



Referat

Uwe Krause

Naives vs. komplexes Sortieren

Uwe Krause

naives vs. komplexes Sortieren

Referat eingereicht im Rahmen der Vorlesung Algorithmen und Datenstrukturen

im Studiengang Angewandte Informatik (AI)
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. C. Klauck

Abgegeben am 27.04.2017

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Einführung | 5 |
| 1.1 Relationen | 5 |
| 1.2 Abbildungen | 6 |
| 1.3 Von der Iteration zur Rekursion | 7 |
| 1.3.1 Verwandtschaft von Rekursion und Iteration | 7 |
| 1.3.2 divide-and-conquer | 8 |
| 1.3.3 Rekursivierung: von Iteration zu Rekursion | 8 |
| 1.4 Von Arrays zu verketteten Listen | 8 |
| 1.5 Operationen in verketteten Listen | 9 |
| 1.5.1 Einfügen | 9 |
| 1.5.1.1 An den Anfang | 9 |
| 1.5.1.2 Ans Ende | 9 |
| 1.5.1.3 In eine sortierte Reihenfolge | 9 |
| 1.5.2 Ermitteln / Entnehmen | 10 |
| 1.5.2.1 Elemente an einer bestimmten Position | 10 |
| 1.5.2.1.1 Erstes Element | 10 |
| 1.5.2.1.2 Letztes Element | 10 |
| 1.5.2.1.3 Element an mittlerer Position | 11 |
| 1.5.2.1.4 Median3 | 11 |
| 1.5.2.1.5 Zufälliger Wert | 11 |
| 1.5.2.2 Element mit bestimmtem Wert innerhalb einer Ordnung | 11 |
| 1.5.2.2.1 Minimalwert | 11 |
| 1.5.2.2.2 Maximalwert | 11 |
| 1.6 Sortieralgorithmen | 12 |
| 1.6.1 Naive Verfahren | 12 |
| 1.6.2 Komplexe Verfahren | 12 |
| 1.7 Effizienz | 12 |
| 2 Insertionsort | 13 |
| 2.1 Ablauf des Codebeispieles mit Array | 13 |
| 2.2 Kernelemente | 14 |
| 2.3 Ablauf in einer rekursiven Datenstruktur | 14 |
| 3 Selectionsort | 14 |
| 3.1 Ablauf des Codebeispieles mit Array | 15 |
| 3.2 Kernelemente | 15 |
| 3.3 Ablauf in einer rekursiven Datenstruktur | 15 |
| 4 Quicksort | 16 |
| 4.1 Ablauf des Codebeispieles mit Array | 17 |
| 4.2 Kernelemente | 18 |
| 4.3 Ablauf in einer rekursiven Datenstruktur | 19 |

| | |
|--|-----------|
| 5 Mergesort | 19 |
| 5.1 Ablauf des Codebeispiels mit Array | 20 |
| 5.2 Kernelemente | 21 |
| 5.3 Ablauf in einer rekursiven Datenstruktur | 22 |
| 6 Testumgebung | 23 |
| 6.1 Zweck und Verfahren | 23 |
| 6.2 Vergleichsdaten | 23 |
| 6.2.1 Verfahren und Strategien | 23 |
| 6.2.2 Ausprägung der zu verarbeitenden Daten | 23 |
| 6.2.3 Problemgröße | 24 |
| 6.2.4 Anzahl der Testdaten | 24 |
| 6.3 Einschränkung der Interpretation | 24 |
| 7 Testergebnisse | 24 |
| 7.1 Laufzeitkomplexität | 25 |
| 7.2 Wechselgröße Quicksort | 27 |
| 8 Anhang | 28 |
| 8.1 Quellenangaben | 28 |
| 8.2 Erklärung zur schriftlichen Ausarbeitung des Referates | 29 |

1 Einführung

Im Zuge der Informationsverarbeitung ist das Ordnen von Datenbeständen wie beispielsweise das Sortieren eines Adressbuches oder eines Bildarchives eine Aufgabe, die immer wieder zu erledigen ist.

Bereits 1973 berichtete Donald E. Knuth von Schätzungen von Computerherstellern, die aussagen, dass mehr als 25 Prozent der Laufzeit fürs Sortieren von Datenbeständen aufgewandt wird¹.

Daraus ließe sich schließen, dass es entweder

1. viele wichtige Anwendungsfälle fürs Sortieren gibt, oder
2. viele Leute unnötigerweise sortieren, oder
3. ineffiziente Sortieralgorithmen weit verbreitet sind.

Die Wahrheit beinhaltet vermutlich Teile aller drei dieser Möglichkeiten, daher ist es notwendig und praktisch, sich mit der Frage auseinanderzusetzen, wie man Datenbestände effizient nach festgelegten Kriterien sortieren kann.

1.1 Relationen

Damit Elemente sortiert ("in eine Ordnung gebracht") werden können, müssen sie miteinander vergleichbar sein, also in Relation zueinander gestellt werden.

Grundlegend wird für das Sortieren von Elementen die binäre Ordnungsrelation "kleiner als" $<$ benötigt, mit der sich eine totale Ordnung der Elemente definieren lässt, wenn folgende Bedingungen erfüllt sind²:

1. Von zwei Elementen ist das eine entweder kleiner als oder gleich wie das andere oder das andere ist kleiner. (Trichotomie)

$$\forall a, b \in M : [(a < b) \vee (a = b) \vee (b < a)]$$
2. Wenn ein Element kleiner als ein anderes und das andere kleiner als ein drittes ist, folgt daraus, dass das erste Element auch kleiner als das dritte ist (Transitivität)

$$\forall a, b, c \in M : [(a < b) \wedge (b < c) \rightarrow (a < c)]$$

Solche Relationen lassen sich für die üblichen Zahlenräume basierend auf der Addition³ der natürlichen Zahlen \mathbb{N} definieren⁴.

¹ Frei übersetzt nach: Knuth, Donald E. (1973): The art of computer programming. Sorting and searching. Amsterdam (Addison-Wesley). S.3

² ebd.

³ Die Ordnung der rationalen Zahlen benötigt zusätzlich die Multiplikation der ganzen Zahlen, aber diese Ausarbeitung beschränkt sich auf die Betrachtung des Raumes der ganzen Zahlen.

⁴ Deiser, Oliver / Lasser, Caroline / Vogt, Elmar / Werner, Dirk (2010): 12 x 12 Schlüsselkonzepte zur Mathematik. Berlin u.a. (Springer-Verlag).

Eine natürliche Zahl n ist kleiner als eine andere Zahl m , wenn eine dritte Zahl k , die nicht Null ist, existiert, für die die Summe der Zahl n und der dritten Zahl k die andere Zahl m ergibt.

$$\forall n, m, k \in \mathbb{N} : [n < m \mid \exists k \neq 0 : (n + k = m)]$$

Wenn eine Zahl aus dem Bereich der ganzen Zahlen \mathbb{Z} als ein Paar von zwei natürlichen Zahlen dargestellt wird, dessen Gesamtwert sich aus der Differenz der ersten (Minuend) und der zweiten Zahl (Subtrahend) ergibt⁵, lässt sich die "kleiner als" Relation für den Bereich der ganzen Zahlen auf der Grundlage der natürlichen Zahlen definieren, indem eine ganze Zahl in der {Minuend, Subtrahend} Darstellung als kleiner als eine andere ganze Zahl in dieser Darstellung erkannt wird, falls die Summe des Minuenden der ersten Zahl und des Subtrahenden der zweiten Zahl kleiner ist, als die Summe des Minuenden der zweiten Zahl und des Subtrahenden der ersten Zahl.

$$\forall m, s, m_2, s_2 \in \mathbb{N} : [\{m, s\} <_{\mathbb{Z}} \{m_2, s_2\} \mid (m + s_2) <_{\mathbb{N}} (m_2 + s)]$$

Wenn in den zu sortierenden Elementen identische Elemente vorkommen können, muss zusätzlich die "gleich" Relation = definiert werden.

Wenn sich bei zwei zu vergleichende Elementen schließen lässt, dass sie gleich sind, wenn weder das eine, noch das andere kleiner ist, kann die "kleiner" Relation bereits ausreichen.

$$\forall a, b \in \mathbb{N} : [a = b \mid \neg(a < b) \wedge \neg(b < a)]$$

Ansonsten wird es notwendig, zusätzlich die Identität⁶ hinzuzuziehen.

$$\text{Id}_M = \{(x, x) \mid x \in M\}$$

Die hier nicht genannten Ordnungsrelationen ("ungleich", "größer als", "größer als oder gleich") setzen sich jeweils aus einer Kombination dieser Relationen zusammen.

