1 Sortier-Algorithmen

Ein grundlegendes Problem, das in der Informatik ausführlich untersucht wurde und zu dem eine ganze Reihe verschiedener Algorithmen bekannt sind, ist das Sortier-Problem. Gegeben sind n Elemente, die als $a[0]$, $a[1]$, $\ldots$, $a[n-1]$ in einem Feld a gespeichert sind. Weiterhin nehmen wir an, daß es eine totale Ordnung "$\leq$" auf den Elementen gibt (d.h. je zwei Elemente können verglichen werden, aus $x \leq y$ und $y \leq z$ folgt $x \leq z$, und aus $x \leq y$ und $y \leq x$ folgt $x = x$). Beispielsweise könnten die Elemente ganze Zahlen mit der üblichen $\leq$-Relation sein, aber auch Strings mit der lexikographischen Ordnung ("ab" $\leq$ "aber" $\leq$ "ach" $\leq$ "bald"). Ziel beim Sortieren ist, die Elemente des Feldes a so umzuordnen, daß am Ende $a[0] \leq a[1] \leq \cdots \leq a[n-1]$ gilt. Ein Sortier-Algorithmus ist ein Algorithmus, der dieses Problem löst.

In der Regel nimmt man an, daß zwei Elemente in konstanter Zeit miteinander verglichen oder vertauscht werden können, und betrachtet die Anzahl der Vergleiche und Vertauschungen, die ein Algorithmus für die Sortierung von n Elementen schlimmstenfalls benötigt, als Komplexität des Algorithmus. Es ist bekannt, daß jeder Algorithmus in diesem Modell im Worst-Case mindestens Komplexität $\Omega(n \log n)$ hat. (In anderen Modellen, wo man spezielle Annahmen über die Repräsentation der zu sortierenden Elemente macht, gibt es zum Teil bessere Komplexitätsabschätzungen, die aber nicht direkt vergleichbar sind. Ein Beispiel ist der weiter unten beschriebene Radix-Sort.)

Viele einfache Sortier-Algorithmen haben im Worst-Case Komplexität $O(n^2)$. Es sind aber auch mehrere Algorithmen bekannt, die immer mit $O(n \log n)$ Vergleichen und Vertauschungen auskommen. Andere bekannte Algorithmen (z.B. Quicksort) haben zwar im Worst-Case Komplexität $O(n^2)$, im Durchschnitt bei zufälligen Eingaben aber nur $O(n \log n)$. (Genauere Informationen zu den im folgenden vorgestellten Sortier-Algorithmen können z.B. in den Algorithmen-Büchern von Robert Sedgewick nachgeschlagen werden.)

1.1 Bubble-Sort

Eine ganz naheliegende Methode zum Sortieren ist die folgende: In jeder Iteration geht man das Feld von vorne nach hinten durch; an Stelle i vergleicht man $a[i]$ mit $a[i+1]$, und in dem Fall, daß $a[i] > a[i+1]$ ist, vertauscht man die beiden. Das macht man so lange, bis bei einer Iteration überhaupt keine Vertauschung mehr nötig war. Dieses Verfahren nennt man Bubble-Sort.

In der ersten Iteration wird auf jeden Fall schon mal das größte Element ganz nach hinten gebracht, spätestens in der zweiten Iteration das zweitgrößte Element an die vorletzte Stelle, usw. Nach spätestens $n-1$ Iterationen ist das Feld sortiert. In der j-ten Iteration muß man mit i gar nicht mehr bis ganz hinten laufen, da die hinteren $j-1$ Elemente schon richtig sortiert sind.

Da man $O(n)$ Iterationen mit je $O(n)$ Vergleichen und Vertauschungen macht, hat Bubble-Sort Komplexität $O(n^2)$. Bubble-Sort läßt sich sehr einfach implementieren und ist recht schnell, wenn das Eingabefeld kurz oder bereits einigermaßen vorsortiert ist.

1.2 Selection-Sort

Ein weiteres recht naheliegendes Verfahren ist Selection-Sort, das wie Bubble-Sort Komplexität $O(n^2)$ hat. Die Idee ist, in der ersten Iteration das kleinste Element im Feld zu bestimmen und nach vorne zu bringen, in der zweiten Iteration das kleinste Element im Restfeld zu bestimmen und an die zweite Stelle zu bringen, usw. Man hat $n-1$ Iterationen, die jeweils $O(n)$ Vergleiche und eine Vertauschung benötigen.

