RunSort

Ein stabiles, sicheres und ordnungsverträgliches

höheres Sortierverfahren

Prof. Dr. W. Kowalk
Universität Oldenburg 14.03.13, kowalk@uni-oldenburg.de

Abstrakt

Wir stellen einen Sortieralgorithmus 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 häufig vorkommenden Anwendungsfällen eine gute Wahl.

Abstract

We present a new sorting 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 easy to understand and to be implemented with performance comparable to Quicksort; thus it is a good choice in many common applications.

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. 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 (unter einem

Lauf (engl. *run*) verstehen wir hier eine vorsortierte Teilfolge in der gegebenen Ordnung der Daten). Die Anzahl der Läufe 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.

Aufbau des Berichts

Der Bericht stellt zunächst das grundlegende Konzept des Algorithmus vor und betrachtet dann die Einzelheiten der Implementierung im Falle des Sortierens auf Feldern; danach wird die Laufzeitkomplexität analytisch und durch Messung von Laufzeiten in Form von Diagrammen angegeben. Als Vergleichsalgorithmus verwenden wir Quicksort. Als nächstes werden die gleichen Untersuchungen für Listen durchgeführt. Ein Algorithmus, der ähnliche Konzepte verfolgt, ist TimSort, so dass auch mit diesem die Laufzeiten verglichen werden. Für kleine Datenmengen werden häufig direkte Sortieralgorithmen verwendet, so dass wir auch noch einen Vergleich mit Insertionsort geben. In einem Resümee fassen wir unsere Ergebnisse zusammen.

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 zusammengemischt werden. Das lässt sich angenähert 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 sich die Anzahl der Vergleiche 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 bei vorsortierten, aber auch bei invers sortierten Listen zum Tragen kommt.

Eine 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 mindestens 2^k bzw. im Mittel 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 außer beim Autor auch in der Standardliteratur, z.B. Sedgewick und Knuth, und unter (*Sorting algorithms*) in Wikipedia.

Mischen auf Feldern

Wir verwenden ein Feld (**seq**) und ein Hilfsfeld (**seq2**). Die Startmethode kann folgendermaßen implementiert werden.

```
Data [] startRunSort(Data seq[]) {
      if (seq == null) return null ;
      if (seq.length <= 1) return seq;
      Data seq2[] = new Data[seq.length];
      int exp = 32-Integer.numberOfLeadingZeros(seq.length-1);//=log₂len
      runSort( seq, seq2, 0, exp, true);
      return seq;
}
```

