SmoothSort

auf Edsger W. Dijkstra geht die Idee eines Sortieralgorithmus zurück, der im ungünstigsten Fall $O(n \cdot \log n)$ Vergleiche benötigt, im günstigsten Fall – wenn die Liste schon sortiert ist – aber nur $O(n)$ Vergleiche und keine Vertauschungen benötigt.
Es handelt sich um eine Sortierung mittels Vertauschungen, welche keinen zusätzlichen extern allozierten Speicher benötigt. Aus diesem Grund kann sie nicht stabil sein, da sonst im ungünstigsten Fall $O(n^2)$ Vergleiche benötigt würden. (Diese Behauptung ist eine Hypothese von mir, von der ich mich zwar recht ausführlich überzeugt habe, die ich aber nicht bewiesen habe.)
Man nennt eine Sortierung stabil, wenn Elemente, die bezüglich der Ordnungsrelation gleich gross sind, ihre ursprüngliche Reihenfolge beibehalten. Ein Beispiel für einen stabilen Sortierungsalgorithmus ist der sogenannte *MergeSort*, welcher eben nicht auf Vertauschungen aufbaut, sondern extern allozierten Platz benötigt. Dieser benötigt immer $O(n \cdot \log n)$ Vergleiche.
Der von Dijkstra als *SmoothSort* bezeichnete Sortieralgorithmus basiert in der Grundidee auf dem schon vor langer Zeit entdeckten *HeapSort*. Dieser baut auf Vertauschungen auf, ist nicht stabil und benötigt ebenfalls immer $O(n \cdot \log n)$ Vertauschungen.

Theoretische Grenzen der Komplexität

Das Sortieren kann als Selektion aus der Menge der Permutationen der Liste verstanden werden. Um eine Permutation aus den $n!$ möglichen auszusondern, werden $n \cdot \log n \simeq \log n!$ Bits an Information benötigt. Ein einzelner Vergleich produziert nur gerade 1 Bit an Information. Also muss man im ungünstigsten Fall damit rechnen, dass ein Sortieralgorithmus $O(n \cdot \log n)$ Vergleiche benötigt.
Wenn die Liste schon vollständig sortiert ist, benötigt man mindestens $n-1$ Vergleiche, um dies festzustellen. Insofern benötigt man im günstigsten Fall $O(n)$ Vergleiche für das Sortieren.

Sortierkomplexität in der Praxis

Eine Reihe von älteren Sortieralgorithmen benötigt mehr als $O(n \cdot \log n)$ Vergleiche. So etwa der *BubbleSort*, der immer $O(n^2)$ Vergleiche benötigt, ober der *QuickSort*, der zwar im „durchschnittlichen“ Fall $O(n \cdot \log n)$, im ungünstigsten Fall aber $O(n^2)$ Vergleiche benötigt. Es gibt einen interessanten „Beweis“, dass in diesem konkreten Fall der ungünstigste Fall der „durchschnittliche“ ist [Literaturangabe nachschauen].
Bis vor wenigen Jahren waren solche ineffizienten Sortiermethoden noch gang und gäbe: Im Officepaket wurde noch 1998 mittels *BubbleSort* sortiert, was sich in unglaublich langen Sortierzeiten äusserte. Da *QuickSort* Teil der C-Run-Time-Library ist, wird dieser ineffiziente Algorithmus von vielen C/C++-Programmierern eingesetzt, obwohl er wirklich nicht empfehlenswert ist.
Erst in der Run-Time-Library von JAVA findet man in den Klassen *Arrays* und *Collections* Sortieralgorithmen, welche im ungünstigsten Fall $O(n \cdot \log n)$ Vergleiche benötigen. Hier hat sich das Team von Joshua Bloch für *MergeSort* als Sortieralgorithmus entschieden, weil das Allozieren von externem Speicherplatz heutzutage nicht mehr so schwer ins Gewicht fällt und weil *HeapSort* nicht stabil ist.
Da der Algorithmus von Dijkstra auch nach seiner Publikation extrem unbekannt geblieben ist, ist nur Wenigen bekannt, dass man heute zwischen dem Vorteil der Stabilität (MergeSort) und dem Vorteil des Unterschreitens der $O(n \cdot \log n)$ Komplexität im Fall von nahezu schon sortierten Listen abwägen kann, wenn man einen Sortieralgorithmus auswählt.
Es ist zu prüfen, ob im Kontext von Datenbank-Indizes oder Suchmaschinen-Indexierung dieser letzte Fall nicht zur massiven Erhöhung der Performance eingesetzt werden kann, da ja jeweils die Differenz zwischen der schon sortierten Datenmenge von gestern und der heute neu zu sortierenden klein ist, und das Sortieren einen gewichtigen Anteil an der Gesamtperformance ausmacht.

Algorithmen

Im Folgenden werden hier die ernstzunehmenden Algorithmen *MergeSort*, *HeapSort* und *SmoothSort* explizit vorgeführt. Aus didaktischen Gründen, wird eine JAVA-Implementation dargestellt, weil JAVA-Kenntnisse heute weit verbreitet sind und es sich um eine klar strukturierte und definierte Sprache handelt. Dabei wird explizit vermieden, von den objekt-orientierten Eigenschaften von JAVA Gebrauch zu machen, damit offensichtlich wird, dass alle vorgestellten Algorithmen ohne grosse Änderungen leicht in C, C++, Pascal, C#, Visual Basic, JavaScript, PL/SQL etc. übertragen werden können.

Klasse Order

Die Klasse *Order* enthält alle drei Algorithmen für das Sortieren von Objekten, die die *List*-Schnittstelle implementieren. Insofern tritt die Klasse *Order* für Zwecke der Sortierung an die Stelle der Klasse *Collections* aus der JAVA-Run-Time-Library. Wie in der Implementation der *sort*-Methode in *Collections* kann man mit der Methoden *sortMerge*, *sortHeap* und *sortSmooth* in *Order* die Elemente eines *List*-Objekts mit Hilfe einer *Comparator*-Klasse oder basierend auf ihrer Implementation der *Comparable*-Schnittstelle sortieren.
Die Klasse *Order* enthält zu diesem Zweck eine private *Comparator*-Klasse, die auf der *Comparable*-Schnittstelle beruht, damit beide Fälle der Aufrufsequenz praktisch gleich behandelt werden können. Schliesslich kapselt sie sämtliche für das Sortieren benötigten List-Eigenschaften in den beiden privaten Methode *le* und *swap*. Die erstere gibt einen booleschen Wert zurück, welcher angibt, ob das durch den ersten Index bezeichnete Listenelement kleiner oder gleich (**l**ess or **e**qual) dem durch den zweiten Index bezeichneten Listenelement ist. Die zweite vertauscht (**swap**) das durch den ersten Index bezeichnete Listenelement mit dem durch den zweiten Index bezeichneten Listenelement. In C/C++ wird man statt einer Klasse *Order* die beiden Prozeduren *le* und *swap* als Prozedurparameter der Sortierfunktionen übergeben.
Mit dieser „Möblierung“ der Klasse *Order* ist die Basis etabliert, auf welcher Sortieralgorithmen implementiert werden können.

