Hybrid-Quicksort

Finn Jannsen, Philipp Schwarz

13. Mai 2019

Inhaltsverzeichnis

1 Einführung

Diese Dokumentation beschreibt eine Sortieralgorithmus-Implementation für Arrays, basierend auf Quicksort. Für den Algorithmus wurden Optimierungen ausgeführt, die reduzierte Laufzeit und effiziente Resourcennutzung zum Ziel haben. In Abschnitt 2 wird darauf eingegangen, wie der Algorithmus realisiert wurde. Anschließend wird in Abschnitt 3.1 geprüft, ob die Implementation korrekt funktioniert und in Abschnitt 3.2 die Performance mit dem zu Grunde liegenden Quicksort verglichen.

2 Implementation

Der Algorithmus wurde als Kasse realisiert, der ein Interface implementiert, welches es ermöglicht, einfach weitere Implementationen, sofern dies gewünscht wird, zu schreiben.

2.1 Pivotauswahl

Bei der Untersuchung von Quicksort stellte sich eine Methode zur Pivot-Auswahl am effizientesten auf: Median-of-three. Der mittlere Wert von mehreren Key-Werten (bei Bedarf auch von mehr als 3) sorgt für eine gleichmäßigere Aufteilung der Partitionierung beim Sortieren. Dadurch finden im allgemeinen weniger Iterationen als bei anderen Pivots, wie z.B. Random oder Right statt, was wünschenswert ist.

Der Code ist in 3.5 dargestellt. Eins von 3 Elementen wird in die Mitte getauscht und als Pivot ausgewählt.

2.2 Sortieren

Der bisherige Quicksort-Algorithmus besteht im wesentlichen aus einer Schleife, in der zuerst das erste Element von links, welches gleich oder größer als Pivot ist, gesucht wird. Danach wird das erste Element von rechts, welches kleiner als das Pivot ist, gesucht. Diese werden dann getauscht. Die Schleife wird verlassen, wenn die jeweiligen Such-Schleifen einander kreuzen. Danach wird das Pivot in diese Mitte an Stelle i zurückgetauscht und der Algorithmus wird zwei mal rekursiv gestartet, einmal um die Liste von ganz links bis i-1 zu sortieren und noch einmal um die Liste von i+1 bis rechts zu sortieren.

Für dieses Hybrid-Quicksort wurde eine Implementation von Quicksort mit Partitionierung in 3 Teile ausgewählt. Das Pivot steht hier anfänglich links. Nun wird der Array von einer Seite an durchiteriert. Ist ein Element kleiner, so wird es nach links getauscht und ein Counter erhöht, sodass das nächste kleinere Element daneben platziert würde. Ist ein Element größer, wird es nach rechts getauscht und ein Counter verringert, um das nächste größere Element daneben zu platzieren. Wenn es gleich dem Pivot ist, so bleibt es stehen. Dadurch überspringt man redundantes Tauschen duplizierter Elemente und durch das Tauschen kleinerer Elemente wird das Pivot automatisch immer wieder in die Mitte getauscht. Danach wird wieder mit den linken und rechten Teilen des Pivots rekursiv sortiert.

Der Code hierfür ist in 3.6 dargestellt.

2.3 Optimierung

Der Algorithmus wurde weiterhin mit Insertionsort bei einer Anzahl an Elementen kleiner als 10 optimiert. Durch diese Praxis wird ein hoher Overhead bei kleinen Arrays durch die rekursiven Aufrufe verhindert. In der Praxis wurden verschiedene Werte für die für den Aufruf von Insertionsort benötigte Anzahl an Elementen getestet, sowohl mit sehr vielen Duplikaten (Begrenzung der Keys auf 100 Werte), als auch mit wenigen. Dies ist zu sehen in Abbildung 3.7. Mit einer Versuchsgröße von 100 zeigt ein recht genauer Durchschnittswert eine eindeutige Verbesserung der Performance mit 10 Insertions.

