FACHARBEIT

Schuljahr 2001/2002

Darstellung und Vergleich von Sortieralgorithmen

Christian Vollmer

Fachbereich: Mathematik (Grundkurs)
Betreuender Lehrer: Herr Reimann
Verfasser: Christian Vollmer, An der Seilerei 45, 45219 Essen
Kontakt: christian.vollmer@epost.de

Inhaltsverzeichnis:

1. Einleitung: der Begriff Algorithmus

Der Begriff „Algorithmus“ wird schon in der Schulmathematik eingesetzt. Hier erlernt man z. B. den Gauß-Algorithmus.
In der Mathematik versteht man unter dem Begriff Algorithmus ein „Rechenverfahren, durch das man nach Durchführung endlich vieler gleichartiger Schritte zum Ergebnis gelangt“[1]. So folgt man beim Gauß-Algorithmus einem bestimmten Schema und gelangt in endlicher Zeit zu den gesuchten Unbekannten der Funktion.
In der Informatik sind Algorithmen als mathematische Berechnungsvorschriften entstanden. Allgemein formuliert werden Algorithmen zur Lösung eines Problems eingesetzt.[2]
Gunter Saake definiert treffend: „Ein Algorithmus ist eine präzise [...] endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.“[3]

2. Eigenschaften

Die von nun an folgenden Beschreibungen und Erklärungen beziehen sich ausschließlich auf Algorithmen im Sinne der Informatik.

a. Formale Eigenschaften von Algorithmen

In der Regel erfüllen Algorithmen zwei Eigenschaften:

- Ein Algorithmus bricht nach endlich vielen Schritten ab (terminierender Algorithmus). Voraussetzung ist die Eingabe erlaubter Parameter.
- Die Schrittfolge ist eindeutig vorgegeben (deterministischer Ablauf) und/oder bei gleicher Parametereingabe ist das Ergebnis immer identisch (determiniertes Ergebnis). Ein Algorithmus zur Ermittlung von Zufallszahlen bildet eine Ausnahme, hier sind immer andere Ergebnisse gefordert.

b. Untersuchung und Studium der Eigenschaften

- Gewöhnlich muss ein Algorithmus zunächst auf seine **Korrektheit** geprüft werden. Da in dieser Facharbeit schon bekannte und häufig untersuchte Algorithmen betrachtet werden, kann von dieser Prüfung abgesehen werden.
- Eine andere Eigenschaft ist die **Effizienz**. Sie ist auch im praktischen Einsatz von großer Bedeutung und deshalb verdient die Auswahl unter den (schon auf Korrektheit geprüften) Algorithmen nach dem Kriterium der Effizienz (in der Praxis) besondere Beachtung. (Die Analyse der Effizienz bildet daher auch den Hauptteil dieser Arbeit. Im Vordergrund steht dabei die mathematische Herleitung einer allgemeinen Formel zur Ermittlung der Geschwindigkeit.)

---

[1] Neues großes Lexikon in Farbe, 1995, S. 20
[2] vgl. Sedgewick 1992, S. 22 und Saake 2002, S. 17 f
[3] Saake 2002, S. 18

c. Vorgehen zur Untersuchung der Effizienz

Von Bedeutung sind hier (eigentlich) der benötigte Speicherplatz und die Rechenzeit, die ein Computer für die Durchführung braucht. Der Speicherplatz wird jedoch in dieser Facharbeit vernachlässigt. Zunächst einmal würde dies den Rahmen der Arbeit sprengen und auch der unmittelbare Zusammenhang zur Mathematik ist nicht mehr gegeben. Außerdem ermöglicht der technische Fortschritt gegenwärtig eine derartige Ausweitung der Speicherkapazitäten, dass die Speicherung selbst größerer Daten kein Problem mehr darstellt.[1]

Wichtig ist die Rechenzeit. Hierbei ist zu beachten, dass das Verfahren selbst, nicht aber die Rechenzeit eines bestimmten Computers für das umgesetzte Programm untersucht werden muss. Die letztendliche Rechenzeit hängt stark von der Leistung des benutzten Rechners ab. Deshalb wird in dieser Arbeit auch größtenteils auf Messergebnisse, die ja doch wieder durch die Rechnerleistung beeinflusst werden, verzichtet. Diese experimentell ermittelten Messergebnisse ließen sich nur schwer auf andere Rechner übertragen, wohingegen eine allgemeine Geschwindigkeitsformel aussagekräftig in Bezug auf alle Rechner ist. Dieses Verhältnis ist unabhängig vom eingesetzten Rechner.

Ermittelt wird daher allgemein das Wachstum der Laufzeit bei wachsender Problemgröße, nicht die genaue Laufzeit. Alternativ – und dies wird vornehmlich bei Verfahren zum Sortieren (Sortieralgorithmen) genutzt – untersucht man die Anzahl der Rechenoperationen und multipliziert diese mit einem konstanten Faktor k, der die Rechenzeit eines **bestimmten** Rechners für eine Rechenoperation enthält. Da das Thema dieser Facharbeit den Vergleich von Sortieralgorithmen beinhaltet, werde ich zu Beginn der Untersuchung letzteres Verfahren anwenden. Hieraus ergibt sich dann später auch das Wachstum der Laufzeit bei wachsender Problemgröße.

Alle anderen aufgeführten Aspekte (Prüfung auf Korrektheit, Untersuchung des Speicherplatzes, Rechenleistung des Computers) können demnach unbeachtet bleiben. Zur Untersuchung ist nur die Anzahl und Art der Rechenoperationen herauszufinden.

d. Sortieralgorithmen

Sortieralgorithmen sind ein einfaches Beispiel für die oben beschriebene Definition von Algorithmen und deren formale Eigenschaften. Sie sind z. T. leicht nachvollziehbar, weil die Problemstellung jedem vertraut ist.

Dies kann man am Beispiel eines Kartenspiels leicht zeigen: Um dies zu sortieren, sucht man die höchste Karte und steckt sie nach vorne, im Rest sucht man dann wieder die höchste Karte. Dieses Verfahren wird immer wiederholt und ist außerdem endlich, denn bei einer endlichen Zahl von Karten ist irgendwann das Ende des Stapels erreicht. Man muss nur genau so oft suchen (d. h., den Kartenstapel durchlaufen; die Zahl der Vergleiche ist natürlich weit höher), wie man Karten hat.

Es gibt einen Algorithmus, der genau dieses Verfahren nachvollzieht. Dieser ist sehr einfach.

---

[1] vgl. Ottmann, Widmeyer 1996, S. 2 ff

Sortieralgorithmen sind in der Informatik von zentraler Bedeutung. Immerhin etwa 25 % der kommerziell verbrauchten Rechnerzeit wird für Sortiervorgänge genutzt[1]. Daher wurde viel Zeit zum Auffinden schneller Algorithmen aufgewendet. Dies geschah gerade in Zeiten langsamer Rechner, wo schon das Sortieren von mehreren 1000 Elementen enormen Zeitaufwand kostete (Vergleich: heute wenige Sekunden). Das Ergebnis waren komplexere Algorithmen, die nicht mehr unseren intuitiven Handlungen entsprechen, aber zum Teil wesentlich schneller arbeiten (was ich in dieser Arbeit noch untersuchen möchte).
Das Thema der Facharbeit lautet „Darstellung und Vergleich von Sortieralgorithmen“. Exemplarisch werden zwei Sortieralgorithmen dargestellt und verglichen. Der erste Algorithmus (InsertionSort) ist einfach und verfährt wie Menschen intuitiv handeln. Dem gegenüber steht einer der populärsten Algorithmen der Informatik, zugleich aber auch ein sehr schwieriger: QuickSort.
Bei der Herleitung einer allgemeinen Geschwindigkeitsformel gehe ich davon aus, dass beide Algorithmen ein unsortiertes, zufällig erstelltes Feld bearbeiten sollen (average-case-Analyse). Wichtig ist aber auch die Betrachtung der Laufzeit jedes Algorithmus' im besten bzw. schlechtesten Fall. Dieser hängt von der Reihenfolge der zu sortierenden Elemente ab. Auch hierfür lässt sich eine Formel mathematisch herleiten.
Zum weiteren Verständnis sind zwei Grundbegriffe notwendig: Beim Sortieren handelt es sich immer um das Ordnen von *Datensätzen* mit bestimmten *Schlüsseln*. Zum Beispiel würden in einem Telefonbuch die Datensätze mit Name, Adresse, Telefonnummer etc. nach dem Schlüssel „Name“ sortiert. Die Sortierung wäre lexikographisch (s. Tabelle 1).

