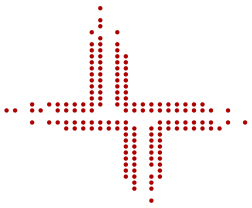

h_da

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi

FACHBEREICH INFORMATIK

Sortieralgorithmen

Tobias Schneider, Fatih Kahraman

5. Juni 2017

Inhaltsverzeichnis

1 Einleitung

1.1 Abstract

Sortieralgorithmen sind heutzutage nicht mehr wegzudenken. Ihre Benutzung ist essentiell für die Verwaltung von vielen Dateien. Sichtbar für den Benutzer wird es z.B. auf Shopping Webseiten, auf welchem man gewisse Artikel nach Wunsch anordnen kann. Wie und welches Sortieralgorithmus hier verwendet wurde, bleibt dem Kunden unbewusst. Interessant für den Entwickler ist die Stabilität und die Schnelligkeit des Sortieralgorithmusses. Diese Seminararbeit konzentriert sich auf 6 populäre Sortierverfahren, welche durch Testfälle unter die Lupe genommen werden. In dieser Seminararbeit werden wir uns auf ein Verfahren konzentrieren, mit dem die Laufzeit berechnet werden kann. Um einen Überblick zu erschaffen, müssen wir uns mit den zahlreichen Eigenschaften eines Sortierverfahrens bekanntlich machen. Im Quelltext wird nachgeforscht, ob zusätzlicher Speicher benötigt wird, wie der Best-Case und Worst-Case aussieht und ob die Stabilität der eingabe gegeben ist .
Die umfangreichen Versuche zeigen, dass es keinen perfekten Sortieralgorithmus existiert. Wir veranschaulichen die Stärken und Schwächen der genannten Verfahren in unterschiedlichen Umgebungen.

1.2 Problem und Relevanz

Das Sortieren von Gegenständen aus der realen Welt oder Elementen aus der virtuellen Welt baut auf die Prozedur des immer sich wiederholenden Akts der Verschiebung oder der Austauschung auf, bis der gewünschte, sortierte Zustand erreicht ist. Hierum handelt es sich um das Ordnen einer Arbeitsumgebung, sei es eine Liste von Zahlen auf dem Rechner, oder der Schreibtisch im eigenen Zimmer. Der Grund, weshalb eine Sortierung überhaupt an erster Stelle ausgeführt wird, bezweckt das leichtere und schnellere Finden von gesuchten Objekten. Ein Stift lässt sich leichter auf einem geordnetem Tisch finden als auf einem ungeordnetem. Mit der Entstehung einer Sortiermethode war es lange nicht getan, denn es leidete an der Performanz der früheren Sortierverfahren. Somit bemühten sich Forscher, immer schnellere und effizientere Sortiermethodiken zu entwickeln. Nun sind wir an einen Zustand gelangt, in wessen die Sortieralgorithmen schon sehr weit entwickelt sind. In dieser Arbeitet werden populäre Sortieralgorithmen in Betracht gezogen, die in verschiedenen Testfällen gezogen werden, um deren Stärken und Schwächen aufzuzeigen.

2 Grundlagen

2.1 O-Notation

Mithilfe der O-Notation, auch Landau-Notation gennant, wird eine Laufzeitberechnung anhand der gegebenen Eingabelänge n bestimmt. Dabei wird ein ungefähres Wachstumsverhalten des Algorithmus als mathematische Funktion definiert. Dies geschieht durch Analyse des Quellcodes und wird üblicherweise für drei Fälle durchgeführt: [1, 2]

Worst-Case: Es wird nach der maximalen Laufzeit des Algorithmus gesucht. Dies geschieht indem eine obere Schranke aufgestellt wird, über die, die Laufzeit nicht steigt. Dieser Fall ist Sortieralgorithmus abhängig und kann nicht immer eine in falscher Reihenfolge sortierte Liste sein.
Best-Case: Stellt die minimale Laufzeit da und wird durch eine untere Schranke realisiert. Auch hier fällt die Laufzeit nicht unter die Schranke. Der Best-Case ist nicht immer eine bereits aufsteigend sortierte Liste sein.
Average-Case: Es wird eine durchschnittliche Laufzeit aufgestellt.

