Java-Bibliotheken für Physik-Projekte S. 27
Agile Serie: Liefern und besser werden S. 66
JavaFX auf dem Raspberry Pi S. 93

JAVA Mag
Java magazin™
Java | Architektur | Software-Innovation

17 gewinnt!

Die neue Java-Version im Überblick

Ausgabe 11.2021
Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

www.javamagazin.de

Der richtige Algorithmus am richtigen Ort – binäre Suche und Sortierverfahren

Skalierbar programmieren

Java führt kontinuierlich neue, nützliche Features ein – die Einführung des Stream API in Java 8 war zum Beispiel eins der größten Highlights der vergangenen Jahre. Aber ist die Aggregierung der Daten mit dem Stream API ein Universalheilmittel? In diesem Artikel möchte ich untersuchen, ob es in bestimmten Fällen eine bessere Alternative gibt – aus Sicht der Komplexität.

von Ikuru Otomo

Einige von euch haben wahrscheinlich schon folgenden Code in einem Programm verwendet, um spontan die Laufzeit einer Logik zu messen:

```
long start = System.currentTimeMillis();
doSomething();
long time = System.currentTimeMillis() - start;
```

Klar, das ist einfach zu implementieren und man kann die Geschwindigkeit des Codes schnell überprüfen. Allerdings gibt es auch Nachteile. Erstens können die Messwerte Ungewissheiten enthalten, weil andere Prozesse, die auf der gleichen Maschine laufen, sie beeinflussen können. Zweitens kann man die Messwerte nicht mit anderen Messwerten vergleichen, die in unterschiedlichen Umgebungen gemessen wurden. Es ist nicht hilfreich festzustellen, dass eine Lösung schneller als die andere ist, wenn sie auf unterschiedlichen Maschinen gemessen wurden, die zum Beispiel unterschiedliche CPUs und RAM haben. Drittens ist es schwer abzuschätzen, wie die Laufzeit sich verlängern würde, falls man zukünftig mit einer größeren Menge an Daten arbeiten würde. Seit das Stream API in Java 8 eingeführt wurde, ist es viel einfacher geworden, die Daten zu filtern und zu aggregieren. Das Stream API bietet sogar die Möglichkeit, die Bearbeitung zu parallelisieren [1]. Sind aber diese Lösungen weiter performant, wenn man mit der 10- oder 100-fachen Menge an Daten arbeiten muss? Gibt es ein Maß, mit dem wir solche Fragestellungen beantworten können?

Zeitkomplexität

Die Zeitkomplexität ist ein Maß, um die zeitliche Effizienz eines Algorithmus grob zu schätzen. Sie fokussiert, wie die Laufzeit zunimmt, sobald die Eingabe länger wird. Wenn man zum Beispiel eine Liste mit n Elementen mit einem *for*-Loop iteriert, dann werden n und die Laufzeit eine lineare Beziehung haben. Wenn man mehrere *for*-Loops hat, die geschachtelt sind und jeweils n-Mal ausgeführt werden, dann wird diese Logik eine exponentielle Auswirkung auf die Laufzeit haben.

Die O-Notation ist eine Möglichkeit, um die Beziehung zwischen der Länge der Eingabe und der Laufzeit darzustellen. Eine lineare Beziehung stellt man mit $O(n)$ dar, $O(n^2)$ stellt ein quadratisches Verhältnis dar, wobei n die Länge der Eingabe ist. Ist die Laufzeit unabhängig von der Länge der Eingabe, also konstant, dann schreibt man $O(1)$. **Abbildung 1** stellt die typischen Werte der O-Notationen dafür dar, wie die Laufzeit wachsen wird, wenn sich die Länge der Eingabe vergrößert.

Es gibt zwei wichtige Regeln für die Darstellung mit Hilfe der O-Notation:

- Nur der Term mit dem höchsten Grad wird berücksichtigt. Beispiel: Wenn die Zeitkomplexität $n + nlogn + n^2$ ist, schreibt man einfach $O(n^2)$, da der Term n^2 die stärkste Auswirkung auf die Laufzeit hat.
- Der Koeffizient wird nicht berücksichtigt. Beispiel: Die Zeitkomplexität von $2n^2$, $3n^2$ und $½n^2$ ist gleichermaßen $O(n^2)$.

Wichtig zu betonen ist, dass sich die Zeitkomplexität nur auf die Skalierbarkeit fokussiert. Besonders wenn n

ein kleinerer Wert ist, kann es möglicherweise passieren, dass ein Algorithmus eine längere Laufzeit hat, obwohl er eine bessere Zeitkomplexität als die anderen aufweist.

