RunSort

Ein stabiles, sicheres und ordnungsverträgliches

höheres Sortierverfahren

Prof. Dr. W. Kowalk

Universität Oldenburg, kowalk@uni-oldenburg.de

Abstrakt

Wir stellen einen Algorithmus vor, dessen maximale Anzahl von Vergleichen und Kopierungen durch die Anzahl der Elemente mal dem Logarithmus der Anzahl der Läufe (*run*) beschränkt ist. Der Algorithmus ist daher sicher, stabil und ordnungsverträglich. Da er einfach zu verstehen und zu implementieren ist und seine Laufzeit mit der von Quicksort konkurrieren kann, ist er in vielen Anwendungsfällen eine gute Wahl.

Abstract

We present a new Algorithm the maximum number of compares and copies is limited by the number of elements times the logarithm of the number of runs in the unsorted sequence; the algorithm is save, stable and adaptive. It is easily understood and to be implemented with performance comparable to Quicksort; thus it is a good choice in many cases.

Einleitung

Sortieren ist weiterhin eine wichtige Anwendung für Algorithmen. Dabei kommt es in heutigen Systemen in erster Linie auf Geschwindigkeit und Sicherheit an; Speicherplatz ist eher von untergeordneter Bedeutung, da gegenwärtige Systeme meistens über sehr viel Speicherplatz verfügen. Außerdem werden Datensätze in modernen Programmiersprachen wie Java meistens nicht mehr direkt im Speicher sondern auf dem Heap abgelegt und nur noch über Referenzen angesprochen, so dass zusätzlicher 'Speicher' i.d.R. nur eine zusätzliche Referenz bedeutet, was z.B. bei doppelt verketteten Listen zum Standardzusatzaufwand gehört.

Sortieren wird heute in vielen Anwendungen verwendet. Neben den üblichen Anwendungen in der Datenverwaltung werden z.B. auch in 3D-Programmen Sortieralgorithmen benötigt, um Objekte in einer Reihenfolge abhängig von der Entfernung zu zeichnen, z.B. semitransparente Objekte, bei denen hintere Objekte durch vordere 'durchscheinen'. Hier spielen alle drei genannten Eigenschaften eine wichtige Rolle. Stabilität wird benötigt, um bei gleicher Entfernung eine gegebene Ordnung nicht zu verletzen, da Objekte sonst 'flackern' würden. Ordnungsverträglichkeit wird benötigt, da sich in diesen Anwendungen die Ordnung einer Liste von Objekten nur wenig oder meist gar nicht ändert, so dass die Sortierung, die mehrere zig-Male in der Sekunde durchgeführt werden muss, in den meisten Fällen hinreichend schnell durchgeführt werden kann.

Das hier entwickelte Verfahren ist einerseits im Mittel etwa so schnell wie Quicksort, in einer Listenvariante sogar schneller. Darüber hinaus ist es stabil und ordnungsverträglich, da seine beste Laufzeit $O(N)$ ist, wenn N die Anzahl der Elemente ist. Dabei werden bei vollständig geordneten Listen nur genau N–1 Vergleiche (und keine Vertauschungen) durchgeführt, also die Minimalzahl. Da das Verfahren außerdem sicher ist, ist die Laufzeit garantiert von einer Laufzeitkomplexität von $O(N+N\cdot\log_2 R)$, wenn R die Anzahl der Läufe (*run*) ist, d.h. der vorsortieren Teilfolgen. Deren An-

zahl ist maximal N, so dass die Laufzeitkomplexität im schlechtesten Fall garantiert $O(N+N \cdot \log_2 N)$ ist, wobei $N+N \cdot \log_2 N$ die absolute Anzahl von Vergleichen darstellt; die absolute Anzahl der Kopieroperationen ist i.d.R. merkbar geringer, aber niemals größer.

Konzept des Algorithmus

Das wesentliche Konzept des Algorithmus basiert auf Sortieren durch Mischen (*merge*). Tatsächlich basiert Sortieren durch Mischen i.d.R. auf einer erweiterten Variante von Sortieren durch Einfügen (*InsertionSort*) von längeren vorsortierten Folgen; wichtig ist jedoch – um die logarithmische Laufzeit garantieren zu können – dass immer etwa gleich lange Folgen gemischt werden. Das lässt sich nur erreichen, wenn jeweils nur Folgen gemischt werden, die aus gleich vielen Teilfolgen zusammengefasst werden; dann ist Mischen, wie es klassisch (als 'externes Sortierverfahren') verwendet wird, von der Ordnung $O(N+N \cdot \log R)$, wenn R die Anzahl der Läufe ist.

Ein Problem bereiten jedoch verschiedene Feinheiten der Implementierung eines Misch-Algorithmus, die ihn i.d.R. langsamer machen als innere Sortierverfahren. Das eine ist insbesondere das Erkennen des Endes eine Laufs, was meistens nur durch Vergleich mit dem nächsten Element einer Liste erreicht werden kann; dieses muss beim Mischen für beide zu sortierenden Listen gemacht werden; zusätzlich muss noch mit dem Vergleichselement der anderen Liste verglichen werden, so dass die Anzahl der Vergleiche sich für jeden Sortierschritt verdreifacht. Dieses wird in unserem Verfahren vermieden, so dass die absolute Anzahl der Vergleiche niemals größer als $N \cdot \log_2 N$ ist (ohne einen beliebigen Faktor C, wie in der Landaunotation $O(N \cdot \log N)$ implizit angenommen wird); sie kann aber geringer sein, was z.B. gravierend auch bei invers sortierten Listen zum Tragen kommt.