Schlüssel	Datensatz
Endig, Martin	Hoppenstr. 15, 0201-8245
Geist, Ingolf	Musterweg 8, 02054-2564
Zettner, Eike	Ruhrstr. 121, 0201-86589

Tabelle 1 - lexikographische Sortierung der Datensätze nach den Schlüsseln

Der Schlüssel gehört zum Datensatz, stellt jedoch häufig nur einen geringen Teil davon dar.
Ich beginne mit der Darstellung der Arbeitsweise von InsertionSort.

3. InsertionSort

a. Darstellung, Beschreibung, Arbeitsweise

InsertionSort bedeutet „Sortieren durch Einfügen“. Und so geht dieser Algorithmus auch vor.
Er folgt dem Verhalten, das die meisten Menschen beim Sortieren von Kartenspielen intuitiv anwenden. Die einfachere Version läuft so ab, dass man die erste Karte eines unsortierten Kartenstapels A auf einen neuen Stapel (der noch keine Karte enthält) B legt. Dann wird immer eine Karte des Stapels A an der richtigen Stelle in Stapel B eingefügt, so dass dieser immer sortiert ist.

[1] vgl. Saake 2002, S. 116

Die verfeinerte Version erledigt dies in einem Stapel. Als Anleitung formuliert müsste man so vorgehen: „Betrachte die Elemente eines nach dem anderen und füge jedes an seinen richtigen Platz zwischen den bereits betrachteten ein (wobei diese sortiert bleiben).“[1]
An Abbildung 1 kann man einen einzelnen Schritt nachvollziehen:

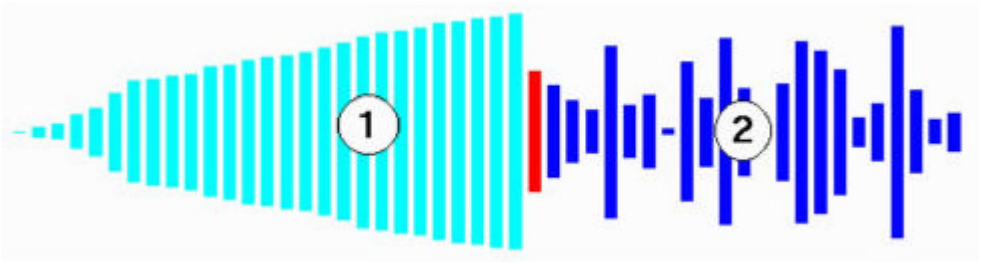

Abbildung 1 - Darstellung eines Sortierschrittes von InsertionSort[2]

Bereich 1 stellt den bereits sortierten Teil dar. Teil 2 enthält die noch unsortierten Datensätze. Der rote Datensatz wird als nächstes betrachtet und dann im sortierten Bereich 1 an der richtigen Stelle eingefügt. Teil 1 bleibt so sortiert. Dann wird der nächste Datensatz aus Teil 2 in Teil 1 eingefügt.
Das genaue Einfügen läuft so ab: Die größeren Elemente links vom roten Datensatz werden um eine Stelle nach rechts verschoben – bis ein Element erreicht wird, das kleiner als das rote Element ist. Hinter diesem Element wird das rote Element eingefügt.
Den Ablauf aller Schritte kann man in Abbildung 3 (**Seite 7**) erkennen. Hier sollen Buchstaben alphabetisch sortiert werden. Das S ist größer als das A und bleibt auf seiner Position. Das O ist kleiner als das S. Also wandert das S nach rechts und das O wird an der zweiten Stelle eingefügt, usw.[3]

1.	A	I	O	R	S	T	N	…
2.	A	I	O	R	S	T	T	…
3.	A	I	O	R	S	S	T	…
4.	A	I	O	R	R	S	T	…
5.	A	I	O	O	R	S	T	…
6.	A	I	N	O	R	S	T	…

Abbildung 2 - Einfügen eines Elements bei InsertionSort

In Abbildung 2 wird jeder einzelne Schritt gezeigt, mit dem das N aus Abbildung 3 (6. Zeile) an die richtige Stelle gelangt. In Zeile 1 von Abbildung 2 wird das N mit dem T verglichen. Da das T größer ist, wandert es nach rechts. In Zeile 3 wird das N mit dem S verglichen – auch dies ist größer und wandert nach rechts usw. Schließlich in der 6. Zeile ist das I nicht größer, so dass es nicht verschoben wird. Hier wird das N eingefügt.

b. Worst-Case-Analyse, Best-Case-Analyse

Zunächst beginnt man mit einer allgemeinen Analyse, indem man den schlechtest- und bestmöglichen Fall bestimmt. Dies hilft bei der späteren Auswahl der Einsatzgebiete.

---

[1] Sedgewick 1992, S. 127
[2] Kaneva, Thiébaut 1997
[3] vgl. Sedgewick 1992, S. 127