Vorteil der Landau-Notation ist, dass sie komplett unabhängig von Hardware und Betriebssystem ist. Mit ihr kann man die Laufzeiten von Algorithmen miteinander vergleichen.
Ein großer Nachteil ist das die Funktionen nur angenähert sind und so nur eine grobe Einschätzung liefern.

2.2 In-Place

Diese Eigenschaft beschreibt ob der Algorithmus neben der zu sortierenden Liste noch weiteren Hilfsspeicherplatz benötigt oder nicht [3]. Wenn der Algorithmus auf dem Array arbeitet und nur einen zwischenspeicher benötigt wird er als In-Place bezeichnet.

2.3 Stabilität

Ein stabiler Algorithmus behält die Reihenfolge von äquivalenten Werten bei. Diese Eigenschaft ist bei Sortierung von Zahlen nicht relevant, aber sobald mehr Dateien damit zusammenhängen gewinnt diese Eigenschaft an relevanz. [2, 3]

2.4 Testumgebung

Die Tests wurden auf einem Intel I5 Prozessor mit einer Taktfrequenz von 2,5 GHz ausgeführt. Zur Implementierung der Sortieralgorithmen wurde die Sprache C++, mithilfe der Entwicklungsumgebung Netbeans 8.2 mit dem GCC Compiler 4.9.3, benutzt.

3 Sortieralgorithmen

Sortierverfahren werden benutzt, um große Datensammlungen in einer bevorzugten Reihenfolge an zu ordnen. Heutzutage herrschen inzwischen eine Menge von Sortieralgorithmen, die ihre Vorteile in verschiedenen Gebieten der IT bekanntlich machen. Welches Verfahren wo benutzt werden soll, hängt von einer gewissen Anzahl von Kriterien ab. Im Folgenden werden sechs populäre Sortieralgorithmen vorgestellt, wie sie Schritt für Schritt im Programm vorangehen.

3.1 BubbleSort

3.1.1 Geschichte

3.1.2 Vorgehensweise

Idee: Größere Elemente steigen im Array nach rechts auf.

Der BubbleSort ist ein Algorithmus der mit zwei ineinander verschachtelten Schleifen arbeitet. Die äußere Schleife definiert bis zu welchem Punkt die innere Schleife läuft. Die innere Schleife vergleicht das aktuelle Element mit dem darauf folgenden, und falls das aktuelle größer ist, werden diese beiden Elemente vertauscht.
Wenn die innere Schleife einen Durchlauf abgeschlossen hat, steht das größte Element am Ende und wird danach nicht mehr berücksichtigt.
Eine Besonderheit wurde noch hinzugefügt, es wird in einem Schleifendurchlauf mittels „swapped“ kontrolliert ob getauscht wurde. Falls nicht getauscht wurde, ist die Liste schon fertig sortiert und der BubbleSort wird abgebrochen. Mit diesem Trick ist der Best-Case O(n) möglich.

3.1.3 Eigenschaften

O-Notation: Da bei einer bereits sortierten Liste keine vertauschungungen erfolgen , ist der BubbleSort nach einem Schleifendurchlauf fertig und beendet sich mit dem break. Beim Worst-Case werden beide Schleifen komplett durchlaufen was zu einer O-Notation von n^2 führt.

Best Case	**Average Case**	**Worst Case**
O(n)	O(n^2)	O(n^2)

Tabelle 1: O(Notation) des BubbleSorts [3]

Best-Case Der Best-Case des BubbleSorts ist eine bereits sortierte Liste. Der Algorithmus geht die Liste einmal durch und merkt es wurde nicht getauscht. Es erfolgt ein break;. Die O-Notation ist deshalb O(n).
Worst-Case Eine in falscher Reihenfolge sortierte Liste ist der Worst-Case des BubbleSorts. Da dadurch beide Schleifen komplett durchlaufen werden ist die O-Notation O(n^2).