So auch die "kleiner als oder gleich" Relation \leq , welche nur noch eine totale Quasiordnung, also eine nicht eindeutige Sortierung ermöglicht, dafür aber den Vergleich identischer Elemente erlaubt.

$$x \leq y \Leftrightarrow (x < y \vee x = y)$$

1.2 Abbildungen

Bei Vergleichen beispielsweise der Reihenfolge der Buchstaben des deutschen Alphabetes, der Wertigkeit der Farben eines Kartenspiels (\clubsuit , \diamond , \heartsuit , \spadesuit) oder der Schönheit eines Urlaubslandes, stehen die Elemente in keiner direkten Ordnungsrelation zueinander.

Um derartige oder andere Elemente trotzdem miteinander vergleichen zu können, muss eine Reihenfolge definiert werden.

⁵ -5 kann ohne Nutzung des Vorzeichens zum Beispiel aus den beiden natürlichen Zahlen 0 und 5 oder 10 und 15 zusammengesetzt werden, -7 als {0,7} oder {5, 12} oder {399, 406}, jedenfalls immer so, dass die Differenz des Paares natürlicher Zahlen dem Wert der ganzen Zahl entspricht.

⁶ "Für jede Menge M ist $\text{Id}_M = \{(x, x) \mid x \in M\}$ eine reflexive, symmetrische und transitive Relation auf M , die Identität auf M ." Zitiert aus Deiser, u.a.: 12 x 12 Schlüsselkonzepte zur Mathematik.

Für die Buchstaben des Alphabetes⁷ oder die Farben des Kartenspiels lässt sich eine eindeutige Reihenfolge per Definition festlegen. Die anfangs beispielhaft genannten zu sortierenden Adressen bestehen aus Zahlen und Buchstaben, welche mit Hilfe der festgelegten Ordnungsrelationen sortiert werden können.

Für komplexere Elemente bietet es sich an, sie wiederum in eine Relation zu setzen, welche sie auf Elemente abbildet, für die bereits eine Ordnung definiert ist.

Für das angesprochene Bildarchiv müssen andere Sortierkriterien definiert werden, welche aber (ggf. über Umwege) letztlich auch auf Elemente abgebildet werden, für die eine Ordnung fest definiert ist. Beispielsweise ließe sich eine linkstotale, rechtseindeutige Abbildung auf die natürlichen Zahlen definieren, welche eine Gewichtung vorgibt, die nicht der alphabetischen Sortierung entspricht.

$$f: \{\text{"Frankreich"}, \text{"Mexiko"}, \text{"Schweden"}\} \mapsto \{1, 2, 3\}$$

1.3 Von der Iteration zur Rekursion

1.3.1 Verwandtschaft von Rekursion und Iteration

“Grundsätzlich gilt, dass man jeden rekursiven in einen iterativen Algorithmus und jeden iterativen in einen rekursiven Algorithmus transformieren kann und dass sich für jede Aufgabenstellung auf mehr oder weniger natürliche Art und Weise eine iterative und eine rekursive Lösungsidee entwickeln lässt. [...]

Trotz dieser aus der gegenseitigen Ersetzbarkeit begründeten Verwandtschaft von Rekursion und Iteration gibt es gravierende Unterschiede zwischen beiden Konzepten.

Die wesentlichen Vorteile rekursiver gegenüber iterativer Algorithmen sind:

- rekursive algorithmische Lösungen sind oft natürlicher und einfacher zu finden als iterative, insbesondere bei rekursiven Aufgabenstellungen,
- die Korrektheit rekursiver Lösungen ist oft einfacher zu prüfen (z.B. mittels mathematischer Induktion) als die iterativer Lösungen, und
- rekursive Lösungen sind im Allgemeinen statisch kürzer und - auch weil verständlicher - änderungsfreundlicher

Doch hat die Rekursion gegenüber der Iteration auch Nachteile, insbesondere sind dies:

- die geringere Effizienz und
- der höhere Speicherbedarf (im Laufzeitkeller).⁸

⁷ Üblicherweise wird hier die Position in der ASCII-Tabelle verwendet, was auf den ersten Blick zu verwunderlichen Ergebnissen führen kann, da die Großbuchstaben in dieser vor den Kleinbuchstaben liegen. In einem derart sortierten Wörterbuch wäre “Zahl” vor “addieren” zu finden.

⁸ Pomberger, G., Dobler, H.: Algorithmen Und Datenstrukturen. München: Pearson Studium, 2008.

1.3.2 divide-and-conquer⁹

Generell können große Probleme angegangen werden, indem man sie solange in kleinere Probleme zerlegt, bis die Lösung für ein kleines Problem trivial ist und die Lösung des nächsthöheren Problems darauf aufbauend gefunden werden kann.

- Ein Maß für die Größe des Problems muss bestimmt werden.
- Für kleine Größen dieses Problems (häufig 1 oder 2) lässt sich meist eine einfache Lösung finden
- Größere Probleme lassen sich lösen, indem das größere auf das kleinere Problem zurückgeführt wird.

Die kleineren Probleme werden rekursiv gelöst, die Ergebnisse werden dann nur noch zur Lösung des Problems zusammengesetzt.

Speziell für die Sortieralgorithmen wird dieses Thema später noch einmal aufgenommen.

1.3.3 Rekursivierung: von Iteration zu Rekursion

Jeder Algorithmus, der Schleifen enthält kann auch in ein System rekursiver Algorithmen transformiert werden.

Allgemein kann eine While-Schleife¹⁰ durch einen rekursiven Algorithmus, der sich selbst aufruft, ersetzt werden, indem er prüft, ob die Bedingung noch gültig ist, eine Aktion durchführt und sich anschließend wieder selbst aufruft.

Eine andere Strategie kann es sein, das iterativ gelöste Problem auf seine Kernelemente zurückzuführen und diese Kernelemente dann rekursiv zu lösen.

Aus den Eigenarten der Rekursion können sich andere Datenstrukturen entwickeln, deren Stärke rekursive Aufgabenstellungen sind.

1.4 Von Arrays zu verketteten Listen

Die in der Aufgabenstellung vorgegebenen Algorithmen sind mithilfe von Feldern (Arrays) realisiert. Eine andere erfolgversprechende Datenstruktur basiert auf dem Prinzip der einfach verketteten Liste, die hier mit einigen grundlegenden und für Sortieralgorithmen wichtigen Operationen vorgestellt und mit Arrays verglichen werden.

Eindimensionale Felder erlauben es über einen Index direkt auf die Inhalte des Feldes an einer bestimmten Position zuzugreifen. Vorteilhaft ist, dass die hierfür benötigte Zeit unabhängig von der Position des Elementes innerhalb des Feldes ist.

Der große Nachteil an diesen Feldern ist jedoch, dass das Entfernen oder Hinzufügen eines Elementes großen Aufwand verursacht. Damit kein Wert überschrieben wird und verloren geht und die Reihenfolge innerhalb des Feldes trotzdem gleich bleibt, müssen die Elemente,

⁹ Rimscha, M.: Algorithmen kompakt und verständlich : Lösungsstrategien am Computer. 3. Aufl.. Berlin Heidelberg New York: Springer-Verlag, 2014.

¹⁰ Jede andere Schleife lässt sich zu einer While-Schleife umformen.

die hinter der entfernten oder hinzugefügten Position liegen, jeweils um eine Position verschoben werden.

Die einfach verkettete Liste gibt den Daten eine andere Struktur, die ebenfalls eine Reihenfolge ihrer Elemente garantiert. Sie besteht aus einer sequenziellen Aneinanderkettung von Knoten, welche entweder ein Element und einen Verweis (link) auf einen Nachfolger enthalten oder leer sind. Ein leerer Knoten enthält kein Element und keinen Nachfolger und repräsentiert somit das Ende der verketteten Liste.

Die Nachteile der Felder entfallen, weil Knoten effizient entfernt oder hinzugefügt werden können, da nur die Verweise angepasst werden müssen. Ein Verschieben aller übrigen Elemente ist nicht notwendig.