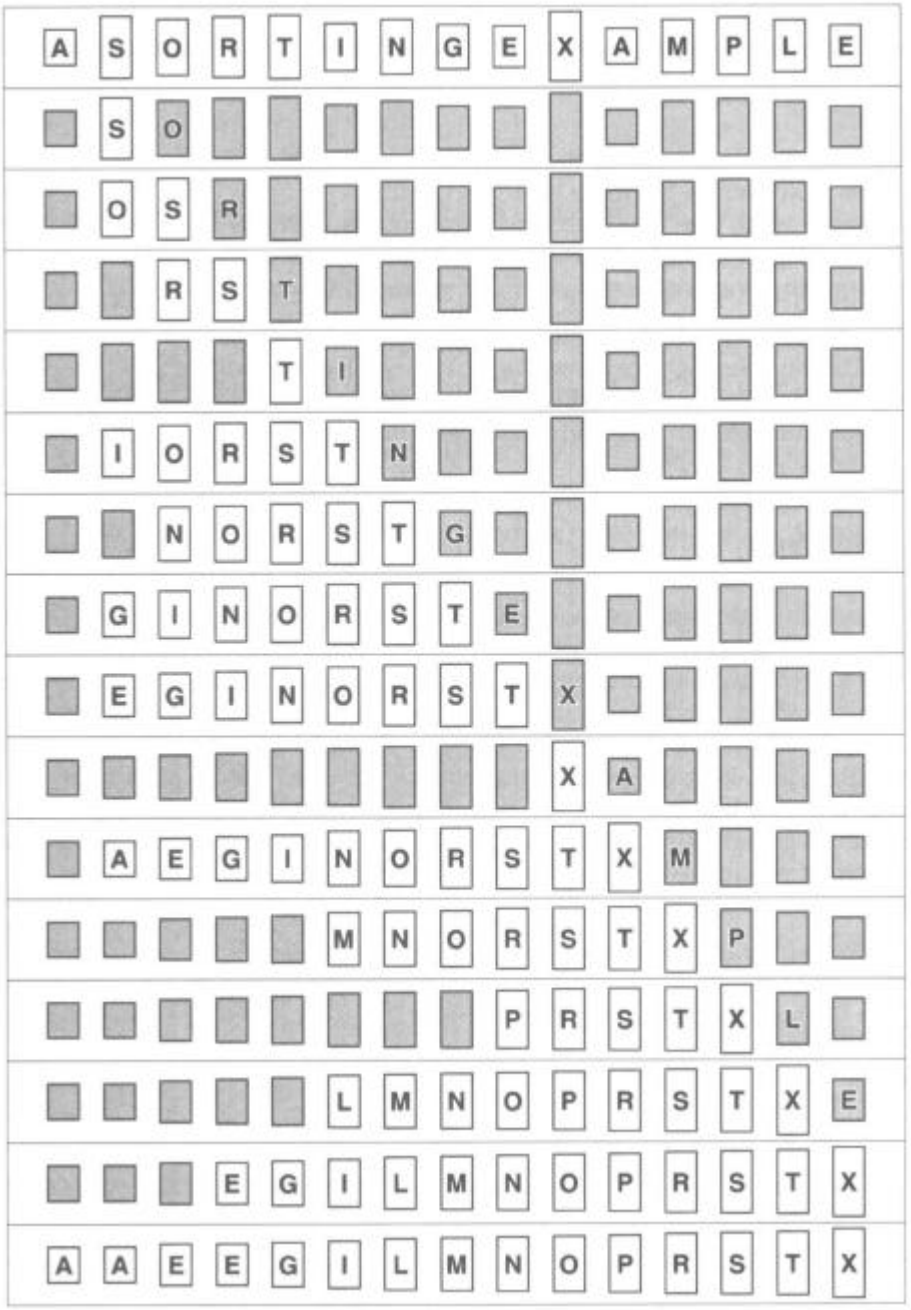

Abbildung 3 - Arbeitsweise von InsertionSort[1]

Die Elemente wandern – sofern erforderlich – nach links. Je weiter sie wandern müssen, um an ihren Platz zu kommen, desto mehr Vergleiche und Austauschoperationen werden benötigt. Dementsprechend steigt der Sortieraufwand. Der schlechtestmögliche Fall („worst-case“) wäre dementsprechend, wenn die Elemente den weitesten Weg gehen müssen. Dies ist der Fall, wenn das Feld in umgekehrter Reihenfolge, also absteigend sortiert, vorliegt. Jedes Element wandert dann bis zum Anfang des Feldes. Schließlich muss das kleinste Element vom rechten Ende bis zum Anfang wandern:

[1] Sedgewick 1992, S. 128

4	3	2	1
3	4	2	1
2	3	4	1
1	2	3	4

Abbildung 4 - Darstellung d. Sortiervorgangs im „worst case“

Die vier muss nicht verglichen werden. Die drei wandert auf Platz 1, also den für sie weitestmöglichen Weg. Dann wandert die zwei nach ganz vorn – wieder den für sie weitestmöglichen Weg – und schließlich die 1.
Wenn man den Elementen einen Index *i* zuordnet, ist die größtmögliche Anzahl an Vergleichen für jedes einzelne Element *i*-1. Element 1 hat keinen Vergleich (1-1 = 0), Element zwei hat einen Vergleich (2-1 = 1), usw. Daraus ergibt sich für die gesamte (und größtmögliche) Anzahl an Vergleichen für *n* Elemente:

$$\text{Vergleiche}(n) \le \sum_{i=2}^{n} (i-1) = n\frac{(n-1)}{2} = \frac{n(n-1)}{2}$$

Die Vereinfachung ergibt sich aus der Gauß'schen Erkenntnis, dass der größte und der kleinste Summand addiert n ergeben, der zweitgrößte und zweitkleinste auch usw.:

Für *n*=5: $=5\left[\begin{array}{l}2\text{-}1\\3\text{-}1\\4\text{-}1\\5\text{-}1\end{array}\right.\left.\begin{array}{l}\\\\\\\end{array}\right\}=5 \quad = 10 = 2*5 = 2n = \frac{5-1}{2}n = \frac{n-1}{2}n$

Diese Zusammenfassung von zwei Summanden zum Ergebnis *n* werde ich in der folgenden Erklärung als Zahlenpaar bezeichnen. Dieses Ergebnis *n* gibt es dementsprechend so oft wie Zahlenpaare. Da ein Zahlenpaar aus zwei Summanden i-1 besteht, gibt es halb so viele Zahlenpaare wie Summanden. Würde die Summe von 1 bis *n* laufen, gäbe es *n* Summanden; da die Summe von 2 bis *n* läuft, gibt es einen Summanden weniger, also *n*-1. Der Wert eines Zahlenpaares (*n*) – also die Summe zweier Summanden i-1 – muss mit der Anzahl der Zahlenpaare

$$\left.\begin{array}{l}\frac{n-1}{2}\end{array}\right\} \text{Anzahl der Summanden} \quad \left.\begin{array}{l}\\\end{array}\right\} \text{Anzahl der Zahlenpaare}$$

multipliziert werden, woraus sich obige Formel ergibt.

Der Fall *n*(*n*-1)/2 – also die größtmögliche Anzahl an Vergleichen – tritt tatsächlich ein, wenn das Feld in umgekehrter Reihenfolge vorliegt. Ansonsten ist die Anzahl der Vergleiche kleiner.

Der bestmögliche Fall für InsertionSort wäre ein bereits (aufsteigend) sortiertes Feld. In diesem Fall finden keine Verschiebungen bzw. Austauschoperationen und lediglich *n*-1 Vergleiche statt, da jedes Element mit Ausnahme des ersten nur mit seinem Vorgänger verglichen wird, also einmal. Der Vorgänger ist kleiner (es wurde ein bereits sortiertes Feld vorausgesetzt), so dass für jedes Element dieser eine Vergleich ausreicht. Das Gleiche gilt für ein Feld, in dem alle Zahlen gleich sind (wobei auch dies bereits sortiert ist). Daher gilt:

Vergleiche(*n*)=*n*-1

BEST CASE	**17**	**25**	**36**	**39**	**48**	**50**	**51**	**59**	**78**	**99**
WORST CASE	**98**	**91**	**74**	**55**	**40**	**35**	**16**	**15**	**8**	**1**

