Vergleich von Sortierverfahren

Seminar: Schreiben wissenschaftlicher Texte im Bereich Informatik

Sebastian Böttger

Knowledge and Data Engineering Group, Electrical Engineering & Computer Science, University of Kassel

Kurzfassung Gegenstand dieser Arbeit ist der Vergleich der Sortierverfahren Radixsort, Mergesort und Quicksort. Alle drei Algorithmen werden zunächst ausführlich vorgestellt und ihre Funktionsweise erläutert. Dabei wird die Komplexität und das resultierende Laufzeitverhalten theoretisch betrachtet und die daraus entstehenden Vor- und Nachteile diskutiert. In einem Versuch werden zudem alle drei Algorithmen unter realen Bedingungen auf einem Computer getestet, um die theoretischen Betrachtungen zu untermauern.

1 Einleitung

Eine häufige Aufgabe in der Informatik ist das Suchen von Daten. Weil dem Menschen das Suchen nach Telefonnummern in Telefonbüchern leichter fällt, werden dessen Einträge sortiert. Wie dem Menschen geht es auch den Computern. In vielen Fällen ist es für den Rechner leichter gesuchte Datensätze zu finden, wenn diese in sortierter Form vorliegen. Weitere Motive Daten zu sortieren sind das Zusammenbringen von zusammengehörigen Dingen oder das Auffinden von Duplikaten.

1.1 Problemdefinition

Formal betrachtet soll eine Folge von Objekten nach einer vorgegebenen Ordnung sortiert werden. Dazu betrachten wir folgende N Datensätze

$$R_1, R_2, \ldots, R_N$$

Jeder Datensatz R_j besitzt einen zugehörigen *Schlüssel* K_j, der den Sortierungsprozess regelt. In einem Datensatz können auch zusätzliche Informationen neben dem Schlüssel enthalten sein, die auf die Sortierung jedoch keinen Einfluss haben.

Die Ordnungs-Relation '$<$' wird auf den Schlüsseln angegeben, so dass für alle drei Werte a, b, c die folgenden Bedingungen erfüllt sind:

1. Genau eine der folgenden Aussagen muss wahr sein: $a < b$, $a = b$, $b < a$

2. Wenn $a < b$ und $b < c$ dann $a < c$ (Transitivität)

Das Ziel einer Sortierung ist es letztlich, eine Permutation $p(1)p(2)\ldots p(N)$ der Datensätze zu bestimmen, welche die Schlüssel in eine nicht abnehmende Reihenfolge bringt:

$$K_{p(1)} \leq K_{p(2)} \leq \ldots \leq K_{p(N)}. \tag{1}$$

Eine Sortierung nennt man stabil, wenn die Reihenfolge der Datensätze, deren Schlüssel gleich sind, während der Sortierung bewahrt bleibt.

$$p(i) < p(j) \text{ für alle } K_{p(i)} = K_{p(j)} \text{ und } i < j. \tag{2}$$

1.2 Komplexitätsanalyse

Die tatsächliche Laufzeit eines Algorithmus wird von vielen Faktoren beeinflusst. Ein wesentlicher Einflussfaktor ist die Geschwindigkeit des Computers, auf dem der Algorithmus ausgeführt wird. Dennoch lassen sich allgemeine Aussagen über die Effizienz eines Programmes machen, indem dessen Komplexität formalisiert wird. Die sogenannte O-Notation wird dafür verwendet. Diese drückt den Anstieg des Zeitaufwandes bei steigendem n aus. $O(n^2)$ besagt demnach, dass die Laufzeit quadratisch zu n wächst. Verglichen mit der Laufzeit $O(n \log n)$ wird auf einen Blick deutlich, dass der Algorithmus mit quadratischer Laufzeit deutlich langsamer ist.

In dieser Arbeit werden die Algorithmen an Hand ihrer Zeitkomplexität untersucht und diese mit konkreten Laufzeiten verglichen. Eine detaillierte Einführung in die Komplexitätsanalyse von Algorithmen ist in [Cor10] zu finden. Da bei Sortieralgorithmen die Schlüsselvergleiche gegenüber den Vertauschungen dominieren, kann die Laufzeitanalyse auf die Anzahl der Vergleiche reduziert werden.

2 Stand der Technik

Die Ursprünge heutiger Sortiertechniken liegen im 19. Jahrhundert. Zu dieser Zeit wurden die ersten Maschinen hierfür erfunden. Knuth schreibt in [Knu75], dass der junge Mitarbeiter Herman Hollerith vom Zensus-Büro eine elektrische Tabelliermaschine und Sortiermaschine entwickelte, um eine bessere statistische Erfassung zu ermöglichen. 100 dieser Maschinen wurden erfolgreich eingesetzt, um die Volkszählung von 1890 durchzuführen. Da die US-Bevölkerung zu dieser Zeit weiter unaufhaltsam zu wachen schien, waren die ursprünglichen Tabulator-Sortierer nicht mehr schnell genug, um die Volkszählung von 1900 zu behandeln. Hollerith entwickelt eine andere Maschine um das Problem zu lösen. Sein neues Gerät hatte eine automatische Karten-Zufuhr [Tru65].

