Effizienz von Sortierverfahren

In der Praxis ist man meist an Programmen - also Implementierungen von Algorithmen in einer Programmiersprache - interessiert, die von Rechnern real ausgeführt werden können. Dabei spielt der Ressourcenverbrauch (Laufzeitverhalten und Speicherplatzbelegung) der Programme eine zentrale Rolle. Erwünscht sind solche Programme, die mit möglichst wenigen Ressourcen auskommen. Im Folgenden wird der Fokus auf die Ressource Rechenzeit gesetzt, da die Betrachtung des Speicherplatzes für verschiedene Algorithmen keine großen Unterschiede deutlichen werden lässt.

Laufzeitmessungen liefern wichtige Informationen über die Einsetzbarkeit von Programmen. Programme, bei denen man sehr lange auf Ergebnisse warten muss, sind - je nach Anwendungsgebiet - oft nicht praxistauglich.

Ziel der folgenden Untersuchungen ist es, das Laufzeitverhalten von Sortieralgorithmen zu ermitteln und die Ergebnisse genauer zu analysieren.

Experimentell bestimmte Laufzeiten (Zeitmessungen) haben den Nachteil, dass sie vom benutzen Rechner und den Bedingungen auf dem Rechner (Prozesse, die dort laufen) abhängen. Solche Rechenzeiten sind also nur bedingt aussagekräftig. Erschwerend kommt manchmal hinzu, dass Rechenzeiten bei manchen Problemstellungen so groß werden, dass man sie bis heute noch nicht ermitteln konnte.

Neben experimentellen Methoden benutzt man in der Informatik daher auch mathematische Methoden zur Einschätzung des Laufzeitverhaltens. Man versucht dabei, das Laufzeitverhalten abzuschätzen und abstrahierend zu beschreiben.

Die Effizienzbetrachtungen erfolgen am Beispiel zweier Sortieralgorithmen:

* Sortieren durch Einfügen (Straight Insertion)
* Quicksort

Das Laufzeitverhalten von Sortieralgorithmen kann durch das Bestimmen der Anzahl der Vergleiche analysieren. Im Folgenden sollen die Vergleiche zwischen Listenelementen gezählt werden. Die Vergleiche, die zu Schleifensteuerungen notwendig sind, sollen hier nicht mitgezählt werden.

Aufgabe 1:

Bestimmen Sie die Anzahl der Vergleiche beim „Sortieren durch Einfügen“ auf linearen Listen.

Quellliste

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **25** | 17 | 32 | 56 | 26 | 19 | 8 | 66 | 29 | 6 | 20 | 27 |
|  | **17** | 32 | 56 | 26 | 19 | 8 | 66 | 29 | 6 | 20 | 27 |
|  |  | **32** | 56 | 26 | 19 | 8 | 66 | 29 | 6 | 20 | 27 |
|  |  |  | **56** | 26 | 19 | 8 | 66 | 29 | 6 | 20 | 27 |
|  |  |  |  | **26** | 19 | 8 | 66 | 29 | 6 | 20 | 27 |
|  |  |  |  |  | **19** | 8 | 66 | 29 | 6 | 20 | 27 |
|  |  |  |  |  |  | **8** | 66 | 29 | 6 | 20 | 27 |
|  |  |  |  |  |  |  | **66** | 29 | 6 | 20 | 27 |
|  |  |  |  |  |  |  |  | **29** | 6 | 20 | 27 |
|  |  |  |  |  |  |  |  |  | **6** | 20 | 27 |
|  |  |  |  |  |  |  |  |  |  | **20** | 27 |
|  |  |  |  |  |  |  |  |  |  |  | **27** |