Tabelle 2 – Beispielfelder InsertionSort

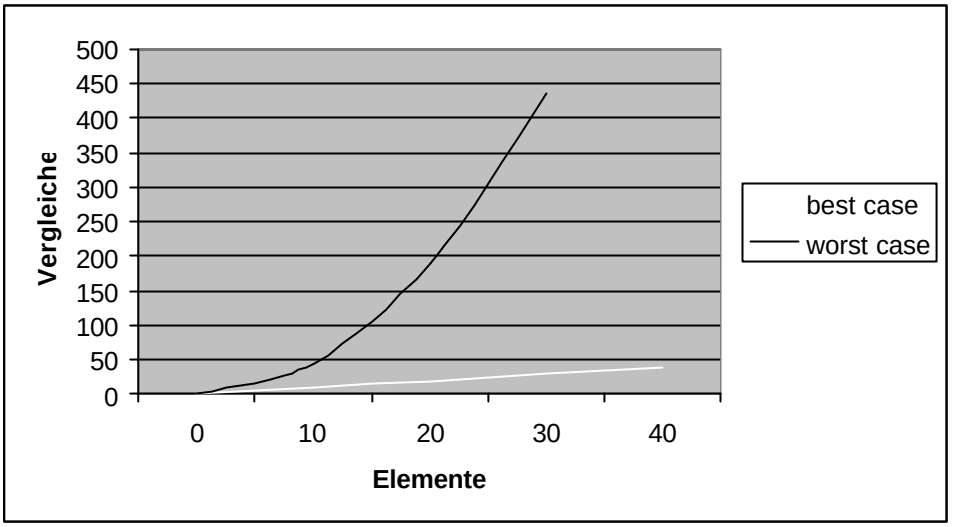

Abbildung 5 - Unterschiede zw. best case und worst case

Tabelle 2 zeigt zwei Beispiele. In Abbildung 5 werden die Anzahl der Vergleiche in Abhängigkeit von der Anzahl der Elemente dargestellt. Man erkennt den enormen Unterschied zwischen dem bestmöglichen und schlechtesten Fall.
Nun muss noch die Anzahl der Austauschoperationen untersucht werden, denn aus Vergleichen und Austauschoperationen setzt sich die Laufzeit zusammen. Im „best case“ sind keine Austauschoperationen nötig, weil alle Elemente schon am richtigen Platz sind. In diesem Fall ist die Zeit also nur abhängig von der Anzahl der Vergleiche:

$$\text{Zeit}(n) \sim n\text{-}1$$

Die Anzahl der Austauschoperationen ist – wie in Punkt 3c. bewiesen wird – annähernd proportional zur Anzahl der Vergleiche. Deshalb ist im „worst case“ der Anstieg der Laufzeit in Abhängigkeit zur Anzahl der Elemente schon an der Anzahl der Vergleiche zu erkennen. Im „worst case“ gilt:

$$\text{Zeit}(n) \sim n(n-1)/2 \approx n^2/2 \sim n^2$$

Im „best case“ läuft InsertionSort demnach linear, also sehr schnell, im „worst case” jedoch fast quadratisch.

c. Geschwindigkeit: Herleitung der Formel

In Abbildung 2 (**Seite 6**) kann man schon eine wichtige Feststellung machen:

> „Zur Bestimmung der Einfügestelle wird stets eine Vergleichsoperation mehr als die Anzahl der Verschiebungen durchgeführt.“[1]

In dem Beispiel werden vier Verschiebungen durchgeführt (von den Buchstaben T, S, R und O). Vergleiche finden aber mit dem T, S, R, O und I

[1] Ottmann, Widmeyer 1996, S. 69

statt – also einmal mehr –, denn man weiß nicht im Voraus, dass das nächste Element (hier: I) kleiner sein wird.

Angenommen ein Element befindet sich schon an der richtigen Stelle: Es muss nicht (0 mal) verschoben werden, es findet aber ein Vergleich statt. Die obige Behauptung ist also richtig.
Bei der Analyse des „worst case“ habe ich angenommen, dass jedes Element den maximalen Weg zurücklegt, also i-1 Vergleiche für jedes Element benötigt werden. Daraus ergab sich für die Anzahl der Vergleiche $C_N = \sum_{i=2}^{N} (i-1)$. (Aus Gründen der Übersichtlichkeit verwende ich hier die Schreibweise C_N, diese ist gleichzusetzen mit der Funktion $C(N)$.) Im „best case“ wurden hingegen N-1 Vergleiche benötigt, weil die Elemente überhaupt nicht vertauscht werden und so auch nur ein Vergleich für jedes Element mit Ausnahme des ersten nötig ist. Im durchschnittlichen Fall kann man davon ausgehen, dass jedes Element den halben Weg zurücklegen muss, so dass man über die bereits erklärte Formel im „worst case“ die Beziehung im durchschnittlichen Fall erhält:

$$C_N = \sum_{i=2}^{N} \frac{i-1}{2}$$

Da nur der halbe Weg zurückgelegt wird, wird auch nur die Hälfte der Vergleiche vom „worst case“ benötigt:

$$C_N = \sum_{i=2}^{N} \frac{i-1}{2} = \frac{\frac{N(N-1)}{2}}{2} = \frac{N(N-1)}{4} \approx \frac{N^2}{4}$$

Die Auflösung der Formel entspricht der Hälfte der im „worst case“ benötigten Vergleiche.

$$\text{Zeit}(N) \approx i\frac{N^2}{4} + j(\frac{N^2}{4} - 1) = i\frac{N^2}{4} + j\frac{N^2}{4} - j = \frac{N^2}{4}(i+j) - j$$

Wie oben beschrieben findet eine Austauschoperationen weniger als Vergleiche statt, also $(N^2/4)-1$. Diese Operationen müssen mit den rechnerabhängigen Kosten für eine Austauschoperation j multipliziert werden. Die Zeit ist etwas geringer als die Gleichung angibt, da bei der Umformung von N(N-1) – eigentlich gleich N^2-N – in N^2 etwas aufgerundet wurde. Je größer das Feld jedoch wird, desto mehr nähern sich die Funktionswerte von N^2-N und N^2 einander. Je größer die Felder werden, desto weniger wirkt sich gleichzeitig die Subtraktion der Kosten für eine einzelne Austauschoperation j auf die Zeit aus. Bei enorm großen Feldern hat diese keine Bedeutung, so dass weiter vereinfacht werden kann:

$$\text{Zeit(N)} < \frac{N^2}{4}(i+j) - j \approx \frac{N^2}{4}(i+j) = k\frac{N^2}{4} \sim kN^2 \sim N^2$$

Der Faktor k beinhaltet (i+j) und stellt somit die rechnerabhängigen Kosten dar.
Damit ergibt sich annähernd

$$\text{Zeit(n)} \sim n^2$$

InsertionSort benötigt im Durchschnitt $(n^2/4)$ die Hälfte der Zeit des „worst case“ $(n^2/2)$. In beiden Fällen ist der Anstieg aber quadratisch.

Da die Anzahl der Vergleiche $n^2/4$ proportional zur Zeit n^2 ist, muss die Anzahl der Vergleiche auch proportional zur Anzahl der Austauschoperationen sein. Ansonsten könnte die Summe aus Vergleichen und Austauschoperationen nicht mehr proportional zur Zeit sein.
Die obige Behauptung – Austauschoperationen seien proportional zu der Anzahl der Vergleiche (s. Punkt 3b.) – ist demnach richtig.

