Sortieralgorithmen

Jan Pöschko

18. Januar 2007

Inhaltsverzeichnis

1 Problemstellung

1.1 Definition

Input: Folge (Feld) von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Folge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

In der Praxis geht es selten nur um einzelne Werte, sondern eher um ganze *records*. Jeder record besteht aus einem *key*, nach dem sortiert wird, und sog. *satellite data*, die mit dem key verschoben wird. (Allerdings muss die satellite data nicht wirklich verschoben werden, wenn man mit Pointern arbeitet.)

Bei der Analyse der Algorithmen wollen wir aber die implementationstechnischen Details ausblenden, um uns auf die eigentliche „Qualität“ der Algorithmen konzentrieren zu können.

1.2 Warum Sortieren?

- Informationsaufbereitung
- Teil eines anderen Algorithmus (z.B. Computergraphik: Sortieren nach Sichtbarkeit)
- verwendete Techniken generell interessant (z.B. *divide and conquer*)
- historisches Problem
- untere Laufzeitschranke des Problems beweisbar, kann für andere Probleme verwendet werden

2 Einfache Sortieralgorithmen

2.1 Bubblesort

Prinzip: Der Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente und vertauscht sie, falls sie in der falschen Reihenfolge vorliegen. Dieser Vorgang wird solange wiederholt, bis keine Vertauschungen mehr nötig sind. Hierzu sind in der Regel mehrere Durchläufe erforderlich.

Je nachdem, ob auf- oder absteigend sortiert wird, steigen die größeren oder kleineren Elemente wie Blasen im Wasser (daher der Name) immer weiter nach oben, d.h. an das Ende der Reihe. Auch werden immer zwei Zahlen miteinander in „Bubbles“ vertauscht.

Der Algorithmus hat einen Zeitbedarf von $O(n^2)$. Er arbeitet *in place*, d.h. es wird kein Speicherbedarf zusätzlich zum gegebenen Feld benötigt. Außerdem ist er *stabil*, d.h. Elemente mit gleichem Wert behalten ihre ursprüngliche Reihenfolge.

Algorithm 1 Bubblesort

```
procedure BUBBLESORT(A)
    for i ← 1 to n − 1 do
        for j ← 1 to i − 1 do
            if A[j] > A[j + 1] then
                exchange A[j] ↔ A[j + 1]
            end if
        end for
    end for
end procedure
```

2.1.1 Varianten

- Baut man eine Abbruchbedingung ein, falls in der inneren Schleife keine Vertauschung notwendig ist, ist der Algorithmus für fast sortierte Felder sehr schnell. Im *worst case* benötigt er aber natürlich trotzdem $O(n^2)$ Zeit.
- SHAKERSORT oder COCKTAILSORT: Die Richtung des Schleifendurchlaufs wird bei jeder Iteration geändert. Am Zeitbedarf ändert das aber nichts.

2.2 Insertionsort

Prinzip: Der Algorithmus entnimmt dem Feld schrittweise ein Element und fügt es an der richtigen Stelle im sortierten Teil ein. Dazu wird es mit allen bisher sortierten Elementen verglichen, bis die richtige Stelle gefunden ist.

Das Verfahren ist vergleichbar mit der Weise, in der man Spielkarten sortiert: Man nimmt sich jeweils eine Karte vom Stapel und fügt sie den sortierten Karten in der Hand hinzu.

Algorithm 2 Insertionsort

```
procedure INSERTIONSORT(A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j − 1
        while i > 0 and A[i] > key do
            A[i + 1] ← A[i]
            i ← i − 1
        end while
        A[i + 1] ← key
    end for
end procedure
```

Die Laufzeit des Algorithmus beträgt sowohl im schlechtesten als auch im durchschnittlichen Fall $O(n^2)$; Im *best case* ist sie allerdings nur $O(n)$. Wie BUBBLESORT

arbeitet auch InsertionSort in place und ist stabil.

2.2.1 Verwandte Sortieralgorithmen

- Shellsort (von Donald L. Shell im Jahre 1959 entwickelt): versucht den Nachteil auszugleichen, dass Elemente oft über weite Strecken verschoben werden müssen. Daher wird erst im letzten Schritt das gesamte Feld sortiert, vorher aber alle zweiten Elemente, davor wiederum alle vierten usw. Das führt zur Laufzeit $O(n \log(n)^2)$. Besser ist z.B., die Folge $1, 4, 13, 40, \ldots$ zu verwenden, da hier weniger Überlappungen entstehen
- Combsort (von S. Lacey und R. Box 1991 vorgestellt): Die Lücke wird jedesmal um den Faktor 1.3 verkleinert; Der Algorithmus endet, wenn die „Kammweite“ 1 ist und keine Vertauschungen mehr stattgefunden haben.

