Sortieren

Christian Kuester

26. Juni 2001

1 Sortieren

1.1 Einleitung

Es gibt eine grosse Anzahl Sortieralgorithmen. In der Klasse derer, die darauf beruhen, dass in einem Schritt jeweils zwei Elemente vertauscht werden, gibt es grundsätzlich zwei verschiedene Arten an das Problem heranzugehen. Die erste, naive Möglichkeit sind zwei geschachtelte Schleifen, so kommt man bei all diesen auf eine quadratische Laufzeit. Variationen dieser Herangehensweise sind zum Beispiel das hier vorgestellte InsertionSort oder auch Selection- und Bubblesort. Jedoch ist es auch möglich Zahlenfelder mit dem Divide and Conquer Prinzip in die richtige Ordnung zu bringen. Die berühmten Vertreter Quick- und MergeSort sollen auch hier vorgestellt werden. Ziel soll hier aber nicht nur die Vorstellung des reinen Algorithmus sein, sondern die Frage der effizienten Implementation. Mit ein wenig Mühe lassen sich deutliche Gewinne in den realen Laufzeiten von Sortieralgorithmen erzielen. Als Beispiel sei kurz QuickSort genannt, der bekannterweise Worst-Case Fälle hat, die ihn von O(n*log(n)) auf $O(n^2)$ zurückfallen lassen. Wir werden später sehen, dass man durch gute Programmiertechnik diese Fälle abfangen kann.

1.2 InsertionSort

Zunächst jedoch eine kurze Einführung an Hand des Beispiels InsertionSort (Sortieren durch Einfügen). Hier wird die Idee dieses Artikels deutlich, Programmiertricks zu verwenden, die sich überaus positiv auf die Geschwindigkeit auswirken.

InsertionSort lässt sich am besten vergleichen mit der Art, wie Kartenspieler ihre Karten aufnehmen und in ihre Hand einsortieren. Hat man eine Anzahl Karten auf der Hand und nimmt die nächste, dann wandern die solange von einer Seite zur anderen, bis die richtige Position gefunden ist. Dort wird die Karte dann eingefügt. Hier eine kleine Demonstration, wie ein vierelementiges Feld von dem Algorithmus sortiert wird. Der Balken "|" ist zeigt an, bis wohin die Sortierung bereits vorgeschritten ist.

```
3 | 1   4   2
1   3 | 4   2
1   3   4 | 2
1   2   3   4
```

Die erste Implementation ist ganz leicht uns besteht nur aus drei einfachen Zeilen.

```
for (i = 0; i < 30000; i++)
    for (j = i; j > 0 && my_array[j-1] > my_array[j]; j--)
        this->swapint(my_array[j-1], my_array[j]);
```

Unser "i", also der Zähler der äusseren Schleife ist gerade unser Balken aus dem vorherigen Beispiel. Es gibt also die Stelle an, an der der bereits sortierte Teil endet. Diese drei Zeilen sind die ganze Implementation. Aber dies ist nicht besonders effizient, wie man als etwas erfahrener Programmierer sofort feststellt, denn ein Funktionsaufruf zum Vertauschen der Elemente ist nicht nötig. Dies erzeugt unnötige Sprünge im Programm und belastet den Stack. Vertauschen wir nun die Elemente lieber "inline". Das führt zu folgendem Programm.

```
for (i = 0; i < 30000; i++)
    for (j = i; j > 0 && my_array[j-1] > my_array[j]; j--) {
        t = my_array[j];
        my_array[j] =  my_array[j-1];
        my_array[j-1] = t;
    }
```

Der Gewinn der Laufzeit lässt sich wirklich sehen. Bei ca. 30.000 Zahlen sortierte diese Implementation das Array auf meinem Rechner ganze zehn Sekunden schneller. (Übrigens optimieren gute Compiler Code so, dass bestimmte Funktionsaufrufe einfach zu "inline" gemacht werden. Aber will man sich nicht unbedingt darauf verlassen und ja wahrscheinlich auch nicht den Compiler sein Programm schreiben lassen.) Aber dies ist noch nicht alles, was man ändern kann. Zwar lässt sich nicht auf den ersten Blick mehr erkennen, wo man noch etwas sparen kann. Aber es wird sicher an dem nächsten Quellcode sichtbar.

```
for (i = 0; i < 30000; i++) {
    t = my_array[i];
    for (j = i; j > 0 && my_array[j-1] > t; j--)
        my_array[j] = my_array[j-1];
    my_array[j] = t;
}
```