Ein mögliche Implementierung zum Sortieren durch Mischen (die auch *Binäres Mischen* (*binaryMerge*) bezeichnet wird) verwendet eine rekursive Halbierung der Daten; diese kann jedoch auf vorgegebene Läufe keine Rücksicht nehmen, bzw. nur marginal berücksichtigen, indem ganze Läufe auf ihre Ordnung verglichen werden. In unserer Implementierung wird die Ordnungsverträglichkeit garantiert, indem immer nur ganze Läufe gemischt werden.

Eine andere Implementierung wird *TimSort* genannt. Hier werden die Längen von zu mischenden Teilläufen kontrolliert und immer möglichst gleich lange Läufe gemischt. Die Information über die Längen verschiedener konsekutiver Läufe wird in einem Stack verwahrt, was die Implementierung sicherlich recht kompliziert macht. In unserer Implementierung wird die Länge nur über die Anzahl der Läufe kontrolliert. Das ist sicherlich ungenauer, aber da jeder Lauf mindestens die Länge 1 hat, im Mittel die Länge 2, und nach k Mischvorgängen die Länge mindestes 2^k bzw. 2^{k+1} beträgt, also exponentiell wächst, ist dieses ausreichend 'ausbalanciert', um etwa gleich lange Läufe zu erhalten, auch wenn einzelne Läufe unterschiedlich lang sind.

Weitere Sortieralgorithmen findet man unter (*Sorting algorithm*) in Wikipedia.

Internes Mischen auf Feldern

Wir verwenden ein Feld (**`field`**) und ein Hilfsfeld (**`field2`**). Die Startmethode kann etwa folgendermaßen implementiert werden.

```
Data [] startRunSort(Data field[]) {
      if (field == null) return null ;
      if (field.length <= 1) return field;
      Data field2[] = new Data[field.length];
      int exp = (int)(Math.log(field.length)/Math.log(2)+1);
      runSort( field, field2, 0, exp, true);
      return field;
}
```

Unser neuer Algorithmus heißt `runSort`. Er ruft sich rekursiv selbst auf, und arbeitet die Anzahl der zu sortierenden Teilläufe über den Parameter **`exp`** ab.

```
int runSort(Data field1[], Data field2[], int from, int exp, boolean up){
      if(exp==0)
            if(up) return findRun( field1,         from);
            else   return copyRun( field1, field2, from);
      int next = runSort( field1, field2, from, exp-1, up);
      if(next>=field1.length) return next;
      int end  = runSort( field1, field2, next, exp-1, !up);
      merge( field1, field2, from, next, end, up);
      return end;
}
```

Es werden drei Hilfsprozeduren verwendet, die selbsterklärend sind: **`findRun`** gibt als Ergebnis die Referenz auf das erste Element hinter dem letzten des Laufs ab **`from`** zurück; **`copyRun`** gibt genau wie **`findRun`** als Ergebnis die Referenz auf das erste Element hinter dem letzten des Laufs ab **`from`** zurück; zusätzlich kopiert **`copyRun`** den Lauf ab **`from`** auf das Hilfsfeld. Die Hilfsprozedur **`merge`** mischt die Läufe von **`field1`** in dem Bereiche **`from`** bis **`next-1`** und **`field2`** in dem Bereich **`next`** bis **`end-1`** nach **`field1`** in den Bereich **`from`** bis **`end-1`**.

Die Idee hinter diesem Algorithmus ist die folgende: ist **`exp=0`**, so wird ein (2^0=1) Lauf 'sortiert', d.h. gefunden und dessen Ende zurückgegeben; der Index (Rückgabewert) verweist auf das erste Feld hinter diesem ersten Lauf. Ist **`exp>0`**, so werden zwei Läufe der Länge $2^{\text{exp-1}}$ rekursiv sortiert und mittels ***`merge`*** zusammengemischt, ergeben also einen Lauf der Länge 2^{exp}. Sollte bereits der erste dieser beiden Läufe alle restlichen Elemente sortiert haben, so wird durch Abfrage auf das Feldende dieser rekursive Aufruf vorzeitig beendet. Damit ist bereits die Funktionalität des Programms vollständig dargelegt und verifiziert.

Laufzeitkomplexität

Bei diesem Algorithmus interessiert besonders die Anzahl der Vergleiche und Kopieroperationen, da diese für die Laufzeitkomplexität ausschlaggebend sind. Die einzelnen Prozeduren haben folgende Laufzeitkomplexität: **`findRun`** findet einen Lauf der Länge N_1 nach N_1 Vergleichen und 0 Kopieroperationen; **`copyRun`** findet einen Lauf der Länge N_1 nach N_1 Vergleichen und N_1 Kopieroperationen. **`merge`** benötigt für das Mischen zweier Läufe der Länge N_1 und N_2 maximal N_1+N_2-1 Vergleiche und N_1+N_2 Kopieroperationen. Die Mischbereiche eines Laufs werden über die Indizes **`from`**, **`next`** und **`end`** bestimmt, so dass ein zusätzlicher Schlüsselvergleich nicht benötigt wird.

Die Anzahl von Vergleichen und Kopieroperationen ist bei **`exp=0`** offenbar höchstens N, wenn N die Länge des nächsten Laufs ist; da wir einen Lauf haben, d.h. R=1, ist der maximale Aufwand $N+N\cdot\log_2 R=N+N\cdot\log_2 1=N$. Somit ist die Aussage, dass der maximale Aufwand durch $N+N\cdot\log_2 R$ beschränkt ist, für diesen Fall korrekt. Also gilt für die Laufzeitkomplexität auch $O(N+N\cdot\log_2 R)$.

Wir beweisen jetzt durch vollständige Induktion die Aussage: Sei die Anzahl der Läufe eine Zweierpotenz $R=2^{\text{exp}}$ und sei der maximale Aufwand für das Sortieren mit ***`runSort`*** durch $N+N\cdot$**`exp`** beschränkt; dann gelte die Aussage: Der maximale Aufwand an Vertauschungen und Vergleichen ist durch $N+N\cdot\log_2 R$ beschränkt.

