Verbesserung von Quicksort

Dennis Sentler, Malte Janssen

20. November 2017

Zusammenfassung

Das Ziel dieser Arbeit ist die Verbesserung des Quicksort-Sortieralgorithmus.[1] Dieser wurde bereits implementiert und läuft effizient nach dem "Divide and ConquerPrinzip.
Allerdings hat der rekursive Algorithmus einen Nachteil bei kleinen zu sortierenden Folgen, denn diese werden immer weiter aufgeteilt und rekursiv bearbeitet bis nur noch ein Element übrig ist. Dieser Nachteil wurde durch eine zusätzliche Implementation eines weiteren Sortieralgorithmus minimiert.

Inhaltsverzeichnis

1 Verfahren

Die Effizienz als auch die Trägheit des Quicksort liegen in der Rekursion. Durch die Teilung muss durch immer weniger Elemente iteriert und verglichen werden. Allerdings geht die Rekursion bis in die kleinste Stufe und es werden auch zwei Elemente noch aufgeteilt, obwohl diese schneller durch einen direkten Vergleich sortiert werden können.

1.1 Insertion-Quicksort

Unsere Lösung ist das Ergänzen des Quicksort mit einem Insertionsort [2, (S. 56)]. Wird ein Array sortiert, so übernimmt Quicksort die grobe Vorarbeit und teilt das Problem, wie gewohnt, in viele kleine. Eine Fallunterscheidung im Quicksort-Algorithmus entscheidet nach der Problemgröße ob weiterhin Quicksort verwendet wird, oder Insertionsort. Durch empirische Untersuchungen konnten wir feststellen, dass diese Schwelle klein gewählt werden muss. Nach unseren Ergebnissen bringt ein Schwellwert von fünf bis 10 die höchste Effizienz.

1.1.1 Schwellwertproblematik

Ein zu hoher Schwellwert (ca. 50) macht den Algorithmus träge, denn wenn der Insertionsort-Algorithmus mit mit einem großen Pool von Elementen arbeitet, müssen alle Elemente unter einander verglichen werden, das treibt die Operationen in die Höhe. Durch die Wahl eines zu geringen Schwellwertes (unter 4) arbeitet der Quicksort ebenfalls nicht mehr schnell genug und vergeudet Operationszeit mit Aufteilungen.

2 Testen des Insertion-Quicksort

Getestet wurde der Algorithmus nach einem typischen Testverfahren für Sortieralgorithmen. Vor jedem Test wurde ein zufällig generiertes Array mit der Problemgröße 100 erstellt. Dieses wurde dem Sortieralgorithmus als Referenz übergeben. Im Anschluss wurde dieses Array mit einer Hilfsmethode Untersucht, ob alle Elemente im Array kleiner oder gleich sind zu dem Folgeelement:

$$array[i] =< array[i+1]$$

Außerdem wurde dieser Test mit den drei implementierten Pivot-Entscheidverfahren durchgeführt.

3 Aufwandsanalyse

Um den Aufwand des Algorithmus zu messen wurden drei Größen untersucht: Die Programmdauer in Nanosekunden, die Anzahl der Arrayoperationen und die Anzahl der Vergleiche. Für jede Arraygröße $N = 10^k (k = 1...6)$ wurden 10 Tests durchgeführt und ein Durchschnitt dieser Ergebnisse lieferte aussagekräftige und vom Zufall unabhängige Werte. Darüber hinaus wurde stets mit dem ursprünglichen Quicksort verglichen um eine Verbesserung darstellen zu können. Die Implementationen des Quicksorts und auch des Quicksort-Bestandteiles vom weiterentwickelten Algorithmus sind identisch und arbeiten beide mit dem 'Median von drei' - Pivotelement[1].

3.1 Vergleiche

Die Anzahl der Vergleiche fällt relativ gleich aus, tatsächlich wird mit dem neuen Sortierverfahren mehr verglichen, als mit dem üblichen Quicksort. Bei einer Problemgröße von 10^1 fällt ein deutlicher Mehraufwand auf.