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

```
// runSort merges 2^exp runs from seq1 starting at from to up?seq1:seq2;
// runSort returns the index of first element of the run after those 2^exp runs.
int runSort(Data seq1[], Data seq2[], int from, int exp, boolean up){
      if(exp==0)
            if(up) return findRun( seq1,       from);    // find next run    (1)
            else   return copyRun( seq1, seq2, from);    // copy run to seq2 (2)
      int next = runSort( seq1, seq2, from, exp-1, up);  // sort to seq1     (3)
      if(next>=seq1.length) return next;           // return if all sorted   (4)
      int end  = runSort( seq1, seq2, next, exp-1, !up); // sort to seq2     (5)
      merge( seq1, seq2, from, next, end, up);     // merge to (up?seq1:seq2)(6)
      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 **(1)**; **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 **(2)**. Die Hilfsprozedur **merge** mischt die Läufe von **seq1** in dem Bereich **from** bis **next-1** und **seq2** in dem Bereich **next** bis **end-1** nach **up?seq1:seq2** 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 **(1,2)**; der Index (Rückgabewert) verweist auf das erste Feld hinter diesem ersten Lauf. Ist **exp>0**, so werden zwei Läufe (bestehend aus 2^{exp-1} sortierten Läufen), 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. Der Anfangswert von **exp** muss offenbar größer oder gleich $\log_2 N$ sein, da dann $2^{exp} >= N$.

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 Schlüsselvergleiche 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.

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

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.

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 maximal N_1+N_2-1 Schlüsselvergleiche und N_1+N_2 Kopieroperationen ist. Der Gesamtaufwand für Schlüsselvergleiche bzw. Kopieroperationen ist also jeweils 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 $N+N\cdot\log_2 R$ Schlüsselvergleiche und Kopieroperationen beschränkt. Also gilt auch für die Laufzeitkomplexität $O(N+N\cdot\log_2 R)$.

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. Zusätzlich zeigen Messungen (s.u.), dass sich gegenüber dem vorigen Fall die Laufzeit in diesem Fall effektiv etwa halbiert. 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 $N\cdot\log N$ ist, wobei der Algorithmus stabil, ordnungsverträglich und sicher ist. 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.

Neben Schlüsselvergleichen und Kopieroperationen gibt es auch einen Kontrolloverhead z.B. durch Vergleiche auf Feldgrenzen und rekursive Funktionsaufrufe. Die Anzahl der rekursiven Funktionsaufrufe von **runSort** ist durch $2\cdot R+\log_2 N$ beschränkt. Gibt es nur einen Lauf, wird als Parameter **exp** dennoch $\log_2 R$ verwendet, so werden tatsächlich nur $\log_2 R+1$ Aufrufe getätigt, da alle Prozeduren mit einem Parameter **exp**>0 einen rekursiven Aufruf durchführen, während für **exp**$=0$ nur das Ende des Laufs einmal gesucht wird. Die folgende Abfrage (4) beendet dann die aufrufende Proze-

dur. Gibt es $R=2^k$ Läufe, so werden in der Aufruftiefe k jeweils zwei Läufe zusammengemischt (also $R/2$ neue Läufe erzeugt), in der Aufruftiefe $k-1$ zwei doppelt lange Läufe zusammengemischt usw., bis sämtliche Läufe zusammengefasst sind. Es werden also maximal $R/2+R/4+...+2+1=R-1=2^k-1$ Läufe gemischt. Zusätzlich mit den letzten Aufrufen, welche einen der R Läufe zurückgeben, ergeben sich daher maximal $2\cdot R-1$ Aufrufe der Methode **`runSort`**. Jeder Aufruf der Methode **`runSort`** benötigt dann einen Vergleich von **`exp`** auf 0, einen Vergleich auf das Ende der Liste und einen Aufruf des Misch-Algorithmus. Daher ist auch der Kontrolloverhead beschränkt durch $2\cdot R$ Methodenaufrufe von **`runSort`** und Vergleiche mit **`exp`** sowie R Aufrufe von **`findSort`** bzw. **`copySort`** sowie **`merge`** und Vergleiche auf das Listenende beschränkt. Der Aufwand für diesen Kontrolloverhead wächst also im wesentlichen linear mit $2\cdot R$, wobei hier die nächste größere Zweierpotenz von R zu nehmen ist (also nicht größer als insgesamt $4\cdot R$).

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 $O(N+N\cdot\log_2 R)$ 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 in vielen Fällen 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 $\lceil \log_2 N \rceil$ 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 in Zeile (4) 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 $2^{\texttt{exp}}$ 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 seq1[],Data seq2[],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 seq1
        if(seq1[from1].key>seq2[from2].key) seq1[to--] = seq1[from1--];
        else                               seq1[to--] = seq2[from2--];
      while(from2>=next) seq1[to--] = seq2[from2--]; // copy rest of seq2
    } else {
      while(from1>=from && from2>=next) // sort to seq2
        if(seq2[from1].key>seq1[from2].key) seq2[to--] = seq2[from1--];
        else                               seq2[to--] = seq1[from2--];
      while(from2>=next) seq2[to--] = seq1[from2--]; // copy rest of seq1
    }
}
```

Das Mischen läuft also von hinten nach vorne. Der Grund dafür liegt darin, dass auf diese Weise der erste Lauf auf dem Ausgangsfeld verbleiben kann, so dass bei nur einem Lauf überhaupt keine Kopieroperationen nötig sind. Die Läufe werden solange vom Hilfsfeld 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 seq1/2`) auf das Ausgangsfeld kopiert werden (`copy rest of seq2/1`).

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 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 dem 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. 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 der Funktionalität ***merge*** 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 **`restList`** auf den Rest der Liste gesetzt wird – und dass die Referenz des letzten Element des Runs **`first`** auf **`null`** gesetzt wird.