Platzkomplexität

Ergänzend zur Zeitkomplexität gibt es ein weiteres Maß, um die Effizienz eines Algorithmus darzustellen: die Platzkomplexität. Diese betrachtet, wie der Speicherbedarf wächst, wenn sich die Länge der Eingabe vergrößert. Wenn man eine Liste mit *n* Elementen in eine neue Liste kopiert, dann ist die Platzkomplexität *O(n)*, weil der Bedarf an zusätzlichem Speicher linear steigt, wenn man mit einer größeren Eingabeliste arbeitet. Braucht ein Algorithmus nur eine konstante Menge Speicher, unabhängig von der Länge der Eingabe, dann ist die Platzkomplexität *O(1)*.

Es gibt oft eine Trade-off-Beziehung zwischen Zeitkomplexität und Platzkomplexität. Je nach Fall ist es wichtig zu überlegen, ob die Laufzeit oder der Speicher wichtiger ist, wenn man mehrere Algorithmen vergleicht.

Binäre Suche

Wie **Abbildung 1** zeigt, hat ein Algorithmus mit der Zeitkomplexität *O(logn)* eine bessere zeitliche Performance als *O(n)*. Die binäre Suche ist einer der Algorithmen, die diese Zeitkomplexität aufweist, und sie ist anwendbar, wenn man einen Zielwert aus einer sortierten Liste suchen möchte. Der Algorithmus vergleicht in jedem Vorgang, ob sich der Zielwert in der linken oder der rechten Hälfte des Suchbereichs befindet. Man kann sich als Beispiel ein Wörterbuch vorstellen: Man wird wahrscheinlich nicht auf der ersten Seite des Wörterbuches anfangen, um das gesuchte Wort zu finden, sondern man wird eine Seite in der Mitte des Buches öffnen und von dort aus die Suche starten.

Abbildung 2 stellt grafisch dar, wie die binäre Suche ablaufen wird, wenn man den Zielwert 7 in einer Liste mit elf Elementen sucht. Das rot markierte Element stellt die Mitte des Suchbereichs des jeweiligen Vorgangs dar. Ist die Anzahl der Elemente des Suchbereichs eine gerade Zahl, dann nimmt man das „linke" Element in der Mitte. In jedem Vorgang vergleicht man, ob der Zielwert (also in diesem Fall die 7) kleiner oder größer als die Mitte ist, und halbiert den Suchbereich, bis man den Zielwert erreicht hat.

Die maximale Anzahl der nötigen Vergleichsvorgänge, um den Zielwert mit der binären Suche zu finden, ist $log_2 n$, während *n* die Länge der Eingabeliste ist. Lasst uns $n = 8$ als Beispiel nehmen: Die Länge des Suchbereichs beginnt mit *8* und verringert sich nach dem ersten Vorgang auf *4*. Nach dem zweiten Vorgang wird sie noch einmal auf *2* halbiert und nach dem dritten Vorgang gibt es nur noch einen einzigen Wert im Suchbereich. Aus diesem Beispiel kann man schlussfolgern, dass die Anzahl der nötigen Vorgänge höchstens ein Logarithmus von 8 zur Basis 2 ist $(log_2 8 = 3)$, denn $2^3 = 8$. In der O-Notation lässt man die Basis aus und schreibt nur *O(logn)*.

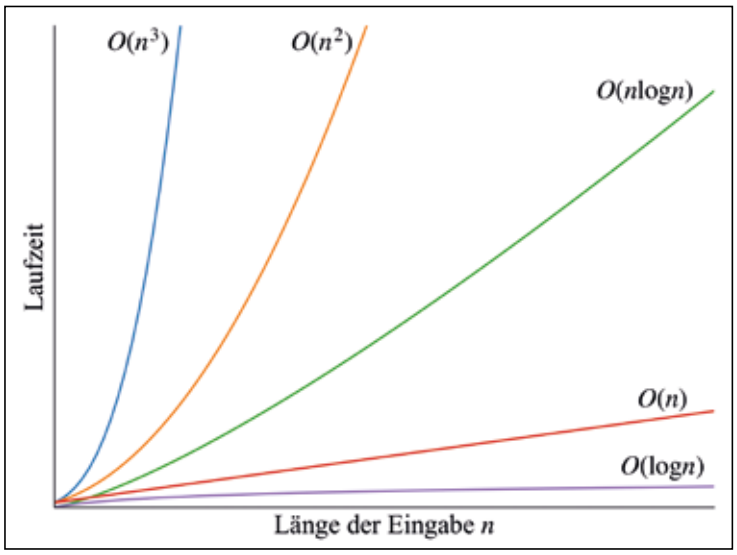

Abb. 1: Die Beziehung zwischen der Laufzeit und der Länge der Eingabe je Zeitkomplexität