Stabilität Da nur jeweils benachbarte Elemente mittels echt größer als verglichen und vertauscht werden, ist der BubbleSort ein stabiler Algorithmus

In-Place Da der Algorithmus keine rekursiven aufrufe durchführt und lediglich zum vertauschen einen temporären Speicherplatz benötigt, arbeitet der Algorithmus mit In-Place und der Hilfsspeicher ist O(1). [3]

3.2 InsertionSort

3.2.1 Geschichte

3.2.2 Vorgehensweise

Idee: Einsortieren von Elementen in eine bereits sortierte Liste.

Der InsertionSort arbeitet wie der BubbleSort mit zwei ineinander verschachtelten Schleifen.Die Anzahl der Elemente ist am anfang auf 2 Elemente festgelegt. Bei jedem äußeren Durchlauf wird die größe des Arrays um eins erhöht, bis zu seiner maximal größe. Dadurch wird immer ein Element zur bereits sortierten Liste hinzugefügt und an die richtige stelle einsortiert. Die bereits sortierten Elemente werde falls nötig nach hinten verschoben.

3.2.3 Eigenschaften

O-Notation:

Best Case	Average Case	Worst Case
O(n)	O(n^2)	O(n^2)

Tabelle 2: O(Notation) des InsertionSorts [3]

Best-Case Der Best-Case des InsertionSorts ist eine bereits sortierte Liste. Da dann die Anzahl der verschiebungen null ist. Da die Äußere Schleife aber n mal aufgerufen wird und die innere auch ist hier die O-Notation O(n^2).
Worst-Case Eine falsch herum sortierte Liste ist der Worst-Case. Ähnlich wie beim Best-Case werden die Schleifen durchlaufen. Es wird aber bei jedem durchgang maximal verschoben. Auch hier ist die O-Notation O(n^2).

Stabilität: Es werden nur benachbarte Elemente miteinander verschoben falls ein Element echt größer ist. Es werden außerdem nur Elemente von links in das Array eingefügt. Dadurch ist die Stabilität gegeben.

In-Place: Der Algorithmus arbeitet nur auf der übergebenden Liste. Es wird nur ein zusätzlicher Speicher für das Vertauschen benötigt deshalb ist der InsertionSort In-Place und benötigt O(1) Hilfsspeicher. [3]

3.3 SelectionSort

3.3.1 Geschichte

3.3.2 Vorgehensweise

Idee: Suche das kleinste Element und stelle es nach vorne.

Mithilfe von zwei ineinander veschachtelter Schleifen wird der SelectionSort implementiert. Dabei wird das Array komplett durchlaufen um das kleinste Element darin zu finden. Danach wird das kleinste Element mit dem Element an der ersten Stelle vertauscht und der Startwert um 1 erhöht. Dies geschieht solange bis das Array schlussendlich sortiert wurde.

3.3.3 Eigenschaften

O-Notation:

Best Case	**Average Case**	**Worst Case**
O(n^2)	O(n^2)	O(n^2)

Tabelle 3: O(Notation) des SelectionSorts [3]

Best-Case Der Best-Case ist eine bereits sortierte Liste, da in diesem fall nicht getauscht werden muss. Es wird aber bei jedem einzelnen Durchgang das Array komplett durchlaufen.
Worst-Case Ähnlich wie beim Best-Case werden die Arrays jedesmal Durchlaufen. Es muss aber bei jedem Durchgang getauscht werden.

Stabilität: Der SelectionSort ist kein Stabiler Algorithmus. Bei der Wahl des kleinsten Elements wird zwar die Stabilität gewährleistet, aber beim vertauschen wird das vorderste Element mit dem kleinsten vertauscht. Hier kann das vorderste hinter ein Element gebracht werden, was die gleiche größe besitzt.

In-Place: Der SelectionSort arbeitet auf dem Array und benötigt zusätzlich nur einen Tauschspeicher weswegen er In-Place ist. Er benötigt Hilfspeicher von O(1) [3].