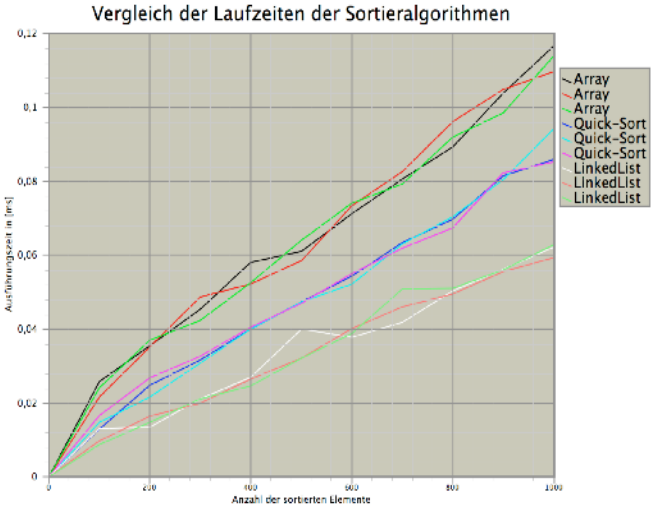
Laufzeitvergleiche

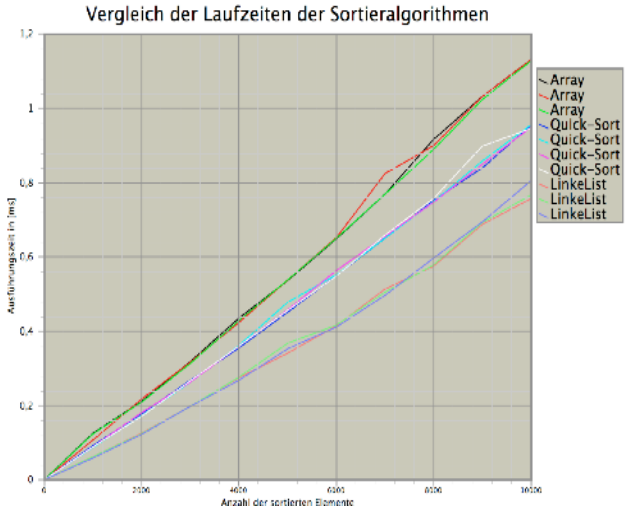
Neben der Anzahl von Vergleichen und Kopieroperationen muss auch der Kontroll-Overhead der rekursiven Programme betrachtet werden. Dieses lässt sich nur mittels Laufzeitvergleichen einigermaßen realistisch durchführen, zumal letztendlich die Laufzeit ausschlaggebend für die quantitative Qualität eines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

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

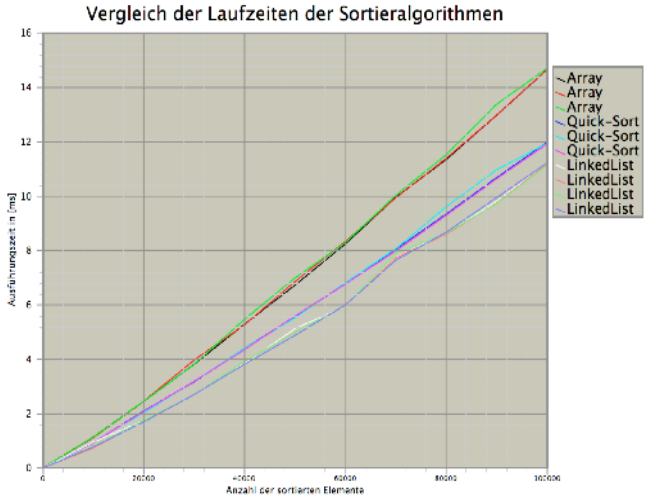
***Hinweise zur Messung**: Sämtliche Läufe wurden auf einem Mac 2.6 GHz Intel Core i7, 16 GB 1600 MHz DDR3-Rechner ausgeführt unter Java 6. Da die HotSpot-Optimierung von Java erst nach häufigeren Schleifendurchläufen aktiv wird, haben wir jeden Algorithmus mit mindestens 1 Million Elementen mehrere 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.*

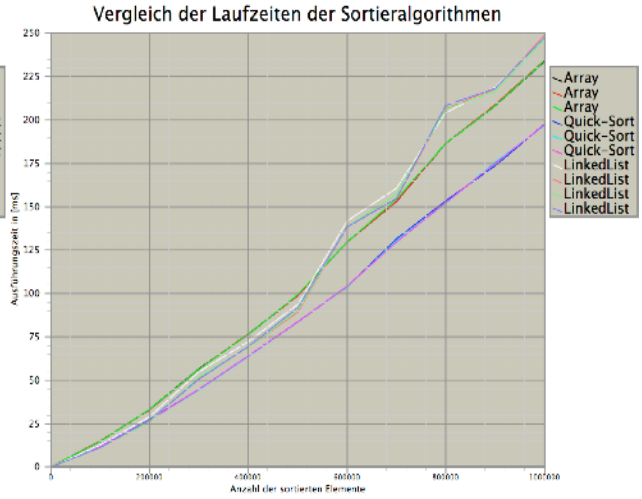
Offenbar liegen die Ergebnisse alle sehr eng beieinander, wobei die verketteten Listen leicht besser zu sein scheinen als die anderen Verfahren. Allerdings ist der Unterschied so gering, dass die Laufzeit hier kaum ausschlaggebend für die Wahl eines Algorithmus sein dürfte.