Gelte diese Aussage für alle **`exp`**$<K$ und sei die Anzahl der Läufe $R=2^K$. Dann sortiert der Algorithmus zweimal 2^{K-1} Läufe mit dem maximalen Aufwand $N_1+N_1\cdot(K-1)$ für den ersten und $N_2+N_2\cdot(K-1)$ für den zweiten Lauf, wobei N_1 und N_2 die Längen der sortierten Läufe (die aus 2^{K-1} Läufen gemischt wurden) sind; ***`runSort`*** mischt diese beiden Läufe mit **`merge`** zusammen, dessen Aufwand durch N_1+N_2 beschränkt ist. Der Gesamtaufwand ist also durch

$$N_1+N_1\cdot(K-1) \ + \ N_2+N_2\cdot(K-1) \ + \ N_1+N_2 \ = \ N+N\cdot(K-1)+N \ = \ N+N\cdot K$$

beschränkt, wenn $N=N_1+N_2$. Wegen $K=\log_2 R$, ist der Aufwand durch $O(N+N\cdot\log_2 R)$ beschränkt; der Logarithmus ist hier immer der Zweierlogarithmus.

Es sei noch einmal betont, dass die Grenzen eines Laufs nur einmal am Anfang (in **`findRun`** bzw. **`copyRun`**) bestimmt werden; danach werden diese durch die Indizes errechnet, so dass bei **`merge`** keine Schlüsselvergleiche, die nicht unbedingt notwendig sind, durchgeführt werden müssen. Die

hier berechneten oberen Grenzen geben also die maximale absolute Anzahl an Schlüsselvergleichen und Kopieroperationen an; es werden hier niemals Vertauschungen gezählt!

Tatsächlicher Aufwand

Bei zufällig erzeugten Folgen ist die (mittlere) Länge von Läufen ziemlich genau 2, die Anzahl somit $R=N/2$. Für die Laufzeitkomplexität bedeutet dieses $O(N+N\cdot\log_2(N/2))=O(N\cdot\log_2 N)$, d.h. die Anzahl von Vergleichen und Kopieroperationen ist durch diese Aufwandsformel exakt beschränkt.

Im schlechtesten Fall gibt es N Läufe, wenn jeder Lauf nur ein Element enthält, die Liste also 'umgekehrt' sortiert ist. In diesem Fall ist der Aufwand $O(N+N\cdot\log_2 N)$, also nicht viel schlechter als im mittleren Fall. Allerdings zeigen Messungen (s.u.), dass die Laufzeit sich in diesem Fall effektiv etwa halbiert gegenüber dem vorigen Fall. Wir erklären dieses Phänomen später.

Im besten Fall ist die Folge bereits sortiert; es gibt also nur einen Lauf: $O(N+N\cdot\log_2 1)=O(N)$. Der Aufwand ist also exakt gleich der minimalen Anzahl von Vergleichen, d.h. es wird jedes Element genau einmal verglichen, und niemals kopiert, wie man auch unmittelbar dem Algorithmus entnehmen kann.

Dieses zeigt, dass die Laufzeit für diesen Algorithmus optimal ist; bessere Laufzeiten kann man mit einem $N\cdot\log N$-Algorithmus, der stabil, ordnungsverträglich und sicher ist, kaum erreichen. Im allgemeinen wird die Laufzeit sogar noch etwas besser sein, da viele Kopieroperationen und teilweise sogar Vergleiche gar nicht durchgeführt werden. Zum einen wird nur die Hälfte der Anfangsläufe kopiert, wenngleich zu Anfang alle Elemente verglichen werden. Außerdem wird das Mischen rückwärts von dem Hilfsfeld (2. Lauf) auf das Ausgangsfeld (1. Lauf) durchgeführt. Sollte das kleinste Element des 2. Laufs größer sein als mehrere Elemente des 1. Laufs, so müssen diese ersten Elemente weder kopiert noch verglichen werden. Dieses wird im Mittel mehrere Male geschehen, so dass auch hierdurch die Laufzeit etwas geringer sein wird.

Die Laufzeitkomplexität wurde für eine Anzahl von Läufen, die durch eine Zweierpotenz beschränkt ist, bestimmt. Dieses ist tatsächlich der einzige Nachteil dieses Algorithmus, da das Mischen sehr verschieden langer Teilfolgen i.d.R. deutlich weniger effektiv ist. Dennoch erhält man auch bei solchen 'schiefen' Lauflängen noch immer sehr gute Ergebnisse, so dass die Laufzeit i.d.R. vergleichbar mit der anderer Sortieralgorithmen wie Quicksort ist.

Implementierungsaspekte

Der Algorithmus hat offenbar die Eigenschaft, extrem einfach zu sein. Seine Implementierung ist kaum fehleranfällig und er ist leicht verständlich; seine Laufzeit ist durch $O(N+N\cdot\log_2 N)$ beschränkt, so dass er auch als sicher anzusehen ist. Da Sortieren nur durch Mischen durchgeführt wird, ist der Algorithmus auch stabil implementierbar, so dass er den wesentlichen Aspekten moderner Sortieralgorithmen *stabil* (*stable*), *ordnungsverträglich* (*adaptive*) und *sicher* (*save*) zu sein genügt. Gegenüber anderen Algorithmen hat er somit nur den Nachteil, ein Hilfsfeld zu benötigen, was aber in heutigen Rechensystemen wegen des verfügbaren großen Speicherplatzes kaum noch ins Gewicht fällt. Darüber hinaus lässt sich dieser Algorithmus auch in einer einfach verketteten Listenvariante implementieren, in der er nicht nur noch schneller läuft, sondern auch mit einem einfachen Zeiger genauso viel Speicherplatz benötigt wie in einem einfachen Feld mit Zeigern auf die Datensätze.