ComparableComparator

Die erwähnte private Klasse, welche einen auf der *Comparable*-Schnittstelle basierenden *Comparator* implementiert, sieht so aus:

```
private static class ComparableComparator
  implements Comparator
{
  public int compare(Object o1, Object o2)
  {
    Comparable c1 = (Comparable) o1;
    Comparable c2 = (Comparable) o2;
    return c1.compareTo(c2);
  } /* compare */
} /* class ComparableComparator */
```

le

Die Vergleichsoperation lautet:

```
private static boolean le(List ls, Comparator c, int i1, int i2)
{
  return (c.compare(ls.get(i1),ls.get(i2)) <= 0);
} /* le */
```

swap

Die Vertauschoperation lautet:

```
private static void swap(List ls, int i1, int i2)
{
  Object oTemp = ls.get(i1);
  ls.set(i1,ls.get(i2));
  ls.set(i2,oTemp);
} /* swap */
```

MergeSort

Der *MergeSort*-Algorithmus basiert auf folgender rekursiven Idee: Man sortiere die rechte und die linke Hälfte der Liste und darauf „mischt“ man die beiden Teile. Dabei geht man so vor, dass man das kleinste element der linken Hälfte mit dem kleinsten Element der rechten Hälfte vergleicht und je nach Resultat das eine oder andere als nächstes Element in die endgültige geordnete Liste einfügt.

Für das Mischen benötigt man zusätzlichen Speicherplatz in Form einer Kopie der originalen Liste:

```
public static void sortMerge(List ls, Comparator c)
{
  List lsScratch = new ArrayList(ls);
  sortMerge(ls,lsScratch,c,0,ls.size());
} /* sortMerge */
```

mergeScratch

Mit Hilfe dieser Zusatzliste kann man dann eine linke Hälfte der Liste von *iFrom* bis *iMiddle* mit einer rechten Hälfte von *iMiddle* bis *iRight* folgendermassen mischen:

```
private static void mergeScratch(
  List ls, List lsScratch, Comparator c, int iFrom, int iMiddle, int iTo)
{
  /* merge from target to scratch array */
  int iIndex = iFrom;
  int iLeft = iFrom;
  int iRight = iMiddle;
  while ((iLeft < iMiddle) && (iRight < iTo))
  {
    if (le(ls,c,iLeft,iRight))
    {
      lsScratch.set(iIndex,ls.get(iLeft));
      iLeft++;
    }
    else
    {
      lsScratch.set(iIndex,ls.get(iRight));
      iRight++;
    }
    iIndex++;
  }
  while (iLeft < iMiddle)
  {
    lsScratch.set(iIndex,ls.get(iLeft));
    iLeft++;
    iIndex++;
  }
  while (iRight < iTo)
  {
    lsScratch.set(iIndex,ls.get(iRight));
    iRight++;
    iIndex++;
  }
  /* copy from scratch to target array */
  for (iIndex = iFrom; iIndex < iTo; iIndex++)
    ls.set(iIndex,lsScratch.get(iIndex));
} /* mergeScratch */
```

sortMerge

Der rekursive Sortiervorgang lautet damit so:

```
private static void sortMerge(List ls, List lsScratch, Comparator c, int iFrom, int iTo)
{
  if (iTo - iFrom > 1)
  {
    int iMiddle = (iFrom + iTo)/2;
    sortMerge(ls,lsScratch,c,iFrom,iMiddle);
    sortMerge(ls,lsScratch,c,iMiddle,iTo);
    mergeScratch(ls,lsScratch,c,iFrom,iMiddle,iTo);
  }
} /* sortMerge */
```

Wenn man die Anzahl Vergleiche dieses Sortieralgorithmus mit demjenigen in *Collections.sort()* vergleicht, stellt man fest, dass dort noch ein Zusatz implementiert ist: Falls nach dem Sortieren der Teillisten das letzte Element der linken Teilliste kleiner oder gleich dem ersten Element der rechten Teilliste ist, braucht nicht gemischt zu werden.

```
private static void sortMerge(List ls, List lsScratch, Comparator c, int iFrom, int iTo)
{
  if (iTo - iFrom > 1)
  {
    int iMiddle = (iFrom + iTo)/2;
    sortMerge(ls,lsScratch,c,iFrom,iMiddle);
    sortMerge(ls,lsScratch,c,iMiddle,iTo);
    if (!le(ls,c,iMiddle-1,iMiddle))
      mergeScratch(ls,lsScratch,c,iFrom,iMiddle,iTo);
  }
} /* sortMerge */
```

Dieser Algorithmus scheint dem *Collections.sort()* sehr genau zu entsprechen.

Damit wird eine gewisse Annäherung an die optimale Sortierung weitgehend sortierter Sequenzen erreicht. Dieses Prinzip könnte man erweitern indem man jedesmal mittels binärem Suchen den kleinstmöglichen Mischbereich bestimmt. Das kostet allerdings zusätzliche Vergleiche für das binäre Suchen.

Diese Methode findet das erste Element im Bereich, das grösser als das Testelement ist:

```
private static int search(List ls, Comparator c, int iFrom, int iTo, int iIndex)
{
  int iLow = iFrom - 1;
  int iHigh = iTo;
  int iTemp;
  while (iHigh - iLow > 1)
  {
    /* iIndex is in interval between iLow and iHigh */
    iTemp = (iLow + iHigh) / 2;
    if (le(ls,c,iTemp,iIndex))
      iLow = iTemp;
    else
      iHigh = iTemp;
  }
  return iHigh;
} /* search */
```

Wenn man stipuliert, dass Elemente mit Index kleiner als *iFrom* kleiner oder gleich Allem sind, während Element mit Index grösser oder gleich *iTo* grösser als Alles sind, ist die Korrektheit dieser Methode leicht einzusehen, indem man die Invarianz der Intervallbehauptung zeigt.
Mit Hilfe dieser binären Suche könnte man den Sortieralgorithmus folgendermassen abändern:

```
private static void sortMerge(List ls, List lsScratch, Comparator c, int iFrom, int iTo)
{
  if (iTo - iFrom > 1)
  {
    int iMiddle = (iFrom + iTo)/2;
    sortMerge(ls,lsScratch,c,iFrom,iMiddle);
    sortMerge(ls,lsScratch,c,iMiddle,iTo);
```

```
      if (!le(ls,c,iMiddle-1,iMiddle))
      {
        /* reduce merge range */
        iTo = search(ls,c,iMiddle,iTo,iMiddle-1);
        iFrom = search(ls,c,iFrom,iMiddle,iMiddle);
        mergeScratch(ls,lsScratch,c,iFrom,iMiddle,iTo);
      }
    }
  } /* sortMerge */
```

