Aufgabe 4: Zara Zackigs Rückkehr

Teilnahme-Id: 60660

Bearbeiter/-in dieser Aufgabe:
Raphael Gaedtke

13. April 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 Voraussetzungen

Definition 1. *Es sei n die Anzahl der Karten im gesamten Kartenstapel. Außerdem sei k die Anzahl der gesuchten Karten (inklusive der zusätzlichen Sicherungskarte) und b die Länge eines einzelnen Passworts.*

Definition 2. *Es seien x_1 und x_2 zwei nichtnegative, ganze Zahlen, deren Binärdarstellung (mit genau b Stellen) als Passwort auf einer Schlüsselkarte interpretiert werden kann. Dann bezeichne $x_1 \oplus x_2$ das bitweise exklusive Oder dieser beiden Zahlen bzw. Schlüsselkarten.*

Für die folgenden Lösungen soll explizit vorausgesetzt werden, dass sich die Schlüsselkarten auf eindeutige Weise aus dem Kartenstapel auswählen lassen.
In der Praxis ist dies nicht gesichert: Zaras Freunde könnten auch zehn beliebige Karten und eine weitere Karte mit deren exklusivem Oder untermischen. Da die auf den Karten abgespeicherten Passwörter verhältnismäßig lang sind, ist dieser Fall auch bei rein zufälligem Auswählen der Karten hinreichend unwahrscheinlich.
Um die Größenordnung dieser Wahrscheinlichkeit abschätzen zu können, wird die Wahrscheinlichkeit für

den Fall, dass die letzte Karte das exklusiver Oder von $(k-1)$ anderer Karten ist, angegeben. Das exklusive Oder von $(k-1)$ anderen Karten kann die letzte Karte in etwa $\binom{n-1}{k-1}$ Fällen sein, während sie insgesamt 2^b verschiedene Werte annehmen kann. Die gesuchte Wahrscheinlichkeit ist also $\frac{\binom{n-1}{k-1}}{2^b}$, was für die aus der Aufgabenstellung bekannten Werte $n = 111$, $k = 11$ und $b = 128$ einen Wert von ungefähr 10^{-25} hat.
Soll diese Möglichkeit trotzdem berücksichtigt werden, bieten alle der im Folgenden beschriebenen Algorithmen die Möglichkeit, sie dahingehend zu erweitern. Hierzu muss lediglich die Abbruchbedingung nach dem Finden einer einzelnen gültigen Lösung entfernt werden.
Außerdem wird davon ausgegangen, dass Karten mit gleichen Passwörtern ununterscheidbar sind - existieren zwei Karten mit demselben Bitmuster, so ist es unerheblich, welche dieser beiden Karten verwendet wird.
Die Karten in der Eingabe werden von 1 bis n und die Bits auf diesen Karten von 1 bis b durchnummeriert.

1.2 Eigenschaften einer gültigen Lösung

Eine gültige Lösung des Problems besteht in der Ausgabe von k Karten, von denen eine als Sicherungskarte das exklusive Oder aller anderen ist. Es sei x der Wert dieser Karte. x ist damit auch das exklusive Oder der $(k-1)$ anderen Karten. Das exklusive Oder aller k gesuchten Karten ist also $x \oplus x$.
Nach der Definition von $\oplus$ ist aber $x \oplus x = 0$, womit das exklusive Oder aller k Karten 0 ist. Umgekehrt hat eine beliebige, nichtleere Menge von Zahlen genau dann ein exklusives Oder von 0, wenn eine Karte das exklusive Oder aller anderen Karten ist (Es ist nämlich niemals $x_1 \oplus x_2 = 0$ für $x_1 \neq x_2$).
Um das Problem zu lösen, muss also eine Menge von k Schlüsselkarten aus dem Stapel ausgesucht werden, deren exklusives Oder 0 ist. In dieser neuen Formulierung des Problems ist es unerheblich, welche der Karten die Sicherungskarte ist.
Aus der Definition von $\oplus$ folgt aber auch, dass die Bedingung für eine Menge von k Karten genau dann erfüllt ist, wenn jedes Bit auf einer geraden Anzahl von Karten gesetzt ist. Mithilfe dieser Beobachtung lässt sich die Lösung des Problems als Lösung eines Gleichungssystems darstellen:

Definition 3. *Es sei $s_{i,m} \in \{0,1\}$ mit $1 \leq i \leq b$ und $1 \leq m \leq n$ eine Zahl, die genau dann 1 ist, wenn das i-te Bit der m-ten Karte gesetzt ist.*

Definition 4. *Es sei $v_m \in \{0,1\}$ mit $1 \leq m \leq n$ eine Zahl, die genau dann 1 ist, wenn die Karte mit der Nummer m zu den k gesuchten Karten gehört.*

Es wird dann das i-te Bit einer jeden Karte betrachtet. Wenn die Anzahl der Karten unter k gesuchten Karten, bei denen das i-te Bit gesetzt ist, gerade sein soll, entspricht dies genau der Aussage

$$\sum_{l=1}^{n} s_{i,l} \cdot v_l \equiv 0 \mod 2. \tag{1}$$

Diese Aussage lässt sich aber für jedes Bit treffen, woraus sich insgesamt b Gleichungen (modulo 2) ergeben. Jede dieser Gleichungen entspricht Gleichung 1 mit einem der b möglichen Werte für die Zahl i. Alle Werte $s_{i,l}$ sind bereits durch die Eingabe vorgegeben, eine Lösung des Gleichungssystems gibt also an, welche der Zahlen v_l auf 1 und welche auf 0 gesetzt werden müssen.
Eine gültige Lösung ist also eine Lösung dieses Gleichungssystems, in der genau k der Zahlen v_l auf 1 gesetzt sind. Im Folgenden wird der Einfachheit halber davon gesprochen, dass ein Index l auf 1 oder 0 gesetzt werde, wenn eigentlich die Zahl v_l gemeint ist.

