

Referat - Algorithmen und Datenstrukturen

Andre Brand

Mat. Nr. 2286926

naives vs. komplexes Sortieren

Referat eingereicht im Rahmen der Vorlesung „Logik und Berechenbarkeit“
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

*Fakultät Technik und Informatik
Department Informatik*

*Faculty of Engineering and Computer Science
Department of Computer Science*

Inhaltsverzeichnis

1 Einleitung

1.1	Thema	1
1.2	Überblick.....	2

2 Hauptteil

2.1	Voraussetzungen	3
2.2	Erste Gedanken	3
2.3	Konzept	
2.3.1	Modulbeschreibungen	4
2.3.2	Vorgegebene Schnittstellen	5
2.4	Lösungsweg	
2.4.1	InsertionSort.....	6
2.4.2	SelectionSort	6
2.4.3	QuickSort	7
2.4.4	MergeSort	8
2.5	Berechnungszeiten der Algorithmen	
2.5.1	Durchführung der Zeitmessung	9
2.5.2	Auswertung der Zeitmessung	10
2.5.3	Tabellen mit Zeiten	11
2.5.4	Grafiken zur Visualisierung	13

3 Schluss

3.1	Fazit	16
3.2	Quellen	17

1. Einleitung

1.1. Thema

Die Aufgabenstellung sieht vor, dass wir gegebene Sortieralgorithmen in Erlang implementieren und diese miteinander vergleichen. Die Suchalgorithmen wurden uns als Java-Code zur Verfügung gestellt und wir mussten diese bestmöglich in der Programmiersprache Erlang-OTP nachbilden.

Dies betraf die Sortieralgorithmen InsertionSort, SelectionSort, QuickSort und Mergesort, die ich im folgenden noch kurz beschreiben werde.

InsertionSort

Hier wird das erste unsortierte Element einer Liste an das vordere Ende dieser Liste gelegt. Danach nimmt man das nächste unsortierte Element und vergleicht es mit dem ersten sortierten Element. Ist das unsortierte Element kleiner, so wird es vor das sortierte Element gesetzt. Ist es größer, so wird es hinter das sortierte Element gesetzt. Dies wird so lange wiederholt, bis keine unsortierten Elemente mehr in der Liste vorhanden sind.

SelectionSort

Hier wird das kleinste Element einer Liste gesucht und als sortiertes Element an das vordere Ende der Liste gesetzt. Danach wird wieder das kleinste Element aus der unsortierten Liste gesucht, und hinter die bereits sortierten Elemente gesetzt. Dies wird so lange fortgesetzt, bis nur noch ein Element in dem unsortierten Teil der Liste vorhanden ist.

QuickSort

Hier wird ein PivotElement benötigt. Die Elemente, die kleiner als das PivotElement sind, kommen in eine neue Liste und die größeren ebenso. Dies wird dann für beide neu erstellten Listen mit neuem PivotElement fortgeführt, bis die Teillisten nur noch aus einem Element bestehen. Diese Listen werden dann der Reihe nach aneinander angehängt, um so als sortierte Liste zu entstehen.

MergeSort

Hier wird die Liste in zwei Listen aufgeteilt. Diese beiden Listen werden nun getrennt betrachtet und wieder in neue Listen aufgeteilt, solange, bis nur noch ein Element pro Liste vorhanden ist. Schließlich werden die Teillisten sortiert zusammengefügt. Führt man dies mit allen Teillisten aus, so ist die Liste sortiert. Diese Variante wird auch als Bottom-Up-MergeSort bezeichnet.

1.2. Überblick

Ich werde zuerst die Voraussetzungen klären und meine ersten Gedanken zu diesem Thema verfassen. Danach werde ich in einer Skizze die Algorithmen erklären und die Vorgaben erläutern.

Danach werde ich meinen Lösungsweg beschreiben und hier auch einige Design-Entscheidungen erläutern. Zudem werde ich noch auf die Zeitmessung der Algorithmen eingehen und diese Zeitmessungen vergleichen. Zuletzt werde ich noch mein Fazit aus der Implementation der Algorithmen ziehen.

2. Hauptteil

2.1. Voraussetzungen

Da Erlang-OTP sehr schwach typisiert ist, muss sichergestellt werden, dass die zu sortierenden Listen auch wirklich nur Zahlen enthalten. Dies wurde in meiner Implementierung nicht berücksichtigt. In meiner Implementierung wird nur sichergestellt, dass die, an die Sortieralgorithmen übergebenen Elemente, Listen sind. So kann ein Atom oder eine Liste als Element der Liste zu einem Fehler führen.

2.2. Erste Gedanken

Zuerst habe ich mich mit den verschiedenen Algorithmen beschäftigt und in diversen Quellen nachgeforscht, wie sie arbeiten. Dies habe ich mit dem gegebenen Java-Code verglichen und begonnen darüber nachzudenken, wie eine Implementation in Erlang-OTP aussehen könnte. Bei diesen Überlegungen und durch die Vorlesung ist schnell klar geworden, dass eine Abarbeitung direkt in Arrays, wie in dem gegebenen Java-Code nicht möglich ist. Aus der unsortierten Liste musste also eine neue, sortierte Liste erstellt werden.