Zwei Zuweisungen sind aus dem inneren Schleifenkörper nach aussen entfernt worden. Auf den ersten Blick scheint es vielleicht nicht mehr korrekt, aber es ist. Die Zuweisung, die nach oben gewandert ist, wies vorher nämlich in der inneren Schleife "t" jedesmal den selben Wert zu. Nämlich den Wert des Elements, das gerade an die richtige Stelle eingefügt werden soll. Die Zuweisung, die nach unten aus der inneren Schleife entfernt wurde braucht auch nur einmal für jedes absickernde Element durchgeführt zu werden, denn es ist die Zuweisung, die es an die richtige Stelle in sortierten Teil einfügt. So haben wir schon eine Menge an Laufzeit gespart. Kann man jedoch noch etwas herausholen? Bedingt ja, abhängig von der Frage: "Wie gut kenne ich meinen Compiler?" Die Programme sind alle in C++ geschrieben.

```
for (i = 0; i < 30000; ++i) {
    t = my_array[i];
```

```
        for (j = i; j > 0 && my_array[j-1] > t; --j)
            my_array[j] = my_array[j-1];
        my_array[j] = t;
    }
```

Wie man sieht, sind hier alle "x++" durch "++x" ersetzt worden, Für die Schleifen spielt das keine Rolle. Manche C++ Compiler jedoch produzieren aus letzterem schnelleren Code. (Quellenangaben siehe unten).

So haben wir hier etwas gute Programmiertechnik angewandt und sind auf erstaunliche Laufzeitunterschiede gestossen. An den Kosten $O(n^2)$ von InsertionSort können all unserer Tricks natürlich nichts ändern. Aber dafür gibt es andere Algorithmen, die sich dem Divide and Conquer Prinzip bedienen. Wie zum Beispiel Quicksort.

1.3 QuickSort

Zum ersten Mal beschrieben wurde QuickSort von C.A.R. Hoare 1962 im Computer Journal. QuickSort teilt das zu sortierende Array in zwei kleinere Teilstücke und sortiert diese rekursiv. Die Teilstücke werden anhand des sogenannten Pivot-Elements vorsortiert. Wählen wir immer das erste Element eines Stückes als Pivot-Element werden alle anderen damit verglichen und anhand ihrer Grösse entweder ins linke oder rechte Teilstück getauscht. Im Quellcode sieht das folgendermassen aus.

```
Qsort1::qsort1( int my_array[], int l, int u)
{
    if (l >= u) return;
    int m = l;
    for (int i = l+1; i <= u; ++i) {
        if (my_array[i] < my_array[l])
           this->swapint(my_array[++m], my_array[i]); }
    this->swapint(my_array[l], my_array[m]);
    this->qsort1(my_array, l, m-1);
    this->qsort1(my_array, m+1, u);
}
```

In den Zeilen 5-6 wird der Zerteilungsschritt durchgeführt, er vertauscht die Elemente verglichen mit dem Pivot-Element entweder nach rechts oder links. Die Laufzeit für grosse Felder ist dank O(n*log(n)) erheblich besser, als das naive InsertionSort. Jedoch hat diese Implementation ein unglückliches Worst-Case verhalten. Ein Test mit einem Feld, dass ausschliesslich aus gleichen Zahlen besteht, braucht in etwa solange wie InsertionSort für ein zufälliges Feld. Grund dafür ist der entartete Rekursionsbaum, der entsteht, wenn man das Feld sortiert. In der kompletten Rekursion bleibt ein Teilstück auf jeden Fall immer leer. So haben wir also eine Laufzeit von $O(n^2)$. Ein intelligenter Zerteilungsschritt ist also nötig um das abzufangen. Dieser Quellcode zeigt es.

```
Qsort3::qsort3( int my_array[], int l, int u )
{
    if (l >= u) return;
    int t = my_array[l]; int i = l; int j = u+1;
```

```
    while(true)
    {
        do { i++; } while ( i <= u && my_array[i] < t );
        do { j--; } while ( my_array[j] > t );
        if ( i > j ) break;
        this->swapint( my_array[i], my_array[j] );
    }
    this->swapint( my_array[l], my_array[j] );
    this->qsort3( my_array, l, j-1);
    this->qsort3( my_array, j+1, u);
}
```