Beispiel 1. *Es sei $n = 3$, $k = 2$ und $b = 3$. Im Kartenstapel gebe es also drei Karten, von denen Zara zwei auswählen muss, deren exklusives Oder 0 ist. Die drei Passwörter seien:*

1. *011*
2. *110*
3. *011*

Daraus ergibt sich folgendes Gleichungssystem:

$$I : 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{2}$$

$$II : 1 \cdot v_1 + 1 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{3}$$

$$III : 1 \cdot v_1 + 0 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{4}$$

Zara muss nun eine Lösung des Gleichungssystems finden, in der genau $k = 2$ der drei Zahlen v_1, v_2 und v_3 auf 1 gesetzt sind.

1.3 Allgemeine Lösung

Um das in Kapitel 1.2 beschriebene Gleichungssystem mit b Gleichungen zu lösen, wird das Gaußsche Eliminationsverfahren zum Lösen von linearen Gleichungssystem adaptiert (im Folgenden wird von „Koeffizienten“ gesprochen, wenn Variablen $s_{i,l}$ gemeint sind, die direkt aus der Eingabe folgen). Es werden zwei verschiedene Operationen angewandt, um das Gleichungssystem in eine für die Lösung günstigere Form zu bringen:

1. Die Gleichungen können ihre Reihenfolge zu tauschen. Dass sich die Lösungsmenge dabei nicht verändert, ist trivial.

2. Eine Gleichung kann zu einer anderen addiert werden (wobei die Koeffizienten nach der Addition durch ihren Zweierrest ersetzt werden). Von zwei Gleichungen des Gleichungssystems wird also eine durch die Summe der beiden ersetzt.
   Die Lösungsmenge des Gleichungssystems ändert sich dabei ebenfalls nicht, weil jede Lösung der ursprünglichen Gleichungen eine Lösung der resultierenden und jede Lösung der resultierenden eine Lösung der ursprünglichen Gleichungen sein muss.
   Da nur die Zweierreste der Koeffizienten betrachtet werden, ist es unnötig, andere Vielfache von Gleichungen zu anderen Gleichungen zu addieren.

Das Gleichungssystem soll mithilfe dieser beiden Operationen in eine Form gebracht werden, in der die i-te Gleichung für jeden möglichen Wert von i entweder der Tautologie

$$0 \equiv 0 \mod 2 \tag{5}$$

entspricht oder genau einen Koeffizienten $s_{i,l} = 1$ hat, für den es kein $x \neq l$ mit $s_{i,x} \neq 0$ gibt. Für die Gleichung gibt es also genau einen Index l, sodass der Koeffizient vor der Variablen v_l in dieser und nur in dieser Gleichung eine 1 ist (Im Kontext des Gaußschen Eliminierungsverfahrens entspräche diese Form der reduzierten Stufenform).
Die Indizes l, für die es einen solchen Koeffizienten gibt, heißen *bekannt*, alle anderen heißen *unbekannt*. Liegt das Gleichungssystem in der beschriebenen Form vor, folgen alle Werte von Variablen v_m mit bekanntem Index m aus den Variablen mit unbekanntem Index.
In einer Gleichung, in der v_m mit einem Koeffizient ungleich 0 vorkommt, haben dann nämlich nur noch unbekannte Variablen ebenfalls einen solchen Koeffizienten. Werden diese Indizes jeweils auf 1 oder 0 gesetzt, verbleibt v_m als einzige Unbekannte und folgt somit direkt aus der Gleichung selbst.
Um diese Form zu erreichen, wird in einer Zahl e zunächst festgehalten, wie viele Gleichungen schon in der gewünschten Form abgelegt wurden. Zu Beginn ist $e = 0$, zur besseren Übersicht werden die Gleichungen so vertauscht, dass die die ersten e Gleichungen die Gleichungen sind, die in e gezählt werden.
Solange $e < b$ ist, müssen noch Gleichungen untersucht und umgeformt werden. Dazu wird in einem Schritt der kleinstmögliche Kartenindex l gesucht, sodass der Koeffizient $s_{i,l}$ in mindestens einer Gleichung gesetzt ist, die nicht zu den ersten e gehört (Gibt es ein solches i nicht, sind alle noch nicht gezählte Gleichungen Tautologien wie in Gleichung 5).
Die gefundene Gleichung wird dann unter die ersten e getauscht und e um 1 erhöht. Zusätzlich wird die Gleichung zu allen Gleichungen addiert, in denen der Koeffizient $s_{i,l}$ noch gesetzt ist. Dann ist die betrachtete Gleichung die einzige, in der dieser Koeffizient gesetzt ist.
Sobald $e = b$ ist oder alle noch nicht bearbeiteten Gleichungen Tautologien sind, werden alle unbekannten Indizes in die Menge U geschrieben. Sind die Werte aller v_u mit $u \in U$ bekannt, lassen sich aus dem umgeformten Gleichungssystem alle v_u direkt bestimmen. Dabei muss angemerkt werden, dass die Menge U keine konstante Menge im mathematischen Sinne ist. In der vorliegenden Dokumentation wird mit einer „Menge“ vielmehr eine zustandsabhängige Datenstruktur bezeichnet, die so wie eine Menge Elemente enthält. In dieser Datenstruktur können dann neue Elemente hinzugefügt und enthaltene gelöscht werden.
Um die gesuchten Karten zu finden, werden alle $2^{|U|}$ möglichen Kombinationen der unbekannten Koeffizienten ausprobiert und die Lösung ausgewählt, bei der genau k Karten gezogen werden. Mit einer „Kombination“ von unbekannten Indizes sind dabei alle Möglichkeiten gemeint, die Werte der unbekannten Indizes aus der Menge $\{0, 1\}$ auszuwählen.

