Berufsakademie Stuttgart - Fachrichtung Informationstechnik

Studienarbeit

für das 5. und 6. Semester im Studiengang Netz- und Softwaretechnik

Untersuchung und Implementierung des introspektiven Sortieralgorithmus Introsort

Erstellt von: Ralph Unden
TIT04NSA, Matrikelnr. 161979
Asangstr. 48
D - 70329 Stuttgart
u.ralph@gmail.com

Betreuer: Prof. Dr. Gebhardt
Fachleiter Informationstechnik
Rotebühlplatz 41
D-70178 Stuttgart
kfg@ba-stuttgart.de

Bearbeitungszeitraum: Oktober 2006 - April 2007

Inhaltsangabe

Die vorliegende Studienarbeit beschäftigt sich mit dem introspektiven Sortieralgorithmus Introsort, der auf dem bekannten Algorithmus Quicksort basiert. Im Zuge dieser Arbeit soll die Entwicklung von Quicksort, wie es erfunden und schrittweise verbessert wurde, hin zu den Versionen die heutzutage verwendet werden, dargelegt werden. Schließlich wird die Lücke zwischen Introsort und seinem Vorgänger geschlossen und die Funktionsweise von Introsort wird im Detail erläutert. Die Überlegungen zur Effizienz aller Algorithmen werden in einem Benchmark bestätigt und am Ende steht eine kurze Zusammenfassung, die eine Hilfestellung darstellt, einen Algorithmus für ein bestimmtes Szenario auszuwählen. Alle Codebeispiele sind in Java.

Abstract

The paper at hand deals with the introspective sort algorithm Introsort which is based on the popular algorithm Quicksort. Over the course of these pages the steps and improvements Quicksort went through, from the form it was created in to how it is employed today, are displayed. Consequently, the gap from Quicksort to Introsort is closed and Introsort's inner workings are discussed in detail. The considerations regarding the efficiency of the algorithms are confirmed in a benchmark. At the very end a short summary is given that also acts as a guideline to select one of the discussed algorithms in a specific scenario. All code samples are written in Java.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Sortieralgorithmen sind ein zentrales Thema in den Grundlagen der Informatik, da Sortieren eine wiederkehrende Aufgabenstellung ist. So wurden im Lauf der Zeit eine ganze Gruppe von Algorithmen geschaffen, die alle unterschiedliche Ansätze verfolgen. Heute ist Quicksort ein beliebter Sortieralgorithmus, da er einfach und effizient ist. Seit seiner Erfindung haben viele versucht den Algorithmus zu verbessern, so dass aktuell verwendete Versionen sehr gut laufen und gründlich erforscht sind. Einer der jüngeren Verbesserungsvorschläge benennt den Algorithmus sogar um und heißt Introsort. Die Namensgebung liegt darin begründet, dass Introspektion verwendet wird, frei formuliert bedeutet das:

Introspektion bzw. Reflexion (engl. reflection) bedeutet in der Programmierung, dass ein Algorithmus Erkenntnisse über seine eigene Struktur gewinnen kann.

Somit stellt diese Veränderung konzeptionell einen Unterschied zu einem nicht-introspektiven Algorithmus dar. Ein weiterer Grund ist, dass sich hinter dem Ansatz von Introsort nicht nur Quicksort, sondern weitere Algorithmen verbergen. Diese Arbeit soll das angedeutete Konzept vollständig erläutern, in dem die Entwicklung von Quicksort, in seiner einfachsten und klassischen Form, über etliche Verbesserungen hin zu Introsort, und somit einem Sortieralgorithmus auf dem aktuellen Stand der Technik, schrittweise erläutert wird. Parallel werden stets anschauliche Überlegungen und Erläuterungen zur Effizienz der besprochenen Algorithmen angestellt. Die so erarbeiteten Ergebnisse werden in Java implementiert und durch eine ausführliche Testreihe im Kapitel Benchmark validiert.

2 Quicksort

Quicksort ist eines der einfachsten und zugleich schnellsten Sortierverfahren. Es wurde zuerst von C. A. R. Hoare in [Hoare 1961] vorgestellt. Dieser Abschnitt stellt die Funktionsweise vor und betrachtet die Effizienz im besten und schlechtesten Fall.

2.1 Funktionsweise

Quicksort arbeitet nach der Divide-and-Conquer-Strategie (deutsch: “teile und herrsche“), welche zur Lösung eines Problems nach drei Schritten vorgeht. Zunächst wird das Problem in

Teilprobleme zerlegt (divide), daraufhin werden die Teilprobleme einzeln gelöst (conquer). Die Lösungen werden dann wieder so zusammengefügt, dass sich die Lösung des ursprünglichen Problems ergibt (combine). Analog zerlegt Quicksort dabei einen zu sortierenden Datensatz in zwei Teile und sortiert daraufhin die beiden Teile unabhängig voneinander. Das Zusammenfügen zum kompletten, sortierten Datensatz erfordert bei geschickter Implementierung keinen zusätzlichen Aufwand. Die genaue Aufteilung des Datensatzes hängt dabei vom Datensatz selber ab. Praktisch wird dies durch die Wahl eines Pivotelementes realisiert. Nach [Sedgewick 1988] müssen für ein Pivotelement an der Stelle i, das den Datensatz in zwei Teile vom linken Rand l bis $i-1$ und von $i+1$ zum rechten Rand r zerlegt, drei Bedigungen gelten:

a. Das gewählte Pivotelement befindet sich an seinem endgültigen Platz.

b. Alle Elemente des Datensatzes von l bis $i-1$ sind kleiner oder gleich groß wie das Pivotelement.

c. Alle Elemente des Datensatzes von $i+1$ bis r sind größer oder gleich groß wie das Pivotelement.

Quicksort besitzt eine Methode, die dafür sorgt, dass diese Bedingungen erfüllt sind. Dies lässt sich sehr einfach realisieren: Zunächst wird willkürlich ein Pivotelement aus dem Datensatz gewählt. Da nicht sichergestellt ist, dass es die erste Bedingung erfüllt, muss es zunächst an seinen endgültigen Platz gebracht werden. Dies wird dadurch erreicht, dass die zweite und dritte Bedingung für das gewählte Element erfüllt werden, da rein logisch gilt $a \wedge b \rightarrow c$. Die Vorgehensweise dabei ist wie folgt: Es wird jeweils ein Zeiger auf den linken und den rechten Rand des Datensatzes gesetzt. Der linke Zeiger durchsucht den Datensatz von links her bis ein Element gefunden wird, das größer oder gleich groß wie das Pivotelement ist. Der rechte Zeiger durchsucht den Datensatz von rechts her bis ein Element gefunden wird, das kleiner oder gleich groß wie das Pivotelement ist. Die beiden gefundenen Elemente sind offensichtlich in ihrer Lage zum Pivotelement am falschen Platz und werden ausgetauscht. Wird so fortgefahren, ist sichergestellt, dass alle Elemente links vom linken Zeiger kleiner und alle rechts vom rechten Zeiger größer als das Pivotelement sind. Begegnen sich die Zeiger, müssen nur noch das Pivotelement und das Element auf das der linke Zeiger zeigt, vertauscht werden. Die selbe Methode wird rekursiv auf die Felder links und rechts vom Pivotelement

angewandt, bis die Teilfeldgröße minimal ist und die korrekte Sortierung trivial ist (bei einem Element).