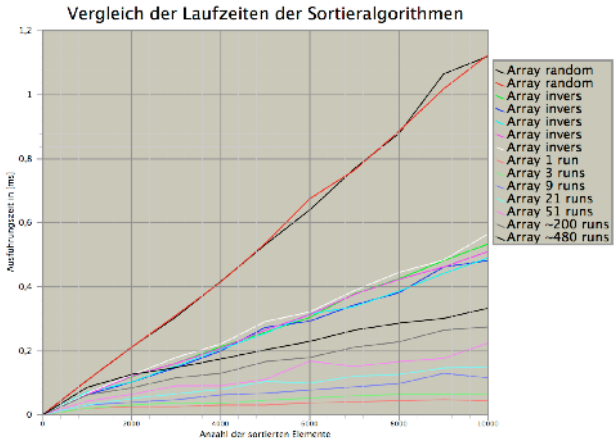


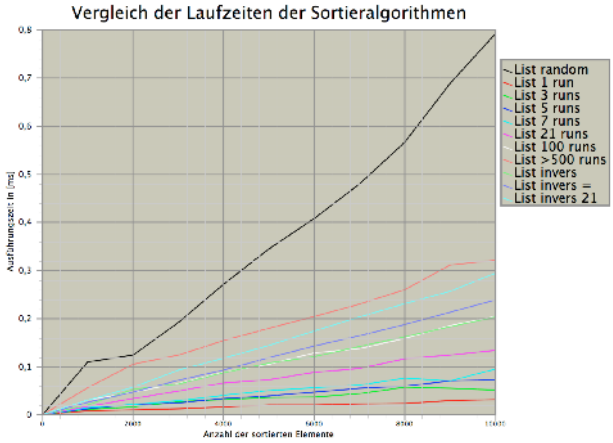
Bei sehr großen Datenmengen – mehr als 250.000 – scheint die verkettete Liste langsamer als Quicksort zu werden, ab 500.000 sogar langsamer als die Array-Variante; in wieweit dieses an der Implementierung oder der Rechnerarchitektur liegt, konnte nicht festgestellt werden. Bei kleineren Datenmengen scheint 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!).

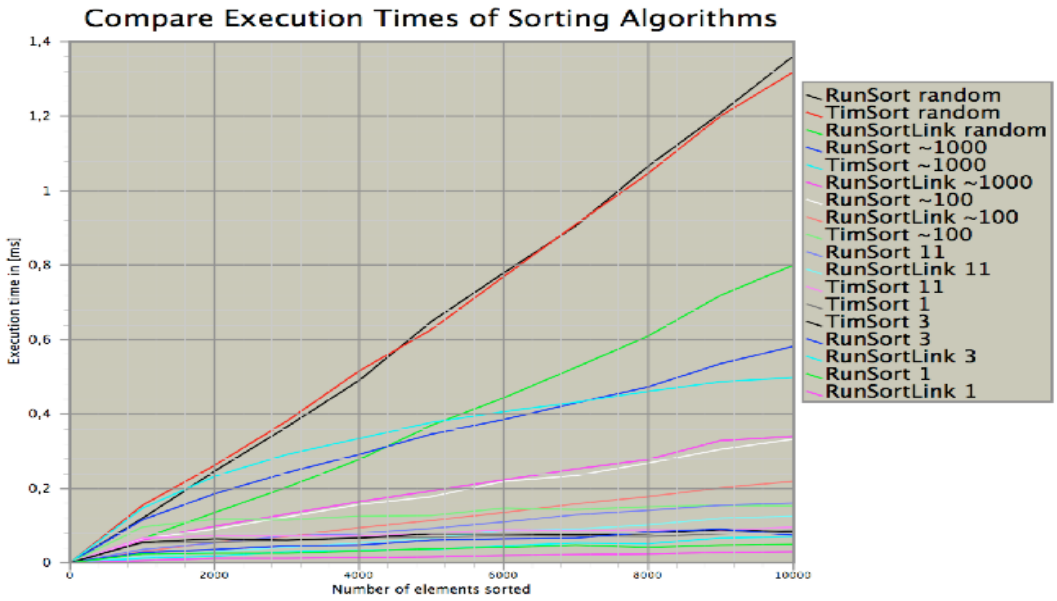
Der Grund könnte darin liegen, dass beim stabilen Mischen inverser Folgen immer nur noch Listen vorkommen, bei denen die größeren Elemente vollständig in der einen, die kleineren vollständig in der anderen Liste liegen. In diesem Fall geht das Mischen sehr effizient vonstatten, sowohl in der Array- als 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 bei **`runSort`** die Anzahl von Vergleichen und Kopieroperationen durch $N \cdot \log_2 R$ beschränkt ist.

Dieses gibt Anlass über eine weitere Verbesserungsmöglichkeit nachzudenken. Wenn das erste

und das letzte Element einer Liste bekannt sind, so kann durch Vergleich vor dem Mischen, ob das letzte Element der einen Liste kleiner oder gleich dem ersten Element der anderen Liste ist, die Anzahl der Vergleiche noch mehr reduziert werden; allerdings erfordert dieses, dass in jedem Mischvorgang zusätzliche Vergleiche durchgeführt werden, was u.U. bei zufällig sortierten Folgen einen Geschwindigkeitsverlust bedeutet. Messungen haben diesen Trend bestätigt, wenngleich in einem geringem Umfang, was aber dennoch einen Zusatznutzen bedeutet. Bei umgekehrt sortierten Listen hatte sich bei teilweiser Vorsortierung ein gewisser Geschwindigkeitsgewinn gezeigt, wenngleich auch nur in einem geringen Umfang.

Eine mögliche Implementierung dieser Veränderung ist im Anhang (Listing **`DataL`**) angegeben.