3.4 MergeSort

3.4.1 Geschichte

3.4.2 Vorgehensweise

Idee: Teile das Array in der mitte in zwei hälften bis das Array nur noch ein Element behält. Füge danach die teile wieder sortiert zusammen.

Der MergeSort baut auf dem Teile und Hersche Prinzip auf. Er besteht aus zwei teilen:

MergeSort: Das Array wird in der Mitte geteilt und danach wird der MergeSort Rekursiv für diese Teilmengen aufgerufen. Dies geschieht solange bis das übergebene Array genau die größe 1 besitzt. Es ist zu beachten das hier auf dem Array gearbeitet wird. Es werden nur neue Start und Endpunkte des Arrays übergeben.
Merge: Sobald ein Teilbereich nicht mehr geteilt werden kann, werden diese Elemente wieder zusammengefügt. Dabei wird der Inhalt aus den beiden Teilarrays in temporäre Arrays gespeichert und dann der größe nach in das Ursprungsarray zurückgeschrieben. Die temporären Arrays werden danach wieder gelöscht.

3.4.3 Eigenschaften

O-Notation:

Best Case	Average Case	Worst Case
O(n log n)	O(n log n)	O(n log n)

Tabelle 4: O(Notation) des MergeSorts [3]

Best-Case: Der Best-Case ist eine bereits sortierte Liste. Der entscheidende Faktor ist das zusammenfügen. Es wird erst die linke hälfte mit der rechten hälfte verglichen und jeweils nur die linke hälfte wird einsortiert. Da nun keine Elemente in der Linken hälfte sind, kann nun der Inhalt der rechten hälfte ohne vergleiche in das Array zurückgeschriebenen werden.
Worst-Case: Der Worst-Case ist eine Liste bei der immer abwechselnd ein Wert von der Linken hälfte und danach ein Wert von der Rechten hälfte einsortiert wird. Dadurch muss immer verglichen werden.

Stabilität: Der MergeSort ist ein Stabiler Sortieralgorithmus. Beim zusammenfügen wird das Linke Array mit kleiner gleich mit dem Rechten Array verglichen. Dadurch werden die von der größe her gleichen Elemente im Linken Element bevorzugt. Es ist somit nicht möglich ein Element im linken zu überholen.

In-Place: Der MergeSort arbeitet nicht In-Place, da der Tauschspeicher jeweils zweimal die hälfte der Teilmengen ist. Da der Algorithmus von Unten nach Oben neue Arrays erstellt ist der zusätzliche Speicherplatz O(n).

3.5 QuickSort

3.5.1 Geschichte

3.5.2 Vorgehensweise

Idee: Finde Stellvertreter Element. Setze links vom Element alles was kleiner ist. Setze rechts alles was größer ist. Führe dies Rekursiv für linke und rechte hälfte aus.

Der QuickSort baut auf dem Prinzip von Teile und Hersche auf. Er besteht aus zwei teilen:

Partition: Es wird ein Stellvertreter Element aus der Liste ausgewählt. Dies kann: das erste Element, das letzte Element, das Element in der mitte oder der Median aus erstes, mitte und letztes sein [4].
Danach werden die Elemente die kleiner sind links davon und Elemente die größer sind rechts davon positioniert.
QuickSort: Nun wird die linke und rechte hälfte rekursiv mit dem QuickSort wieder aufgerufen.

3.5.3 Eigenschaften

O-Notation:

Best Case	Average Case	Worst Case
O(n log n)	O(n log n)	O(n^2)

Tabelle 5: O(Notation) des QuickSorts [3]

Best-Case: Der Best-Case des QuickSorts ist Pivot abhängig. Das Pivot Element ist der Punkt wo das Array aufgeteilt wird. Man möchte deshalb das das Pivot Element in der mitte ist um eine aufteilung in zwei möglichst gleichgroße hälften zugewährleisten. Zusätzlich werden keine vertauschungen durchgeführt. Dies entspricht bei Pivot Element Mitte einer sortierten Liste.
Worst-Case: Der Worst-Case ist n^2. Er entsteht wenn das Pivot Element jeweils das Erste oder Letzte Element ist. Dadurch wird vom Array immer nur ein Element abgezogen, was im endeffekt einem Sortieralgorithmus mit zwei ineinander verschachtelter Schleifen gleicht.