Zudem musste sichergestellt werden, dass die gegebenen Java-Algorithmen nicht zu stark verfälscht werden und so ein anderer Algorithmus, der zwar sortiert, aber nicht dem ursprünglichen Algorithmus entspricht. Hierbei wären die Vorgaben nicht eingehalten worden.

Eine weitere Frage war, welche in Erlang bereits enthaltenen Funktionen benutzt werden dürfen. Ich habe mich hierbei dazu entschieden, so wenig gegebene Funktionen zu nutzen, wie es mir möglich ist. Zu den von mir genutzten Funktionen zählen:

length/1 – Zur Feststellung der Länge der Liste

is_list/1 – Zur Feststellung, dass das übergebene Element eine Liste ist

random:uniform/1 – Zur Berechnung eines randomisierten Index für das Random-Pivot-Element

Zudem habe ich für meine Tests und für die Zeiterfassung Funktionen genutzt, die in dem gegebenen Modul util.erl vorhanden waren.

2.3. Konzept

2.3.1. Modulbeschreibungen

InsertionSort

Nimm das erste Element aus dem unsortierten Bereich und lege es in einen für die Sortierung vorgesehenen Bereich. Nimm dann das nächste Element aus dem unsortierten Bereich und vergleiche es mit den Elementen in dem sortierten Bereich. Ist das Element kleiner, lege es vor die anderen Elemente, ist es größer, lege es dahinter. Führe dies so lange durch, bis keine weiteren Elemente im unsortierten Bereich sind.

SelectionSort

Suche das kleinste Element des unsortierten Bereiches und lege es in den sortierten Bereich. Suche nun wieder nach dem kleinsten Element des unsortierten Bereichs und lege das Element hinter die bereits sortierten Elemente, da es nur größer oder gleich der anderen Elemente sein kann. Wiederhole dies, bis kein Element mehr im unsortierten Bereich ist.

QuickSort

Suche zuerst nach dem angegebenen Pivot-Element. Dies kann zu Anfang, zum Ende, in der Mitte, ein Median oder ein zufälliges Element sein. Teile anhand des Pivot-Elementes den unsortierten Bereich in zwei Hälften auf. Der eine Bereich enthält kleinere, der andere größere Elemente wie das Pivot-Element. Führe dies so lange durch, bis die Bereiche nur noch maximal ein Element haben und führe sie dann der Reihe nach zusammen.

MergeSort

Teile den Bereich in zwei gleichgroße Hälften. Führe dies solange durch, bis jeder Teilbereich maximal ein Element enthält. Füge nun die Teilbereiche wieder zusammen und achte beim Zusammenfügen auf die richtige Reihenfolge der Elemente.

2.3.2. Vorgegebene Schnittstellen

InsertionSort

Für diesen Algorithmus wurde die Schnittstelle `ssort:insertionS(<Liste>)` vorgegeben. Eine Liste muss übergeben und auch zurückgeliefert werden können.

SelectionSort

Für diesen Algorithmus wurde die Schnittstelle `ssort:selectionS(<Liste>)` vorgegeben. Eine Liste muss übergeben und auch zurückgeliefert werden können.

QuickSort

Für diesen Algorithmus wurde die Schnittstelle `ksort:qsort(<pivot_methode>,<Liste>,<switch-number>)` vorgegeben.
 <pivot_methode> - kann die Werte left/middle/right/median/random annehmen
 <switch-number> - Listen, deren Länge kürzer ist, als diese Nummer, werden nicht mit dem QuickSort-Algorithmus sortiert, sondern wahlweise mit Insertion- oder SelectionSort.

MergeSort

Für diesen Algorithmus wurde die Schnittstelle `ksort:msort(<Liste>)` vorgegeben. Eine Liste muss übergeben und auch zurückgeliefert werden können.

Technische Vorgaben

Alle Sortieralgorithmen sollen in Erlang implementiert werden und mit Zahlen in einer Erlang-Liste (`[]`) arbeiten.

2.4. Lösungsweg

2.4.1. InsertionSort

Hierbei nehme ich das vorderste Element der unsortierten Liste und füge es in eine neue, leere Liste ein. Dann wird das nächste Element der unsortierten Liste entnommen, mit dem Element der sortierten Liste verglichen und je nachdem ob dieses Element größer oder kleiner ist, vor bzw. hinter das Element der sortierten Liste eingefügt. Dieses Vorgehen wird für alle weiteren Elemente der unsortierten Liste wiederholt, bis die unsortierte Liste keine weiteren Elemente mehr enthält. Das Einfügen geschieht mit folgenden Befehlen:

Wenn das einzufügende Element kleiner als der Kopf der sortierten Liste ist, wird es an das vordere Ende angefügt:

```
insert([H | T], Elem) when Elem =< H ->
[Elem | [H | T]];
```

Wenn das einzufügende Element nicht kleiner als der Kopf der sortierten Liste ist, wird der Rest der Liste überprüft:

```
insert([H | T], Elem) ->
[H | insert(T, Elem)].
```

2.4.2. SelectionSort

Hierbei sucht eine von mir erstellte Funktion das kleinste Element aus der unsortierten Liste. Dieses wird in eine neue, leere Liste eingefügt.