2.2 Effizienz

Dieser Abschnitt betrachtet die Effizienz von Quicksort bezüglich des Zeitaufwands. Dabei bezieht sich die Betrachtung, wie in allen folgenden Effizienzbetrachtungen, auf die für Sortieralgorithmen übliche Problemgröße, die Anzahl der zu sortierenden Datensätze.

Zeitaufwand im best case: Im besten Fall zerlegt Quicksort den jeweiligen Datensatz auf jeder Ebene in zwei gleich lange Teile.

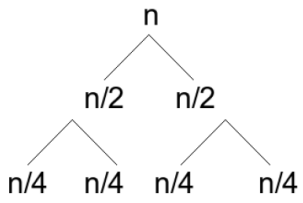

Abbildung 1: Aufrufbaum von Quicksort im best case

Dies geht weiter, bis die Feldgröße 1 ist. Die Rekursionstiefe lässt sich sehr einfach errechnen, wenn man das Problem von unten nach oben betrachtet. Die Frage ist: Wie oft muss die minimale Feldlänge (wie im vorigen Abschnitt erwähnt 1) verdoppelt werden bis man zum Ausgangsfeld kommt?

$$2^x = n \Leftrightarrow x = \frac{log(n)}{log(2)} = log_2(n)$$

Somit ergeben sich $log_2(n)$ Ebenen auf denen jeweils n Elemente bearbeitet werden müssen. Insgesamt ist der Aufwand genau $\Theta(n \cdot log_2(n))$.

Zeitaufwand im worst case: Im schlechtesten Fall zerlegt Quicksort einen Datensatz der Länge n in zwei Teile der Länge 1 und $n - 1$. Dann entartet der Aufrufbaum gemäß folgender Abbildung:

Somit ergeben sich $n - 1$ Ebenen, auf denen sich die Anzahl der zu bearbeitenden Elemente jeweils um 1 verringert. Nach Gauss[1]:

[1]Die Gauss'sche Summenformel wird in dieser Arbeit noch einige Male gebraucht, jedoch wird nicht wiederholt explizit auf ihre Verwendung hingewiesen. Der Beweis für die Aussage befindet sich in Anhang "A2 Beweis der Gauss'schen Summenformel".

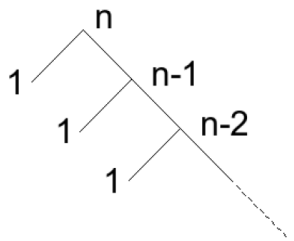

Abbildung 2: Aufrufbaum von Quicksort im worst case

$$n + n - 1 + ... + 1 = \frac{n \cdot (n+1)}{2}$$

Damit ist das Verhalten im worst case durch $\Omega(n^2)$ gegeben. Dieser Fall kann unabhängig von der Strategie zur Wahl des Pivotelementes eintreten. Dies gilt für die Betrachtung aller Quicksortvarianten.

3 Vertiefung spezieller Aspekte von Quicksort

Die Abschnitte dieses Kapitels diskutieren Erweiterungen zu dem grundsätzlichen Konzept von Quicksort aus dem vorigen Kapitel und legen die Basis für spätere Betrachtungen.

3.1 Wahl des Pivotelements

Das Verhalten von Quicksort beim Sortieren wird maßgeblich durch die Wahl des Pivotelements beeinflußt.

Das optimale Pivotelement: Wie in Kapitel 2.2 besprochen, verläuft Quicksort am besten, wenn das Pivotelement den zu sortierenden Datensatz nach der Partitionierung in zwei gleich große Mengen teilt. Außerdem ist nach Kapitel 2.1 Bedingung, dass das Pivotelement sich dann auf seinem endgültigen Platz befindet. Diese Anforderungen erfüllt das wertmäßig mittlere Element des Datensatzes.

Da die Suche nach diesem Element jedoch auf jeder Rekursionsebene zusätzlich einen Aufwand von $\mathcal{O}(n)$ bedeutet und somit die Effizienz von Quicksort deutlich verschlechtern würde, wird diese Möglichkeit von vornherein ausgeschlossen.

Verwendung eines zufälligen Pivotelements: Wird ein zufälliges Pivotelement gewählt ist es unwahrscheinlich, dass das schlechteste Verhalten auftritt. Jedoch wird dem Algo-

rithmus ein nichtdeterministisches Element hinzugefügt. Das heißt, dass die vom Algorithmus benötigte Zeit zum Sortieren des identischen Datensatzes mit jedem Durchlauf variiert. Diese Varianz deckt das volle Spektrum vom best zum worst case ab. Somit kann der Unterschied in der Laufzeit für dasselbe Problem extrem groß sein. Das ist für manche Anwendungen nicht wünschenswert. Außerdem wird ein Zufallszahlengenerator benötigt. Auch wenn die Anforderungen an die erzeugten Zufallszahlen nicht besonders hoch sind, kann dies auf einfachen Hardwareplattformen ein Problem darstellen.

Das Median-of-3 Pivotelement: Eine weitere Alternative ist die Strategie, das positionsmäßig erste, mittlere und letzte Element im Datensatz zu betrachten und aus diesen dreien das wertmäßig mittlere als Pivotelement zu wählen. Dazu wird der Median-of-3 Algorithmus verwendet. Um ein Ergebnis zu erlangen, muss jedes der drei Elemente mit jedem anderen verglichen werden. Tabelle 1 zeigt eine Entscheidungstabelle, die den Ablauf darlegt. Bei der Implementierung mit `if...else` Kontrollstrukturen[2] ist für die Erreichung jeder Ebene ein Vergleich notwendig. Ausnahme sind die beiden äußeren Pfade, die schon nach zwei Vergleichen zum Ergebnis führen. Somit ist der Aufwand sowohl konstant als auch sehr klein. Verglichen werden die drei Elemente a, b und c:

$a > b$	wahr			falsch		
$b > c$	wahr	falsch		wahr		falsch
$a > c$	egal	wahr	falsch	wahr	falsch	egal
Ergebnis:	b	c	a	a	c	b

Tabelle 1: Entscheidungstabelle des Median-of-3 Algorithmus

Diese Methode verbessert Quicksort im Vergleich zu einer trivialen Wahl des Pivotelements, z. B. immer das erste, mittlere oder letzte Element, ungemein. Es wird sehr unwahrscheinlich, dass ein worst-case ähnlicher Fall eintritt. Der absolute worst-case ist sogar ausgeschlossen, da dazu das größte oder kleinste Element ausgewählt werden müsste, jedoch wird im schlimmsten das zweitgrößte oder zweitkleinste Element gewählt (sofern keine Elemente doppelt auftreten). Zusätzlich ist für ein (fast) vorsortiertes Array, was einen häufigen Anwendungsfall darstellt, in den meisten Partitionie-

[2] Siehe die Methode `medianof3()` im Anhang "C2 Die Klasse Introsort".

rungsvorgängen der best case gegeben. In der Praxis wird diese Strategie oft verwendet und hat sich bewährt.