Stabilität: Der QuickSort ist kein stabiler Sortieralgorithmus. Beim Anwenden von Partition werden die Elemente vertauscht und es kann vorkommen das gleiche Elemente beim kopieren die reihenfolge vertauschen.

In-Place: Es wird auf dem Array gearbeitet weswegen der Algorithmus In-Place arbeitet. Der Hilfsspeicher ist im Worst-Case O(n), da durch die rekursiven Aufrufe von QuickSort zusätzlicher Platz auf dem Stack benötigt wird.[4]

3.6 HeapSort

Geschichte

Vorgehensweise

Idee: Erstellen eines Binärbaums. Größtes Element steht immer in der Wurzel. Stelle es an die letzte Stelle des Arrays und verringer die Arraygröße um eins. Führe dies rekursiv durch bis das Array sortiert ist.

Der HeapSort besteht aus zwei Funktionen:

Heapify: Mithilfe der Heapify Funktion wird das Array von unten nach oben durchwandert. Dabei wird das größte Element aus Knoten und derren Linken und Rechten Kind im Knoten gespeichert.
HeapSort: Wenn Heapify abgeschlossen ist, steht das größte Element in der Wurzel. Nun wird dieses Element an die letzte stelle des Arrays kopiert und anschließend wird die größe des Arrays um eins verringert. Danach wird Heapify erneut aufgerufen.

Eigenschaften

O-Notation:

Best Case	Average Case	Worst Case
O(n log n)	O(n log n)	O(n log n)

Tabelle 6: O(Notation) des HeapSorts [3]

Best-Case: Der Best-Case des HeapSort wird durch minimale vertauschungen beim ausführen von heapify erreicht. Dabei sollte so wenig wie möglich vertauscht werden. Dies ist durch eine Liste mit nur äquivalenten Werten der Fall, da in diesem Fall nicht vertauscht wird.
Worst-Case: Der Worst-Case wird durch maximale vertauschungen beim ausführen der Heapify Funktion.

Stabilität: Da Beim vertauschen das Element in der Wurzel mit dem letzten vertauscht wird, kann es dadurch zu instabilen vertauschungen kommen.

In-Place: Das Array wird intern zu einem Binärbaum auf dem dann sortiert wird. Diese Umformung wird nur mathematisch durchgeführt wodurch kein zusätzlicher Speicherplatz benötigt wird. Es wird zum vertauschen aber ein temporärer zwischenspeicher benötigt.
Der HeapSort arbeitet In-Place und hat einen Hilfsspeicherwert von O(1) [3].

3.7 Evaluierung

In den vorangegangenen Kapiteln wurden die Grundlagen gelegt. In dem nun folgenden Kapitel geht es um die empirische überprüfung der bereits vorgestellten Sortieralgorithmen. Zum Einstieg in die Evaluation betrachten wir erst einmal die unterschiedlichen Funktionstypen und derren Wachstum.