Ein wesentlicher Nachteil der Listen ist jedoch, dass sie für diese Operationen immer von Anfang der Liste bis zu der Position, auf die zugegriffen werden soll, durchlaufen werden muss, was rechenintensiv ist und somit zeitraubend.

1.5 Operationen in verketteten Listen

1.5.1 Einfügen

Ein Element wird der Liste an einer bestimmten Position innerhalb der Reihenfolge hinzugefügt, die Länge der Liste erhöht sich also um 1.

Das Ergebnis der Operation ist jeweils die neue Liste mit dem hinzugefügten Wert.

1.5.1.1 An den Anfang

Die Liste wird erweitert, indem ein Knoten erstellt wird, der aus dem einzufügenden Element und einem Verweis auf die Liste besteht.

1.5.1.2 Ans Ende

Die Liste wird komplett durchlaufen und sobald der leere Knoten gefunden wird, welcher das Ende der Liste repräsentiert, wird hier ein neuer Knoten eingefügt, der das einzufügende Element enthält und auf den leeren Knoten verweist.

1.5.1.3 In eine sortierte Reihenfolge

Wenn die Elemente der Liste in einer festgelegten Reihenfolge sind (hier beispielsweise aufsteigend sortiert) und diese Reihenfolge beibehalten werden soll, werden Elemente nicht an eine vorgegebene Position einsortiert, sondern finden ihren Platz in der sortierten Liste, indem für jeden Knoten überprüft wird, ob das nachfolgende Element größer¹¹ als das einzufügende Element ist.

- Ist das nachfolgende Element höherwertig als das aktuelle Element, wird das Element an dieser Stelle wie vorher beschrieben eingefügt.

¹¹ für aufsteigende Sortierung

- Ist das nachfolgende Element nicht höherwertiger, wird im Rest der Liste weiter gesucht.
- Wurde das Ende der Liste erreicht, kann es kein höherwertiges Element mehr geben, der Platz des einzufügenden Elementes ist also am Ende der Liste und wird hier eingefügt.

1.5.2 Ermitteln / Entnehmen

Zum Ermitteln eines bestimmten Elementes muss die einfach verkettete Liste von Anfang an, Element für Element durchlaufen werden. Dabei muss jeder Knoten darauf geprüft werden, ob er die gewünschten Eigenschaften besitzt, bis der richtige ermittelt wurde.

Das Ergebnis ist jeweils eine Kombination aus

- dem Wert des Elementes des ermittelten Knoten
- der resultierenden Liste ohne den ersten Knoten, der den ermittelten Wert hat.

Für alle hier vorgestellten Entnahmemethoden wird ausdrücklich das erste Vorkommen des Wertes entfernt, welches, wenn Werte mehrfach vorkommen, sich auch in einem Knoten an einer anderen, als der ermittelten Position befinden kann. Für die hier vorgestellten Verfahren ist das nicht weiter wichtig, da die Werte immer nur aus einem sowieso unsortierten Bereich¹² entnommen werden.

Die Länge der Liste reduziert sich also um 1.

1.5.2.1 Elemente an einer bestimmten Position

Da verkettete Listen wie erwähnt keine festen Indizes haben, muss beim Zugriff auf bestimmte Positionen mitgezählt werden, wie oft die rekursive Funktion bereits aufgerufen wurde.

1.5.2.1.1 Erstes Element

Das erste Element der Liste wird entnommen.

(Ein Mitzählen der aktuellen Position kann hier entfallen.)

1.5.2.1.2 Letztes Element

Das Element, das an der Position der Länge der Liste steht, wird entnommen.

Anstatt die Elemente zu zählen, wird die Liste komplett durchlaufen, bis ein Knoten erreicht wurde, dessen Nachfolger der leere Knoten ist, der das Ende der Liste repräsentiert.

¹² Mit der geringfügigen Ausnahme eventueller Best-Case-Sortierungen mit mehrfachen Gleichen.

1.5.2.1.3 Element an mittlerer Position

Die Länge der Liste wird ermittelt, halbiert und der Wert des Elementes des Knotens an dieser Position wird ermittelt.

Das erste Vorkommen des Wertes wird entfernt, wie oben beschrieben.

1.5.2.1.4 Median3

Als Medianwert wurde definiert, dass das erste, das mittlere¹³ und das letzte Element der Liste untersucht werden. Ihre Werte werden miteinander verglichen und der Wert der in der Mitte¹⁴ liegt, wird als Median3-Wert festgehalten.

Das erste Vorkommen des Wertes wird entfernt, wie oben beschrieben.

1.5.2.1.5 Zufälliger Wert

Der Wert eines Elementes, welches mit einem Zufallszahlengenerator der eine Zahl zwischen 1 und der Länge der Liste ausgewählt gefunden wird, wird festgehalten.

Das erste Vorkommen des Wertes wird entfernt, wie oben beschrieben.

1.5.2.2 Element mit bestimmtem Wert innerhalb einer Ordnung

Soll ein bestimmtes Element abhängig von seinem Wert gefunden werden, muss der Wert jedes Knotens mit dem gesuchten Wert verglichen werden.

Entspricht der Wert nicht dem Kriterium, muss der Rest der Liste weiter durchlaufen werden.

Entspricht der Wert des aktuellen Knoten dem Kriterium, wurde das Element gefunden.

Bei sortierten Listen kann die Suche abgebrochen werden, wenn aus dem Wert des aktuell überprüften Elementes geschlossen werden kann, dass das gesuchte Element nicht mehr im ungeprüften Bereich der sortierten Liste enthalten sein kann.

1.5.2.2.1 Minimalwert

Wählt den kleinsten Wert aus der Liste aus, indem der erste Wert ausgewählt wird, der im Vergleich mit allen anderen Werten der kleinste ist.

1.5.2.2.2 Maximalwert

Wählt den größten Wert aus der Liste aus, indem der erste Wert ausgewählt wird, der im Vergleich mit allen anderen Werten der größte ist.

1.6 Sortieralgorithmen

Um zu verstehen, mit welchen Problemstellungen man allgemein bei der Entwicklung von Sortieralgorithmen konfrontiert wird, wie man diese löst und welche Eigenschaften daraus

¹³ mittlere Position innerhalb der Liste: a, X, c

¹⁴ mittlerer Wert: $a < X < c$

für die Algorithmen resultieren, reicht es, wenige Sortieralgorithmen beispielhaft kennenzulernen.

1.6.1 Naive Verfahren

Die naiven Verfahren sind diejenigen, die Menschen auch intuitiv zum Anordnen von Dingen (zum Beispiel Spielkarten in einer Hand) nutzen.

Das Insertion- und das Selectionsort-Verfahren gehören in diese Kategorie und werden in Teil 2 und 3 behandelt.

1.6.2 Komplexe Verfahren

Wie in dem Abschnitt zu dem Übergang von iterativen zu rekursiven Beschreibungen einer Problemlösung erwähnt, kann eine gute Strategie darin bestehen, das Problem solange in kleinere Teilprobleme aufzuteilen, bis das einfachste Problem, welches leicht zu beherrschen ist, erreicht wurde. Darauf aufbauend lässt sich dann die Lösung für das nächst größere Problem finden.

Diese Strategie lässt sich oft bei Suchalgorithmen anwenden:

- Die Anzahl n der zu sortierenden Elemente ist ein gutes Maß für die Größe des Sortierproblems.
- Im Fall $n = 1$ gibt es nichts zu tun, denn ein einziges Element ist immer sortiert.
- Eine Strategie kann daher sein, den zu sortierenden Bereich solange aufzuteilen, bis das Problem bei der Größe $n = 1$ angekommen ist.

Sowohl das Quicksort-, als auch das Mergesortverfahren arbeiten auf diesem Prinzip und werden in Teil 4 und 5 behandelt.

1.7 Effizienz

Die vorgestellten Verfahren unterscheiden sich in ihrer Effizienz, was dafür sorgt, dass bei gleichen zu sortierenden Datensätzen je nach verwendetem Algorithmus unterschiedliche Laufzeiten üblich sind. Die zu sortierende Datensätze können zusätzlich unterschiedliche Ausprägungen haben und für unterschiedliche Ausprägungen können jeweils andere Sortierverfahren für wiederum unterschiedliche Problemgrößen das schnellste Ergebnis liefern.