Es bleibt aber zweifelhaft, ob sich der Zusatzaufwand für die Suche durch Einsparungen im Teilweise sortierten Fall lohnen. Man müsste analog der Behauptung, dass beim QuickSort der ungünstigste Fall der Normalfall ist, ein Mass definieren, wieviel Vorsortierung der Normalfall ist und anhand dieser Annahme prüfen, ob sich die Modifikation lohnt.

sortSwap

Von Siegfried Sielaff stammt eine hübsche Variante des *MergeSort*, welche er *SwapSort* nennt. Es handelt sich um ein „in place"-Verfahren – wenn auch in der hier vorgeführten rekursiven Variante mit einem Stack der Grösse $O(\log n)$. Generell ist hier zu sagen, dass ich die Versionen mit expliziter Rekursion vorführe, weil sie viel einfacher verständlich sind als diejenigen mit eliminierter Endrekursion. Ausserdem sind Call-Overhead und der resultierende Stackbedarf heutzutage wohl kaum mehr wert, dass man dafür Verständlichkeit opfert.
Dieser SwapSort ist stabil und benötigt bei schon sortierten Listen deutlich weniger Vergleiche. Eine untere Grenze wäre aber herzuleiten. Die Methode *sortSwap* sieht genauso aus, wie die Methode *sortMerge*, ausser dass der Aufruf von *mergeScratch* durch denjenigen der neuen Methode *merge* ersetzt wird.
Der Trick besteht darin, dass die *merge*-Methode rekursiv gestaltet wird. Es wird eine Grösse *iSize* bestimmt, welche die Eigenschaft hat, dass die letzten *iSize* Elemente in der linken Teilliste alle grösser sind als die ersten *iSize* Elemente in der rechten Teilliste. Man tauscht diese daher aus und wendet dann den Merge rekursiv an. Einziger Schönheitsfehler dieses Verfahrens: auf den ersten Blick sieht es so aus, als ob die Rekursion den beiden Teilbäume nicht in der Mitte teilt und somit die Rekursionstiefe $O(n \log n)$ nicht gewährleistet ist.

split

Die Bestimmung der Grösse *iSize* wird als binäre Suche implementiert:

```
private static int split(List ls, Comparator c, int iFrom, int iMiddle, int iTo)
{
  int iLow = -1;
  int iHigh = Math.min(iMiddle-iFrom, iTo-iMiddle);
  while (iHigh - iLow > 1)
  {
    /* iSize is in interval between iLow and iHigh */
    int iSize = (iLow + iHigh) / 2;
    if (!le(ls,c,iMiddle-1-iSize,iMiddle+iSize))
      iLow = iSize;
    else
      iHigh = iSize;
  }
  return iHigh;
} /* split */
```

merge

Damit sieht die neue rekursive Methode *merge* für das Mischen „in place" so aus:

```
private static void merge(List ls, Comparator c, int iFrom, int iMiddle, int iTo)
{
  /* split */
  int iSize = split(ls,c,iFrom,iMiddle,iTo);
  if (iSize > 0)
  {
    /* swap last iSize elements of left half with first iSize elements of right half */
    for (int i = 0; i < iSize; i++)
      swap(ls,iMiddle-iSize+i,iMiddle+i);
    /* everything to the left is now less or equal to everything on the right */
    if (iMiddle - iSize > iFrom)
      merge(ls,c,iFrom, iMiddle-iSize, iMiddle);
    if (iTo > iMiddle + iSize)
      merge(ls,c,iMiddle, iMiddle+iSize, iTo);
  }
} /* merge */
```

Bei diesem Verfahren ist noch zu berücksichtigen, dass es mehr Vertauschungen als Vergleiche kostet. Da aber im allgemeinen Vertauschungen von Pointers stattfinden, während Vergleichsoperationen teuer werden können, ist dies in der Praxis vorzuziehen.
Da die Summe der Längen der zu mischenden Teile bei der Rekursion immer halbiert wird, ist dieser Algorithmus tatsächlich begrenzt durch $O(n \log n)$ Vergleiche. Sielaff gibt noch eine Reihe von Verbesserungen seines Algorithmus an, wo aber die einfache Vertauschungsoperation zweier Elemente durch zyklisches Vertauschen einer grösseren Anzahl Elemente ersetzt wird.

HeapSort

Auch Dijkstras *SmoothSort* ist eine Variante des *HeapSort*. Dieser basiert auf der Grundidee, dass die Daten zuerst halbgeordnet in einem „Heap“ arrangiert werden und in einem zweiten Schritt, dieser in eine Totalordnung übergeführt wird.

Heap

Als „Heap“ bezeichnet man einen (meist binären) Baum, wo die Wurzel grösser oder gleich aller ihrer Kinder ist und wo alle Teilbäume ihrerseits die Heapeigenschaft besitzen. Wenn man Daten in einem Heap arrangiert hat, sind sie zwar nicht sortiert, sie sind aber auch nicht mehr unsortiert. Vielmehr haben wir eine Art Halbordnung erzeugt.
Alle *HeapSort*-Algorithmen basieren darauf, dass man zuerst aus einer unsortierten Liste einen Heap aufbaut und darauf den Heap abbaut und daraus eine sortierte Liste herstellt. Bei beiden Operationen stösst man auf die Situation, wo man fast einen Heap hat. D.h. mit der möglichen Ausnahme der Wurzel erfüllen alle anderen Teile des Baums die Heapeigenschaft. Die Operation *sift* verwandelt einen solchen Beinahe-Heap in einen echten Heap, indem man die Wurzel mit dem Maximum der Söhne vertauscht, wenn die Heapeigenschaft nicht erfüllt ist, und dies rekursiv fortführt, bis ein Heap vorliegt. Diese Operation benötigt maximal soviele Vergleiche und Vertauschungen wie der Baum tief ist. Wenn dieser einigermassen balanciert ist, sind dies $O(\log n)$ Vergleiche.
Nach dieser Beschreibung ist noch offen, wie man eine lineare Liste auf einen solchen Baum abbildet. Im „klassischen“ *HeapSort* ist die Baumstruktur der Elemente mit Index $0 \ldots n-1$ gegeben durch die Angabe, dass die Kinder eines Elements mit Index i die Elemente mit den Indizes $2i+1$ und $2i+2$ sind, sofern diese kleiner als n sind. Falls keiner dieser Indizes kleiner als n ist, handelt es sich um ein Blatt des Baums. Falls nur einer kleiner als n ist, hat der Teilbaum bei i nur ein Kind.
Dijkstra hat die schöne Sitte eingeführt, Prozeduren als Prädikatentransformationen zu betrachten. Besonders beim Suchen und Sortieren erweist es sich als äusserst nützlich, seine Technik der „Beweisens“ von Programmen wenigestens als Richtlinie zu benutzen, da rein „intuitives“ Programmieren auf diesem Gebiet auf endlose Irrwege führt.