Abbildung 1 – Entwicklung der unsortierten Liste

Zielliste

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| **25** |  |  |  |  |  |  |  |  |  |  |  | Vergleiche: |
| **17** | 25 |  |  |  |  |  |  |  |  |  |  | Vergleiche: |
| 17 | 25 | **32** |  |  |  |  |  |  |  |  |  | Vergleiche: |
| 17 | 25 | 32 | **56** |  |  |  |  |  |  |  |  | Vergleiche: |
| 17 | 25 | **26** | 32 | 56 |  |  |  |  |  |  |  | Vergleiche: |
| 17 | **19** | 25 | 26 | 32 | 56 |  |  |  |  |  |  | Vergleiche: |
| **8** | 17 | 19 | 25 | 26 | 32 | 56 |  |  |  |  |  | Vergleiche: |
| 8 | 17 | 19 | 25 | 26 | 32 | 56 | **66** |  |  |  |  | Vergleiche: |
| 8 | 17 | 19 | 25 | 26 | **29** | 32 | 56 | 66 |  |  |  | Vergleiche: |
| **6** | 8 | 17 | 19 | 25 | 26 | 29 | 32 | 56 | 66 |  |  | Vergleiche: |
| 6 | 8 | 17 | 19 | **20** | 25 | 26 | 29 | 32 | 56 | 66 |  | Vergleiche: |
| 6 | 8 | 17 | 19 | 20 | 25 | 26 | **27** | 29 | 32 | 56 | 66 | Vergleiche: |

Abbildung 2 - Entwicklung der sortierten Liste

Aufgabe 2

Bestimmen Sie die Anzahl der Vergleiche beim „Quicksort“ auf linearen Listen.

In der oberen Zeile werden die Listenelemente vor dem Aufteilen der Liste dargestellt. Hier sind Vergleiche zwischen Listenelementen erforderlich. In der zweiten Zeile werden die Teillisten nach dem Zusammenfügen dargestellt. Das Zusammenfügen kann ohne weitere Vergleiche durchgeführt werden.



Abbildung 3 – Darstellung der Teillisten beim Quicksort

Lösung:

Es sind nur Vergleiche zwischen Listenelemente gezählt worden, Vergleiche die bei der Schleifensteuerung notwendig sind, sind hier nicht gezählt worden.

Straight Insertion: 39 Vergleiche

Quicksort: 26 Vergleiche

Quelltexte der Sortieralgorithmen:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23** | **public void sortiere1() {**  **// Straight Insertion**  **List<Integer> hilfsliste = new List<Integer>();**  **while (!testListe.isEmpty()) {**  **testListe.toFirst();**  **Integer aktueller = testListe.getContent();**  **testListe.remove();**  **hilfsliste.toFirst();**  **boolean gefunden = false;**  **while (!gefunden && hilfsliste.hasAccess()) {**  **if (aktueller.intValue() < hilfsliste.getContent().intValue()) {**  **gefunden = true;**  **} else {**  **hilfsliste.next();**  **}**  **}**  **if (hilfsliste.hasAccess()) {**  **hilfsliste.insert(aktueller);**  **} else {**  **hilfsliste.append(aktueller);**  **}**  **}**  **testListe.concat(hilfsliste);**  **}** |

Abbildung 4 – Quelltext „Straight Insertion“

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28** | **public void sortiere2() {**  **// Quicksort**  **testListe = quicksort(testListe);**  **}**  **private List<Integer> quicksort(List<Integer> pListe) {**  **List<Integer> ergebnis = new List<Integer>();**  **if (!pListe.isEmpty()) {**  **pListe.toFirst();**  **Integer pivot = pListe.getContent();**  **pListe.remove();**  **List<Integer> kleiner = new List<Integer>();**  **List<Integer> groesser = new List<Integer>();**  **while (!pListe.isEmpty()) {**  **Integer aktuelles = pListe.getContent();**  **if (aktuelles.intValue() < pivot.intValue()) {**  **kleiner.append(aktuelles);**  **} else {**  **groesser.append(aktuelles);**  **}**  **pListe.remove();**  **}**  **kleiner = quicksort(kleiner);**  **groesser = quicksort(groesser);**  **ergebnis = kleiner;**  **ergebnis.append(pivot);**  **ergebnis.concat(groesser);**  **}**  **return ergebnis;**  **}** |

Abbildung 5 – Quelltext „Quicksort“

Aufgabe 3

Bestimmen Sie die Stellen im Quelltext, an denen die Zähler eingebaut werden können.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **public void erzeugeZufallsliste(int pAnzahl) {**  **Random generator = new Random(pAnzahl);**  **zaehler = 0;**  **testListe = new List<Integer>();**  **for (int i = 1; i <= pAnzahl; i++) {**  **int zahl = generator.nextInt(77777) + 1;**  **testListe.append(new Integer(zahl));**  **}**  **}** |

Abbildung 6 – Quelltext „Erzeugung der Zufallsbelegung“

Zur Erzeugung der Zufallszahlen wird ein Objekt der Klasse **Random** gewählt. Der Vorteil besteht darin, dass man durch Angabe eines Parameters beim Aufruf des Konstruktors die Folge der Zufallszahlen festlegen kann. Wählt man bei beiden Sortieralgorithmen den gleichen Parameter, so muss jeweils die gleiche Folge der zufälligen Werte sortiert werden.