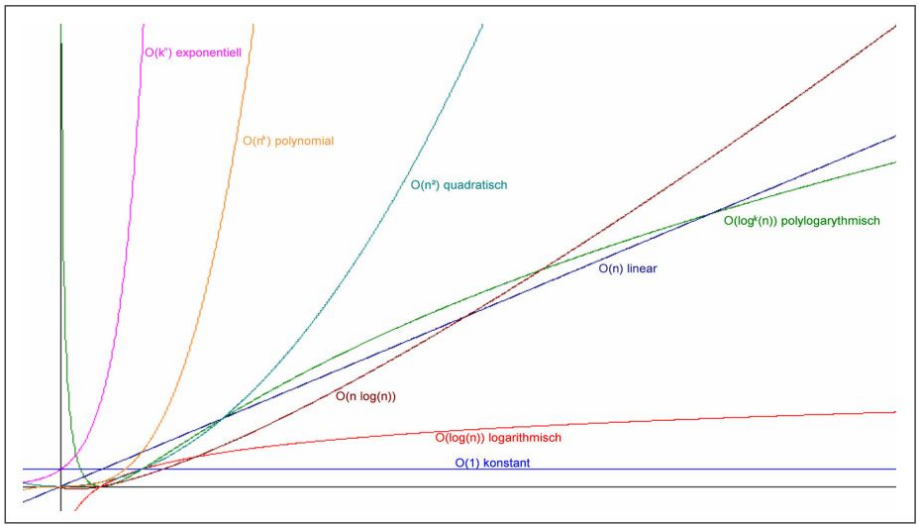

Abbildung 1: Überblick der einzelnen Funktionen und derren Verlauf [2]

Es wird sofort deutlich, dass die lineare und die n log n Funktionen ein deutlich niedrigeres Wachstum besitzen als die quadratische.

Algorithmus	**Best Case**	**Average Case**	**Worst Case**
BubbleSort	O(n)	O(n^2)	O(n^2)
InsertionSort	O(n)	O(n^2)	O(n^2)
SelectionSort	O(n^2)	O(n^2)	O(n^2)
MergeSort	O(n log n)	O(n log n)	O(n log n)
QuickSort	O(n log n)	O(n log n)	O(n^2)
HeapSort	O(n log n)	O(n log n)	O(n log n)

Tabelle 7: O(Notation) der vorgestellten Algorithmen [3]

Nun wäre es aber fahrlässig nur ahand der O-Notation einen Algorithmus auszuwählen. Denn wie im Kapitel zur O-Notation bereits erwähnt, werden konstanten entfernt.

3.7.1 Testfälle

Um die Algorithmen untereinander zu vergleichen wurden einige Testfälle definiert.
Die Code Grundlagen zu den Algorithmen wurden von folgenden Seiten entnommen: [5, 6, 7, 8, 9, 10]

Zufalls-Werte: Den einzelnen Algorithmen wurden die glechen Zufallszahlen übergeben und die Zeit gemessen bis diese die Liste sortiert haben. Es wurden Arrays mit der größe von 100.000 Elementen übergeben

BubbleSort	**SelectionSort**	**InsertionSort**	**QuickSort**	**MergeSort**	**HeapSort**
34793,6	14644	12765,4	15,8	46,4	31,4

Tabelle 8: Ergebnisse von Zufallswerten in Milisekunden(Durchschnittswerte)

An diesem Test erkennt man relativ schnell das die logarithmischen Algorithmen einen riesigen Vorteil gegenüber den quadratischen Algorithmen besitzen wenn es um zufällig sortierte Listen geht.

Sortierte: Es wird eine bereits sortierte Liste den einzelnen Sortieralgorithmen übergeben. Dies ist der Best-Case vom BubbleSort , InsertionSort, QuickSort(mit Pivot Element in der Mitte) und dem MergeSort.

BubbleSort	**SelectionSort**	**InsertionSort**	**QuickSort**	**MergeSort**	**HeapSort**
< 1	14640,6	< 1	6,4	31,2	34

Tabelle 9: Ergebnisse von bereits Sortierter Liste in Milisekunden(Durchschnittswerte)

An diesem Testfalll erkennt man die stärke von O(n) im vergleich zu Logarithmischen Algorithmen.

Absteigend Sortiert: Den Algorithmen wird eine absteigend Sortierte Liste übergeben. Dies ist der Worst-Case des BubbleSorts, InsertionSort und SelectionSorts.

BubbleSort	**SelectionSort**	**InsertionSort**	**QuickSort**	**MergeSort**	**HeapSort**
32427,8	15290,6	30072,2	9,2	31,2	40,8

Tabelle 10: Ergebnisse von abgsteigend Sortierter Liste in Milisekunden(Durchschnittswerte)

sortiert aber letztes Element ist kleinstes: Es wird eine aufsteigend sortierte Liste, mit der besonderheit, das dass letzte Element das kleinste ist,

den Algorithmen übergeben.