3.2 Gezielte Erzeugung des worst case

Der worst case ist unabhängig von der Wahl des Pivotelements. Für jede Strategie, die so wenig Aufwand mit sich bringt, dass es Sinn ergibt sie zu implementieren, geht der Aufwand gegen $\mathcal{O}(n^2)$, da der Aufrufbaum auf ähnliche Weise entartet. Üblich sind heute Quicksortvarianten, die zufällige Pivotelemente wählen oder die Median-of-3 Strategie verfolgen. Es ist jedoch nicht möglich, einen Datensatz zu erstellen, der gezielt das schlechteste Verhalten in einem Quicksort mit zufälligen Pivotelementen hervorruft. Deswegen beschränkt sich die Betrachtung in dieser Arbeit, auch im Kapitel 5, stets auf die zweite Möglichkeit. Diese spezielle Permutation in einem Datensatz zu betrachten ist essentiell, um die Verbesserung von Introsort gegenüber Quicksort zu messen. In [Musser 1997] werden diese Permutationen als sogenannte Median-of-3-Killersequenzen bezeichnet und nach folgendem Muster beschrieben:

Definition: Sei $k \in \mathbb{N}$, positiv und gerade. Dann ist die Permutation der Zahlen $1, 2, ..., 2k$ eine Median-of-3 Killersequenz:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \dots & k-2 & k-1 & k & k+1 & k+2 & k+3 & \dots & 2k-1 & 2k \\ 1 & k+1 & 3 & k+3 & 5 & \dots & 2k-3 & k-1 & 2k-1 & 2 & 4 & 6 & \dots & 2k-2 & 2k \end{pmatrix}$$

Allerdings wird in der Literatur kein Hinweis auf die algorithmische Erzeugung dieser Permutation gegeben. Dafür ist folgende Beschreibung besser geeignet:

Definition: Sei wie oben $k \in \mathbb{N}$, positiv, gerade und a ein Feld der Länge $2k$ von $a[1]$ bis $a[2k]$. Dann ist für i von 1 bis k bei ungeraden i $a[i] = i$ und $a[i+1] = k + i$. Für alle i ist $a[k + i] = 2 \cdot i$.

Die Umsetzung in eine Javafunktion zur automatischen Erzeugung ist in Anhang "C1 Algorithmische Erzeugung von Median-of-3 Killersequenzen"beschrieben.

3.3 Begrenzung der minimalen Teildateilänge

Eine weitere Verbesserung von Quicksort ist es für kleine Teildateien einen für dieses Problem optimierteren Algorithmus als Quicksort zu verwenden. Quicksort ist in den untersten Ebenen nicht mehr optimal, da für kleine Teildateien mit Längen zwischen 1 und 3 Elementen die

Abarbeitung trivial ist. Eine bessere Methode bietet Insertionsort, welches in Anhang “B1 Insertionsort“ beschrieben wird.

Umgesetzt wird dies in Quicksort, indem nicht mehr überprüft wird, ob der linke Rand kleiner als der rechte Rand ist, sondern, ob der Abstand größer der minimalen Teildateilänge ist, die zugelassen werden soll. Dann ergeben sich jedoch zwei Möglichkeiten zur Behandlung. Entweder wird die Teildatei sofort mit Insertionsort sortiert oder sie bleibt unsortiert und am Ende wird Insertionsort auf das gesamte Feld angewandt. Beide Methoden funktionieren ähnlich gut, wobei die zweite einfacher zu implementieren ist. Nach [Sedgewick 1988] bringt diese Modifikation eine Verkürzung der Laufzeit von circa 20% für minimale Teildatenlängen von 5 bis 25.

4 Introsort

Im folgenden Kapitel wird die Funktionsweise von Introsort konzeptionell sowie implementierungsspezifisch betrachtet. Daraufhin wird Introsort auf seine Effizienz theoretisch untersucht.

4.1 Funktionsweise

Zusätzlich zu den beschriebenen Verbesserungen für Quicksort aus dem vorigen Kapitel führt Introsort eine weitere Abänderung durch, die das introspektive Element integriert. Dies geschieht durch Überwachung der Rekursionstiefe, die der Algorithmus in Abhängigkeit von der Länge des Datensatzes erreicht. Da für dieses Merkmal die Werte in best und worst case bekannt sind, kann ein sinnvoller Wert als Grenze in diesem Intervall gewählt werden. Wird dieser überschritten, so deutet dies darauf hin, dass sich die Laufzeit auf dem Weg befindet quadratisch anzuwachsen. Die Reaktion darauf ist es, die Sortierstrategie zu wechseln. Das heißt, ein anderer Sortieralgorithmus wird auf das Teilproblem angewandt. Hierzu wird in [Musser 1997] Heapsort vorgeschlagen. Die Funktionsweise von Heapsort ist prinzipiell jedoch nicht weiter interessant. Wichtig ist nur das qualitative Verhalten, es könnten auch andere effiziente Sortieralgorithmen verwendet werden. Damit die implementierte Variante jedoch vollständig dokumentiert ist, befindet sich die Beschreibung von Heapsort in Anhang “B2 Heapsort“.

4.2 Implementierung

Um die Implementation möglichst einfach zu halten, werden Arrays sortiert. Statt Iteratorklassen kommen nur Integer als Zeiger zum Einsatz. Das folgende Stück Quelltext zeigt die innere Schleife von Introsort. Die Argumente sind `lo` und `hi`, welche die aktuell zu verarbeitende Teildatei aus dem Gesamtarray nach oben und unten abgrenzen. Die Variable `size_threshold` bezeichnet die in Abschnitt 3.3 diskutierte minimale Teildateilänge. `depth_limit` ist eine Zählvariable, die ganz zu Beginn auf die maximale Rekursionstiefe gesetzt wird. Vor jedem erneuten Aufruf der inneren Schleife wird sie um eins verkleinert. Wird der Wert 0 erreicht, ist dies das Kriterium um, zu Heapsort, zu wechseln. Die Methode `partition()` ist identisch zu ihrem Quicksortäquivalent.

```
1  private static void introsort_loop (int lo, int hi, int depth_limit)
2  {
3          while (hi-lo > size_threshold)
4          {
5                  if (depth_limit == 0)
6                  {
7                          heapsort(lo, hi);
8                          return;
9                  }
10                 depth_limit--;
11                 int p=partition(lo, hi,
12                         medianof3(lo, lo+((hi-lo)/2)+1, hi-1));
13                 introsort_loop(p, hi, depth_limit);
14                 hi=p;
15         }
16 }
```

In dieser Implementierung werden Teildateilängen die kleiner, als der `size_threshold` sind einfach ignoriert, das heißt, am Ende läuft Insertionsort über das ganze Array. Die von Musser vorgeschlagene Länge ist 16. Jenachdem wie wenig zufällig die zu sortierenden Datenfelder sind, lässt sich Introsort für eine konkrete Anwendung optimieren. Zu beachten ist auch, dass `size_threshold` Vorrang vor dem Parameter `depth_limit` hat, welcher somit die maximale Anzahl an Rekursionsebenen zur Erreichung der minimalen Teildateilänge angibt. Wird diese erhöht, kann die Tiefenbegrenzung verkleinert werden und umgekehrt.

4.3 Effizienz

Da bereits eine Effizienzbetrachtung zu Quicksort durchgeführt wurde, ist es nicht notwendig für Introsort in best und worst case zu unterscheiden. Introsort verhält sich im besten Fall genau so wie Quicksort. Die Operationen zur Verwaltung und Überprüfung der Rekursionstiefe stellen einen Overhead dar, der in jedem Teildurchlauf auftritt. Dabei handelt es sich um einen kleinen, konstanten Faktor, der das qualitative Verhalten nicht verändert. Von Bedeutung hingegen ist die Betrachtung des worst case Verhaltens.