TimSort



Wie bereits erwähnt, haben **`timSort`** und **`runSort`** viele Ähnlichkeiten. Zum Vergleich wurde hier eine Implementierung von **`timSort`** in Java verwendet, die scheinbar auch im Java-Compiler verwendet wird. Der Vergleich der Laufzeiten zwischen **`timSort`** und **`runSort`** fällt zugunsten von **`timSort`** aus, wobei bei zufällig gegebenen Folgen die Laufzeiten ungefähr gleich sind, bzw. für die Listenversion deutlich schneller ist.; bei Folgen mit weniger Läufen als ca. $N/2$ wird **`timSort`** bei größeren Datenmengen etwas schneller. **`timSort`** scheint von der expliziten Behandlung von gegebenen Teilfolgen – auch inversen – zu profitieren, was den Algorithmus allerdings auch deutlich komplexer macht, während **`runSort`** im Prinzip mit einem sehr kurzem Programm auskommt. Darüber hinaus lässt sich **`runSort`** auch einfach auf Listen übertragen, was in den meisten Fällen zu einer Verbesserung der Laufzeiten führt.

Das Diagramm vergleicht **`timSort`** mit **`runSort`** und der Listenversion von **`runSort`**. Die Zahlen hinter den Algorithmen geben die (teilweise angenäherte) Anzahl von Läufen bei 10.000 Elementen an. Die Tests wurden je Messpunkt zehnmal wiederholt und gemittelt. Für den Vergleich wurde hier die Comparator-Methode von Java verwendet, damit dieses realistisch mit **`timSort`** verglichen werden kann, wo dieses ebenfalls so durchgeführt wird.

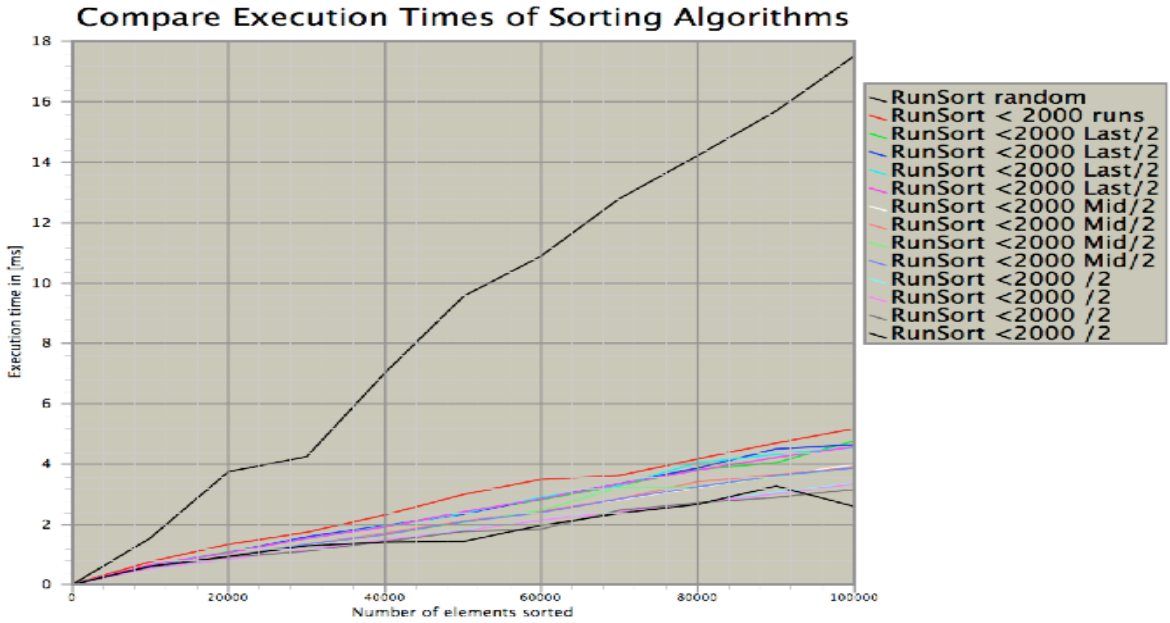
Man sieht, dass alle Algorithmen den gleichen Trend haben. **`timSort`** hat einen etwas höheren Overhead, so dass für eine kleinere Anzahl von Elementen die Laufzeiten etwas größer sind. Z.B. sind für 1000 Läufe im unteren Bereich des Diagramms die Laufzeiten für **`timSort`** etwas schlechter als für **`runSort`**, wenngleich nicht sehr viel, so dass dieses für die Entscheidung für oder gegen einen Algorithmus kaum ins Gewicht fallen dürfte. In den meisten Fällen ist jedoch die Listenversion von **`runSort`** dem Algorithmus **`timSort`** deutlich überlegen.

`timSort` verwendet einen internen Stack für die Ermittlung der Längen der zu mischenden Läu-

fe, wofür ein festes Feld verwendet wird. Es bleibt zu untersuchen, ob hier womöglich ein Stacküberlauf zu einem Sicherheitsproblem führen könnte, was bei **`runSort`** niemals der Fall ist.

