Aufgabe 4: Zara Zackigs Zurückkehr

Teilnahme-ID: 60963

Tobias Hettler

24. April 2022

Inhaltsverzeichnis

1 Algorithmus 1

1.1 Lösungsidee

1.1.1 Repräsentation

Lemma 1.1. Die XOR-Operation ist äquivalent zur bitweisen Addition Modulo 2.

Beweis. Es gibt nur 4 Möglichkeiten, wie 2 Bits verechnet werden können, daher kann jede Möglichkeit einzeln überprüft werden.

Tabelle 1: Bitweise Addition Modulo 2 ist äquivalent zur XOR-Operation

A	**B**	**A + B mod 2**	**A ⊕ B**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

■

Da die XOR-Operation äquivalent zur bitweisen Addtition Modulo 2 ist, betrachte ich von nun an nur noch die bitweise Addition Module 2. Das hat den Vorteil, dass normale algebraische Regeln verwendet werden können.

Alle Karten haben b Bits. Somit kann man nun jede Karte als einen b-dimensionalen in einem b-dimensionalen Vektorraum auffassen. Des weiteren können die Vektoren nur die Werte 0 und 1 annehmen. Und da vorher die Äquivalenz von XOR zur bitweisen Addition Modulo 2 bewiesen wurde, kann als Vektorraum der Restring $\mathbb{Z}/2\mathbb{Z}$ benutzt werden.

Die Aufgabenstellung kann nun umformuliert werden. Es sei k die Anzahl an Karten, die Zara hat (inklusive der zusätzlichen Karte). Es wird nun also aus der Liste M an Vektoren eine Teilmenge $S \subset M$ mit $|S| = k-1$ gesucht, sodass diese als lineare Kombination einen weiteren Vektor $v \in M, v \notin S$ ergeben.

1.1.2 Berechnung

Definition 1.1 (msb Funktion)**.** Die Funktion $msb(v)$ für einen Vektor v gibt die erste Dimension an, in der der Vektor v eine 1 hat.

Um dies zu berechnen geht man folgendermaßen vor: Die Liste der Vektoren wird der Reihe nach durchgegangen. Bei jedem Vektor v_i wird überprüft, ob dieser als eine lineare Kombination aus den Vektoren $v_1 - v_{i-1}$ darstellbar ist. Ist das der Fall, wird noch überprüft, wie viele Vektoren benötigt wurden, um v_i darzustellen. Wenn $k-1$ Vektoren verwendet wurden, wurde eine Lösung gefunden.

Die Frage ist nun, wie man effizient berechnen kann, ob ein Vektor als eine lineare Kombination der bisher gesehenen Vektoren darstellbar ist.

Hierzu wird nach und nach eine Liste P an Vektoren angelegt, die letzendlich eine Basis für den Vektorraum bildet. Zudem ist jeder Vektor in P als eine lineare Kombination aus den Vektoren aus M darstellbar. Zudem ist die Liste P nach der $msb(v)$ Funktion aufsteigend sortiert. Der erste Vektor in P hat den kleinsten Wert für msb, der letzte den größten.

Es wird folgendermaßen vorgegangen: Anfänglich ist P leer. Der erste Vektor, der überprüft wird, wird einfach P hinzugefügt.

Nun seien zu einem bestimmten Zeitpunkt bereits 1 oder mehr Vektoren in P. Dann wird für den nächsten Vektor v überprüft, ob dieser mit den bis zu diesem Zeitpunkt betrachteten Vektoren darstellbar ist. Dazu werden die Vektoren in P der Reihe nach durchgegangen und es wird jeweils übeprüft, ob $msb(v) = msb(P_i)$.
Wenn $msb(v) = msb(P_i)$, wird v durch $v + P_i$ ersetzt und man macht mit dem neuen v beim nächsten Vektor in P weiter und überprüft wieder die Bedingung.
Wenn $msb(v) \neq msb(P_i)$, wird v nicht verändert und man macht mit unverändertem v bei dem nächten Vektor in P weiter.
Wenn während dieses Prozesses irgendwann $v = 0$ wird, werden die verbleibenden Vektoren in P übersprungen.

v_{res} sei der Vektor, der aus v geworden ist, nachdem man alle Vektoren in P durchgegangen ist.
Ist $v_{res} = 0$, so ist v durch eine lineare Kombination der Vektoren v_1 bis v_{i-1} aus M darstellbar.
Wenn $v_{res} \neq 0$, wird v_{res} P hinzugefügt. v_{res} wird so in P eingefügt, dass P wie oben beschrieben sortiert bleibt.

Man kann nun also effizient feststellen, ob ein Vektor mit den bisher betrachteten Vektoren darstellbar ist. Da nur eine Lösung gefunden wurde, wenn $k-1$ der originalen Vektoren verwendet wurde, muss jetzt noch überprüft werde, wie viele Vektoren für die Darstellung benötigt werden.