4. QuickSort

a. Darstellung, Beschreibung, Arbeitsweise

QuickSort ist ein sehr schneller Algorithmus. Daher stammt sein Name. Entwickelt wurde er schon 1960 von C. A. R. Hoare. QuickSort arbeitet nach dem Prinzip „divide and conquer"/„divide et impera" (teile und herrsche). Das Prinzip ist, eine Datei in zwei Teile zu zerlegen und diese unabhängig voneinander auf die gleiche Art zu sortieren.
Dabei wird folgendermaßen vorgegangen: Man sucht sich willkürlich ein Element x. Daraufhin durchsucht man das Feld von links, bis ein Element $a_i > x$ gefunden wird; und von rechts, bis ein Element $a_j < x$ erreicht ist. Diese Elemente werden vertauscht. So gelangt das kleine Element nach links, das große nach rechts. Dieser Vorgang wird weitergeführt, bis man sich beim Durchlaufen aus beiden Richtungen trifft.[1]
Im linken Teil sind nun alle Elemente kleiner als x, im rechten größer als x. Zwischen diesen Feldern muss nicht mehr ausgetauscht werden. Das Problem wurde soweit vereinfacht, dass man nur noch die beiden einzelnen Teile sortieren muss. Hierbei geht man genauso vor.[2]

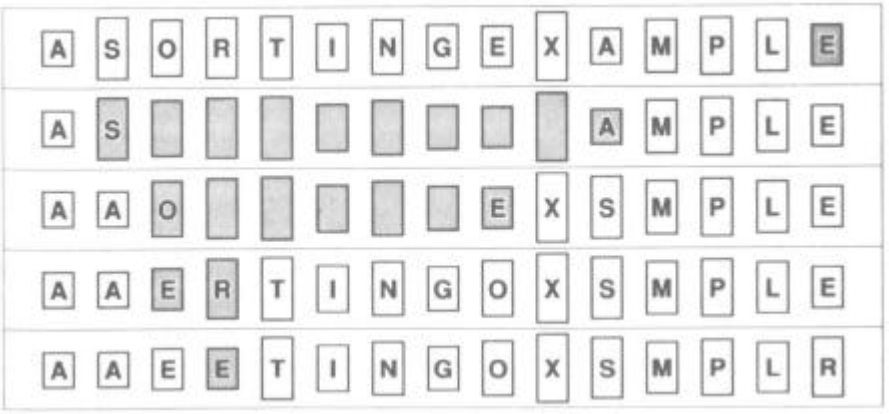

Abbildung 6 - Zerlegen bei QuickSort[3]

An Abbildung 6 kann man diesen Prozess nachvollziehen. „Das am weitesten rechts befindliche Element E wird als das zerlegende Element gewählt. Zuerst wird das Durchsuchen von links bei S unterbrochen, dann das Durchsuchen von rechts bei A (wie die zweite Zeile [...] zeigt), und dann werden diese Elemente vertauscht. Im nächsten Schritt wird das Durchsuchen von links bei O unterbrochen, dann das Durchsuchen von rechts bei E (wie die dritte Zeile [...] zeigt), dann werden diese beiden Elemente vertauscht. Danach treffen sich die Zeiger. Das Durchsuchen von

[1] vgl. Gierhardt 2001
[2] vgl. Gierhardt 2001
[3] Sedgewick 1992, S. 147

links wurde bei R unterbrochen und das Durchsuchen von rechts bei E. Die richtige Operation besteht nun darin, das rechts stehende E mit dem R zu vertauschen, wodurch die zerlegte Datei entsteht, die in der letzten Zeile [...] gezeigt wird."[1]

b. Worst-Case-Analyse, Best-Case-Analyse

Das „ideale" Feld gibt es bei QuickSort nicht, da die Geschwindigkeit auch von der Wahl des zerlegenden Elements abhängt. Das Beste für QuickSort wäre, wenn jede Zerlegung die Datei halbiert. Die hängt aber eben nicht nur vom Feld allein, sondern auch von der Wahl des zerlegenden Elements (Pivot-Element) ab. Dies kann man zufällig wählen, man kann auch (wie oben) das am weitesten rechts liegende Element auswählen. In dem Fall kommt Tabelle 3 dem Ziel einer genauen Halbierung sehr nahe.

BEST CASE	**1**	**59**	**3**	**70**	**5**	**60**	**42**	**61**	**24**	**50**

Tabelle 3 - best case QuickSort

Zumindest beim ersten Durchgang wird das Feld durch die 50 genau halbiert. Auch die beiden entstehenden Teilfelder werden durch das jeweilige zerlegende Element halbiert (zweiter Durchgang). QuickSort würde hier sehr schnell laufen, auch wenn im dritten Durchgang die Teilung nicht mehr ideal abläuft. Bei der Herleitung der drei Geschwindigkeitsformeln setze ich voraus, dass immer das am weitesten rechts liegende Element als Pivot-Element gewählt wird.

Wie erwähnt ist der beste Fall eine genaue Halbierung. Damit würde die Anzahl der Vergleiche

$$C_N = 2C_{N/2} + N$$

genügen. $2C_{N/2}$ beinhaltet hierbei die Kosten des Sortierens für die zwei Teildateien und N die Kosten für die Prüfung eines jeden Elements.
Hiernach wird N so oft halbiert, d. h. durch 2 dividiert, bis nur zwei Elemente übrig bleiben. Die Anzahl der Teilungen wird also durch $\log_2 N$ beschrieben. Bei jeder Teilung wird N addiert, muss also mit der Anzahl der Teilungen (s. Logarithmus) multipliziert werden[2]:

$$C_N = N \log_2 N$$

Für die Geschwindigkeit müssen nun noch die Austauschoperationen betrachtet werden. Bei einer Teilung, die das Feld genau halbiert, befindet sich jedes Element mit einer Wahrscheinlichkeit von 50 % in der richtigen Hälfte, zu 50 % muss es in die andere Hälfte gebracht werden. Da bei den Vergleichen das Pivot-Element mit jedem anderen Element verglichen wird, muss also jedes zweite Element ausgetauscht werden. Anders ausgedrückt müssen halb so viele Elemente vertauscht werden wie Vergleiche stattfinden. Daraus *ergäbe* sich:

$$\text{Austauschoperationen}(n) = \text{Vergleiche}(n)/2$$

Nun muss an eine wichtige Eigenschaft von QuickSort gedacht werden: Bei einem Austausch wechseln gleich zwei Elemente ihren Platz. Wenn also ein Element von (Teil-)Feld 1 in (Teil-)Feld 2 getauscht werden muss, so wird

[1] Sedgewick 1992, S. 147
[2] für andere Herleitungswege vgl. Saake 2002, S. 133 f

nur ½ Austauschoperation benötigt. Denn gleichzeitig gelangt ein Element aus Feld 2 in Feld 1. Es werden also zwei Vertauschungen mit einer Operation durchgeführt, wodurch sich die Behauptung einer halben Austauschoperation ergibt. Die Vertauschungen müssen also noch einmal durch 2 dividiert werden, um die Austauschoperationen zu erhalten:

$$\text{Austauschoperationen}(n) = \text{Vergleiche}(n)/2/2$$

$$\text{Austauschoperationen}(N) = \frac{1}{4} N \log_2 N$$

Damit ergibt sich für den Zeitaufwand:

$$\text{Zeit}(n) = jn \log_2 n + i\frac{1}{4} n \log_2 n = n \log_2 n (j + i\frac{1}{4}) = kn \log_2 n \text{ bzw.}$$

$$\text{Zeit}(n) \sim n \log_2 n .$$

Der Faktor k beinhaltet rechnerabhängige Kosten (j+i/4). Hier ist zu erkennen, dass die Austauschoperationen bei QuickSort für den Anstieg der Zeit nur eine geringe Rolle spielen, denn die Funktion für den Anstieg der Zeit stimmt mit der für die Anzahl der Vergleiche überein.
Ideal wäre natürlich, wenn so wenig Austauschoperationen wie möglich, bestenfalls keine, stattfinden. Dann verhielte sich die Zeit proportional zu den Vergleichen. Dies soll jetzt gezeigt werden, so dass man im „best case" immer zu dem Ergebnis kommt: Die Zeit verhält sich proportional zu $n \log_2 n$.
Der Algorithmus ist für einfache Dateien sehr uneffizient. Bei einer bereits sortierten Datei „entarten die Zerlegungen"[1], denn bei jedem Aufruf scheidet nur ein Element aus.[2]