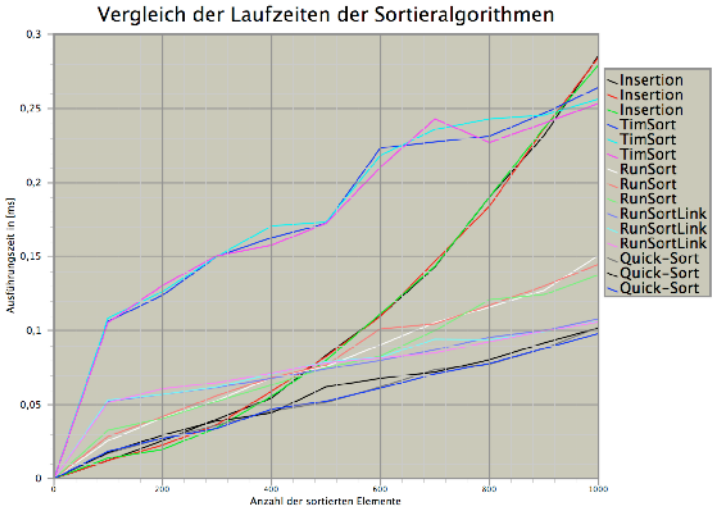
Als möglicher Vorteil von **`timSort`** könnte man annehmen, dass dort bei stark unterschiedlichen Längen der Anfangs gegebenen Läufe durch die Analyse der Längen der Zwischenläufe häufiger etwa gleich lange Läufe gemischt werden können. Das ist bei **`runSort`** sicherlich nicht der Fall. Deshalb haben wir auch dieses simuliert und als Ergebnis das folgende Diagramm erhalten.

Es wurden künstliche Läufe erzeugt, wobei die erste Hälfe (/2), die letzte Hälfte (Last/2) bzw. zu Anfang und am Ende ein Laufe der Länge $1/4 \cdot N$ (Last/2) verwendet wurde, d.h. die Längen der Läufe waren extrem unterschiedlich. Sämtliche dieser Sortierungen liefen schneller als im Falle zufälliger Daten bzw. einer beschränkten Anzahl zufälliger Läufe (weniger als 2000). Dabei scheint ein Trend zu bestehen, dass ein lange Lauf am Anfang die kürzeste Sortierzeit hat, am Ende die längste. Die Unterschiede sind allerdings wieder sehr gering. Daher ist **`runSort`** auch in diesen Situationen ein effizienter und robuster Algorithmus.



Insertionsort

Wir betrachten hier einen direkten Sortieralgorithmus, der bekanntlich eine quadratische Laufzeitkomplexität hat. Es stellt sich die Frage, in wieweit die hier vorgestellten Sortieralgorithmen auch für kleine Datenmengen mit direkten Sortieralgorithmen konkurrieren können. Wir haben uns für Sortieren durch Einfügen entschieden, da dieses das schnellste unter den direkten Verfahren ist, das darüber hinaus auch noch stabil und ordnungsverträglich ist.



Die Laufzeiten wurden unter den bereits genannten Bedingungen (10 Wiederholungen je Datenpunkt, 3 Folgen je Algorithmus) gemessen und in dem Diagramm eingetragen. Für kleinere Laufzeiten (<500) ist das direkte Sortierverfahren das schnellste. Darüber hinaus sind die Unterschiede analog den vorher genannten für kleine Laufzeiten, wobei RunSort in der Listenvariante am besten abschneidet, TimSort am schlechtesten. TimSort wird sogar erst ab knapp 1000 Elementen schneller als Insertionsort. Für weniger als 400 Elemente ist offenbar RunSort in der Listenvariante um einiges langsamer als RunSort auf Feldern. QuickSort wurde auch aufgenommen, wegen seiner schlechteren Qualität jedoch außer Konkurrenz.

Vereinfachungen

Man kann diesen Algorithmus etwas vereinfachen. Z.B. kann man statt eines Laufs nur ein Element wählen (dieses entspricht nicht dem Fall einer umgekehrt sortierten Folge!). Bei zufälligen Daten scheint das Programm nicht langsamer zu werden. Bei vorsortierten Läufen ist es etwas langsamer. Wem der Algorithmus noch zu kompliziert ist, kann diese Vereinfachung z.B. so implementieren:

```
static <T> int runSort(T seq1[], T seq2[], int from, int exp, boolean up,){
  if(exp==0) { if(!up) seq2[from] = seq1[from];  return ++from;}
  int next = runSort( seq1, seq2, from, exp-1, up, c);
  if(next>=seq1.length)  return next;
  int end = runSort( seq1, seq2, next, exp-1, !up, c);
  return merge( seq1, seq2, from, next, end, up, c);
}
```

Vermutlich gibt es nur wenige sinnvolle Anwendungen hierfür, außer wenn es z.B. auf Programmlänge ankommt. In Fällen, in denen ausschließlich zufällig sortierte Folgen vorliegen, stellt zumindest die Laufzeit keinen Nachteil dar.

Resümee

Bei den hier vorgestellten Algorithmen mit dem anschaulichen Namen **`runSort`** geht es in erster Linie um die höhere Qualität, d.h. Ordnungsverträglichkeit und Stabilität sowie Sicherheit. All diese Eigenschaften hat QuickSort nicht, wobei gerade die Sicherheit als sehr wichtig angesehen wird (z.B. wird in einigen kommerziellen Anwendungen bei QuickSort die Rekursionstiefe untersucht, und sollte diese zu groß werden wird ein anderer Algorithmus – z.B. HeapSort – verwendet); aber auch die anderen Eigenschaften werden in modernen Anwendungen immer häufig gefordert.