Diese Technik wird heute noch bei nicht-vergleichsbasierten Sortierverfahren angewandt. Genau genommen ist Holleriths Sortiermaschine die Basis von dem

in digitalen Computern verwendeten Radixsort (vgl. 3.1) und ähnlichen distributiven Sortierverfahren. Voraussetzung ist, dass die Schlüssel der zu sortierenden Daten nur aus Zeichen eines endlichen Alphabets bestehen. Zusätzlich muss eine Totalordnung zwischen den Zeichen des Alphabets bestehen (z. B. bei Zahlen aus $\mathbb{N}$ mit Ordnung $a_1 \leq a_2 \leq \ldots \leq a_k$, bei Buchstaben des lateinischen Alphabets $a < b < c... < z$ oder wie beim Kartenspiel $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$). Im wesentlichen besteht die Idee des verteilten Sortierens aus zwei Schritten. Die Partitionierung teilt die Daten auf Fächer auf und anschließend werden im Schritt 2 die Daten aus den Fächern wieder aufgesammelt. Zunächst werden alle Daten aus dem Fach mit der niedrigsten Wertigkeit eingesammelt, wobei die Reihenfolge der darin befindlichen Elemente nicht verändert werden darf. Danach werden die Elemente des nächsthöheren Faches eingesammelt und an die schon aufgesammelten Elemente angefügt. Dies wird fortgeführt, bis alle Fächer wieder leer sind. Die beiden Schritte werden für jede 10er-Stelle der Schlüssel (bei deziamlen Schlüsseln) wiederholt, wobei mit der letzten Stelle begonnen wird und in der letzten Iteration die erste Stelle verwendet wird.

Die Idee des Mischens (*merging*) geht zurück auf eine spätere Erfindung (1938), den IBM Collator. Mit seinen zwei Kartenzufuhren, konnte er zwei sortierte Kartendecks zu einer Einheit in nur einem Durchgang mischen [Knu75].

Die Entwicklung der Computer ging Hand in Hand mit der Entwicklung der Sortiertverfahren. Laut [Knu70] gibt es Hinweise darauf, dass eine Sortierroutine das erste Programm war, das jemals für eine speicherbaren Computer geschrieben wurde. Die Designer von EDVAC[1], darunter John von Neumann, waren besonders an Sortierverfahren interessiert, weil diese das Potential von nicht-numerischen Anwendungen eines Rechners verkörpern. Von Neumann entwickelte für den EDVAC eine Implementierung von Mergesort. Es setzte sich die Erkenntnis durch, dass Maschinen, die anspruchsvolle mathematische Verfahren ausführen können, auch in der Lage sein müssen, Daten zu sortieren und zu klassifizieren. 1946 beschrieb John Mauchly Insertionsort und Binary Insertionsort, sowie die Beobachtung, dass das erste Verfahren im Durchscnitt über $^{N^2}/_4$ Vergleiche benötigt, während letzteres nie mehr als ca. $n \log n$ Vergleiche braucht, jedoch der Zugriff auf die dafür benötigte komplexe Datenstruktur diesen Geschwindigkeitsvorteil wieder einbüßt. Mauchly gründete später zusammen mit J. P. Eckert die Firma Eckert-Mauchly Computer Corporation (EMCC), die den BINAC[2] und später den UNIVAC[3] entwickelten.

Neue Rechenmaschinen ermöglichten neue Sortierverfahren. In den 50er Jahren folgten nun auch vermehrt theoretische Ansätze. Zu erwähnen ist in diesem Zusammenhang die Arbeit von E. H. Friend [Fri56]. Er entwickelte umfangreiche Beschreibungen von etlichen internen und externen Sortieralgorithmen, wo

[1] EDVAC – Electronic Discrete Variable Automatic Computer: Die erste Rechenmaschine nach der Von-Neumann-Architektur

[2] BINAC: Binary Automatic Computer. Ein direkter Nachfolger der ENIAC, der auf das binäre Zahlensystem setzte

[3] UNIVAC: Universal Automatic Calculator. Nutzte erstmals als externen Speicher ein Magnetband.

er ein besonderes Augenmerk auf Pufferungstechniken und die Eigenschaften von Magnetbändern legte. Er führte neue Sortier-Methoden ein und entwickelte die mathematischen Eigenschaften der älteren Sortiermodelle. Daniel Goldenberg machte für fünf verschiedene Sortierverfahren und Best- und Worst-case-Analyse. Howard B. Demuth legte 1956 in seiner Doktorarbeit das Fundament für die Theorie zur Komplexitätsberechnung [Knu75].

In den frühen 60er Jahren folgten noch einige bemerkenswerte Errungenschaften. 1962 veröffentliche C. A. R. Hoare seinen legendären Quicksort-Algorithmus [Hoa62], der bis heute Maßstab für gute Sortierverfahren ist. 1964 folgte Williams Heapsort [Wil64].

In den 80er und 90er Jahren wurden Ideen für parallele Sortierverfahrenen für Computer mit mehreren CPUs diskutiert. Gegenwärtig scheint die (Grundlagen-)Forschung an Sortierverfahren größtenteils abgeschlossen zu sein. Heute wird insbesondere noch zum Sortieren mit riesigen Datenmengen im Datenbankbereich geforscht. Auch Verfahren die die speziellen Eigenschaften von GPUs nutzen und dadurch schnelleres Sortieren ermöglichen sind noch Gegenstand der Forschung [GGKM06].

3 Algorithmen

Wie im vorherigen Kapitel gezeigt, gibt es eine ganze Reihe von Sortieverfahren. Die meisten von ihnen sind jedoch für ein bestimmtes Sortierproblem entwickelt. Einen Sortieralgorithmus der sich für alle Anwendungsfälle als optimal erweist, gibt es so nicht.

Sortierverfahren lassen sich unterschiedlich einteilen. Eine Unterscheidung ist die zwischen interne und externe Sortierverfahren. Die Anwendung eines internen Sortierverfahrens setzt voraus, dass die zu sortierende Menge an Elementen in den Hauptspeicher des Computers passt. Es ist also notwendig, jederzeit auf jedes Element der Menge zugreifen zu können. Vertreter interner Sortierverfahren sind beispielsweise HeapSort, QuickSort und BubbleSort. Im Gegensatz dazu sind externe Sortierverfahren in der Lage Teilmengen zu sortieren und diese späterr in die Restmengen einzusortieren. Externe Sortierverfahren finden Verwendung in großen Datenbanksystemen, bei denen die Größe der Datenmengen im Terrabyte-Bereich liegt. Ein Algorithmus der zum externen Sortieren geeignet ist, ist MergeSort. Fast alle externen Sortierverfahren basieren auf dem Prinzip des separaten Sortierens und des anschließenden Mischens.