WORST CASE	**1**	**2**	**3**	**4**	**5**	**35**	**45**	**49**	**60**	**63**

Tabelle 4 - worst case QuickSort

So verkleinern sich die Felder nur sehr langsam, nämlich mit jeder Teilung um 1 Element. Das Pivot-Element (zerlegendes Element) wird in der 1. Teilung mit n-1 Elementen verglichen, in der zweiten Teilung mit n-2 (da ein Element aussortiert wurde), bei der dritten Teilung mit n-3, usw. Allgemein gilt damit n-i Vergleiche in einem Schritt, wenn i die Nummer der Teilung ist. Die Vergleiche aller Schritt müssen summiert werden (s. Gleichung). Damit ergibt sich für den ungünstigsten Fall:

$$C_N = \sum_{i=1}^{n} (n - i) = \frac{n}{2}(n - 1)$$

Die letzte Vereinfachung beruht wieder auf dem bereits dargestellten Gauß'schen Trick.
Die Anzahl der Austauschoperationen im „worst case" hängt entscheidend von der Implementation, also der letztendlichen Umsetzung der Sortieridee in Computersprache für die einzelnen Rechner ab. Bei manchen Programmen wird immer das letzte Element mit sich selbst vertauscht, so dass n Austauschoperationen stattfinden. An Tabelle 4 erkennt man jedoch, dass ein Austausch gar nicht mehr nötig ist, da schon alle Elemente am richtigen Platz angeordnet sind. Dies wird auch in gängigen

[1] Sedgewick 1992, S. 150
[2] vgl. Sedgewick 1992, S. 151

Implementationen nachvollzogen; dementsprechend ergeben sich 0 Austauschoperationen. Damit hängt die Zeit nur von der Anzahl der Vergleiche ab.
Der Teil n(n-1) liegt sehr nah an n², so dass sich ungefähr n²/2 – proportional zu n² – ergibt. Man kann also sagen, im „worst case" verläuft QuickSort (fast) quadratisch:

worst case: Zeit(n) ~ n²

c. Geschwindigkeit: Herleitung der Formel

Ein zerlegendes Element N muss sich mit jedem anderen Element vergleichen (N-1 Vergleiche), hinzu kommen zwei Vergleiche, wenn sich die von links und rechts kommenden Zeiger treffen, es ergeben sich N+1 Vergleiche. Hinzu kommen die Vergleiche der Teilfelder. So ergibt sich die rekurrente Beziehung

$$C_N = N+1+\frac{1}{N}\sum_{1\le k\le N}(C_{k-1}+C_{N-k}),\ \text{für}\ N\ge 2,\ \text{mit}\ C_1=C_0=0.$$

Die rechte Definition ist leicht einsehbar, denn wenn kein oder nur ein Element sortiert werden soll, ist keine Sortierung und damit kein Vergleich notwendig. Für das zerlegende Element k gilt $1\le k\le N$, wobei k jeden Wert mit gleicher Wahrscheinlichkeit annehmen kann. In jedem Fall liegen damit zwei Felder der Größe k-1 und N-k vor. Um den Durchschnitt (arithmetisches Mittel) zu erhalten, werden die Anzahl der Vergleiche aller möglichen Zerlegungen addiert und durch die N Möglichkeiten geteilt, d. h. mit 1/N multipliziert.
Diese recht komplizierte Gleichung kann man vereinfachen. Zunächst ist jedes Teilfeld in der Summe zweimal enthalten, denn

$$\underbrace{C_0 + C_1 + \ldots + C_{N-1}}_{\sum_{1\le k\le N} C_{k-1}} = \underbrace{C_{N-1} + C_{N-2} + \ldots + C_0}_{\sum_{1\le k\le N} C_{N-k}}$$

Deshalb sind C_{k-1} und C_{N-k} gleichzusetzen (s. Gleichung oben). Es gilt

$$C_N = N+1+\frac{2}{N}\sum_{1\le k\le N} C_{k-1}.$$

Nun wird die Summe entfernt. Hierzu multipliziert man beide Seiten mit N und subtrahiert dann die gleiche Formel auf beiden Seiten für N-1.

$$NC_N = N^2 + N + 2\sum_{1\le k\le N} C_{k-1} \Leftrightarrow$$

$$NC_N - (N-1)C_{N-1} = N(N+1) + 2(\sum_{i\le k\le N} C_{k-1}) - N(N-1) - 2(\sum_{i\le k\le N-1} C_{k-1})$$

N²+N wird vereinfacht durch N(N+1). Wenn man nun 1 von N subtrahiert, erhält man (N-1)((N-1)+1); vereinfacht ergibt sich (N-1)N = N(N-1), wie es in der obigen Gleichung auftaucht.
Das zweite Summenzeichen hebt nun alle Summanden des ersten Summenzeichens auf – mit Ausnahme von C_{k-1} für den Fall k=N. Von den Summen bleibt also C_{k-1} für k=N gleich C_{N-1} übrig. Es ergibt sich
$NC_N - (N-1)C_{N-1} = N(N+1) - N(N-1) + 2C_{N-1}$ und weiter vereinfacht

$$\Leftrightarrow NC_N = N(N+1) - N(N-1) + 2C_{N-1} + (N-1)C_{N-1} =$$
$$N(N+1) - N(N-1) + C_{N-1}(2 + (N-1)) =$$
$$N(N+1) - N(N-1) + C_{N-1}(N+1) =$$
$$N(N+1-N+1) + C_{N-1}(N+1) =$$
$$2N + C_{N-1}(N+1)$$
Also: $NC_N = (N+1)C_{N-1} + 2N$.

Mittels Division beider Seiten durch $N(N+1)$ ergibt sich die Beziehung

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}.$$

Man erkennt hier eine rekurrente Beziehung. Bei der Fortführung wird immer das $\frac{C_x}{x+1}$ ersetzt. In der Formel steht $\frac{C_{N-1}}{N} + \frac{2}{N+1}$. In diesem Fall wird dann (bei der Fortführung) das $\frac{C_{N-1}}{N}$ ersetzt, das $\frac{2}{N+1}$ bleibt stehen. Diese Ersetzung geschieht nach der Formel am Anfang der Gleichung $\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$. Hierbei muss der Wert von N beachtet werden, der um 1 reduziert ist. Dann folgt $\frac{C_{N-1}}{N}$ gleich $\frac{C_{N-2}}{N-1} + \frac{2}{N}$ (hier wurde das N auf beiden Seiten um 1 reduziert). Diese Beziehung lässt sich somit fortsetzen mit

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \ldots = \frac{C_2}{3} + \sum_{3 \le k \le N} \frac{2}{k+1}.$$

Diese Schritte werden in der Summe zusammengefasst. Hier beginnen die Kosten mit denen für das Sortieren von 3 Elementen und summieren sich dann mit denen bis N Elementen.