isTrusty

Die Methode *isTrusty* verkörpert die Heapeigenschaft des Teilbaums bei *iRoot*:

```
private static boolean isTrusty(List ls, Comparator c, int iRoot, int iSize)
{
  boolean bTrusty = isDubious(ls,c,iRoot,iSize);
  int iLeft = 2*iRoot + 1;
  if (iLeft < iSize) // do we have a left son?
  {
    if (le(ls,c,iLeft,iRoot)) // is left son less or equal to root?
    {
      int iRight = 2*iRoot + 2;
      if (iRight < iSize) // do we have a right son?
      {
        if(!le(ls,c,iRight,iRoot)) // is right son less or equal to root?
          bTrusty = false;
      }
    }
    else
      bTrusty = false;
  }
  return bTrusty;
} /* isTrusty */
```

isDubious

Die Methode *isDubious* verkörpert die Fast-Heap-Eigenschaft, d.h. die Heapeigenschaft mit möglicher Ausnahme bei der Wurzel des Teilbaums bei *iRoot*.

```
private static boolean isDubious(List ls, Comparator c, int iRoot, int iSize)
{
  boolean bDubious = true;
  int iLeft = 2*iRoot + 1;
  if (iLeft < iSize) // do we have a left son?
  {
    if (isTrusty(ls,c,iLeft,iRoot)) // is sub tree with root iLeft trusty?
    {
      int iRight = 2*iRoot + 2;
      if (iRight < iSize) // do we have a right son?
      {
        if (!isTrusty(ls,c,iRight,iRoot)) // is sub tree with root iRight trusty?
          bDubious = false;
      }
    }
    else
      bDubious = false;
  }
  return bDubious;
} /* isDubious */
```

sift

Die Methode *sift* verwandelt einen zweifelhaften Teilbaum bei *iRoot* in einen vertrauenswürdigen.

```
private static void sift(List ls, Comparator c, int iRoot, int iSize)
{
  assert isDubious(ls,c,iRoot,iSize): "precondition of sift not fulfilled!";
  int iLeft = 2*iRoot + 1;
  if (iLeft < iSize)
  {
    int iRight = iLeft + 1;
    if (iRight < iSize)
    {
      if (!le(ls,c,iRight,iLeft))
        iLeft = iRight;
    }
    /* iLeft is now maximum (or only) son */
    if (!le(ls,c,iLeft,iRoot))
    {
      swap(ls,iLeft,iRoot);
```

```
      /* tail recursion */
      sift(ls,c,iLeft,iSize);
    }
  }
  assert isTrusty(ls,c,iRoot,iSize): "postcondition of sift not fulfilled!";
} /* sift */
```

isSortedHigh

Im Laufe des Sortierens ist jeweils der rechte Teil der Liste schon sortiert. Dies drückt sich in der Bedingung *isSortedHigh* aus. Diese Bedingung hat nichts mit der Heapeigenschaft zu tun, sondern behandelt die Elemente von ls als lineare Liste. Sie besagt, dass die Elemente von *iRoot* bis *iSize*-1 in einer Totalordnung vorliegen und dass alle Elemente von 0 bis *iRoot*-1 kleiner oder gleich dem ersten Element aus dieser Teilliste ist, sofern diese nicht leer ist.

```
private static boolean isSortedHigh(List ls, Comparator c, int iRoot, int iSize)
{
  boolean bSorted = true;
  for (int i = iRoot+1; bSorted && (i < iSize); i++)
  {
    if (!le(ls,c,i-1,i))
      bSorted = false;
  }
  if (iRoot < iSize)
  {
    for (int i = 0; bSorted && (i < iRoot); i++)
    {
      if (!le(ls,c,i,iRoot))
        bSorted = false;
    }
  }
  return bSorted;
} /* isSortedHigh */
```

sortHeap

Mit Hilfe dieser Strukturmethoden für binäre Bäume, sieht die Implementation von *HeapSort* nun so aus:

```
public static void sortHeap(List ls, Comparator c)
{
  int iSize = ls.size();
  /* build heap */
  for (int iRoot = iSize/2; iRoot > 0; )
  {
    iRoot--;
    sift(ls,c,iRoot,iSize);
  }
  /* dismantle heap */
  for (int iRoot = iSize; iRoot > 1; )
  {
    assert isSortedHigh(ls,c,iRoot,iSize): "dismantle invariant in sortHeap not fulfilled!";
    iRoot--;
    swap(ls,iRoot,0);
    sift(ls,c,0,iRoot);
  } /* sortHeap */
} /* sortHeap */
```

In der ersten Schleife sorgt man dafür, dass alle Teilbäume rechts von *iRoot* vertrauenswürdige Heaps werden. Da die obere Hälfte ohnehin aus Blättern des Baumes besteht, kann man bei $iSize/2$ anfangen.
In der zweiten Schleife ist die Zusatzinvariante erfüllt, dass von *iRoot* an rechts schon die grössten *iSize-iRoot* Elemente der Liste in der endgültigen Sortierung eingeordnet sind, während die verbleibenden Elemente der Liste links in einem vertrauenswürdigen Heap vorliegen.
Da in diesem Heap das Element mit Index 0 gleich dem Maximum der Elemente von 0 bis *iRoot*-1 ist, kann man *iRoot* dekrementieren und Element 0 mit Element *iRoot* vertauschen. Dadurch bleibt die Bedingung erfüllt, dass rechts von *iRoot* die grössten Elemente in der endgültigen Ordnung vorliegen. Um nun die Bedingung wieder herzustellen, dass links ein vollständiger Heap vorliegt, muss man die Operation *sift* durchführen, welche den zweifelhaften in einen vertrauenswürdigen Heap verwandelt.
Natürlich darf man die Anzahl Vergleiche und Vertauschungen nur messen, wenn die Assertions nicht ausgewertet werden. Typischerweise wertet man sie beim Entwicklen und Testen im Debug-Modus aus, während sie zur Laufzeit abgeschaltet sind.

Sequenz

Nun kommen wir endlich zum *SmoothSort* von Edsger Dijkstra. Es handelt sich um eine Variante des *HeapSort*, welche schon vorhandene Sortierungen auszunützen versucht. Beim „klassischen“ *HeapSort* ist die Baumstruktur sehr „binär“ auf die lineare Liste verteilt und dadurch wird eine anfänglich vorhandene Ordnung kräftig durcheinander gemischt, bevor sie danach wieder herstellt wird. Auch benötigt *HeapSort* auch bei einer total sortiert vorliegenden Liste immer $O(n \log n)$ Vergleiche.