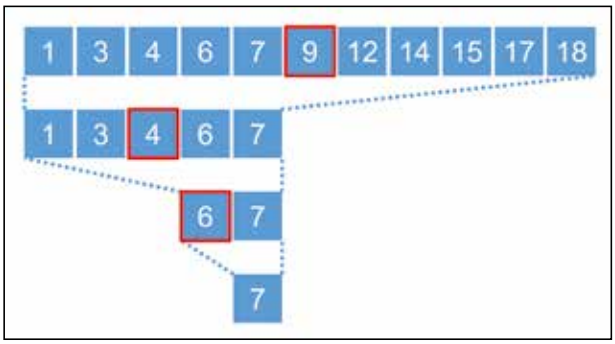

Abb. 2: Ablauf der binären Suche

In Java sind die Implementierungen der binären Suche in den Methoden *java.util.Arrays.binarySearch* [2] und *java.util.Collections.binarySearch* zu finden [3]. Wenn man mit einem Array arbeitet, kann man die Methoden in der Klasse *java.util.Arrays* benutzen. Arbeitet man mit einer Liste, dann sind die Methoden in der Klasse *java.util.Collections* anwendbar.

Sortierverfahren

Es gibt mehrere Arten von Sortierverfahren, die jeweils unterschiedliche Zeitkomplexitäten und Platzkomplexitäten haben. Typische Sortierverfahren, die in der Praxis verwendet werden, sind Quicksort, Mergesort und deren Varianten. Die Zeitkomplexität dieser beiden Verfahren ist im Durchschnitt *O(nlogn)* [4], [5]. Es gibt auch Sortierverfahren, die über bessere Zeitkomplexitäten verfügen, allerdings weisen diese meistens Einschränkungen in der Anordnung der Eingabeliste auf oder benötigen spezielle Hardware.

In Java sind die Methoden für die Sortierverfahren in *java.util.Arrays.sort* [2] und *java.util.Collections.sort* implementiert [3]. Seit Java 8 bietet das *List*-Interface auch die Methode *sort* [6], und das Stream API hat auch die intermediäre Operation *sorted* [1]. Nach der Java-Dokumentation sind diese Methoden standardmäßig mit Quicksort, Timesort oder Mergesort implementiert, allerdings kann das je nach Anbieter des JDK variieren.