Daher lohnt es sich, in Teil 6 eine Testumgebung einzurichten, die es erlaubt, die vorgestellten Algorithmen unter fairen Bedingungen miteinander zu vergleichen, um in Teil 7 Testergebnisse vorzulegen, die es ermöglichen, für konkrete Anwendungsfälle den jeweils am besten geeigneten Algorithmus zu wählen.

2 Insertionsort

“Damit der Algorithmus von Quicksort verwendet werden kann, ist die Schnittstelle (ssort:insertionS(<Liste>)) einzuhalten. Technische Vorgabe: die Zahlen sind in der Erlang-Liste [] gehalten und dort zu sortieren. Der in der Vorlesung vorgestellte Algorithmus ist so auf die Verwendung von Listen (statt array) zu transformieren, dass das Kernkonzept erhalten bleibt! Die Begründung dazu ist im Code als Kommentar aufzuführen.”

Das Einfügesortieren ist eines der am einfachsten zu verstehenden Sortierverfahren, weil es auch von vielen Menschen für das systematische Ordnen von Gegenständen intuitiv verwendet wird.

Beispielsweise wird beim Kartenspielen eine neue Spielkarte in eine “Hand” bereits sortierter Spielkarten eingefügt, indem sie (von rechts oder links beginnend) solange mit den in der Hand befindlichen Karten verglichen wird, bis die richtige Position gefunden ist.

```
InsertionSort() {
    for ( int i = 2; i ≤ N ; i++ ) {
        int j = i;
        Datensatz t = a[i];
        int k = t.key;
        while(a[j-1].key > k) {
            a[j] = a[j-1];
            j = j-1;
        }
        a[j] = t;
    }
}
```

2.1 Ablauf des Codebeispielles mit Array

Der vordere Bereich des Arrays (alles vor Position i) wird als bereits sortiert betrachtet, der hintere Bereich ist noch unsortiert.

Das jeweils erste Element des unsortierten Bereiches (an der Position i) wird aus dem unsortierten Bereich herausgenommen, indem es in einer Hilfsvariablen t zwischengespeichert und anschließend in den bereits sortierten Bereich eingefügt wird, indem jedes Element, welches sich links der aktuellen Position befindet und einen höheren Wert als das Element im Zwischenspeicher hat, um eine Position nach rechts kopiert wird, um Platz für das einzufügende Element zu schaffen.

Wird hierbei ein Element gefunden, das einen niedrigeren Wert aufweist als das sich im Zwischenspeicher befindende Element, so wird das Element aus dem Zwischenspeicher an die Stelle rechts daneben kopiert.

2.2 Kernelemente

Unabhängig von einer konkreten Implementation lässt sich das Verfahren in folgende Schritte aufteilen:

Zu Beginn betrachtet man das erste Element des zu sortierenden Bereiches als bereits sortiert.

Der Rest der Liste wird sortiert, indem jedes folgende Element in den sortierten Bereich eingefügt wird. Das jeweils nächste Element wird in den bereits sortierten Bereich eingefügt, indem es mit jedem Element des sortierten Bereiches verglichen wird, bis die richtige Position gefunden wurde.

Das Entnehmen des Elementes ist einfach, der aufwändige Teil des Verfahrens stellt das Herausfinden der richtigen Position dar.

2.3 Ablauf in einer rekursiven Datenstruktur

Wie in der Einleitung geschildert ist es in einer rekursiven Datenstruktur (beispielsweise der Erlang-Liste als Implementation einer einfach verketteten Liste) nicht möglich (und auch nicht notwendig), mit Hilfe eines Index auf eine bestimmte Position innerhalb des Feldes (Array) zuzugreifen.

Die Liste wird aufgeteilt, indem für jeden Rekursionsschritt das erste Element entnommen wird.

Beim Erreichen des Endes der Liste (repräsentiert durch den leeren Knoten) wird der Beginn einer neuen Liste zurückgegeben, in welche anschließend die in den vorherigen Schritten abgetrennten Element wie in der Einleitung beschrieben eingefügt werden.

3 Selectionsort

“Damit der Algorithmus von Quicksort (oder auch den Tests) verwendet werden kann, ist die Schnittstelle (ssort:selectionS(<Liste>)) einzuhalten. Technische Vorgabe: die Zahlen sind in der Erlang-Liste [] gehalten und dort zu sortieren. Der in der Vorlesung vorgestellte Algorithmus ist so auf die Verwendung von Listen (statt array) zu transformieren, dass das Kernkonzept erhalten bleibt! Die Begründung dazu ist im Code als Kommentar aufzuführen.”

```
SelectionSort() {
    int minimum = 0;
    for ( int i = 1; i ≤ N-1 ; i++ ) {
        minimum = findMinStartingAt(i);
        swap(i,minimum);
    }
}
```

```

}

int findMinStartingAt(int i) {
    int min = i;
    for ( int j = i+1; j ≤ N; j++) {
        if(a[j].key < a[min].key)
            min = j;
    }
}

```

3.1 Ablauf des Codebeispiels mit Array

Der zu sortierende Bereich wird Element für Element durchlaufen.

Der kleinste Wert, der sich in der Liste hinter dem aktuell betrachteten Element befindet wird gesucht, indem der Rest der Liste Position für Position durchlaufen wird und sich die Position des kleinsten Wertes gemerkt wird. Hier wird zunächst der Wert des aktuellen Elementes als Minimum angenommen. Diese Annahme wird jedes mal korrigiert, wenn ein Wert gefunden wird, der geringer ist als die vorhergehende Annahme.

Das so ermittelte kleinste Element wird mit dem mit dem aktuellem getauscht, indem das kleinste Element aus dem Rest der Liste entnommen wird und das aktuelle Element an die vorherige Position des kleinsten Elementes verschoben wird.

3.2 Kernelemente

Unabhängig von einer konkreten Implementation lässt sich das Verfahren in folgende Schritte aufteilen:

Die Sortierte Liste wird aufgebaut, indem der kleinste Wert des unsortierten Bereiches ermittelt und an das Ende des sortierten Bereiches positioniert wird.

Im Gegensatz zum Insertionsort ist das Herausfinden der richtigen Position der einfache Teil, aufwändig ist das Ermitteln des kleinsten Elementes.

3.3 Ablauf in einer rekursiven Datenstruktur

Da es in der verketteten Liste keine festen Indizes gibt, ist es nicht notwendig, die Elemente miteinander zu tauschen.

Es reicht, die Liste aufzubauen, indem das kleinste Element der Liste entfernt¹⁵ und an die aktuelle Position gesetzt wird.

¹⁵ Wie in dem Abschnitt über die Listenoperationen erklärt wurde.

Mit dem Rest der Liste wird genauso verfahren, solange bis sich nur noch ein Element in dem unsortierten Bereich befindet (Abbruchbedingung).

4 Quicksort

“Technische Vorgabe: die Zahlen sind in der Erlang-Liste [] gehalten und dort zu sortieren. Der in der Vorlesung vorgestellte Algorithmus ist so auf die Verwendung von Listen (statt array) zu transformieren, dass das Kernkonzept erhalten bleibt! Die Begründung dazu ist im Code als Kommentar aufzuführen. Als Schnittstelle ist `ksort:qsort(<pivot-methode>,<Liste>,<switch-number>)` vorgegeben und einzuhalten. Dabei kann `<pivot-methode>` die Werte `left / middle / right / median / random` annehmen. Die Zahl `<switch-number>` entscheidet, ab welcher Länge Insertion Sort (oder Selection Sort) eingesetzt wird, d.h. Listen, die kürzer als diese Zahl sind, werden dann mit diesen Algorithmen sortiert und nicht mehr mit Quicksort.”

```
quickSort(int ilinks, int irechts) {
    if (ilinks < irechts) {
        int i = quickSwap(ilinks,irechts);

        quickSort(ilinks,i-1);
        quickSort(i+1,irechts);
    }
}

int quickSwap(int ilinks, int irechts) {
    int i = ilinks;
    int j = irechts-1;
    int pivot = a[irechts].key;

    while(i <= j) {
        while((a[i].key <= pivot) && (i < irechts)) i++;
        // a[i].key > pivot
        while((ilinks <= j) && (a[j].key > pivot)) j--;
        // a[j].key <= pivot
        if ( i < j ) swap(i,j);
    }

    swap(i,irechts); //Pivotelement in die Mitte tauschen
    return i;
}
```


4.1 Ablauf des Codebeispielles mit Array

In diesem Codebeispiel wird ein Feld mit Zugriffsmöglichkeiten auf einen Index (Array) mittels des Quicksort-Verfahren sortiert, indem der zu sortierende Bereich rekursiv in immer kleinere Bereiche aufgeteilt wird.