3 Testen und Verifikation

3.1 Verifizieren

Der Algorithmus wurde auf seine korrekte Funktionalität getestet. Hierzu zählt natürlich das Vorliegen des Ergebnisses in der korrekten Reihenfolge. Alle Tests wurden erfolgreich mit unterschiedlichen Eingabewerten absolviert.

3.2 Aufwandsanalyse

Für die Aufwandsanalyse wurde der Klasse ein Counter eingeführt, der die Tauschoperationen zählt, als auch die Ausführungszeit beobachtet.

Bisherige Untersuchungen von Quicksort haben bereits einen Asymptotischen Aufwand wie folgt ergeben:

Rechenoperationen bei Best- und Average Case:

$$T(n) = \mathcal{O}(n * ln(n))$$

Rechenoperationen bei Worst Case:

$$T(n) = \mathcal{O}(n^2)$$

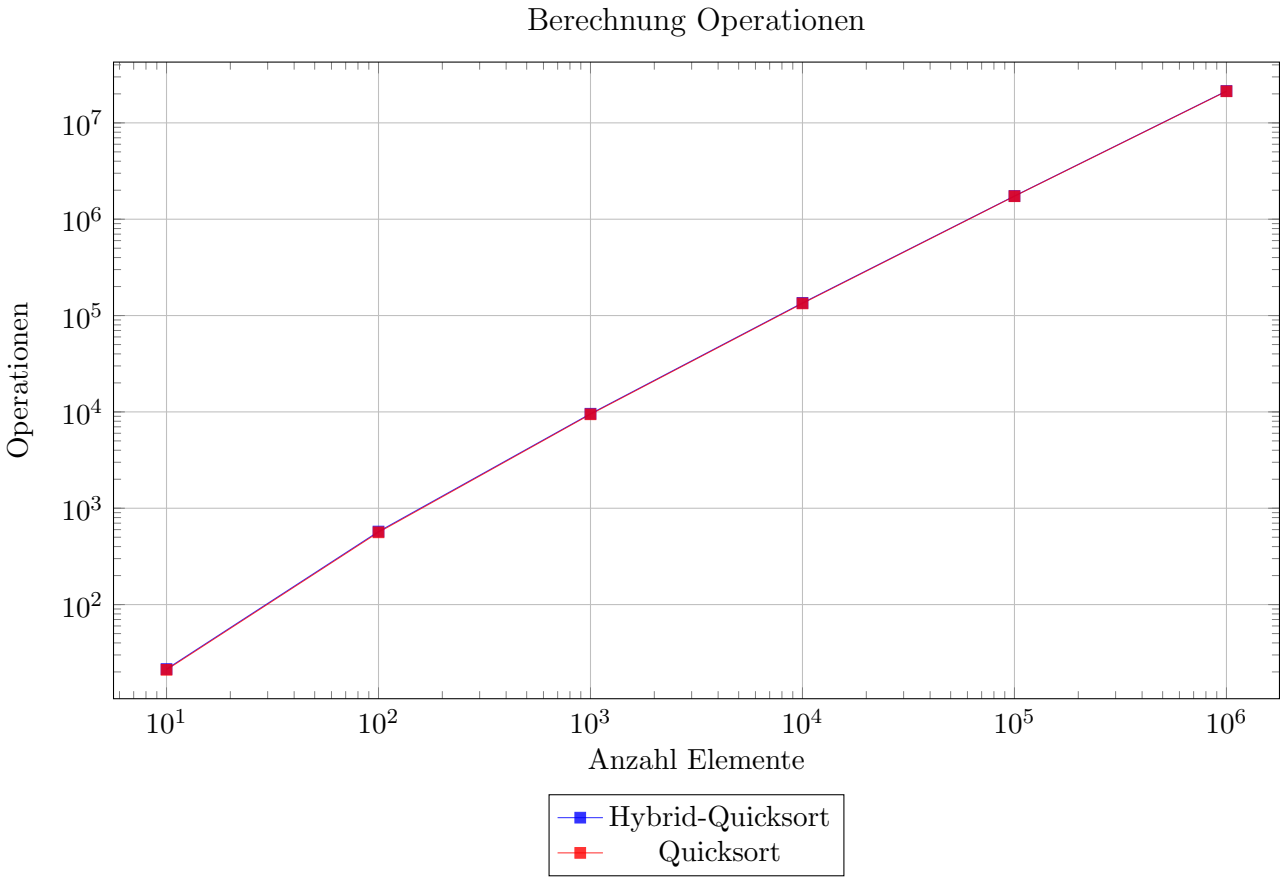

Abbildung 3.1: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen

Da sich der Rechenaufwand für Worst Case bei unserer Implementation sich nicht ändert und dieser auch besonders selten ist, wurden dieses mal zufällig Sortierte Arrays beobachtet. Außerdem wurden dieses mal statt komplett zufälligen Key-Werten welche mit folgender Beschränkung ausgewählt:

$$700 * N \leq key \leq 800 * N$$

Dies hat zur Folge, dass in der zu sortierenden Menge vermehrt duplizierte Keys auftreten. Um unter anderem mit dieser Hürde besser umgehen zu können wurden die in 2.3 angesprochenen Optimierungen eingeführt. Es wurden 100 Durchläufe des Tests mit den Größen $N = 10^k, k = 1, ..., 6$ durchgeführt, um einen angemessenen Mittelwert zu ermitteln.

Es wurden zuerst viele einzigartige Keys verwendet, um die allgemeine Performance zu vergleichen. Die Ergebnisse für die Rechenoperationen sind in Abbildung 3.1, für die Laufzeit in 3.2 zu sehen. Es ist ersichtlich, dass das Hybrid-Quicksort womöglich durch Insertionsort eine wenig höhere Anzahl an Rechenoperationen hat, dafür aber einen zeitlichen Vorteil erzielt. Dies liegt an der effizienteren Nutzung der Resourcen.

Diese Versuche wurden auch für eine vermehrte Anzahl an Duplikaten (max. 100 eindeutige keys) durchgeführt, was zeigt, wie effizient Hybrid-Quicksort im Gegensatz zu normalem Quicksort mit dieser Hürde umgehen kann. Die Ergebnisse sind in Abbildung 3.3 und 3.4 zu sehen.