Danach wird das nächste kleinste Element aus dem Rest der unsortierten Liste entnommen und hinter das zuletzt eingefügte Element, also an das hintere Ende, der sortierten Liste gehängt. Hierzu wird die sortierte Liste in einer von mir implementierten Funktion gedreht und das Element an das vordere Ende gelegt. Danach wird diese Liste wieder gedreht, um wieder in der richtigen Reihenfolge ist.

2.4.3. QuickSort

Da bei dieser Funktion mehrere Parameter vorgegeben waren, unter anderem eine Auswahl der Pivot-Methode und eine Switch-Nummer, habe ich mich hier für den Einsatz von Guards entschieden.

In der ersten Funktionsdeklaration wird überprüft, ob die Länge der Liste kleiner oder gleich der Switch-Nummer ist. Sofern dies der Fall ist, wird die Funktion InsertionSort genutzt, da diese in meiner Implementation effizienter ist.

Wenn dies nicht der Fall ist, so wird zwischen den verschiedenen Pivot-Methoden unterschieden. Das jeweilige Pivot-Element wird durch eine durch mich implementierte Funktion geliefert.

Falls eine falsche Pivot-Methode übergeben wurde, wird ein Atom, welches den Fehler beschreibt, zurückgeliefert.

Zudem wird ein Atom zurückgeliefert, wenn das zu sortierende Element keine Liste ist.

Wenn der Algorithmus mit der übergebenen Pivot-Methode gestartet wird, sucht eine eigens implementierte Funktion das jeweilige Pivot-Element heraus.

Wenn left als Pivot-Methode übergeben wird, wird das erste Element der zu sortierenden Liste als Pivot-Element genommen.

Wenn middle als Pivot-Methode übergeben wird, dann wird mit Hilfe der Länge der Liste die Mitte bestimmt, zu dem Element an der mittleren Position gegangen und dieses wird als Pivot-Element zurückgeliefert.

Wenn right als Pivot-Methode übergeben wird, so wird das letzte Element der Liste gesucht. Hierzu gehen wir so lange durch die Liste, bis der Rest der Liste leer ist. Der Kopf wird als Pivot-Element zurückgegeben. Hierbei wurde bewusst nicht die Funktion zum drehen der Liste genutzt, da hier kein Element angefügt werden muss.

Wenn median als Pivot-Methode übergeben wird, so werden die Pivot-Elemente zu left, middle und right gesucht, und das Pivot-Element genommen, welches vom Wert zwischen den beiden anderen liegt.

Wenn random als Pivot-Methode gewählt wird, wird mit der Funktion `random:uniform/1` ein zufälliger Wert erzeugt, welcher im Index der Liste liegt und das Element an dieser zufällig bestimmten Position wird als Pivot-Element genutzt.

Das Pivot-Element wird dann zur Unterteilung der unsortierten Liste genutzt. Jedes Element der unsortierten Liste wird mit dem Pivot-Element verglichen. Elemente die kleiner als das Pivot-Element sind, kommen in die sogenannte linke Liste. Elemente die größer als das Pivot-Element sind, kommen in die sogenannte rechte Liste. Falls Elemente vorhanden sind, die den selber Wert haben, wie das Pivot-Element, werden diese in eine eigene mittlere Liste eingefügt. Dies ist notwendig, da es sonst bei Listen, die Duplikate enthalten, zu einer Endlosschleife führen kann. Die so erstellte linke und rechte Liste werden dann mit der selben Strategie weiter bearbeitet, bis nur noch ein Element pro Liste vorhanden ist.

Nun müssen diese erzeugten Listen nur noch der Reihe nach konkatisiert werden.

Dies geschieht in der von mir implementierten Funktion `concat/2`.

2.4.5. MergeSort

Die unsortierte Liste wird zuerst von einer durch mich implementierten Funktion in der Mitte in zwei Listen geteilt. Dies wird dann für beide Listen wiederholt, bis maximal ein Element pro Liste vorhanden ist. Diese Listen werden dann mit Hilfe der von mir implementierten Funktion `merge/2` wieder zusammengefügt. Beim zusammenfügen wird darauf geachtet, dass immer das kleinere Element zuerst in die neue Liste eingefügt wird. Dies wird durch folgende Funktion sichergestellt:

```
do_merge(Merged, [LeftH | LeftT], [RightH | RightT]) ->
  if
    LeftH <= RightH ->
      do_merge([LeftH | Merged], LeftT, [RightH | RightT]);
    true ->
      do_merge([RightH | Merged], [LeftH | LeftT], RightT)
  end.
```

Da die sortierte Liste in umgekehrter Reihenfolge zusammengesetzt wird, muss sie, sobald beide anderen Listen leer sind, also kein weiteres Element zum Einfügen vorhanden ist, gedreht werden.