Anzahl der Vergleiche bei zufälliger Reihenfolge in der Ausgangsliste

|  |  |  |
| --- | --- | --- |
| Listenelemente | Straight Insertion | Quicksort |
| 100 | 2447 | 588 |
| 200 | 10504 | 1516 |
| 300 | 21059 | 2946 |
| 400 | 41243 | 3633 |
| 500 | 63694 | 4722 |
| 600 | 90570 | 5660 |
| 700 | 119695 | 7310 |
| 800 | 166348 | 8294 |
| 900 | 202772 | 10362 |
| 1000 | 242009 | 10708 |

Abbildung 7 – Darstellung der Anzahl der Vergleiche (zufällige Reihenfolge)

Datenreihe 1: Straight Selektion, Datenreihe 2: Quicksort

Anzahl der Vergleiche bei aufsteigender Reihenfolge in der Ausgangsliste

|  |  |  |
| --- | --- | --- |
| Listenelemente | Straight Insertion | Quicksort |
| 100 | 4950 | 4950 |
| 200 | 19900 | 19900 |
| 300 | 44850 | 44850 |
| 400 | 79800 | 79800 |
| 500 | 124750 | 124750 |
| 600 | 179700 | 179700 |
| 700 | 244650 | 244650 |
| 800 | 319600 | 319600 |
| 900 | 404550 | 404550 |
| 1000 | 499500 | 499500 |

Abbildung 8 – Vergleiche bei aufsteigender Reihenfolge

Anzahl der Vergleiche bei absteigender Reihenfolge in der Ausgangsliste

|  |  |  |
| --- | --- | --- |
| Listenelemente | Straight Insertion | Quicksort |
| 100 | 99 | 4950 |
| 200 | 199 | 19900 |
| 300 | 299 | 44850 |
| 400 | 399 | 79800 |
| 500 | 499 | 124750 |
| 600 | 599 | 179700 |
| 700 | 699 | 244650 |
| 800 | 799 | 319600 |
| 900 | 899 | 404550 |
| 1000 | 999 | 499500 |

Abbildung 9 – Vergleiche bei absteigender Reihenfolge

Die Anzahl der Datensatzvergleiche bei der Ausführung eines Sortieralgorithmus hängt manchmal nicht nur von der Problemgröße n (d.h. der Anzahl der Listenelemente) ab, entscheidend ist auch, wie stark die Liste bereits vorsortiert ist.

Bei einer Kostenanalyse unterscheidet man daher oft die folgenden Fälle:

* best case (bester Fall): der Fall, in dem bei der Ausführung des Algorithmus die wenigsten Kosten anfallen
* worst case (schlechtester Fall): der Fall, in dem bei der Ausführung des Algorithmus die meisten Kosten anfallen
* average case (durchschnittlicher Fall): eine Mittelung der Kosten über alle Fälle

Für die Praxis am wichtigsten ist meist der durchschnittliche Fall. Dieser ist aber schwer zu ermitteln.

Aufgabe 4

Erklären Sie, wie die Ergebnisse in den Abbildungen 8 und 9 entstehen.

Analyseergebnisse für „Straight Selection“ und „Quicksort“

Datenreihe 1: Anzahl der Vergleiche bei zufälliger Eingabe (Straight Insertion)

Datenreihe 2:

Datenreihe 3: Anzahl der Vergleiche bei zufälliger Eingabe (Quicksort)

Datenreihe 4:

Abbildung 10 - Graphenvergleich

Sortieren durch Einfügen

Der schlechteste Fall: Es befinden sich alle zu sortierenden Objekte bereits sortiert in der Quellliste. Man muss jetzt nur nach und nach alle Objekte der Quellliste an das Ende der Zielliste anfügen. Um an der richtigen Position anfügen zu können, muss man alle in der Zielliste befindlichen Objekte prüfen, ob sie auch wirklich einen kleineren Schlüsselwert haben. Die Anzahl der Vergleiche wächst also mit jedem Objekt, das in die Zielliste eingefügt werden muss.