Dijkstras Grundidee ist nun, dass man den Heap nicht binär auf die Liste verteilt, sondern als „post order traversal“. D.h. in der Liste kommt immer zuerst der linke Teilbaum, dann der rechte und dann die Wurzel. Wenn man die binären Bäume so anordnet, ergibt sich sofort, dass diese immer komplette binäre Bäume sein müssen (man kann nicht einen Teilbaum weglassen), und, dass solche binären Bäume nur gewisse feste Baumgrössen S_i haben können, die folgender Rekursion gehorchen: $S_i = S_j + S_k + 1$, wo entweder $j = i-1 \wedge k = i-1$ oder $j = i-2 \wedge k = i-1$ gilt. Im ersten Fall erhält man die Folge (0,) 1, 3, 7, 15, 31 ... bzw. allgemein $S_i = 2^i - 1$, im zweiten Fall die Folge (-1), 1, 1, 3, 5, 9, ... Diese Zahlenfolge ist offenbar nahe mit derjenigen der Fibonacci-Zahlen verwandt und wird von Dijkstra als Folge der Leonardo-Zahlen bezeichnet. Der *SmoothSort*-Algorithmus kann mit beiden Sequenzen realisiert werden.

Wenn man nun eine Liste beliebiger Grösse sortieren will, muss man in Kauf nehmen, dass diese aus mehreren binären Bäumen im „post order traversal“ zusammengesetzt wird. Zweckmässigerweise fängt man mit dem grössten binären Baum an, der innerhalb der Liste platz hat und füllt dann mit sukzessive kleiner werdenden Bäumen auf. Eine solche Sequenz von binären Bäumen nennt Dijkstra eine „concatenation sequence“. Hier nenne ich sie einfach Sequenz. Da es für das Laufzeitverhalten des Algorithmus wünschenswert ist, die Anzahl der binären Bäume in einer Sequenz minimal zu halten, ist es vorteilhaft, die Leonardo-Zahlen statt der balanciert-binären Baumgrössen zu verwenden. Diese Bemerkung ist verwandt mit der Tatsache, dass man bei Stückelung der Währung in Fibonacci-Beträge eine minimale Anzahl Münzen benötigt, um jeden beliebigen Preis zu bezahlen.

Um nun eine Sequenz an der Stelle *iRoot* zu codieren, werden zwei Zahlen benutzt: Die Grösse *iCurrent* S_n des binären Teilbaums, dessen Wurzel *iRoot* ist, die vorhergehende Grösse *iPrevious* S_{n-1} , die für die rekursive Berechnung der Leonardo-Zahlen benötigt wird, und eine „Bitmap“ *iStretches*, in der das i-te Bit gesetzt ist, wenn links von *iRoot* (exklusive den Stretch, der *iRoot* enthält) ein binärer Teilbaum der Grösse S_{n+i} vorkommt. Dort können nur grössere

vorkommen. Ausserdem kann jede Grösse höchstens einmal vorkommen, ausser am rechten Ende der Sequenz, wo zwei gleich grosse Teilbäume vorkommen können. (Im Leonardo-Fall können diese nur die Grösse 1 haben.)

Noch eine Bemerkung zur maximalen Sortiergrösse. Es dürfte klar sein, dass mit dieser „Bitmap“ mit 32-bit Integers der maximale binäre Baum höchstens rund eine Milliarde Elemente haben kann. Da auf traditionellen Architekturen der Adressraum ohnehin auf vier Milliarden Bytes beschränkt ist und pro Element wohl mindestens ein Pointer von 4 Bytes benötigt wird, ist die Verwendung von 32-bit Integers durchaus angebracht. Wer für die Einführung von 64-bit-Architekturen gewappnet sein will, sollte die drei, die Sequenz beschreibenden Zahlen zu 64-bit Integers (in JAVA als „long“ bezeichnet) machen.

Für die JAVA-Implementation verwandeln wir die Codierung der Sequenz in drei Integers in eine Class, damit wir sie in Manipulationsmethoden gleichzeitig verändern können. Um eine optimale Performance zu erreichen, ist es wünschenswert, den Objekt- und Aufruf-Aufwand für diese Kleinstmethoden zu vermeiden, indem man direkt im Code die paar Zeilen einfügt, welche benötigt werden. In C/C++ würde sich eine Ausgestaltung in Form von Preprocessor-Makros besonders gut eignen.

Leonardo

Die Klasse der Leonardo-Zahlen und ihre Manipulationen implementiert man so:

```
private static class Leonardo
{
  /* data members */
  private int m_iSize;
  private int m_iPrevious;
  /* access */
  public int getSize() { return m_iSize; }
  private void setSize(int iSize) { m_iSize = iSize; }
  private int getPrevious() { return m_iPrevious; }
  private void setPrevious(int iPrevious) { m_iPrevious = iPrevious; }
  /* default constructor */
  public Leonardo()
  {
    setSize(1);
    setPrevious(1);
  }
  /* copy constructor */
  public Leonardo(Leonardo lSize)
  {
    setSize(lSize.getSize());
    setPrevious(lSize.getPrevious());
  } /* copy constructor */
  /* recursion */
  public void up()
  {
    int iTemp = m_iSize;
    setSize(m_iPrevious + m_iSize + 1);
    setPrevious(iTemp);
  }
```

```
  public void down()
  {
    int iTemp = m_iPrevious;
    setPrevious(m_iSize - m_iPrevious - 1);
    setSize(iTemp);
  }
} /* class Leonardo */
```