2.5. Berechnungszeiten der Algorithmen

2.5.1. Durchführung der Zeitmessung

Zur Berechnung der Zeiten, die ein Algorithmus zur Sortierung einer Liste benötigt, habe ich die Funktion `logging/2` aus dem gegebenen Modul `util.erl` verwendet. Einen Zeitstempel habe ich jeweils vor und nach der Ausführung des Algorithmus mit Hilfe der Funktion `os:timestamp/0` erzeugt. Beide Zeitstempel wurden in Millisekunden umgerechnet und dann der Startzeitpunkt vom Endzeitpunkt abgezogen. Die Differenz beschreibt die Zeit in Millisekunden, die der Algorithmus zur Sortierung benötigt hat.

Die Zeiterfassung wurde von mir in einem Modul `zeitmessung.erl` implementiert. In diesem Modul kann eine Funktion gestartet werden, die alle Algorithmen mit Listen diverser Längen von 10 bis 50.000 Elementen durchführt und diese in Dateien, benannt nach den Algorithmen, bei QuickSort zusätzlich mit der PivotMethode, speichert.

Zudem kann eine einzelne Zeitmessung mit einer übergebenen Länge durchgeführt werden.

Die Ergebnisse werden ebenfalls wie oben beschrieben in Textdateien gespeichert.

Außerdem kann zu jedem Sortieralgorithmus ein einzelner Test mit einer übergebenen Länge gestartet werden, zu dem das Ergebnis auch in eine Textdatei gespeichert wird.

Zur Auswertung habe ich mir zuerst unsortierte Listen mit bis zu 50.000 Elementen erzeugt und bei der Durchführung der Sortierung zu jedem Algorithmus die Zeit erfasst und in eine Textdatei geschrieben. Diese Ergebnisse habe ich dann zum Vergleich in eine Tabelle eingetragen.

Da dies noch nicht genug zum Vergleichen war, habe ich zusätzlich die Zeitmessung für sortierte und gedrehte Listen, sowie für Listen mit Duplikaten mit bis zu 5.000 Elementen durchgeführt.

Die Ergebnisse sind als Tabelle angefügt.

2.5.2. Auswertung der Zeitmessungen

Anhand der angefügten Tabelle erkennt man, dass die Algorithmen MergeSort und alle Varianten des MergeSort sich nicht großartig vom Aufwand her unterscheiden. Auffällig ist, dass meine Implementation des Algorithmus zur SelectionSort sehr langsam und schwerfällig ist. Dies kommt eventuell durch meine eigene Implementation der Suche nach dem kleinsten Element oder durch das wiederholte Drehen der Liste zustande. Zudem ist auffällig, dass alle Implementationen bei Listen mit bis zu hundert Elementen weniger als eine Millisekunde benötigt. Ab einer Listenlänge von ca. 200 Elementen ist es ratsam, für die Sortierung mit MergeSort oder mit QuickSort zu arbeiten.

Dies gilt auch für Listen, in denen Elemente als auch als Duplikate vorhanden sein können. Bei den bereits sortierten Listen fiel auf, dass Quicksort mit dem linken, sowie dem rechten Pivot-Element deutlich ineffizienter wurde. Dies ist allerdings bei dem Algorithmus nicht verwunderlich, da alle Elemente in der Liste nur größer oder kleiner als das Pivot-Element sein können. So wird jedes Element einzeln betrachtet und die Vorteile von Quicksort können nicht genutzt werden. So ist es besser, ein Element aus der Mitte der Liste als Pivot-Element zu bestimmen.

Verwundert hat mich, dass InsertionSort bei gedrehten Listen unheimlich effizient war. Aber da hier das jeweils nächste Element der unsortierten Liste genommen und an den vorderen Teil der sortierten Liste gesetzt wird, ist dies nicht weiter verwunderlich. Das Vorgehen ist ähnlich dem zum Drehen einer Liste.