- Ein Bereich mit nur einem Element kann nicht weiter sortiert werden und wird daher nicht weiter aufgeteilt.
- Der letzte (rechteste)¹⁶ Wert des zu sortierenden Bereiches wird als Pivotelement¹⁷ ausgewählt.
- Die Größe des zu sortierenden Bereiches des Feldes wird solange reduziert, bis in dem zu sortierenden Bereich nur noch weniger als zwei Elemente verbleiben.
 - Ausgehend vom Anfang des zu sortierenden Bereiches bis zu seiner Begrenzung nach rechts wird die Positionen des ersten Elementes ermittelt, dessen Wert nicht kleiner oder gleich (also größer) als der des Pivotelementes ist.
 - Ausgehend vom Ende des zu sortierenden Bereiches bis zu seiner Begrenzung nach links wird die Position des (von rechts gesehen) ersten Elementes ermittelt, dessen Wert nicht größer (also kleiner oder gleich) als der des Pivotelementes ist.
 - Wenn sich die Positionen der auf diese Art ermittelten Werte nicht überkreuzt hat, (also die Position des linken Elementes nicht bereits die Grenze des rechten Bereiches überschritten hat) werden die auf diese Art ermittelten Werte miteinander getauscht.
 - (Die Elemente, für die das Auswahlkriterium nicht zutrifft, sind bereits auf der richtigen Seite und müssen daher nicht getauscht werden. Der Auswahlbereich verkleinert sich also jeweils um die Anzahl der Elemente, die dem Auswahlkriterium genügen.)
- Aus der Verschiebung der linken Grenze des (jetzt nicht mehr) zu sortierenden Bereiches ergibt sich eine gedachte Mitte¹⁸ zwischen den jeweils kleineren (oder gleichwertigen) Elementen, in die das Pivotelement, dessen Position ja außerhalb des ursprünglich zu sortierenden Bereiches liegt, eingefügt wird, indem es mit dem Wert an dieser Position getauscht wird.

¹⁶ oder der erste, ein mittlerer (ggf. durch Medianverfahren ermittelt) oder ein zufälliger Wert (je nach gewünschter Variante)

¹⁷ en. "pivot" = dt. "Drehpunkt" / "Achse". Alle Elemente werden abhängig vom Wert dieser Achse angeordnet.

¹⁸ Welche sich allerdings nur im Idealfall der gleichmäßigen Verteilung der zu sortierenden Daten tatsächlich in der Mitte zweier gleich großer Bereiche mit den jeweils größeren und kleineren Elementen befindet.

- Auf diese Art wird sichergestellt, dass sich alle Elemente, die kleiner als oder dem Pivotelement gleichwertig sind, in dem Bereich vor dem Pivotelement befinden, dessen Position jetzt die gedachte Mitte ist. Alle Werte, die größer sind, befinden sich in dem Bereich rechts von ihm.

Die Grenzen des Bereiches der kleineren (oder gleichen) Zahlen definieren sich also aus der ursprünglichen linken Grenze und dem Element links des Mittelpunktes. Der Bereich der größeren Elemente beginnt bei der Position des Elementes, das dem Mittelpunkt folgt und endet bei der ursprünglichen rechten Grenze. Diese beiden Bereiche werden nun jeweils auf dieselbe Weise behandelt, bis sie so lange zerlegt wurden, dass sie weniger als zwei Elemente beinhalten. Die Gesamtzahl der zu sortierenden Elemente (unabhängig von der Aufteilung in die jeweiligen Bereiche) reduziert sich um das Element, das den Mittelpunkt definiert.

Im günstigsten Fall wurde das Pivotelement so gewählt, dass beide Bereiche die gleiche Größe haben. Im ungünstigsten Fall ist einer der Bereiche leer, während alle Elemente (abzüglich des Mittelpunktes) in dem anderen Bereich liegen.

Der Aufwand der mit der Begrenzung des Feldes und der Verschiebung der Grenzen betrieben wird, wird in Kauf genommen, damit es möglich ist, immer auf dem selben Array zu arbeiten. Indem ein höherer Programmieraufwand investiert und die Grenzgebiete verwaltet werden (Rechenzeit), kann erreicht werden, dass die zu sortierenden Daten "in-place", also innerhalb des ursprünglichen Feldes sortiert werden. Der auf diese Art implementierte Quicksort-Algorithmus muss die Daten nicht kopieren und kommt daher auch unabhängig von der Größe der Eingangsdaten mit einem Speicherplatz aus, der nicht größer als das Feld der zu sortierenden Daten ist.

4.2 Kernelemente

Unabhängig von einer konkreten Implementation lässt sich das Verfahren in folgende Schritte aufteilen:

Für den zu sortierenden Bereich wird ein Wert (Pivotelement) ermittelt, der als gedachter Mittelwert¹⁹ fungiert. Dies kann beispielsweise dadurch geschehen, dass das erste, das letzte, ein mittleres (bezogen auf Position oder Wert beispielsweise durch das Median3-Verfahren) oder ein zufälliges Element des zu sortierenden Bereiches entnommen²⁰ wird.

Anschliessend wird der zu sortierende Bereich in zwei Partitionen aufgeteilt, wobei die eine Partition alle Elemente enthält, die kleiner als oder gleichwertig wie der Mittelwert sind und die andere alle Elemente enthält, die höherwertiger sind als der Mittelwert.

- Dies wird erreicht, indem alle Elemente des zu sortierenden Bereiches mit dem Pivotelement verglichen werden und anschließend in die Partition mit den kleineren oder gleichen, oder in die Partition mit den größeren Elementen verschoben werden.

¹⁹ welcher aber nur im Idealfall tatsächlich in der mitte liegt, wie bereits beschrieben.

²⁰ Der zu sortierende Bereich verkleinert sich um ein Element, siehe Beschreibung in der Einleitung.

Anschließend werden beide Partitionen jeweils auf die gleiche Weise aufgeteilt.

Die Gesamtzahl der zu sortierenden Elemente reduziert sich also um das Pivotelement.

Die Sortierung des Gesamtbereiches ergibt sich durch die Sortierungen der jeweils bis zum letzten Element aufgeteilten Partitionen.

4.3 Ablauf in einer rekursiven Datenstruktur

Da die Positionen innerhalb der verketteten Liste nicht entscheidend sind²¹, lässt sich der Algorithmus dahingehend vereinfachen, dass die Liste sortiert wird, indem sie in zwei Partitionen aufgeteilt wird, für die gilt, dass die Elemente, die kleiner als oder gleich wie der Pivotwert sind, in die eine Partition kommen und alle anderen Werte in die Partition der größeren Elemente.

Diese beiden Partitionen werden jeweils rekursiv auf die gleiche Art bearbeitet.

Rekursionsabbruch ist das Erreichen einer Partition mit einem Element.

Der Rückgabewert ist eine Konkatenierung der rekursiv verarbeiteten linken Partition, des Pivotelementes und der rekursiv bearbeiteten rechten Partition.

