PROJEKTAUFGABE AE

Remove Duplicates - Spotify playlist cleaner

RAPHAEL DRECHSLER

INHALTSVERZEICHNIS

1 PROBLEMSTELLUNG "REMOVE DUPLICATES"

Problem
Gegeben ist eine Sequenz. Diese Sequenz enthält ggf. Duplikate.
Ziel des umzusetzenden Algorithmus ist das Entfernen der Duplikate aus dieser Sequenz.

Für die Umsetzung eines entsprechenden Algorithmus sollen die zwei folgenden Ansätze betrachtet werden:

- Naiver Ansatz: Nutzung eines einfachen Arrays
- Ansatz im Fokus: Nutzung einer Hash-Table

Naiver Ansatz
In der Implementierung nach dem naiven Ansatz würden alle Daten einer Sequenz in einem Array gespeichert werden, insofern sie nicht bereits im Array enthalten sind. Beim betrachten eines Elementes aus der Sequenz muss also in der naiven Umsetzung das komplette Array nach einem identischen Element durchsucht werden. Im in Hinblick auf die Laufzeit schlimmsten Fall, muss daher jedes Element im Array mit dem aktuell untersuchten Element des Sequenz verglichen werden. Bei einer Sequenz-Liste der Größe n müssen also im wors-case

$$\sum_{k=1}^{n-1} k \tag{1}$$

Vergleichsoperationen durchgeführt werden. Betrachtet man die folgende Umformung entsprechend der Gaußschen Summenformel

$$\sum_{k=1}^{n-1} k = \frac{(n-1)^2 + (n-1)}{2} = \frac{1}{2}(n-1)^2 \tag{2}$$

ergibt sich für eine Umsetzung dieses Ansatzes eine theoretische obere Komplexitätsgrenze von

$$O(n^2) \tag{3}$$

Ansatz Hash-Table
Der Ansatz "Hash-Table" setzt an dem Punkt der Komplexitätsbetrachtung an Würde man für die Einordnung der Elemente in das Array eine Direktadressierung verwenden, so würde man ein Element selbst als Schlüssel interpretieren, mit dem ein Feld im Array adressiert wird.

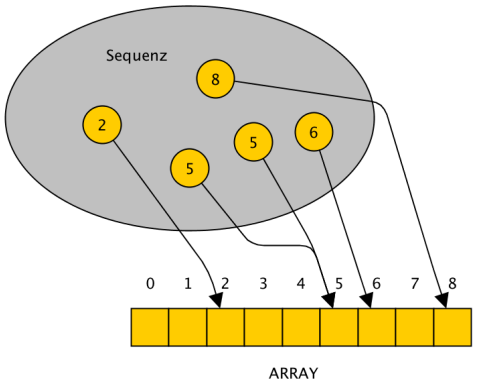

Abbildung 1: Skizze: Prinzip der direkten Adressierung *nach* [1]

Der Zeitaufwand für die Prüfung, ob ein Element bereits im Array gespeichert ist, wäre dabei $O(1)$.
Ist das Universum $U = 0, 1, ..., m$, in denen sich die Schlüssel der Elemente befinden klein, lässt sich somit schnell auf ein Array der Größe $A[0..m-1]$ zugreifen. Ab einer bestimmten Größe des Universums kann eine Umsetzung der Direkt-Adressierung Aufgrund der erforderlichen Größe des Ziel-Arrays nicht mehr sinnvoll bzw. möglich sein.
Die Hash-Table löst dieses Problem, indem sie das große Universum U einem kleineren Array $A[0..m-1]$ gegenüberstellt. Die Adressierung der Array-Felder pro Element erfolgt weiterhin auf Grundlage des Element-Wertes. Um nur existierende Schlüssel zu erhalten wird zur Ermittlung des Schlüssels eine Funktion h (sogenannte Hash-Funktion) eingesetzt, welche die Werte der Elemente im Universum U auf existierende Schlüssel abbildet. *vgl.*[1]

$$h : U -> 0, ..., m-1 \tag{4}$$

Die Skizze gestaltet sich wie folgt:

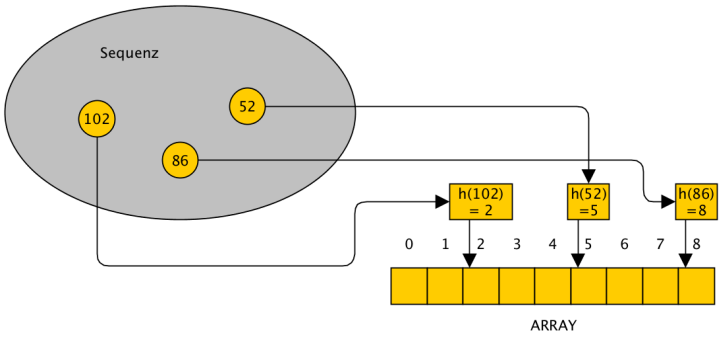

Abbildung 2: Skizze: Prinzip der Adressierung in einer Hash-Table*nach* [1]

Dadurch, dass gilt $|U| > m$ wird über Anwendung des Schubfachprinzips ersichtlich, dass die Hash-Funktion h nicht injektiv ist.
Fälle in denen gilt

$$h(a) = h(b) | a, b \in U, a \neq b \tag{5}$$

werden als Kollision bezeichnet. *vgl.*[1]
Bei der Umsetzung des Algorithmus muss also eine entsprechende Strategie zur Auflösung solcher Kollisionen mit betrachtet werden.

Kollisions-Auflösung im python dict-Objekt
Als Kollisions-Auflösungs-Strategie soll im Rahmen der Umsetzung das Verfahren implementiert werden, welches im python-Standard beim Zugreifen auf *dict*-Objekte genutzt wird. Durch die ggf. notwendige Behandlung von Kollisionen ergibt sich für einen Zugriff auf ein Feld des Arrays bei einer entsprechenden Implementierung der folgende Zeitbedarf.

Operation	average-case	worst-case
Element einfügen	$O(1)$	$O(n)$
Auf Element zugreifen	$O(1)$	$O(n)$

Tabelle 1: Zugriffs-Komplexität im python dict [2]

2 ANWENDUNGSSZENARIO SPOTIFY PLAYLISTS

Als Anwendungsfall soll das disjunkte Vereinigen von Titeln mehrerer Spotify-Playlists betrachtet werden.
In Spotify lassen sich für eine geöffnete Playlist per Tastatur-Kurzbefehl "Strg + A" und "Strg + C" alle Titel der Playlist in die Zwischenablage kopieren. Die Titel werden dabei als URI repräsentiert.

```
https://open.spotify.com/track/757530vPBymdi31CtXstxP
https://open.spotify.com/track/1Qi256uJuMihknGuuFcQoC
https://open.spotify.com/track/0JFBf2PloRfMkPg5DjXhDx
https://open.spotify.com/track/7iXF2W9vKmDoGAhlHdpyIa
```

Abbildung 3: Beispiel: Spotify-Titel URIs

Die URIs werden durch den Anwender in einem separatem *.txt*-File gesammelt. Somit werden mehrere Playlists in diesem *.txt*-File vereinigt.
Die Bereinigung der Duplikate soll nun der zu implementierende Algorithmus übernehmen.
Nach Abschluss der Bereinigung lassen sich alle URIs des *.txt*-Files in die Zwischenablage kopieren und per Tastatur-Kurzbefehl "Strg + V" in Spotify in eine (sinnvollerweise neue) Playlist einfügen.