Die „Verwaltung“ der in der Sequenz verwendeten Grössen von binären Bäumen sieht so aus:

```
private static class Sequence
{
  /* data members */
  private Leonardo m_lSize;
  private int m_iStretches;
  /* access */
  public Leonardo getSize() { return m_lSize; }
  private void setSize(Leonardo lSize ) { m_lSize = lSize; }
  private int getStretches() { return m_iStretches; }
  private void setStretches(int iStretches) { m_iStretches = iStretches; }
  /** default constructor */
  public Sequence()
  {
    setSize(new Leonardo());
    setStretches(0);
  }
  /** copy constructor */
  public Sequence(Sequence sSize)
  {
    setSize(new Leonardo(sSize.getSize()));
    setStretches(sSize.getStretches());
  }
  /** returns underlying current size of binary heap */
  public int size()
  {
    return m_lSize.getSize();
  } /* size */
  /** empty returns true, if there are no stretches to the left */
  private boolean empty()
  {
    return m_iStretches == 0;
  } /* empty */
  /** length returns total size of ternary heap */
  public int length()
  {
    int iLength = size();
    Sequence s = new Sequence(this);
    while (!s.empty())
    {
```

```
    s.upToPrevious();
    iLength = iLength + s.size();
  }
  return iLength;
} /* length */
/** mergeable is given, if bit 1 is set */
public boolean mergeable()
{
  return (m_iStretches & 2) != 0;
} /* mergeable */
/** permanent returns true, if ternary tree at root is permanent */
public boolean permanent(int iRoot, int iTotalSize)
{
  boolean bPermanent = true;
  int iSize = m_lSize.getPrevious();
  if (iSize < 0)
    iSize = 0;
  if ((iRoot + 1) + iSize + 1 <= iTotalSize)
    bPermanent = false;
  return bPermanent;
} /* permanent */
/** up goes to next size */
public void up()
{
  m_lSize.up(); /* size of left stretch */
  setStretches(m_iStretches >> 1);
} /* up */
/** down goes to previous size */
public void down()
{
  m_lSize.down();
  setStretches(m_iStretches << 1);
} /* down */
/** upToPrevious finds next stretch size in sequence */
public void upToPrevious()
{
  /* find next occupied stretch */
  while (!empty() && !mergeable())
    up();
  up();
  clear();
} /* upToPrevious */
/** downToOne reduces current size to one */
public void downToOne()
{
  set();
  /* go down to one */
  down();
  while (size() > 1)
    down();
```

```
  } /* downToOne */
  /** set adds current size to left sequence */
  public void set()
  {
    m_iStretches = m_iStretches | 1;
  } /* set */
  /** clear removes current size from left sequence */
  public void clear()
  {
    m_iStretches = m_iStretches & ~1;
  } /* clear */
} /* class Sequence */
```

Binäre und ternäre Teilbäume

In einem gewissen Sinn kann man eine Sequenz als ternären Baum betrachten. Die Wurzel ganz rechts hat als rechten und mittleren Teilbaum einen binären Baum und ganz links den Rest der Sequenz als ternären Baum. Es ist somit nicht verwunderlich, dass die Eigenschaften *isTrusty* und *isDubious* nun je in einer binären als auch ternären Variante benötigt werden. Dasselbe gilt von der Methode *sift*.

isTrusty2

Der Test der „Zuverlässigkeit“ des binären Baums an der Stelle *iRoot*, dessen Grösse durch *lSize* gegeben ist, sieht so aus:

```
private static boolean isTrusty2(List ls, Comparator c, int iRoot, Leonardo lSize)
{
  boolean bTrusty = isDubious2(ls,c,iRoot,lSize);
  int iSize = lSize.getSize();
  if (iSize > 1)
  {
    Leonardo l = new Leonardo(lSize);
    l.down();
    int iLeft = iRoot - iSize + l.getSize();
    bTrusty = bTrusty && le(ls,c,iLeft,iRoot);
    l.down();
    int iRight = iRoot - 1;
    bTrusty = bTrusty && le(ls,c,iRight,iRoot);
  }
  return bTrusty;
} /* isTrusty2 */
```

isDubious2

Der zugehörige Test der Zweifelhaftigkeit lautet:

```
private static boolean isDubious2(List ls, Comparator c, int iRoot, Leonardo lSize)
```

```
{
  boolean bTrusty = true;
  int iSize = lSize.getSize();
  if (iSize > 1)
  {
    Leonardo l = new Leonardo(lSize);
    l.down();
    int iLeft = iRoot - iSize + l.getSize();
    bTrusty = bTrusty && isTrusty2(ls,c,iLeft,l);
    l.down();
    int iRight = iRoot - 1;
    bTrusty = bTrusty && isTrusty2(ls,c,iRight,l);
  }
  return bTrusty;
} /* isDubious2 */
```

sift2

Die *sift*-Methode, welche den binären Baum an der Stelle *iRoot* mit der durch *lSize* gegebenen Grösse, verwandelt einen zweifelhaften in einen vertrauenswürdigen binären Baum:

```
private static void sift2(List ls, Comparator c, int iRoot, Leonardo lSize)
{
  assert isDubious2(ls,c,iRoot,lSize):"precondition of sift2 not fulfilled!";
  int iSize = lSize.getSize();
  if (iSize > 1)
  {
    Leonardo l = new Leonardo(lSize);
    /* left sub tree */
    l.down();
    int iLeft = iRoot - iSize + l.getSize();
    int iRight = iRoot - 1;
    /* make iLeft maximum son of balanced tree */
    if (!le(ls,c,iRight,iLeft))
    {
      iLeft = iRight;
      l.down();
    }
    if (!le(ls,c,iLeft,iRoot))
    {
      swap(ls,iLeft,iRoot);
      /* tail recursion */
      sift2(ls,c,iLeft,l);
    }
  }
  assert isTrusty2(ls,c,iRoot,lSize):"postcondition of sift2 not fulfilled!";
} /* sift2 */
```

isTrusty3

Die Vertrauenswürdigkeit des ternären Baums muss die ganze Sequenz der binären Bäume testen.

```
private static boolean isTrusty3(List ls, Comparator c, int iRoot, Sequence sSize)
{
  boolean bTrusty = isDubious3(ls,c,iRoot,sSize);
  bTrusty = bTrusty && isTrusty2(ls,c,iRoot,sSize.getSize());
  int iSize = sSize.size();
  if (iSize > 1)
  {
    Leonardo l = new Leonardo(sSize.getSize());
    l.down();
    int iLeft = iRoot - iSize + l.getSize();
    bTrusty = bTrusty && le(ls,c,iLeft,iRoot);
    l.down();
    int iRight = iRoot - 1;
    bTrusty = bTrusty && le(ls,c,iRight,iRoot);
  }
  int iStep = iRoot - iSize;
  if (iStep >= 0)
    bTrusty = bTrusty && le(ls,c,iStep,iRoot);
  return bTrusty;
} /* isTrusty3 */
```

isDubious3

Damit das funktioniert, muss die Zweifelhaftigkeit analog definiert sein.

```
private static boolean isDubious3(List ls, Comparator c, int iRoot, Sequence sSize)
{
  boolean bDubious = isDubious2(ls,c,iRoot,sSize.getSize());
  int iSize = sSize.size();
  int iStep = iRoot - iSize;
  if (iStep >= 0)
  {
    Sequence s = new Sequence(sSize);
    s.upToPrevious();
    bDubious = bDubious && isTrusty3(ls,c,iStep,s);
  }
  return bDubious;
} /* isDubious3 */
```