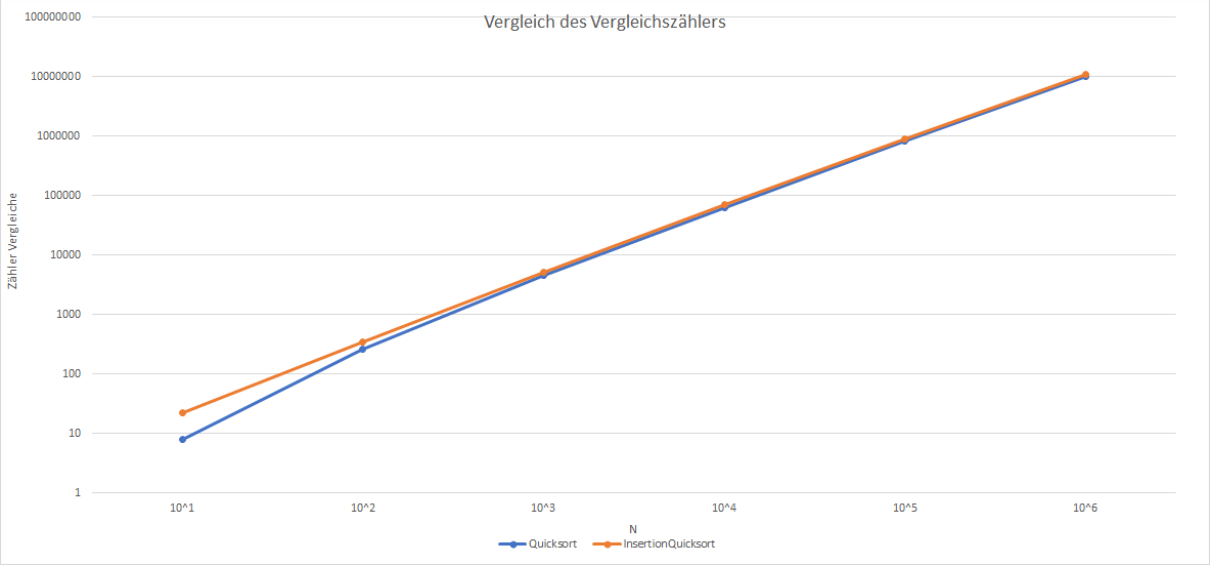

Abbildung 1: Anzahl der Vergleiche

3.2 Arrayoperationen

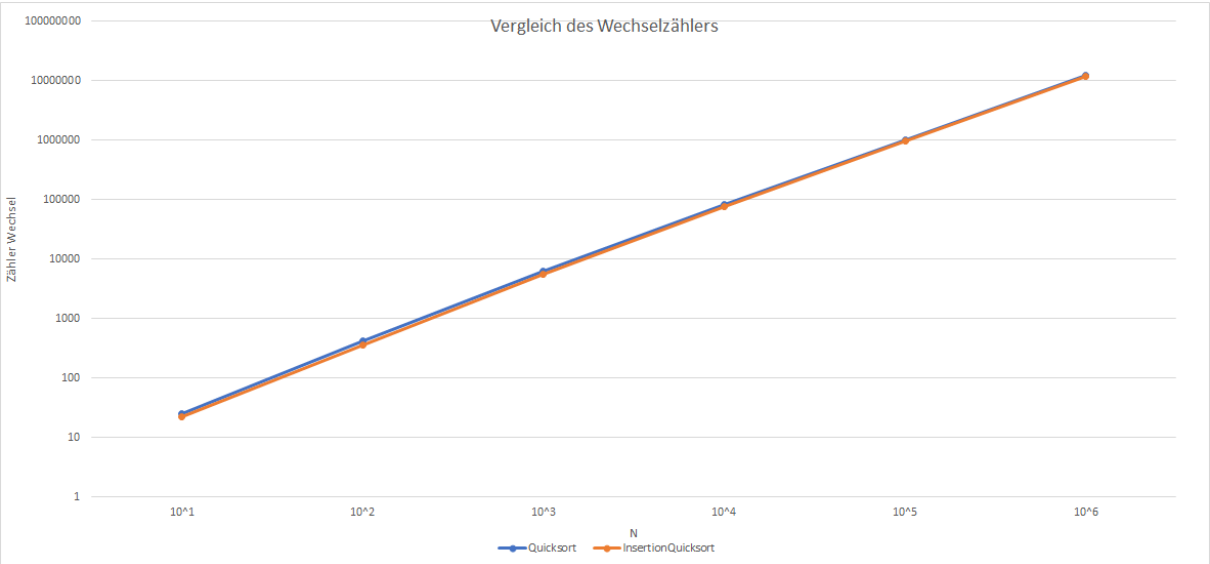

Abbildung 2: Anzahl der Arrayoperationen

Unter den Arrayoperationen werden die Operationen zusammengefasst die dafür Zuständig sind die Elemente auf dem Array zu verschieben. Dieser sieht bei beiden Verfahren ebenfalls konstant und linear aus. Allerdings lässt sich bei dem weiterentwickelten Verfahren eine Einsparung von durchschnittlich 10% an Arrayoperationen (Siehe Abb. 6 auf Seite 6)

erzielen. Mit zunehmender Gesamtgröße des Arrays wirkt sich dieser Effekt weniger stark aus.

3.3 Rechenzeit

Die entscheidende Größe ist allerdings die tatsächlich gebrauchte Rechenzeit, denn Vergleiche und Arrayoperationen können unterschiedlich komplex ausfallen und können im schlimmsten Fall sogar weitere Operationen verstecken, die im Hintergrund tatsächlich ausgeführt werden. Deswegen wurde ebenfalls noch eine, in Java implementierte, Stoppuhr genutzt um den kompletten Sortierprozess zu in Nanosekunden zu messen.