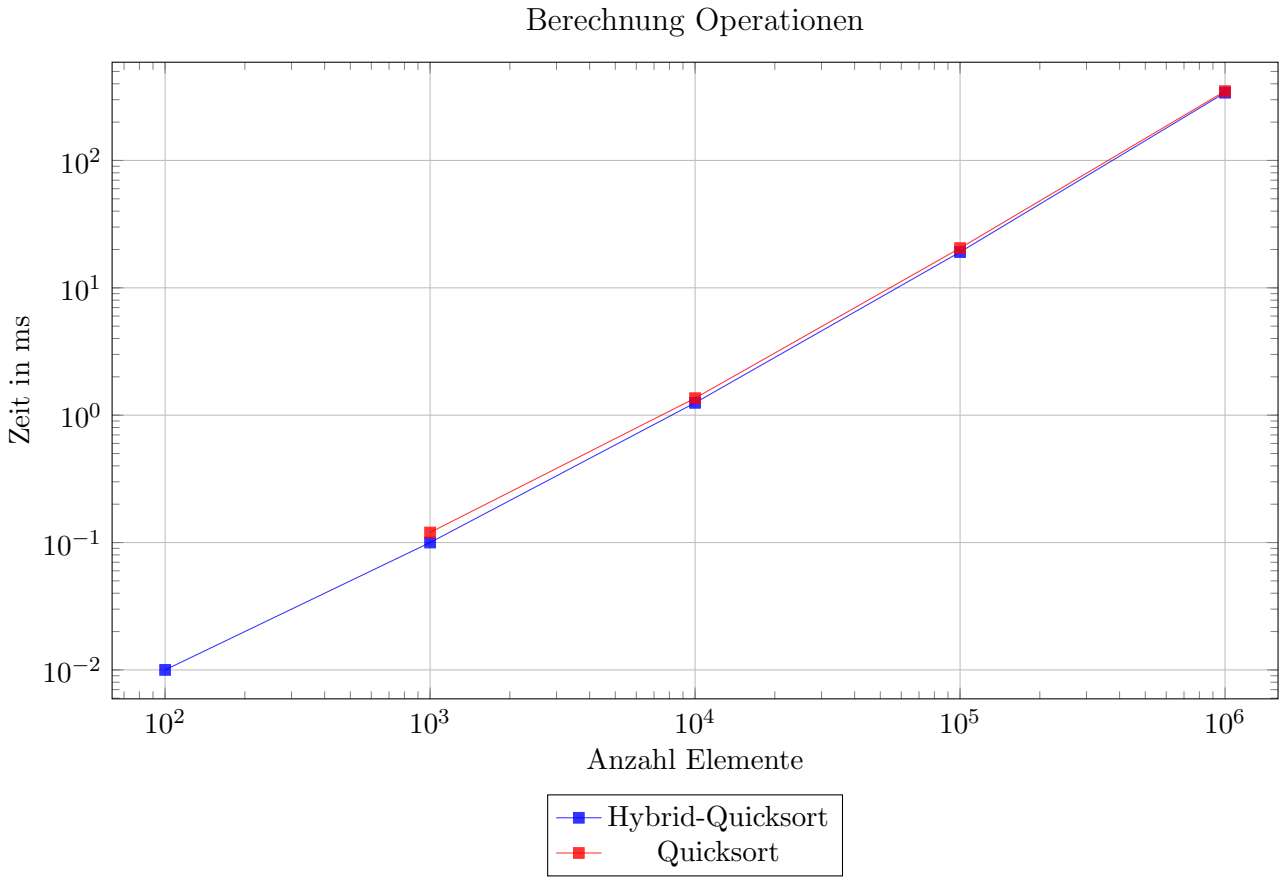

Abbildung 3.2: Quantitativer Vergleich zu Quicksort anhand Laufzeit

