Bonusaufgabe: Zara-Zackings-Zurückkehr

Teilnahme-ID: 61045

Bearbeiter/-in dieser Aufgabe:
Linus Schumann

24. April 2022

Inhaltsverzeichnis

1 Lösungsidee

Zuerst wird die Idee zur Lösung der beiden Problemteile (A und B) genauer beschrieben.

1.1 Finden der richtigen Karten (Teil A)

1.1.1 Verständnis vom exklusiven Oder

Zuerst muss das Verständnis vom exklusiven Oder genau geklärt werden. Mit zwei Eingaben ist dies nicht sonderlich kompliziert, da der Output nur wahr ist, wenn genau ein Input wahr ist und genau ein Input falsch ist.
Bei mehr als zwei Inputwerten ist der Output nur wahr, wenn eine ungerade Anzahl an Inputwerten auch wahr ist.
Dies kann man in der boolschen Algebra durch folgende Notation (Gleichung (1)) ausdrücken.

$$X = A \oplus B \oplus C \oplus ... \tag{1}$$

Außerdem kann man aus dieser Gleichung entnehmen, dass man, um die XOR-Summe zu bilden, iterativ die Gleichung von links nach rechts abarbeiten kann.

1.1.2 Intuitive/Einfache Lösungsidee

Die intuitive und einfache Lösungsidee, um alle Lösungskarten zu finden, ist einfach alle möglichen Kombinationen auszuprobieren. Die Anzahl der Möglichkeiten kann mit Hilfe von Gleichung (2) berechnet werden. Dies funktioniert zwar relativ schnell für kleinere Eingabedateien wie z.B. stapel0 (4845 Kombinationen), allerdings gibt es viel zu viele Kombinationen bei z.B. stapel2 ($\approx 4.732 * 10^{13}$ Kombinationen). Daher ist diese Lösungsidee nicht Zielführend.

$$C(n,r) = \frac{n!}{r!(n-r)!} \quad (Für\ n \geq r \geq 0) \tag{2}$$

n = Anzahl an Zahlen insgesamt
r = Anzahl an Zahlen einer Kombination
C = Anzahl an Kombinationen

1.1.3 Weitere Überlegungen

Da die einfache Lösungsidee zu keinem Ergebnis führt, mussten weitere Überlegungen gemacht werden, die die Effizienz des Algorithmus deutlich verbessern. Diese Ideen werden nun im folgenden aufgeführt.

1. Gleiche Werte ergeben im exklusiven Oder immer 0
2. Die XOR-Summe aus allen Häuserkarten und der Schlüsselkarte ist 0, da die Schlüsselkarte ja das exklusive Oder der Häuserkarten abbildet.
3. Aus diesem Grund muss man nicht wie z.B. bei stapel2 immer 10er Kombinationen berechnen, sondern man kann dies in zwei Teile aufteilen (z.B. 5er Kombinationen und 6er Kombinationen). Danach kann man dann gleiche Werte suchen, wobei ein Wert aus einer 5er Kombination und ein Wert aus einer 6er Kombinationen stammt.

1.2 Mögliche Karten für bestimmtes Haus (Teil B)

Um mögliche Karten für ein bestimmtes Haus zu finden, kann man die in Teil A gefundenen Karten aufsteigend sortieren. Es ist allerdings nicht möglich die Schlüsselkarte unter den anderen Karten zu finden, da jede Karte, die "Schlüsselkarte" bzw. das exklusive Oder der anderen Karten ist. Daher gibt es drei unterschiedliche Szenarien, wo die Schlüsselkarte zu finden sein könnte:

1. Schlüsselkarte liegt in der sortierten Liste vor dem Index der gesuchten Woche
   ⇒ Die richtige Karte liegt an der Stelle: Index der Woche + 1
2. Schlüsselkarte liegt in der sortierten Liste auf dem Index der gesuchten Woche
   ⇒ Die richtige Karte liegt auch an der Stelle: Index der Woche + 1

3. Schlüsselkarte liegt in der sortierten Liste nach dem Index der gesuchten Woche
   ⇒ Die richtige Karte liegt an der Stelle: Index der Woche

Daraus ergeben sich zwei Stellen an denen die richtige Karte liegen kann (Index der Woche bzw. Index der Woche + 1).

1.3 Finaler Ablauf des Algorithmus

Nun wird der grundlegende Ablauf des Algorithmus beschrieben, die Umsetzung der einzelnen Schritte wird genauer in Abschnitt 2.3 erklärt.