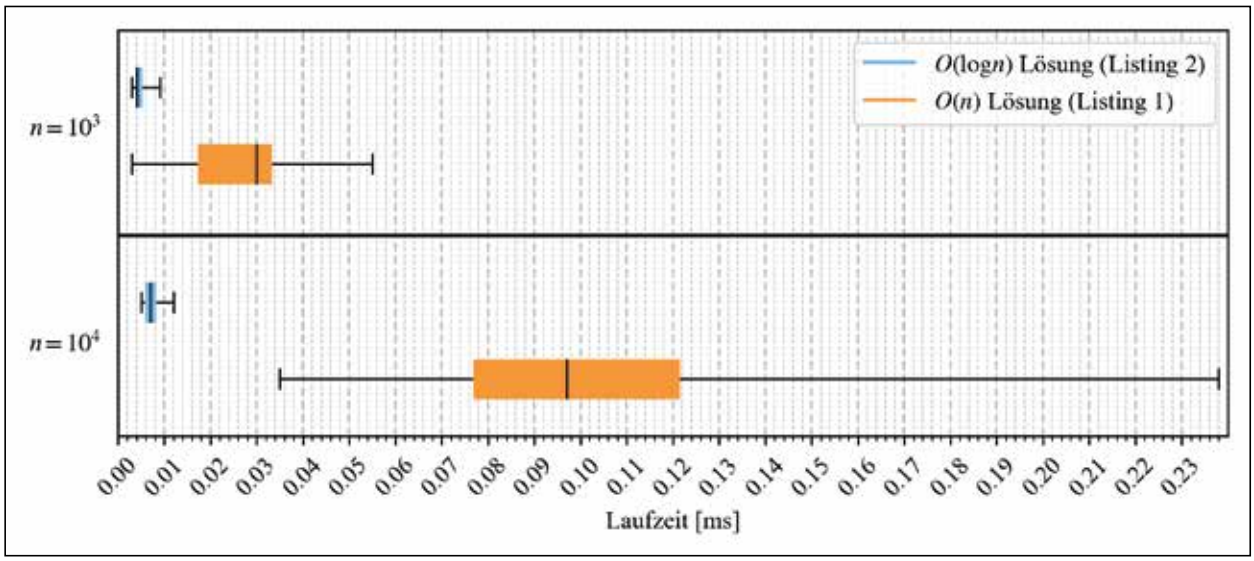

Abb. 3: Laufzeiten der jeweiligen Lösungen der Aufgabe 1

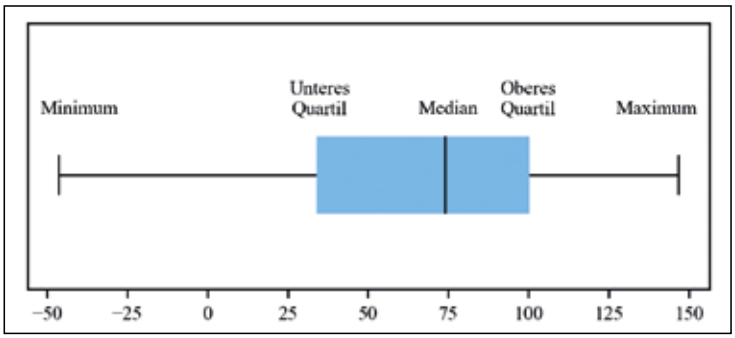

Abb. 4: Box-Plot

Aufgabe 1: Suche in einer sortierten Liste

Die erste Aufgabe ist, den Zielwert in einer bereits sortierten Liste zu finden. Eine mögliche Lösung wäre, die *contains*-Methode des *List*-Interface zu verwenden (Listing 1).

Listing 1

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int target = 19;

// solution
boolean answer = list.contains(target);
```

Listing 2

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int target = 19;

// solution
boolean answer = Collections.binarySearch(list, target) >= 0;
```

Die Zeitkomplexität dieser Lösung ist *O(n)*, weil diese im schlimmsten Fall die ganze Liste durchsucht, bis man das Ende der Liste erreicht hat. Eine andere Lösung wäre es, den Vorteil zu nutzen, dass die Eingabeliste bereits sortiert, also die binäre Suche anwendbar ist (Listing 2). *Collections.binarySearch* liefert einen Integer größer oder gleich *0*, wenn der Zielwert in der Liste enthalten ist. Die Platzkomplexität der beiden Lösungen ist *O(1)*, weil sie nur eine konstante Menge an Speicher brauchen, um das Ergebnis festzusetzen, unabhängig von den Eingabewerten.

Ich habe Testdaten aus 10^3 und 10^4 Elementen generiert und mit diesen die Laufzeit der beiden Lösungen verglichen. Die Zielwerte für die Suche sind in regelmäßigen Abständen ausgewählt, und für den jeweiligen Zielwert wurde die Laufzeit mehrmals gemessen. Die Tests wurden auf einem Windows-10-Rechner mit Intel Core i7-1065G7 CPU 1.30GHz und 32 GB RAM ausgeführt. Das verwendete JDK ist Amazon Corretto 11.0.11 und die Laufzeiten wurden mit Hilfe der Java Microbenchmark Harness [7] gemessen.

Abbildung 3 stellt die Ergebnisse für die jeweilige Länge der Eingaben als Box-Plot dar. Jeder Box-Plot enthält die Messzeiten der Aufrufe, die mit verschiedenen Zielwerten ausgeführt wurden. Der Box-Plot ist eine grafische Darstellung der Verteilung der Messergebnisse und stellt den Median, die zwei Quartile (deren Intervall die mittleren 50 Prozent der Daten enthält) und die beiden Minimum- und Maximumwerte der Daten dar (**Abb. 4**). Man kann in **Abbildung 3** erkennen, dass die Laufzeit der Lösung in Listing 1 mehr gestreut ist als die in Listing 2 mit der binären Suche. Das liegt daran, dass die Laufzeit der Listing-1-Lösung stark davon abhängt, an welcher Stelle in der Liste sich der Zielwert befindet. Diese Tendenz wird deutlicher, wenn man die Ergebnisse zwischen den Testfällen $n = 10^3$ und $n = 10^4$ vergleicht. Die Worst-Case-Laufzeit von Listing 1 ist zwischen den beiden Testfällen deutlicher gestiegen als die von Listing 2.