Bis die Rekursionstiefenbegrenzung greift, ist das Verhalten wie im worst case von Quicksort. Auf den Rest des Datensatzes wird dann Heapsort angewendet. Heapsort verläuft unabhängig von der Permutation der Daten im Array immer mit dem selben Aufwand $\mathcal{O}(n \cdot log(n))$. Sei r die maximale Rekursionstiefe und n die Länge des Feldes:

$$\begin{aligned}\mathcal{O}(n) &= \sum_{i=0}^{r}(n-i) + (n-r)\cdot log(n-r) \\ &= \sum_{i=0}^{r} n - \sum_{i=0}^{r} i + (n-r)\cdot log(n-r) \\ &= r\cdot n - \frac{r\cdot(r+1)}{2} + (n-r)\cdot log(n-r)\end{aligned}$$

Die Betrachtung des Terms wird dadurch erschwert, dass r kein konstanter Faktor ist, sondern von n abhängt. Musser schlägt für $r = 2 \cdot log_2(n)$ vor. Den Term weiter zu vereinfachen ist aufwändig, jedoch lässt sich das qualitative Verhalten über die Betrachtung der drei Summanden analysieren. $2 \cdot n \cdot log_2(n)$ und $(n-r)\cdot log(n-r)$ verhalten sich beide einzeln wie $\mathcal{O}(n \cdot log(n))$. Die Addition verändert dies nicht und führt lediglich zu einem konstanten Faktor k vor dem $n \cdot log(n)$, der in der Groß-O Notation jedoch wegfällt. Der dritte Faktor $\frac{log_2(n)\cdot(log_2(n)+1)}{2}$ verhält sich wie $\mathcal{O}(log_2(n)^2)$. Er wirkt sich durch das negative Vorzeichen zwar positiv auf das Verhalten des gesamten Terms aus, ist jedoch zu klein, um das Gesamtverhalten in der Groß-O Notation zu verändern.

Abbildung 3 zeigt dies präzise. Im Bild sind drei Funktionen, die Aufwandsgleichung für den worst case von Introsort $g(x)$, $h(x) = x \cdot log(x)$ und $i(x) = x^2$. Dabei wird deutlich, dass Introsort, wie oben abgeleitet, im schlechtesten Fall weit davon entfernt ist quadratisch anzuwachsen. Jedoch ist auch zu einem echten $x \cdot log(x)$ ein starker Überhang sichtbar.

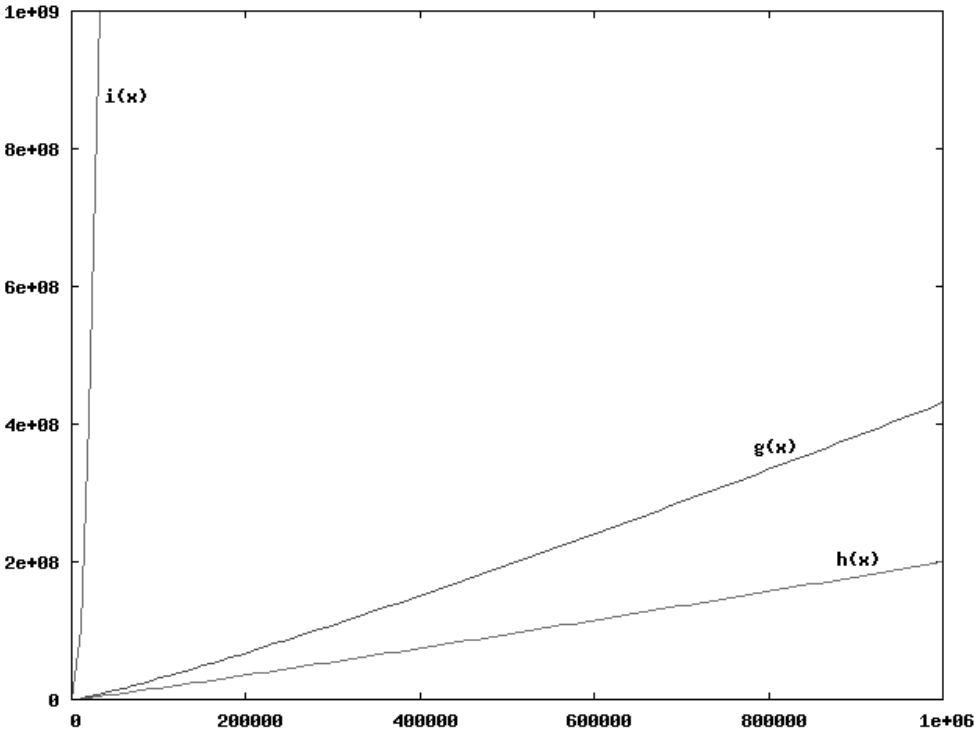

Abbildung 3: Zeiteffizienz von Introsort im worst case

5 Benchmark

Im folgenden Kapitel werden Ergebnisse der praktischen Analyse vorgestellt, die die theoretischen Effizienzbetrachtungen der vorigen Kapitel bestätigen sollen. Des Weiteren wird Introsort für verschiedene Problemstellungen mit anderen Sortieralgorithmen verglichen. Die am naheliegendste Möglichkeit ist es, die Zeit des Sortiervorgangs zu messen. Dies bringt jedoch den Nachteil mit sich, dass die Ressourcenzuteilung im unterliegenden Betriebssystem eine kaum abschätzbare Verzerrung des Messergebnis darstellt und so können damit nach [Wagenknecht 2003] nur grobe Ergebnisse erzielt werden. Besser ist es die Messungen von den technischen Eigenschaften der verwendeten Maschinen abzukoppeln und die elementaren Rechenschritte, die Grundoperationen, zu betrachten. Diese lassen sich ebenfalls, mit Hilfe von Zählern, messen. Im Fall der Sortieralgorithmen, die verglichen werden, sind die beiden Grundoperationen Vergleiche und Zuweisungen. Komplexe Rechnungen werden nicht vollzogen und die Verwendung von Kontrollstrukturen, wie verschiedenen Schleifen, ist im Allgemeinen so schnell, dass sie vernachlässigt werden können. In der Diskussion der Ergebnisse

ist, auch wenn Basisoperationen gezählt und nicht die Zeit gemessen wird, beim Vergleich der Algorithmen von Geschwindigkeit die Rede. Wenn ein Algorithmus für die selbe Aufgabe weniger Operationen wie ein zweiter Algorithmus benötigt, ist er unter der Annahme, dass jede gleichartige Operation gleich lange dauert, schneller. In den folgenden Abschnitten werden beide Testarten durchgeführt.

5.1 Basisoperationenbenchmark

Für den Test mit den Basisoperationen werden vier Algorithmen betrachtet. “Introsort“ in der besprochenen Implementierung, das heißt, mit Wahl des Pivotelement nach Median-of-3-Strategie, minimaler Teildateilänge von 16 und Rekursionstiefenbegrenzung durch $2 \cdot log_2(n)$. Als Stopper wird Heapsort angwandt. Der zweite Kandidat ist “Quicksort (opt.)“ ebenfalls mit Wahl des Pivotelement nach Median-of-3-Strategie und minimaler Teildateilänge von 16. “Quicksort“ ist eine simple Implementierung vom gleichnamigen Algorithmus, das jedoch auch ein Median-of-3-Pivotelement wählt. Der vierte Algorithmus ist “Heapsort“ nach der selben Implementierung, wie sie Introsort zum Abfangen des worst case benutzt. Sortiert werden jeweils Arrays mit zufälligen Zahlen, vorsortierten Zahlen und in Abschnitt 3.2 diskutierte Median-of-3-Killersequenzen in verschiedenen Längen. Die Auswertung der Ergebnisse bezieht sich im Folgenden auf die Tabellen im Anhang “D2 Tabellen des Basisoperationenbenchmark“.