Hierzu wird für jeden Vektor in P gespeichert, aus welchen der ursprünglichen Vektoren aus M dieser aufgebaut ist. Wenn man nun einen Vektor überprüft, ob dieser darstellbar ist, speichert man sich jeweils die Vektoren, die einen Vektor P_i aufbauen, wenn v mit $v+P_i$ ersetzt wurde. So lässt sich diese Bedingung auch überprüfen.

1.2 Korrektheit

Es ist auf den ersten Blick nicht ganz klar, dass dieser Algorithmus auch funktioniert. Daher werde ich im folgenden die Korrektheit beweisen.

Lemma 1.2. Alle Vektoren in P sind durch lineare Kombinationen von Vektoren aus M darstellbar.

Beweis. Zunächst einmal betrachten wir den Fall $|P| = 1$. Dieser Fall tritt nur ein, wenn zuvor P leer war und dann der erste Vektor einfach P hinzugefügt wird. Da dieser Vektor sowieso in M ist, gilt die Bedingung für $|P| = 1$.

Es gibt also ein n mit $|P| = n$ bei dem die Bedingung gilt. Nun gilt es also zu zeigen, dass wenn alle bisherigen Vektoren in P die Bedingung erfüllen, auch der nächste Vektor, der hinzugefügt wird die Bedingung erfüllt.

Jeder Vektor, der P hinzugefügt wird, ist wie oben beschrieben eine lineare Kombination aus einem Vektor aus M und bis zu $|P|$ Vektoren aus P. Da jeder Vektor, der bis zu diesem Zeitpunkt P hinzugefügt wurde eine lineare Kombination aus Vektoren aus M ist, ist der neue Vektor auch eine lineare Kombination von Vektoren aus M. ■

Lemma 1.3. Das Ergebniss der Addition von zwei Vektoren im Vektorraum $\mathbb{Z}/2\mathbb{Z}$ ist nur 0, wenn beide Vektoren identisch sind.

Beweis. Da Vektoren Dimension für Dimension addiert werden, und jede Dimension nur zwei mögliche Zustände hat, kann man auch hier jede mögliche Kombination einzeln durchgehen.

Tabelle 2: Mögliche Ergebnisse bei der Addition von zwei Vektoren in einer Dimension

$v_1[x]$	$v_2[x]$	$v_1[x] + v_2[x]$
0	0	0
0	1	1
1	0	1
1	1	0

Aus der Tabelle ist ersichtlich, dass $A + B \mod 2$ nur 0 ist, wenn $A = B$ gilt. Damit ein Vektor 0 sein kann, muss jede einzelne Dimension 0 sein. Dies ist nur der Fall, wenn in jeder Dimension x $v_1[x] = v_2[x]$ gilt.

■

Lemma 1.4. Wenn v_{res} nach dieser Operation 0 ist, so ist v als eine lineare Kombination von Vektoren aus M darstellbar.

Beweis. Es sei L mit $L \subseteq P$ die Menge der Vektoren, die auf v addiert wurden, um v_{res} zu erhalten. Des weiteren sei l die lineare Kombination aller Vektoren in L.

Da $v + l = 0$ gilt, folgt aus Lemma 1.3, dass $v = l$ gilt.
Des weiteren gilt wegen Lemma 1.2 für alle Vektoren $p_i \in P$, dass p_i als lineare Kombination mit Vektoren aus M darstellbar ist. Daher ist l auch mit Vektoren aus M darstellbar und da $v = l$, gilt dasselbe auch für v. ■

Lemma 1.5. Jeder Vektor in P hat einen andereren Wert für die msb Funktion.
Formal: $\forall v \in P : \nexists u \in P \backslash \{v\} : msb(v) = msb(u)$.

Beweis. Zunächst der Fall $|P| = 1$. In diesem Fall gibt es nur einen Vektor und somit keinen anderen mit dem gleichen msb Wert. Die Bedingung ist erfüllt.

Nun gilt es zu zeigen, dass, wenn alle Vektoren in P die Bedingung erfüllen, der nächste Vektor der eingefügt wird diese auch erfüllt.
Wenn man die Vektoren in P durchgeht, um einen Vektor v zu überprüfen, gibt es jeweils 2 Fälle. Entweder $msb(v) = msb(P_i)$ oder $msb(v) \neq msb(P_i)$.