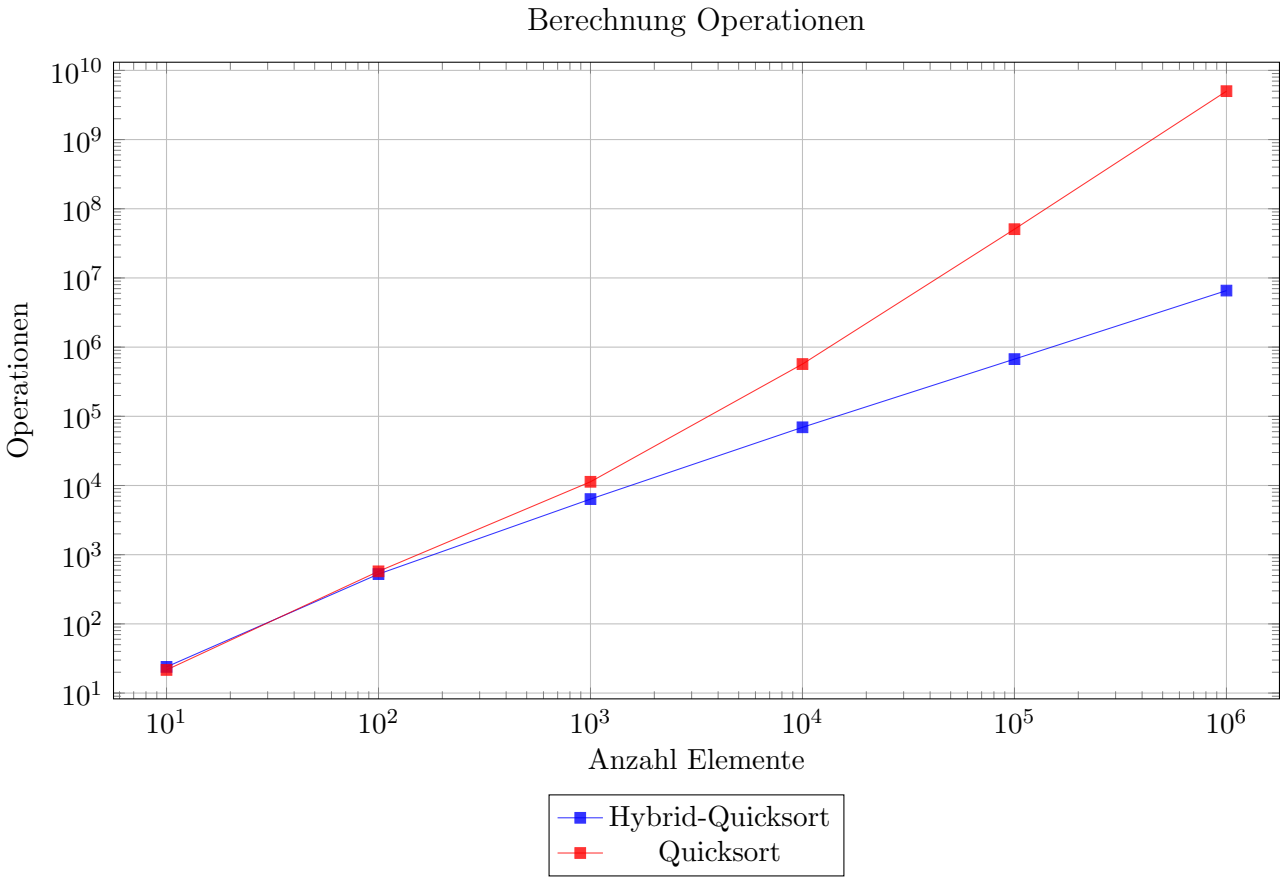

Abbildung 3.3: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen mit vermehrten Duplikaten