3 FUNKTIONSWEISEN DER ALGORITHMEN

Im Rahmen der Lösungs-Umsetzung des "Remove Duplicates"-Problems sollen drei verschiedene Implementierungen in der Sprache python betrachtet werden.
Dabei soll das Hauptaugenmerk auf der ersten Implementierung liegen. Diese soll die Funktionalität der *dict*-Objekte in python nachempfinden, indem Sie die entsprechende hash-Funktion und die entsprechende Kollisions-Auflösungs-Strategie implementiert.
Zum Vergleich sollen zwei weitere Implementierungen betrachtet werden:

- der in Kapitel 1 beschriebene naive Ansatz
- eine Lösung, welche die tatsächlichen python *dict*-Objekte nutzt

Im Folgenden werden die Implementierungen mittels Pseudo-Code beschrieben.

3.1 Funktionsweise Haupt-Algorithmus: dict-like

Zunächst wird eine hash-table in Form eines Arrays initialisiert.

Haupt-Algorithmus: Hash-Tabelle initialisieren

Ergebnis : table, tableSize
1 tableSize ← 8;
2 table ← Array der Größe tableSize;

Anschließend werden die Elemente (URIs) aus der Sequenz-Datei des Nutzers in die angelegte hash-table geschrieben. Tritt dabei ein Element doppelt auf, wird dieses als Duplikat erkannt und ignoriert.
Als hash-Funktion wird dabei die python-Standardfunktion *hash()* eingesetzt. Die Strategie zur Kollisionsauflösung entspricht derjenigen, die bei python *dict*-Objekten Anwendung findet und lässt sich dem folgenden Pseudocode entnehmen.

Haupt-Algorithmus: Elemente in Array einfügen

```
   Daten : Sequenz − Datei
   Ergebnis : table
 1 für Jede URI in Sequenz − Datei tue
 2 |   index ← hash(URI);
 3 |   uriHandled ← flase;
 4 |   pertub ← None;
 5 |   tue
 6 |   |   wenn bereits Element in table[index] enthalten dann
 7 |   |   |   wenn table[index] == URI dann
 8 |   |   |   |   //Duplikat liegt vor - URI wird ignoriert
 9 |   |   |   |   uriHandled ← true;
10 |   |   |   sonst
11 |   |   |   |   wenn pertub besitzt Wert dann
12 |   |   |   |   |   pertub = pertub >> 5;
13 |   |   |   |   |   index ← ((5 ∗ slotindex) + 1 + pertub)%tableSize;
14 |   |   |   |   sonst
15 |   |   |   |   |   pertub = index;
16 |   |   |   |   Ende
17 |   |   |   Ende
18 |   |   sonst
19 |   |   |   table[index] ← URI;
20 |   |   |   uriHandled ← true;
21 |   |   Ende
22 |   solange !uriHandled;
23 |   wenn URIs in table > 2/3 ∗ tableSize dann
24 |   |   tue
25 |   |   |   tableSize ← tableSize ∗ 2
26 |   |   solange 4 ∗ |URIs in table| >= tableSize;
27 |   |   newTable ← table;
28 |   |   Kopiere alle URIs in table unter Errechnung neues Schlüssels in newTable;
29 |   |   table ← newTable;
30 Ende
```

Anschließend werden die in der hash-table enthaltenen URIs in eine Output-*.txt*-Datei geschrieben. Diese enthält damit die Duplikat-freie Liste der URIs und kann vom Anwender genutzt werden.

3.2 Funktionsweise Vergleichs-Algorithmus 1: Naiver Ansatz

Der in Kapitel 1 skizzierte naive Ansatz gestaltet sich grundlegend wie folgt.

Algorithmus Naiver Ansatz

```
   Daten : Sequenz − Datei
   Ergebnis : table
 1 table ← [];
 2 für Jede URI in Sequenz − Datei tue
 3 |   uriInTable ← false;
 4 |   für Jede processedUri in table tue
 5 |   |   wenn processedUri == URI dann
 6 |   |   |   uriInTable ← true;
 7 |   Ende
 8 |   wenn uriInTable dann
 9 |   |   //Duplikat erkannt, ignoriere es
10 |   sonst
11 |   |   table.append(URI);
12 |   Ende
13 Ende
```

Anschließend erfolgt die Ausgabe der Duplikat-freien Liste per Erzeugen einer Output-Datei.

3.3 Funktionsweise Vergleichs-Algorithmus 2: python dict

Die Funktionsweise des zweiten Vergleichs-Algorithmus ist diejenige, deren Verhalten in der Umsetzung des Haupt-Algorithmus angestrebt wird.
Folglich zeichnet sich der Quellcode der Implementierung durch die Nutzung von python-Standard-Operatoren aus. Der wesentliche Teil der Implementierung umfasst die folgenden Zeilen Code:

```
uriDict = {}
for uri in sequenzFile:
   if uri not in uriDict:
      uriDict[uri] = 1
   else:
      #ignore duplicate
      pass
```

Analog zu den anderen zwei Implementierung erfolgt im Anschluss eine Ausgabe aller Elemente des dicts per Output-File.

Alle drei initialen Implementierungen finden sich im Abgabeordner unter */Code/02_ImplementierungOhneVerbesserung*. Zum Aufruf der Algorithmen über die Kommandozeile muss die zu untersuchende *.txt*-Datei als Parameter angegeben werden. Beispielhaft sei hier ein Aufruf der ersten Implementierung gezeigt.

```
python 1_dictLike.py AimeeMann_980DS.txt
```

4 MESSDATEN

Zur Beschaffung von möglichst großen und verschiedenen Playlists als Messdaten wurde zunächst ein Test-Daten-Generator implementiert. Dieser wird in 4.1 näher beschrieben.
Im Folgenden wurde ein kurzes Programm implementiert, welches die Spotify Web-API nach Titeln anfragt und somit echte Datensätze erhält. Dieser Echt-Daten-Abgreifer wird in 4.2 näher beschrieben.
Zudem wurde für das Durchführen von Messreihen mit wachsender Playlist-Größe ein Vorgehen etabliert, um die benötigten .txt-Files zu erzeugen. Dieses wird in 4.3 erläutert.

4.1 Test-Daten-Generator

Der Aufruf des Generators erfolgt mit zwei optionalen Parametern und einem obligatorischem Parameter.

```
testDataGenerator.py <n> [dupFactor] [fileSuffix]
```

Der Parameter *n* entspricht der Anzahl der Elemente, welche die zu generierende Liste enthalten soll. Der *dupFactor* entspricht der Wahrscheinlichkeit mit der ein Element in der Liste ein Duplikat ist. Der *fileSuffix* kann bei mehreren ausgegebenen Dateien mit gleicher Parametrisierung für eine unterschiedliche Benennung genutzt werden.

Ein Beispiel für einen Aufruf könnte wie folgt aussehen.

```
python testDataGenerator.py 100000 .4 Run1
```

Im Wesentlichen erzeugt der Test-Daten-Generator eine *.txt*-Datei im sequence-File-Format. Die dynamischen Teile der URIs, (welche einen base62-Code [3] der Länge 22 darstellen) werden dabei zufällig erzeugt und ergeben im Regelfall keine tatsächlich existierende URI.
Die Funktionsweise des Generators ist dabei wie folgt zu beschreiben.