Eine weitere Möglichkeit der Unterscheidung ist, ob es ein vergleichendes Sortierverfahren ist. Zu diesen Vertretern gehört Quicksort, Mergesort, Heapsort, aber auch InsertionSort und BubbleSort. All diese Verfahren eint die Eigenschaft, dass die sortierte Reihenfolge, die sie bestimmen, auf Schlüsselvergleichen basiert. Zu den *nicht*-vergleichenden Verfahren gehört Countingsort, Radixsort und Bucketsort [Cor10].

In den folgenden Abschnitten werden Radixsort, Mergesort und Quicksort gegenüber gestellt, die Funktionsweise verglichen und deren Laufzeiten sowohl theoretisch als auch durch die Evaluation praktischer Versuche verglichen.

3.1 Radixsort

Radixsort oder auch Distributionsort genannt, ist ein stabiles, out-of-place Sortierverfahren mit linearer Laufzeit.

Funktionsweise Die Grundidee von Radixsort, hat seine Ursprünge, wie bereits in Kapitel 2 erwähnt, Ende des 19., Anfang des 20. Jahrhunderts. Der Algorithmus fand zu dieser Zeit Verwendung in Lochkartensortiermaschinen. Eine Lochkarte war in 80 Spalten organisiert. In jeder Spalte gab es 12 Plätze, in denen ein Loch gestanzt werden konnte. Die Maschine konnte so programmiert werden (mechanisch), dass eine gegebene Spalte für alle Karten untersucht werden konnte. In Abhänigkeit des Platzes des gestanzten Loches der gerade untersuchten Spalte, wurde die Karte in eine der 12 Kästen sortiert [Cor10]. Wenn eine Zahl d Stellen hat, wird dieser Schritt d mal für alle d Stellen der Zahlen ausgeführt. Begonnen wird mit der geringsten Wertigkeit, also der letzten Stelle. Darauffolgend die vorletzte Stelle usw. Bei der ersten Stelle angekommen, ist das Ergebnis eine sortierte Folge von Zahlen. Das Prinzip kann nur funktionieren, wenn für gleiche Ziffern einer Wertigkeit, die relative Ordnung der jeweils kleineren Wertigkeit beibehalten wird. Abbildung 1 zeigt eine siebenelementige Folge mit dreistelligen Zahlen. Jede Spalte zeigt die Reihenfolge der Folge nach dem jeweiligen Sortierschritt, wobei die erste Spalte (Spalte 0) die unsortierte Ausgangsfolge ist. Betrachten wir die Zahlen 234 und 231. In Spalte 0 kommt die 234 vor 231, die Reihenfolge ist also falsch. Im ersten Sortierschritt wird 231 über die 234 geschoben, da die letzte Stelle kleiner ist ($1 < 4$). Beide Zahlen sind jetzt, bezüglich ihrer letzten Stelle, in der richtigen Reihenfolge. Diese relative Ordnung bleibt für die folgenden Schritte erhalten. Sortierschritt 2 ordnet die Zahlen nach ihrer 10er-Stelle. Die 231 und 234 folgen nun aufeinander. Die Sortierung der beiden Zahlen nach ihrer letzten Stelle bleibt allerdings erhalten. Da auch die 100er-Stelle beider Zahlen identisch ist, ändert sich im Schritt 3 zwischen beiden Zahlen nichts mehr. Die Ordnung der jeweiligen Ziffern ist also stabil. Diese Eigenschaft garantiert die Funktionsfähigkeit von Radixsort.

Radixsort funktioniert auch in anderen Zahlensystemen. So können damit auch bequem binäre Zahlen sortiert werden. Das Funktionsprinzip ist exakt das gleiche, jedoch werden nur 2 Fächer benötigt – für 0 und 1. Dafür werden mehr Sortierschritte gebraucht. Angenommen der Wertebereich der zu sortierenden Zahlen liegt zwischen 0 und 255; das entspricht einer 8-Bit Zahl ($256 = 2^8$). Um nun nach jeder Ziffer/Stelle zu sortieren werden also acht Durchläufe benötigt um eine Folge von 8-Bit-Zahlen zu sortieren.

Laufzeit Sind n d-stellige Zahlen gegeben, dann sortiert Radixsort in linearer Laufzeit. In d Schritten zur Aufteilung auf die Fächer wird jede Zahl jeweils einmal „angefasst". Daraus ergibt sich die Laufzeit $\Theta(d \cdot n)$.

Vor- und Nachteile Zunächst erweckt die lineare Laufzeit von Radixsort den Anschein, dass dieser Algorithmus sehr schnell und deswegen vergleichsbasierten Algorithmen vorzuziehen ist. Da bei kleinem d die Laufzeit besser ist, als

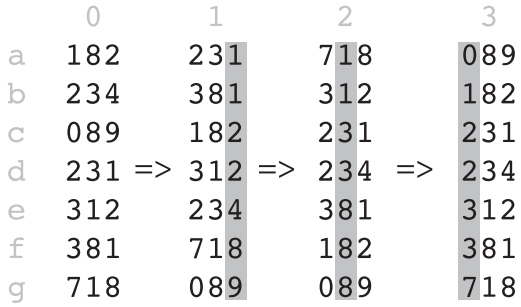


Abb. 1: Funktionsweise von Radixsort auf einer Liste mit sieben dreistelligen Zahlen. Die erste Spalte ist die unsortierte Folge, die folgenden Spalten sind die Ergebnisse des jeweiligen Sortierungsschrittes.