Beispiel 2. *Die Bearbeitung und Lösung des Gleichungssystems werden anhand von Beispiel 1 veranschaulicht. Zu Beginn ist das Gleichungssystem*

$$I : 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{6}$$

$$II : 1 \cdot v_1 + 1 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{7}$$

$$III : 1 \cdot v_1 + 0 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{8}$$

mit $e = 0$. Der kleinstmögliche Kartenindex l, der nicht unter den ersten $e = 0$ Gleichungen gesetzt ist, ist $l = 1$, der beispielsweise in Gleichung II gesetzt ist.
Also wird e auf 1 gesetzt und Gleichung II mit Gleichung I getauscht. Außerdem wird Gleichung II zu allen Gleichungen addiert, in denen der Index $l = 1$ gesetzt ist, was in diesem Beispiel auf Gleichung III zutrifft. Es ergiben sich drei neue Gleichungen:

$$I : 1 \cdot v_1 + 1 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{9}$$

$$II : 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{10}$$

$$III : 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{11}$$

(Die Nummerierung der Gleichungen ändert sich auch dann nicht, wenn Gleichungen vertauscht werden. Wird von einer Nummer gesprochen, ist damit die entsprechende Gleichung im jeweils letzten Schritt gemeint.)
Der kleinstmögliche Kartenindex l, der nicht unter den ersten $e = 1$ Gleichungen gesetzt ist, ist jetzt $l = 2$, der beispielsweise in Gleichung II gesetzt ist. Gleichung II muss nicht getauscht, dafür aber noch zu Gleichung I und Gleichung III addiert werden. Außerdem wird e auf 2 erhöht:

$$I : 1 \cdot v_1 + 0 \cdot v_2 + 1 \cdot v_3 \equiv 0 \mod 2 \tag{12}$$

$$II : 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{13}$$

$$III : 0 \cdot v_1 + 0 \cdot v_2 + 0 \cdot v_3 \equiv 0 \mod 2 \tag{14}$$

Unter den noch nicht bearbeiteten Gleichungen verbleibt nur noch die Tautologie in Gleichung III, womit das Gleichungssystem selbst nicht weiter bearbeitet wird. Es ergibt sich daraus $U = \{3\}$, weil die Indizes 1 und 2 in den obigen Gleichungen als bekannt markiert wurden.
Die möglichen Kombinationen von Indizes in U sind also $v_3 = 0$ und $v_3 = 1$. Für $v_3 = 0$ ergibt sich $(v_1, v_2, v_3) = (0, 0, 0)$ und aus $v_3 = 1$ genau $(v_1, v_2, v_3) = (1, 0, 1)$. In der zweiten Lösung sind genau k Indizes auf 1 gesetzt, woraus folgt, dass die gesuchten Karten die erste und die dritte sind.

1.4 Verbesserungen für Eingaben mit großem Kartenstapel

Mit dem in Kapitel 1.3 beschriebenen Algorithmus lassen sich Eingaben lösen, in denen U wenige Elemente enthält. Da hierfür im schlechtesten Fall aber $2^{|U|}$ Kombinationen von unbekannten Indizes ausprobiert werden müssen, ergibt sich aber eine exponentielle Laufzeit, die für weniger kleine Werte von $\mid U \mid$ zu groß ist.
Die Anzahl der auszuprobierenden Kombinationen kann allerdings verringert werden, wenn diejenigen ausgeschlossen werden, in denen die Werte von mehr als k unbekannten Indizes auf 1 gesetzt werden: Sind nämlich schon unter diesen mehr als k auf 1 gesetzt, sind in der Auswahl auch ohne Berücksichtigung der bekannten Indizes schon zu viele Karten enthalten.
Ausprobiert werden also zunächst alle Möglichkeiten, 0 unbekannte Indizes auf 1 zu setzen, dann einen, dann zwei usw. Insgesamt ergeben sich

$$\sum_{i=0}^{k} \binom{\mid U \mid}{i} \tag{15}$$

Kombinationen.