1. Schritt: **Datei einlesen (Umsetzung: Abschnitt 2.3.1)**
   Zuerst wird die Datei eingelesen und es werden, sowohl die Karten als auch die anderen Daten gesichert.
2. Schritt: **XOR-Summen berechnen und speichern (Umsetzung: Abschnitt 2.3.2)**
   Nun wird wie in Abschnitt 1.1.3 beschrieben, alle Kombinationen berechnet. Die größe dieser Kombinationen liegt dabei bei der Häfte der Gesamtanzahl an Karten (abgerundet). Diese Werte werden dann für jede Kombination mit den passenden Indices gespeichert.
3. Schritt: **XOR-Summen aufsteigend sortieren (Umsetzung: Abschnitt 2.3.3)**
   Danach werden diese eben berechneten Werte aufsteigend sortiert.
4. Schritt: **2. XOR-Summen berechnen und nach gleichen Werten suchen (Umsetzung: Abschnitt 2.3.4)**
   Jetzt wird für jede Kombination mit der Größe der Gesamtanzahl an Karten (aufgerundet) die Liste, die in den letzten Schritten berechnet wurde, nach gleichen Werten durchsucht. Wenn ein gleicher Wert gefundenen wurde und kein Index doppelt vorhanden ist, ist die Lösung gültig.
5. Schritt: **Lösen von Teil B (Umsetzung: Abschnitt 2.3.5)**
   Im letzten Schritt wird noch Teil B gelöst. Dabei werden die Karten aufsteigend mit Hilfe des Bubblesort-Algorithmus sortiert und nach den Überlegungen in Abschnitt 1.2 die beiden möglichen Karten ausgegeben.

1.4 Alternativer Ablauf

Alternativ zu dem im letzten Abschnitt beschriebenen Ablauf ist auch folgender ähnlicher Ablauf möglich. Dabei werden als Unterschied zum anderen Ablauf die 2. XOR-Summen auch in der gleichen Liste wie die 1. XOR-Summen gespeichert. Der 3. Schritt der Sortierung kann dann hinter diese Berechnung und Speicherung der 2. XOR-Summen geschoben werden. Dies bietet dann die Möglichkeit ein einziges Mal über die Gesamte Liste zu iterieren und so gleiche Werte zu finden.

Dieser Ablauf ist in der Theorie deutlich schneller, allerdings wird auch sehr viel mehr RAM-Kapazität benötigt, die in diesem Fall nicht zur Verfügung steht. Somit konnte dieser Ablauf nicht implementiert werden.

2 Umsetzung

2.1 Allgemeines

Im folgenden wird die Umsetzung, der in Abschnitt 1 beschriebene Lösungsidee, näher erläutert. Grundsätzlich wurde diese Idee dabei in Java, genauer gesagt in den Dateien "Bonusaufgabe-Zara-Zackigs-Zurueckkehr.java" und "xorSumIdx.java" implementiert. In der Datei "Main.java" wird die Main-Funktion definiert und die Eingabedatei ausgewählt. Alle diese eben genannten Dateien befindet sich unter dem Haupt-Verzeichnis der Aufgabe im Verzeichnis "./source/".

Um das implementierte Programm zu starten, kann das Batch-Script (Windows) oder das Shell-Script (Mac, Linux) genutzt werden. Beide befinden sich im Haupt-Verzeichnis der Aufgabe. Eine direkte Ausführung der .jar-Datei ist nicht möglich, da die Verzeichnis-Pfade auf das Verzeichnis des Batch- bzw. Shell-Script angepasst wurden.

Unter dem Haupt-Verzeichnis der Aufgabe im Verzeichnis "./beispieldaten/" befinden sich alle in dieser Dokumentation aufgeführten Beispiele und unter dem Verzeichnis "./beispielausgaben" befinden sich dementsprechend die gesicherten Ausgaben, die auch bei Ausführung des Programms auf der Konsole ausgegeben werden. Damit letztere besser von den Beispieldaten unterschieden werden können, werden diese mit der Dateiendung ".out" gespeichert, sind aber im Klartext lesbar und können dementsprechend wie ganz normale ".txt" Dateien geöffnet werden.

2.2 Verwendete Module

Für die Implemtierung in Java, werden verschiede Module benötigt, deren Aufgabe und Vorteile im folgenden beschrieben wird.

java.io.File, java.io.Scanner und java.io.FileNotFoundException Mit Hilfe des "File" Modules wird die Datei geladen und als File Objekt gespeichert. Mit dem Scanner Module wird dann danach dieses File Objekt gelesen. Um Fehler beim Laden der Datei "aufzufangen", muss zusätzlich noch das Module "FileNotFoundException" importiert werden.

java.io.PrintStream Durch dieses Modul kann nicht nur direkt auf die Konsole, sondern auch in eine Datei geschrieben werden.

java.util.ArrayList und java.util.List Mit diesen Modulen ist es möglich Listen zu erstellen, die im Vergleich zu einem normalem Array keine feste Länge haben.

java.util.Arrays Das Modul "Arrays" kann zum Beispiel ein Integer-Array in einen lesbaren formatierten String konvertieren, der zum Beispiel direkt auf die Konsole geschrieben werden kann.

2.3 Einzelne Schritte des Algorithmus

2.3.1 Datei einlesen

Beim Einlesen der Datei wird die Funktion "readFile" genutzt, die alle 3 Konstanten als Integer Werte einließt und danach alle Karten in ein Array. Dabei wird jede Karte in diesem Array auf 2 Long Werte aufgeteilt, da der größte Datentyp in Java Long mit 64 bits ist. Bei den größeren Beispielen werden allerdings 128 bit Karten verwendet, daher wird ein Array der Länge 2 verwendet, um eine Karte zu speichern.