die Laufzeit der meisten rekursiven vergleichsbasierten Algorithmen, die meist eine mittlere Laufzeit von $\Theta(n \log n)$ haben. Allerdings unterscheiden sich „die in der O-Notation versteckten konstanten Faktoren"[Cor10]. Radixsort führt zwar weniger Läufe über die Schlüssel durch, aber jeder einzelne Lauf benötigt eine signifikat längere Zeit im Vergleich zu Quicksort. Ob Radixsort einem vergleichenden Sortierverfahren vorzuziehen ist, hängt auch von den Eingabedaten ab. Datumsangaben werden beispielsweise oft in der Form `YYYY-MM-DD` gespeichert und nicht als 32- oder 64-Bit-Zahl (Unixzeit). Um diese zu sortieren, muss also zuerst nach dem Jahr (`YYYY`), dann nach dem Monat (`MM`) und zuletzt nach dem Tag (`DD`) sortiert werden. Die einzelnen Teile eines Datums werden also als Stellen einer Zahl betrachtet. Ein vergleichender Algorithmus würde an dieser Aufgabe scheitern. Er würde die Sortierung nach Jahr möglicherweise wieder zerstören, während er nach Monaten sortiert. Die Eigenschaft, dass das Sortieren nach den Stellen stabil ist, prädestiniert Radixsort für solche Aufgaben.

Zu den Nachteilen von Radixsort gehört die Tatsache, dass sich negative Zahlen nicht sortieren lassen. Negative Ganze Zahlen werden mit einem Vorzeichen-Bit dargestellt, das die richtige Aufteilung in Kästen unmöglich macht. Ein weiterer Nachteil ist, dass Radixsort out-place arbeitet. Wenn geringer Verbrauch von Hauptspeicher ein Kriterium für das Sortierproblem ist, dann ist ein in-place Verfahren vorzuziehen.

3.2 Mergesort

Mergesort (zu deutsch: Sortieren durch Mischen) ist ein stabiler, auf Schlüsselvergleichen basierender, out-place arbeitender Sortieralgorithmus. Er wurde 1945 von John von Neumann als ein Programm für den EDVAC vorgestellt (vgl. Kapitel 2).

Funktionsweise Mergesort verdankt seinen Titel seiner Funktionsweise und lässt sich anhand eines Kartenspiels gut erklären. Zwei aufgedeckte Kartenstapel

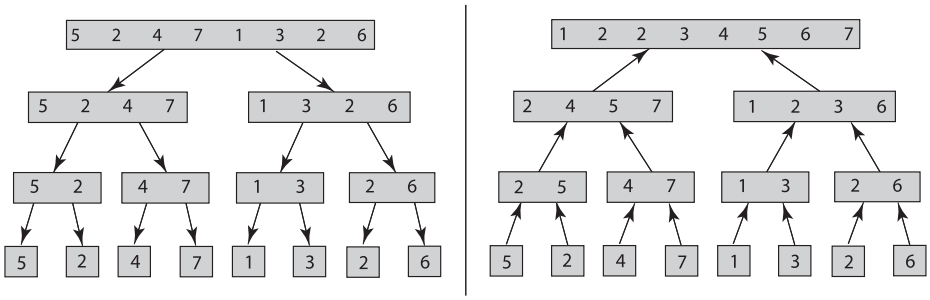


Abb. 2: Beispiel Mergesort: Der linke Teil zeigt die rekursive Aufteilung in n Teilfolgen, der rechte Teil das Mischen zweier Teilfolgen in eine für jede Rekursion.

liegen auf dem Tisch und sollen zu einem sortierten Stapel zusammengemischt werden.[4] Unter der Voraussetzung, dass beide Ausgangsstapel in sich sortiert sind und die kleinste Karte oben liegt, lässt sich diese Aufgabe intuitiv lösen indem die jeweils kleinere Karte von den beiden Stapeln mit dem Gesicht nach unten auf einen neuen Stapel gelegt wird. Der Schritt wird solange wiederholt bis einer der beiden Stapel aufgebraucht ist. Die übrig gebliebenen Karten des anderen Stapels werden einfach umgedreht auf den neuen Stapel gelegt. Um nun vorsortierte Teilstapel zu bekommen, wird der Ursprungsstapel zunächst solange in der Mitte geteilt, bis er aus n 1-elementigen Teilstapeln besteht. Eine Folge aus nur einem Element ist trivialerweise immer in sich sortiert. Nun kann Schritt für Schritt das sortierte Zusammenmischen durchgeführt werden, bis nur noch ein Stapel übrig ist. Abbildung 2 zeigt das Prinzip mit Hilfe der unsortierten Folge $\langle 5, 2, 4, 7, 1, 3, 2, 6\rangle$.

Der Algorithmus teilt sich demnach in zwei wesentliche Schritte auf:

1. Das rekursive Zerlegen der zu sortierende Folge in n 1-elementige Teilfolgen
2. Das Mischen jeweils zweier Teilfolgen zu einer in sich sortierten Teilfolge

Dieses Vorgehen wird auch Divide and Conquer (Teile und Beherrsche) genannt, das eine allgemeine algorithmische Methode ist, durch die versucht wird durch Eigenschaften der Rekursion Geschwindigkeitsvorteile zu erzielen. Um ein gegebenes Problem zu lösen, ruft sich eine Funktion ein- oder mehrmals selber auf, um das eine große Problem in viele kleine Probleme zu zerlegen. Das kleine Teilproblem lässt sich leichter lösen als das große Gesamtproblem. Die Teillösungen werden dann kombiniert, um die Gesamtlösung herzustellen. In unserem Beispiel ist der Vorteil, dass 1-elementige Folgen automatisch sortiert sind und nur noch richtig zusammengefügt werden müssen. Es macht also durchaus Sinn die Folge so aufzuteilen.

Algorithmus 1 zeigt die Arbeitsweise in Pseudocode. Die Funktion MERGESORT zerlegt die Folge A in jeweils zwei möglichst gleich große Teile, einen

[4] Der Begriff Mischen ist als Übersetzung ins Deutsche nicht ganz korrekt, da im Deutschen unter Karten mischen eher das zufällige anordnen von Karten verstanden wird.