1.5 Eingaben mit wenigen Sicherungskarten

Hat k kleinere Werte, so lässt sich zur Lösung der Eingabedatei ein weiterer Algorithmus mit akzeptabler Laufzeit finden. Erklärt wird dieser am Beispiel $k = 5$, das aus der Eingabedatei stapel5.txt stammt. Die Lösungsidee lässt sich aber auch auf andere Werte von k anwenden.
Dieser Algorithmus nutzt die Beobachtung, dass zwei Teilmengen der k Schlüsselkarten das gleiche exklusive Oder haben müssen, weil das exklusive Oder dieser beiden exklusiven Oders nach Kapitel 1.2 den Wert 0 haben muss.
Die Schlüsselkarten werden dann von 1 bis 5 nummeriert in der Reihenfolge, in der sie im von 1 bis n nummerierten Kartenstapel aufgeführt sind. Dann wird die dritte dieser fünf Karten aus dem Kartenstapel ausgewählt (genauer wird jede der $n - 4$ Möglichkeiten für die dritte Karte ausprobiert, bis eine Lösung gefunden wurde). Der Index dieser dritten Karte sei im Folgenden d.
Dann müssen sich links von d zwei weitere Schlüsselkarten befinden. Es werden also alle exklusiven Oders von Kartenpaaren links von d gebildet und in eine Hashmap geschrieben, die es erlaubt, in konstanter Zeit abzufragen, ob eine bestimmte Zahl in dieser enthalten ist (siehe zu dieser Laufzeit auch die Dokumentation zu Aufgabe 2). Diese Hashmap ordnet außerdem jedes exklusive Oder zweier Karten links von d dem Paar von Kartenindizes zu, sodass eine Lösung aus diesem exklusiven Oder rekonstruiert werden kann.
Anschließend werden alle Kartenpaare rechts von d durchgegangen. Für jedes dieser Kartenpaare wird das exklusive Oder dieses Paares und der Karte d gebildet. Ist die resultierende Zahl ein Element der Hashmap für die Paare links von d, so wurde die Lösung gefunden.

1.6 Laufzeitüberlegungen

Es wird zuerst die Laufzeit des in Kapitel 1.3 und 1.4 beschriebenen und verbesserten Algorithmus betrachtet.
Um das Gleichungssystem umzuformen, werden maximal b Schritte ausgeführt. In jedem dieser Schritte wird zunächst der kleinste nocht nicht bekannte, aber gesetzte Index gesucht. Im gesamten Verlauf des Verfahrens werden dabei maximal nb Koeffizienten überprüft, sodass dieser Schritt insgesamt eine Laufzeit von $O(nb)$ hat.
Dann werden in jedem Schritt eventuell zwei Gleichungen vertauscht, was eine Laufzeit von $O(n)$ hat, und eine Gleichung zu anderen Gleichungen addiert. Die Laufzeit hierfür ist maximal $O(nb)$, weil dafür in maximal b Gleichungen maximal n Koeffizienten durchgegangen bzw. verändert werden müssen.
Für das Umformen des Gleichungssystems ergibt sich damit insgesamt eine Laufzeitkomplexität von $O(nb + b(n + nb))$ (das Überprüfen der Koeffizienten in $O(nb)$ und b Schritte, in denen jeweils eine Gleichung in $O(n)$ an den richtigen Platz getauscht und in $O(nb)$ zu anderen Gleichungen addiert wird). Durch Umformen und Kürzen aller konstanten Größen ergibt sich eine Gesamtlaufzeit von $O(nb^2)$.
Das Überprüfen einer bekannten Kombination von unbekannten Indizes hat eine Laufzeit von $O(nb)$, weil dafür alle n Koeffizienten von b Gleichungen durchgegangen werden müssen. Die Laufzeit des ersten Algorithmus insgesamt wird dann noch bestimmt durch die Anzahl a der Kombinationen von unbekannten Indizes, die durchprobiert werden müssen. Die Laufzeit des Überprüfens der Kombinationen ist dann $O(anb)$. Weil nb für die gegebenen Eingabegrößen aber verhältnismäßig kleine Werte annimmt, wird im Folgenden der Einfachheit halber immer eine Laufzeit von $O(a)$ angegeben.
Nach Kapitel 1.4 beträgt a ohne weitere Laufzeitverbesserungen $2^{|U|}$, was zu einer exponentiellen Laufzeit von $O(2^n)$ führt (Es muss offensichtlich $\mid U \mid< n$ sein.). Diese Laufzeit wird stark reduziert durch die Beobachtung, dass nur Kombinationen von unbekannten Indizes verwendet werden, in denen bis zu k dieser Indizes auf 1 gesetzt werden, weil dann nur noch $\sum_{i=0}^{k} \binom{|U|}{i}$ Kombinationen ausprobiert werden müssen. Diese Laufzeit lässt sich annähern durch $O(n^k)$ als obere Schranke.
Es zeigt sich, dass diese Zahl beispielsweise bei den Eingabedateien stapel3.txt und stapel4.txt zu groß für eine akzeptable Laufzeit ist (in diesen Eingabedateien gibt es deutlich mehr Variablen als Gleichungen, womit auch die Menge U größer wird). Bei diesen Eingabedateien lässt sich das Problem allerdings lösen, indem erst alle Kombinationen mit einem unbekannten Index, dann mit zwei usw. ausprobiert werden, weil unter den unbekannten Indizes bei diesen speziellen Eingaben nie mehr als 5 gesetzt werden müssen.
Dies ist allerdings nicht immer gegeben. Ist dies nicht der Fall, kann die Laufzeit des Verfahrens noch gesenkt werden, wenn das Gleichungssystem in mehreren Versuchen so umgeformt wird, das unterschiedliche Indizes in der Menge U stehen. Gibt es i disjunkte Möglichkeiten für die Menge U, können in der Menge mit der geringsten Anzahl der gesuchten Karten maximal $\frac{k}{i}$ stehen.
Unterschiedliche Mengen U können erreicht werden, indem die bekannten Indizes beim Umformen des Gleichungssystems in anderer Reihenfolge ausgesucht werden. Da aber nicht gesichert ist, dass es die

Möglichkeit disjunkter Mengen U gibt, ist die obere Grenze für die Laufzeit des Verfahrens weiterhin $O(n^k)$.