2.3.2 XOR-Summen berechnen und speichern

Um die ersten Kombinationen (mit der Länge: Gesamtanzahl der Karten / 2 (abgerundet)) zu berechnen, daraus die XOR-Summe zu bilden und danach zu speichern, wird die Funktion "initCombinations" aufgerufen. Diese Funktion erstellt zuerst, ein Array mit allen ganzahligen Werten von 1 bis zur Gesamtanzahl an Karten. Danach wird die Funktion "combinations" gestartet, in der rekursiv alle Kombinationen berechnet werden. Wenn in dieser Rekursion eine vollständige Kombinationen berechnet wurde, wird für diese die Funktion "calculateXorSumAndSave" aufgerufen. In dieser wird die XOR-Summe, wie in Abschnitt 1.1.1 beschrieben, berechnet und zusammen mit den Indices der Karten dieser Kombinationen gespeichert. Dazu wird ein neues Objekt der Klasse "xorSumIdx" erstellt, das passende Attribute für diese Werte besitzt. Dieses Objekt wird dann anschließend der global definierten ArrayList "xorSums" hinzugefügt.

2.3.3 XOR-Summen aufsteigend sortieren

Zum Sortieren der XOR-Summen wird der Quicksort Algorithmus genutzt, der in Abschnitt 2.4.2 genauer beschrieben wird. Dieser wird mit der Funktion "quickSort" gestartet. Alternativ könnten auch in Java eingebaute Sortieralgorithmen genutzt werden, die manchmal etwas schneller sind.

2.3.4 2. XOR-Summen berechnen und nach gleichen Werten suchen

Nun werden die 2. XOR-Summen berechnet (mit der Länge: Gesamtanzahl der Karten / 2 (aufgerundet)). Dazu werden wieder die Funktionen "initCombinations" und "combinations" genutzt. Der Unterschied liegt dann nur darin, dass bei einer vollständigen Kombinationen die Funktion "calculateXorSumAndSearch" aufgerufen wird. Dabei wird dann die "xorSums" Liste mit Hilfe der binären Suche (Abschnitt 2.4.3)

nach der neu berechneten XOR-Summe durchsucht. Bei einem Treffer wird die Rekurion gestoppt und die Kombination der Liste und die neu berechnete Kombination werden gesammelt mit der Funktion ”printResult” ausgegeben.

2.3.5 Teil B lösen

Um Aufgabenteil B zu lösen, wird zuerst die Liste mit Hilfe des Bubblesort Algorithmus (Abschnitt 2.4.1) aufsteigend sortiert. Die Verwendung von einem langsamen Algorithmus macht in diesem Fall keinen Unterschied, da maximal 11 Karten sortiert werden. Dann wird den User gefragt, von welcher Woche er die möglichen Karten haben möchte. Von dem Index dieser Woche wird dann die Karte, sowie die nächste Karte ausgegeben (vgl. Abschnitt 1.2).

2.4 Verwendete bekannte Algorithmen

2.4.1 Bubblesort

Der Bubblesort Algorithmus ist ein Sortieralgorithmus, der eine Laufzeitkomplexität von $\mathcal{O}(n^2)$ hat und damit eher langsam ist. Der Algorithmus funktioniert so wie in Abb. 1 veranschaulicht. Grundsätzliches wird für jedes Element die ganze Liste einmal durchgegangen. Da immer das aktuelle Elemente mit dem nächsten verglichen wird und event. getauscht wird, ist nach jedem Durchlauf das größte Element ganz hinten. Daher wird das Ende der Liste für jeden Durchlauf um eins nach vorne verschoben.

