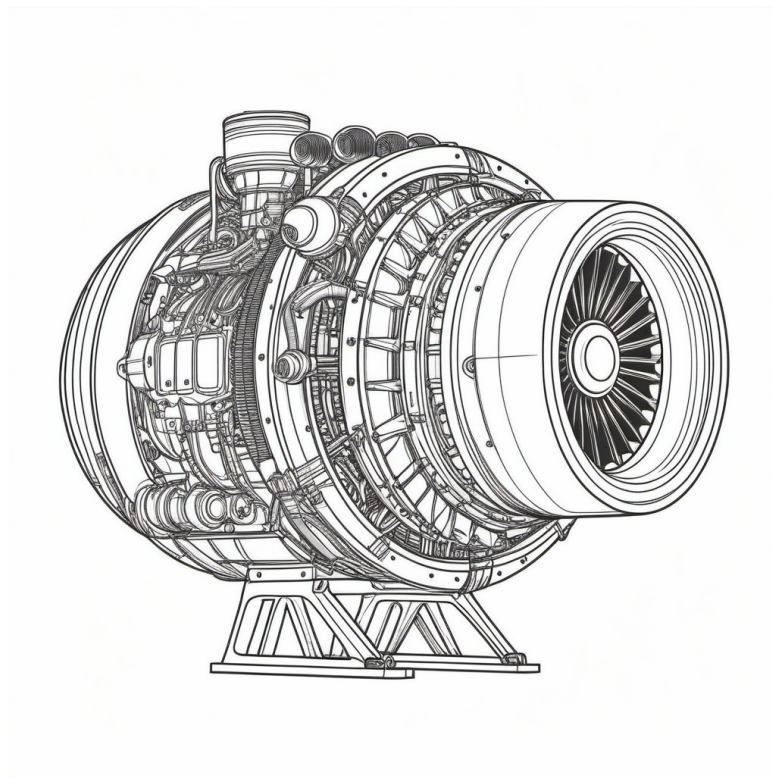


PROJEKTBERICHT

Sortieralgorithmen

*Datenbank-Sortierung des prozentualen Turbinenschubs von
Flugzeugen auf verschiedenen Flugrouten*



Name: Fabian Bauriedl
Matrikelnummer: 9085643

*Studiengang: Data Science und Künstliche Intelligenz
Kurs: WDS123
Fach: Fortgeschrittenes Programmieren
Dozent: Steffen Merk
Datum: 27.05.2024*

Inhaltsverzeichnis

1.	Einleitung.....	3
2.	Vorstellung und Auswahl der Sortieralgorithmen	4
2.1	Selection Sort.....	4
2.2	<i>Bubble Sort</i>	4
2.3	<i>Insertion Sort</i>	5
2.4	<i>Quick Sort</i>	5
2.5	<i>Merge Sort</i>	6
2.6	<i>Auswahl des passenden Algorithmus</i>	7
3.	Implementierung des Merge-Sort Algorithmus.....	8
3.1	<i>Row.java</i>	8
3.2	<i>AggregatedRow.java</i>	8
3.3	<i>Aggregator.java</i>	8
3.4	<i>Sorter.java</i>	9
4.	Schlussworte.....	10
5.	Eigenständigkeitserklärung	11

1. Einleitung

Die Fluggesellschaft 'Horizon Airlines' hat eine Software entwickelt, mit der unter Berücksichtigung von Wetter- und Umgebungsfaktoren, sowie Leistungsdaten der Flugzeuge, Treibstoff gespart werden kann.

Das jeweilige Flugzeug kann den Triebwerkschub in Echtzeit auf die Umgebungsbedingungen anpassen. Die Airline hat verschiedene Flugzeuge in ihrer Flotte, die je nach Flugroute für den Flug ausgewählt werden. Je nach überflogenem Ort auf einer Flugroute muss mit einer gewissen vorgegebenen Geschwindigkeit geflogen werden - bei Start und Landung fliegen die Flugzeuge jeweils tiefer und langsamer, bei starken Winden wird der Schub deutlich erhöht, um die Reisegeschwindigkeit zu halten.