Der in Kapitel 1.5 beschriebene Algorithmus wird in etwa n Schritten ausgeführt. In jedem dieser Schritte werden alle Kombinationen von Karten links bzw. rechts von der Mittelkarte ausgeführt, wobei für jede dieser Kombinationen Operationen mit konstanter Laufzeit ablaufen. Links und rechts der Mittelkarte können jeweils maximal n Karten stehen, von denen $\frac{k}{2}$ in einer Kombination enthalten sein müssen. Es gibt also insgesamt $n^{\frac{k}{2}}$ solcher Kombinationen.
Die Laufzeit eines einzelnen Schritts ist demnach $O(n^{\frac{k}{2}})$ und die Laufzeit des Algorithmus insgesamt $O(n \cdot n^{\frac{k}{2}})$. Vereinfachend kann von einer Gesamtlaufzeit von $O(n^{\frac{k}{2}})$ ausgegangen werden.
Während die theoretische Laufzeitkomplexität des zweiten Algorithmus damit geringer ist als die des ersten, läuft der erste Algorithmus für viele Eingaben mit größeren Werten von k schneller, wenn die oben angesprochenen Verbesserungen ausgeführt werden.

1.7 Aufsperren des nächsten Hauses

Nach dem Finden der 11 Karten kann Zara das nächste Haus (mit der Nummer j) aufsperren, ohne dafür mehr als einen Fehlversuch zu benötigen.
Die gefundenen Karten sortiert sie dabei aufsteigend, so wie auch die Passwörter in aufsteigender Reihenfolge den Häusern zugeordnet sind. Es gibt dann zwei Möglichkeiten:

1. Das Passwort auf der Sicherheitskarte ist lexikografisch größer oder gleich dem Passwort w_j. In diesem Fall ist die j-te Karte in der Sortierung die Schlüsselkarte für das Haus j.
2. Das Passwort auf der Sicherheitskarte ist lexikografisch kleiner als w_j. In der Sortierung liegt die Sicherheitskarte noch vor der j-ten Schlüsselkarte, weswegen diese die $(j + 1)$-te Schlüsselkarte im Stapel ist.

Zara muss also nur die gefundenen Karten aufsteigend sortieren und die j-te und die $(j + 1)$-te Karte ausprobieren.

2 Umsetzung

Die Lösungsideen werden in C++ umgesetzt.
Dabei werden drei separate Programme umgesetzt, von denen eines auf Basis von Kapitel 1.3 die Eingabedateien stapel0.txt, stapel1.txt und stapel2.txt, eines auf Basis von Kapitel 1.4 die Eingabedateien stapel3.txt und stapel4.txt und das letzte mithilfe des in Kapitel 1.5 beschriebenen Algorithmus die Eingabedatei stapel5.txt bearbeiten.
Das zweite Programm kann alle Eingaben bearbeiten, die auch das erste bearbeiten kann, durch diese Umsetzung soll aber die Übersichtlichkeit der in Kapitel 1.4 beschriebenen Verbesserung erhöht werden.

2.1 Umsetzung in $O(2^n)$

Nach dem Einlesen der Eingabe wird das aus dieser folgenden Gleichungssystem in einem Container des Typs `vector<vector<short>>` gespeichert, wobei der Eintrag mit dem Indexpaar (i, j) den Koeffizienten $s_{i,j}$ enthält. Zu Beginn steht auf der rechten Seite einer jeden Gleichung die Zahl 0, weshalb diese Seite zunächst nicht gespeichert werden muss.
Umgeformt wird dieses Gleichungssystem in einer `while`-Schleife, die abgebrochen wird, wenn alle Gleichungen entweder eine Tautologie oder genau einen bekannten Index beinhalten. Innerhalb der Schleife wird der nächste bekannte Index gesucht und die dabei übersprungenen unbekannten Indizes im Container `unknown` gespeichert.
Danach wird die Gleichung mit dem neuen bekannten Index in zwei ineinander verschachtelten for-Schleifen zu anderen Gleichungen addiert, in denen dieser Koeffizient ebenfalls eine 1 ist. Schlussendlich werden noch alle Indizes, die auf den letzten bekannten Index folgen, in den Container `unknown` geschoben.
Um dann alle möglichen Kombinationen von unbekannten Indizes durchzugehen, wird eine solche Kombination zunächst als Teilmenge der unbekannten Indizes betrachtet, die alle dieser Indizes enthält, die auf 1 gesetzt werden.

Über diese lässt sich dann mithilfe von BitMagic iterieren: Eine solche Teilmenge wird als Zahl interpretiert, die in der Binärdarstellung im k-ten Bit auf 1 gesetzt ist, wenn die Teilmenge den k-ten unbekannten Index enthält. Alle Teilmengen lassen sich also Zahlen zwischen 0 und $2^{|U|}-1$ zuordnen, die nacheinander durchgegangen werden können.
Für eine spezifische Kombination von unbekannten Indizes werden die Werte der bekannten Indizes in zwei weiteren, ineinander verschachtelten `for`-Schleifen bestimmt. Gibt es eine Kombination, in der genau k der Zahlen auf 1 gesetzt wurden, wird diese ausgegeben und das Programm beendet.