Dieses Ergebnis ist exakt, jedoch als Merkformel zu kompliziert und dadurch eher unbrauchbar. Nach Robert Sedgewick entspricht es aber „ziemlich genau“[1] der Summe $\sum_{1 \le k \le N} \frac{2}{k}$. Damit gilt

$$\frac{C_N}{N+1} \approx 2 \sum_{1 \le k \le N} \frac{1}{k}.$$

Diese kann durch das Integral

$$2 \int_1^N \frac{1}{x} dx$$

annähernd bestimmt werden. Der genaue Rechenschritt wird hier nicht erklärt, der Zusammenhang ist aber offensichtlich: So wie das Summenzeichen gibt auch das Integral eine Summe an, wobei beim Integral lediglich „Streifen“ addiert werden. Das Integral hat die Lösung ln$N$, so dass die letzten Schritte in der Gleichung

$$\frac{C_N}{N+1} \approx 2 \sum_{1 \le k \le N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx = 2 \ln N$$

[1] Sedgewick 1992, S. 152

zusammenfassend wiedergegeben werden können. Dieses Ergebnis mit N multipliziert ergibt für die ungefähre Anzahl der Vergleiche durchschnittlich

$$\text{Vergleiche}(n)=2n\ln n.$$

Eine interessante Erkenntnis lässt sich aus folgender Formel ableiten:

$$\underbrace{2N \ln N}_{\text{average}} \approx 1{,}38\underbrace{N \log_2 N}_{\text{best}}$$

Im durchschnittlichen Fall ist bei QuickSort die Anzahl der Vergleiche nur um ungefähr 38 % größer als im besten Fall.[1]

Für die Geschwindigkeit müssen nun noch die Austauschoperationen betrachtet werden. Für die Austauschoperationen gilt auch hier:

$$\text{Austauschoperationen}(n) = \text{Vergleiche}(n)/4,$$

was auch bei der empirischen Analyse (siehe Tabelle 5) belegt wird (vgl. auch Punkt 4b. dieser Arbeit).

$$\text{Austauschoperationen}(n) = \frac{1}{2} n \ln n$$

Damit ergibt sich ein Zeitaufwand von

$$\text{Zeit}(n)= j 2n \ln n + i\frac{1}{2} n \ln n,$$

wobei i und j die Kosten für die jeweilige Operation beinhalten.

Weiter ergibt sich

$$\text{Zeit}(n)= n \ln n\underbrace{(2j + \frac{1}{2} i)}_{k} = kn \ln n$$

k = rechnerabhängige Kosten

Wenn man nun die Kosten für Operationen, die auf jedem Rechner unterschiedlich sind, vernachlässigt, ergibt sich für den Zeitaufwand die Beziehung

$$\text{Zeit}(n) \sim n \ln n$$

Durch den zugrunde liegenden Logarithmus ist die Laufzeit sehr gering. Hierauf wird im abschließenden Vergleich genauer eingegangen.

Die obige Formel ist mathematisch hergeleitet und somit exakt. Im Testlauf in der Praxis kann man dies erkennen (s. Tabelle 5 und Tabelle 6).

Elemente	Zeit (ms.)	Vergleiche	Austauschoperationen
1 000 000	1750	29746653	6759397
2 000 000	3840	62716850	14493171
4 000 000	8080	130175617	30987568
8 000 000	16860	270433024	66107487
16 000 000	88100	562232713	140079874

Tabelle 5 - Messwerte QuickSort (empirische Analyse)

[1] bzgl. Herleitung vgl. u. a. Sedgewick 1992, S. 151 f, und Meinel 1991, S. 137 f

Elemente	Vergleiche	Austauschoperationen
1 000 000	-7 %	2 %
2 000 000	-7 %	0 %
4 000 000	-7 %	-2 %
8 000 000	-6 %	-4 %
16 000 000	-6 %	-5 %

Tabelle 6 - Differenz d. Messergebnisse zur Formel

Bei der Analyse dieser Tabellen muss bedacht werden, dass die obigen Formeln den Durchschnitt der Summe aller möglichen Permutationen eines Feldes angeben. Da die Testläufe mit zufällig erstellten Feldern stattgefunden haben, müssen sie dem Ergebnis der Formeln nahe kommen. Je mehr Durchläufe ausgeführt werden, desto exakter werden die Ergebnisse zur Formel passen. Doch schon hier sieht man die Korrektheit der Formel, denn die tatsächlichen Vergleiche weichen nur um maximal 7 % von der durchschnittlichen Anzahl an Vergleichen ab. So wurden bei einer Million Elemente etwas weniger (2 %) Austauschoperationen als durchschnittlich benötigt, diese Zeitersparnis wird durch etwas höheren Zeitaufwand (2 %) bei vier Million Elementen wieder aufgehoben, so dass der Durchschnitt der Austauschoperationen in etwa mit dem Messergebnis der Operationen für ein, zwei und vier Million Elemente übereinstimmt.

5. Vergleich

a. Gegenüberstellung

Zunächst eine zusammenfassende Darstellung der hergeleiteten Formeln. Die Formeln zeigen den Anstieg der Laufzeit in Abhängigkeit von der Anzahl der Elemente:

Sortierung	InsertionSort	QuickSort
best case	n-1(linear)	$n \operatorname{ld} n$
average case	n^2 (quadratisch)	$n \ln n$
worst case	n^2 (quadratisch)	n^2 (quadratisch)

Tabelle 7 - Anstieg des Sortieraufwands (Funktionen sind proportional zur Zeit)

Zunächst kann man sagen, dass ein linearer Anstieg (wie es bei InsertionSort im „best case“ der Fall ist) für einen Algorithmus optimal ist. Werden die Elemente verdoppelt, verändert sich die Laufzeit für jedes einzelne Element nicht.
Liegt der Laufzeit ein Logarithmus zugrunde (wie z. B. $n \operatorname{ld} n$), läuft der Algorithmus immer noch sehr schnell. Wird n verdoppelt, ist das Ergebnis zwar mehr als doppelt so groß, aber nicht wesentlich mehr.
Ein Algorithmus mit einer quadratischen Laufzeit ist nur auf kleine Probleme anwendbar. Werden die Elemente verdoppelt, vervierfacht sich schon die Laufzeit.
Demnach ist InsertionSort im „best case“ sehr schnell, für große Felder mit zufälliger Permutation aber völlig ungeeignet. In diesem Fall ist QuickSort mit seiner Laufzeit von $n \ln n$ noch sehr schnell und hat daher auch seinen Namen.

b. Unterschiedliche Stärken

Im „best case“ ist InsertionSort ideal. Wie bereits erörtert gilt dies für schon sortierte Felder. Bereits sortierte Felder sind für QuickSort der „worst case“. Daraus ergibt sich, dass InsertionSort eingesetzt werden sollte, wenn beinahe sortierte Felder sortiert werden sollen (s. Abbildung 7).

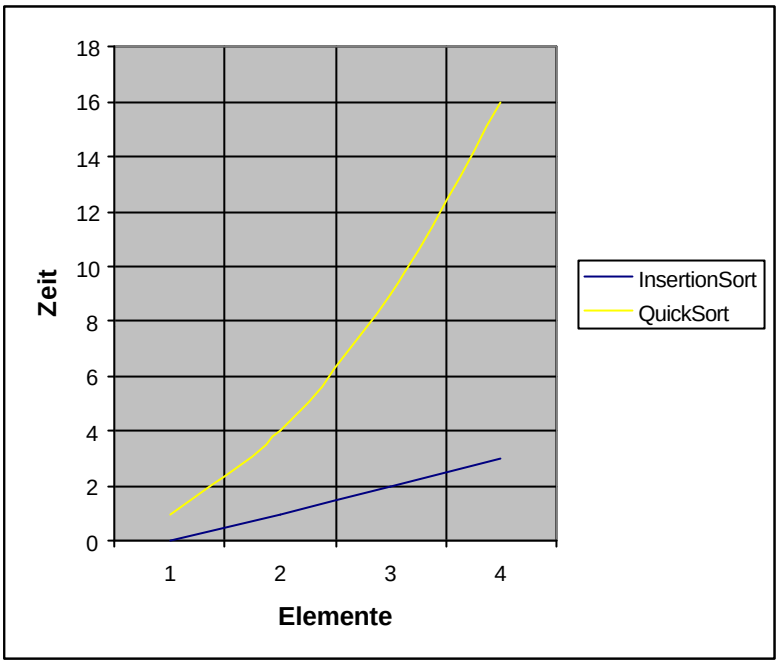

Abbildung 7 - Anstieg des Zeitaufwands bei sortierten Feldern

QuickSort hingegen ist bei unsortierten Feldern schnell. Hier liegt der Laufzeit ein Logarithmus zugrunde, der sehr langsam steigt, wobei InsertionSort quadratisch läuft.