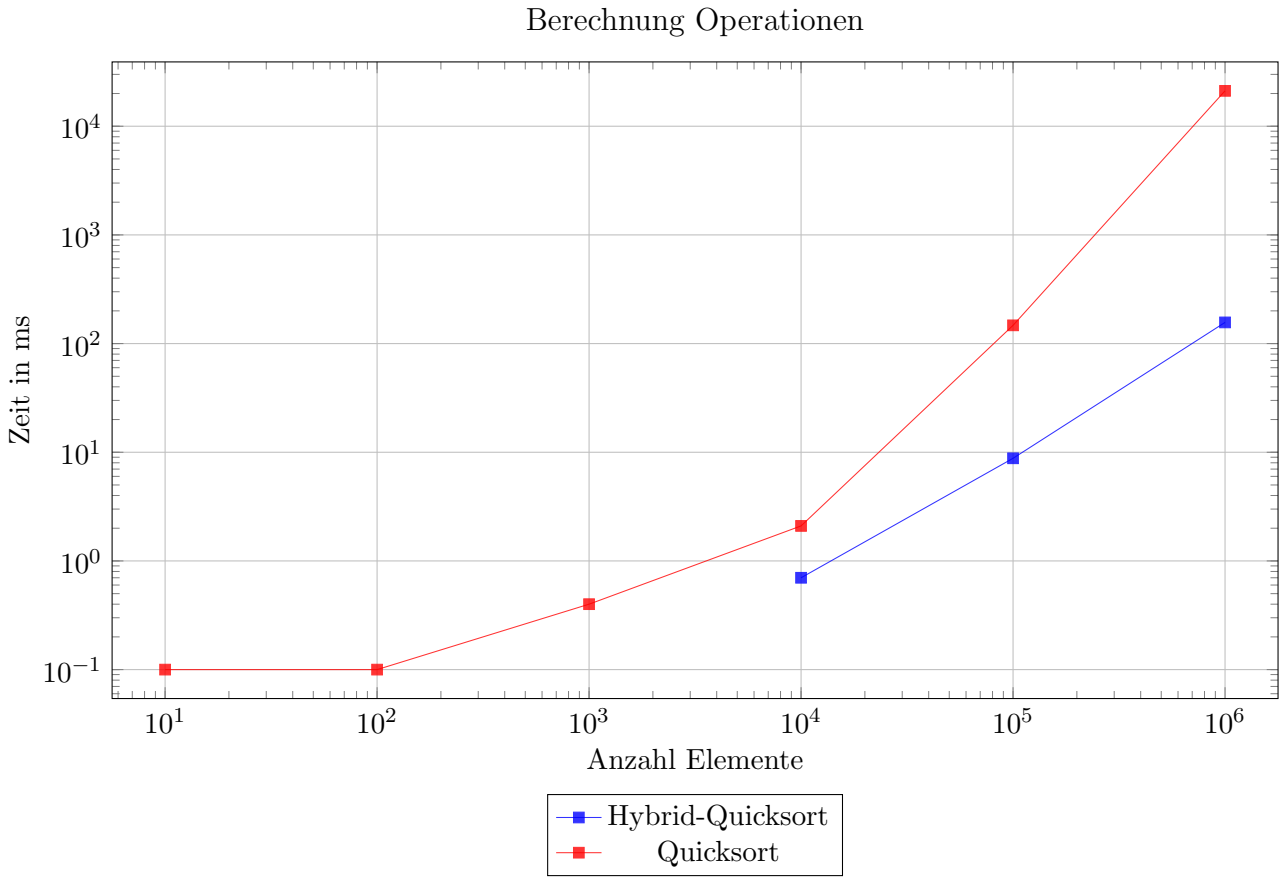

Abbildung 3.4: Quantitativer Vergleich zu Quicksort anhand Laufzeit mit vermehrten Duplikaten

```
//find pivot with median of 3
int center = (left + right) / 2;

if (list[left].getKey() > list[center].getKey()) {
    increaseCounter();
    swap(list, left, center);
}
if (list[left].getKey() > list[right].getKey()) {
    increaseCounter();
    swap(list, left, right);
}
if (list[center].getKey() > list[right].getKey()) {
    increaseCounter();
    swap(list, center, right);
}
```

Abbildung 3.5: Code-Ausschnitt Pivotsuche

```
int low = left, high = right;
int pivot = list[left].getKey();
int i = left;

while (true) {
    if (list[i].getKey() < pivot) {
        // smaller than pivot, accumulate from the left and eventually shift pivot to i
        swap(list, i, low);
        i++;
        low++;
        Counter++;
    } else if (list[i].getKey() > pivot) {
        // bigger than pivot, accumulate from the right
        swap(list, i, high);
        high--;
        Counter++;
    } else {
        // ran over pivot, skip
        i++;
        Counter++;
    }
    if (i > high) {