2.2 Umsetzung in $O(n^k)$

Um die in Kapitel 1.4 beschriebenen Verbesserungen umzusetzen, wird das im letzten Kapitel beschriebene Programm abgewandelt, wobei das Einlesen, Speichern und Bearbeiten des Gleichungssystems unangetastet bleiben.
Das Bestimmen einer Lösung dieses Gleichungssystems bei bekannten Werten der unbekannten Indizes wird in die Funktion `solveEquations()` ausgelagert.
Dann müssen w der unbekannten Indizes auf 1 gesetzt und die Funktion `solve Equations()` für eine solche Kombination aufgerufen werden. Dafür werden in einer `for`-Schleife alle möglichen Werte von w in aufsteigender Reihenfolge durchgegangen. Dabei werden alle Möglichkeiten für eine solche Bestimmung in der rekursiven Funktion `seekSolution()` konstruiert, die dann für jede solche Kombination die Funktion `solveEquations()` aufruft.

2.3 Umsetzung in $O(n^{\frac{k}{2}})$

Um die in Kapitel 1.5 beschriebene Lösungsidee umzusetzen, müssen die Passwörter selbst gespeichert und häufig das exklusive Oder zweier solcher Karten abgefragt werden.
Die Passwörter werden als Strings mit den Zeichen „0“ und „1“ verwaltet und das exklusive Oder zweier solcher Strings in der Funktion `bitXOR()` ermittelt.
Dann können in einer for-Schleife alle Möglichkeiten für die dritte Karte (im Programm wird immer von $k = 5$ ausgegangen) durchgegangen werden. Für jede dieser Möglichkeiten wird zunächst über alle Paare von Karten links dieser Karte iteriert. Die exklusiven Oder dieser Karten werden in einem Container des Typs `unordered_map<string, pair<int, int>>`, der jedes exklusive Oder den Indizes des ursprünglichen Passwortpaares zuordnet und in konstanter Zeit abfragen lässt, ob ein bestimmtes exklusive Oder enthalten ist, gespeichert.
Für jedes Paar rechts von der dritten Karte kann dann überprüft werden, ob das richtige exklusive Oder in diesem Container ist. Wurde es gefunden, wird die Lösung ausgegeben und das Programm abgebrochen.

3 Beispiele

Die Zahl in der ersten Zeile gibt an, wie viele Karten insgesamt ausgegeben werden.

Lösung für die Eingabedatei stapel0.txt:

5
10111000001100111000010101011111 0
00111101010111000110100110011001
11111110001011010001000000110111
11010111111010111101101111110000
10101100111111011010100011100000

Lösung für die Eingabedatei stapel1.txt:

9
00010000111010011000011111101100100
0010000001111001111101111101111100
1101001101011011010100110101011 1
00110100000101010010000111101001 0
1111001110101100100100001011111 0
00110110000110101101010111111111010

11110111100100010100100001001110
00100011100111011010111011100011
11000111111010110100000101110100

Lösung für die Eingabedatei stapel2.txt:

11
1110111010101110011110111100011100110111101101010101111100011010
0011010001100000111101000010001100100000011101100010001011101000

0110101110100011011101000110000111000001100011010110001011101110
0110011011110111011100110101101111000011110111011101011111100111

0010101111100010101101011011110010011000000000001101001100111101
1001011001000010001101010110110010101110100100001011100011010001

1010101100000110110000010111111111001100011001110010101101111101
1000111111101111101000111111010000001011011011111111010011011110

1000000000010010011001100100011000000000001010101101001001000010
0111010110110101010010101000101110101100101000110010100100111011

0010100001100001001011101110101101101011100010010011010111101101
1110111100101100001001110010100001101001110001000100010011111100

1100001100010011011100010110010010110101011001101101010110100100
0011110100010001001010000110010101010010001100010001101110100000

1010111111001001001010011110110001001111100001010100110010000111
1000100010010010011010010101011111101011000001111110000000111011

1101111000010100110111110011000011101001101110111101011111011011
0111011010001001101101100111010001000011000001010111100101111111

0110100100101100010100111111101011000001000101100111010100101010
1011000100000001100001100011010110101011110110100000100101001011

0111011001111000111001111000110110111010010100000001000001011000
1010001110101000000011010011000011010010110110100101111101101000

Lösung für die Eingabedatei stapel3.txt:

11
0111011010011100100001101110010010101011111100101111000000101100
1101100101100111001011100000011010010011100111100101011010010111

0010000011100111000110101000111111001111000101111010011001010110
0010011000100111101011001111001111111011110110011111001010100001

1100000101100100011010011101111111101111101101101011111010001011
0000101111000111001010100101100000111011101011010101100111010100

1011011101001011111011001100010101110100001111110000100000110011
1111111001001100011100011110011111101000111010111000011110110101

1011100010111111000101111101010101011001100010000110110011000101
0110000000110000110111101010000110010001000110100011001001100110

1100101101011111111011101000100010010000010110011101010011111010
0000100111110001110001011000000000100111011001001110000011001111 0

0101000010110111001111000111001101001100111111100000100000010000
0010111100001110100001001001111101111001010011110110111110011011

1011000000100110011011010100010011001110010110011010111101111110
1000001110100011000001100000010101111111000000000101111010010001

0111110110001010110011001110101101010100110100011001111110101011
0000000111000110101001111111110100100001100010011111100010110100

1011111101010100110000010110110011110100001010001010010000100111
1010110100100101100011101100011010100000001101010100111010001011 1

0111000111111010100001001110001111111110011110110111000101010001
0110001100010101000010100010100001010000010100001000101110101100