sift3

Die Methode *sift3* implementiert dann die Verwandlung eines zweifelhaften in einen vertrauenswürdigen ternären Baum. Sie entspricht der Funktion, die in Dijkstras Text *trinkle* heisst.

```
private static void sift3(List ls, Comparator c, int iRoot, Sequence sSize)
{
  assert isDubious3(ls,c,iRoot,sSize): "precondition of sift3 not fulfilled!";
  int iSize = sSize.size();
  int iStep = iRoot - iSize;
  if (iStep >= 0)
  {
    if (iSize > 1)
    {
      Leonardo l = new Leonardo(sSize.getSize());
      /* left son of balanced tree */
      l.down();
      int iLeft = iRoot - iSize + l.getSize();
      /* right son of balanced tree */
      int iRight = iRoot - 1;
      /* make iLeft maximum son of balanced tree */
      if (!le(ls,c,iRight,iLeft))
      {
        iLeft = iRight;
        l.down();
      }
      if (le(ls,c,iStep,iLeft))
      {
        /* just like siftTwo */
        if (!le(ls,c,iLeft,iRoot))
        {
          swap(ls,iLeft,iRoot);
          sift2(ls,c,iLeft,l);
        }
        iStep = -1; // already sifted
      }
    }
    if (iStep >= 0)
    {
      /* step son is greater than either son in balanced tree */
      if (!le(ls,c,iStep,iRoot))
      {
        swap(ls,iStep,iRoot);
        /* tail recursion */
        Sequence s = new Sequence(sSize);
        s.upToPrevious();
        sift3(ls,c,iStep,s);
      }
    }
  }
  else
    sift2(ls,c,iRoot,sSize.getSize());
  assert isTrusty3(ls,c,iRoot,sSize): "postcondition of sift3 not fulfilled!";
} /* sift3 */
```

sortSmooth – erster Versuch

Mit Hilfe dieser eingeführten Elemente, kann man schon einen Sortieralgorithmus implementieren, welcher ternäre Heaps auf- und abbaut. Dieser würde etwa so aussehen:

```
public static void sortSmooth(List ls, Comparator c)
{
  /* build ternary heap */
  Sequence sSize = new Sequence();
  int iRoot = 0;
  while (iRoot < ls.size()-1)
  {
    assert sSize.length() == iRoot + 1: "length of sequence on building incorrect!";
    assert isTrusty3(ls,c,iRoot,sSize):"loop invariant in building sequence not fulfilled!";
    iRoot++;
    /* reestablish invariant for next iRoot */
    if (sSize.mergeable())
    {
      sSize.up();
      sSize.up();
    }
    else
      sSize.downToOne();
    sift3(ls,c,iRoot,sSize);
  }
  /* dismantle ternary heap */
  while (iRoot > 1)
  {
    assert sSize.length() == iRoot + 1: "length of sequence on dismantling incorrect!";
    assert isTrusty3(ls,c,iRoot,sSize): "loop invariant in dismantling sequence not fulfilled!";
    assert isSortedHigh(ls,c,iRoot,ls.size()): "right hand side is not ordered on dismantling!";
    /* reestablish invariant */
    if (sSize.size() > 1)
    {
      /* two sub heaps need to be integrated in ternary heap */
      int iSize = sSize.size();
      /* size of left stretch */
      sSize.down();
      int iLeft = iRoot - iSize + sSize.size();
      sift3(ls,c,iLeft,sSize);
      /* add left sub tree to concatenation sequence */
      sSize.set();
      /* size of right stretch */
      sSize.down();
      int iRight = iRoot - 1;
      sift3(ls,c,iRight,sSize);
    }
    else
      sSize.upToPrevious();
    iRoot--;
```

```
    }
  } /* sortSmooth */
```

Die Assertion *isSortedHigh* entspricht Dijkstras Prädikat *P0*. Die Assertion *isTrusty3* entspricht Dijkstras Prädikaten *P1*, *P3* und *P4*. Die Assertion *sSize.length() == iRoot* + 1 entspricht Dijkstras Prädikat *P2*, wobei zu berücksichtigen ist, dass die Definition der Bitmap leicht von derjenigen bei Dijkstra abweicht, um eine allfällige Umstellung auf binäre Stretchgrösse zu erleichtern.

Der Grund, dass diese Version des Sortierens noch nicht die endgültige ist, liegt vor allem darin begründet, dass hier immer noch zuviel „gearbeitet" wird. So dürfte klar sein, dass es beim Aufbauen des ternären Baums nur dann nötig ist, die ternäre Vertrauenswürdigkeit herzustellen, wenn der momentane Stretch nicht sowieso in einem grösseren binären Baum absorbiert wird. Analog kann beim Abbauen der Sequenz benutzt werden, dass schon bekannt ist, dass alle binären Teilbäume binär vertrauenswürdig sind.

Die erste dieser Beobachtungen hat Dijkstra in seinem Artikel nicht völlig korrekt mit seinen Bedingungen *P3'* und *P4'* abzufangen versucht. Es geht aber nicht darum, die Bedingung *isTrusty3* nur vom Vorgänger des momentanen binären Baums zu fordern, sondern darum, sie nur dann zu fordern, wenn der momentane binäre Baum „permanent" ist.

Die zweite dieser Beobachtungen hat Dijkstra in seiner Prozedur *semitrinkle* benutzt. Wir berücksichtigen hier beide Fälle, indem wir geeignete Prädikate definieren.

isHalfDubious3

Die Bedingung isHalfDubious3 unterscheidet sich von isDubious3 nur darin, dass auf der ersten Zeile vorausgesetzt wird, der momentane binäre Teilbaum sei schon vertrauenswürdig.

```
private static boolean isHalfDubious3(List ls, Comparator c, int iRoot, Sequence sSize)
{
  boolean bDubious = isTrusty2(ls,c,iRoot,sSize.getSize());
  int iSize = sSize.getSize().getSize();
  int iStep = iRoot - iSize;
  if (iStep >= 0)
  {
    Sequence s = new Sequence(sSize);
    s.upToPrevious();
    bDubious = bDubious && isTrusty3(ls,c,iStep,s);
  }
  return bDubious;
} /* isHalfDubious3 */
```

siftHalf3

Die Methode *siftHalf3* kann sich entsprechend darauf konzentrieren, nur den Stiefsohn zu testen.

```
private static void siftHalf3(List ls, Comparator c, int iRoot, Sequence sSize)
{
  assert isHalfDubious3(ls,c,iRoot,sSize): "precondition of siftHalf3 not fulfilled!";
  int iSize = sSize.getSize().getSize();
```

```
  int iStep = iRoot - iSize;
  if (iStep >= 0)
  {
    /* root is not less than left or right */
    /* step son (exists!) is root of next stretch to the left */
    if (!le(ls,c,iStep,iRoot))
    {
      swap(ls,iStep,iRoot);
      /* tail recursion */
      Sequence s = new Sequence(sSize);
      s.upToPrevious();
      sift3(ls,c,iStep,s);
    }
  }
  assert isTrusty3(ls,c,iRoot,sSize): "postcondition of siftHalf3 not fulfilled!";
} /* siftHalf3 */
```

Es ist anhand der Prädikatentransformation offensichtlich, dass wir beim Abbau des ternären Baums alle *sift3* durch *siftHalf3* ersetzen können, ohne die Korrektheit des Algorithmus zu verändern. Damit ersparen wir uns einige überflüssige Vergleiche.