Horizon Airlines möchte die Software nun weiterentwickeln und aus den gesammelten Daten realistische Berechnungen des benötigten Treibstoffs für die nächsten Flüge durchführen. Hierfür werden die Daten nach Flugrouten gruppiert und anhand der durchschnittlichen Schubkraft der Triebwerke auf jeder Route können Treibstoffmengen für zukünftige Flüge errechnet werden.

Die Herausforderungen für die Softwareentwickler umfassen unter anderem die Zusammenfassung aller gesammelten Messpunkte in gruppierte Flugrouten mit den jeweiligen durchschnittlichen Messwerten. Diese gruppierten Routen sollen nach durchschnittlichem Triebwerksschub aufsteigend sortiert werden können, was der Airline zukünftig mehr Möglichkeiten zur internen Auswertung von Flugdaten eröffnet. Zusätzlich enthält der Flugdatensatz zu Projektbeginn über eine Milliarde Datenpunkte. Um eine zuverlässige und effiziente Nutzung der Software zu gewährleisten, müssen die Daten möglichst schnell sortiert werden. Die Sortierung soll die volle Leistungsfähigkeit der Datenbankserver von Horizon Airlines ausnutzen und parallel ausgeführt werden können.

Im Folgenden werden mögliche Sortialgorithmen für die Datenbanksortierung verglichen. Die Auswahl eines optimalen Algorithmus für diese großen Datenbank wird begründet und implementiert.

2. Vorstellung und Auswahl der Sortialgorithmen

Ein Sortialgorithmus dient dazu, Elemente einer Liste oder Datenbank in eine bestimmte Reihenfolge zu bringen. Dies kann nach numerischen Werten, alphabetischer Reihenfolge oder anderen Vergleichskriterien erfolgen.

Sortialgorithmen sind aufgrund der effizienten Organisation von Daten entscheidend für die gute Performance einer Software. Sie variieren in ihrer Effizienz und Implementierung bei unterschiedlich großen Datenmengen, die Wahl des passenden Algorithmus hängt somit von den spezifischen Anforderungen an die Software und der Datenmenge ab.

Auf den folgenden Seiten werden fünf wichtige Sortialgorithmen näher vorgestellt.

2.1 Selection Sort

Der Selection Sort wählt das kleinste Element aus einer unsortierten Liste und verschiebt es an den Anfang der Liste (Index 0). Der Vorgang wird nun für die ganze Liste wiederholt und das jeweils nächstgrößere Element wird an das vorherige (kleinere) angehängt, sodass die Liste aufsteigend sortiert wird.

Die Implementierung des Selection Sort ist sehr einfach und die Speicherkomplexität bleibt für unterschiedlich große Datenmengen gleich konstant bei $O(1)$, da immer nur ein Element zwischengespeichert werden muss – das nächstgrößere aus dem unsortierten Teil der Liste. Für die Zeitkomplexität sieht dies allerdings anders aus. Da die Position der nächstgrößeren Elemente nicht bekannt ist, muss die Liste bei jeder Wiederholung bis zum Ende der Liste durchsucht werden. Erst beim letzten Element ist sichergestellt, dass das nächstkleinere gefunden wurde. Die Zeitkomplexität liegt somit bei $O(N^2)$, für große Datenmengen ist die Performance nicht optimal.

Des Weiteren sortiert der Selection Sort nicht stabil. Dies bedeutet, dass die Anordnung von Elementen mit gleichem Sortierkriterium nach der Sortierung verändert sein kann.

Der Selection Sort eignet sich gut für kleine Datenmengen, sofern die Stabilität der Sortierung nicht relevant ist.

2.2 Bubble Sort

Der Bubble Sort Algorithmus ist ebenfalls einfach zu implementieren.

Er geht die Liste Element für Element durch. Ist das aktuelle Element größer als das jeweils nächste Element, werden sie miteinander vertauscht. Am Ende eines Durchlaufs ist das größte Element der Liste am letzten Index.