Lösung für die Eingabedatei stapel4.txt:

11
1100010111000100100000101110100010011001100111101110100101101011
0100111000100001000100000011000010101111001001101101010111011101

1010101000001111111100111101111000100010011101000001001011110100
0001010001100010110011110111111010111000000001100011011111000011 0

1000110000101100110100101100011011010100110100001000010110101010
1100110110111111010110011001001101100100001011110000111010000111

1111001011000110001010011000100111000111101001110010001101010010
1001000100111100101000001000011101010011101000111001000000001111

1110001000000011110100111111100100110011101110100100011100111100
1111000010000101100001000011000011001011101101010000101111100001

0010110000011110111100001000000010110111111100001100111111111000
1111000000101111101100010010000010100001010111110110000101001010

0000011101101001010110111000111110100111001010001100000100000110
1110101111101001000100000000111101111110011010110100001000110111 10

0011011001011100100100111100111110101001110000010000000110001010
1000111001000100111000100110111111100100010010100100111011110100

0010110000111000111001111000010011000000000011101101100111010001
0100001010000110110010000111110011000100111101001100100010010000

0100001100100110101100111101110111101001011011101011111011011111
1001000100010110101100101111110000011000100111011000001101000111

1000000100010111001101010001100011010011010011110010001000100001
1011000001010111100110111011101110001011110000100100001110000101

Lösung für die Eingabedatei stapel5.txt:

5
1000010011101010001111100100110110011011100101010100010000001001

1101010001001101000111111110000110100010100111000100001001011011
1010111011001100100110001100110001011101001000000011011111100100
0101111111000111000000101111100010111010110101000100000011001000
1010000110101100101110111001100011011110111111010111000101111110

Es wird noch eine Eingabedatei stapel6.txt als eigenes Beispiel hinzugefügt, die die in Beispiel 1 beschriebene Situation enthält. Diese Eingabe soll deutlich machen, dass die vorgestellten Algorithmen auch mit identischen Schlüsselkarten umgehen können.

Die Eingabedatei stapel6.txt:

3 1 3
011
110
011

Lösung für die Eingabedatei stapel6.txt:

2
011
011

4 Quellcode

4.1 Umsetzung in $O(2^n)$

Die Variablen, in denen die Eingabe gespeichert wird:

```
int numcards; //Anzahl der Karten im Stapel
int wanted; //Anzahl der Karten, die aus dem Stapel ausgewaehlt werden sollen
//(inklusive Sicherungskarte)
int bits; //Laenge der Passwortkarten
ofstream fileOutput; //Ausgabedatei
vector<string> cards(0); //Speicher fuer die Karten in Form von Bitstrings
```

Der Container für das Gleichungssystem:

```
vector<vector<short>> equations(bits, vector<short> (numcards, 0));
//Container fuer die Koeffizienten des Gleichungssystems
//Bestimmen dieser Koeffizienten:
for(int i = 0; i < bits; i++)
{
    for(int j = 0; j < numcards; j++)
    {
        if(cards[j][i] == '1')
        {
            equations[i][j] = 1;
        }
    }
}
```

Das Bearbeiten des Gleichungssystems:

```
//Es folgt die Durchfuehrung der Gauss-Adaption
int settled = 0; //Anzahl der bereits an ihrem Ort festgelegten Gleichungen
int position = 0; //Nummer k der aktuell zu untersuchenden Variable v_k
vector<short> tautology(numcards, 0); //Koeffizienten einer wahren Gleichung 0 = 0
vector<int> unknown(0); //Indizes k, bei denen v_k nicht als erster gesetzter Koeffizient
//einer Gleichung auftauchen konnte, der aber gesetzt ist (entspricht der Menge U)

while(settled < (int) equations.size()) //Durchgehen der Variablen
{
    //Untersuchen, ob das aktuelle Bit in mindestems einem Passwort gesetzt ist:
    int index = -1; //Index einer Karte mit dem aktuellen, gesetzten Bit
```

```
    for(int i = settled; i < (int) equations.size(); i++)
    {
        if(equations[i][position] == 1)
        {
            index = i;
            break;
        }
    }

    if(index == -1)
    {
        //Das aktuelle Bit ist in keiner spaeteren Gleichung gesetzt
        //Wurde es in einer bereits gefundenen gesetzt?
        bool used = false;
        for(int i = 0; i < settled; i++)
        {
            if(equations[i][position] == 1)
            {
                used = true;
                break;
            }
        }
        if(used) //Es wurde schon einmal gesetzt
        {
            unknown.push_back(position);
        }

        position++;
    }
    else
    {
        //Das Bit ist in mindestens einem Passwort gesetzt
        swap(equations[settled], equations[index]); //Tauschen der Gleichung an die richtige Position
        //Addieren der Gleichungen:
        for(int i = 0; i < (int) equations.size(); i++)
        {
            if(i != settled && equations[i][position] == 1) //In der Gleichung ist das Bit gesetzt
            {
                for(int j = position; j < numcards; j++)
                {
                    if(equations[settled][j] == 1)
                    {
                        equations[i][j] = !equations[i][j];
                    }
                }
                if(equations[i] == tautology)
                {
                    //Die Gleichung enthaelt nur die wahre Aussage 0 = 0
                    equations.erase(equations.begin()+i);
                    i--;
                }
            }
        }
        settled++;
        position++;
    }
}
//Indizes, die groesser sind als die Anzahl der Gleichungen sind unbekannt:
while(position < numcards)
{
    unknown.push_back(position);
    position++;
}
```