Tabelle 3 im Anhang zeigt die benötigten Basisoperationen zum sortieren eines Arrays mit zufällig angeordneten Elementen. Dabei lassen sich folgende Beobachtungen machen: Introsort und das optimiertere Quicksort sind 10% bis 15% schneller als das nicht optimierte. Der Mehraufwand von Introsort gegenüber dem optimierten Quicksort zur Überwachung der Rekursionstiefe ist im kleinen Prozentbereich. Heapsort ist am langsamsten und braucht im Schnitt doppelt so lange wie die Quicksorts und Introsort.

Der nächste Test mit einem vorsortierten Datensatz (siehe Tabelle 4) zeigt für die Zusammenhänge zwischen Introsort und Quicksort die selben Ergebnisse wie der vorhergegangene. Tatsächlich interssant macht ihn der Vergleich Ergebnisse der einzelnen Algorithmen mit den Ergebnissen aus dem Test mit den zufälligen Zahlenfolgen. Die Algorithmen mit Median-of-3-Strategie verlaufen für das vorsortierte Array in ihrem best case und sind über 40% schneller

als für zufällige Permutationen. Heapsort hingegen benötigt sogar leicht mehr Operationen für die vorsortierte Folge.

Der letzte und wichtigste Test in dem Median-of-3-Killersequenzen sortiert werden (siehe Tabelle 5) macht deutlich wie Introsort den worst case von Quicksort verbessert. Schon für sehr kurze Sequenzen mit 1000 Elementen ist Introsort doppelt so schnell wie das optimierte Quicksort. Je größer die Felder werden, umso größer wird diese Faktor: bei 4000 Elementen ist Introsort 5.8 mal so schnell, bei 16000 Elementen 19 mal so schnell, bei 64000 Elementen schon 65.7 mal schneller, et cetera. Die größte gemachte Messung mit 256000 Elementen bringt den verwendeten Datentyp `double` in Java[3], der zum Zählen der Operationen des optimierten Quicksorts verwendet wird, bereits in die unpräzise Exponentialschreibweise. Interessant ist auch ein Vergleich zwischen den beiden Quicksorts, die Begrenzung der minimalen Teildateilänge wirkt im worst case stärker optimierend, als für die anderen Fälle und bringt Verringerung der benötigten Basisoperationen von beinahe 50%. Heapsort ist in diesem Fall der schnellste Sortieralgorithmus, Introsort ist stets deutlich langsamer, da es bis zum Wechsel auf Heapsort sehr ineffizient läuft. So benötigt Heapsort nur 63% bis 64% der gezählten Operationen von Introsort.

5.2 Zeitbenchmark

Der Zeitbenchmark betrachtet nur zufällig sortierte Felder verschiedener Längen. Dafür wird eine größere Gruppe von Algorithmen betrachtet: verschiedene Quicksortvarianten, Qsort aus der FreeBSD Bibliothek, Psort von Plan9, Dsort nach Dijsktra, Heapsort und Introsort. Als Plattform dient ein Rechner mit AMD Athlon 64 3200+, 2GB Arbeitsspeicher, Ubuntu Linux 6.10 (Kernelversion 2.6.17-11-generic) und Java 1.5. Um die Probleme, die im Zuge der unbeeinflußbaren Ressourcenverteilung durch das Betriebssystem entstehen, auszugleichen, wurde jeder Algorithmus für jede Arraylänge mit der identischen zufälligen Verteilung 100 mal gemessen und eine durchschnittliche Zeit zum Sortieren ermittelt. Die Ergebnisse befinden sich in Tabelle 2 in Anhang “D1 Tabellen des Zeitbenchmark“. Die einzige Quicksortvariante im Test, die den Mehraufwand von Introsort deutlich macht ist Qsort von FreeBSD nach [Bentley 1993]. Qsort ist circa 5% - 8% schneller. Jedoch ist der worst case auch hier $\mathcal{O}(n^2)$

[3] Wertebereich von $-1.79769313486231570 \cdot 10^{308}$ bis $1.79769313486231570 \cdot 10^{308}$

[QSO 1993]. Ein Test gegen den worst case von Qsort würde eine andere Verteilung als die vorgestellten Median-of-3-Killersequenzen benötigen, da die Wahl des Pivotelements bei Qsort noch komplexer ist. Das Ergebnis wäre jedoch ähnlich wie im vorherigen Kapitel bei der Untersuchung der Basisoperationen.

6 Schluß

Zusammenfassend lässt sich feststellen, dass Introsort dem gesteckten Ziel gerecht wird. Wie gezeigt wurde schwächt es den schlechtesten Fall von Quicksort stark ab. Die Entscheidung für einen bestimmten Sortieralgorithmus hängt jedoch immer von den Randbedingungen ab. Introsort ist ein allgemein einsetzbarer Sortieralgorithmus, der einen starken Kompromiss zwischen Quicksort und Heapsort bietet. Wenn es darum geht, eine quadratische Laufzeit auszuschließen und keine Information über die Art der Vorsortierung der Datensätze vorliegt, steht als Alternative zu Quicksort Heapsort zur Verfügung. Jedoch ist Heapsort im Durchschnitt etliche Male langsamer als die durchschnittliche Laufzeit von Quicksort. Introsort bietet einen besten und durschnittlichen Aufwand, der sehr nahe an Quicksort liegt und einen worst case der knapp über Heapsort liegt. Das Eintreten des worst case ist jedoch, genau bei Quicksort, sehr unwahrscheinlich. Liegen ausreichende Informationen über die Sortierung der Eingangsdaten vor, so ist Introsort uninteressant, da der schnellere Quicksortalgorithmus mit einer Strategie zur Wahl des Pivotelements implementiert werden kann, die den Eintritt des worst case ausschließt. Heapsort ist im Vergleich zu den anderen Alternativen nur in einem Szenario interessant, in dem für Quicksort und Introsort ständig mit dem worst case gerechnet werden muss. Für sehr stark vorbestimmte Datensätze sind in der Regel besser angepasste Algorithmen vorzuziehen, die sogar mit linearem Aufwand laufen können.

Anhang A: Mathematik

Dieser Anhang stellt dem Leser die wichtigsten mathematischen Zusammenhänge kurz dar, die zur Nachvollziehung der Rechnungen notwendig sind.

A1 Basisumrechnung bei Logarithmen

Um den Logarithmus einer Zahl r zur Basis b mit dem Logarithmus zu einer beliebigen Basis a zu bestimmen gilt nach [Papula 1990]:

$$log_b(r) = \frac{log_a(r)}{log_a(b)}$$

A2 Beweis der Gauss'schen Summenformel

Zu beweisen ist, dass gilt:

$$\sum_{i=1}^{n} = \frac{n \cdot (n+1)}{2}$$

Führe den Beweis mit vollständiger Induktion.

Anfang: $n = 1$: $1 = \frac{1 \cdot (1+1)}{2}$ Ist korrekt.

Annahme: Die Formel gelte für jede Zahl $n = k \in \mathbb{N}^+$.

Schritt: Zeige, dass die Formel auch für jede Zahl $n = k + 1$ gilt.

$$\begin{aligned} 1 + 2 + 3 + \cdots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\ &= (k+1) \cdot \left(\frac{k}{2} + 1\right) \\ &= (k+1) \cdot \frac{k+2}{2} \\ &= \frac{(k+1) \cdot (k+2)}{2} \\ &= \frac{(k+1) \cdot ((k+1)+1)}{2} \end{aligned}$$

Was zu beweisen war.