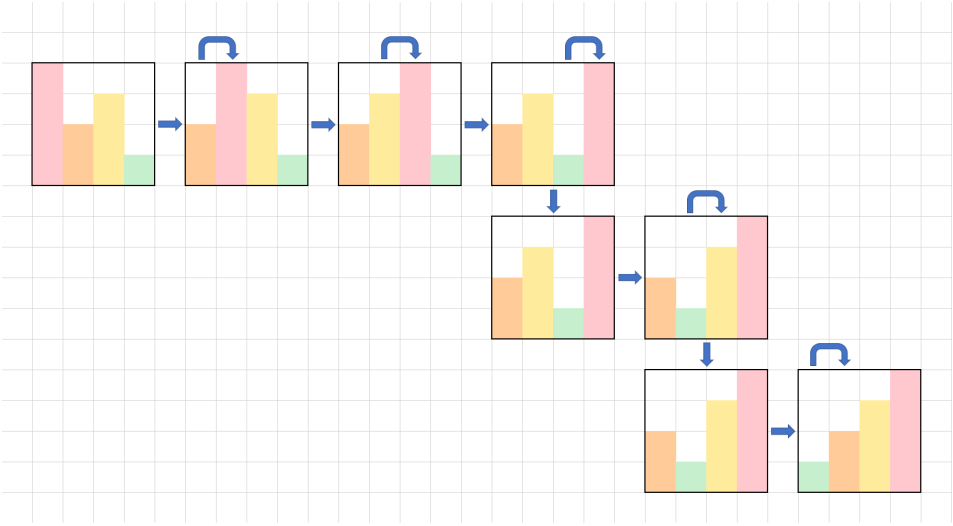

Abbildung 1: Funktionsweise des Bubblesort Algorithmus

2.4.2 Quicksort

Der Quicksort Algorithmus ist ein Sortieralgorithmus, der eine durchschnittliche Laufzeitkomplexität von $\mathcal{O}(n \log n)$ hat, daher zu den schnellsten Sortieralgorithmen gehört, aber auch eher komplexer ist. Grundsätzlich funktioniert der Algorithmus rekursiv und die Liste wird immer weiter in Teillisten geteilt. Dabei wird die Liste immer am Pivot Element geteilt, sodass eine Liste mit Werten die kleiner als das Pivot Element und eine Liste mit Werten die größer als das Pivot Element sind entsteht.

Teilen der Liste Um die Liste an Hand des Pivot Element passend zu teilen, werden zwei Pointer erstellt (low und high), die auf das erste bzw. das letzte Element zeigen. Dann wird so lange der high-Pointer verringert wie das Element an dieser Stelle größer als das Pivot Element ist. Danach passiert das gleiche nur umgekehrt mit dem low-Pointer. Danach steht, wenn der low-Pointer noch kleiner als der high-Pointer ist, an der low-Position ein Element das größer als das Pivot Element ist und an der high-Position ein Element das kleiner als das Pivot Element ist. Daher müssen nun diese beiden Elemente getauscht werden. Der Algorithmus endet dann, wenn die eben genannte Bedinung, dass der low-Pointer kleiner als der high-Pointer sein muss, nicht mehr gegeben ist. Dann wird, wenn das Pivot Element kleiner als das Element an der Stelle des low-Pointers ist, das Pivot Element noch mit jenem Element getauscht. Dann wird die aktuelle Stelle des Pivot Element zurückgegeben, damit die Liste an dieser Stelle geteilt

werden kann. Einen Einblick in den Grundsätzlichen Ablauf gibt Abb. 2 und das Teilen wurde in der Funktion "quickSortDivideList" umgesetzt.

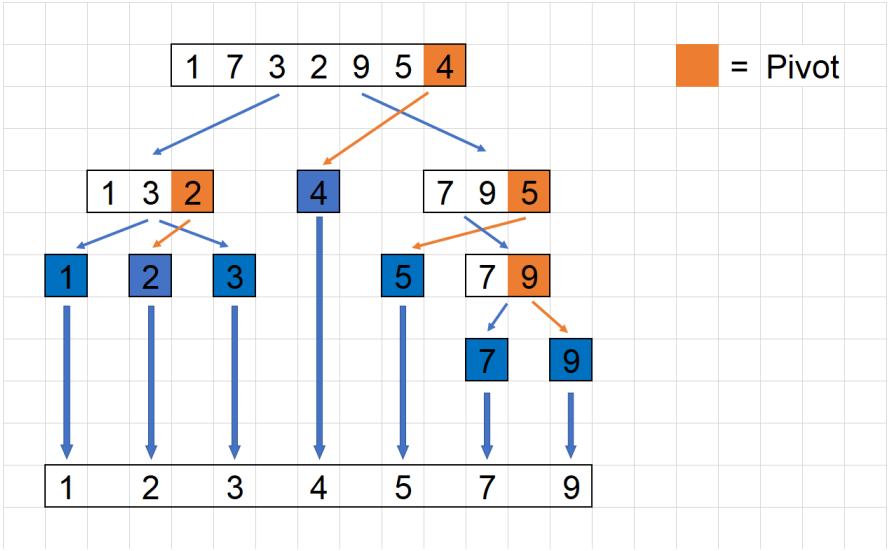

Abbildung 2: Funktionsweise des Quicksort Algorithmus

2.4.3 Binäre Suche

Grundsätzlich gibt es zwei unterschiedliche Arten eine Liste zu durchsuchen, die lineare und die binäre Suche. Bei der linearen Suche muss dabei die Liste im Gegensatz zur binären Suche nicht sortiert sein. Die lineare Suche hat mit $\mathcal{O}(n)$ im Vergleich zur binären Suche mit $\mathcal{O}(\log_2 n)$(im besten Fall sogar $\mathcal{O}(1)$) allerdings eine sehr viel größere Laufzeitkomplexität. Daher ist vor allem bei sehr langen Listen und sehr vielen Suchvorgängen, wie in diesem Fall, die binäre Suche deutlich schneller.

Die grundsätzliche Idee der binären Suche liegt darin die Liste immer weiter einzugrenzen. Dazu werden zwei Begrenzungen (low und high) genutzt. Zuerst wird dann die Mitte der Liste berechnet und dieses Mittelelement mit dem gesuchten Element verglichen. Danach wird nur der Teil der Liste weiterverwendet in dem das gesuchte Element liegen könnte. Am Ende bleibt nur noch ein Element. Dieses Element muss dann entweder mit dem gesuchten Element übereinstimmen oder das gesuchte Element ist nicht in der Liste vorhanden.