In diesem Algorithmus wird bei der Implementierung darauf geachtet, dass die Rekursionstiefe ausreichend ist, indem der Parameter **`exp`** auf $1+\log_2 N \geq \lceil \log_2 N \rceil$ gesetzt wird. Man geht also pessimistischer Weise davon aus, dass die Daten invers sortiert sind. Dieses bedeutet tatsächlich, dass es meistens einen Rekursionsaufruf zu viel gibt. Allerdings bricht der Algorithmus wegen der zusätzlichen Abfrage nach dem ersten Sortierlauf ab, sobald das Feldende erreicht ist, so dass dieses tatsächlich nur einen zusätzlichen Rekursionsaufruf für den mittleren Fall bedeutet. Im besten Fall bei

einer sortierten Liste bedeutet dieses zwar $\log_2 N$ zusätzliche Rekursionsaufrufe, scheint aber wegen der Vorzüge akzeptabel zu sein.

Sollte man aus diesem oder anderen Gründen **`exp`** kleiner wählen als $\lceil \log_2 N \rceil$, so kann man durch überprüfen des 'Rests' der sortierten Folge feststellen, ob sämtliche Läufe behandelt wurden. Dazu ist lediglich der letzte Wert der Variablen **`end`** zu überprüfen; er muss natürlich größer als das Feldende sein. Dieses lässt sich am effizientesten implementieren, indem die erste aufgerufene Prozedur eine eigene Implementierung erhält; da die Prozeduren sehr kurz sind, ist dieses sicherlich kein übermäßiger Aufwand. Sollte dann erkannt werden, dass die Folge nicht vollständig sortiert wurde, kann der Sortiervorgang mit entsprechend neuem Parameter neu aufgenommen werden (es sind ja bereits 2^{exp} Läufe sortiert!)

Die logische Variable **`up`** gibt an, ob das Feld auf das Ausgangsfeld (**`up==true`**) oder auf das Hilfsfeld (**`up==false`**) sortiert werden soll. Solche logischen Abfragen kosten u.U. etwas Laufzeit, könnten also auch durch verschiedene Prozeduren, die die beiden Fälle explizit behandeln, implementiert werden. Unsere Messungen haben allerdings keine Laufzeitunterschiede ergeben, so dass dieses wohl eher eine theoretische Überlegung ist. Eine Alternative wäre es evtl., die beiden Felder beim Aufruf zu vertauschen, was den Algorithmus allerdings deutlich unübersichtlicher machen würde. Wenn es sehr auf die Laufzeit ankommt, wäre dieses aber evtl. noch eine Verbesserungsmöglichkeit.

Der **`merge`**-Algorithmus wurde folgendermaßen implementiert.

```
void merge(Data field1[],Data field2[],int from,int next,int end,boolean up){
  int from1 = next-1, from2 = end-1, to = end-1;
    if(up) {
      while(from1>=from && from2>=next) // sort to field1
        if(field1[from1].key>field2[from2].key) field1[to--] = field1[from1--];
        else                                    field1[to--] = field2[from2--];
      while(from2>=next) field1[to--] = field2[from2--]; // copy rest of field2
    } else {
      while(from1>=from && from2>=next) // sort to field2
        if(field2[from1].key>field1[from2].key) field2[to--] = field2[from1--];
        else                                    field2[to--] = field1[from2--];
      while(from2>=next) field2[to--] = field1[from2--]; // copy rest of field1
    }
}
```

Die Läufe werden somit solange kopiert, bis das Hilfsfeld leer ist. Sollte dieses der Fall sein (**`from2<next`**), so stehen die anderen Daten bereits auf dem Ausgangsfeld, wo sie ohne weitere Aktion dort belassen werden können. Ansonsten muss das Hilfsfeld nach der ersten **`while`**-Schleife (`sort to field1/2`) auf das Ausgangsfeld kopiert werden (`copy rest of field2/1`).

Algorithmus mit einfach verketteter Liste

Wie bereits mehrfach angedeutet, lässt sich *`runSort`* mit einfach verketteten Listen ebenfalls implementieren, wobei man sich einerseits das Hilfsfeld erspart (das sortieren kann also *in situ* durchgeführt werden), andererseits aber auch die Geschwindigkeit deutlich besser wird.

Das Programm hat folgendes Aussehen.

```
static Data restList;
Data runSort(Data first, int exp) {
      // return first run; seperate it from rest; set restList
  if(exp==0) return seperateRun(first);
      // sort 2^(exp-1) runs from first; rest reference residual sequence
  first = runSort( first, exp-1);
  if(restList==null) return first; // return sorted list, if done
      // sort 2^(exp-1) runs from rest; rest references residual sequence
  return merge( first, runSort( restList, exp-1));// return merged list
}
```

Das Programm berechnet für **`exp=0`** einen Lauf ab **`first`**. Die Referenz des letzten Elements dieses

ersten Laufs wird in *seperateRun* auf `null` gesetzt! Die Parameterübergabe findet hier einmal über den Ergebnisparameter statt, der auf den Anfang des Laufs bzw. der sortierten Läufe zeigt, zum anderen über die globale Variable *restList*, die auf den Rest der Liste zeigt.

Ist `exp>0`, so werden zunächst 2^{exp-1} Läufe ab `first` berechnet und auf `first` gespeichert; auf *restList* steht der Anfang der restlichen Liste, oder `null`, falls die restliche Liste leer ist. Ab *restList* werden danach die nächsten 2^{exp-1} Läufe berechnet; *restList* zeigt danach auf den Anfang der restlichen Liste. Schließlich werden `first` und die zuletzt sortierten 2^{exp-1} Läufe ab *restList* gemischt und das Anfangselement dieser sortierten Liste zurückgegeben.