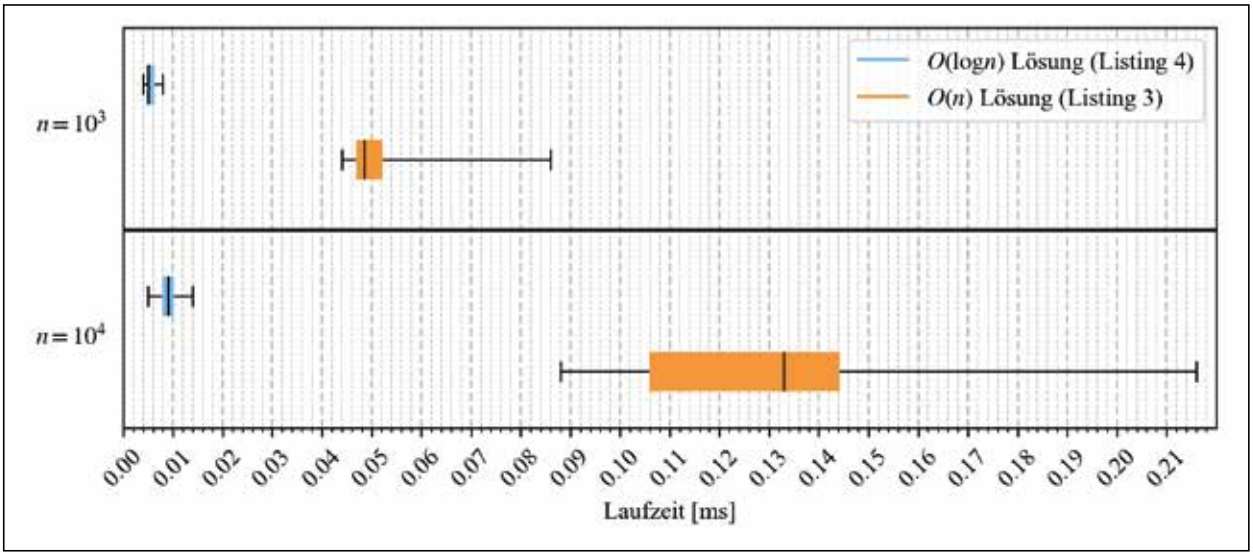

Abb. 5: Laufzeiten der jeweiligen Lösung der Aufgabe 2

Aufgabe 2: Suche eines Wertebereichs in einer sortierten Liste

Die nächste Aufgabe besteht darin, das Auftreten der Werte in einer bereits sortierten Liste zu zählen, die größer oder gleich *a* und kleiner als *b* sind, also $a \leq x_i < b$, wobei x_i der jeweilige Wert in der Eingabeliste ist. Die Voraussetzungen sind, dass die Eingabewerte *a* und *b* immer $a \leq b$ erfüllen und die Eingabeliste keine Duplikate enthält. Eine intuitive Idee wäre, mit der intermediären Operation *filter* im Stream API nur die Elemente zu sammeln, die innerhalb des angegebenen Wertebereichs liegen, und schließlich die Anzahl der Elemente mit der terminalen Operation *count* zu zählen (Listing 3).

Die Zeitkomplexität dieser Lösung ist *O(n)*, weil man einmal durch die ganze Liste iterieren muss, um für jedes Element in der Liste zu überprüfen, ob der Wert im Wertebereich liegt. Wäre es aber möglich, auch für diese Aufgabe die binäre Suche zu verwenden? Wie wäre es, wenn wir die folgenden beiden Informationen festsetzen könnten:

- Position des Wertes *a* in der Eingabeliste, wenn er enthalten ist. Ansonsten die Position in der Eingabeliste, an der man den Wert *a* einfügen kann.
- Position des Wertes *b* in der Eingabeliste, wenn er enthalten ist. Ansonsten die Position in der Eingabeliste, an der man den Wert *b* einfügen kann.

Die Differenz der beiden berechneten Positionen ist die Anzahl der Elemente, die sich zwischen den beiden Schwellenwerten befinden. In dieser Lösung wird die binäre Suche zweimal ausgeführt, aber weil man in der O-Notation den Koeffizienten nicht berücksichtigt, ist die Zeitkomplexität dieser Lösung immer noch *O(logn)*. Aus dem gleichen Grund wie in Aufgabe 1: Die Platzkomplexität der beiden Lösungen ist *O(1)*.