Algorithmus Test-Daten-Generator

```
   Daten : n, [dupFactor], [fileSuffix]
   Ergebnis : .txt − File
 1 wenn dupFactor nicht gegeben dann
 2 |   dupFactor ← zufälliger Faktor aus {0.0, 0.1, ...0.9};
 3 linkList ← [];
 4 für i in {0, ..., n − 1} tue
 5 |   wenn i > 0 dann
 6 |   |   wenn Zufallsentscheidung mit Wahscheinlichkeit dupFactor = positiv dann
 7 |   |   |   linkList.append(linkList[i − 1]);
 8 |   |   sonst
 9 |   |   |   linkList.append(neue zufällige URI);
10 |   |   Ende
11 |   sonst
12 |   |   linkList.append(neue zufällige URI);
13 |   Ende
14 Ende
15 erzeuge neues .txt − File ggf mit Suffix fileSuffix;
16 Schreibe in 1. Zeile von .txt − File "secquenceString";
17 Schreibe Elemente aus linkList in Zufalls-Reihenfolge in .txt − File;
```

Die resultierende Datei *(Beispielsweise "generatedTestData100000_0.4dupFac_Run1.txt")* ist dann bereit zur Verarbeitung durch einen implementierten Algorithmus oder lässt sich zum Erstellen mehrerer Playlist für eine Messreihe einsetzen. *(Siehe 4.3)*

Die Implementierung des Algorithmus findet sich im Abgabeordner unter *Code/01_Daten_Generieren/01_TestDaten/testDataGenerator.py*.

4.2 Echt-Daten

Für das Beschaffen von großen Playlists mit echten Track-URIs wurde eine Java-Anwendung umgesetzt, welche mithilfe eines Wrappers [4] die Spotify API abfragt. Um eine beliebig große Menge an URIs abzufragen, wurden das unten dargestellte Vorgehen für das Abfragen der API implementiert. Dabei müssen eine Mindest-Anzahl an URIs, die erhalten werden sollen und die URI eines Interpreten als Start-Interpreten angegeben werden.

Der entsprechende Java-Projektordner findet sich im Abgabeordner unter *Code/01_Daten_Generieren/02_EchtDaten/realDataGetter*.

Algorithmus Echt-Daten-Abgreifer

```
   Daten : minimalNoOfUris, startArtist
   Ergebnis : outputFile.txt
 1 Autorisierungsprozess zwischen Anwendung und Spotify API;
 2 outputTracklist ← [];
 3 searchProcessedArtists ← [];
 4 searchUnprocessedArtists ← [];
 5 processedArtists ← [];
 6 searchUnprocessedArtists.add(startArtistUri);
 7 tue
 8 |  für artist in api.getRelatedArtists(searchUnprocessedArtists.get(0)) tue
 9 |  |  wenn artist nicht in processedArtists dann
10 |  |  |  searchUnprocessedArtists.add(artist); für album in
   |  |  |   api.getAlbumsForArtist(artist) tue
11 |  |  |  |  für track in album tue
12 |  |  |  |  |  outputTracklist.add(track);
13 |  |  |  |  Ende
14 |  |  |  Ende
15 |  |  |  processedArtists.add(artist);
16 |  Ende
17 |  searchProcessedArtists.add(searchUnprocessedArtists.get(0));
18 |  searchUnprocessedArtists.remove(0);
19 solange |outputTracklist| <= minimalNoOfUris UND
    searchUnprocessedArtists ist nicht leer;
20 Erzeuge outputFile.txt;
21 Schreibe in Zeile 1 von outputFile.txt "secquenceString";
22 für URI in outputTracklist tue
23 |  Schriebe URI in outputFile.txt;
24 Ende
```

Analog zu 4.1 kann die generierte Datei im Anschluss verarbeitet werden.

4.3 Skalieren der Daten-Listen

Um später ganze Messreihen auszuführen, bei denen die Anzahl der Elemente, die in den Playlists enthalten sind, im Verlaufe der Messreihe zunimmt, wurde ein python-Skript erstellt, welches ein Sequenz-File einliest und nach eine Neue Datei erstellt, welche nur die Anzahl an Zeilen (zuzüglich der Zeile "secquenceString") beinhaltet, welche als zweiter Parameter übergeben wird.

Der Aufruf erfolgt beispielsweise wie folgt:

```
python cutToLen.py generatedTestData100_0.5dupFac.txt 40
```

Um viele Dateien für eine Messreihe zu generieren, wird ein kurzes shell-Skript je Messreihe angepasst und anschließend eingesetzt.

Das python- und das shell-Skript finden sich im Abgabeordner unter *Code/01_Daten_Generieren/03_DatenSkalieren*.

5 VORBEREITEN DER MESSUNGEN

Bevor der die Algorithmen-Laufzeiten untereinander und mit den jeweiligen theoretischen Laufzeiten verglichen werden, soll zunächst das Profiling und das Optimieren der einzelnen Implementierungen erfolgen. (Kapitel 5.1 - 5.3)
Die Vorbereitung der Messungen wird mit Kapitel 5.4 abgeschlossen, in dem beleuchtet wird, welche Messungen auf welche Weise erfolgen.

5.1 Profiling und Optimierung Haupt-Algorithmus

Generell soll für die Laufzeit-Messungen nur das Abarbeiten der Track-URIs, also das schreiben dieser in die hash-table bzw. das dict im Fokus stehen. Daher werden für alles Weitere die Funktionalitäten für das Schreiben des Ergebnis-Files auskommentiert.
Für den Hauptalgorithmus liegt die entsprechende *.py*-Datei im Abgabeordner unter *Code/03_Messen_und_Verbessern/01_Alg_1/1_dictLike_improvemento.py*.

Für das Profiling wird ein mittels *testDataGenerator.py* erstelltes *.txt*-File mit Größe $n = 100.000$ und der Duplikat-Wahrscheinlichkeit von $dupFac = 2\%$ verwendet.
Das Profiling erfolgt über die Nutzung des python-Profilers *cProfile* [5].

Die erste Visualisierung des Profils mittels *Snakeviz*[6]zeigt:

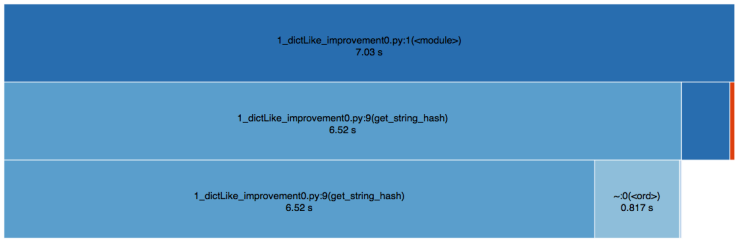

Abbildung 4: Darstellung initiales Profiling von Algorithmus 1

Die Funktion *get_string_hash* macht einen Großteil der Arbeitszeit aus. Diese Funktion ist der python-Standard-Funktion *hash()* nachempfunden. Das Implementieren dieser Funktion sollte dem tieferen Verständnis dienen.
Die wesentliche Verbesserung besteht also im ersetzten der *get_string_hash*-Funktion durch den python-Standard.
Der angepasste Algorithmus ist im Abgabeordner unter *Code/03_Messen_und_Verbessern/01_Alg_1/1_dictLike_improvement1.py* enthalten.

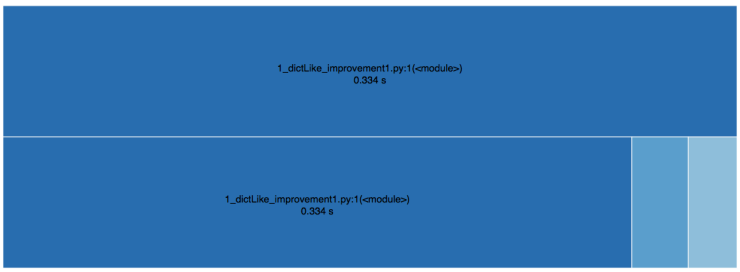

Abbildung 5: Darstellung Profiling von Algorithmus 1 nach 1. Verbesserung

Würde man diesem Prinzip der Optimierung folgen, also alles, wofür es python-Standard-Funktionalitäten gibt mit python-Standard-Funktionalitäten lösen, so würde der auf diesem Wege final optimierte Algorithums zu dem zweiten Vergleichs-Algorithmus (Kapitel 3.3.) identisch sein.
Daher soll im nächsten Schritt einer potentiellen Verbesserung das Tailoring angesetzt werden.