1.3 Insertion-Sort

Die Idee beim Insertion-Sort ist, wieder in Iterationen vorzugehen und sicherzustellen, daß nach der i-ten Iteration die Elemente, die ursprünglich an Positionen 0 bis i im Feld standen, sortiert an Positionen 0 bis i stehen. In der i-ten Iteration muß man also das Element $a[i]$ richtig in das schon sortierte Anfangsfeld $a[0], \ldots, a[i-1]$ einfügen. Auch hier ist die Komplexität $O(n^2)$, da in jeder der $n-1$ Iterationen bis zu $O(n)$ Vertauschungen nötig sind (im schlimmsten Fall landet $a[i]$ an Stelle 0 im Feld, und alle Elemente $a[0], \ldots, a[i-1]$ müssen um Eins nach hinten geschoben werden).

Insertion-Sort ist gut, wenn das Feld bereits einigermaßen vorsortiert ist oder wenn das Feld sehr wenige Elemente enthält.

1.4 Shell-Sort

Eine interessante und sehr knifflig zu analysierende Variante von Insertion-Sort ist Shell-Sort. Hier werden Teilfelder des Feldes a betrachtet, die aus jedem h-ten Element bestehen (für geeignete Werte von h). Für ein Feld a mit $n = 10$ Elementen gibt es für $h = 4$ zum Beispiel die folgenden Teilfolgen:

$$
\begin{array}{l}
a[0], a[4], a[8] \\
a[1], a[5], a[9] \\
a[2], a[6] \\
a[3], a[7]
\end{array}
$$

Eine Iteration von Shell-Sort besteht nun darin, für einen bestimmten Wert von h alle entsprechenden Teilfolgen mittels Insertion-Sort zu sortieren. Von Iteration zu Iteration wird der h-Wert verkleinert. Wenn $h = 1$ erreicht ist, gibt es nur noch eine einzige Teilfolge, die dem ganzen Feld entspricht, und nach deren Sortierung ist das Feld sortiert.

Als günstige Wahl für die h-Werte hat sich erwiesen, in jedem Schritt den Wert von h durch 3 zu dividieren und abzurunden. Um den Startwert von h zu berechnen, startet man mit $h = 1$ und wiederholt die Anweisung $h = 3 * h + 1$, bis $h \geq n$ gilt; dann teilt man den Wert einmal durch 3 und rundet ab: das ergibt den Startwert.

Es läßt sich (auf komplizierte Weise!!) zeigen, daß Shell-Sort Komplexität höchstens $O(n^{1.5})$ hat. Man kann sich leicht überzeugen, daß es sehr schwierig ist, Worst-Case-Beispiele für Shell-Sort zu konstruieren, bei denen der Algorithmus viele Vergleiche und Vertauschungen macht.

1.5 Quicksort

Einer der am häufigsten benutzten und in vielen Fällen schnellsten Sortier-Algorithmen ist Quicksort (erfunden von C.A.R. Hoare im Jahr 1960). Der Algorithmus arbeitet rekursiv nach dem Divide-and-Conquer-Prinzip (das Problem in kleinere Probleme zerschlagen und diese dann rekursiv lösen). Um ein Feld zu sortieren, wird zunächst ein bestimmtes Element p des Feldes, genannt Pivot-Element, ausgewählt. Dann wird das Feld so umgeordnet, daß links (bei den kleinen Indizes, etwa in $a[0], \ldots, a[i-1]$) Elemente stehen, die kleiner oder gleich p sind, daß $a[i] = p$ gilt, und daß rechts (in $a[i+1], \ldots, a[n-1]$) diejenigen Elemente stehen, die größer als p sind. Nun reicht es, die beiden Teilfelder $a[0], \ldots, a[i-1]$ und $a[i+1], \ldots, a[n-1]$ zu sortieren: dadurch ergibt sich automatisch ein insgesamt korrekt sortiertes Feld. Die Sortierung der beiden Teilfelder erledigt Quicksort rekursiv.

Wenn einer der rekursiven Quicksort-Aufrufe ein Teilfeld sortieren soll, das sehr wenige Elemente enthält (weniger als k für eine geeignete Konstante k, z.B. $k = 10$), so wird meistens ein anderer Algorithmus (z.B. Bubble-Sort oder Insertion-Sort) zur Sortierung dieses Teilfeldes verwendet, da diese einfacheren Algorithmen für kurze Felder in der Regel effizienter sind.

Ein sehr wichtiger Gesichtspunkt bei der Implementierung von Quicksort ist die Wahl des Pivot-Elementes. Quicksort erreicht eine gute Laufzeit (Komplexität $O(n \log n)$), wenn bei jedem Aufteilungsschritt die beiden erzeugten Teilfelder etwa gleich groß sind. Ist dagegen bei vielen Aufteilungen eines der beiden Teilfelder viel größer als das andere, so ist die Komplexität $O(n^2)$. Es ist also wichtig, ein Pivot-Element zu wählen, das etwa "in der Mitte" aller Elemente liegt.

