Aufgabe 2: Schwierigkeiten

Team-ID: 00005

Team: Bug-Busters

Bearbeiter/-innen dieser Aufgabe:  
Benjamin Kania

17. November 2024

Inhaltsverzeichnis

[Lösungsidee 1](#__RefHeading___Toc2232_4178761632)

[Umsetzung 1](#__RefHeading___Toc2234_4178761632)

[Beispiele 2](#__RefHeading___Toc2236_4178761632)

[Quellcode 2](#__RefHeading___Toc2238_4178761632)

# Lösungsidee

## Problemstellung

Ich brauche einen Algorithmus, bei dem den Aufgaben jeweils ein Index zugeordnet wird, der den Schwierigkeitsgrad angibt. Die Indizes aus den verschiedenen Klausuren müssen in eine gemeinsame Reihenfolge gebracht werden. Den Algorithmus erkläre ich unten unter Fall 1. Ein weiteres Problem ist, dass es in den verschiedenen Klausuren Widersprüche gibt, z. B. ist bei der Beispieleingabe „schwierigkeiten0“ die Aufgabe F in Klausur 2 leichter als G, aber in Klausur 4 ist es umgekehrt. Daher wird zunächst nach Widersprüchen gesucht und der Nutzer soll eine Reihenfolge angeben. Die Suche beschreibe ich unter Fall 2.

## Fall 1: Der Sortieralgorithmus, wenn es keine Widersprüche gibt.

Als erstes gebe ich jeder Aufgabe in jeder Klausur einen Index, der von seiner Position ausgeht (beginnend mit 1 links, nach rechts dann 2, 3 usw.).

Dieser Index repräsentiert die Schwierigkeit einer Aufgabe. Nun bringe ich die Klausuren so zusammen, dass die Reihenfolge der Indizes von Buchstaben gleichbleibt (der Abstand kann größer werden): ich fange damit an, für jede Aufgabe in der ersten Klausur zu schauen, ob diese Aufgabe auch in Klausur 2 vorhanden ist. Da jede Aufgabe insgesamt nur einen Index haben kann, muss ich sie gleichsetzen, und zwar erhöhe ich den kleineren Index auf den größeren. Damit die Reihenfolge gleichbleibt, werden danach in der Klausur, in der etwas verändert wurde (also die mit der Aufgabe mit kleinerem Index) die Werte auf den auf den erhöhten Wert plus +1, +2, +3 gesetzt, usw.

Nachdem wir das für jede Aufgabe in der ersten und zweiten Klausur gemacht haben, macht das Programm das mit jeder Klausur für jede andere Klausur. Am Ende muss ich die Aufgaben nur nach Indizes sortieren und nur die ausgeben lassen, die gefragt sind.

Dieser Algorithmus setzt jedoch voraus, dass es keine Konflikte in den Daten gibt, da dieser sonst in eine Endlosschleife gerät (siehe Fall 2).

## Fall 2: Suche nach Widersprüchen und Korrektur

Da unser Algorithmus nur Daten sortieren kann, die keine Widersprüche enthalten, brauchen wir eine Möglichkeit, diese zu beseitigen. Ich habe mich dafür entschieden, dass der Nutzer selbst diese Konflikte beseitigt (mit Hilfe des Programms), da es sonst passieren könnte, dass das Programm neue Konflikte erstellt, wenn es probiert, diese aufzulösen.

Ein Konflikt passiert immer dann, wenn die Reihenfolge von zwei Aufgaben in zwei Klausuren unterschiedlich ist, also zum Beispiel A < B und B < A (ich schreibe dafür kurz einmal AB und einmal BA). Also bilde ich für jede Klausur für alle Aufgaben alle Buchstabenpaare, bei denen der Buchstabe der leichteren Aufgabe links und der der schweren Aufgabe rechts steht

(z.B. in Klausur 1: A<<BC ergibt die Paare -> AB AC BC, aber zum Beispiel nicht CA). Außerdem bilde ich diese Paare einmal einheitlich sortiert (ich nutze eine alphabetische Sortierung). Dann schaut das Programm, ob es ein alphabetisch sortiertes Paar zweimal gibt. Dadurch findet man die Fälle, wo es mindestens zwei Informationen über die Reihenfolge der Schwierigkeit gibt. Dann vergleicht das Programm die dazugehörigen, nicht alphabetisch sortierten Paare. Wenn die nicht alphabetisch sortierten Paare beide gleich sind, gibt es kein Problem, wenn sie nicht gleich sind, wissen wir, dass es ein Konflikt gibt (also zum Beispiel einmal A< B und B<A: sortiert beides AB, aber unsortiert AB und BA -> ergibt Widerspruch).

Der Nutzer bekommt dann eine Eingabeaufforderung (Prompt) im Terminal mit dem Hinweis, dass es einen Konflikt gibt, und wird gefragt, wie er diesen auflösen will. Er kann dann entweder einen Buchstaben löschen oder einheitlich sagen, welcher Buchstabe größer (bzw. welche Aufgabe schwieriger) sein soll. Manche Konflikte kann man so jedoch immer noch nicht auflösen, weil durch das Auflösen, immer neue Konflikte entstehen. Diese Konflikte muss der Nutzer per Hand in der Datei selbst lösen. (Beispiel …)

# Umsetzung

Zunächst werden die Daten (Klausuren) geladen und die < entfernt, da diese nicht benötigt werden.