2.3 Schranke für einfache Algorithmen

Satz. *Jeder Sortieralgorithmus, der nur benachbarte Elemente vertauscht, benötigt* $\Omega(n^2)$ *Vergleiche im worst case.*

Beweisidee. Wir betrachten eine verkehrt sortierte Folge. Das erstes Element benötigt dann $n - 1$ Vertauschungen, um ans Ende der Folge zu kommen. Dabei ändert sich die Stellung der restlichen Elemente untereinander nicht. Das zweite Element benötigt nun $n - 2$ Vertauschungen usw.

Insgesamt benötigen wir also mindestens

$$\sum_{i=1}^{n}(i-1) = \Omega(n^2)$$

Vertauschungen. □

3 Optimale Sortieralgorithmen

3.1 Mergesort

MergeSort wurde erstmals 1945 durch John von Neumann vorgestellt.

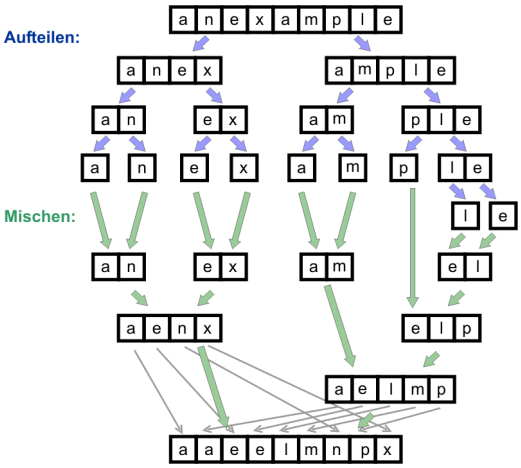
Der Algorithmus benötigt $O(n \log n)$ Zeit. Werden geeignete Datenstrukturen verwendet (z.B. verkettete Listen), arbeitet er in place. Er ist jedenfalls stabil.

3.2 Quicksort

Quicksort wurde 1962 von Sir Charles Anthony Richard Hoare entwickelt.

Der Sortieralgorithmus hat zwar eine worst case Laufzeit von $\Theta(n^2)$ hat, in der Praxis ist er aber dennoch oft einer der besten Algorithmen. Die erwartete Laufzeit ist $\Theta(n \log n)$, wobei die „versteckten konstanten Faktoren“ sehr niedrig sind.

Abbildung 1: Arbeitsweise von Mergesort



Algorithm 3 Mergesort

```
procedure MergeSort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A,p,q,r)
    end if
end procedure
```

3.2.1 Beschreibung

Quicksort basiert auf dem *divide and conquer* Prinzip.

Divide: Zerlege das Feld $A[p \dots r]$ in zwei (möglicherweise leere) Teilfelder $A[p \dots q-1]$ und $A[q+1 \dots r]$, sodass die Elemente in ersterem $\leq A[q]$ und die Elemente in zweiterem $\geq A[q]$ sind.

Conquer: Sortiere die zwei Teilfelder durch rekursiven Aufruf von Quicksort.

Die Wahl des *Pivot-Elements* $A[q]$ ist also entscheidend. Mögliche Ansätze sind

- das erste, letzte oder mittlere Element des Feldes,
- den Median dieser drei Elemente,
- den Median der gesamten Liste
- oder ein zufälliges Element zu wählen.