5 Mergesort

“Die Schnittstelle `ksort:msort(<Liste>)` ist vorgegeben und einzuhalten. Technische Vorgabe: die Zahlen sind in der Erlang-Liste `[]` gehalten und dort zu sortieren. Der in der Vorlesung vorgestellte Algorithmus ist so auf die Verwendung von Listen (statt `array`) zu transformieren, dass das Kernkonzept erhalten bleibt! Die Begründung dazu ist im Code als Kommentar aufzuführen.”

```
mergeSort(int left, int right) {
    if (left < right) {
        int mid = (right + left)/2;

        mergeSort(left, mid);
        mergeSort((mid+1), right);

        merge(left, mid, right);
    }
}

merge(int left, int mid, int right) {
    Datensatz tmp[(right-left)+1];
    int tmp_i = 0;
    int i = left; int j = mid+1;
```

²¹ vgl. Abschnitt 1 “Von Arrays zu verketteten Listen”

```

while ((i <= mid) && (j <= right)) {
    if (a[i].key < a[j].key) {
        tmp[tmp_i] = a[i]; i++;
    } else {
        tmp[tmp_i] = a[j]; j++;
    };
    tmp_i++;
}

while (i <= mid) {
    tmp[tmp_i] = a[i]; tmp_i++; i++;
};
while (j <= right) {
    tmp[tmp_i] = a[j]; tmp_i++; j++;
};

tmp_i = 0;
for (i=left; i <= right; i++) {
    a[i] = tmp[tmp_i]; tmp_i++;
}
}

```

5.1 Ablauf des Codebeispiels mit Array

Die Anzahl der zu sortierenden Elemente wird verkleinert, indem das Feld rekursiv in zwei Partitionen aufgeteilt wird, deren Mittelpunkt sich durch die Positionen innerhalb des zu sortierenden Feldes ergibt. Die mittlere Position als Kriterium für die Aufteilung wird ermittelt, indem die Summe des linken und des rechten Index halbiert wird. Dadurch ergeben sich die beiden Partitionen. Eine reicht vom Anfang des linken Bereiches bis zur Mitte (inklusive), die andere vom Element nach der Mitte bis zum rechten Rand. Diese beiden Felder werden jeweils solange rekursiv weiter zerteilt, bis die Problemgröße $n=1$ erreicht wurde.

Das auf diese Art zerteilte Feld wird wieder zusammengefügt und alle Elemente an die richtige Stelle innerhalb des ursprünglichen Feldes kopiert.

- Hierfür wird ein temporärer Zwischenspeicher definiert, der Platz für alle Elemente des aktuell zusammenzufügenden Bereiches hat.
Dieser Zwischenspeicher ist initial inhaltsleer²², der vorderste freie Platz ist also an der ersten Position (hier: 0).
- Nun werden alle Elemente beider Partitionen miteinander verglichen, solange die linke Grenze des linken Feldes noch nicht die rechte Grenze des linken Feldes (die Mitte) erreicht hat und die linke Grenze des rechten Feldes noch nicht die rechte Grenze erreicht hat.

²² In der Literatur finden sich auch ähnliche Merge-Sort-Algorithmen, die einen temporären Bereich definieren, der initial den gleichen Inhalt wie das unsortierte Feld hat.

(Deutlicher ausgedrückt werden die Elemente verglichen, solange beide zu untersuchenden Bereiche noch mindestens ein Element beinhalten.)

- Das erste Element des linken Bereiches wird mit dem ersten Element des rechten Bereiches (Mitte + 1) verglichen. Das kleinere dieser beiden Elemente wird in den temporären Speicher an die vorderste freie Position kopiert. Die linke Grenze des Bereiches, aus dem dieser Wert entnommen wurde, verschiebt sich also um eine Position nach rechts. Der zu sortierende Bereich wird also um ein Element kleiner.
- Hierdurch ergibt sich die richtige Reihenfolge innerhalb des Zwischenspeichers.
- Die erste freie Position des Zwischenspeichers ist nun also um eine Stelle nach rechts²³ gerutscht.
- Da die linke und die rechte Partition auf diese Weise unregelmäßig verkleinert werden, wird der Fall eintreten, dass eine der Partitionen leer läuft. Dann können keine zwei Elemente mehr zum Vergleich herangezogen werden.
- Die zusammenzufügenden Partitionen sind in sich bereits in der richtigen Reihenfolge (entweder weil sie nur ein Element enthalten, oder im vorhergehenden Rekursionsschritt bereits sortiert wurden). Das erste und somit auch kleinste Element des verbleibenden Bereiches wurde im letzten Vergleich mit dem letzten Element des anderen Bereiches bereits als größer als das aktuell hochwertigsten Element im Zwischenspeicher identifiziert. Daher können die übrigbleibenden Elemente in unveränderter Reihenfolge hintereinander in den Zwischenspeicher geschrieben werden.
- Auf diese Weise sind alle Elemente im Zwischenspeicher in der richtigen Ordnung.
- Der Zwischenspeicher kann nun, so wie er ist, in den Ursprungsbereich geschrieben werden, wobei die erste Position im Ursprungsbereich der ersten Position im Zwischenspeicher entspricht.

5.2 Kernelemente

Unabhängig von einer konkreten Implementation lässt sich das Verfahren in folgende Schritte aufteilen:

Der zu sortierende Bereich wird halbiert, indem aus der Anzahl der enthaltenen Elemente die Hälfte ermittelt wird. Die erste Hälfte (inklusive des Mittelelementes) ergibt eine Partition, der Rest die andere.

Die so entstehenden Partitionen werden jeweils rekursiv weiter verarbeitet, bis jede Partition nur noch ein Element enthält.

Nach der Aufteilung werden die Elemente der Partitionen in der richtigen Reihenfolge zusammengefügt, indem das jeweils erste Element der Bereiche miteinander verglichen wird, wobei das niedrigwertigere Element aus seinem Bereich entnommen und in den

²³ Es kann also sein, dass der erste "freie" Platz außerhalb des Zwischenspeichers liegt. Dies ist aber kein Problem, da es keinen Fall gibt der versucht etwas an dieser Position einzufügen.

Zwischenspeicher kopiert wird. Dies wird wiederholt, solange beide Bereiche noch mindestens ein Element beinhalten, das miteinander verglichen werden kann.

Sobald einer der Bereiche kein Element mehr beinhaltet, wird der andere Bereich an das Ende des Zwischenspeichers angehängt.

Der Zwischenspeicher, der nun die richtige Reihenfolge hat, ist das Ergebnis dieses Rekursionsschrittes.

Nachdem alle Rekursionsschritte abgearbeitet wurden, ist das Ergebnis der sortierte ursprüngliche Bereich.

5.3 Ablauf in einer rekursiven Datenstruktur

Listen mit einem oder keinem Element werden als bereits sortiert betrachtet.

Längere zu sortierende Listen können in der Mitte aufgespaltet werden²⁴, indem sie rekursiv bis ans Ende durchlaufen werden, wobei während des Durchlaufs die Elemente gezählt und, sobald das Ende erreicht wurde, die Anzahl halbiert wird. Die Listen werden zusammengesetzt, indem alle Elemente hinter der so errechneten Mitte in eine Partition und alle Elemente vor der Mitte in eine andere Partition abgelegt werden.

Die auf diese Weise entstandenen Partitionen werden rekursiv weiter verarbeitet, wobei die Ergebnisse der Rekursion zusammengefügt werden.

Die Ergebnisse der Rekursionen werden zusammengefügt, indem die jeweils ersten Elemente der zusammenzufügenden Partitionen miteinander verglichen werden, sofern dies möglich ist.

Ist das erste Element der linken Partition kleiner, wird es aus der linken Partition entnommen und an die aktuelle Position geschrieben, ansonsten wird das erste Element der rechten Partition entnommen.

Der Rest der Partition, aus dem das Element entnommen wird, wird zusammen mit der anderen Partition, aus dem kein Element entnommen wurde, auf die gleiche Weise weiter zusammengefügt.

Dies geschieht solange, bis eine der Partitionen leer gelaufen ist. Dann wird die andere Partition direkt zurückgegeben und muss nicht weiter verarbeitet werden. (Abbruchbedingung)

Auf diese Art entsteht eine Liste, in der die Elemente sortiert sind.

Im Vergleich zum Arbeiten auf einem Array entfällt das Anlegen eines Zwischenspeichers.

²⁴ Eine einfachere, leichter zu programmierende Alternative ist die Länge der Liste zu ermitteln, zu halbieren und die Liste anschließend aufzuteilen, was jedoch bedeutet, dass die Liste alleine für diesen Schritt $(n + n/2)$ mal durchlaufen werden muss, anstatt nur n mal.

6 Testumgebung

“Implementieren Sie eine Testumgebung, die die Laufzeit misst. Unterschiedliche Einstellungen, z.B. Anzahl der Zahlen, Strategie bei Quicksort ertc, sollen möglich sein.”

6.1 Zweck und Verfahren

Um Vorhersagen über die zu erwartende Laufzeit eines Verfahrens (im günstigsten, im ungünstigsten und im durchschnittlichen Fall) machen zu können, um Verfahren und ihr Laufzeitverhalten miteinander vergleichen zu können und um (unter Berücksichtigung der Problemkomplexität) den Wert bzw. das Potenzial von Optimierungen beurteilen zu können, werden die Laufzeiten der Verfahren gemessen und miteinander verglichen.