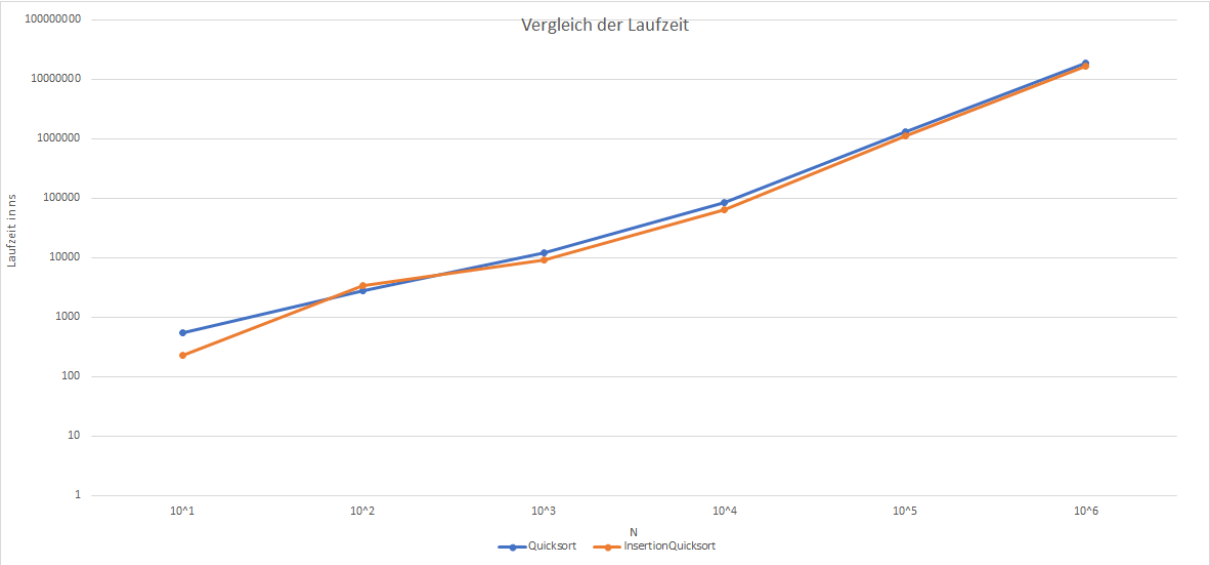

Abbildung 3: Anzahl der Arrayoperationen

Das Diagramm zeigt für beide Varianten ebenfalls einen linearen Aufwand. In der Regel wird der weiterentwickelte Algorithmus um eine gewisse Zeit schneller fertig (Siehe Abb. 6 auf Seite 6). Bei sehr kleinem Basisarray wurde eine Rechenzeitersparnis von 57% errechnet. Bei zunehmender Problemgröße ist dieser Effekt ebenfalls schwächer geworden. Außerdem gab es einen konstanten und deutlichen Ausreißer in den Messergebnissen: bei einer Größe von $N = 10^2$ wurde stets ein, dem Trend gegen-laufender, zusätzlicher Mehraufwand gemessen. Dieser blieb auch bei mehreren Wiederholungen der Tests und auch bei Veränderungen des Schwellwertes erhalten.

3.4 Tabellen

N	Zähler Vergleiche	Zähler Wechsel	Laufzeit in ns
10^1	8	25	538
10^2	267	421	2822
10^3	4580	6266	12213
10^4	64007	82120	85854
10^5	828276	1020924	1293339
10^6	10312375	12362139	18623217

Abbildung 4: Aufwand des Quicksort

N	Zähler Vergleiche	Zähler Wechsel	Laufzeit in ns
10^1	22	22	231
10^2	344	363	3335
10^3	5245	5541	9032
10^4	70899	75086	65225
10^5	896132	950705	1122707
10^6	10990550	11658739	16853339

Abbildung 5: Aufwand des Insertion-Quicksort

N	Zähler Vergleiche	Zähler Wechsel	Laufzeit in ns
10^1	175%	-12%	-57%
10^2	29%	-14%	18%
10^3	15%	-12%	-26%
10^4	11%	-9%	-24%
10^5	8%	-7%	-13%
10^6	7%	-6%	-10%
DS	41%	-10%	-19%

Abbildung 6: Prozentuale Entwicklung

4 Ergebnis

Als Ergebnis lässt sich diese Weiterentwicklung des Quicksortvorfahrens als durchaus positiv bewerten. In vielen Punkten wurde eine Aufwandseinsparung gesehen und verdeutlicht. Vor allem die Arrayoperationen und die rechenintensiven, kleinformatigen Rekursionen konnten reduziert werden. Das spiegelt sich im Endeffekt in der Gesamtlaufzeit des Algorithmus wieder, bei der durchschnittlich 19% an Rechenzeit eingespart werden konnte.

Literatur

[1] M. Janssen, D. Sentler, Quicksort und Pivotelementwahl, Hochschule für angewandte Wissenschaften Hamburg, 11/2017

[2] S. Pareigis, Algorithmen und Datenstrukturen für Technische Informatiker, Vorlesungsskript, Hochschule für angewandte Wissenschaften Hamburg, WS 2017/18