Algorithmus 1 Mergesort Algorithmus

```
 1: function MERGESORT(A, p, r)
 2:     if p < r then
 3:         q ← ⌊(p + r)/2⌋
 4:         MERGESORT(A, p, q)
 5:         MERGESORT(A, q + 1, r)
 6:         MERGE(A, p, q, r)
 7:     end if
 8: end function
 9: function MERGE(A, p, q, r)
10:     n_1 ← q − p − 1
11:     n_2 ← r − q
12:     erzeuge L_{1...n_1+1} und R_{1...n_2+1}
13:     for i ← 1 to n_1 do
14:         L_i ← A_{p+i−1}
15:     end for
16:     for j ← 1 to n_2 do
17:         R_j ← A_{q+j}
18:     end for
19:     L_{n_1+1} ← ∞
20:     R_{n_2+1} ← ∞
21:     for k ← p to r do
22:         if L_i ≤ R_j then
23:             A_k ← L_i
24:             i ← i + 1
25:         else
26:             A_k ← R_j
27:             j ← j + 1
28:         end if
29:     end for
30: end function
```

linken $p...q$ und einen rechten $q+1...r$ indem der Index q ermittelt wird (Zeile 3). Anschließend ruft sich die Funktion selbst auf (Zeile 4 und 5) um die beiden Teilfolgen in weitere Teilfolgen zu zerlegen. Die Funktion MERGE wendet einen Trick an, damit nicht in jedem Schritt überprüft werden muss, ob man bereits an dem Ende einer Teilfolge angekommen ist. Dafür wird der Wert ∞ verwendet. Dieser Wert kann niemals der kleinere sein (es sei denn beide Teilfolgen haben das Element ∞ erreicht). So wird sichergestellt, dass die verbleibenden Elemente der anderen Folge an die neue Folge angehängt werden. Das eigentliche Mischen passiert in der Schleife von Zeile 21 bis 29. Die If-Bedingung in Zeile 22 stellt fest ob das aktuelle Element von der linken oder von der rechten Folge auf die neue Folge gelegt werden soll. Wenn genau $r-p+1$ Elemente in der neuen Folge abgelegt worden sind, kann der Algorithmus stoppen [Cor10].

Laufzeit Da Mergesort stabil ist, ist seine Laufzeit, sowohl im Worst-, Best als auch im Average-Case, stets gleich. Für die Laufzeit gilt die Rekursionsformel:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n\,.$$

Durch Lösen des Mastertheorems ergibt sich eine Laufzeit von $T(n) \in \Theta(n \log n)$.

Vor- und Nachteile Da Mergesort – wie auch Radixsort – out-place arbeitet, eignet sich der Algorithmus für große Datenmengen, die nicht in den Hauptspeicher passen. Die zu verschmelzenden Listen können direkt von einem externen Speicher, wie einer Festplatte, geladen werden. Da die Elemente der Listen sequentiell abgearbeitet werden, eignet sich Mergesort gut zur Sortierung von Datenstrukturen wie verketteten Listen.

3.3 Quicksort

Der Algorithmus QuickSort wurde von C. A. R. Hoare 1962 vorgestellt [Hoa62]. Hoare beschäftigte sich damals mit maschineller Übersetzung natürlicher Sprachen. Er wollte Sätze aus dem Russischen in das Englische übersetzen. Da das Wörterbuch, alphabetisch sortiert, auf einem Magnetband vorlag, war es notwendig, die Wörter des zu übersetzenden Satzes ebenfalls alphabetisch zu sortieren. Auf diese Weise hat er versucht alle Wörter eines Satzes bei einem Durchlauf des Bandes zu übersetzen [Shu09].

Quicksort ist – genauso wie Mergesort – ein auf Schlüsselvergleichen basierender Algorithmus, der ebenfalls auf dem Prinzip Teile und Beherrsche (Divide and Conquer) aufbaut und somit rekursiv arbeitet. Der Algorithmus zerlegt in jedem Rekursionsschritt die Ausgangsfolge in zwei Teile und sortiert diese unabhängig von einander. Im Unterschied zu Mergesort wird allerdings das Verschmelzen vermieden. Stattdessen werden Elemente innerhalb der Folge vertauscht (swap). Dadurch ist sichergestellt, dass Quicksort in-place arbeitet.

Funktionsweise Bei der Zerlegung wird die Folge nicht wie bei Mergesort in der Mitte geteilt, sondern an ein zu wählendes Referenzelement – Pivot[5] genannt – zerlegt. Alle Elemente der linken Teilfolge sind kleiner als das Pivot und alle Elemente der rechten sind analog dazu größer. Das Pivot kann, je nach Umsetzung, beliebig gewählt werden. Im hier betrachteten Beispiel wird das erste Element einer Folge als Pivot gewählt.

Algorithmus 2 zeigt den Ablauf. Zunächst wird das Pivotelement gewählt. Danach werden die Zeiger initialisiert: l wird auf Position 1 gesetzt, r auf das letzte Element der Folge. In Zeile 7 bis 9 wird der Zeiger l solange nach rechts verschoben, bis er auf ein Element der Folge zeigt, das größer oder gleich dem Pivot ist. In Zeile 10 bis 12 wird analog dazu der Zeiger r solange nach links verschoben, bis er auf ein Element zeigt, das kleiner oder gleich dem Pivot ist. Nun