Anschließend wird die Liste von vorne durchgegangen, bis das zweitgrößte Element in der Liste vor das Größte gesetzt wird. Dieser Vorgang wird für die ganze Liste wiederholt, bis die Liste von hinten, der Größe nach absteigend, sortiert ist.

Der Bubble Sort hat wie der Selection Sort eine Speicherkomplexität von $O(1)$, da nur jeweils das nächste Element der Liste zwischengespeichert wird.

Für jedes Element muss die Liste einmal durchgegangen werden, um den jeweils nächstkleinere Element korrekt zu platzieren. Somit hat der Bubble Sort eine Zeitkomplexität von $O(N^2)$ und ist ebenfalls nicht für große Datenmengen geeignet.

Allerdings ist er im Gegensatz zum Selection Sort stabil, da gleichgroße Elemente nicht miteinander vertauscht werden und die ursprüngliche Reihenfolge dieser so nicht verändert wird.

2.3 Insertion Sort

Auch der Insertion Sort teilt die Liste in einen sortierten Teil und einen unsortierten. Zu Beginn gilt nur das erste Element als sortiert. Nun wird das zweite Element ausgewählt und links oder rechts dem ersten Element platziert. Dieser Vorgang wird nun für jedes Element der unsortierten Liste durchgeführt, wobei das jeweilige Element immer an die richtige Stelle im sortierten Teil eingesetzt wird. Bei jedem Einsetzen eines Elements erweitert sich der sortierte Teil der Liste um dieses Element.

Der Insertion Sort kann etwas performanter als der Selection- oder Bubble Sort sein. Da der sortierte Teil immer bereits bekannt ist, muss nicht für jede Wiederholung die ganze Liste durchsucht werden, sondern nur der sortierte Teil, bis zu der Position, an der das neue Element aus dem unsortierten Teil eingefügt wird.

Je sortierter die Daten bereits vor der Ausführung des Algorithmus sind, desto schneller sortiert der Insertion Sort.

Für den Worst-Case beträgt die Zeitkomplexität jedoch ebenfalls $O(N^2)$.

Die Speicherkomplexität liegt auch hier bei $O(1)$, da nur ein neues Element zwischengespeichert wird. Der Algorithmus ist stabil, da durch das Einsetzen die ursprüngliche relative Reihenfolge nicht verändert wird.

Somit eignet er sich gut für kleine Datenmengen, er kann aber auch für größere, bereits leicht vorsortierte Daten verwendet werden.

2.4 Quick Sort

Beim Quick Sort wird zunächst ein Pivotelement in der Liste gewählt, welches als Startpunkt für die Sortierung fungiert. Die Liste wird zum Start in zwei Teilbereiche unterteilt:

In Elemente, die kleiner oder gleich dem Pivotelement sind und in Elemente, die größer als das Pivotelement sind. Hierfür wird jedes Element in der Liste mit dem Pivotelement verglichen und in den entsprechenden Teilbereich einsortiert.

Dieser Prozess wird Partitionierung genannt.

Der Algorithmus wird anschließend immer wieder rekursiv jeweils zunächst auf den linken, anschließend auf den rechten Teilbereich angewendet.

Die Teilbereiche werden immer kleiner und mit jedem neuen Teilbereich wird ein neues Pivotelement gewählt, an dem geteilt wird.

Auf diese Weise wird die Liste schrittweise weiter sortiert. Die vollständige Sortierung ist allerdings erst beendet, sobald die letzten Teilbereiche nur noch aus einem Element bestehen.

Der Quick Sort Algorithmus kann bei guten Voraussetzungen der Daten sehr schnell sein. Bei ungünstigen Voraussetzungen ist der Quick Sort jedoch auch stark in der Performance beeinträchtigt, bei einer Zeitkomplexität von $O(N^2)$. Hat die Liste bestimmte Muster oder ist die ungefähre Sortierung der Liste bereits bekannt, kann die richtige Wahl des Pivotelements einen erheblichen Einfluss auf die Sortiergeschwindigkeit haben. Quick Sort kann in-place mit einer Speicherkomplexität von $O(1)$ durchgeführt werden. Die Sortierung ist jedoch nicht stabil, da es vorkommen kann, dass durch die Partitionierung Elemente mit gleichen Sortierkriterien die ursprüngliche Reihenfolge verlieren.

