	8	**13**	**21**	1	1
2	**5**	**8**	**13**	**21**	1	0
2	**5**	**8**	**13**	**21**	1	0
2	**5**	**8**	**13**	**21**	1	0
					10	**5**

Python-Implementation

```
def bubblesort(L):
    '''Sortiert eine Liste L aufsteigend/inplace'''
    n = len(L)
    # nach dem i-ten Durchlauf steht das grösste
    # Element von L[0], L[1], ... L[n-i-1]) an
    # Position n-i-1:
    for i in range(0, n-1):
        # durchlaufe j=0, j=1, ..., j=n-i-2 und
        # tausche das Element L[j] 'nach oben',
        # falls L[j] > L[j+1]:
        for j in range(0, n-i-1):
            if (L[j] > L[j+1]):
                (L[j], L[j+1]) = (L[j+1], L[j])
```

4 Quicksort

Quicksort wurde von C. Antony R. Hoare zu Beginn der 60er-Jahre entwickelt [The Computer Journal (1962) 5, Seiten 10–15].

Es basiert auf dem Prinzip von *divide and conquer* (*Teile und herrsche*) und sortiert in den meisten Fällen sehr schnell.

Schritt 1

Zuerst wird ein Pivotelement („Scharnier“, „Angelpunkt“) bestimmt. Dazu wählt man ein Element, das möglichst in der Mitte der zu sortierenden Daten liegt (*Median*).

In der Praxis werden für die Wahl des Pivots verschiedene Möglichkeiten:

- Wähle den Median des ersten, mittleren und letzten Elements. (*median of three*)
- Wähle ein zufälliges Element. (*randomized quicksort*)

Der Einfachheit halber wählen wir in den Übungen und Beispielen jeweils das Element an der mittleren Position $\lfloor (n-1)/2 \rfloor$.

Schritt 2

Zerlege die Liste so in zwei Teillisten, dass alle Elemente, die kleiner als das Pivotelement sind, in der linken Teilliste und alle Elemente, die grösser als das Pivot sind, in der rechten Teilliste versammelt sind. Am Ende von Schritt 2 befindet sich das Pivot-Element an seiner korrekten Position.

Schritt 3

Wiederhole Schritt 1 und 2 rekursiv auf den Teillisten unterhalb und oberhalb der Position des zuletzt bestimmten Pivotelements. Die Rekursion bricht ab, wenn nur noch eine Teiliste der Länge 1 zu sortieren ist.

Beispiel (5,13,2,9,21,8,7,1)

5	13	2	**9***	21	8	7	1
5	1	2	**9***	21	8	7	13
5	1	2	7	21	8	**9***	13
5	1	2	7	**9***	8	21	13
5	1	2	7	8	**9***	21	13
5	1	**2***	7	8		21	13
2*	1	5	7	8		21	13
1	**2***	5	7	8		21	13
		5	**7***	8		21	13
						21*	13
						13*	21

Analyse

- Die Behebung aller Fehlstände in Schritt 2 kostet maximal $n/2$ Schritte, ist also in linearer Zeit zu bewältigen.
- Beim rekursiven Aufruf des Verfahrens auf der linken und rechten Teilliste kommt es darauf an, ob das gewählte Pivotelement die Liste in zwei etwa gleich grosse Teile zerlegt. Wenn ja, benötigt man asymptotisch $\mathcal{O}(\log_2(n))$ Rekursionsschritte.

Kombiniert man diese beiden Laufzeiten, erhält man eine Laufzeit im Best Case und Average Case von $\mathcal{O}(n \log_2(n))$.

Falls jeweils das grösste oder kleinste Element der (Teil-)Liste gewählt wird, benötigt man $\mathcal{O}(n)$ Rekursionsschritte, was zu einer quadratischen Zeitkomplexität führt.

Code

```
# Quicksort-Verfahren
def quicksort(L):
    qsHelper(L, 0, len(L)-1)

def qsHelper(L, leftIndex, rightIndex):
    # Wenn Teilliste mehr als 1 Element hat:
    if (leftIndex < rightIndex):
        pivot = fragment(L, leftIndex, rightIndex)
        qsHelper(L, leftIndex, pivot-1)
        qsHelper(L, pivot+1, rightIndex)

def fragment(L, leftIndex, rightIndex):

    # Pivotelement: (abgerundeter) mittlerer Index
    p = (leftIndex + rightIndex) // 2

    # Index auf die linke und rechte Startposition
    i = leftIndex
    j = rightIndex

    # Solange die Indizies verschieden sind:
    while (i < j):

        # gehe bis zu ersten Fehlstand links:
        while (i < pivot and L[i] <= L[pivot]):
            i = i+1

        # gehe bis zum ersten Fehlstand rechts:
        while (j > pivot and L[j] >= L[pivot]):
            j = j-1

        # Behebe Fehlstand durch Austausch ...
        (L[i], L[j]) = (L[j], L[i])

        # ... und passe ggf. die Pivotposition an:
        if i == pivot:
            pivot = j
        else:
            if j == pivot:
                pivot = i

    return pivot
```