        // don't run over already iterated elements
        break;
    }
}
```

Abbildung 3.6: Code-Ausschnitt Sortieren

Viele Duplikate

Kein Insert			Insert 10			Insert 100		
N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs
10	29.64	0	10	23.8	0	10	23.25	0.01
100	571.3	0.02	100	513.23	0	100	2447.28	0
1000	6587.53	0.02	1000	6419.15	0.07	1000	16357.26	0.11
10000	66579.74	0.44	10000	66548.55	0.5	10000	64931.09	0.66
100000	665092.01	8.15	100000	668592.79	7.81	100000	662118.18	8.03
1000000	6649592.4	161.02	1000000	6611998.45	147.22	1000000	6695958.91	159.14

Wenig Duplikate

Kein Insert			Insert 10			Insert 100		
N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs
10	30.74	0	10	21.87	0	10	22.04	0
100	657.5	0.02	100	577.57	0.02	100	2464.69	0
1000	10433.3	0.11	1000	9568.93	0.09	1000	21261.65	0.15
10000	143825.56	1.3	10000	135667.7	1.23	10000	253308.34	1.41
100000	1827608.13	19.72	100000	1746850.13	19.47	100000	2922396.04	21.43
1000000	22316105.8	343.26	1000000	21383728.4	338.14	1000000	33143857.1	356.4

Abbildung 3.7: Durchschnittswerte unterschiedlicher Insertionsorts von 100 Durchführungen bei vielen und wenigen Duplikaten