Listing 4 zeigt eine Beispielimplementierung dieser Lösung. Zu beachten ist, dass dieser Code nicht funktionieren wird, wenn die Eingabeliste Duplikate enthält. Wie in der Dokumentation von *Collections.binarySearch* beschrieben ist [3], garantiert die Methode nicht, dass die gesuchte Position in der Liste zurückgegeben wird, falls der Zielwert mehrmals enthalten ist.

Collections.binarySearch liefert einen Integer größer oder gleich 0 zurück, wenn der Zielwert in der Liste ent-

Listing 3

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int a = 14, b = 19;

// solution
long answer = list.stream().mapToInt(Integer::intValue)
                .filter(value -> a <= value && value < b).count();
```

Listing 4

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int a = 14, b = 19;

// solution
int lower = Collections.binarySearch(list, a);
int upper = Collections.binarySearch(list, b);
lower = lower < 0 ? ~lower : lower;
upper = upper < 0 ? ~upper : upper;
int answer = upper - lower;
```

Listing 5

```
// input
List<Integer> list = List.of(23, 18, 15, 38, 8, 24);

// solution
OptionalInt answer = list.stream().mapToInt(Integer::intValue).max();
```

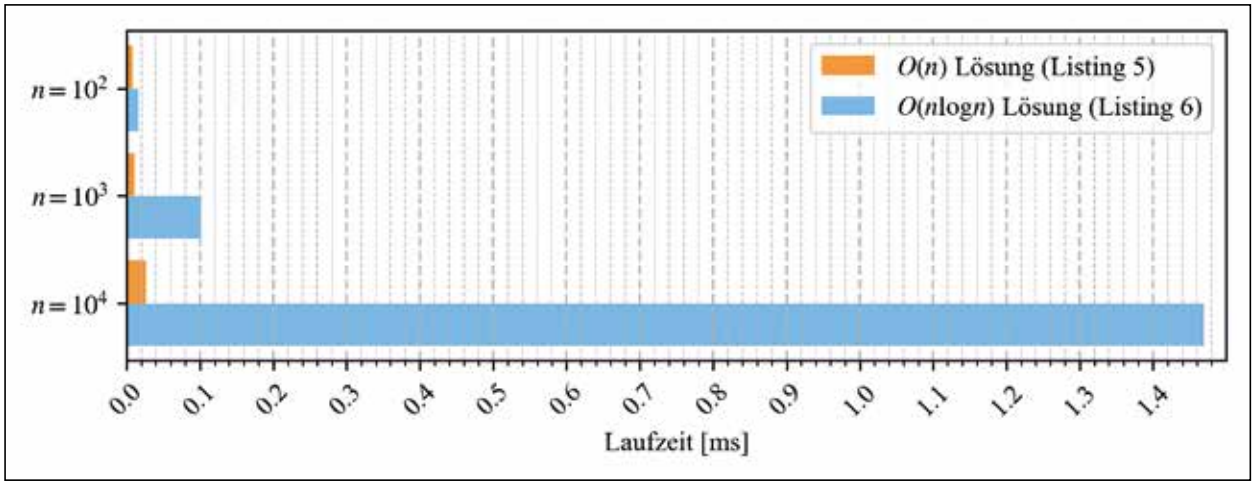

Abb. 6: Durchschnittliche Laufzeiten der jeweiligen Lösung der Aufgabe 3

halten ist. Ansonsten liefert sie einen negativen Wert zurück wobei *-(insertion point)-1* ist. Der *insertion point* ist die Position in der Liste, an der der Zielwert eingefügt werden sollte, damit die Liste weiterhin sortiert bleibt. Um den *insertion point* aus dem Rückgabewert *-(insertion point)-1* zurückzuberechnen, kann man einfach den bitweisen NICHT-Operator ~ verwenden.

Genau wie in Aufgabe 1 stellt **Abbildung 5** die Laufzeiten der beiden Lösungen als Box-Plot dar, die mit verschiedener Länge der Eingabe und der Zielwerte gemessen wurden. Auch hier ist gut zu erkennen, dass die Lösung in Listing 4 mit binärer Suche stabilere Laufzeiten hat als die in Listing 3.