Allerdings kann die Ausführung parallel erfolgen. Die Ausführungsgeschwindigkeit wird dabei gesteigert, indem die Sortierungen der jeweils linken und rechten Listenhälften gleichzeitig ausgeführt werden.

2.5 Merge Sort

Der letzte hier vorgestellte Sortieralgorithmus ist der Merge Sort. Wie der Name schon verrät, wird die Sortierung hier durch Aufteilen und anschließendes Zusammenfügen einer Liste erzielt.

Der Merge Sort funktioniert wie der Quick Sort rekursiv. Zunächst wird die Liste mit jeder Rekursionsschleife in der Mitte zerteilt (in einen linken und einen rechten Bereich), bis nur noch einzelne Elemente übrig sind.

Im zweiten Schritt werden die einzelnen Elemente jeweils mit ihrem Nachbarelement verglichen und wieder in einer neuen, sortierten Menge zusammengefügt.

Diese Mengen werden im Anschluss wieder mit ihren Nachbarmengen zu einer größeren Menge zusammengefügt. Das jeweils erste Element in jeder Menge wird mit dem ersten Element der Nachbarmenge verglichen und das kleinere in die neue Menge eingesetzt, anschließend vergleicht man die nächsten Elemente der beiden Nachbarmengen mit den restlichen, noch nicht eingesetzten und setzt wieder das jeweils kleinere in die neue Menge ein.

Diese Zusammenführung wird so lange durchgeführt, bis man in der letzten Rekursionsschleife die vollständig sortierte Liste erhält.

Das Besondere am Merge-Sort Algorithmus ist eine garantierte und konstante Laufzeit von $O(N \log N)$, jeweils im Best-, Average- und Worst Case. Der logarithmische Teil $O(\log N)$ der Zeitkomplexität resultiert aus der Aufteilung in Teilmengen der Liste, das Zusammenführen dieser Teilbereiche hat die Zeitkomplexität $O(N)$, die proportional zur Größe der jeweiligen Bereiche ist. Die Laufzeit des Merge Sort ist dadurch sogar berechenbar.

Er kann parallel ausgeführt werden, wobei jeweils die linke und rechte Seite der Liste gleichzeitig aufgeteilt und sortiert werden. Die Geschwindigkeit der Sortierung wird auf diese Weise weiter gesteigert.

Durch das Aufteilen in temporäre Teillisten besteht allerdings eine Speicherkomplexität von $O(N)$ – deutlich größer als bei den bisher vorgestellten Algorithmen. Dafür ist der Merge-Sort ein stabiler Algorithmus, die ursprüngliche Reihenfolge der Elemente mit gleichem Sortierkriterium bleibt bestehen.

In den meisten Fällen spielt der Speicherplatz jedoch kaum eine Rolle, da meistens mehr als genug Hardware-Ressourcen verfügbar sind.

Der Merge-Sort Algorithmus ist durch die konstante Zeitkomplexität von $O(N \log N)$ und mit einer garantierten Stabilität ideal für große Datenmengen.

2.6 Auswahl des passenden Algorithmus

Die Fluggesellschaft Horizon Airlines muss entscheiden, welchen Sortieralgorithmus sie in ihrer Software verwendet, um die mehr als eine Milliarde Datenpunkte ihrer Flugdaten nach einzelnen Flugrouten zu sortieren. Die Server, auf denen die Daten der Airline liegen, sind sehr leistungsfähig, weshalb der Speicherbedarf nicht relevant für die Auswahl des Algorithmus ist. Die Geschwindigkeit, mit der die Sortierung in einzelne Flugrouten durchgeführt wird, allerdings umso mehr. Die Sortierung muss für eine zuverlässige Funktion der Software so effizient und zuverlässig wie möglich durchgeführt werden. Der Algorithmus sollte parallel ausführbar sein, um die Leistungsfähigkeit der Server voll auszunutzen. Für die Auswahl des optimalen Algorithmus, werden im Folgenden alle vorgestellten Algorithmen miteinander verglichen.