2.5.3. Tabelle mit Zeiten

Unsortierte Listen ohne Duplikate

Anzahl Elemente	Sortieralgorithmen / Angabe der Zeit in Millisekunden							
	insertionS	selectionS	mergeSort	quickSort				
				left	middle	right	median	random
10	5	0	2	0	0	0	0	0
20	0	0	0	0	0	0	1	0
30	0	0	0	0	1	0	0	0
40	0	0	0	1	0	0	0	0
50	0	1	0	0	0	0	1	0
60	0	1	0	0	0	0	1	0
70	0	1	0	0	0	1	0	0
80	0	1	0	0	0	1	0	1
90	0	1	0	1	0	1	1	0
100	1	1	0	1	0	1	1	1
200	2	4	1	2	1	2	2	2
300	4	8	3	3	3	3	3	2
400	6	13	4	4	5	4	5	5
500	11	19	6	6	7	6	6	8
600	14	29	9	8	9	8	9	8
700	18	39	12	12	11	12	12	13
800	24	57	14	15	15	15	15	16
900	32	75	18	18	20	19	19	19
1000	38	84	22	24	23	23	23	28
2000	152	351	93	91	92	91	90	99
3000	334	786	208	205	205	209	210	211
4000	626	1518	386	380	373	378	383	388
5000	997	2247	622	632	631	618	630	662
6000	1436	3413	882	885	903	885	899	964
7000	1986	4976	1284	1256	1253	1236	1263	1329
8000	2701	5845	1657	1670	1658	1654	1646	1632
9000	3275	7598	2030	2059	2045	2101	2088	2104
10000	4087	9609	2552	2593	2606	2558	2568	2575
15000	9727	21950	6112	6148	6168	6151	6206	6038
20000	17632	37910	11196	11234	11267	11122	11114	11020
25000	28214	63889	18006	18104	18106	18224	18124	17844
30000	41422	99244	26371	26203	26420	26458	26382	26525
35000	56708	122070	36541	36331	36602	36434	36116	35543
40000	75235	167464	48271	48386	48232	48306	48114	47783
45000	95574	221475	61486	61373	61390	61587	61462	60744
50000	118622	246518	76522	76396	76432	77754	75850	75539

Sortierte und gedrehte Listen, sowie Listen mit Duplikaten

Sortiert	Sortieralgorithmen / Angabe der Zeit in Millisekunden							
				quickSort				
Anzahl Elemente	insertionS	selectionS	mergeSort	left	middle	right	median	random
100	1	1	0	1	0	1	1	0
200	1	3	0	3	0	4	1	0
500	6	17	1	19	1	21	2	1
1000	28	57	1	80	2	87	3	3
2000	115	232	5	316	5	348	6	6
5000	760	1460	11	2005	14	2193	15	15

Gedreht	Sortieralgorithmen / Angabe der Zeit in Millisekunden							
				quickSort				
Anzahl Elemente	insertionS	selectionS	mergeSort	left	middle	right	median	random
100	2	0	3	1	0	1	0	3
200	0	3	0	4	0	4	1	1
500	0	17	1	20	2	21	2	2
1000	0	62	2	83	3	91	3	3
2000	0	231	4	316	5	350	6	5
5000	0	1428	11	2008	19	2186	20	17

Mit Duplikaten	Sortieralgorithmen / Angabe der Zeit in Millisekunden							
				quickSort				
Anzahl Elemente	insertionS	selectionS	mergeSort	left	middle	right	median	random
100	0	1	0	0	1	0	0	0
200	0	3	0	1	0	1	0	1
500	4	17	2	1	1	1	2	1
1000	14	63	2	2	3	2	3	3
2000	64	237	4	5	6	5	6	6
5000	386	1458	12	13	19	15	16	16