Dies kann durch die “Instrumentierung” erfolgen, bei der innerhalb des Programmcodes an bestimmten Stellen Zeitstempel gemessen und protokolliert werden, oder durch den Einsatz statistischer Verfahren, bei denen die Gesamtlaufzeiten der Algorithmen gemessen und miteinander verglichen werden.

6.2 Vergleichsdaten

6.2.1 Verfahren und Strategien

Da sich das Verhalten des Quicksortverfahren deutlich durch die angewandte Strategie verändert, kann man jede Kombination aus Quicksortverfahren und der angewandten Strategie als eigenes Verfahren betrachten und so besser miteinander vergleichen. Es ergeben sich also für das Quicksortverfahren fünf zu vergleichende Strategien.

$$Q = \{\text{Quicksort}\}$$

$$S = \{\text{/left, /middle, /right, /median3, /random}\}$$

$$|Q \times S| = 5$$

Gemeinsam mit den anderen Sortierv Verfahren ergeben sich so insgesamt acht miteinander vergleichbare Verfahren.

$$V = \{\text{Selectionsort, Insertionsort, Quicksort/left, Quicksort/middle, Quicksort/right, Quicksort/median3, Quicksort/random, Mergesort}\}$$

$$|V| = 8$$

6.2.2 Ausprägung der zu verarbeitenden Daten

Bei der Grobanalyse von Algorithmen werden üblicherweise für die Ausprägung der zu verarbeitenden Daten die Fälle betrachtet, die für die zu vergleichenden Algorithmen am günstigsten (best case), am ungünstigsten (worst case) und durchschnittlich (average case) sind.

Bezogen auf die Sortieralgorithmen sind dies die Fälle, in denen Daten vorsortiert (jeweils aufsteigend oder absteigend) oder zufällig verteilt sind. Bei den zufällig verteilten Werten gibt

es zusätzlich noch den Fall, der gleichwertige Elemente erlaubt.

$Z = \{\text{sorted, reversed, randomized, randomizedwithduplicates}\}$
 $|Z| = 4$

6.2.3 Problemgröße

Die Laufzeit der Verfahren hängt deutlich von der Anzahl der zu verarbeitenden Daten ab. Für einen aussagekräftigen Vergleich müssen daher Messungen mit einer unterschiedlichen Anzahl von Elementen gemacht werden.

Für die Sortieralgorithmen lassen sich (beispielsweise) aussagekräftige Werte in einer zu vertretenden Messzeit ermitteln, wenn zehn Abstufungen vorgenommen werden, in einem Bereich von Listen der Länge 1 bis 4500, mit einem Abstand von jeweils 500 Elementen.

$|N| = 10$

6.2.4 Anzahl der Testdaten

Aus einer Kombination dieser Faktoren ergibt sich die Gesamtzahl von 320 durchzuführenden Messungen.

$|M| = |V \times Z \times N| = 8 * 4 * 10 = 320$

6.3 Einschränkung der Interpretation

Der Vollständigkeit halber sei erwähnt, dass eine reine Zeitmessung ungünstig ist, weil es zum einen erfordert, das jeweilige Verfahren erst einmal zu implementieren und zum anderen diversen Ungenauigkeiten unterliegt (Einfluss unterschiedlicher Hardware, Einfluss des Betriebssystems, Einfluss der Programmiersprache, Einfluss des Compilers und der verwendeten Optionen).

Um die grundlegenden Charakteristika der Verfahren kennenzulernen, reicht eine solche Zeitmessung aber trotzdem aus, weshalb diese Testergebnisse in Teil 7 grob ausgewertet werden.²⁵

7 Testergebnisse

“Führen Sie (auf den Rechnern des Labors wegen der Vergleichbarkeit) mit der unter 5. implementierten Testumgebung aussagenkräftige Messungen durch, mit der Sie die Laufzeitkomplexität der einzelnen Algorithmen abschätzen können (Zufallszahlen, aufsteigend bzw. absteigend sortiert). Zudem ist die "Wechselgröße" bei Quicksort durch Tests zu ermitteln. Erstellen Sie ein pdf, indem die Messungen dokumentiert werden (Versuchsaufbau, Resultate, Interpretation, geforderte Nachweise etc.).”

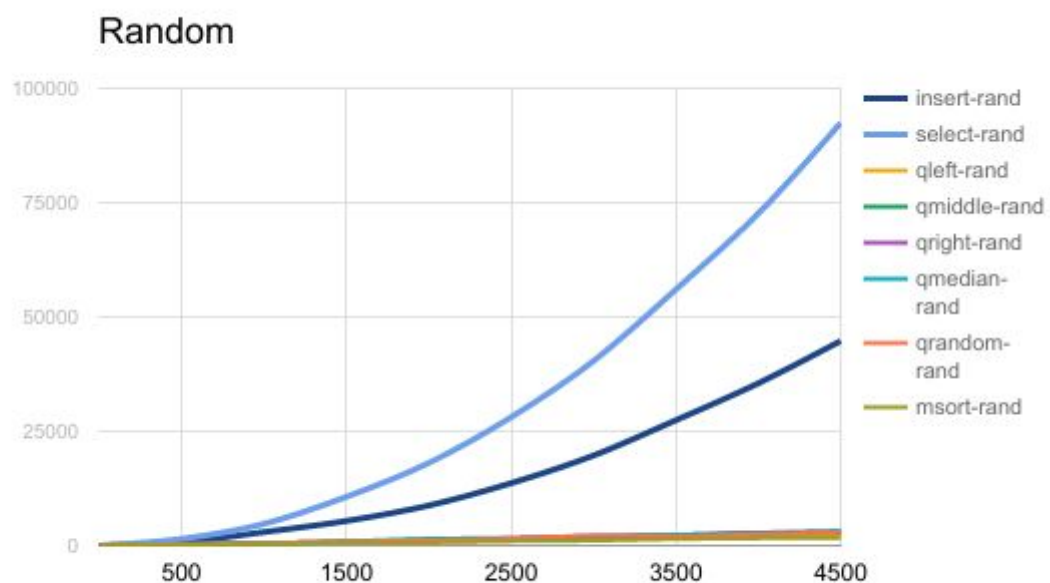
²⁵ Um bereits in der Planungsphase das passende Verfahren für das Problem festzulegen, kann es wertvoll sein, eine theoretische Feinanalyse auf Grundlage der verschiedenen Konzepte, unabhängig der letztendlichen Implementation durchzuführen.

Da der Vergleich der ermittelten Testergebnisse nur eine Orientierung zur Abschätzung bieten soll, sollte es ausreichen, die grob ermittelten Werte auch grob auszuwerten. Entscheidend ist die sich ergebende Kurve, die Genauigkeit der Werte ist zweitrangig.

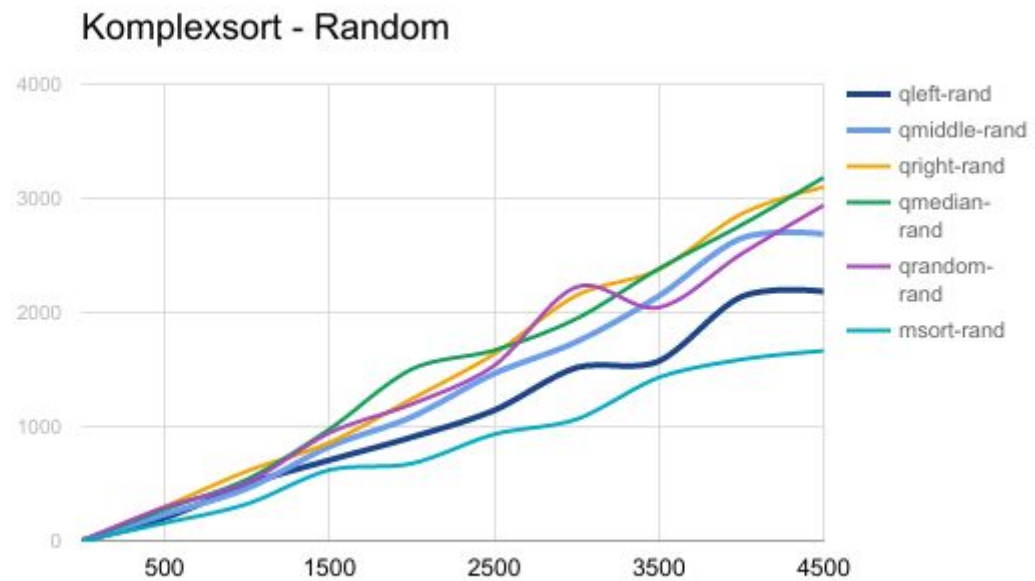
7.1 Laufzeitkomplexität

Aus der angehängten Results.pdf, welche Kurven für jede Mögliche Kombination zeigt, sollen hier einzelne Erkenntnisse exemplarisch ohne Anspruch auf Vollständigkeit gezeigt werden.