In Abb. 3 wird eine Beispielsuche nach der Zahl 12 gezeigt.

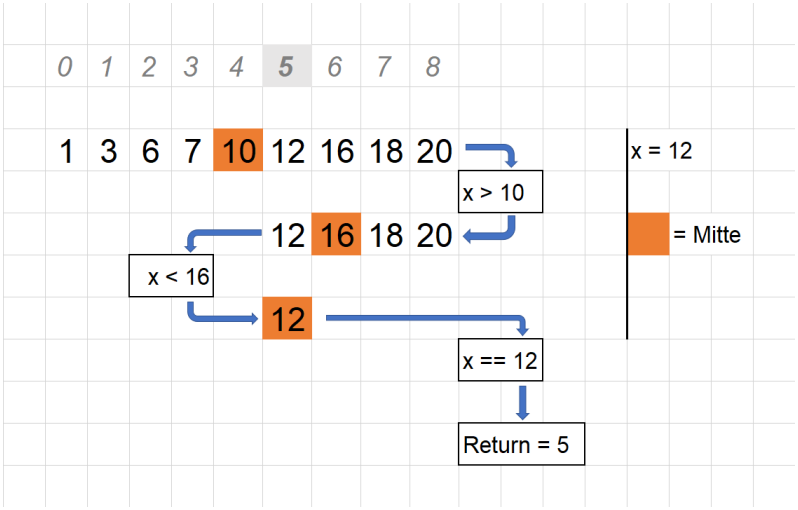

Abbildung 3: Funktionsweise der binären Suche

3 Beispiele

3.1 Beispiel 0

```
Solution keys at: [2, 3, 5, 9, 11]
Solution keys:
  2 : 0 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1
  3 : 1 1 1 1 1 1 1 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 1
  5 : 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0 0 0
  9 : 1 0 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0
 11 : 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 1 0 1 0 1 0 1 1 1 1 1 1 0
Possible keys for week 2:
  3 : 1 0 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0
  5 : 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 1 0 1 0 1 0 1 1 1 1 1 1 0
```

3.2 Beispiel 1

```
Solution keys at: [1, 2, 4, 6, 7, 9, 11, 14, 15]
Solution keys:
  1 : 0 0 1 0 0 0 0 0 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 0
  2 : 1 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 1
  4 : 0 0 1 1 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 1 0
  6 : 1 1 1 1 0 0 1 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 1 1 0
  7 : 0 0 1 1 0 1 1 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0
  9 : 1 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0 0 1 1 1 0
 11 : 0 0 1 0 0 0 1 1 1 0 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 1
 14 : 1 1 0 0 0 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 0 1 0 0
 15 : 0 0 0 1 0 0 0 1 1 1 0 1 0 0 1 1 0 0 0 1 1 1 1 1 1 0 1 1 0 0 1 0 0
Possible keys for week 2:
  2 : 0 0 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 0
  4 : 0 0 1 0 0 0 1 1 1 0 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 1
```

3.3 Beispiel 2

```
Solution keys at: [9, 20, 26, 53, 57, 71, 76, 78, 89, 95, 99]
Solution keys:
  9 : 0 1 1 0 1 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0
    ↪ 0 0 1 1 0 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1
    ↪ 1 0 0 1 1 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 0
    ↪ 0 1 1 1
 20 : 0 0 1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 0 0 0
    ↪ 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1
    ↪ 1 0 1 0 1 0 1 1 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 0 1
    ↪ 0 0 0 1
 26 : 1 0 1 0 1 0 1 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 1
    ↪ 1 0 0 1 1 1 0 0 1 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1
    ↪ 0 0 0 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0 1
    ↪ 1 1 1 0
 53 : 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1
    ↪ 0 1 0 1 0 1 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 0
    ↪ 0 1 0 1 0 1 0 0 0 1 0 1 1 1 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 1 1
    ↪ 1 0 1 1
 57 : 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 0
    ↪ 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 0 0 1
    ↪ 0 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 1 1 1
    ↪ 1 1 0 0
 71 : 1 1 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 0 1 0 0 1 0 1 1 0 1 0 1 0 1
    ↪ 1 0 0 1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 1 0 0 1
    ↪ 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0
    ↪ 0 0 0 0
 76 : 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 0 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0 0 1 1 1 1 1 0
    ↪ 0 0 0 1 0 1 0 1 0 0 1 1 0 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 1
    ↪ 0 1 0 0 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1
    ↪ 1 0 1 1
```

```
 78 : 1 1 0 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 1 1 1 1 0 0 1 1 0 0 0 0 1 1 1 0 1 0 0 1 1 0
    ↪  1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 0 0 1 0 0 1 1 0 1
    ↪  1 0 1 1 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 1 1 1 0 0 1 0 1 1 1
    ↪  1 1 1 1
 89 : 0 1 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0 0 1 0
    ↪  0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0
    ↪  0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0
    ↪  1 0 1 1
 95 : 0 1 1 1 0 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1 1 0 0 0 1 1 0 1 1 0 1 1 1 0 1 0 0 1
    ↪  0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0
    ↪  0 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1 0
    ↪  1 0 0 0
 99 : 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 0 1 1 1 1 0
    ↪  1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1
    ↪  1 0 1 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 0 0 0 1 0 1 1 1 0
    ↪  1 0 0 0
Possible keys for week 2:
 20 : 0 0 1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 0 0 0
    ↪  0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1
    ↪  1 0 1 0 1 0 1 1 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 0 1
    ↪  0 0 0 1
 26 : 0 1 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0 1 0
    ↪  0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0
    ↪  0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0
    ↪  1 0 1 1
```