[5] Pivot: franz. – Dreh- oder Angelpunkt

Algorithmus 2 Quicksort Algorithmus

```
 1: function QUICKSORT(K)
 2:     if |K| > 1 then
 3:         p ← K_0
 4:         l ← 0
 5:         r ← |K| − 1
 6:         while l ≤ r do
 7:             while K_l < p do
 8:                 l ← l + 1
 9:             end while
10:             while K_r > p do
11:                 r ← r − 1
12:             end while
13:             if l ≤ r then
14:                 SWAP(K_l ↔ K_r)
15:                 l ← l + 1
16:                 r ← r − 1
17:             end if
18:         end while
19:         QUICKSORT(K_{0...r})
20:         QUICKSORT(K_{r+1...|K|−1})
21:     end if
22: end function
```

werden die beiden Elemente ausgetauscht, wenn sich l und r noch nicht gekreuzt haben (ab Zeile 13). Das Element das kleiner ist als das Pivot steht nun links, das größere rechts vom Pivot. Die Schleife läuft solange bis sich beide Zeiger kreuzen. Anschließend ruft sich Quicksort rekursiv selbst auf. Die linke Hälfte wird nun rechts begrenzt durch r, die rechte Hälfte links durch $r + 1$ (Zeile 19 und 20). Die SWAP-Funktion bedient sich einer Hilfsvariablen die den einen Wert temporär zwischenspeichert. Am Ende einer Rekursion sind die Elemente zwar noch nicht sortiert, aber linksseitig in jedem Fall immer kleiner oder gleich dem Pivot und rechtsseitig immer größer oder gleich dem Pivot. Durch das rekursive Zerlegen der Folge in immer kleinere Teilfolgen, entsteht dann die Sortierung der Gesamtfolge. Abbildung 3 zeigt wie der Quicksort-Algorithmus die unsortierte Folge in immer unterschiedlich große Teilprobleme zerlegt. In jedem Schritt werden Elemente durch das Vertauschen am Pivot umgeordnet. Schritt für Schritt wird sich so der sortierten Folge angenähert.[6]

Laufzeit Die Laufzeit von Quicksort ist davon abhängig ob die Zerlegung balanciert oder unbalanciert ist, also die zerlegten Teilfolgen in jeder Rekursion etwa gleichgroß sind. Im besten Fall ist die Aufteilung so, dass die eine Folge

[6] Eine sehr detailierte Präsentation der Funktionsweise und des Wirkens der Rekursion lässt sich online unter `http://www.bluffton.edu/~nesterd/java/SortingDemo.html` finden.

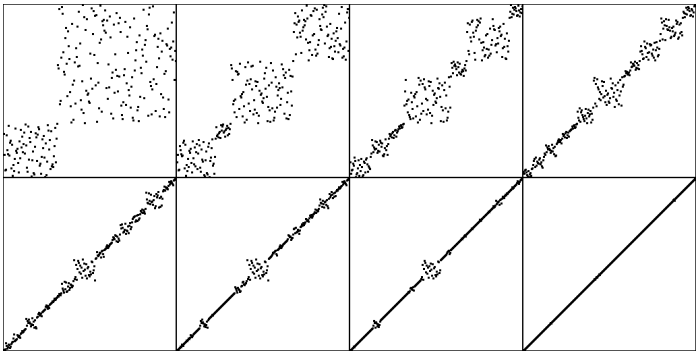


Abb. 3: Die Zahlen von 1 bis n in zufälliger Reihenfolge werden mit Quicksort sortiert. Grafik: Deutsche Wikipedia, User: Odif. Lizenz: Creative Commons by-sa 3.0 http://creativecommons.org/licenses/by-sa/3.0/de/legalcode

die Größe $\lfloor n/2 \rfloor$ und die andere die Größe $\lceil n/2 \rceil - 1$ hat. Die Rekursionsgleichung für die Laufzeit lautet dann $T(n) \leq 2T(n/2) + \Theta(n)$. Wegen Fall 2 des Mastertheorems ergibt sich die Laufzeit $T(n) \in \Theta(n \log n)$ [Cor10].

Der schlechteste Fall tritt auf, wenn die Zerlegung ein Teilproblem mit $n-1$ Elementen und eines mit 0 Elementen erzeugt. Wenn diese unbalancierte Zerlegung in jedem Rekursionsschritt auftritt, ist die Laufzeit $\Theta(n^2)$. Die Zerlegung kostet die Zeit $\Theta(n)$. Da der rekursive Aufruf auf einem Feld der Größe 0 nichts tut, sondern nur in die aufrufende Routine zurückkehrt, gilt $T(0) = \Theta(1)$. Daraus ergibt sich folgende Rekursionsgleichung:

$$T(n) = T(n-1) + \Theta(n)\,.$$

Werden die Kosten auf alle Rekursionsschritte aufsummiert, entsteht eine geometrische Reihe, die $\Theta(n^2)$ ergibt [Cor10].

Der Durchschnittliche Fall, also die mittlere Laufzeit, ist viel näher am besten Fall dran, als im schlechesten. Der Grund dafür besteht darin, dass jede Aufteilung mit konstanten Verhältnis auf ein Rekursionsbaum der Tiefe $\Theta(\log n)$ führt. Die Kosten in jeder Ebene sind $\Theta(n)$. Die Laufzeit verhält sich asymptotisch immer $\Theta(n \log n)$ bei konstantem Verhältnis der Aufteilung [Cor10].

Um festzustellen wie eine balancierte Folge aussieht, muss man sich vor Augen führen, welche Bedinung erfüllt sein müssen, damit Quicksort die Folge in zwei etwa gleichgroße Teile aufspaltet. Das kann nur dann geschehen, wenn das Pivot der Median der jeweiligen (Teil-)Folge ist. Wenn die Folge bereits auf- oder absteigend sortiert ist, ist das erste Element immer das kleinste oder das größte der jeweiligen Folge. Wenn dann das erste oder letzte Element als Pivot gewählt worden ist, folgt daraus, dass Quicksort diese in eine 0-elementige und eine $(n-1)$-elementigen Folge aufspaltet. Damit ist das Laufzeitverhalten von Quicksort bei bereits vorsortierten oder annähernd sortierten Folgen $\Theta(n^2)$ und damit schlechter als das Worst-Case-Verhalten von Radix- oder Mergesort.