Der Zeilungschritt hier, der in den Zeilen 6-11 stattfindet, ist besser, er geht von links und von rechts solange zur Mitte, bis sich beide Zähler überschneiden. Dies verhindert das Problem bei einem Feld aus gleichen Zahlen. Leider ist diese Art der Problemlösung immer noch nicht ganz Worst-Case frei. Betrachten wir die Sortierung eines Feldes, das schon (vor-)sortiert ist, dann haben wir ein ähnliches Bild wie zuvor. Der Rekursionsbaum ist wieder entartet, da wir das Pivot-Element so wählen, dass es immer das erste des Teilstückes ist. So bekommen wir in diesem Fall auch komplett leere Teilstücke auf einer Seite. Die Laufzeit ist in diesem Fall auch $O(n^2)$. Leider ist es natürlich wahrscheinlich, dass ein solcher Fall einmal auftritt. Umgehen lässt sich dieses Problem eigentlich ziemlich einfach. Die Idee ist, das Pivot-Element nicht fix zu wählen, sondern zufällig zu ermitteln.

```
Qsort4::qsort4( int my_array[], int l, int u )
{
    if ((u-l) < this->cutoff) return;
    int swapper = this->randint(l, u);
    int temp = 0;
    temp = my_array[swapper]; my_array[swapper] = my_array[l];
    my_array[l] = temp;
    int t = my_array[l]; int i = l; int j = u+1;
    while (true) {
        do { i++; } while (i <= u && my_array[i] < t);
        do { j--; } while (my_array[j] > t);
        if (i > j) break;
        temp = my_array[i];
        my_array[i] = my_array[j];
        my_array[j] = temp;
    }
    this->swapint(my_array[l], my_array[j]);
    this->qsort4( my_array, l, j-1);
    this->qsort4( my_array, j+1, u);
}
```

in Zeile 4 sehen wir die Funktion, die uns ein zufälliges Element aus dem Feld liefert. Tatsächlich lässt sich damit der Worst-Case für ein vorsortiertes Feld elegant umgehen. Ein weiterer Trick ist hier auch zur Anwendung gekommen. Es fällt auf, dass die Abbruchbedingung für die Rekursion nun anders ist. Sie hält bei einer bestimmten Tiefe an, je nach Wahl der Variable "cutoff". So entstehen nach der Ausführung von QuickSort noch kleine unsortierte Teilstücke im

Feld. Man lässt nun noch einen InsertionSort darüber laufen. Dies hilft ebenfalls noch etwas Laufzeit zu sparen, da man an Rekursionsschritten spart. Ein anderer Algorithmus, der auch nach dem Divide and Conquer Prinzip arbeitet ist MergeSort. Dieser hat bekanntermassen keine Worst-Case Fälle wie QuickSort, jedoch lassen sich dort ebenfalls Verbesserungen herbeiführen.