Anhang B: Weitere Sortieralgorithmen

Dieser Teil des Anhangs stellt die Sortieralgorithmen Insertionsort und Heapsort vor. Ihre exakte Funktionsweise ist für das Verständnis von Introsort nicht unbedingt erforderlich, jedoch werden Kenntnisse über ihr qualitatives Verhalten und ihre Effizienz in die Erklärung von Quicksort einbezogen. Diese beziehen sich auf die folgenden beiden Abschnitte.

B1 Insertionsort

Insertionsort (Sortieren durch Einfügen) ist ein elementares Sortierverfahren ohne namentlich bekannten Erfinder.

Funktionsweise

Der Algorithmus entnimmt einer unsortierten Eingabemenge ein beliebiges Element und fügt es an richtiger Stelle in die anfangs leere Ausgabemenge ein. Da die Ausgabemenge stets sortiert ist, kann die richtige Position etwa durch binäre Suche relativ einfach bestimmt werden. Die eigentlich rechenintensive Operation ist jedoch das Einfügen, denn in einem Datenfeld müssen von der Einfügeposition an alle folgenden Elemente um ein Feld weiter kopiert werden, um dem Element, das eingefügt wird, Platz zu schaffen. Abbildung 4 zeigt den Algorithmus als Struktogramm (ohne binäre Suche).

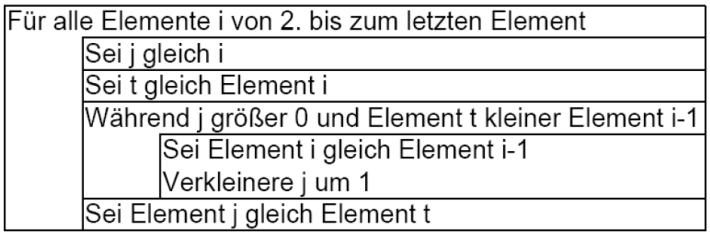

Abbildung 4: Struktogramm von Insertionsort

Effizienz

Im oben beschriebenen Algorithmus wird stets von hinten her die Position des einzufügenden Elementes gesucht. Im worst case wird der Platz für das einzufügende Element erst am Anfang der sortierten Ausgabemenge gefunden. Das heißt in der while-Schleife werden Folgen

der Länge $1, 2, 3, ..., n-1 = \frac{n \cdot (n-1)}{2}$ durchsucht. Deswegen ist das Verhalten quadratisch mit $\Omega(n^2)$. Dieser Fall tritt ein, wenn die Folge zu Anfang in absteigender Reihenfolge sortiert ist.

Umgekehrt ist das Verhalten optimal wenn die Eingabemenge aufsteigend vorsortiert ist. Dann muss das einzufügende Element an die letzte Position der Ausgabemenge geschrieben werden. Hierzu sind keine Vertauschungen notwendig und der Algorithmus läuft mit dem Aufwand $\mathcal{O}(n)$. Insertionsort zeigt auch ein gutes Verhalten, wenn einzelne Elemente in ein bereits sortiertes Feld eingegliedert werden sollen, die Feldgröße sehr klein ist oder ein langes Feld, wie im Fall von Introsort, bereits fast vollständig sortiert ist.

B2 Heapsort

Die Idee von Heapsort geht auf den Artikel [Williams 1964] zurück. Die erste komplette und effiziente Implementierung, die fast mit dem heutigen Heapsort identisch ist, wurde in [Floyd 1964] geleistet.

Funktionsweise

Heapsort verwendet zum Sortieren eine besondere Datenstruktur, die als Heap[4] bezeichnet wird. Die Definition der Datenstruktur basiert auf der eines fast vollständigen binären Baumes. Ein binärer Baum heißt fast vollständig, wenn alle Schichten, außer möglicherweise der letzten, voll besetzt sind.

Definition: Ein Knoten in einem binären Baum hat Heapeigenschaft, wenn beide Nachfolger kleiner sind. Ein binärer Baum ist ein Heap, wenn alle Knoten diese Eigenschaft erfüllen. Ein binärer Baum heißt Semi-Heap, wenn alle Knoten bis auf die Wurzel Heapeigenschaft haben.

Um den Algorithmus Heapsort zu erläutern, ist zunächst die Einführung der Methode “versickern“ notwending.

Definition: Die Wurzel eines (Teil-)Baumes wird versickert, wenn sie so lange mit dem größeren ihrer Nachfolger vertauscht wird, bis sie Heap-Eigenschaft hat.

[4]Die Bezeichnung Heap wird auch für den Speicherplatz verwendet, den ein Computer für dynamisch erzeugte Variablen benutzt. Die beiden Begriffe haben jedoch inhaltlich nichts miteinander zu tun.

Heapsort macht zunächst aus einem binären Baum mit zufälligen Zahlen einen Heap. Dies wird dadurch erreicht, dass alle Knoten, die keine Blätter sind, auf allen Ebenen einmal von ihrer Position aus versickert werden. Das Ergebnis ist ein Heap. Wenn die zu sortierenden Daten als Heap arrangiert sind, lässt sich das größte Element an der Wurzel entnehmen. Um das nächste Element zu entnehmen, müssen die verbleibenden Daten erneut zu einem Heap umgeordnet werden. Dies geschieht, indem zunächst ein beliebiges Blatt des Baumes von seiner Position an die Wurzel gesetzt wird. Das Ergebnis ist ein Semi-Heap. Um diesen Semi-Heap wieder zu einem Heap zu machen, wird die neue Wurzel versickert. Dies wird so lange wiederholt bis die komplette Baumstruktur abgebaut ist. In einer tatsächlichen Implementierung wird ein Array nur gedanklich auf die Baumstruktur abgebildet.

Effizienz

Ein vollständiger binärer (Teil-)Baum mit n Knoten hat die Tiefe $log_2(n)$. Somit werden für das Versickern eines Elements von der Wurzel $log_2(n)$ Schritte benötigt. Beim Umwandeln des binären Baums in einen Heap, werden, wie in unten stehender Summenformel ersichtlich, folgende Schritte benötigt. Auf jeder Ebene i von $log_2(n) - 1$ bis 0 befinden sich in einem binären Baum genau 2^i Elemente. Jedes dieser Elemente muss versickert werden, die maximale Schrittzahl dafür hängt vom Abstand i von der Wurzel ab. In der Wurzel sind es wie vorher festgestellt maximal $log_2(n)$ Schritte, also sind es für die darunterliegenden Ebenen höchstens $log_2(n) - i$ Schritte. Aus der Summenformel für die Schrittzahl $\mathcal{S}_1$ ergibt sich eine weiterführende Betrachtung, werden konkrete Werte eingesetzt:

$$\begin{aligned}
\mathcal{S}_1 &= \sum_{i=0}^{log_2(n)-1} 2^i \cdot (log_2(n) - i) = \frac{n}{2} + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + ... + 2 \cdot (log_2(n) - 1) + log_2(n) \\
2 \cdot \mathcal{S}_1 &= n + \frac{n}{2} \cdot 2 + \frac{n}{4} \cdot 3 + ... + 2 \cdot log_2(n) \\
2 \cdot \mathcal{S}_1 - \mathcal{S}_1 &= n + \frac{n}{2} + \frac{n}{4} + ... + 2 - log_2(n) \\
&= \sum_{i=1}^{log_2(n)} 2^i - log_2(n) = n + (n-1) - 1 - log_2(n)
\end{aligned}$$