isPartiallyTrusty3

Die Bedingung *isPartiallyTrusty3* bedeutet, dass alle Stretches in der Sequenz vertrauenswürdige binäre Heaps sind und der „permanente“ Teil der Sequenz ein vertrauenswürdiger ternärer Heap ist.

```
private static boolean isPartiallyTrusty3(
  List ls, Comparator c, int iRoot, Sequence sSize, int iTotalSize)
{
  boolean bTrusty = true;
  /* if permanent then trusty3 else (trusty2 and stepson is partiallytrusty3) */
  if (sSize.permanent(iRoot,iTotalSize))
    bTrusty = isTrusty3(ls,c,iRoot,sSize);
  else
  {
    bTrusty = isTrusty2(ls,c,iRoot,sSize.getSize());
    int iSize = sSize.size();
    int iStep = iRoot - iSize;
    if (iStep >= 0)
    {
      Sequence s = new Sequence(sSize);
      s.upToPrevious();
      bTrusty = bTrusty && isPartiallyTrusty3(ls,c,iStep,s,iTotalSize);
    }
  }
  return bTrusty;
} /* isPartiallyTrusty3 */
```

isPartiallyDubious3

Die Bedingung *isPartiallyDubious3* bedeutet, dass die Sequenz partiell vertrauenswürdig ist, ausser vielleicht an der Wurzel.

```
  private static boolean isPartiallyDubious3(
    List ls, Comparator c, int iRoot, Sequence sSize, int iTotalSize)
  {
    boolean bDubious = true;
    /* if permanent the dubious3 else (dubious2 and stepson is partiallytrusty3) */
    if (sSize.permanent(iRoot,iTotalSize))
      bDubious = isDubious3(ls,c,iRoot,sSize);
    else
    {
      bDubious = isDubious2(ls,c,iRoot,sSize.getSize());
      int iSize = sSize.size();
      int iStep = iRoot - iSize;
      if (iStep >= 0)
      {
        Sequence s = new Sequence(sSize);
        s.upToPrevious();
        bDubious = bDubious && isPartiallyTrusty3(ls,c,iStep,s,iTotalSize);
      }
    }
    return bDubious;
  } /* isPartiallyDubious3 */
```

siftPartially3

Die Methode *siftPartially3* ist nun die Prädikatentransformation von *isPartiallyDubious3* zu *isPartiallyTrusty3*:

```
  private static void siftPartially3(List ls, Comparator c, int iRoot, Sequence sSize, int
iTotalSize)
  {
    assert isPartiallyDubious3(ls,c,iRoot,sSize, iTotalSize):
           "precondition of siftPartially3 not fulfilled!";
    if (sSize.permanent(iRoot,iTotalSize))
      sift3(ls,c,iRoot,sSize);
    else
      sift2(ls,c,iRoot,sSize.getSize());
    assert isPartiallyTrusty3(ls,c,iRoot,sSize,iTotalSize):
           "postcondition of siftPartially3 not fulfilled!";
  } /* siftPartially3 */
```

sortSmooth

Mit diesen Reduktionen lautet nun die endgültige Implementation von *SmoothSort*:

```
  public static void sortSmooth(List ls, Comparator c)
```

```
{
  /* build ternary heap */
  Sequence sSize = new Sequence();
  int iRoot = 0;
  while (iRoot < ls.size()-1)
  {
    assert sSize.length() == iRoot + 1: "length of sequence on building incorrect!";
    assert isPartiallyTrusty3(ls,c,iRoot,sSize,ls.size()):
           "loop invariant in building sequence not fulfilled!";
    iRoot++;
    /* reestablish invariant for next iRoot */
    if (sSize.mergeable())
    {
      sSize.up();
      sSize.up();
    }
    else
      sSize.downToOne();
    siftPartially3(ls,c,iRoot,sSize,ls.size());
  }
  /* dismantle ternary heap */
  while (iRoot > 1)
  {
    assert sSize.length() == iRoot + 1: "length of sequence on dismantling incorrect!";
    assert isTrusty3(ls,c,iRoot,sSize): "loop invariant in dismantling sequence not fulfilled!";
    assert isSortedHigh(ls,c,iRoot,ls.size()): "right hand side is not ordered on dismantling!";
    /* reestablish invariant */
    if (sSize.size() > 1)
    {
      /* two sub heaps need to be integrated in ternary heap */
      int iSize = sSize.size();
      /* size of left stretch */
      sSize.down();
      int iLeft = iRoot - iSize + sSize.size();
      siftHalf3(ls,c,iLeft,sSize);
      /* add left sub tree to concatenation sequence */
      sSize.set();
      /* size of right stretch */
      sSize.down();
      int iRight = iRoot - 1;
      siftHalf3(ls,c,iRight,sSize);
    }
    else
      sSize.upToPrevious();
    iRoot--;
  }
} /* sortSmooth */
```

Fazit

Die eher undurchsichtige Herleitung des nicht-trivialen Algorithmus von Dijkstra konnte mit Hilfe seiner Methode der Post- und Preconditions verbessert und korrigiert werden. Damit ist dieser Algorithmus in der Praxis überhaupt erst zugänglich geworden. Man müsste nun ein Mass für die Fastsortiertheit definieren und dann zeigen, wie sich die verschiedenen Algorithmen verhalten. Dank eher einfachen Tricks benötigen alle im Fall der totalen Sortiertheit die minimale Anzahl Vergleiche (n-1) und Vertauschungen (0).

Durch Zufall bin ich bei der Implementation von SmoothSort auf den neuen „in place“-MergeSort *SwapSort* gestossen. Dieser hat sich als in jeder Hinsicht optimal erwiesen: Er ist im Gegensatz zu *SmoothSort* stabil, er ist im Gegensatz zu *MergeSort* „in place“, Er benötigt nur unwesentlich mehr Vergleiche als der klassische *MergeSort*. Er scheint generell weniger Vergleiche zu benötigen als *SmoothSort*. Die Tatsache, dass er etwas mehr Vertauschungen benötigt, muss insbesondere darum nicht stören, da hierfür ohnehin nur Pointer bewegt werden und schnelle Buffer-Moves verwendet werden könnten.

Wenn man diese Sortieralgorithmen in der Praxis anwendet, zeigt sich, dass der „klassische“ *MergeSort* mit modernen Verbesserungen dem *SmoothSort* fast überall überlegen ist. Ausserdem hat er den Vorteil der Stabilität. Die Vergleichbarkeit ist aber nicht ganz gegeben, da es sich nicht um eine „in place“-Sortierung handelt.

Ein sehr guter Kandidat für den „besten“ Sortieralgorithmus ist *SwapSort*, der ebenfalls in der Praxis immer weniger Vergleiche zu benötigen scheint als *SmoothSort*.