Durchprobieren aller Kombinationen unbekannter Indizes:

```
//Der Container unkown enthaelt eine Liste der Variablen v_k, fuer die es am sinnvollsten ist,
//alle Werte durchzuprobieren.
int guess = unknown.size(); //Anzahl dieser Variablen
//Iterieren durch alle Moeglichkeiten fuer diese Variablen mit BitMagic:
for(long long i = 0; i < (1<<guess); i++)
{
```

```
    vector<int> res(equations.size(), 0); //Werte auf der rechten Seite der Gleichung
    vector<int> output(0); //Indizes der auszugebenden Karten

    for(int j = 0; j < guess; j++)
    {
        if((i & (1<<j)) != 0) //Durchgehen durch alle unbekannten Indizes
        {
            for(int a = 0; a < (int) equations.size(); a++) //Aktualisieren der Gleichungen
            {
                if(equations[a][unknown[j]] == 1)
                {
                    res[a] = !res[a];
                }
            }
            output.push_back(unknown[j]);
            //Einfuegen von unbekannten Indizes, die auf 1 gesetzt wurden
        }
    }

    //Auswaehlen unter den weiteren Karten:
    int position = 0; //Position des Indizes, der aktuell bestimmt wird
    for(int i = 0; i < (int) equations.size(); i++)
    {
        while(equations[i][position] == 0)
        {
            position++;
        }
        if(res[i] == 1) //Ein auf 1 gesetzter bekannter Index wurde gefunden
        {
            output.push_back(position);
        }
    }

    if((int) output.size() == wanted) //Die Loesung wurde gefunden!
    {
        //Ausgabe
    }
}
```

4.2 Umsetzung in $O(n^k)$

Viele Teile des Quelltextes dieser Programms gleichen denen aus dem vorherigen Kapitel. Diese Teile werden hier ausgelassen.
Das Untersuchen der Kombinationen von unbekannten Indizes:

```
//Der Container unkown enthaelt eine Liste der Variablen v_k, fuer die es am sinnvollsten ist,
//alle Werte durchzuprobieren.
int guess = unknown.size(); //Anzahl dieser Variablen
//Iterieren durch alle Moeglichkeiten, maximal vier(fuenf) dieser Variablen auf 1 zu setzen
//Dabei ist es unsinnig, alle diese Variablen auf 0 zu setzen, da dies die triviale Loesung
//ergibt, in der alle Variablen 0 sind
for(int i = 1; i < 11; i++)
{
    vector<int> output(0); //Indizes der auszugebenden Karten
    //Suchen einer Loesung
    seekSolution(wanted, equations, output, fileOutput, cards, unknown, guess, i, 0);
}
```

Implementierung der Funktion `seekSolution()`:

```
void seekSolution(int wanted, vector<vector<short>>& equations, vector<int> output,
    ofstream& fileOutput, vector<string>& cards, vector<int>& unknown, int guess,
    int remaining, int position)
{
    //Diese Funktion geht rekursiv alle Moeglichkeiten durch, eine Anzahl von Variabeln aus
    //dem Container unknown auf 1 zu setzen
    //Fuer jede dieser Moeglichkeiten wird die Funktion solveEquation aufgerufen
    //Im Container output werden die entsprechenden Indizes gespeichert
    //Die Variable remaining haelt fest, wie viele Indizes noch gewaehlt werden muessen,
```

```
    //die Variable position, ab welchem Index dies moeglich ist
    if(remaining == 0) //Alle Indizes wurden gewaehlt
    {
        solveEquations(wanted, equations, output, fileOutput, cards);
    }
    else
    {
        for(int i = position; i < guess; i++)
        {
            //Hinzufuegen eines weiteren Indizes:
            output.push_back(unknown[i]);
            seekSolution(wanted, equations, output, fileOutput, cards, unknown, guess,
            remaining-1, i+1);
            output.pop_back();
        }
    }
}
```

4.3 Umsetzung in $O(n^{\frac{k}{2}})$

Umsetzung der Funktion, die das exklusive Oder zweier Passwörter zurückliefert:

```
string bitXOR(string a, string b)
{
    //Diese Funktion liefert das exklusive Oder zweier Passwortkarten zurueck
    string output = "";
    for(int i = 0; i < (int) a.length(); i++) //a und b haben gleiche Laenge
    {
        if(a[i] == b[i])
        {
            output += "0";
        }
        else
        {
            output += "1";
        }
    }
    return output;
}
```

Das Finden der Lösung:

```
bool found = false; //Wurde die Loesung schon gefunden?
for(int split = 2; split < numcards-2 && !found; split++) //Index der dritten Karte
{
    unordered_map<string, pair<int, int>> pairXOR;
    //Speichert jeden XOR-Wert zweier Karten vor Split und die zwei Indizes der Karten
    for(int i = 0; i < split-1; i++)
    {
        for(int j = i+1; j < split; j++)
        {
            pairXOR[bitXOR(cards[i], cards[j])] = {i, j};
        }
    }

    //Durchgehen aller XOR-Paare auf der rechten Seite von split:
    for(int i = split+1; i < numcards-1 && !found; i++)
    {
        for(int j = i+1; j < numcards; j++)
        {
            if(pairXOR.count(bitXOR(bitXOR(cards[i], cards[j]), cards[split])) > 0)
            {
                //Eine Loesung wurde gefunden!
                found = true;
                //Ausgabe
                break;
            }
        }}}
```