Die globale Variable *restList*, die hier verwendet werden muss, da Java-Methoden nicht mehrere Werte zurückgeben können, wird nur einmal in der Prozedur *seperateRun* gesetzt; ihr Wert ist jeweils eine Referenz auf den Anfang der restlichen Liste. Es ist tatsächlich nicht nötig, diese Variable in den hinteren Teilen der Methode *runSort* zu setzen, da sie automatisch immer hinter das Ende des letzten verarbeiteten Laufs zeigt.

Man sieht auch hier, dass der Algorithmus recht einfach implementierbar und leicht verständlich ist. Überraschend ist sicherlich auch die Kürze, mit welcher der Algorithmus implementiert werden kann. Eine noch kürzere Variante wäre

```
static Data runSort(Data first, int exp) {
      if(first==null) return null;
      if(exp==0)      return seperateRun(first);
      return merge( runSort( first, exp-1), runSort( restList, exp-1));
}
```

die allerdings dem Verständnis nicht förderlich ist, aber wegen der Abfrage auf die leere Liste sogar etwas sicherer ist als die erste Variante ist. Man beachte, dass die Auswertung der Parameter der Methode *merge* Seiteneffekte erzeugt – das Setzen der Variablen *restList* – so dass die Reihenfolge, in welcher die Parameter ausgewertet werden, wichtig ist.

Für die Laufzeitkomplexität gilt hier das gleiche wie beim Sortieren auf Feldern, da die logische Struktur die gleiche ist wie dort.

Wir verwenden hier nur einfache verkettete Listen; sind die Listen doppelt verkettet, was in vielen Standardimplementierung der Fall ist, so kann die Liste wie hier gezeigt nur mit ihren Vorwärtsverkettungen sortiert werden. Zum Schluss kann unabhängig von der Sortierung durch eine einfache Schleife die Rückwärtsverkettung nachgetragen werden:

```
run=list; run.previous=null;
while(run.next!=null) { run.next.previous=run; run=run.next;}
```

Dieses vergrößert den Aufwand um N Vergleiche auf `null` und $2 \cdot N$ Zuweisungen.

Ggf. lässt sich dieser Algorithmus auch auf parallelen Architekturen implementieren, in dem z.B. zwei Hälften der Folge auf verschiedenen Rechnerkernen gleichzeitig gemischt und die fertigen Läufe zusammengemischt werden. Die Laufzeit könnte sich damit fast halbieren, bzw. bei noch größerer Aufteilung weiter verringern. Wenn das Mischen stabil erfolgt – d.h. die erste Hälfte priorisiert wird – dann bleibt der Algorithmus stabil; allerdings könnte die Ordnungsverträglichkeit etwas leiden, weil z.B. vollständig geordnete Folgen zunächst zerlegt und dann wieder zusammengesetzt werden müssen.

Implementierungsaspekte

Für das Mischen wurde die folgende Methode verwendet.

```
static Data list = new Data();
static Data merge(Data f, Data s) {
      if(f==null)return s; if(s==null)return f;
      Data run = list;
      while(true){
        if(f.key<=s.key) { run.next = f; f = f.next; if(f==null)          {
                                                 run.next.next = s;
                                                 return list.next;  }
```

```
        } else             { run.next = s; s = s.next; if(s==null)          {
                                                       run.next.next = f;
                                                       return list.next;  }
        }
        run = run.next;
}       }
```

Diese Methode verwendet ein statisches Objekt **`Data`** ***`list`***, um eine nicht leere Referenz auf das erste Objekt (zunächst **`run`**) zu erhalten. Die sortierte Liste beginnt dann bei **`list.next`**, was daher auch der Rückgabewert ist. Ansonsten wird hier die übliche Sortierung zweier Listen durch Mischen verwendet. Ist das erste Element aus der ersten Liste (die vor der zweiten angeordnet ist) kleiner oder gleich dem Element aus der zweiten Liste, so wird dieses in die sortierte Liste eingehängt und von der ersten Liste entfernt; ist das Element aus der zweiten Liste kleiner, so wird dieses eingehängt und von der zweiten Liste abgetrennt; dadurch wird die Stabilität der Liste bewahrt. Offenbar ist es notwendige Voraussetzung, dass die beiden Listen mit einem **`next==null`** beendet sind; dann endet durch das Anhängen der verbleibenden Daten der noch nicht leeren Liste die sortierte Liste ebenfalls mit **`null`**.

Die andere Hilfsmethode findet das Ende eines einzelnen Laufs.

```
Data seperateRun(Data first){ //seperate first run, return first element of run
      Data aux=first;
      while(aux.next!=null && aux.key<=aux.next.key) aux = aux.next;
      restList = aux.next;    // set restList to first element of rest
      aux.next = null;        // set last element of run to null
      return first;           // return first element of run
}
```

Die Implementierung ist offenbar kanonisch, der Aufwand ist im wesentlichen gleich der Länge des Laufs. Man beachte, dass hier die Referenz auf den Rest der Liste gesetzt wird – und dass das letzte Elemente des Runs auf **`null`** gesetzt wird.