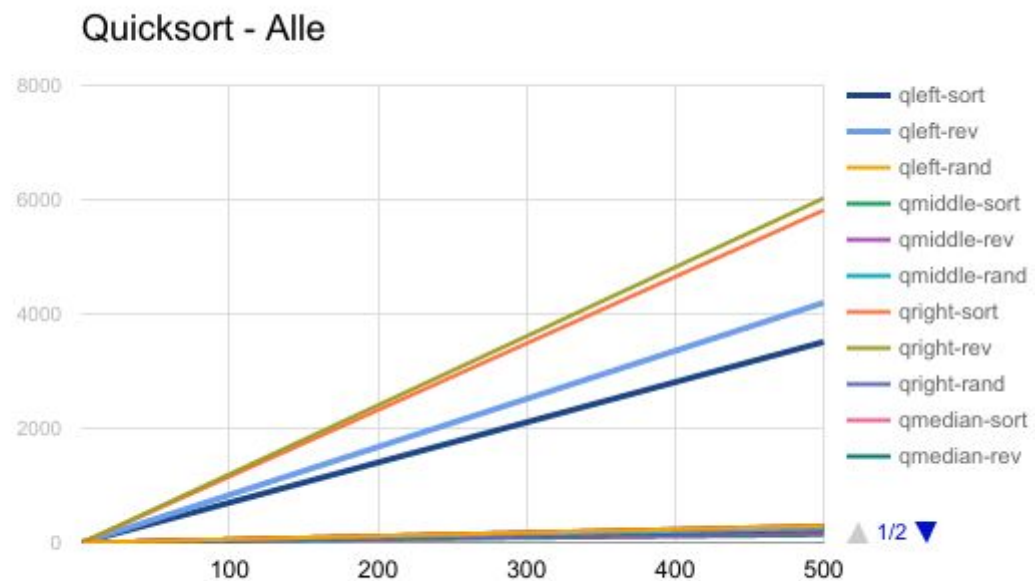
Ein Vergleich aller Laufzeiten für zufällige Werte zeigt wie weit Insertion-Sort und Selection-Sort von den komplexeren Verfahren entfernt sind.



Ein Vergleich der komplexeren Verfahren mit zufälligen Daten zeigt, dass das Mergesortverfahren hier deutlich das schnellste Verfahren ist.

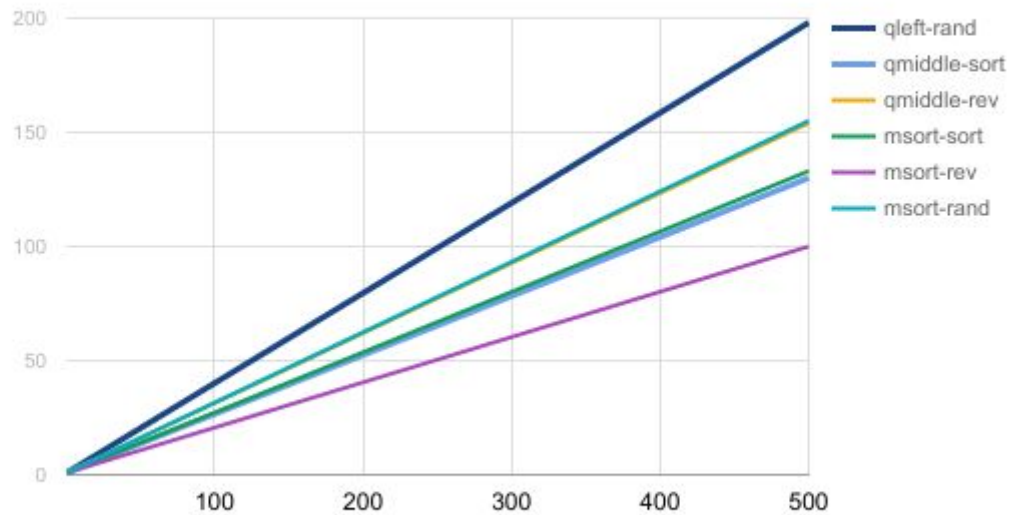


Der Vergleich aller Quicksort-Verfahren zeigt, dass es auch hier deutliche Unterschiede bei den Zuständen der Eingaben gibt.



Für jeden Fall (sorted, reversed, random) werden die komplexen Verfahren verglichen.

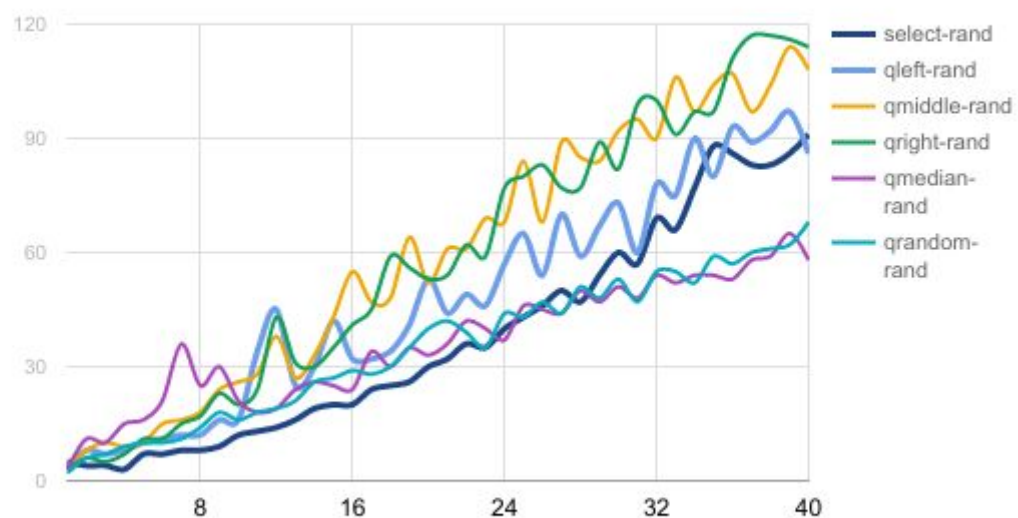
Schnelste Varianten für Eingabedaten



7.2 Wechselgröße Quicksort

Laut meinen Messungen ist für den Fall der zufällig verteilten Daten ein Wechselwert um 28 angebracht, da hier die Selection-Sort (Blaue Linie) langsamer wird als die beiden schnellsten Varianten des Quicksorts.

Wechselgröße



8 Anhang

8.1 Quellenangaben

Deiser, Oliver. *12 X 12 Schlüsselkonzepte Zur Mathematik*. Berlin: Springer Spektrum, 2016. Print.

Knuth, Donald Ervin. *The Art of Computer Programming*. Reading, MA: Addison-Wesley Pub., 1973. Print.

Meinel, Christoph. *Mathematische Grundlagen Der Informatik Mathematisches Denken Und Beweisen: Eine Einführung*. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 2015. Print.

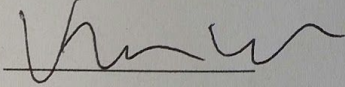
Pomberger, Gustav, and Heinz Dobler. *Algorithmen Und Datenstrukturen*. München: Pearson Studium, 2008. Print.

Rimscha, Markus. *Algorithmen Kompakt Und Verständlich Lösungsstrategien Am Computer*. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 2014. Print.

Weicker, Karsten, and Nicole Weicker. *Algorithmen Und Datenstrukturen*. Wiesbaden: Springer, 2013. Print.

8.2 Erklärung zur schriftlichen Ausarbeitung des Referates

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie die aus fremden Quellen (dazu zählen auch Internetquellen) direkt oder indirekt übernommenen Gedanken oder Wortlaute als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

A handwritten signature in black ink, consisting of a series of loops and a long horizontal stroke at the end, positioned above a short horizontal line.