Algorithm 4 Quicksort

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure

function PARTITION(A, p, r)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange A[i] ↔ A[j]
        end if
    end for
    exchange A[i + 1] ↔ A[r]
    return i + 1
end function
```

3.2.2 Analyse

Wir wollen die *randomisierte* Version von QUICKSORT analysieren, bei der das Pivot-Element zufällig gewählt wird.

Im worst case wird das Pivot-Element (trotz zufälliger Wahl) immer am Rand gewählt, sodass das Feld in ein großes Feld und ein einzelnes Element aufgespalten wird. Der Algorithmus „entartet“ somit, und die worst-case-Laufzeit ist $\Theta(n^2)$.

Um die durchschnittliche Laufzeit zu analysieren, machen wir zuerst folgende Beobachtung: Bei jedem Aufruf von PARTITION wird ein Pivot-Element gewählt, das danach in keinem Aufruf von PARTITION oder QUICKSORT mehr beachtet wird. Somit kann es insgesamt nicht mehr als n Aufrufe von PARTITION geben. Der Zeitbedarf jedes dieser Aufrufe ist proportional zu der Anzahl der Iterationen bzw. Vergleiche (Zeile 4). Sei also X die Anzahl der Vergleiche in PARTITION, dann ist die Laufzeit von QUICKSORT $O(n + X)$.

Zur Vereinfachung der Analyse nennen wir die Elemente des Feldes $z_1, z_2, \ldots, z_n$, wobei z_i das i-kleinste Element sein soll. Definiere mit

$$X_{ij} = I\{z_i \text{ wird verglichen mit } z_j\}$$

eine Indikatorfunktion. Gesucht ist nun der Erwartungswert von

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Dieser lässt sich aufgrund der Linearität des Erwartungswerts berechnen als

$$\begin{aligned} E(X) &= E\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E(X_{ij}) \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P\{z_i \text{ wird verglichen mit } z_j\}. \end{aligned}$$

Wir wissen: Sobald ein Pivot-Element x mit $z_i < x < z_j$ gewählt wird, werden z_i und z_j zu keinem späteren Zeitpunkt mehr verglichen. Wird umgekehrt z_i oder z_j als Pivot-Element gewählt, werden sie genau einmal miteinander verglichen. Daraus ergibt sich

$$\begin{aligned} P\{z_i \text{ wird verglichen mit } z_j\} =& P\{z_i \text{ oder } z_j \text{ ist erstes Pivot-Element in } Z_{ij}\} \\ =& P\{z_i \text{ ist erstes Pivot-Element in } Z_{ij}\} \\ &+ P\{z_j \text{ ist erstes Pivot-Element in } Z_{ij}\} \\ =& \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}. \end{aligned}$$

Zusammenfassend gilt also

$$\begin{aligned} E(X) &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) \\ &= O(n \log n). \end{aligned}$$

Somit gilt auch für QUICKSORT insgesamt die durchschnittliche Laufzeit $O(n \log n)$.

3.3 Schranke für vergleichbasierte Algorithmen

Satz. *Jeder auf Vergleichen basierende Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im worst case.*

Beweisidee.
- Vergleiche erzeugen *Entscheidungsbaum*
- Jede der $n!$ Permutationen der Folge entspricht einem Blatt
- Höhe des Baumes: $h \geq \log(n!) = \Omega(n \log n)$

□

4 Sortieren in linearer Zeit

4.1 Counting sort

Wir nehmen hier an, dass jeder der n zu sortierenden Werte eine ganze Zahl im Bereich 0 bis $k \in \mathbb{N}$ ist.

Prinzip: Für jedes Element x bestimmen wir die Anzahl der Elemente, die kleiner als x sind. Mit dieser Information können wir x direkt an seiner richtigen Position im sortierten Feld einfügen.

COUNTING-SORT benötigt $\Theta(k+n)$ Zeit, für den Fall $k = O(n)$ also $\Theta(n)$. Das Verfahren ist stabil und hat einen zusätzlichen Speicherbedarf von $O(k+n)$.

Algorithm 5 Counting sort

```
procedure COUNTING-SORT(A, B, k)
    for i ← 0 to k do
        C[i] ← 0
    end for
    for j ← 1 to length(A) do
        C[A[j]] ← C[A[j]] + 1
    end for
                                ▷ C[i] enthält nun die Anzahl der Elemente gleich i
    for i ← 1 to k do
        C[i] ← C[i] + C[i − 1]
    end for
                        ▷ C[i] enthält nun die Anzahl der Elemente kleiner oder gleich i
    for j ← length(A) downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
    end for
end procedure
```

4.2 Radix sort

Prinzip: Sortiere die Zahlen nach ihren Ziffern. Würde man allerdings — wie man wahrscheinlich intuitiv verfahren würde — mit der höchstwertigen Ziffer anfangen, müsste man sich zahlreiche Zwischenergebnisse merken. Fängt man hingegen mit der niedrigstwertigen Ziffer an, hat man das Problem nicht. Allerdings muss der Algorithmus, mit dem nach den einzelnen Ziffern sortiert wird, stabil sein, damit die bisherige Ordnung nicht zerstört wird!

Algorithm 6 Radix sort

```
procedure RADIX-SORT(A,d)
    for i ← 1 to d do
        Verwende einen stabilen Algorithmus zur Sortierung von A bezüglich Ziffer i
    end for
end procedure
```

4.3 Bucket sort

BUCKET-SORT läuft in linearer Zeit, wenn die Eingabewerte gleichverteilt sind.

Algorithm 7 Bucket sort

```
procedure BUCKET-SORT(A)
    n ← length(A)
    for i ← 1 to n do
        Insert A[i] into list B[⌊A[i]⌋]
    end for
    for i ← 0 to n − 1 do
        Sort list B[i] mittels INSERTIONSORT
    end for
    Hänge die Listen B[0], B[1], ..., B[n − 1] zusammen
end procedure
```

List of Algorithms

Abbildungsverzeichnis

Literatur

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, Second Edition. MIT Press, Cambridge, 2001.

[2] Donald E. Knuth, The Art of Computer Programming, Volume 3. Sorting and Searching. Addison-Wesley, 1998.

[3] Udi Manber, Introduction to Algorithms. A Creative Approach. Addison-Wesley, 1989.

[4] Wikipedia: Sortieralgorithmus (`http://de.wikipedia.org/wiki/Kategorie:Sortieralgorithmus`), 14. Jänner 2007.