BubbleSort	SelectionSort	InsertionSort	QuickSort	MergeSort	HeapSort
13956,4	14553,6	< 1	< 1	34,4	31

Tabelle 11: Ergebnisse von abgsteigend Sortierter Liste mit letztes kleinstes in Milisekunden(Durchschnittswerte)

An diesem Test erkennt man, dass der BubbleSort in jedem durchgang nur einen tausch durchführen muss, während der SelectionSort in jedem durchgang tauschen muss. Dies erklärt warum der BubbleSort schneller ist.

Testfälle mit den Logarithmischen Sortieralgorithmen: Da die Ergebnisse der Logarithmischen Tests relativ klein ausfallen, erhöhen wir die Anzahl der Elemente auf 1.000.000.

Testfall	QuickSort	MergeSort	HeapSort
Zufall	159,2	425	515,4
aufsteigend	62,6	325	390,5
absteigend	63	355,6	415,4
sortiert, ende klein	59,8	325	387,4

Tabelle 12: Testfälle mit 1.000.000 Elemente in Milisekunden(Durchschnittswerte)

Worst-Case QuickSort: Da der QuickSort in keinen bisherigen Testfällen negativ aufgefallen ist, benötigen wir einen spezial Testfall für den Worst-Case des QuickSorts. Der Worst-Case des QuickSorts tritt ein wenn das Pivot Element das kleinste, oder größte Element ist. Da unser Pivot Element in den bisherigen Testfällen immer das Element in der Mitte war, ändern wir für diesen Test die Pivotwahl. Wir wählen immer das erste Element in der Liste und als Array übergeben wir eine absteigend sortierte Liste.

BubbleSort	QuickSort	Selection	Insertion
465,6	218,8	137,4	259,2

Tabelle 13: Testfälle mit 10.000 Elemente in Milisekunden(Durchschnittswerte)

Die Ergebnisse zeigen, dass der QuickSort im Worst-Case ungefähr gleichauf mit dem InsertionSort ist. Es ist aber zu beachten, dass der QuickSort nur wenige Vertauschungen durchgeführt hat und das dies der Worst-Case des InsertionSorts ist.

3.7.2 Vergleich der Ergebnisse

BubbleSort: Der BubbleSort ist wohl einer der am einfachsten zu verstehenden Algorithmen und er bietet weiterhin auch Stabilität und arbeitet In-Place.

Das sind leider auch die einzigen Vorteile dieses Algorithmus. Er benötigt im vergleich zu anderen n^2 Algorithmen in einigen Tests um den Faktor zwei mehr Zeit. Weiterhin tritt der Best-Case zu selten ein um wirklich relevant zu sein. Er dient heute hauptsächlich nur dem Einstieg in Sortieralgorithmen und sollte wirklich nur zu lernzwecken verwendet werden.

SelectionSort: Der SelectionSort besitzt immer eine Laufzeit von O(n^2) was ihn von der O-Notation zum schlechtesten Algorithmus macht. Er liefert aber immer eine konstante Laufzeit und ein großer Vorteil ist, dass die Anzahl der Vertauschungen maximal n sind, da in jedem inneren Schleifen durchlauf nur ein einziges mal getauscht wird.

InsertionSort: Der InsertionSort bietet Stabilität und arbeitet In-Place. Im vergleich zum BubbleSort ist der InsertionSort in allen bereichen überlegen.

MegeSort: Der MergeSort ist der einzige hier vorgestellte Algorithmus der mit einer Laufzeit von O(n log n) die Stabilität der Eingabe gewährt und sollte, falls man die Stabilität benötigt, gewählt werden. Ein großes Problem ist aber der zusätzliche Hilfsspeicherbedarf von O(n). Dies bedeutet das für eine Liste von zum Beispiel 2GB noch zusätzlich 2GB benötigt werden.