## Umsetzung von Fall 1

Zunächst werden die Klausuren und die Aufgaben in Vektoren geschrieben, die Hashmaps mit den Einträgen Buchstabe und Index enthalten. Dann wird eine „while-Schleife“ durchlaufen, bis keine Indizes mehr geändert werden. In dieser „while-Schleife“ setze ich eine Variable namens „changed“ auf (false), um die „while-Schleife“ nach diesem Durchlauf zu stoppen. Dann gehe ich mit zwei „for-Schleifen“ durch jede mögliche Kombination an Klausuren (in Hashmaps gespeichert). Anschließend schaue ich für jeden Buchstaben in die erste Hashmap und prüfe, ob dieser in der zweiten enthalten ist. Wenn dies der Fall ist, überprüfe ich, dass die beiden Werte nicht übereinstimmen. Denn wenn beide übereinstimmen, muss ich nichts ändern. Daraufhin rechne ich mit der „max () Funktion“ den größten Wert aus. Diesen Wert füge ich für beide Klausuren in die Hashmap ein. Außerdem setze ich die Variable „changed“ wieder auf (true), damit sich die „while-Schleife“ noch einmal ausführt. Mit einer weiteren „if -Abfrage“ schaue ich, welcher Wert aus welcher Klausur größer war. Daraufhin wird eine Funktion aufgerufen, welche die restlichen Buchstaben entsprechend erhöht.

## Umsetzung von Fall 2

Um die Konflikte zu finden, nutze ich externe Bibliotheken. Die erste ist Polars, damit ich mit Dataframes arbeiten kann. Die zweite ist Itertools, damit ich meine Buchstabenpaare nach dem Alphabet sortieren kann. Als erstes wird eine Funktion namens „make\_df“ aufgerufen, die einen Dataframe mit den sortierten Paaren, den unsortierten Paaren und den Klausurnummern erstellt und zurückgibt.

Diesen Dataframe nutzt daraufhin die Funktion „locate\_conflicts“, welche eine Aggregation ausführt, um die sortierten Paare zusammenzufassen und nur diese zu benutzen, die mehr als einmal vorkommen. Sie haben außerdem noch eine Spalte mit den unsortierten Paaren, zusammengefasst in einem Array aus Strings.

Die Spalten dieses Dataframes werden dann in einzelne Vektoren aufgeteilt, damit man diese besser weiterverarbeiten kann.

Die sortierten Paare und die unsortierten Paare werden dann in eine Funktion namens „get\_conflicts“ gegeben.  
Diese Funktion erstellt eine große „for-Schleife“, die jedes sortierte Paar und die Anzahl der absolvierten Durchläufe angibt.

Daraufhin finde ich den Vektor für das sortierte Paar mit Hilfe der absolvierten Durchläufe.

Aus diesem Vektor entferne ich alle Duplikate, sodass zwei gleiche, unsortierte Paare zu einem werden. Somit kann ich mit dem Aufruf von „len ()“ überprüfen, ob der Vektor ein Element oder mehrere hat. Wenn er mehr als ein Element besitzt, wissen wir, dass es zwei unterschiedliche Paare gibt und somit ein Konflikt vorliegt. Wenn der Vektor nur ein Element besitzt, gibt es kein Konflikt, weil alle Paare gleich sind.

## Eingabefunktionalität (Prompt)

Wenn ein Konflikt gefunden wurde, zeigt das Programm die jetzigen Klausurdaten an, indem es über alle Elemente des Datenvektors iteriert. Daraufhin kann der Nutzer sich aussuchen, ob er entweder einheitlich festlegen möchte, in welcher Reihenfolge die Paare sein sollen oder ob er einen der beiden Buchstaben aus den Daten löschen will. Wenn er sich dafür entscheidet, das Paar einheitlich festzulegen, versucht das Programm in jedem Eintrag des Klausur-Vektors die Funktion „replace\_pairs“ aufzurufen. Diese Funktion schaut, ob das Paar in der richtigen Reihenfolge vorhanden ist und verändert den Klausur-Vektor, falls dies der Fall ist.

Wenn ein Buchstabe verändert werden soll, wird vom Nutzer auch die betreffende Klausur angegeben. Anhand dieser Klausurnummer wird mit der Funktion „replace“ der Buchstabe entfernt.

# Beispiele

Bei welcher Einagbe gibt es welche Ausgabe? Mit/ohne Widerspruch zeigen

Genügend Beispiele einbinden! Die Beispiele von den BWINF-Webseiten sollten hier diskutiert werden, aber auch eigene Beispiele sind sehr gut – besonders wenn sie Spezialfälle abdecken. Bitte jedoch nicht 30 Seiten Programmausgabe hier einfügen!

# Quellcode

Schleife für Sortierung

Schleife für Widersprüche

Unwichtige Teile des Programms sollen hier nicht abgedruckt werden. Dieser Teil sollte nicht mehr als 2–3 Seiten umfassen, maximal 10.