3.4 Beispiel 3 und 4

Diese beiden Beispiele konnten leider auf meinem PC nicht ausgeführt werden, da zu wenig RAM-Speicherkapazität zur Verfügung steht. Mit mehr RAM-Kapazität könnte allerdings dann auch die in Abschnitt 1.4 beschriebene Lösungsidee genutzt werden und der Algorithmus würde insgesamt schneller laufen.

3.5 Beispiel 5

```
Solution keys at: [70, 77, 163, 167, 185]
Solution keys:
 70 : 1 0 0 0 0 1 0 0 1 1 1 0 1 0 1 0 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 1 0
    ↪  0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1
 77 : 1 1 0 1 0 1 0 0 0 1 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0
    ↪  0 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1
163 : 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 0 1 0 0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 1 1 0 1 0 0
    ↪  1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0 1 0 0
167 : 0 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 1 0 1 1
    ↪  0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0
185 : 1 0 1 0 0 0 0 1 1 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 1 1
    ↪  1 1 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 1 0
Possible keys for week 2:
 77 : 1 0 0 0 0 1 0 0 1 1 1 0 1 0 1 0 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 1 0
    ↪  0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1
163 : 1 0 1 0 0 0 0 1 1 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 1 1
    ↪  1 1 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 1 0
```

---

4 Quellcode

4.1 Main-Klasse

```java
import javax.swing.JFileChooser;
import java.io.FileNotFoundException;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        String currentFolder = System.getProperty("user.dir") + "\\beispieldaten";
        JFileChooser chooser = new JFileChooser(currentFolder);
```

```
        int response = chooser.showOpenDialog(null);
        if(response == 0){
            String absolutPath = chooser.getSelectedFile().getAbsolutePath();
            Bonusaufgabe.solve_problem(absolutPath);
        }
    }
}
```

../source/Main.java

4.2 Bonusaufgabe-Klasse

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

import java.io.PrintStream;

import java.util.ArrayList;
import java.util.List;

import java.util.Arrays;

public class Bonusaufgabe {
    // init vars for reading file
    private static int nOfCards;
    private static int nOfZaraCards;
    private static int nOfBits;
    private static long[][] cards;

    // init xorSum list
    private static final List<xorSumIdx> xorSums = new ArrayList<>();

    // init vars for combinations
    private static boolean saveXorSum = true;
    private static boolean finish = false;
    private static int c = 0;

    // init solution array
    private static short[] solution;

    // init console and file output
    private static PrintStream consoleOutput;
    private static PrintStream fileOutput;

    /**
     * This function starts the algorithm and calculates the right cards
     *
     * @param filename path of selected file : String
     */
    public static void solve_problem(String filename) throws FileNotFoundException {
        initOutput(filename);

        long start;
        long stop;

        //FIRST STEP//
        consoleOutput.println("### Read file....");
        start = System.currentTimeMillis();
        readFile(filename);
        stop = System.currentTimeMillis();
        consoleOutput.printf("### Finished! in %f seconds%n\n", ((double) stop - (double)
    ↪  start) / 1000);

        //SECOND STEP//
        consoleOutput.println("### Calculate xor_sums....");
        start = System.currentTimeMillis();
        initCombinations(nOfZaraCards / 2);
        stop = System.currentTimeMillis();
```

```
        consoleOutput.printf("###␣Finished!␣in␣%f␣seconds%n\n", ((double) stop - (double)
    ↪  start) / 1000);
58
        //THIRD STEP//
60      consoleOutput.println("###␣Sort␣Array....");
        start = System.currentTimeMillis();
62      //xorSums.sort(new xorSumIdxComparator());
        quickSort(xorSums, 0, xorSums.size() - 1);
64      stop = System.currentTimeMillis();
        consoleOutput.printf("###␣Finished!␣in␣%f␣seconds%n\n", ((double) stop - (double)
    ↪  start) / 1000);
66
        //FOURTH STEP//
68      saveXorSum = false;
        c = 0;
70      consoleOutput.println("###␣Search␣Array...");
        start = System.currentTimeMillis();
72      initCombinations((nOfZaraCards / 2) + 1);
        stop = System.currentTimeMillis();
74      consoleOutput.printf("###␣Finished!␣in␣%f␣seconds%n\n", ((double) stop - (double)
    ↪  start) / 1000);
76      //LAST STEP//
        printResult();
78
        partB();
80
    }
82
    /**
84   * This function initializes the Output to console and file
     *
86   * @param filename path of selected file : String
     */
88  private static void initOutput(String filename) throws FileNotFoundException {
        consoleOutput = System.out;
90      String file = filename.split("\\.")[0];
        String[] parts = file.split("\\\\");
92      file = parts[parts.length - 1];
        fileOutput = new PrintStream("./beispielausgaben/" + file + ".out");
94  }