Aufgabe 3: Größten Wert aus einer unsortierten Liste suchen

Die Aufgabe ist nun, den größten Wert einer unsortierten Liste zu finden, die aus Integern besteht. Eine mögliche Lösung mit Hilfe des Stream API wäre die Verwendung von *IntStream* und seiner terminalen Operation *max* [8] (Listing 5).

Listing 6

```
// input
List<Integer> list = Arrays.asList(23, 18, 15, 38, 8, 24);

// solution
list.sort(Collections.reverseOrder());
int answer = list.get(0);
```

Listing 7

```
// input
List<Integer> list = Arrays.asList(23, 18, 15, 38, 8, 24);
int k = 3;

// solution
list.sort(Collections.reverseOrder());
List<Integer> answer = list.subList(0, k);
```

Diese Lösung hat die Zeitkomplexität *O(n)* und die Platzkomplexität *O(1)*. Eine andere Idee wäre, die Liste einmal absteigend zu sortieren und den ersten Wert in der Liste zurückzugeben (Listing 6). Wie bereits erwähnt bietet Java mehrere Wege, um eine Liste zu sortieren. Um absteigend zu sortieren, muss man in Java einen Komparator angeben, der rückwärts vergleicht, weil die Liste standardmäßig aufsteigend sortiert wird. Auch zu beachten ist, dass man keine unveränderbare Liste verwenden darf, ausgenommen dann, wenn man mit der intermediären Operation *sorted* im Stream API arbeitet, weil die *sort*-Methoden die Liste direkt verarbeiten. Unveränderbare Listen sind zum Beispiel Listen, die mit der *List.of*-Methode erstellt wurden.

Diese Lösung hat die Zeitkomplexität *O(nlogn)*. Allerdings ist die Platzkomplexität dieser Lösung abhängig vom Sortierverfahren, das in der Implementierung der *sort*-Methode verwendet wurde. Wie bereits in **Abbildung 1** zu erkennen war, ist die Zeitkomplexität *O(nlogn)* schlechter als *O(n)*. Tatsächlich ist in **Abbildung 6** zu sehen, dass, wenn sich die Länge der Eingabeliste *n* vergrößert, die Laufzeit der Lösung aus Listing 6 mit der Sortierung drastischer steigt als bei Listing 5. In der nächsten Aufgabe werden wir allerdings sehen, dass es in bestimmten Fällen doch eine gute Idee ist, die Liste zu sortieren.

Aufgabe 4: Die größten k-Elemente aus einer unsortierten Liste suchen

In der letzten Aufgabe haben wir festgestellt, dass eine Sortierung nicht nötig ist, wenn man nur den größten Wert einer unsortierten Liste wissen möchte. Wie wäre es dann, wenn man die *k* größten Werte aus der Liste braucht? Das heißt, wenn *k* = 3, dann muss man die drei größten Werte aus der Liste herausfinden (vorausgesetzt, *k* ist kleiner als die Länge der Eingabe). In diesem Fall reicht es nicht mehr aus, einmal durch die Eingabeliste zu iterieren, aber die Lösung mit der Sortierung wird weiter funktionieren (Listing 7).

Diese Lösung kann man mit einer Priority Queue (Vorrangwarteschlange) noch leicht optimieren. Die Priority Queue, die in Java mit binärem Heap implementiert ist [9], ist eine abstrakte Datenstruktur, mit