Vor- und Nachteile Ein Nachteil von Quicksort ist seine Instabilität, die durch das Vertauschen entsteht. Das kann aber auch ein Vorteil sein, da Quicksort aus selben Grund in-place arbeitet und kaum zusätzlichen Speicher benötigt. Die quadratische Laufzeit bei bereits sortierten oder fast sortierten Folgen ist problematisch. Auch wenn es auf dem ersten Blick für die Praxis wenig relevant erscheint, gibt es durchaus Situation wo ein solches Szenario auftreten kann. Wenn beispielsweise einige wenige Elemente an eine bereits sortierte Folge von Werten angehängt werden und diese wieder sortiert werden muss, ist das ein durchaus plausibler Fall.

In der mittleren Laufzeit ist Quicksort bisher ungeschlagen, da er pro Rekursion verhältnismäßig wenige Schritte benötigt und c dadurch, im Vergleich zu Mergesort sehr klein ist.

4 Experimente

Nach den anfänglichen Betrachtungen haben wir nur ein theoretisches Bild von der zu erwartenden Laufzeit der Sortierverfahren. Im folgenden Abschnitt wird der Versuchsaufbau erläutert, mit dem die Algorithmen im praktischen Einsatz getestet wurden. Darauffolgend werden die Ergebnisse diskutiert und mit den theoretischen Überlegungen verglichen.

4.1 Versuchsaufbau

Wie in Kapitel 3 gezeigt, haben alle drei Verfahren je nach Anwendungsfall ihre Vor- und Nachteile. Um dennoch eine einheitliche Vergleichsbasis zu schaffen, wurde folgendes Versuchsumfeld erstellt:

Gemessen wird die Laufzeit für unterschiedlich große Listen. Gestartet wird bei $n = 2.500$ in 2.500er-Schritten bis einschließlich 250.000.

Um einen aussagekräftigen Mittelwert zu erhalten wird außerdem jede Meßreihe 500 mal wiederholt.

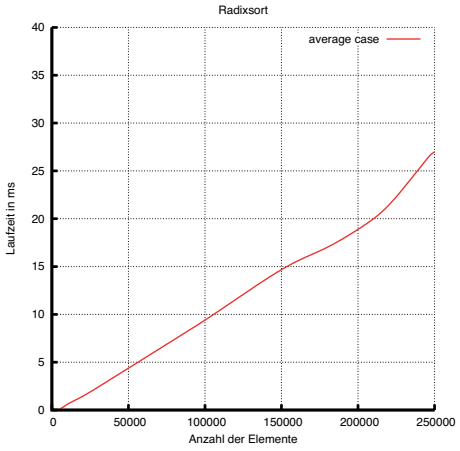
Für alle Experimente wurden Java-Programme geschrieben, die die hier vorgestellten Algorithmen umsetzen. Vor Aufruf und nach Ablauf des jeweiligen Algorithmus wird mittels der Java-Funktion `System.currentTimeMillis()` die Zeit ermittelt, so kommt durch Feststellung der Differenz aus beiden Werten die Laufzeiten zustande. Die Zufallsfolgen werden durch eine Java-Funktion erstellt, die zunächst ein Array der Größe n erstellt und mittels

$$z_{end} = z \cdot (r_{max} - r_{min}) + r_{min} \text{ wobei } z \in \mathbb{R} \text{ und } 0 \leq z < 1$$

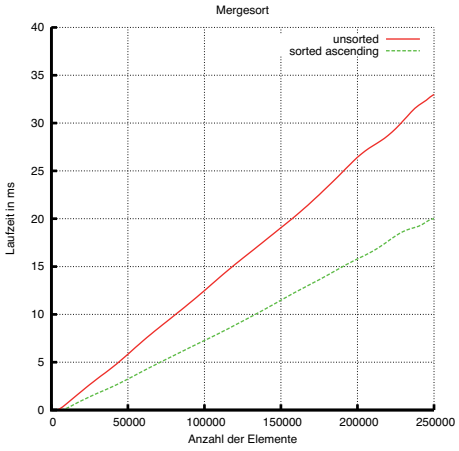
Zufallswerte zwischen r_{min} und r_{max} erstellt. Die Zufallszahl zwischen 0 und 1 (z) liefert die Java-Funktion `Math.random()`).

4.2 Vergleich der einzelnen Algorithmen

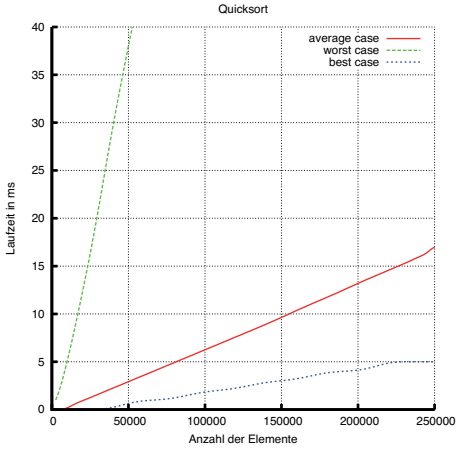
Laufzeitverhalten von Radixsort Da das Laufzeitverhalten von Radixsort in allen Fällen gleich ist, zeigt Abbildung 4a nur den Average-Case an. Wie erwartet ergibt sich asymptotisch eine Gerade. Die Laufzeit ist somit linear.



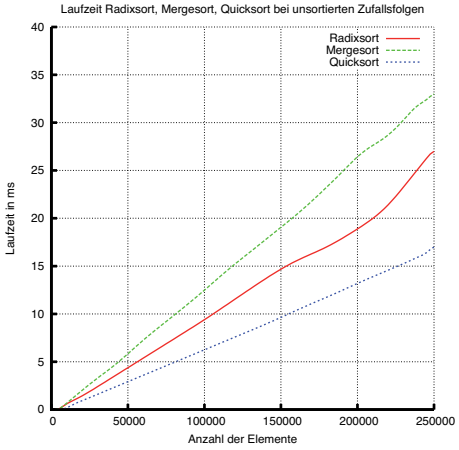
(a) Laufzeit Radixsort



(b) Laufzeit Mergesort



(c) Laufzeit Quicksort



(d) Laufzeit bei unsortierten Zufallsfolgen

Abb. 4: Ergebnisse der gemessenen Laufzeiten aller drei Algorithmen im Vergleich

Laufzeitverhalten von Mergesort Für Mergesort wurde das Laufzeitverhalten bei unsortierten Zufallsfolgen mit sortierten verglichen. Abbildung 4b zeigt, dass Mergesort bei zweiterem geringfügig schneller ist. Auch hier entsteht der Eindruck, dass sich das Laufzeitverhalten von Mergesort linear verhält. Das ist möglicherweise dem relativ geringem n verschuldet. Bei größeren Folgen sollte die Dämpfung im Anstieg zu sehen sein.