1.4 MergeSort

Das Prinzip von MergeSort ist dem von QuickSort ähnlich. Hier wird das zu sortierende Feld ebenfalls erstmal in zwei kleiner Teilstücke zerlegt. Diese werden aber erstmal nicht sortiert. Zunächst wird solange rekursiv in Teilstücke zerlegt, bis einzelne oder Paare von Elementen vorliegen. Dann beginnt der "Merge"-Teil. Er benötigt bei Mergesort unbedingt ein temporäres Array in das die Elemente richtig einsortiert werden. Dann wird das temporäre Array in das ursprüngliche zurückkopiert. Ist die Rekursion beendet, dann ist das Array komplett sortiert.

```
Msort1::RekMergeSort( int my_array[], int Lo, int Hi ) {
    if (Lo < Hi) {
        int split = (Lo + Hi + 1)/2;
        this->RekMergeSort(my_array, Lo, split-1);
        this->RekMergeSort(my_array, split, Hi);
        int alen = (Hi-Lo+1); int *temp_arr = new int[alen];
        for (int i = 0, j = Lo,  k = split; i < alen; i++)
            if ((k > Hi)||(j < split) && (my_array[j] < my_array[k])){
                temp_arr[i] = my_array[j]; j++; }
            else {
                temp_arr[i] = my_array[k]; k++; }
        for (int i=0; i < alen; i++) my_array[Lo+i] = temp_arr[i];
    }
}
```

In den Zeilen 4 und 5 wird Mergesort rekursiv aufgerufen. Der schwierige Schritt des Mischens befindet sich ind den Zeilen 7-12. Hier wird, nicht unmittelbar erkennbar, das Einsortieren in das temporäre Array, dass in Zeile 6 erstellt wurde, vorgenommen. Am Schluss werden die sortierten Zahlen in das Orginalarray zurückkopiert. Es wird klar, dass Mergesort auf jeden Fall mehr Speicherplatz benötigt, als das Quicksortverfahren. Die Geschwindigkeit unterscheidet sich nur geringfügig von dem eben vorgestellten Worst-Case freien Quicksort. Die Entscheidung, welches Verfahren zur Anwendung kommen sollte, hängt auch vom verfügbaren Speicherplatz ab. Jedoch bleibt beim externen Sortieren meist nur die Wahl von Mergesort. Man kann das oben aufgeführte Mergesort noch optimieren, wenn man nicht in jedem Rekursionsschritt ein neues Array erstellt. Die wirkt sich positiv auf die Geschwindigkeit aus, benötigt aber natürlich ein zweites Array, in dem zwischengespeichert werden kann in der Grösse des Orginalarrays.

```
Msort2::RekMergeSort( int my_array[], int Lo, int Hi )
{
    if (Lo >= Hi) return;
    if ((Hi - Lo) <= 2){
```

```
        int mid = (Hi+Lo)/2;
        if (my_array[Lo]  > my_array[mid])
           this->swapint(my_array[Lo], my_array[mid]);
        if (my_array[mid] > my_array[Hi])
           this->swapint(my_array[mid], my_array[Hi]);
        if (my_array[Lo]  > my_array[mid])
           this->swapint(my_array[Lo], my_array[mid]); }
    else {
        int split = (Lo + Hi +1)/2;
        int len = Hi-Lo+1;
        this->RekMergeSort(my_array, Lo, split-1);
        this->RekMergeSort(my_array, split, Hi);
        for (int i=0, j=Lo, k=split; i < len; i++){
            if ((k > Hi)||(j < split) && (my_array[j] < my_array[k])){
                this->temp_arr[i] = my_array[j]; j++; }
            else { this->temp_arr[i] = my_array[k]; k++; }
        }
        for (int i=0; i < len; i++) my_array[Lo+i] = this->temp_arr[i];
    }
}
```

Hier sieht man, dass nicht in jedem Schritt ein neues Array erzeugt wird, sondern sich der Algorithmus eines vorher erstellten Arrays bedient. Zusätzlich wird aber noch eine weitere Idee ausgenutzt. Die Rekursion wird bei drei Elementen abgebrochen. Die Elemente werden schnell Inline getauscht, das spart Rekursionsschritte. Diese Implementation ist geringfügig schneller, als die vorherige. Ist aber sicher vorzuziehen.

1.5 Quellen und Literatur

1. Jon Bentley, Programming Pearls, Addison-Wesley
2. Donald Knuth, The Art of Computer Programming - Volume 3, Addison-Wesley
3. Bob Sedgewick, Algorithmen, Addison-Wesley
4. Effective C++, Addison-Wesley