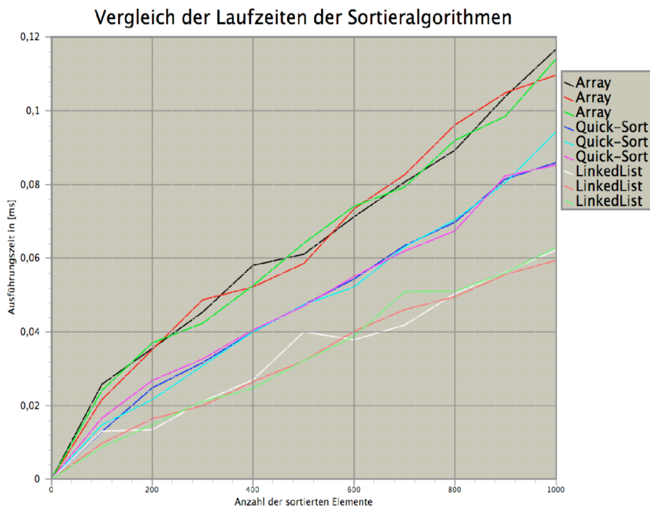
Laufzeitvergleiche

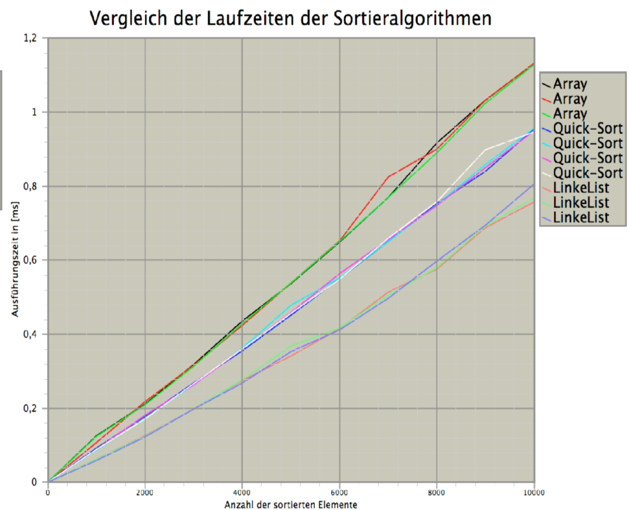
Neben der Anzahl von Vergleichen und Kopieroperationen muss auch der Kontroll-Overhead der rekursiven Programme betrachtet werden. Dieses lässt sich nur mittels Laufzeitvergleichen einigermaßen realistisch durchführen, zumal letztendlich die Laufzeit ausschlaggebend für die quantitative Qualität eines Algorithmus ist.

Wir haben die beiden hier vorgestellten Algorithmen mit Quicksort in Java-Implementierungen verglichen. Die folgenden Diagramme zeigen die jeweiligen Laufzeiten.

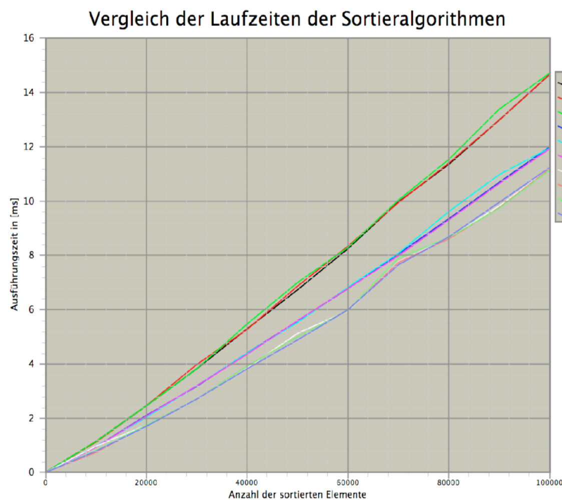
Hinweise zur Messung*: Sämtliche Läufe wurden auf einem Mac 2.6 GHz Intel Core i7, 16 GB 1600 MHz DDR3-Rechner ausgeführt unter Java 6. Da die HotSpot-Optimierung von Java erst nach häufigeren Schleifendurchläufen aktiv wird, haben wir jeden Algorithmus mit mindestens 1 Million Elementen mehrere Male ausgeführt, ehe Messungen vorgenommen wurden. Jede Kurve wurde in jedem Punkt zehn Mal mit zufälligen Daten gemessen und die Laufzeiten gemittelt. Es wurden jeweils drei oder mehr derartiger Kurven ermittelt, um die (geringe) Varianz der Ergebnisse zu kontrollieren.*

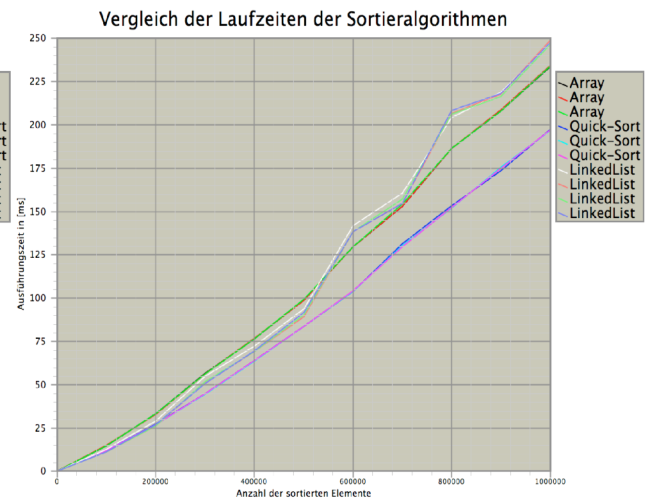
Offenbar liegen die Ergebnisse alle sehr eng beieinander, wobei die verketteten Listen leicht besser zu sein scheinen als die anderen Verfahren. Allerdings ist der Unterschied so gering, dass die Laufzeit hier kaum ausschlaggebend für die Wahl eines Algorithmus sein dürfte.