2.5.3. Grafiken zur Visualisierung

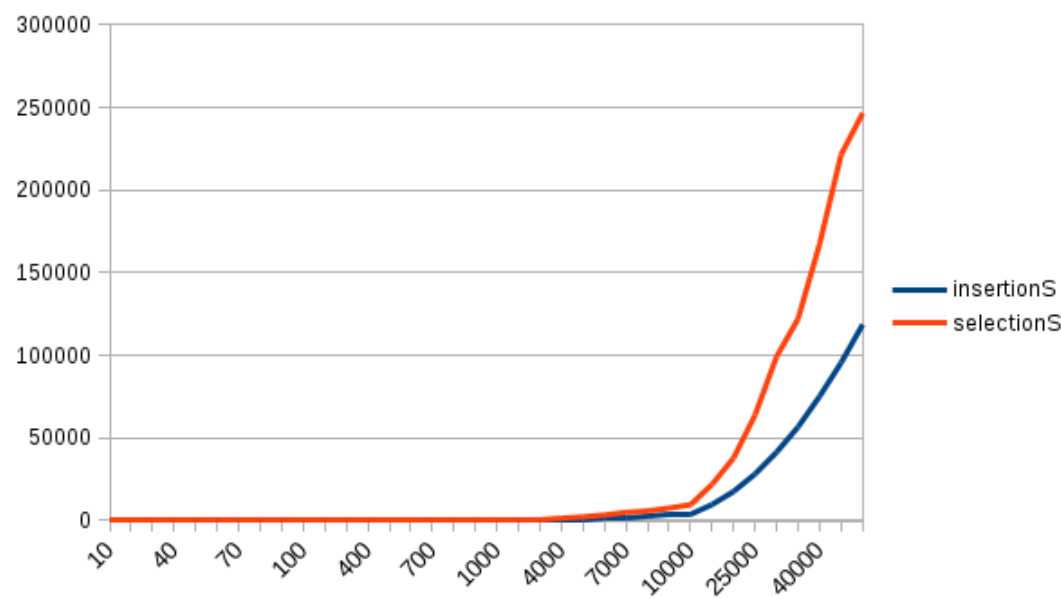


Abbildung 1 - naive Sortiervverfahren im Vergleich

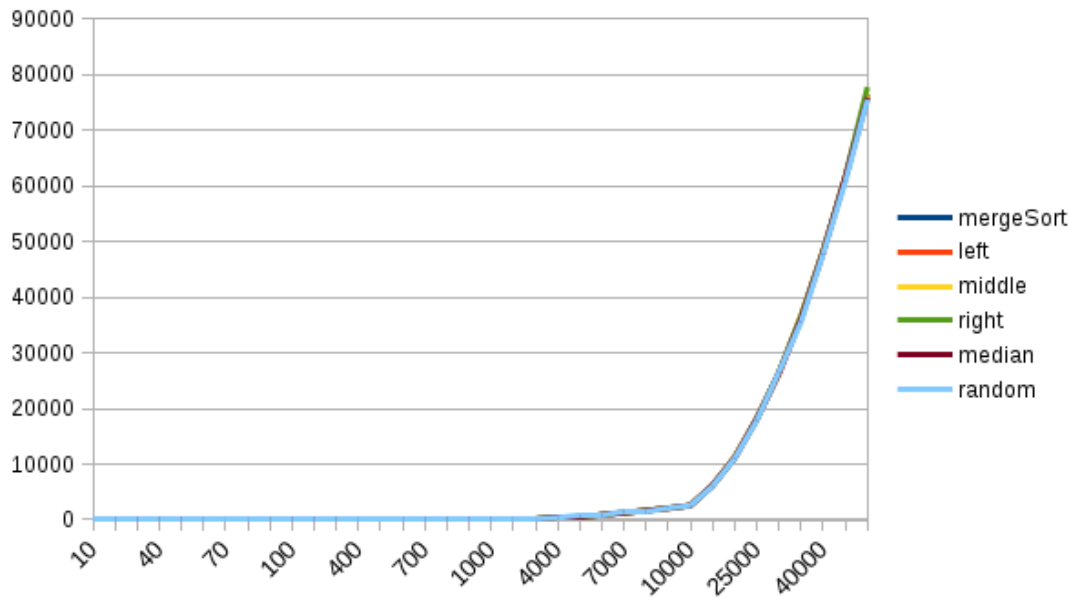


Abbildung 2 - komplexe Sortiervverfahren im Vergleich

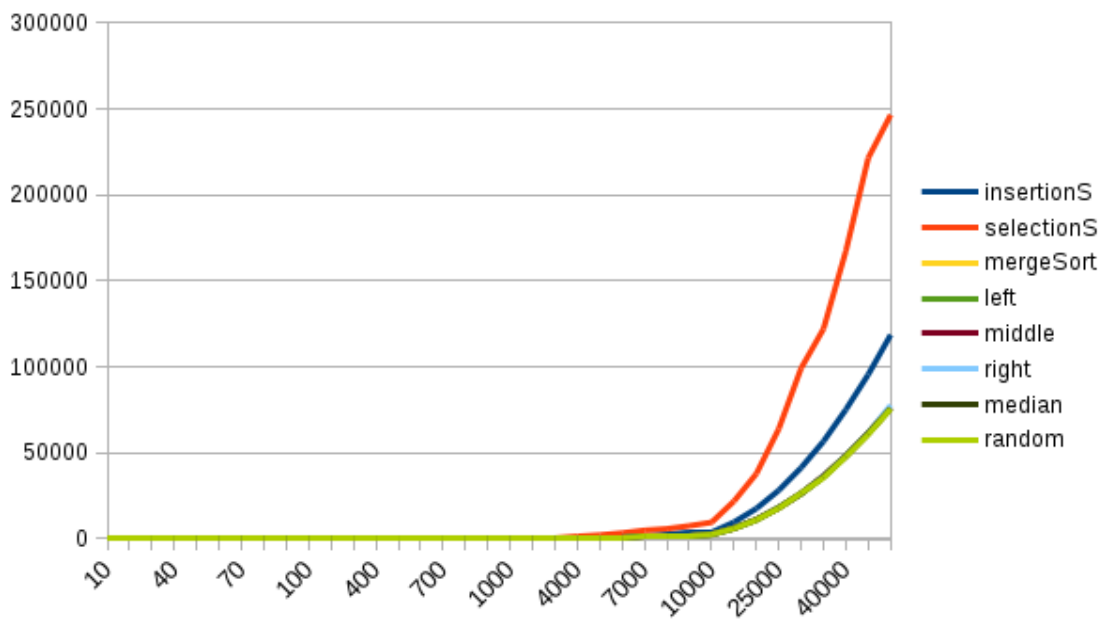


Abbildung 3 - naive und komplexe Sortiervverfahren im Vergleich

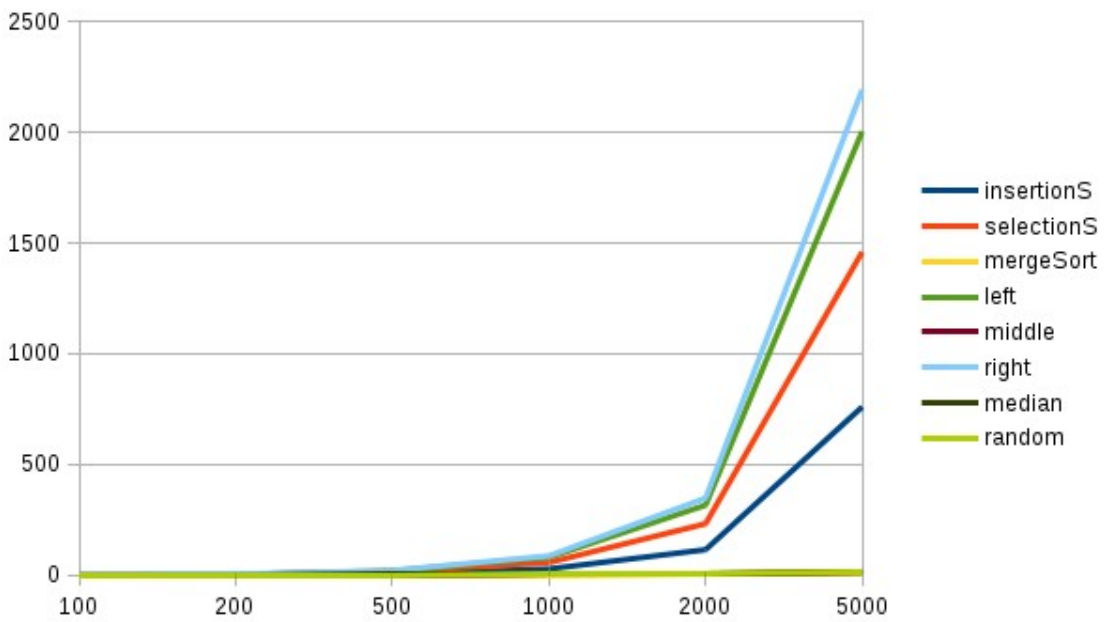


Abbildung 4 - Zeitmessung bei bereits sortierten Listen

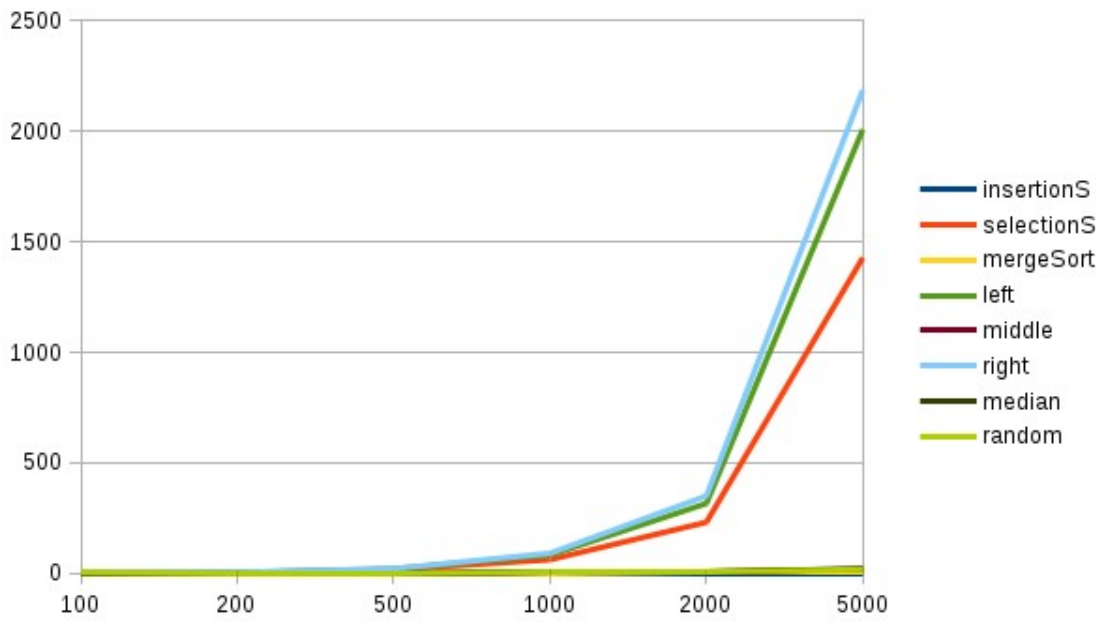


Abbildung 5 - Zeitmessung bei gedrehten Listen

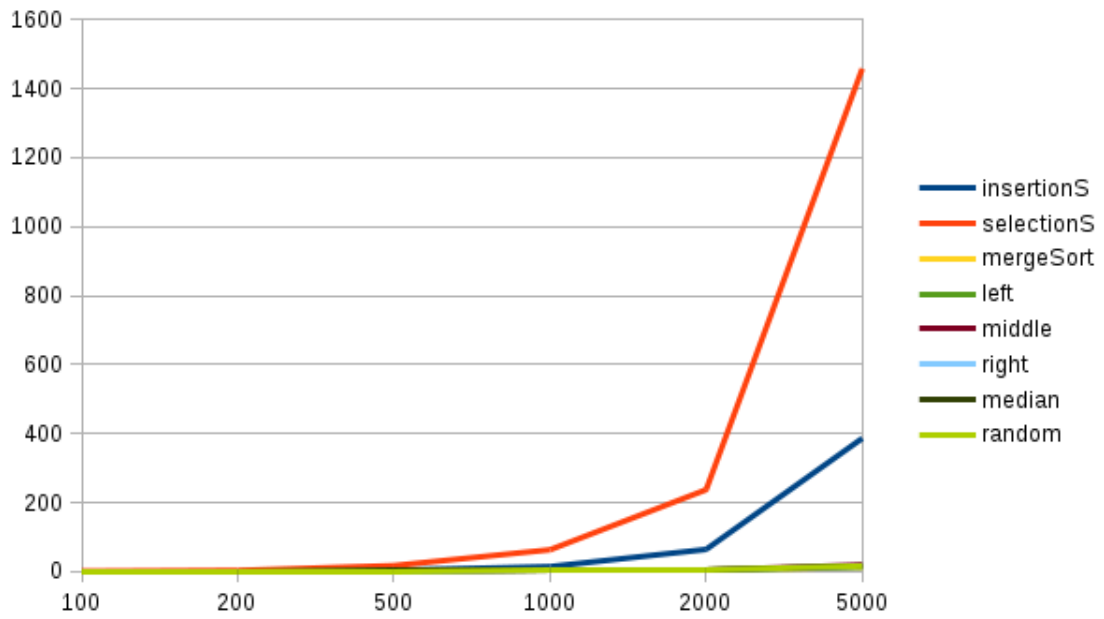


Abbildung 6 - Zeitmessung bei unsortierten Listen mit Duplikaten

3. Schluss

3.1. Fazit

Die Implementierung der Java-Algorithmen in Erlang ist nicht uneingeschränkt möglich, da zum Beispiel Listen in Erlang nicht verändert werden können. So kann auf dieser nicht gearbeitet werden, wie zum Beispiel in Java auf einem Array gearbeitet werden würde. So muss in Erlang immer wieder eine neue Liste erzeugt werden. Da dies jedoch keine Performance-Nachteile mit sich bringt, ist dies zu vernachlässigen.