Somit gilt $\mathcal{S}_1 < 2 \cdot n$, womit der Aufwand für die Umwandlung von binären Bäume in Heaps nur linearen Aufwand, also $\mathcal{O}(n)$ bedeutet. Um den Baum abzubauen ist der Aufwand $\mathcal{S}_2$. Es muss n-mal ein Blatt von der Wurzel her versickert werden, was maximal $log_2(n)$ Schritte

benötigt. Tatsächlich ist der Aufwand viel kleiner, da der Baum in diesem Vorgang stets kleiner wird. Präzise ist es:

$$\mathcal{S}_2 = \sum_{i=1}^{n} log(i)$$

Jedoch reicht zur Betrachtung die Aussage $\mathcal{S}_2 < n \cdot log_2(n)$. Somit ist der Gesamtaufwand:

$$\begin{aligned} \mathcal{S}_{Gesamt} &= \mathcal{S}_1 + \mathcal{S}_2 \\ &= \mathcal{O}(n) + \mathcal{O}(n \cdot log(n)) \\ &= \mathcal{O}((n+1) \cdot log(n)) = \mathcal{O}(n \cdot log(n)) \end{aligned}$$

Anhang C: Quelltext

Dieser Anhang enthält den Quelltext auf den in der Arbeit verwiesen wird.

C1 Algorithmische Erzeugung von Median-of-3 Killersequenzen

Diese Methode erzeugt Felder vom Typ `Sortierbar` mit einer spezifierbaren Länge `length`, die mit den Integerwerten von 1 bis `length` bestückt sind.

```
1  private static Sortierbar[] createMedianOf3KillerArray(int length)
2  {
3          if ((length%2)==1)
4                  return null;
5          int k=length/2;
6          Sortierbar[] a = new Sortierbar[length];
7          for (int i=1; i<=k; i++)
8          {
9                  if((i\%2) == 1)
10                 {
11                         a[i-1] = new SortierbareInteger(i);
12                         a[i] = new SortierbareInteger(k+i);
13                 }
14                 a[k+i-1] = new SortierbareInteger(2*i);
15         }
16         return a;
17 }
```

C2 Die Klasse Introsort

Der vollständige Quelltext der Klasse Introsort:

```
1  public class Introsort implements Sortierer
2  {
3          /*
4           * Class Variables
5           */
6      private static Sortierbar[] a;
7      private static int size_threshold = 16;
8
9      /*
10      * Public interface
11      */
12     public void sortiere(Sortierbar[] a0)
13     {
14         a=a0;
15         introsort_loop(0, a.length, 2*floor_lg(a.length));
16         insertionsort(0, a.length);
17     }
18
19     public void sortiere(Sortierbar[] a0, int begin, int end)
20     {
21         if (begin<end)
22         {
23                 a=a0;
24                 introsort_loop(begin, end, floor_lg(end-begin));
25                 insertionsort(begin, end);
26         }
27     }
28
29     /*
30      * Quicksort algorithm modified for Introsort
31      */
32     private static void introsort_loop (int lo, int hi, int depth_limit)
33     {
34         while (hi-lo > size_threshold)
35         {
36                 if (depth_limit == 0)
```

```
37                {
38                        heapsort(lo, hi);
39                        return;
40                }
41                depth_limit--;
42                int p=partition(lo, hi,
43                                medianof3(lo, lo+(lo+hi)/2+1, hi-1));
44                introsort_loop(p, hi, depth_limit);
45                hi=p;
46        }
47    }

49    private static int partition(int lo, int hi, Sortierbar x)
50    {
51        int i=lo, j=hi;
52        while (true)
53        {
54                while (a[i].kleiner(x)) i++;
55                j--;
56                while (x.kleiner(a[j])) j--;
57                if(!(i<j))
58                        return i;
59                exchange(i,j);
60                i++;
61        }
62    }

64        private static Sortierbar medianof3(int lo, int mid, int hi)
65        {
66                if (a[mid].kleiner(a[lo]))
67                {
68                        if (a[hi].kleiner(a[mid]))
69                                return a[mid];
70                        else
71                        {
72                                if (a[hi].kleiner(a[lo]))
73                                        return a[hi];
74                                else
```

```
75                                      return a[lo];
76                      }
77              }
78              else
79              {
80                      if (a[hi].kleiner(a[mid]))
81                      {
82                              if (a[hi].kleiner(a[lo]))
83                                      return a[lo];
84                              else
85                                      return a[hi];
86                      }
87                      else
88                              return a[mid];
89              }
90      }

92      /*
93       * Heapsort algorithm
94       */
95      private static void heapsort(int lo, int hi)
96      {
97              int n = hi-lo;
98              for (int i=n/2; i>=1; i--)
99              {
100                downheap(i,n,lo);
101             }
102             for (int i=n; i>1; i--)
103             {
104                exchange(lo,lo+i-1);
105                downheap(1,i-1,lo);
106             }
107     }

109     private static void downheap(int i, int n, int lo)
110     {
111        Sortierbar d = a[lo+i-1];
112        int child;
```

```
113             while (i<=n/2)
114             {
115                     child = 2*i;
116                     if (child<n && a[lo+child-1].kleiner(a[lo+child]))
117                     {
118                             child++;
119                     }
120                     if (!d.kleiner(a[lo+child-1])) break;
121                     a[lo+i-1] = a[lo+child-1];
122                     i = child;
123             }
124             a[lo+i-1] = d;
125         }

127         /*
128          * Insertion sort algorithm
129          */
130         private static void insertionsort(int lo, int hi)
131         {
132                 int i,j;
133                 Sortierbar t;
134                 for (i=lo; i<hi; i++)
135                 {
136                         j=i;
137                         t = a[i];
138                         while(j!=lo && t.kleiner(a[j-1]))
139                         {
140                                 a[j] = a[j-1];
141                                 j--;
142                         }
143                         a[j] = t;
144                 }
145         }

147     /*
148      * Common methods for all algorithms
149      */
150     private static void exchange(int i, int j)
```

```
151     {
152         Sortierbar t=a[i];
153         a[i]=a[j];
154         a[j]=t;
155     }

157         private int floor_lg(int a)
158         {
159                 return (int)(Math.floor(Math.log(a)/Math.log(2)));
160         }
161 }
```

C3 Implementierte Interfaces

Die erarbeitete Realisierung des Introsortalgorithmus implementiert das Interface Sortierer:

```
1  public interface Sortierer
2  {
3          void sortiere (Sortierbar[] f, int von, int bis);
4      /**
5        *       Sortiert das Feld f zwischen von und bis,
6        *       wobei von eingescjlossen ist und bis nicht
7        *       eingeschlossen ist.
8        *       Es wird aufsteigend sortiert.
9        **/

11         void sortiere (Sortierbar[] f);
12 }
```

Sortiert werden Arrays, die das Interface Sortierbar implementieren:

```
1  public interface Sortierbar
2  {
3          boolean kleiner (Sortierbar ob);
4  }
```

Anhang D: Benchmarkergebnisse

Im Folgenden werden die Ergebnisse aus den Benchmarks dargestellt, die in Kapitel 5 interpretiert werden. Die Angabe Länge entspricht der Anzahl der Elemente im zu sortierenden Array und durch den Faktor 1000 dividiert.