HeapSort: Der HeapSort liefert konstante Ergebnisse, die leicht schlechter sind als beim MergeSort. Er achtet nicht auf die Stabilität, arbeitet dafür aber In-Place mit O(1).

QuickSort: Der QuickSort schneidet in useren Testfällen am besten ab. Dies erklärt sich dadurch, das der Worst-Case so gut wie nie auftritt. Das Hauptproblem des QuickSorts ist aber die Wahl des Pivot Element, da links und rechts davon jeweils abgeschnitten wird. Am Worst-Case Testfall haben wir das Pivot als erstes Element der Liste gewählt und eine sortierte Liste übergeben. Dies hat dazu geführt, dass in jedem rekursiven Aufruf nur ein Element von der Liste abgeschnitten wurde. Dies bedeutet das auf dem Stack die Funktion n mal liegt. Bei zu großen Eingaben kann dies zu einem Stackoverflow führen.

3.7.3 Relevanz der O-Notation

Anhand der Testfälle erkennt man, das die O-Notation keine genaue vergleiche zwischen Sortieralgorithmen zulässt. Das ist aber auch nicht ihr Sinn. Mithilfe der O-Notation ist es relativ einfach Algorithmen in eine Laufzeitspalte einzuordnen um ein Feedback zu dessen Laufzeit zu bekommen. Danach kann man sich mit der richtigen Laufzeit im vergleich zu anderen Algorithmen auseinandersetzen.

4 Schluss

4.1 Zusammenfassung und Ausblick

4.1.1 Fazit

Ergebnisse zeigen, dass bei dem SelectionSort keine großen Schwankungen zwischen den Testfällen bei konstanter Listengröße gibt. Nicht besser konnte der BubbleSort abschneiden. Der seltene Auftritt der Best-Cases macht ihn nicht zu einem besseren Algorithmus als der SelectionSort. Im Durchschnitt ist der InsertionSort in jedem Gebieten dem BubbleSort überlegen. Der MergeSort is stabil, doch Schwäche zeigt er in der Speicherverwaltung, denn er benötigt doppelt so viel Speicher zum Sortieren. Der QuickSort liefert die besten Ergebnisse und arbeitet In-Place, doch eine Gefahr in den Worst-Case n^2 zu geraten. Der HeapSort ist nicht stabil, arbeitet In-Place und hat konstantere Ergebnisse als der MergeSort und sein Worst-Case ist ebenso O(n log n).

4.1.2 Ausblick

In der Neuzeit werden Sortieralgorithmen aus den vergangen Jahren verarbeitet, kombiniert oder auch weiterentwickelt. Der LibrarySort ist eine direkte Verbesserung des InsertionSorts. Die durchschnittliche Laufzeit des LibrarySort ist O(n log n), besser als die des InsertionSorts O(n^2) [11]. Eine Kombination des QuickSort und des HeapSort ist der IntroSort [12]. Dieses Verfahren benutzt zuerst den QuickSort, wechselt aber dann über auf den HeapSort, um das Worst-Case Szenario des QuickSort zu eliminieren.

5 Literaturverzeichnis

Literatur

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[2] C. Rehn, "Sortieralgorithmen unter mathematischen gesichtspunkten."

[3] N. Faujdar and S. P. Ghrera, "Analysis and testing of sorting algorithms on a standard dataset," IEEE, 2015.

[4] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, pp. 847–857, 10 1978.

[5] `http://www.java-programmieren.com/bubblesort-java.php`.

[6] `http://www.java2novice.com/java-sorting-algorithms/selection-sort/`.

[7] `http://www.java-programmieren.com/insertionsort-java.php`.

[8] http://www.geeksforgeeks.org/merge-sort/.

[9] http://www.algolist.net/Algorithms/Sorting/Quicksort.

[10] http://www.geeksforgeeks.org/heap-sort/.

[11] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, "Insertion sort is o(n log n)," *Theory of Computing Systems*, 2006.

[12] O. O. Moses, "Bidirectional bubble sort approach to improving the performance of introsort in the worst case for large input size,"