Betrachtet man die URIs, wird klar, dass die hash-Funktion nur auf den 22-stelligen Base62-Code angewendet werden muss, da der restliche Teil für alle URIs identisch ist.

```
https://open.spotify.com/track/757530vPBymdi31CtXstxP
https://open.spotify.com/track/1Qi256uJuMihknGuuFcQoC
```

Der entsprechend angepasste Algorithmus ist im Abgabeordner unter *Code/03_Messen_und_Verbessern/01_Alg_1/1_dictLike_improvement2.py* enthalten. Die Ausgabe-Funktion muss für eine Anwendung um den statischen Teil der URIs erweitert werden. Dies wurde in der Datei, obgleich die Ausgabefunktion für die Messungen auskommentiert bleibt, getan.

Da die Profile bezüglich ihrer gesamt benötigten Laufzeit für einen sichtbaren Unterschied zu nahe beieinander lagen, wurde das Profiling der vorherigen Version *1_dictLike_improvement1.py* und das Profiling der Verbesserten Variante mit einem *.txt*-File der Größe $n = 1.000.000$ wiederholt.
Die Verbesserung ist marginal. Bei mehrfachem Messen zeigte sich, dass durch Schwankungen anhand Gesamt-Laufzeit nicht erkannt werden konnte welche Implementierung schneller ist. Durch das Profiling jedoch stellte sich heraus, dass die benötigte Zeit für das Anwenden der Hash-Funktion bei in jedem Profil für den zweiten Algorithmus geringer war, weshalb die Anpassung im Rahmen des Tailorings als sinnvoll anerkannt und beibehalten werden kann.

Durchlauf	Vor Tailoring gesamt	Nach T. gesamt	Vor T. hash()	Nach T. hash()
1	2,65s	2,54s	0,185s	0,143s
2	2,61s	2,62s	0,183s	0,146s
3	2,74s	2,43s	0,188s	0,138s

Tabelle 2: Ergebnisse des Tailorings: Benötigte Zeit gesamt und für hash-Funktion

Abschließend seien hier die Zeitbedarfe für je einen Durchlauf einer Verbesserungs-Stufe des ersten Algorithmus für die oben verwendete Playlist mit $n = 100.000$ Elementen dargestellt:

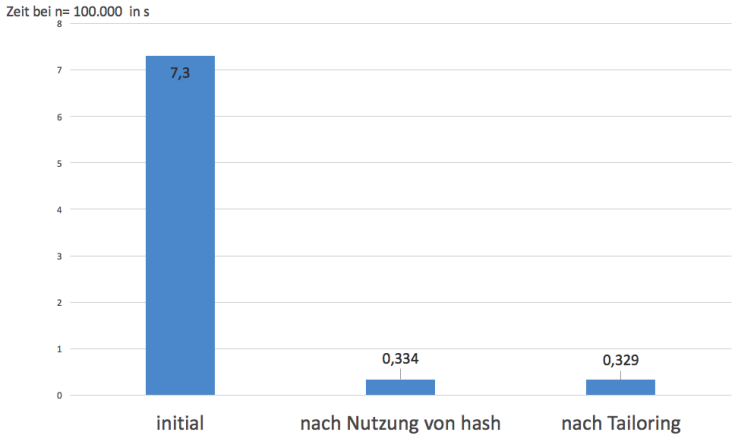

Abbildung 6: Zusammenfassung der Optimierung von Algorithmus 1

5.2 Profiling und Optimierung Vergleichs-Algorithmus 1

Analog zu der in Kapitel 5.1 gewonnenen Erkenntnis, soll die Verbesserung des Algorithmus in zwei Schritten ablaufen:

1. Ersetzen von implementierten Funktionalitäten, die mit python-Standards gelöst werden können (Iterieren durch Array ablösen durch Verwendung des *in*-Operators)
2. Anpassen des Algorithus analog zum in 5.1 durchgeführten Tailoring

Die entsprechenden Implementierungen sind im Abgabeordner unter *Code/03_Messen_und_Verbessern/02_Alg_2/* enthalten.

Die zusammengefasste Verbesserung gestaltet sich für ein Beispiel der Laufzeit mit $n = 10.000$ für Algorithmus 2 wie folgt:

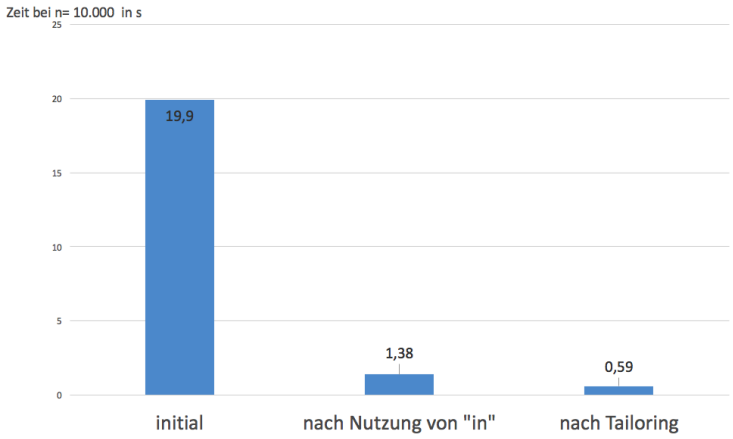

Abbildung 7: Zusammenfassung der Optimierung von Algorithmus 2

Die Verbesserung ist ähnlich zu der in Kapitel 5.1 erzielten, wobei jedoch das Tailoring für Algorithmus 2 eine größere Wirkung zeigt.

5.3 Profiling und Optimierung Vergleichs-Algorithmus 2

Da in dieser Umsetzung bereits die python-Standardfunktionen genutzt werden, soll hier nur das Tailoring betrachtet werden. Die entsprechenden Implementierungen sind im Abgabeordner unter *Code/03_Messen_und_Verbessern/03_Alg_3/* enthalten.

Die Verbesserung wird erst ersichtlich, wenn man die Anzahl der URIs in der Liste deutlich erhöht. Ab $n = 10.000.000$ zeigt sich der Effekt:

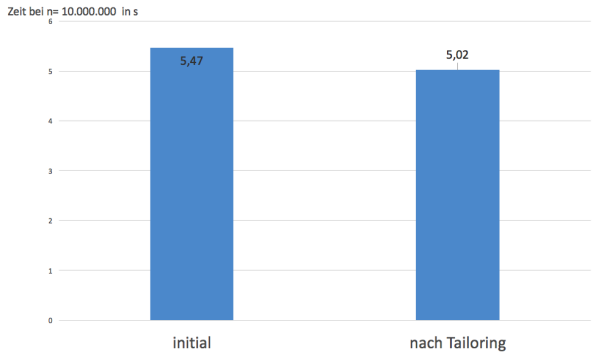

Abbildung 8: Zusammenfassung der Optimierung von Algorithmus 3

5.4 Wie wird gemessen?

Mit den vorangegangen Kapiteln 5.1 - 5.3 soll die Optimierung der Implementierungen für das Projekt als abgeschlossen betrachtet werden. Eine ausführbare Implementierung für alle drei Algorithmen findet sich im Abgabeordner unter */05_Finale_Implementierung/*.
Nun sollen die Implementierungen dahingehend erweitert werden, um mit ihnen Zeitmessungen durchführen zu können. Die Laufzeit wird dabei mithilfe der Funktion *time()* des python-Moduls *time* gemessen. Die Funktion *time()* liefert die seit dem 1.1.1970 vergangene Zeit in Sekunden als Gleitkommazahl. [7] Nach dem folgenden Schema wird somit die Laufzeit errechnet und in der Konsole ausgegeben.

```
import time

#read file...

#start time measuring
start = time.time()

#actual algorithm...

#stop time measuring
end = time.time()

#print it to console
elapsedTime = str(end-start)
print elapsedTime.replace('.',',')

#close file
#write output (commented out)
```

Um mehrere Messungen für eine Messreihe mit steigendem n durchzuführen wird wie folgt verfahren.