D1 Tabellen des Zeitbenchmark

Alle Zeiten in Millisekunden als Durchschnitt von 100 Messungen.

Länge (in TSD)	1	4	16	64	256	1024
Quicksort (Lehrbuch)	0.83	3.41	7.73	43.02	228.39	1155.47
QuicksortOhneRekursionsProbleme	0.98	1.73	7.77	42.54	232.5	1153.4
QuicksortOhneRekursionsProblemeHelbig	0.95	1.73	7.81	43.6	226.59	1187.25
Qsort (FreeBSD)	0.93	1.57	6.54	34.98	179.52	919.5
Psort (Plan9)	1.05	1.88	8.69	44.43	243.55	1213.03
Dsort (Dijkstra)	1.17	2.18	9.07	48.97	240.04	1191.03
QuicksortSchwinn	1.17	2.13	11.53	86.73	905.84	14226.31
Introsort	0.95	1.67	7.46	40.16	210.49	1055.4
Heapsort	1.12	2.35	12.23	79.68	470.25	2328.95

Tabelle 2: Zeitaufwand zum Sortieren eines zufälligen Arrays

D2 Tabellen des Basisoperationenbenchmark

Die Anzahl der Operationen ist in allen Tabellen auf die erste Nachkommastelle gerundet.

Länge (in TSD)	Algorithmus	Zuweisungen	Vergleiche	Gesamt
1	Introsort	21.6	17.7	39.3
	Quicksort (opt.)	21.5	17.7	39.2
	Quicksort	22.4	23.2	45.5
	Heapsort	36.1	36.3	72.4
4	Introsort	102.7	83.1	185.8
	Quicksort (opt.)	102.2	83.1	185.4
	Quicksort	105.4	106.2	211.6
	Heapsort	172.5	177.4	349.9
16	Introsort	470.9	377.8	848.7
	Quicksort (opt.)	469.2	377.8	847.0
	Quicksort	495.7	493.8	989.5
	Heapsort	803.3	838.3	1641.6
64	Introsort	2141.2	1709.3	3850.5
	Quicksort (opt.)	2134.6	1709.3	3843.9
	Quicksort	2198.8	2155.5	4354.3
	Heapsort	3658.6	3863.1	7521.7
256	Introsort	9629.7	7666.5	17296.3
	Quicksort (opt.)	9603.1	7666.5	17269.6
	Quicksort	9790.2	9499.1	19289.4
	Heapsort	16423.7	17498.0	33921.7
1024	Introsort	42482.9	33676.1	76159.0
	Quicksort (opt.)	42376.6	33676.1	76052.7
	Quicksort	43347.4	41711.0	85058.4
	Heapsort	72873.9	78192.9	151066.7

Tabelle 3: Basisoperationen zum Sortieren eines zufälligen Arrays

Länge (in TSD)	Algorithmus	Zuweisungen	Vergleiche	Gesamt
1	Introsort	10.6	9.8	20.4
	Quicksort (opt.)	10.5	9.8	20.3
	Quicksort	13.5	14.1	27.6
	Heapsort	38.2	38.1	76.3
4	Introsort	50.4	47.1	97.5
	Quicksort (opt.)	50.1	47.1	97.2
	Quicksort	61.9	64.6	126.5
	Heapsort	181.4	185.0	366.5
16	Introsort	233.8	220.3	454.1
	Quicksort (opt.)	232.6	220.3	452.8
	Quicksort	279.6	290.2	569.8
	Heapsort	837.8	868.8	1706.5
64	Introsort	1063.2	1009.1	2072.4
	Quicksort (opt.)	1058.3	1009.1	2067.4
	Quicksort	1246.2	1288.9	2535.1
	Heapsort	3797.4	3986.2	7783.7
256	Introsort	4765.0	4548.6	9313.6
	Quicksort (opt.)	4745.2	4548.6	9293.8
	Quicksort	5496.8	5667.5	11164.4
	Heapsort	17003.1	18005.9	35009.0
1024	Introsort	21108.0	20242.4	41350.5
	Quicksort (opt.)	21028.9	20242.4	41271.3
	Quicksort	24035.3	24718.0	48753.4
	Heapsort	75269.3	80325.5	155594.8

Tabelle 4: Basisoperationen zum Sortieren eines vorsortierten Arrays

Länge (in TSD)	Algorithmus	Zuweisungen	Vergleiche	Gesamt
1	Introsort	54.6	53.6	108.1
	Quicksort (opt.)	110.5	108.3	218.8
	Quicksort	206.5	208.8	415.3
	Heapsort	35.9	35.8	71.7
4	Introsort	270.6	270.3	540.9
	Quicksort (opt.)	1585.3	1574.2	3159.5
	Quicksort	3095.5	3103.6	6199.0
	Heapsort	171.1	174.8	346.0
16	Introsort	1270.7	1285.8	2556.5
	Quicksort (opt.)	24419.0	24366.9	48785.9
	Quicksort	48473.5	48502.3	96975.8
	Heapsort	796.8	827.9	1624.7
64	Introsort	5806.9	5931.1	11737.9
	Quicksort (opt.)	386043.4	385804.8	771848.2
	Quicksort	770320.7	770420.7	1540741.4
	Heapsort	3636.1	3824.5	7460.6
256	Introsort	26052.9	26799.7	52852.6
	Quicksort (opt.)	6153857.4	6152777.5	ca. $1.23 * 10^7$
	Quicksort	-	-	-
	Heapsort	16322.3	17329.7	33652.0

Tabelle 5: Basisoperationen zum Sortieren eines Arrays mit Median-of-3-Killersequenz

Quellen- und Literaturverzeichnis

[QSO 1993] (1993). *Qsort 3 FreeBSD manpage*. `http://www.freebsd.org/cgi/man.cgi?query=qsort&sektion=3&apropos=0&manpath=freebsd`, eingesehen am 19.02.2007.

[Bentley 1993] Bentley, Jon L. (1993). *Engineering a Sort Function*. Software - Practice and Experience. Volume 23, Issue 11, S. 1249-1265.

[Floyd 1964] Floyd, Robert W. (1964). *Algorithm 245: Treesort 3*. Communications of the ACM. Volume 7, Issue 12, S. 701.

[Hoare 1961] Hoare, C. A. R. (1961). *Algorithm 64: Quicksort*. Communications of the ACM. Volume 4, Issue 7, S. 321.

[Musser 1997] Musser, David R. (1997). *Introspective sorting and selection algorithms*. Rensselaer Polytechnic Institute, Troy, NY 12180, `musser@cs.rpi.edu`.

[Papula 1990] Papula, Lothar (1990). *Mathematische Formelsammlung für Ingenieure und Naturwissenschaftler*. Vieweg & Sohn. 3. Auflage, S. 9.

[Sedgewick 1988] Sedgewick, Robert (1988). *Algorithmen*. Addison-Wesley. 2. Auflage.

[Wagenknecht 2003] Wagenknecht, Christian (2003). *Algorithmen und Komplexität*. Fachbuchverlag Leipzig.

[Williams 1964] Williams, J. W. J. (1964). *Algorithm 232: Heapsort*. Communications of the ACM. Volume 7, Issue 6, S. 347-348.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass:

1. ich meine Studienarbeit eigenständig und ohne fremde Hilfe angefertigt habe.
2. ich die Übernahme wörtlicher Zitate aus der Literatur/Internet sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.
3. ich meine Studienarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

(Ort, Datum) (Unterschrift)