96  /**
     * This function prints to the console and to the output file
98   *
     * @param message message that has to be printed : String
100  */
    private static void printBoth(String message) {
102     fileOutput.print(message);
        consoleOutput.print(message);
104 }

106 /**
     * This function reads the input file
108  *
     * @param filename path of selected file : String
110  */
    private static void readFile(String filename) throws FileNotFoundException {
112     File file = new File(filename);
        Scanner scanner = new Scanner(file);
114
        nOfCards = scanner.nextInt();
116     nOfZaraCards = scanner.nextInt();
        nOfBits = scanner.nextInt();
118
        cards = new long[nOfCards][2];
120
        for (int i = 0; i < nOfCards; i++) {
122         String s = scanner.next();
            cards[i][0] = Long.parseUnsignedLong(s.substring(0, (nOfBits / 2)), 2);
124         cards[i][1] = Long.parseUnsignedLong(s.substring((nOfBits / 2) + 1), 2);
        }
126
```

```
    }

    /**
     * This function starts the recursive Combination function
     *
     * @param r amount of numbers per combination : int
     */
    private static void initCombinations(int r) {
        short[] a = new short[nOfCards];
        for (short i = 0; i < nOfCards; i++) {
            a[i] = i;
        }
        combination(a, new short[r], 0, 0, r);
    }

    /**
     * This function works recursive and calculates the next combination
     *
     * @param list    array of numbers to choose from : short[]
     * @param current current combination array : short[]
     * @param start   next index after last number in combination : int
     * @param index   current index of number in combination : int
     * @param r       amount of numbers per combination : int
     */
    private static void combination(short[] list, short[] current, int start, int index,
    ↪ int r) {
        if (!finish) {
            if (index >= r) {
                if (saveXorSum)
                    calculateXorSumAndSave(current);
                else
                    calculateXorSumAndSearch(current);
                return;
            }
            for (int i = start; i < list.length && list.length - i >= r - index; i++) {
                current[index] = list[i];
                combination(list, current, i + 1, index + 1, r);
            }
        }
    }

    /**
     * This function calculates the xorSum of a combination and stores the indices and
    ↪ the value into an xorSumIdx Object
     *
     * @param list list of combination indices : short[]
     */
    private static void calculateXorSumAndSave(short[] list) {
        count();
        xorSumIdx xor = new xorSumIdx(new long[2], list.clone());
        for (int x : list) {
            xor.xorSum[0] = xor.xorSum[0] ^ cards[x][0];
            xor.xorSum[1] = xor.xorSum[1] ^ cards[x][1];
        }
        xorSums.add(xor);
    }

    /**
     * This function calculates the xorSum of a combination and search the calculated
    ↪ xorSum for the same value
     *
     * @param list list of combination indices : short[]
     */
    private static void calculateXorSumAndSearch(short[] list) {
        count();
        long[] xorSum = new long[2];
        for (int x : list) {
            xorSum[0] = xorSum[0] ^ cards[x][0];
            xorSum[1] = xorSum[1] ^ cards[x][1];
        }
        int res = binarySearch(xorSum, 0, xorSums.size() - 1);
        if (res != -1) {
            if (validSolution(list, xorSums.get(res).idx)) {
```

```
                finish = true;
198             }

200         }
        }
202
        /**
204      * This function uses binary search to search the xorSum list
         *
206      * @param x value to search for : long[]
         * @param l left limit of binary search : int
208      * @param r right limit of binary search : int
         * @return index of found value : int
210      */
        private static int binarySearch(long[] x, int l, int r) {
212         int mid = (l+r) / 2;
            if (r < l) {
214             return -1;
            }
216         if (xorSums.get(mid).xorSum[0] == x[0] && xorSums.get(mid).xorSum[1] == x[1]) {
                return mid;
218         } else if (xorSums.get(mid).xorSum[0] > x[0]) {
                return binarySearch(x, l, (mid - 1));
220         }
            return binarySearch(x, mid + 1, r);
222     }

224     /**
         * This function counts the number of combinations that are already calculated
226      */
        private static void count() {
228         c++;
            if (c % 1000000 == 0) {
230             consoleOutput.printf("%d␣combinations␣calculated!\n", c);
            }
232     }

234     /**
         * This function checks if a solution is valid (if indices are there only once)
236      *
         * @param list1 first half of indices : short[]
238      * @param list2 second half of indices : short[]
         * @return if solution is valid or not : boolean
240      */
        private static boolean validSolution(short[] list1, short[] list2) {
242         solution = new short[nOfZaraCards + 1];
            int i;
244         for (i = 0; i < list1.length; i++)
                solution[i] = list1[i];
246         for (int j = 0; j < list2.length; j++)
                solution[i + j] = list2[j];
248         for (int j = 0; j < solution.length - 1; j++) {
                if (solution[j] == solution[j + 1]) {
250                 return false;
                }
252         }
            return true;
254     }