	ZEITKOMPLEXITÄT (im Worst Case)	SPEICHERKOMPLEXITÄT	STABILITÄT	PARALLELE AUSFÜHRUNG
Selection Sort	$O(N^2)$	in-place: $O(1)$	nicht stabil	nicht möglich
Bubble Sort	$O(N^2)$	in-place: $O(1)$	stabil	nicht möglich
Insertion Sort	$O(N^2)$	in-place: $O(1)$	stabil	nicht möglich
Quick Sort	$O(N^2)$	in-place: $O(1)$	nicht stabil	möglich
Merge Sort	$O(N \log N)$	out-of-place: $O(N)$	stabil	möglich

Tabelle: Übersicht über die Eigenschaften der Algorithmen

Da der Selection-, Bubble- und Insertion-Sort mit steigender Datenmenge deutlich mehr Zeit in Anspruch nehmen und keiner der drei parallel ausgeführt werden kann, fallen diese für die Airline direkt aus der Auswahl.

Der Quick-Sort Algorithmus kann zwar parallel ausgeführt werden und in der Theorie effizient mit großen Datenmengen umgehen, die Voraussetzungen hierfür sind aber bereits grob vorsortierte Daten. Da der Airline-Datenbank im Alltag des Flugbetriebs ständig neue Daten hinzugefügt werden, ist mit dem Quick Sort eine zuverlässig schnelle Sortierung in der Software nicht garantiert – auch wenn er parallel und dadurch effizienter ausgeführt werden kann.

Die sinnvollste Lösung für die Airline-Software ist der Merge-Sort Algorithmus.

Die Laufzeit, die er für die Sortierung benötigt, ist proportional zur Größe der zu sortierenden Datenmenge. Durch das deterministische Verhalten beim immer gleichen Ausführen von Teilungs- und Vereinigungs-Schritten ist die Laufzeit der Sortierung nicht nur sehr effizient, sie ist zusätzlich auch berechenbar, wobei die ursprüngliche Reihenfolge der Sortierung beibehalten wird (Stabilität).

Mit ihren leistungsfähigen Servern kann Horizon Airlines den Merge Sort sogar parallel ausführen lassen. So entsteht eine unschlagbare Kombination aus Effizienz und Zuverlässigkeit beim Sortieren von großen Datenmengen.

Die Implementierung innerhalb der Software wird im nächsten Kapitel gezeigt.

3. Implementierung des Merge-Sort Algorithmus

3.1 Row.java

```
public record Row(String routeName, double thrustInPercent) {  
}
```

3.2 AggregatedRow.java

```
public record AggregatedRow (String aggregatedRouteName,  
                             double averageThrustInPercent) {  
}
```

3.3 Aggregator.java

```
public class Aggregator {  
  
    public ArrayList<AggregatedRow> aggregate(ArrayList<Row> allRows) {  
  
        HashMap<String, AggregatedRow> airplaneRoutes = new HashMap<>();  
        for (Row r : allRows) {  
            if (!airplaneRoutes.containsKey(r.routeName())) {  
                double thrustSum = 0;  
                int counter = 0;  
                for (int i = 0; i < allRows.size(); i++) {  
                    if (r.routeName().equals(allRows.get(i).routeName())) {  
                        thrustSum += allRows.get(i).thrustInPercent();  
                        counter++;  
                    }  
                }  
                double averageThrust = (counter != 0) ? thrustSum/counter:0.0;  
                airplaneRoutes.put(r.routeName(),  
                                   new AggregatedRow(r.routeName(),  
                                                       averageThrust));  
            }  
        }  
        return new ArrayList<>(airplaneRoutes.values());  
    }  
}
```