Laufzeitverhalten von Quicksort Um Average-, Worst- und Best-Case für Quicksort messen zu können, waren zusätzliche Überlegungen notwendig. Für den durchschnittlichen Fall wurden, wie oben beschrieben, unsortierte Zufallsfolgen erstellt. Der Worst-Case wurde konstruiert, indem die Zufallsfolgen sortiert wurden. Da das linkeste Element als Pivot gewählt wird, ist dies auch immer das kleinste. Dadurch entsteht für die Rekursion der ungünstigste Fall (vgl. Abschnitt 3.3 Laufzeit). Das für die Laufzeit günstigste Pivot-Element ist der Median einer Folge. Um diesen Fall (Best-Case) zu konstruieren, wurde eine aufsteigend sortierte Folge mit dem mittleren Element als Pivot gewählt.

Dass Quicksort ein durchaus ambivalentes Laufzeitverhalten aufzeigt, haben die theoretischen Betrachtungen bereits gezeigt. Die Experimente haben diese Einschätzung bestätigt. Abbildung 4c zeigt den deutlichen Unterschied zwischen dem Best- und Worst-Case. Es kann aber auch herausgelesen werden, dass der Average-Case deutlich näher am Best-Case ist, als am Worst-Case. Die Kurve für den Worst-Case deutet, wie zu erwarten war, eine Parabel an, während beim Best-Case die Kurve eine logarithmische Dämpfung erkennen lässt. Die scheinbar lineare Laufzeit im Average-Case lässt sich auch hier wieder mit dem geringen n der Experimente erklären.

Laufzeitvergleich bei unsortierten Zufallsfolgen Abbildung 4d zeigt die Ergebnisse der Messungen für unsortierte Zufallsfolgen aller Sortierverfahren im direkten Vergleich. Hier zeigt sich, dass Quicksort bei beliebigen n immer schneller ist als seine Konkurrenten und Mergesort analog dazu den dritten Platz auf dem Siegertreppchen einehmen muss.

5 Fazit

Trotz schlechter Laufzeit kann die Tatsache, dass Mergesort out-place arbeitet der entscheidene Vorteil für ihn gegenüber Quicksort sein. Wenn große Mengen sortiert werden müssen, kann Mergesort die Daten von einem externen Speicher nachladen. Quicksort arbeitet in-place und muss alle Daten in den Hauptspeicher laden. Für Datenmengen im Terrabyte-Bereich ist das momentan unmöglich.

Überzeugende Laufzeiten zeigte Radixsort. Seine Eigenschaft, dass er für alle Stellen stabil bleibt, macht ihn einsetzbar für die Sortierung nach mehr als ein Kriterium.

Sowohl die theoretischen Betrachtungen als auch die praktischen Versuche haben gezeigt, dass Quicksort der effizienteste Algorithmus ist. Die erhobenen Daten untermauern aber auch seine vergleichsweise katastrophale Laufzeit im Worst-Case. Der Grund hierfür ist das Pivot-Element, das – durch seine Position in der Eingabefolge und je nach Anordnung der Elemente in der Eingabefolge – das Laufzeit-Verhalten des Algorithmus extrem verändern kann. Um dieses Problem zu lösen gibt es eine Vielzahl von Optimierungen. Ein Beispiel ist, dass ein zufälliges Pivot gewählt wird. Das verkleinert die Wahrscheinlichkeit, dass bei einer sortierten Folge der höchste oder niedrigste Wert als Pivot gewählt wird

[Cor10]. Mit diversen Tricks kann auch versucht werden, in jeder Folge ein Pivot zu wählen, das möglichst nah am Median liegt. Eine Idee ist, drei Elemente aus der Folge zu wählen und das mittlerste als Pivot zu verwenden. Auch eine Kobination von unterschiedlichen Verfahren ist ein häufig verwendeter Lösungsansatz. Quick- und Insertionsort lassen sich gut kombinieren, da Insertionsort bei einer sortierten Folge lineare Laufzeit hat [Sed78].

Literatur

[Cor10] Thomas H. Cormen. *Algorithmen - eine Einführung*. Oldenbourg, München, 3., überarb. und erw. aufl. edition, 2010.

[Fri56] Edward H. Friend. Sorting on electronic computer systems. *J. ACM*, 3(3):134–168, 1956.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[Hoa62] C. A. R. Hoare. Quicksort., 1962.

[Knu70] Donald E. Knuth. Von neumann's first computer program. *ACM Comput. Surv.*, 2(4):247–260, 1970.

[Knu75] Donald Ervin Knuth. *The art of computer programming: Sorting and searching*. Addison-Wesley series in computer science and information processing. Addison-Wesley, Upper Saddle River, NJ [u.a.], 2. print. edition, 1975.

[Sed78] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[Shu09] Len Shustek. An interview with c.a.r. hoare. *Communications of the ACM*, 52(3), 2009.

[Tru65] Leon E. Truesdell. The development of punch card tabulation in the bureau of the census, 1890-1940 : with outlines of actual tabulation programs, 1965.

[Wil64] John William Joseph Williams. Algorithm 232: heapsort. *Communications of the ACM*, 7(6):347–348, 1964.