Zu den Algorithmen ist zu sagen, dass, wie erwartet, MergeSort und QuickSort gerade bei längeren Listen deutlich effizienter sind als InsertionSort und SelectionSort. Zudem hat sich gezeigt, dass bei meiner Implementierung InsertionSort nochmal deutlich effizienter arbeitet, als SelectionSort.

Ich vermute, dass es daran liegt, dass ich beim Einfügen der Elemente in die sortierte Liste eine eigenständig implementierte Funktion zum drehen der Liste verwende. Diese ist, so meine Meinung, verbesserungswürdig, konnte aber aus zeitlichen Gründen nicht mehr optimiert werden.

Eine Verwendung einer direkt in Erlang implementierten Funktion könnte zur Verbesserung führen, wurde aber bewusst von mir nicht vorgenommen, da so ein Lerneffekt erzielt wird.

3.2. Quellen

Original-Dokumentation zu Erlang:

<http://erlang.org/doc/man/erlang.html> (letzter Aufruf: 26.04.17)

Infos zu den Algorithmen:

Robert Sedgewick, Kevin Wayne: Algorithmen und Datenstrukturen, Pearson 2014

<https://de.wikipedia.org/wiki/Quicksort> (letzter Aufruf: 27.04.17)

<https://de.wikipedia.org/wiki/Mergesort> (letzter Aufruf: 27.04.17)

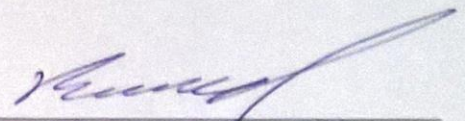
<https://de.wikipedia.org/wiki/Insertionsort> (letzter Aufruf: 27.04.17)

<https://de.wikipedia.org/wiki/Selectionsort> (letzter Aufruf: 27.04.17)

ERKLÄRUNG ZUR SCHRIFTLICHEN AUSARBEITUNG DES REFERATES

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates selbstständig und ohne fremde Hilfe 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. Zudem erkläre ich, dass der zugehörige Programmcode von mir selbstständig implementiert wurde ohne diesen oder Teile davon von Dritten im Wortlaut oder dem Sinn nach übernommen zu haben. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Hamburg, den 27.04.2017
Ort, Datum



Unterschrift