3.4 Sorter.java

```
public class Sorter {

    private static void mergeSort(ArrayList<AggregatedRow> aggregatedRows) {
        int length = aggregatedRows.size();
        if (length <= 1) return;

        int middleIndex = length / 2;
        ArrayList<AggregatedRow> leftSide =
            new ArrayList<>(aggregatedRows.subList(0, middleIndex));
        ArrayList<AggregatedRow> rightSide =
            new ArrayList<>(aggregatedRows.subList(middleIndex,
                                                    length));

        mergeSort(leftSide);
        mergeSort(rightSide);
        merge(aggregatedRows, leftSide, rightSide);
    }

    private static void merge(ArrayList<AggregatedRow> aggregatedRows,
                              ArrayList<AggregatedRow> leftSide,
                              ArrayList<AggregatedRow> rightSide) {

        int leftIndex = 0;
        int rightIndex = 0;
        for (int i = 0; i < aggregatedRows.size(); i++) {
            if (leftIndex >= leftSide.size()) {
                aggregatedRows.set(i, rightSide.get(rightIndex++));
            } else if (rightIndex >= rightSide.size()) {
                aggregatedRows.set(i, leftSide.get(leftIndex++));
            } else if (leftSide.get(leftIndex).averageThrustInPercent()
                <= rightSide.get(rightIndex).averageThrustInPercent()) {
                aggregatedRows.set(i, leftSide.get(leftIndex++));
            } else if (leftSide.get(leftIndex).averageThrustInPercent()
                > rightSide.get(rightIndex).averageThrustInPercent()) {
                aggregatedRows.set(i, rightSide.get(rightIndex++));
            }
        }
    }

    public void sort(ArrayList<AggregatedRow> aggregatedRows) {
        mergeSort(aggregatedRows);
    }
}
```

4. Schlussworte

Nach der detaillierten Betrachtung der fünf vorgestellten Sortieralgorithmen (Selection Sort, Bubble Sort, Insertion Sort, Quick Sort und Merge Sort) zeigt sich, dass der Merge Sort die optimale Wahl für die Anforderungen von Horizon Airlines darstellt.

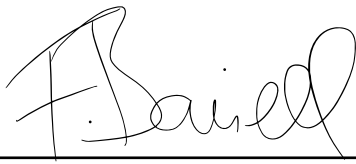
Selection-, Bubble- und Insertion Sort sind einfache Algorithmen, die bei kleinen Datenmengen schnell sein können - bei größeren jedoch sehr ineffizient werden. Für Datensätze wie die Flugdaten von Horizon Airlines, mit einer enormen Größe von über einer Milliarde Datenpunkte, sind diese Algorithmen nicht geeignet. Sie brauchen zu viel Zeit für die Sortierung. Der Quick Sort kann schon deutlich performanter sein, jedoch ist seine Laufzeit nicht berechenbar und kann im Worst-Case auch zu hohen Performance-Einbußen führen. Der Merge Sort hingegen überzeugt durch seine Stabilität und eine konstant berechenbare Laufzeit. Trotz eines höheren Speicherbedarfs im Vergleich zu in-place Algorithmen, bietet Merge Sort durch seine hohe Effizienz und Zuverlässigkeit klare Vorteile bei der Sortierung großer Datenmengen.

Durch die Implementierung von Merge Sort in der Software von Horizon Airlines können die gesammelten Messpunkte effizient gruppiert und sortiert werden. Dies ermöglicht die gewünschte präzise und schnelle Berechnung des benötigten Treibstoffs für zukünftige Flüge. Der Merge Sort Algorithmus erfüllt alle an die Software gestellten Anforderungen hinsichtlich Zuverlässigkeit bei großen Datenmengen und paralleler Ausführung des Algorithmus für eine effiziente Sortierung. So leistet die Software einen wichtigen Beitrag zur Effizienzsteigerung und Nachhaltigkeit bei Horizon Airlines.

5. Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Ravensburg, 27.05.1997

A handwritten signature in black ink, appearing to read 'F. Baill', written over a horizontal line.

(Datum, Unterschrift)