Hybrid-Quicksort

Finn Jannsen, Philipp Schwarz

11. 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 3.3 dargestellt. Eins von 3 Elementen wird in die Mitte getauscht und als Pivot ausgewählt.

2.2 Sortieren

Der Algorithmus der Sortiert ist im wesentlichen eine 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.

Der Code ist in 3.4 dargestellt.

2.3 Optimierung

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.

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:

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

Dieses mal wurden 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 vermehr duplizierte Keys auftreten. Um unter anderem mit dieser Hürde besser umgehen zu können wurden die in 2.3 angesprochenen Optimierungen eingeführt. Es wurden 10 Durchläufe des Tests mit den Größen $N = 10^k, k = 1, ..., 6$ durchgeführt, um einen angemessenen Mittelwert zu ermitteln. Die Beobachtungen der Rechenoperationen und Laufzeit sind in den Abbildungen 3.1 und 3.2 zu sehen.

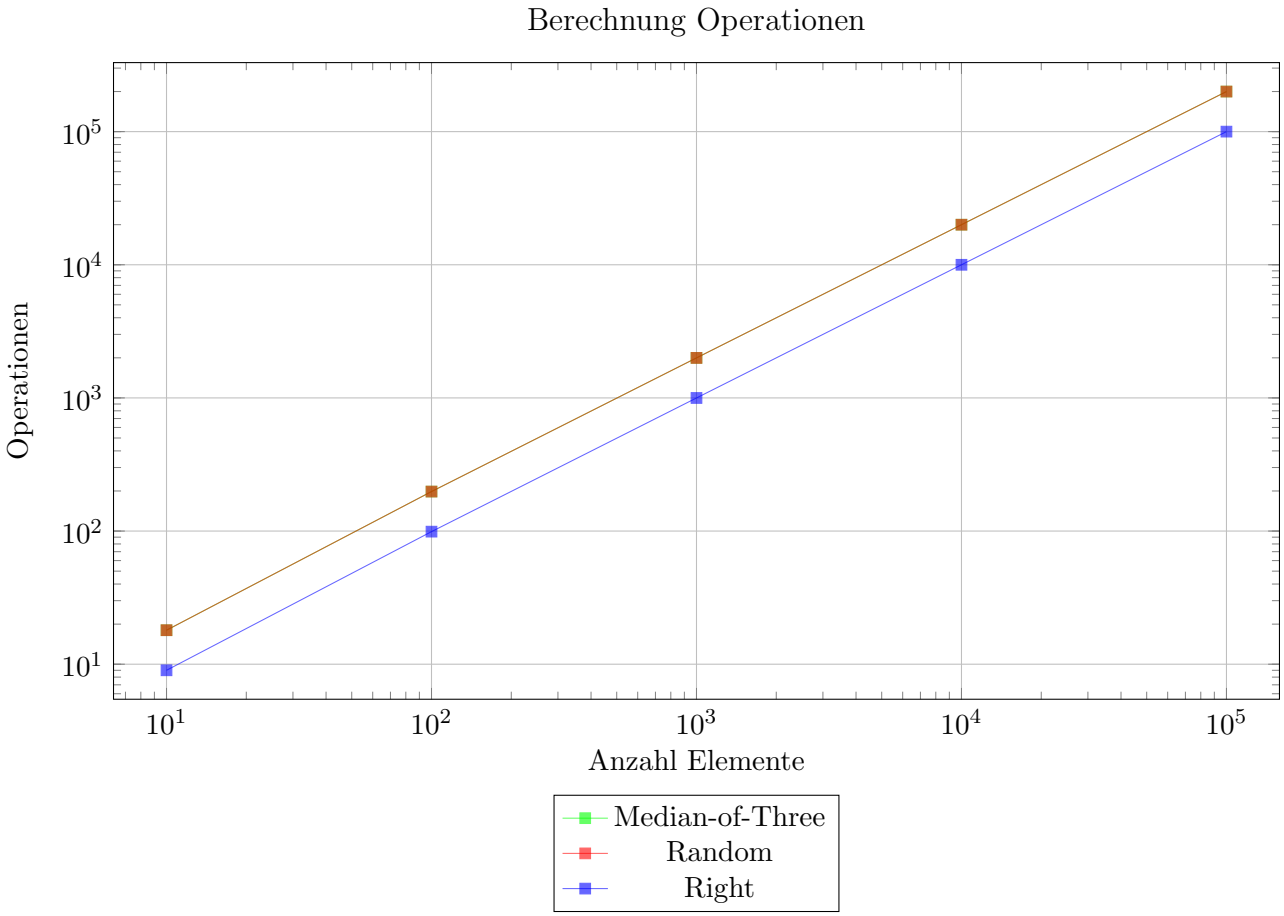

Abbildung 3.1: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen

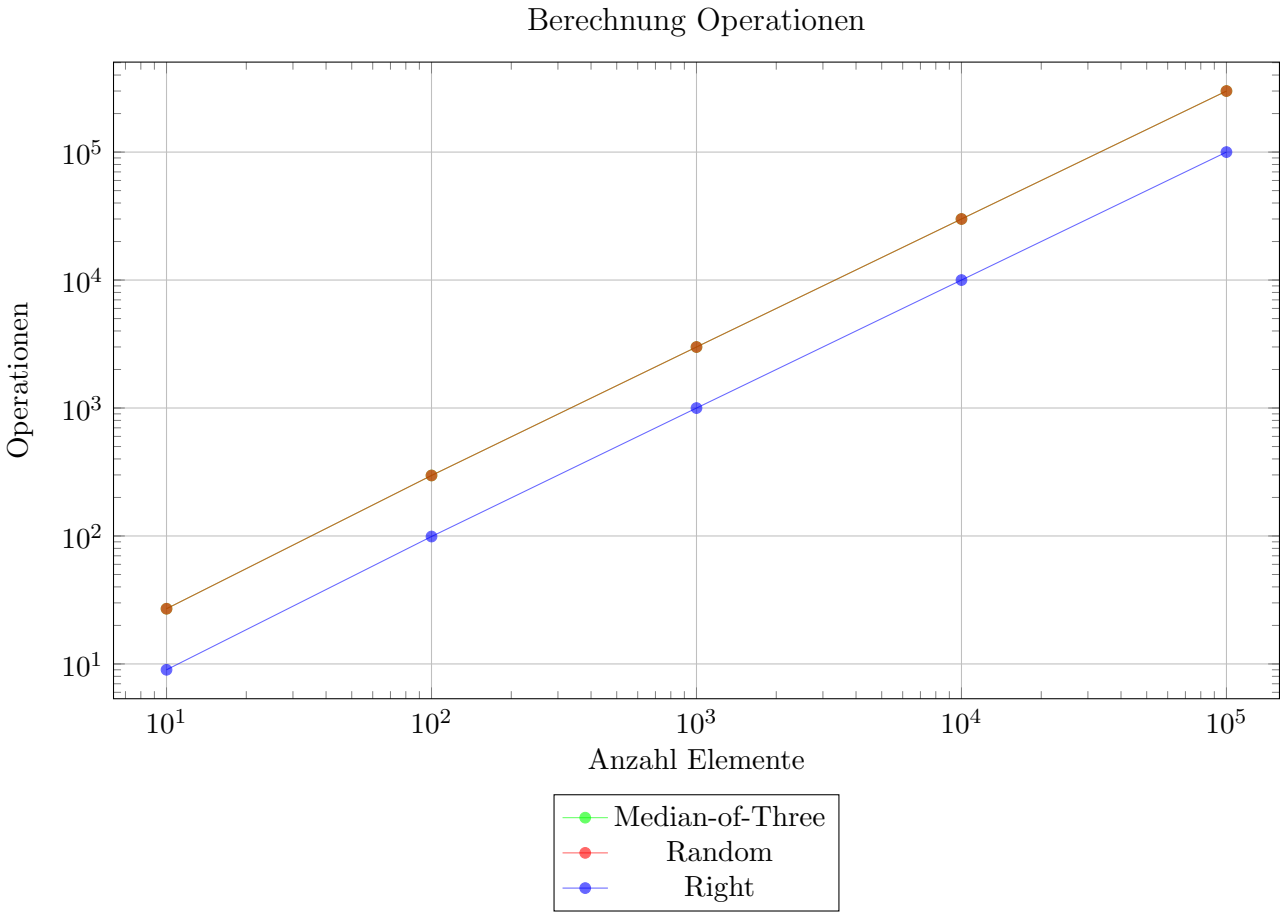

Abbildung 3.2: Quantitativer Vergleich zu Quicksort anhand Laufzeit

```
//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 3.3: Code-Ausschnitt Pivotsuche

```
iterationLoop:
while (true) {
    // increase i, which starts as the smallest index, until the Node it indexes is equal
    while (list[i].getKey() < list[pivot].getKey()) {
        increaseCounter();
        i++;
    }

    // decrease j, which starts as the largest index, until the Node it indexes is smalle
    while (j > i && list[j].getKey() >= list[pivot].getKey()) {
        increaseCounter();
        j--;
    }

    // break out of the loop when we iterated over every entry
    if (i >= j) {
        break iterationLoop;
    }
    // swap i, which points at an element larger than the pivot with j, which points at a
    swap(list, i, j);
}

// swap the pivot to the right part
swap(list, i, right);
// sort everything right and left of the pivot
sort(list, left, i-1, pivotType);
sort(list, i+1, right, pivotType);
```

Abbildung 3.4: Code-Ausschnitt Sortieren