256     /**
         * This function prints a card in binary
258      *
         * @param card card that has to be printed : long[]
260      * @param x    index of card : short
         */
262     private static void printOutCard(long[] card, short x) {
            String pre_output = String.format("%" + ((nOfBits / 2)) + "s", Long.
        ↪ toUnsignedString(card[0], 2)).replace('␣', '0') + String.format("%" + ((nOfBits /
        ↪ 2)) + "s\n", Long.toUnsignedString(card[1], 2)).replace('␣', '0');
264         StringBuilder output = new StringBuilder();
            for (int i = 0; i < pre_output.length(); i++) {
266             output.append('␣');
                output.append(pre_output.charAt(i));
```

```
268         }
            printBoth(String.format("%3d␣:", x));
270         printBoth(output.toString());
        }
272
        /**
274      * This function prints the calculated cards
         */
276     private static void printResult() {
            printBoth("Solution␣keys␣at:␣" + Arrays.toString(solution) + "\n");
278         printBoth("Solution␣keys:␣\n");
            for (short x : solution) {
280             printOutCard(cards[x], x);
            }
282     }

284     /**
         * This function solves partB (calculates possible cards for a week)
286      */
        private static void partB() {
288         long[][] solutionValues = new long[solution.length][2];
            for (int i = 0; i < solution.length; i++)
290             solutionValues[i] = cards[solution[i]];
            bubbleSort(solutionValues);
292         int answer = askQuestion(true);
            printBoth("Possible␣keys␣for␣week␣" + answer + ":␣\n");
294         answer--;
            printOutCard(solutionValues[answer], solution[answer]);
296         printOutCard(solutionValues[answer + 1], solution[answer + 1]);

298     }

300     /**
         * This function gets the users input which week has to be calculated
302      *
         * @param first true if function is called for the first time : boolean
304      * @return week that has to be calculated : int
         */
306     private static int askQuestion(boolean first) {
            Scanner scanner = new Scanner(System.in);
308         consoleOutput.print(first ? "Which␣week?␣:␣" : "Invalid␣week,␣value␣must␣be␣
        ↪ between␣1␣and␣" + nOfZaraCards + "?␣:␣");
            int answer = scanner.nextInt();
310         if (answer <= nOfZaraCards && answer > 0) {
                return answer;
312         } else {
                return askQuestion(false);
314         }
        }
316
        /**
318      * This function implements the bubbleSort algorithm to sort the solution values
         *
320      * @param list solution values : long[][]
         */
322     private static void bubbleSort(long[][] list) {
            long[] Cache;
324         for (int i = 0; i < list.length; i++) {
                for (int j = 0; j < list.length - 1 - i; j++) {
326                 if (Long.compareUnsigned(list[j][0], list[j + 1][0]) > 0) {
                        Cache = list[j + 1];
328                     list[j + 1] = list[j];
                        list[j] = Cache;
330                 }
                }
332         }
        }
334
        /**
336      * This function sorts a List of xorSumIdx Objects by using the quickSort algorithm
         * @param list list of xorSumIdx Objects : List<xorSumIdx>
338      * @param low low pointer of quickSort algorithm : int
         * @param high high pointer of quickSort algorithm : int
```

```
340      */
     private static void quickSort(List<xorSumIdx> list, int low, int high) {
342          if (low < high) {
                 int pivot = quickSortDivideList(list, low, high);
344              quickSort(list, pivot + 1, high);
                 quickSort(list, low, pivot - 1);
346          }
     }
348
     /**
350       * This function sorts a list by the pivot element and is used by the quickSort
     ↪ algorithm
          * @param list list of xorSumIdx Objects : List<xorSumIdx>
352       * @param low low pointer of quickSort algorithm : int
          * @param high high pointer of quickSort algorithm : int
354       * @return index of pivot element
          */
356  private static int quickSortDivideList(List<xorSumIdx> list, int low, int high) {
         long pivot = list.get(high).xorSum[0];
358      int lowCache = low;
         int highCache = high;
360      xorSumIdx Cache;
         while (low < high) {
362          while (high > lowCache && list.get(high).xorSum[0] >= pivot) {
                 high--;
364          }
             while (low < highCache && list.get(low).xorSum[0] < pivot) {
366              low++;
             }
368          if (low < high) {
                 Cache = list.get(low);
370              list.set(low, list.get(high));
                 list.set(high, Cache);
372          }
         }
374      if (list.get(low).xorSum[0] > pivot) {
             Cache = list.get(low);
376          list.set(low, list.get(highCache));
             list.set(highCache, Cache);
378      }
         return low;
380  }
}
```

../source/Bonusaufgabe.java

4.3 xorSumIdx-Klasse

```
1  public class xorSumIdx {
       public final long[] xorSum;
3      public final short[] idx;
       public xorSumIdx(long[] xorSum, short[] idx){
5          this.xorSum = xorSum;
           this.idx = idx;
7      }
   }
```

../source/xorSumIdx.java