1. Erzeugen der Ausgangsdaten per *testDataGenerator.py* oder Java Anwendung *realDataGetter*
2. Erstellen mehrerer Playlists mit steigender Titelzahl n auf Grundlage der Ausgangsdaten mittels *cutToLen.py* und *shCutToLen.sh*
3. Ausführen der zu untersuchenden Implementierung für alle generierten *.txt*-Files mittes zuvor angepasstem shell-Skript *shRunScript.sh*

Das shell-Skript *shRunScript.sh* und die für die Laufzeit-Messungen angepassten Implementierungen sind im Abgabeordner unter */04_Mess-Skripte/* enthalten.
Damit sind die Vorbereitungen für die Messungen abgeschlossen. In den folgenden Kapiteln sollen die untenstehenden Aspekte untersucht werden.

1. Vergleich der Laufzeit zwischen Echt- und Test-Daten.
2. Abgleich der Laufzeit gegen die theoretische Laufzeit der Algorithmen.
3. Grafischer Vergleich der Algorithmen untereinander.
4. Durchführung statistischer Hypothesentests.

6 VERGLEICH ECHT VS. TEST-DATEN

In diesem Kapitel sollen stichprobenartig die Laufzeiten der einzelnen Algorithmen beim Verarbeiten von Echt- und Test-Daten betrachtet werden. Dabei wird wie folgt verfahren.

1. Für drei verschiedene Ausgangs-Künstler mithilfe des Echt-Daten-Abgreifers Ausgangs-Playlists erzeugen.
2. Mithilfe des Test-Daten-Generators drei Listen erstellen, wobei jede einer Echt-Daten-Liste in Länge und Dupikat-Faktor nachempfunden wird
3. Einkürzen der drei Listen auf die Längen $n = \{33.000, 100.000, 300.000\}$
4. Durchführen dreier Messungen pro Liste und Algorithmus.

Die unten dargestellten Graphen zeigen die aus den drei Messungen erzeugten Mittelwerte.
Der Vergleich für den Haupt-Algorithmus zeigt folgenden Graphen:

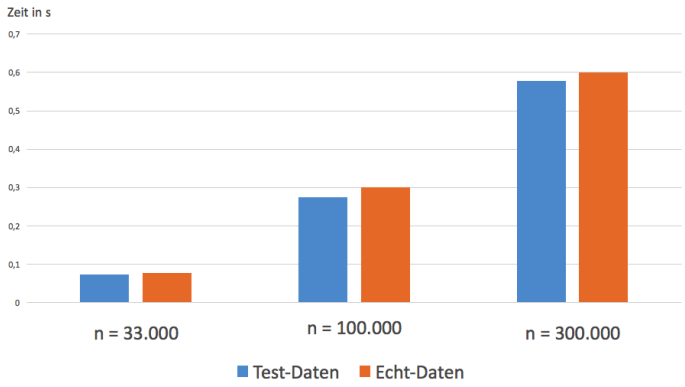

Abbildung 9: Vergleich Echt- und Test-Daten für Algorithmus 1

Die Graphen für die Vergleichs-Algorithmen zeigen ein ähnliches Bild.

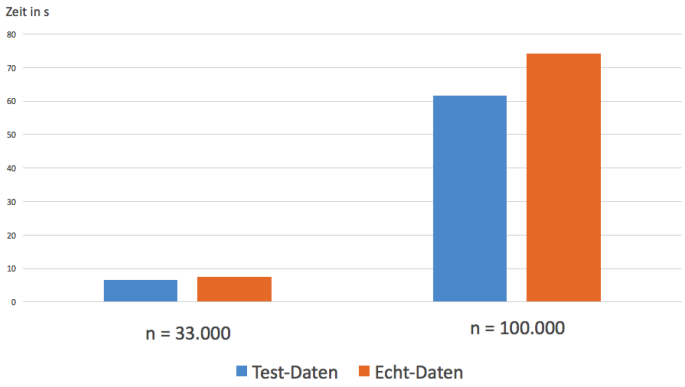

Abbildung 10: Vergleich Echt- und Test-Daten für Algorithmus 2

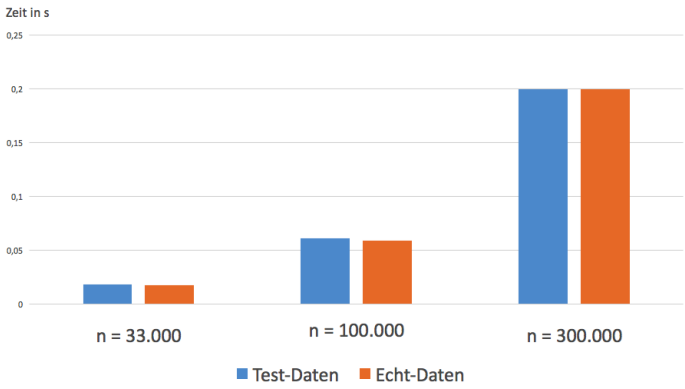

Abbildung 11: Vergleich Echt- und Test-Daten für Algorithmus 3

Es lässt sich festhalten, dass sich die Laufzeiten für alle drei Algorithmen zumindest für die vorgenommenen Stichproben nicht signifikant voneinander unterscheiden. Eventuelle erkennbare Unterschiede, wie das abweichen der Laufzeiten für $n = 100.000$ in Algorithmus 2 könnten mit weiteren Laufzeitmessungen untersucht werden. Für die Projektarbeit soll dieser Verglich jedoch genügen.
Für alle folgenden Messungen sollen Echt-Daten verwendet werden.

7 VERGLEICH MIT THEORETISCHER LAUFZEIT

Um pro Algorithmus den Vergleich mit der theoretischen Laufzeit anzustellen, werden pro Algorithmus folgende Laufzeitmessungen durchgeführt:

- Pro Messreihe: Playlists mit aufsteigender Titel-Anzahl $n = \{5000, 10.000, ..., 100.000\}$
- Durchführung von 9 Messreihen:
  Für drei verschiedene Playlists mit unterschiedlichem Ausgangs-Künstler
  Durchführen dreier Messungen pro Echt-Daten-Playlist.

Die Messergebisse werden zunächst in einem Boxplot-Diagramm dargestellt. Zur Überprüfung der theoretischen Laufzeit wird die y-Achse des Diagrammes entsprechend der jeweiligen theoretischen Laufzeit skaliert.