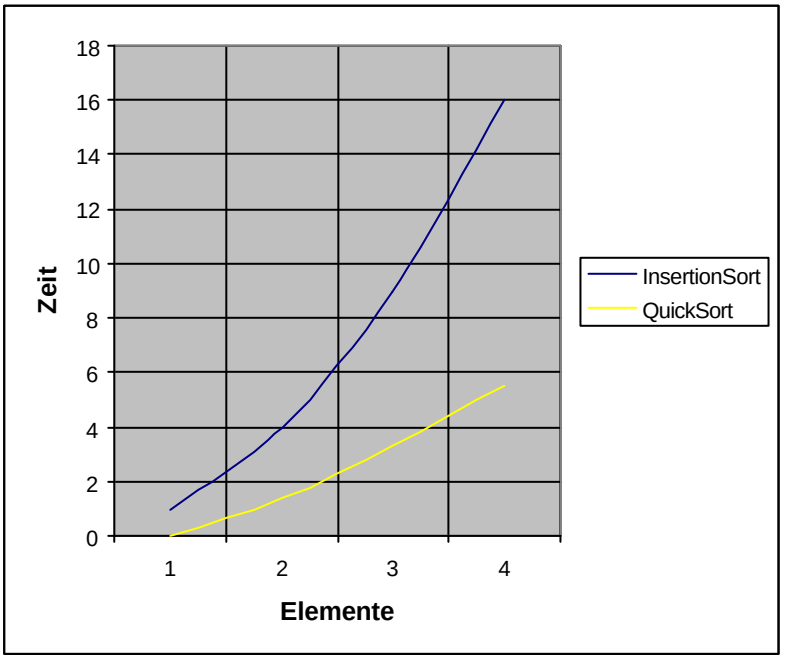

Abbildung 8 - Anstieg des Zeitaufwands bei unsortierten Feldern

In beiden Abbildungen sieht man den quadratischen Anstieg. In Abbildung 7 wird die lineare Steigung von InsertionSort deutlich. QuickSort steigt bei

unsortierten Feldern (Abbildung 8) etwas stärker als linear, ist aber immer noch erheblich schneller als InsertionSort.
Da die Implementation für QuickSort kompliziert ist, lohnt sich beim Sortieren von wenigen kleinen Dateien der Einsatz nicht. Hier ist InsertionSort effektiver, da die Implementation und auch die Beseitigung von Fehlern zu viel Zeit benötigen. Dann lohnt sich InsertionSort – selbst wenn das Feld unsortiert ist.

c. Einsatzgebiete

Es ist nun klar, in welchen Fällen InsertionSort bzw. QuickSort eingesetzt werden sollten:

InsertionSort	QuickSort
fast sortierte Felder	unsortierte Felder
wenige kleine Dateien	

Tabelle 8 – theoretische Einsatzgebiete

Nun kann man Beispiele herleiten, in welchen Fällen in der Praxis welcher Algorithmus eingesetzt werden sollte.
InsertionSort ist zum Beispiel bei der Sortierung von Telefonbüchern geeignet, da ein großer Teil schon sortiert ist und nur noch (am Ende) neu eingefügte Datensätze an die richtige Stelle gesetzt werden müssen. Das Gleiche gilt für bereits existierende Lexika, die lediglich um einige neue Stichwörter erweitert werden. Auch für das Aktualisieren von Sporttabellen (Fußball etc.) ist InsertionSort geeignet, weil sich die Plätze der Mannschaften nicht sehr stark verschieben und damit die Tabelle eigentlich schon fast sortiert ist.
QuickSort ist geeignet, wenn man nicht weiß, ob das Feld bereits sortiert ist, und man somit von einer zufälligen Permutation ausgehen muss. Hier gibt es unendlich viele Möglichkeiten. Beispielsweise könnte eine Bank ihre Konten, die bislang nach Kontonummer geordnet sind, nach dem Guthaben ordnen wollen, um so wichtige Kunden von weniger lukrativen leicht unterscheiden zu können. Hier ist das Guthaben unabhängig von der Kontonummer, das Guthaben ist also zufällig verteilt.

InsertionSort	QuickSort
Telefonbücher	diverse Spezialaufgaben wie Banken (s. beschriebener Fall)
Lexika	
Sporttabellen	

Tabelle 9 - praktische Einsatzgebiete

d. Abschließende Bewertung

Abschließend kann man sagen, dass es verschiedene Algorithmen gibt, die zwar das gleiche Problem lösen, aber erhebliche Geschwindigkeitsunterschiede aufweisen. Außerdem gibt es verschiedene Algorithmen, die nur in bestimmten Fällen geeignet sind.
So ist QuickSort im Durchschnitt eindeutig schneller, aber in bestimmten Fällen ist InsertionSort trotzdem unschlagbar. **Eine eindeutige Stellungnahme** als abschließenden Vergleich, welcher der beiden

Algorithmen nun besser ist – was im Allgemeinen schneller bedeutet – **ist nicht möglich**.
Für die Praxis gilt daher, dass man nicht irgendeinen Algorithmus wählen sollte, sondern das Problem zuvor genau analysieren muss. Daraufhin kann man dann den am besten geeigneten Algorithmus aussuchen.

Literaturverzeichnis

Saake, Gunter. Algorithmen und Datenstrukturen : eine Einführung mit Java. 1. Auflage. Heidelberg: dpunkt-Verlag, 2002

Neues großes Lexikon in Farbe. o. O.: Buch und Zeit, 1995

Sedgewick, Robert. Algorithmen in C++. 1. Auflage. Bonn, München, Paris: Addison-Wesley, 1992

Ottmann, Thomas; Widmeyer, Peter. Algorithmen und Datenstrukturen. 3. Auflage. Berlin, Oxford: Spektrum, 1996

Gierhardt, Horst. Sortieren durch Zerlegen (Quicksort) [online]. Oktober 2001. http://www.informatik.gierhardt.de/info12/Quicksort.pdf. 20.01.2002

Kaneva, Bilian; Thiébaut, Dominique. Sorting Algorithms [online]. 1997. http://cs.smith.edu/~thiebaut/java/sort/demo.html. 20.01.2002

Meinel, Christoph. Effiziente Algorithmen. Entwurf und Analyse. 1. Auflage. Leipzig: Fachbuchverlag, 1991

Sahni, Sartaj. Data structures, algorithms, and applications in C++. Singapore: McGraw-Hill Book, 1998

Baase, Sara. Computer algorithms. Introduction to Design and Analysis. 2. Auflage. o. O.: Addison-Wesley Publishing Company, o. J.

Wegener, Ingo. Effiziente Algorithmen für grundlegende Funktionen. Mit zahlreichen Aufgaben und Beispielen. Stuttgart: Teubner, 1989

Güting, Ralf Hartmut. Datenstrukturen und Algorithmen. Stuttgart: Teubner, 1992

Goos, Gerhard. Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen. Berlin, Heidelberg, New York: Springer, 1996

Erklärung:

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.