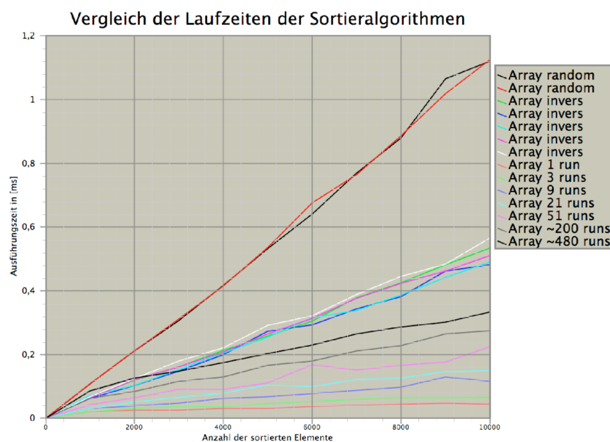


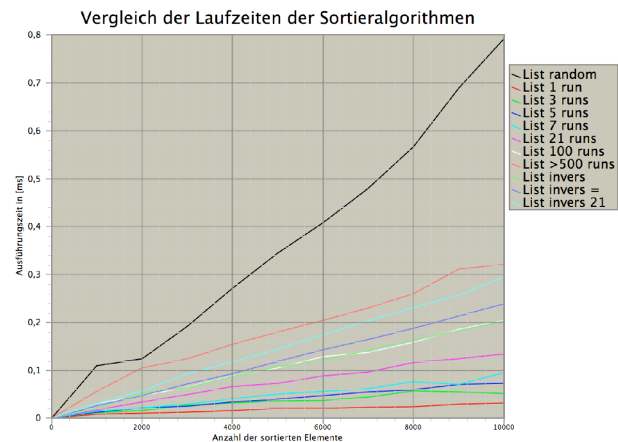
Bei sehr großen Datenmengen – mehr als 250.000 – scheint die verkettete Liste langsamer als Quicksort zu werden, ab 500.000 sogar langsamer als die Array-Variante; in wieweit dieses an der Implementierung oder der Rechnerarchitektur liegt, konnte nicht festgestellt werden. Bei kleineren Datenmengen scheint jedoch die verkettete List die besten Ergebnisse zu liefern, und sogar mit schnellen direkten Sortierverfahren konkurrieren zu können.





Auch die Ordnungsverträglichkeit lässt sich messen, existiert also real und nicht nur theoretisch.





Das Diagramm zeigt für Folgen mit verschieden vielen zufällig erzeugten Läufen die Laufzeiten, die offenbar direkt mit der Anzahl der Läufe wachsen. Bei nur einem Lauf wächst die Laufzeit offenbar proportional der Anzahl der Elemente, wie die Implementierung auch nahelegt. Aber auch bei wenigen bis zu einer mittleren Anzahl von Läufen wächst die Laufzeit proportional dieser Anzahl, so dass sich der Algorithmus für vorsortierte Mengen sehr effizient einsetzen lässt.

Bemerkenswert ist die Erkenntnis, dass invers sortierte Folgen offenbar wesentlich schneller sortiert werden als zufällige Folgen, nämlich mehr als doppelt so schnell. Dieses Phänomen findet sich auch bei anderen höheren Verfahren, die zumindest tendenziell bessere Laufzeiten bei vorsortierten Folgen haben (Quicksort gehört auch dazu!).

Der Grund könnte darin liegen, dass beim stabilen Mischen inverser Folgen immer nur noch Listen vorkommen, bei denen die größeren Elemente vollständig in der einen, die kleineren vollständig in der anderen Liste liegen. In diesem Fall geht das Mischen sehr effizient vonstatten, sowohl in der Array- als auch in der Listenimplementierung. Bei der Listenimplementierung wird in diesem Fall eine Liste stets ungekürzt an die andere angehängt, so dass für die zweite Liste keinerlei Vergleiche oder Kopierungen durchgeführt werden müssen, was hier zu einer erheblichen Laufzeitverbesserung führt. Daher steht dieses Verfahren im Gegensatz zu vielen anderen Sortierverfahren, bei denen eine inverse Vorsortierung i.d.R. zu deutlich längeren Laufzeiten führt, z.B. InsertionSort. Unsere theoretische Abschätzung von oben konnte uns diese Ergebnisse nicht vorhersagen. Quicksort ist bei sortierten Folgen ebenfalls deutlich schneller als bei zufälligen, auch bei inversen Folgen (da die Anzahl der Vertauschungen dann deutlich geringer wird)! Allerdings reicht Quicksort bei wenigen Läufen R bei weitem nicht an den Geschwindigkeitsgewinn von `runSort` heran, da hier die Anzahl von Vergleichen und Kopieroperationen durch $N \cdot \log_2 R$ beschränkt ist.

Resümee

Als Ergebnis bleibt festzuhalten, dass der qualitativ höherwertige Algorithmus `runSort` durchaus quantitativ mit Quicksort mithalten kann. Da er aber stabil, ordnungsverträglich und sicher ist, und da er darüber hinaus auch etwas einfacher implementiert werden kann, sollte er ggf. die erste Wahl sein, wenn es auf derartige Eigenschaften ankommt. Insbesondere eine verkettete Liste kann bei Vorliegen entsprechender Datenstrukturen vorteilhaft mit `runSort` sortiert werden. In Java werden beispielsweise zum Sortieren von verketteten Objekten diese zunächst in ein Array 'gedumpt', sortiert und dann wieder in einer Liste umgerechnet; dieses lässt sich mir `runSort` sicherlich schneller und eleganter lösen. Der Standard für Java 6 über die Klasse `Collection` schreibt hierzu:

```
public static <T> void sort(Collection.List<T> list, Comparator<? super T> c)
```

...

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. The specified list must be modifiable, but need not be resizable. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the $n^2 \log(n)$ performance that would result from attempting to sort a linked list in place.

`runSort` hat bei einer zufällig ungeordneten Folge von Elementen die schlechteste Laufzeit. Sowohl teilweise oder ganz vorsortierte Folgen, aber auch invers teilweise oder ganz sortierte Folgen benötigen deutlich geringere Laufzeiten, so dass dieser Algorithmus in vielen realistischen Fällen deutlich besser als viele andere – Quicksort eingeschlossen – sein dürfte.