Deterministische Strategien, die als Pivot-Element das Element in einer festen Position des Feldes verwenden (z.B. das erste Element, das letzte Element, oder das Element an Position $n/2$), sind bei recht natürlichen Eingaben (z.B. vorsortiert oder absteigend vorsortiert) oft sehr schlecht. Viel besser ist es, entweder das Pivot-Element zufällig zu wählen (d.h. alle Elemente $a[0], \ldots, a[n-1]$ sind gleich wahrscheinlich) oder sogar (*median-of-three*-Methode) drei zufällige Elemente x, y und z auszuwählen und das mittlere der drei als Pivot zu verwenden (d.h. wenn $x \leq y \leq z$ gilt, so wählt man y als Pivot). Bei zufälliger Pivot-Wahl ist die erwartete Komplexität von Quicksort $O(n \log n)$.

Nach der Wahl des Pivot-Elementes muß das Feld entsprechend umgeordnet werden (d.h. in zwei Teilfelder partitioniert). Wichtig ist, daß man diese Partitionierung effizient in Zeit $O(n)$ implementieren kann. Man wandert einfach mit zwei Zeigern (Indizes) durch das Feld: einem Index l, der bei 0 beginnt und nach rechts wandert, und einem Index r, der bei $n-1$ beginnt und nach links wandert. Man wandert mit beiden Indizes, bis $a[l] > p$ und $a[r] \leq p$ gilt, und vertauscht dann $a[l]$ und $a[r]$ (oder ist fertig, wenn schon $l > r$ gilt). Der Partitionierungsschritt läuft damit (im Prinzip) wie folgt ab:

```
1  while a[l] ≤ p do l++;
2  while a[r] > p do r--;
3  if l < r then
4        swap(a[l], a[r]);
5        goto 1;
6  else "fertig";
```

Bei der Implementierung muß man zusätzlich sehr aufpassen, daß die Partitionierung auch in Spezialfällen richtig funktioniert, z.B. wenn alle Elemente im Feld gleich sind (dann würde in obigem Pseudo-Code die erste while-Schleife über den rechten Rand des Feldes hinauslaufen).

Ein Problem bei Quicksort ergibt sich in dem Fall, daß sehr viele Elemente gleich dem Pivot-Element sind; im oben beschriebenen Verfahren werden diese Elemente bei der Partitionierung alle dem linken Teilfeld zugeordnet, was den Nachteil hat, daß z.B. bei einem aus identischen Elementen bestehenden Feld das rechte Teilfeld völlig leer ist. In diesem Fall empfiehlt es sich, dafür zu sorgen, daß alle Elemente, die gleich p sind, nach der Partitionierung in der Mitte des Feldes stehen, so daß die rekursiven Aufrufe für das linke und rechte Teilfeld überhaupt keine Elemente gleich p bearbeiten müssen.

1.6 Merge-Sort

Nun kommen wir zu einem Sortier-Algorithmus, der im Unterschied zu den bisher beschriebenen Algorithmen Komplexität $O(n \log n)$ im Worst-Case sicherstellt: Merge-Sort. Die Idee ist wie bei Quicksort ein rekursives Divide-and-Conquer-Verfahren. Um immer Komplexität $O(n \log n)$ zu erreichen, muß man dafür sorgen, daß die Teilfelder für die beiden rekursiven Aufrufe etwa gleich groß sind. Bei Merge-Sort macht man das, indem man einfach das Feld $a[0], \ldots, a[n-1]$ in die Teilfelder $a[0], \ldots, a[n/2-1]$ und $a[n/2], \ldots, a[n-1]$ aufteilt (ohne Umordnung der Elemente). Nach den rekursiven Aufrufen liegen die beiden Teilfelder sortiert vor. Um ein insgesamt sortiertes Feld zu bekommen, müssen nun lediglich die beiden Teilfelder richtig gemischt werden, d.h. die Elemente der Teilfelder müssen entsprechend verzahnt werden. Dieses Mischen läßt sich leicht in Zeit $O(n)$ implementieren; damit erhält man für die Laufzeit von Merge-Sort die Rekursionsgleichung $T(n) = 2 \cdot T(n/2) + cn$ für eine geeignete Konstante c. Die Lösung dieser Rekursionsgleichung ist bekanntlich $O(n \log n)$.

1.7 Radix-Sort

Nachdem alle bisher beschriebenen Algorithmen nur mit Vergleichen und Vertauschungen arbeiten, kommen wir nun abschließend zu einem Algorithmus, der in die Repräsentationen der Elemente „hineinschaut“. Wir nehmen an, daß jedes Element x aus einer Reihe von Stellen $x_1, \ldots, x_k$ besteht, wobei sich die Stellen x_i als Zahlen zwischen 0 und $M-1$ repräsentieren lassen und die Stellenanzahl k für jedes Element verschieden sein kann. Weiter muß die gewünschte Ordnung der Elemente so sein, daß $x \leq y$ genau dann gilt, wenn $x_1, x_2, \ldots$ lexikographisch kleiner oder gleich $y_1, y_2, \ldots$ ist.