Die Anzahl der Vergleiche ist also

Der beste Fall: Es befinden sich alle zu sortierenden Objekte genau verkehrt herum in der Quellliste. Um ein Element in die Zielliste einzufügen, muss man nur das erste Element der Zielliste prüfen, ob es größer als das Element in der Quellliste ist. Stellt man dieses fest, muss das nächste Element vorne eingefügt werden muss. Es sind also insgesamt **n-1** Vergleiche erforderlich.

Der allgemeine Fall: Um das Objekt an der Stelle i in der Liste einzusortieren benötigt man **i-1 V**ergleiche.

Anzahl der Vergleiche ist proportional zu **n2**.

Quicksort

Man kann zeigen, dass die Laufzeit, z.B. wenn die Eingabefolge schon sortiert ist, auch das Sortierverfahren sehr langsam ist. Es sind genauso viele Vergleiche notwendig wie bei Straight Insertion im schlechtesten Fall.

Wenn das Pivotelement geeignet ist, erfolgt eine Aufspaltung eine Quellliste in zwei Listen, die ungefähr gleich lang sind. Ordnet man die Teillisten in einem Baum an, so erhält man einen Baum der Tiefe **log2(n)**, wobei n die Anzahl der Elemente ist. In jeder Ebene die Baumes sind maximal Vergleiche notwendig. Die Anzahl der vergleiche ist somit proportional zu **n\*log2(n)**. Das Verfahren ist somit gut, wenn die Anzahl der Elemente groß und die Verteilung zufällig ist.

Prinzipielle Frage

Wie schnell kann ein Sortieralgorithmus überhaupt sein?

Entscheidungsbaum zum Sortieren von drei Elementen:



Abbildung 11 - Entscheidungsbaum

Das Ausführen des Sortieralgorithmus für eine bestimmte Eingabe entspricht dem Verfolgen eines Weges von der Wurzel zu einem Blatt.

Ein Sortieralgorithmus muss alle Permutationen (alle möglichen Reihenfolgen) der Eingabeelemente erzeugen können.

Worst-Case: Die Länge des längsten Weges von der Wurzel zum Blatt.

Untere Schranke für die Laufzeit

Untere Schranke für die Höhe aller Entscheidungsbäume, in dem jede Permutation als erreichbares Blatt vorkommt.

Aufgabe 5:

Bestimmen Sie die Tiefe eines optimalen Entscheidungsbaumes zur Sortierung von Listen mit 5, 6, … , 10 Elementen.

Sortierverfahren, angewandt auf n Elemente, die auf den Operationen

* Vergleich von Elementen
* Vertauschen von Elementen

basieren, erreichen bestenfalls einen Aufwand von , wobei n die Anzahl der zu sortierenden Elemente ist.

Begründung:

Im Entscheidungsbaum gibt es **n!** Blätter (unterschiedliche Eingaben). Die Tiefe des Entscheidungsbaum zeigt, wie viele Vergleich im ungünstigen Fall notwendig sind.

Um eine Aussage über Listen beliebiger Länge treffen zu können, muss man wissen, wie viele Anordnungen jeweils möglich sind: Bei n Listenelementen gibt es **n! = 1\*2\*...\*n** verschiedene mögliche Anordnungen der Listenelemente.

Fasst man beide Ergebnisse zusammen, so erhält man folgende Aussage: Um eine Liste mit n Elementen zu sortieren, benötigt man einen Entscheidungsbaum, der eine Tiefe der Größenordnung **log2(n!)** hat.

Jeder vergleichsbasierte Sortieralgorithmus hat demnach ein worst-case-Verhalten, das durch die Funktion **K(n) = log2(n!)** abgeschätzt werden kann.

Es gilt die folgende Ungleichung: .

Mit dieser Abschätzung erhält man .

Hieraus ergibt sich, dass jeder vergleichsbasierte Sortieralgorithmus ein worst-case-Verhalten hat, das nach unten durch die Funktion abgeschätzt werden kann. Betrachtet man - wie üblich - nur das asymptotische Verhalten, so zeigt dies, dass das worst-case-Verhalten eines beliebigen Sortieralgorithmus von der Ordnung **n\*log2(n)** sein muss.

Damit ist eine Schranke zur Beschreibung der Komplexität des Sortierproblems gefunden. Die Komplexität des Sortierproblems ist von der Ordnung **n\*log2(n)** - sofern man ein vergleichsbasiertes Berechnungsmodell zu Grunde legt.