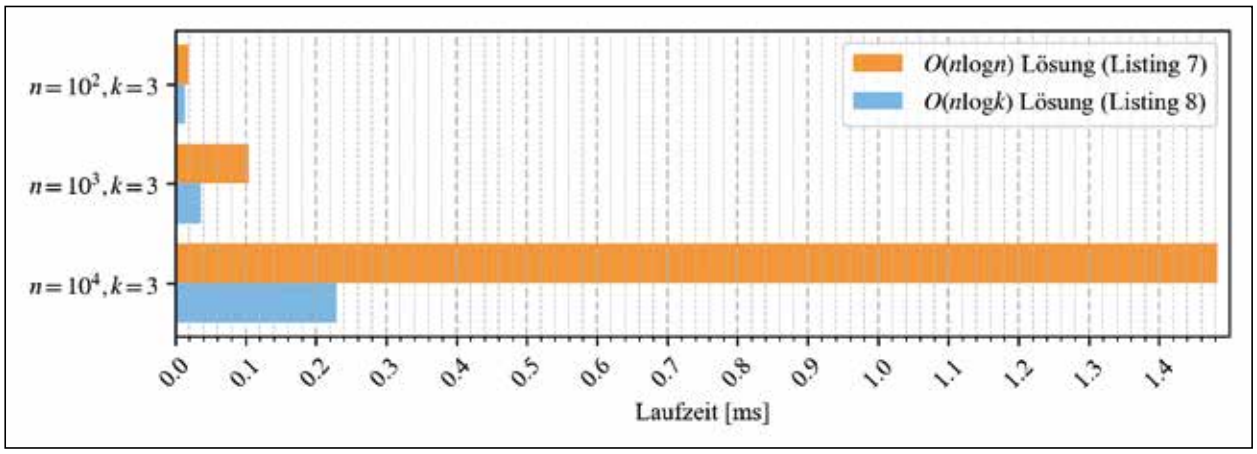

Abb. 7: Durchschnittliche Laufzeiten der jeweiligen Lösung der Aufgabe 4

der man den kleinsten Wert (oder den größten Wert, je nachdem, welcher Komparator angegeben ist) in der Queue abfragen kann. Generell ist die Zeitkomplexität für das Hinzufügen und Löschen der Werte *O(logn)*, für die Abfrage des kleinsten Werts ist sie *O(1)*, während *n* die Länge der Priority Queue ist.

In unserem Fall fügt man die einzelnen Elemente aus der Eingabeliste in die Priority Queue und löscht jeweils den kleinsten Wert, sobald die Länge der Priority Queue größer als *k* ist. Schließlich fügt man die einzelnen Elemente aus der Priority Queue in eine Liste ein. Listing 8 zeigt eine Beispielimplementierung dieser Lösung. Als kleine Optimierung ist die Priority Queue mit einer initialen Kapazität *k+1* instanziiert, weil sie höchstens *k+1* Elemente beinhalten kann. Die Zeitkomplexität dieser Lösung ist *O(nlogk)*, weil man *n* Elemente aus der Eingabeliste jeweils in die Priority Queue einfügt, aber die Länge der Priority Queue auf *k* begrenzt ist. Die Platzkomplexität ist *O(k)*, weil man temporär *k* Elemente in der Priority Queue behält, um schließlich die Rückgabeliste zu erstellen.

Abbildung 7 stellt die durchschnittlichen Laufzeiten der jeweiligen Lösungen dar, die mit verschiedenen Längen der Eingabeliste *n* gemessen wurden. Je größer die Differenz zwischen *n* und *k*, umso größer wirkt sie sich auf die Laufzeit aus.

Listing 8

```
// input
List<Integer> list = List.of(23, 18, 15, 38, 8, 24);
int k = 3;

// solution
Queue<Integer> queue = new PriorityQueue<>(k+1);
for(int v : list) {
  queue.offer(v);
  if(queue.size() > k) {
    queue.poll();
  }
}
List<Integer> answer = Stream.generate(queue::poll)
  .takeWhile(Objects::nonNull).collect(Collectors.toList());
```

Fazit

In diesem Artikel habe ich die Ideen der Zeit- und Platzkomplexitäten zusammengefasst und insbesondere verglichen, wie sich die Zeitkomplexität auf die Laufzeit auswirkt, wenn man mit einer größeren Menge an Daten arbeitet. Es ist eine gute Praxis, die beiden Maße im Hinterkopf zu behalten und bei Trade-offs andere Kriterien wie die Lesbarkeit oder Wartbarkeit des Codes zu berücksichtigen. Bei kleineren Datenmengen ist das Stream API ein sehr mächtiges Werkzeug. Allerdings ist die Zeitkomplexität grundsätzlich *O(n)*, wenn man über die gesamte Eingabe filtert oder sucht und keinen vorzeitigen Abbruch vollzieht. Wenn die Möglichkeit besteht, dass künftig die Größe der Eingabe wächst, sollte man von Anfang an überlegen, ob es keine bessere Lösung aus Sicht der beiden Komplexitäten gibt.

Ikuru Otomo schloss den Master of Information Science and Technology an der Hokkaido University ab und arbeitet zurzeit als Professional Software Developer bei der sidion GmbH.

Links & Literatur

[1] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/stream/Stream.html

[2] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Arrays.html

[3] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collections.html

[4] https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/quick/quick.htm

[5] https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/merge/merge.htm

[6] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/List.html

[7] https://github.com/openjdk/jmh

[8] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/stream/IntStream.html

[9] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/PriorityQueue.html