Anhänge

Referenzen

[**Sorting algorithm (Wikipedia)**]

Eine Übersicht über verschiedene Sortieralgorithmen.

[**TimSort (Wikipedia)**]

Timsort sortiert Läufe abhängig von ihrer Länge; es werden jeweils möglichst gleich lange Läufe zusammengefasst. Dieses wird auf jeden Fall die Laufzeit verringern. Allerdings werden hier die Lauflängen explizit überprüft, was den Algorithmus vermutlich schwerer korrekt zu implementieren gestattet. Die Information über die Längen der Läufe werden explizit in einem Stack gesammelt und untere Läufe solange gemischt, bis sie mit oberen Läufen verknüpft werden können.

[**MergeSort (Wikipedia)**]

Mischen zweier Folgen bis zum Ende eines Laufes in einer Folge. Dann Mischen der nächsten beiden Folge. Wiederholen, bis Anzahl der Läufe auf eins reduziert ist. Laufzeit ist $O(N \cdot \log_2 N)$, wobei Vergleiche dreimal so oft wie bei anderen Verfahren vorgenommen werden müssen.

[**BinarySort (Wikipedia bezeichnet dieses als MergeSort)**]

Rekursive Aufteilung der Feldelement bis zur Länge 1 und mischen der Teilfelder. Laufzeit ist $O(N \cdot \log_2 N)$. Keine Berücksichtigung von Läufen, d.h. nicht ordnungsverträglich aber stabil.

[**InsertionSort (Wikipedia)**]

Einfügen eines Elements in eine geordnete Teilfolge, bis alle Elemente einsortiert sind. Laufzeit ist $O(N^2)$; ordnungsverträglich und stabil.

[**QuickSort (Wikipedia)**]

Halbieren der Menge von Elementen in kleinere und größere Elemente. Fortsetzen, bis alle Halbierungen die Länge 1 haben. Laufzeit ist im Mittel $O(N \cdot \log_2 N)$, im schlechtesten Fall $O(N^2)$. Quicksort ist im Mittel schnell, aber nicht sicher durch $O(N \cdot \log_2 N)$ beschränkt. Weder ordnungsverträglich noch stabil.

[**HeapSort (Wikipedia)**]

Erstellen spezieller Halbordnung (Heap) auf Feld. Setzen des größten Elements ans Ende, Restauration des Heap mit restlichen Elementen usw., bis alle Elemente sortiert. Laufzeit ist $O(N \cdot \log_2 N)$. Keine Berücksichtigung von Läufen, d.h. nicht ordnungsverträglich und auch nicht stabil.

Listings

```
class RunSort { // sort array
      static public int runSort(Data field1[], Data field2[],
                                          int from, int exp, boolean up){
            if(exp==0)
                  if(up) return findRun( field1, from);
                  else   return copyRun( field1, field2, from);
            int next = runSort( field1, field2, from, exp-1, up);
            if(next>=field1.length) return next;
            int end = runSort( field1, field2, next, exp-1, !up);
            merge( field1, field2, from, next, end, up);
            return end;
      }
      static public int findRun( Data field[], int from){
            from++;
            while(from<field.length && field[from-1].key<=field[from].key)
                  from++;
            return from;
      }
      static public int copyRun( Data field1[], Data field2[], int from){
            field2[from] =field1[from];
            from++;
            while(from<field1.length && field1[from-1].key<=field1[from].key){
                  field2[from] =field1[from];
                  from++;
            }
            return from;
      }
      static public int merge( Data field1[], Data field2[],
                              int from, int next, int end, boolean up){
            int from1 = next-1, from2 = end-1, to = end-1;
            if(up){ // sort to field1
                  while(from1>=from && from2>=next)
                        if(field1[from1].key>field2[from2].key)
                                    field1[to--] = field1[from1--];
                        else        field1[to--] = field2[from2--];
                  while(from2>=next) field1[to--] = field2[from2--];
            }else {
                  while(from1>=from && from2>=next)
                        if(field2[from1].key>field1[from2].key)
                                    field2[to--] = field2[from1--];
                        else        field2[to--] = field1[from2--];
                  while(from2>=next) field2[to--] = field1[from2--];
            }
            return end;
      }

}

class Data{ // sort linked list
      public int key;
      public Data next;
      public Data(int key2) {
            this.key = key2;
      }
      /**
       * restList references rest of the list
       */
      static Data restList;
      /**
    * runSort sets restList to rest of sequence, returns first merged 2^exp runs
       */
      static Data runSort(Data first, int exp) {
            if(exp==0) return seperateRun(first);
            first = runSort( first, exp-1);
            if(restList==null) return first;
            return merge( first, runSort( restList, exp-1));
      }
      static Data list = new Data(0);
      /**
```

```
     * merge merges f and s into one list and returns the sorted list
     * @param f first  sorted list to merge; ends with null-referenced element
     * @param s second sorted list to merge; ends with null-referenced element
     * @return merged and sorted list; ends with null-referenced element
     */
    static Data merge(Data f, Data s) {
        if(f==null)return s; if(s==null)return f;
        Data  run = list;
        while(true){
            if(f.key<=s.key) {
                run.next = f;
                f = f.next;
                if(f==null) {
                    run.next.next = s;
                    return list.next;
                }
            } else {
                run.next = s;
                s = s.next;
                if(s==null) {
                    run.next.next = f;
                    return list.next;
                }
            }
            run = run.next;
        }
    }
    /**
     * seperateRun
     * find first run; seperate it from rest; set restList to rest of sequence
     * @param first: reference to first element of run
     * @return reference: to first element of run
     */
    static Data seperateRun(Data first) {
        Data run=first;
        while(run.next!=null && run.key<=run.next.key) run = run.next;
        restList = run.next;
        run.next = null;
        return first;
    }
}
```