7.1 Haupt-Algorithmus

Das Boxplot-Diagramm für den Haupt-Algorithmus zeigt das folgende Bild:

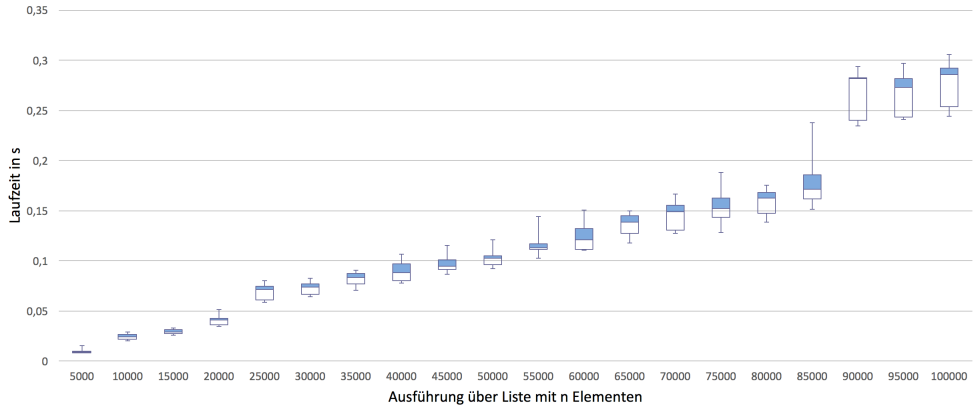

Abbildung 12: Darstellung der Laufzeitmessungen für Algorithmus 1 im Boxplot-Diagramm

Wird dieser mit der in Kapitel 1 beschriebenen theoretischen Laufzeit von $O = (n)$ skaliert, zeigt sich das folgende Bild:

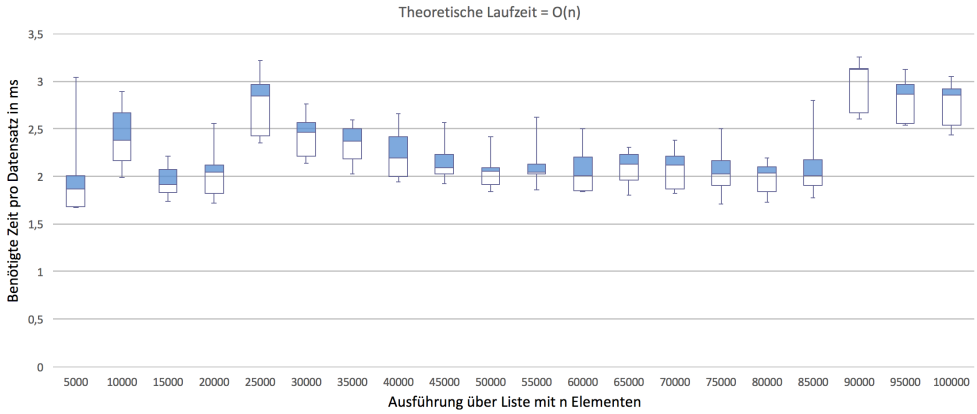

Abbildung 13: Laufzeitmessungen für Algorithmus 1 im skalierten Boxplot-Diagramm

Es wird ersichtlich, dass die Laufzeiten pro Datensatz nahezu in einer Geraden liegen, was die theoretische Laufzeit bestätigt.
Es lässt sich jedoch ein leichter Anstieg der Geraden erkennen. Der Grund dafür zeigt sich im unskalierten Boxplot-Diagramm. In diesem ist ein sprunghafter Anstieg der Laufzeit für die n-Werte $n = 25.000$ und $n = 90.000$ zu abzulesen. Als Ursache dafür lässt sich wiederum das Kopieren aller Werte der hash-table in die neu

angelegte, vergrößerte hash-table vermuten *(Siehe Kapitel 3.1 - Haupt-Algorithmus: Elemente in Array einfügen, Zeilen 32-29)*

7.2 Vergleichs-Algorithmus 1

Für den Vergleichs-Algorithmus 1 zeigt das Ausgangs-Boxplot-Diagramm und das mit der in Kapitel 1 hergeleiteten theoretischen Laufzeit von $O = (n^2)$ skalierte Diagramm, dass die Laufzeit-Überlegung als bestätigt betrachtet werden kann.

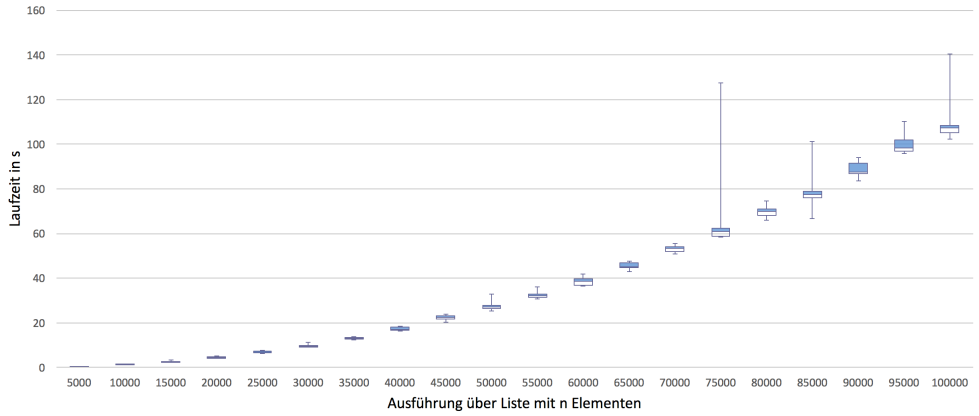

Abbildung 14: Darstellung der Laufzeitmessungen für Algorithmus 2 im Boxplot-Diagramm

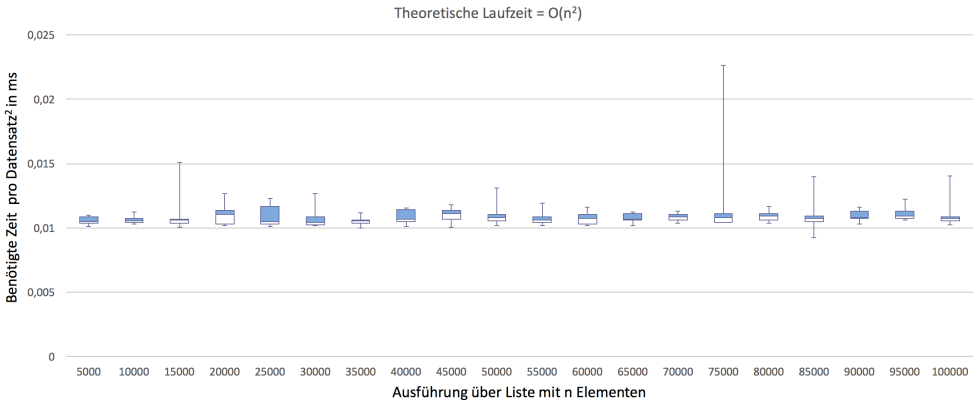

Abbildung 15: Laufzeitmessungen für Algorithmus 2 im skalierten Boxplot-Diagramm

7.3 Vergleichs-Algorithmus 2

Die Betrachtung der Laufzeiten in den entsprechenden Diagrammen für den dritten Algorithmus, zeigen, dass auch hier die theoretische Laufzeit bestätigt werden kann. Der für den Haupt-Algorithmus festgestellte sprunghafte Anstieg der Laufzeit für $n = 25.000$ und $n = 90.000$ wird ebenfalls, wenn auch sehr viel dezenter, ersichtlich.

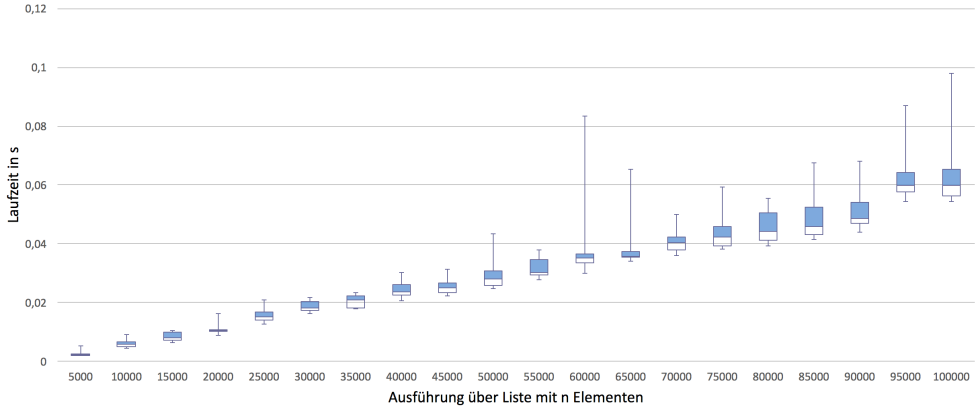

Abbildung 16: Darstellung der Laufzeitmessungen für Algorithmus 3 im Boxplot-Diagramm