Ein weiterer Aspekt ist die Einfachheit dieses Algorithmus, da er im Prinzip in wenigen Zeilen notiert werden kann, zuzüglich einiger einfacher Hilfsfunktionen. Die Einfachheit impliziert neben weniger und verständlicherem Code auch eine höhere Korrektheit bei der Implementierung. Ein weitere Vorteil von **`runSort`** liegt darin, dass dieser Algorithmus auch einfach für Listen implementiert werden kann, mit größtenteils verbesserten Laufzeiten. Dieses stellt wohl eine wesentliche

Neuerung dar, zumal heutige Datenstrukturen häufig mit Listen implementiert werden.

runSort hat bei einer zufällig ungeordneten Folge von Elementen die schlechteste Laufzeit, die aber immer noch durch $N \cdot \log N$ beschränkt ist. 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 Algorithmen sein dürfte – Quicksort eingeschlossen; auch im schlechteren Fällen ist **runSort** meistens besser als Quicksort, und kann darüber hinaus besonders vorteilhaft auf Listen angewendet werden.

Vergleich mit anderen Sortieralgorithmen

Im Vergleich zu **runSort** hat **timSort** die gleichen Qualitätseigenschaften. Der Unterschied liegt somit zum einen in der Laufzeit, zum anderen in der einfacheren Implementierbarkeit von **runSort**. Zumal die Laufzeiten dieser Algorithmen lassen sich nicht ohne weiteres vergleichen, da für kleinere Datenmengen **runSort** meist schneller als **timSort** ist, während für größere Datenmenge beide in etwa gleich auf liegen; der Grund dürfte in der intensiven Vorbehandlung und weiteren Overhead zum Ausgleich der Längen der Läufe durch **timSort** liegen. Erst bei bereits vorsortierten Datenmengen mit einer geringeren Anzahl von Läufen als bei zufälligen Folgen scheint **timSort** deutlich besser zu sein als die Array-Variante von **runSort**, jedoch nicht als die Listenvariante.

Eine weiteres Handicap von **timSort** könnten ein interner Stack für die Speicherung von Lauf-Längen sowie einige freie Parameter sein, die zwar eine Optimierung ermöglichen, aber u.U. ungünstig gesetzt werden. Insbesondere der Stack könnte – wenn er zu klein gewählt wird – durch Überlauf einen Fehler erzeugen. Dieses kann hier jedoch nicht diskutiert werden.

Ein Vergleich mit einem direkten Sortierverfahren zeigt, dass **runSort** nicht schneller, aber vergleichbar schnell ist, und ggf. auch statt eines direkten Sortierverfahrens verwendet werden könnte, wenn es auf die etwas größeren Laufzeiten nicht ankommt. Allerdings ließen sich natürlich beim Aufsetzen des Algorithmus auch entsprechende Fallunterscheidungen treffen und bei kleineren Datenmenge **InsertionSort** statt **runSort** aufrufen, was wegen dieses ebenfalls sehr einfachen Algorithmus, der ebenfalls ordnungsverträglich und stabil ist, wohl vertretbar wäre.

Als Ergebnis bleibt festzuhalten, dass der qualitativ höherwertige Algorithmus **runSort** durchaus quantitativ mit dem bekannten (und bewährten) Quicksort mithalten kann. Da **runSort** aber stabil, ordnungsverträglich und sicher ist, und da **runSort** darüber hinaus auch einfacher implementiert werden kann als z.B. **timSort**, sollte **runSort** 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 eine 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 könnte hier wohl effektiv eine Verbesserung im Sinne des Einsparen von Speicherplatz und Laufzeit ermöglichen.

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.

// Source (31.12.2012) "http://cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/raw_files/new/src/share/classes/java/util/TimSort.java"

[**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.

Allgemeine Literatur zu Algorithmen:

Budd: „Classic Data Structures in Java“. Addison-Wesley 2001.

Kleinberg, Tardos: „Algorithm Design“. Pearson/Addison-Wesley 2006.

Knuth: „The Art of Computer Programming, Volume 3; Sorting and Searching“. Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), xiv+780pp.+foldout. ISBN 0-201-89685-0

Kowalk: „System, Modell, Programm“. Springer 1996.

Kowalk: „Animierte Algorithmen“. Vorlesungsskript. Uni Oldenburg.

Weiss: „Data Structures & Algorithm Analysis in Java“. Addison-Wesley 1999.