Wenn man beispielsweise ein Feld mit 32-Bit-Integer-Zahlen (ohne Vorzeichen) sortieren will, könnte man sich diese Zahlen als aufgebaut aus 4 Stellen zu je 8 Bit denken. (In diesem Fall haben alle Elemente die gleiche Stellenanzahl.) Wenn die zu sortierenden Elemente Strings sind, dann ist es natürlich, die einzelnen Zeichen der Strings als Stellen zu je 8 Bit (ASCII-Code) zu interpretieren. (Wenn die Strings verschiedene Längen haben, ist das also ein Beispiel für Elemente mit unterschiedlichen Stellenanzahlen.) In beiden Fällen haben wir $M = 2^8 = 256$.

Um die Grundidee von Radix-Sort zu verstehen, betrachten wir zunächst den Fall, daß wir ein Feld a mit n Zahlen aus dem Bereich von 0 bis $M-1$ sortieren wollen (d.h. die Zahlen haben nur eine Stelle). Wir verwenden ein Feld *count* mit M Zählern, die wir mit 0 initialisieren. Dann gehen wir die n Elemente des Eingabefeldes der Reihe nach durch und erhöhen bei Element $a[i]$ die Variable $count[a[i]]$ um Eins. Das benötigt nur lineare Zeit, und anschließend haben wir in *count* die Information, wie oft jedes einzelne Element vorkommt. Nach Ausführung der Schleife

for $i = 1$ **to** $M-1$ **do**
  $count[i] = count[i] + count[i-1];$
od

ist der Wert $count[i]-1$ genau der größte Index, an dem im sortierten Feld ein Element gleich i stehen wird. Nun können wir die Elemente aus dem Feld a mittels folgender Schleife in sortierter Reihenfolge in ein Feld b umkopieren:

for $i = n-1$ **downto** 0 **do**
  $b[count[a[i]]-1] = a[i];$
  $count[a[i]] = count[a[i]] - 1;$
od

Anschließend kann man das Feld b wieder in das Feld a zurückkopieren, wenn man das Ergebnis in a benötigt. Diese Methode wird auch als *distribution counting* bezeichnet, weil man zuerst zählt, wie oft jedes Element vorkommt. Nachteil ist, daß man ein zweites Feld b der selben Größe wie a benötigt. Vorteil ist, daß das Verfahren Laufzeit $O(M+n)$ hat und daher in dem Fall, daß M kleiner als $n \log n$ ist, schneller als vergleichsbasierte Sortierverfahren ist.

Radix-Sort ist die Verallgemeinerung von *distribution counting* auf Elemente, die aus mehreren Stellen bestehen. Nehmen wir zunächst an, daß alle Elemente die gleiche Stellenanzahl k besitzen. Wie man sich leicht überlegen kann, funktioniert dann folgendes Sortierverfahren, das *(Straight) Radix Sort* genannt wird:

for $j = k$ **downto** 1 **do**
  sortiere a mittels *distribution counting* nach der j-ten Stelle;
od

Wichtig ist dabei, daß bei der Sortierung nach der j-ten Stelle die Reihenfolge derjenigen Elemente, die in Stelle j identisch sind, nicht verändert wird. (Man nennt einen solchen Sortierschritt *stabil.*) Der oben angegebene Pseudo-Code für *distribution counting* ist stabil, so daß er problemlos übernommen werden kann.

Die Laufzeit für Radix-Sort ist offensichtlich $O(k \cdot (M+n))$, da jede der k Iterationen Zeit $O(M+n)$ benötigt. Beim Sortieren von 32-Bit-Integers ($k = 4$ Stellen zu je 8 Bit, d.h. $M = 256$) ist das Verfahren also offensichtlich gut geeignet.

Wenn die einzelnen Elemente verschiedene Stellenanzahlen haben können, muß man sich überlegen, wie man Radix-Sort passend modifizieren kann. Eine einfache Lösung wäre natürlich, bei Elementen mit kleinen Stellenanzahlen einfach so viele Nullen hinten anzuhängen, bis alle Elemente gleich viele Stellen haben. Wenn zum Beispiel beim Sortieren von Strings wenige Strings mehrere hundert Zeichen lang sind, ist das offensichtlich etwas ungünstig, weil für jede der vielen Stellen einmal *distribution counting* gemacht wird. Daher sollte man versuchen, sich bessere Lösungen zu überlegen!