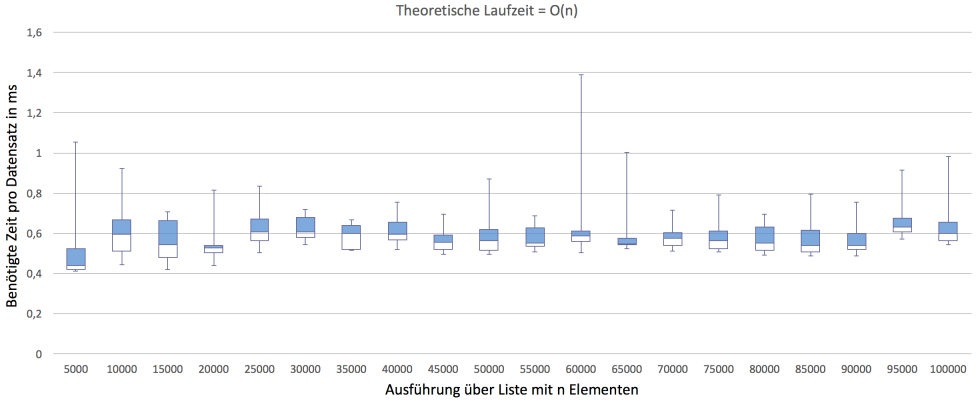

Abbildung 17: Laufzeitmessungen für Algorithmus 3 im skalierten Boxplot-Diagramm

8 VERGLEICH DER ALGORITHMEN

8.1 Grafischer Vergleich

Für den grafischen Vergleich der Algorithmen-Laufzeiten untereinander, werden für allen drei Algorithmen Laufzeitmessung für die folgenden Probleminstanzen durchgeführt:

- Für mehrere Playlists mit wachsende Titel-Anzahl $n = \{20, 40, ..., 1000\}$
  Pro n: 10 verschiedene Echtdaten-Tracklisten
  Pro Liste: 3 Messdurchläufe

Die Messreihen wurden über das Ablehnen von Messungen außerhalb des Bereiches $[\mu - \sigma, \mu + \sigma]$ bereinigt, wobei μ der Erwartungswert und σ die Standardabweichung der Messwerte ist.
Der direkte grafische Vergleich stellt sich wie folgt dar.
Die Laufzeit für Algorithmus 2 nimmt, wie in der Betrachtung der theoretischen

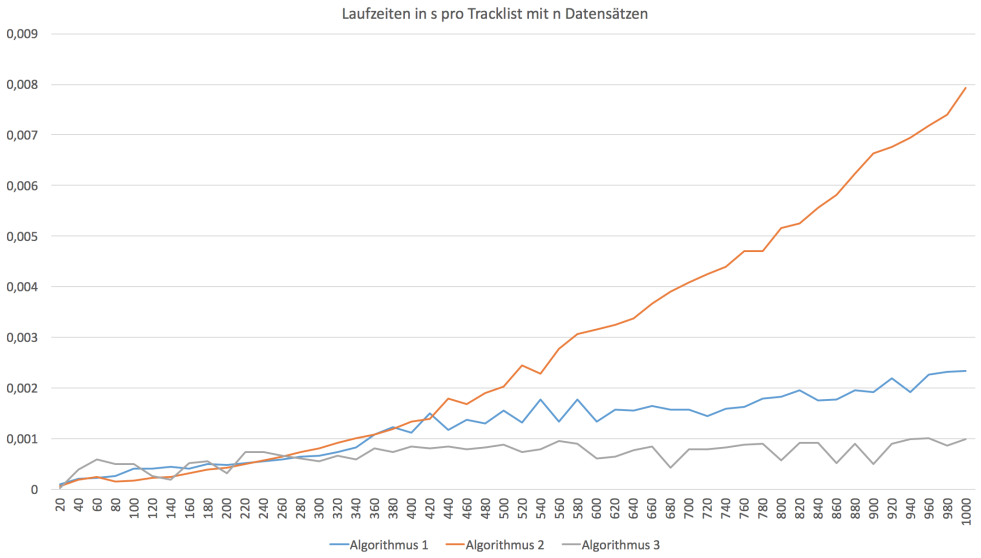

Abbildung 18: Direkter grafischer Vergleich der Algorithmen miteinander

Laufzeit bestätigt exponentiell zu. Um jedoch einen besseren Eindruck für das Laufzeitverhalten von Algorithmus 1 und algorithmus 3 zu gewinnen, wird für diese ein weiterer Vergleich mit höheren n-Werten angestellt:

- Für mehrere Playlists mit wachsende Titel-Anzahl $n = \{2000, 4000, ..., 100.000\}$
  Pro n: 10 verschiedene Echtdaten-Tracklisten
  Pro Liste: 3 Messdurchläufe

In diesem Vergleich wird die Relation der Laufzeiten deutlich. Ebenfalls sind für beide Implementierungen die in Kapitel 7.1 festgestellten sprunghaften Anstiege der Laufzeiten aufgrund der hash-Tabellen-Kopier-Operationen festzustellen. Insge-

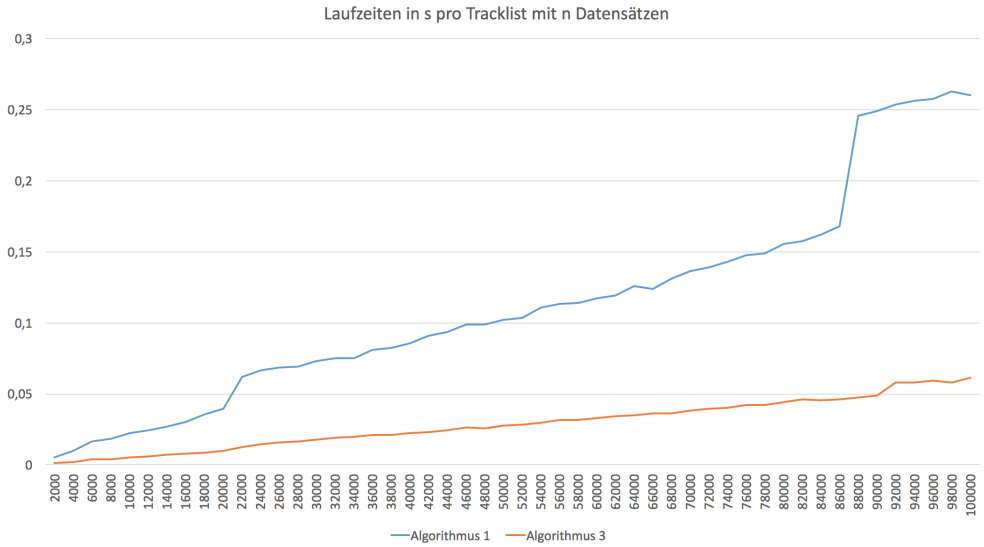

Abbildung 19: Direkter grafischer Vergleich der Algorithmen 1 und 3 miteinander

samt lässt sich aus der grafischen Betrachtung die folgende Ordnung der Algotithmen-Laufzeiten anzunehmen.

$$Alg3_{Laufzeit} < Alg1_{Laufzeit} < Alg2_{Laufzeit} \tag{6}$$

8.2 Statistischer Hypothesen-Test

Auf Grundlage der im vorherigen Kapitel durchgeführten Messungen für $n = \{20, 40, ..., 1000\}$ soll nun eine exakte Analyse mittels statistischer Hypothesentests durchgeführt werden. Dabei soll die angestellte Überlegung

$$Alg3_{Laufzeit} < Alg1_{Laufzeit} < Alg2_{Laufzeit} \tag{7}$$

als Ausgangspunkt verwendet werden.
Um zu überprüfen, ab welcher Playlist-Größe die einzelnen Teile der Relation sicher gelten, sollen durch die folgenden Hypothesentests drei Aussagen geprüft werden:

1. Algorithmus 1 ist schneller als Algorithmus 2
2. Algorithmus 3 ist schneller als Algorithmus 2
3. Algorithmus 3 ist schneller als Algorithmus 1

Algorithmus 1 schneller als Algorithmus 2
Aus der Vermutung, dass Algorithmus 1 schneller ist, als Algorithmus 2 ergeben sich für den linksseitigen Hypothesentest die Null-Hypothese

$$H_0 : \tilde{\mu_1} \geqslant \tilde{\mu_2} \tag{8}$$

und die Alternativ-Hypothese

$$H_0 : \tilde{\mu_1} < \tilde{\mu_2} \tag{9}$$

wobei $\tilde{\mu}$ je den Mittelwerten aller Messungen für eine Probleminstanz entspricht.
Nun soll überprüft werden, ob die Nullhypothese pro n bestätigt werden kann oder zu widerlegen ist. Dabei soll ein frei gewähltes Signifikanzniveau von $\alpha = 0,02\%$ verwendet werden.
Über die Differenzen aller Messwerte $d_i = MesswertAlg2_i - MesswertAlg1_i$ werden dazu der Erwartungswert μ_d und die Standardabweichung σ_d ermittelt. Auf

dieser Grundlage wird der t-Wert $t = \sqrt{n}\frac{\mu_d}{\sigma_d}$ ermittelt und mit dem Wert der t-Funktion für $t(0,98;29) \approx 2,15$ verglichen.
Die Nullhypothese ist dabei abzulehnen, wenn gilt $t < 2,15$
Das Resultat gestaltet sich wie folgt:

n	$t = \sqrt{n}\frac{\mu_d}{\sigma_d}$	$t(0,98;29)$	Ergebnis
20	-4,675596311	2,15	Null-Hypothese nicht abgelehnt
40	-2,267865616	2,15	Null-Hypothese nicht abgelehnt
60...380	...	2,15	Null-Hypothese nicht abgelehnt
400	1,551055352	2,15	Null-Hypothese nicht abgelehnt
420	0,540433623	2,15	Null-Hypothese nicht abgelehnt
440	4,598993754	2,15	Null-Hypothese abgelehnt
460	3,17978148	2,15	Null-Hypothese abgelehnt
480...980	...	2,15	Null-Hypothese abgelehnt
1000	35,12686104	2,15	Null-Hypothese abgelehnt

Tabelle 3: Ergebnisse Hypothesentest "Algorithmus 1 schneller als Algorithmus 2"

Folglich, kann bei einem angenommenen Signifikanzniveau von $\alpha = 0,02\%$ erst ab einer Playlistgröße von $n = 440$ angenommen werden, dass Algorithmus 1 eine geringere Laufzeit hat, als Algorithmus 2.

Algorithmus 3 schneller als Algorithmus 2
Das Verfahren für den linksseitigen Hypothesentest ist dem obigen analog.
Es gelten Null-Hypothese:

$$H_0 : \tilde{\mu_3} \geqslant \tilde{\mu_2} \tag{10}$$

und Alternativ-Hypothese

$$H_0 : \tilde{\mu_3} < \tilde{\mu_2} \tag{11}$$

Das oben genutzte Signifikanzniveau von $\alpha = 0,02\%$ soll auch hier angewendet werden. Es ergibt sich die folgende Tabelle:

n	$t = \sqrt{n}\frac{\mu_d}{\sigma_d}$	$t(0,98;29)$	Ergebnis
20	4,184966485	2,15	Null-Hypothese abgelehnt
40	-4,405691129	2,15	Null-Hypothese nicht abgelehnt
60...280	...	2,15	Null-Hypothese nicht abgelehnt
300	1,822671007	2,15	Null-Hypothese nicht abgelehnt
320	2,486409842	2,15	Null-Hypothese abgelehnt
340...980	...	2,15	Null-Hypothese abgelehnt
1000	39,26755684	2,15	Null-Hypothese abgelehnt

Tabelle 4: Ergebnisse Hypothesentest "Algorithmus 3 schneller als Algorithmus 2"

Folglich, kann bei einem angenommenen Signifikanzniveau von $\alpha = 0,02\%$ erst ab einer Playlistgröße von $n = 320$ und für die untersuchte Größe $n = 20$ angenommen werden, dass Algorithmus 3 eine geringere Laufzeit hat, als Algorithmus 2.

Algorithmus 3 schneller als Algorithmus 1
Diese Untersuchung erfolgt analog der zwei vorangegangenen.
Es gelten Null-Hypothese:

$$H_0 : \tilde{\mu_3} \geqslant \tilde{\mu_1} \tag{12}$$

und Alternativ-Hypothese

$$H_0 : \tilde{\mu_3} < \tilde{\mu_1} \tag{13}$$

Das oben genutzte Signifikanzniveau von $\alpha = 0,02\%$ soll weiterhin angewendet werden. Es ergibt sich die folgende Tabelle:

n	$t = \sqrt{n}\frac{\mu_d}{\sigma_d}$	$t(0,98;29)$	Ergebnis
20	7,690375185	2,15	Null-Hypothese abgelehnt
40	-2,382882348	2,15	Null-Hypothese nicht abgelehnt
60...180	...	2,15	Null-Hypothese nicht abgelehnt
180	-0,568085214	2,15	Null-Hypothese nicht abgelehnt
200	2,239280659	2,15	Null-Hypothese abgelehnt
220	0,059793204	2,15	Null-Hypothese nicht abgelehnt
240...280	...	2,15	Null-Hypothese nicht abgelehnt
300	1,338695858	2,15	Null-Hypothese nicht abgelehnt
320	2,243603114	2,15	Null-Hypothese abgelehnt
340...980	...	2,15	Null-Hypothese abgelehnt
1000	8,43624219	2,15	Null-Hypothese abgelehnt

Tabelle 5: Ergebnisse Hypothesentest "Algorithmus 3 schneller als Algorithmus 1'

Es kann also bei einem angenommenen Signifikanzniveau von $\alpha = 0,02\%$ erst ab einer Playlistgröße von $n = 320$ sowie für die untersuchten Größen $n = 20$ und $n = 200$ angenommen werden, dass Algorithmus 3 eine geringere Laufzeit hat, als Algorithmus 1.

9 FAZIT

Neben dem erlangen von Kenntnissen über das Durchführen von Laufzeituntersuchungen zieht der Autor dieser Arbeit den folgenden Schluss:

"Wenn schon python, dann Standard-Funktionalitäten verwenden."

LITERATUR/QUELLEN

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Jonathan Hartley. Time complexity - python wiki. https://wiki.python.org/moin/TimeComplexity. Zuletzt bearbeitet: 05.06.2017, Zugriff: 01.02.2018.

[3] Spotify uris and ids - spotify web api user guide. https://developer.spotify.com/web-api/user-guide/#spotify-uris-and-ids. Zugriff: 01.02.2018.

[4] A java wrapper/client for the spotify web api. https://github.com/thelinmichael/spotify-web-api-java.

[5] The python standard library documentation - 26.4.3. profile and cprofile module reference. https://docs.python.org/2/library/profile.html#module-cProfile. Zugriff: 01.02.2018.

[6] Snakeviz - a browser based graphical viewer for the output of python's cprofile module. https://jiffyclub.github.io/snakeviz/.

[7] The python standard library documentation - 15.3. time — time access and conversions. https://docs.python.org/2/library/time.html#time.time. Zugriff: 02.02.2018.