Sedgewick, Wayne: „Algorithms“. Addison-Wesley Longman, Amsterdam; Auflage: 4th revised

edition. (9. März 2011)

Listings

```
class RunSort { // sort array
	static public int runSort(Data seq1[], Data seq2[],
	                                        int from, int exp, boolean up){
		if(exp==0)
			if(up) return findRun( seq1, from);
			else   return copyRun( seq1, seq2, from);
		int next = runSort( seq1, seq2, from, exp-1, up);
		if(next>=seq1.length) return next;
		int end = runSort( seq1, seq2, next, exp-1, !up);
		merge( seq1, seq2, from, next, end, up);
		return end;
	}
	static public int findRun( Data seq[], int from){
		from++;
		while(from<seq.length && seq[from-1].key<=seq[from].key)
			from++;
		return from;
	}
	static public int copyRun( Data seq1[], Data seq2[], int from){
		seq2[from] =seq1[from];
		from++;
		while(from<seq1.length && seq1[from-1].key<=seq1[from].key){
			seq2[from] =seq1[from];
			from++;
		}
		return from;
	}
	static public int merge( Data seq1[], Data seq2[],
	                              int from, int next, int end, boolean up){
		int from1 = next-1, from2 = end-1, to = end-1;
		if(up){ // sort from last to first element towards seq1
			while(from1>=from && from2>=next)
				if(seq1[from1].key>seq2[from2].key)
					seq1[to--] = seq1[from1--];
				else
					seq1[to--] = seq2[from2--];
			while(from2>=next) seq1[to--] = seq2[from2--];
		}else { // sort from last to first element towards seq2
			while(from1>=from && from2>=next)
				if(seq2[from1].key>seq1[from2].key)
					seq2[to--] = seq2[from1--];
				else
					seq2[to--] = seq1[from2--];
			while(from2>=next) seq2[to--] = seq1[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;
       }
}

class DataL { // Sort Lists with known first and last element of each run
  public DataL next;
  int key = -1;
  public DataL(int key) { this.key = key; }
  /**
  * restList references rest of the total list
  * lastList references last element of current run
  */
  static DataL restList, lastList;
  /**
   * runSort sets restList to rest of sequence
   * sets lastList to last element of current run,
   * return   first which points to merged 2^exp runs
   */
  static DataL runSort(DataL first, int exp) {
      // set restlist, lastList, return first run
    if(exp==0) return seperateRun(first);
      // sort 2^exp-1 sorted runs sets restlist, lastList,
    first = runSort( first, exp-1);
    DataL last = lastList; // last hold reference to last element of run first
      // sort 2^exp-1 sorted runs sets restlist, lastList,
    if ( restList==null ) return first; // ready, return first
    DataL second = runSort( restList, exp-1);
     // merge first..last with second..lastList, set lastList, return merged
    return merge( first, last, second, lastList);
  }
  static DataL list = new DataL(-1);
  static DataL merge(DataL f,DataL flast, DataL s, DataL slast) {
    if(f==null)return s; if(s==null)return f;
    if(flast.key<=s.key) { flast.next = s;  lastList = slast; return f;
```

```
    } else if(slast.key<=f.key) { slast.next = f; lastList = flast; return s;
    }
    DataL  run = list;
    while(true){
      if(f.key<=s.key) {
        run.next = f;
        f = f.next;
        if(f==null) {
            run.next.next = s;
            lastList = slast;
            return list.next;
        }
      } else {
        run.next = s;
        s = s.next;
        if(s==null) {
            run.next.next = f;
            lastList = flast;
            return list.next;
        }
      }
      run = run.next;
  }//end while  } // end mege()
  //seperate first run, return first element of run, set lastElement
  static DataL seperateRun(DataL first) {
    DataL aux = first;
    while(aux.next!=null && aux.key<=aux.next.key) aux = aux.next;//find run end
    restList = aux.next;      // set restList to rest of List
    aux.next = null;          // set last element of run to null
    lastList = aux;           // set lastList to last element of run
    return first;             // return first element of run
  }
}

// Insertionsort
//Insert Elements at correkt Position; move all other Elements
 static public void insert(DataSet[] seq, Comparator <? super DataSet> c) {
   if (seq.length <= 1) return;       // array of length 1 is always sorted
   for (int start = 1; start < seq.length; start++) {//
     DataSet current = seq[start];// reference next element to be inserted
     int index = start - 1;
     for ( ; index >= 0; index--) {    // search insertion position
       if (c.compare(seq[index], current)<=0) break; //
       seq[index + 1] = seq[index];  // shift right
     }
     seq[index + 1] = current;        // insert
 } }

static public void quickSort(DataSet[] Feld, Comparator <? super DataSet> c) {
    quickSort(Feld, 0, Feld.length - 1, c);
}
static public void quickSort(DataSet[] seq, int left, int right,
                                          Comparator <? super DataSet> c) {
    if (left < right) {
      int indexL = left, indexR = right;
      DataSet comp = seq[ (indexL + indexR)/2 ];
      while (indexL <= indexR) {
        while (c.compare(comp, seq[indexL])>0) indexL++;
        while (c.compare(comp, seq[indexR])<0) indexR--;
        if (indexL <= indexR) {
          DataSet swap = seq[indexL];
          seq[indexL] = seq[indexR];
          seq[indexR] = swap;
          indexL++;
          indexR--;
      } }
      quick(seq, left, indexR, c);
      quick(seq, indexL, right, c);
  } }
```