Zuerst der Fall $msb(v) = msb(P_i)$. In diesem Fall wird v durch $v + P_i$ ersetzt. Da sowohl v als auch P_i in der Dimension $msb(v) = msb(P_i)$ eine 1 haben, hat das Ergebnis $v + P_i$ hier eine 0. Somit gilt $msb(v + P_i) \neq msb(P_i)$. Zudem muss $msb(P_i + v) > msb(P_i)$ gelten, da $v + P_i$ in der Dimension $msb(P_i)$ eine 0 hat. Sollte es also einen anderen Vektor $P_j \in P$ geben, sodass $msb(P_j) = msb(v + P_i)$, ist dieser weiter hinten in P, da P nach den msb. Somit ist der neue Wert $msb(v + P_i)$ nicht gleich wie der msb Wert eines Vektors der vor P_i in der Liste ist. Die Bedingung ist also erfüllt.

Nun der Fall $msb(v) \neq msb(P_i)$. Hier wird v nicht verändert und es gilt $msb(v) \neq msb(P_i)$ also ist hier die Bedingung auch erfüllt.

■

Lemma 1.6. Die Vektoren in P sind linear unabhängig

Beweis. Um dies zu beweisen betrachtet man wieder die msb Werte der Vektoren. Nach Lemma 1.5 haben alle Vektoren in P einen unterschiedlichen msb Wert. Die Vektoren in P sind nach aufsteigendem msb Wert sortiert. Es sei P_i mit $P_i \in P$ ein beliebieger Vektor aus P. Zunächst reichen alle Vektoren mit Index $j > i$ nicht aus, um den Vektor P_i darzustellen, denn keiner dieser Vektoren hat an der Stelle $msb(P_i)$ eine 1.
Das heißt man bräuchte auf jeden Fall Vektoren mit Index $j < i$. Nimmt man einen beliebiegen Vektor P_j mit $j < i$, hat dieser an der stelle $msb(P_j)$ eine 1. Das heißt, wenn dieser Vektor mit anderen Vektoren P_i darstellen soll, braucht es einen weiteren Vektor, der an der Stelle $msb(P_j)$ auch eine 1 hat, um diese wieder auszugleichen. So einen Vektor gibt es jedoch nur mit Index $c < j$. Dies kann man so weiterführen, bis man bei P_1 ankommt. Kein anderer Vektor in P hat an $msb(P_1)$ eine 1. Somit kann P_1 nicht in einer linearen Kombination für P_i enthalten sein. Daher kann man also den Vektor P_i nicht als lineare Kombination von Vektoren aus P darstellen.

■

Lemma 1.7. Die lineare Hülle der Vektoren aus P ist identisch mit der linearen Hülle der Vektoren M_1 bis $M_{|P|}$.

Beweis. Zunächst sei O die Menge der Vektoren M_1 bis $M_{|P|}$. Des Weiteren ist $\langle O \rangle$ die lineare Hülle von O. Nach Lemma 1.2 gilt: $\forall P_i \in P : P_i \in \langle O \rangle$. Des Weiteren gilt sind wegen Lemma 1.6 alle Vektoren in P linear unabhängig. Nach dem Steinitz Exchange Lemma lässt sich nun jeder Vektor in O durch einen Vektor in P ersetzen, sodass $\langle O \rangle$ sich nicht verändert. Da $|O|=|P|$, lässt sich jeder Vektor in O ersetzen, bis O zur ursprünglichen Menge P geworden ist.

■

Theorem 1.8. Wenn ein Vektor v durch die bisher betrachteten Vektoren darstellbar ist, so ist $v_{res} = 0$.

Beweis. Nach Lemma 1.7 ist die Hülle der Vektoren in P die gleiche wie die der ersten $|P|$ Vektoren in M. Das heißt, wenn ein Vektor mit den ersten $|P|$ Vektoren aus M darstellbar ist, ist er auch mit den Vektoren in P darstellbar.

Wenn man überprüft, ob ein Vektor v darstellbar ist, geht man die Vektoren in P der Reihe nach durch. Nehmen wir man ist gerade bei P_i. Da $msb(P_i)$ eindeutig ist, gibt es keinen anderen Vektor in P, der nach P_i kommt und an der der Stelle $msb(P_i)$ eine 1 hat.
Nun gibt es zwei Möglichkeiten. Entweder v hat an der Stelle $msb(P_i)$ eine 1 oder eine v hat an dieser Stelle eine 0. Hat v an dieser Stelle eine 1, so muss v durch $v + P_i$ ersetzt werden, da es ansonsten keinen Weg mehr gibt v an der Stelle $msb(P_i)$ auf 0 zu bekommen. Hat v an der Stelle $msb(P_i)$ eine 0 darf v nicht durch die Kombination ersetzt werden, da ansonsten v an dieser Stelle eine 1 hätte, die durch keine der anderen folgenden Vektoren in P entfernt werden kann.

Da die lineare Hülle der Vektoren aus P identisch mit der linearen Hülle der ersten $|P|$ Vektoren aus M ist, muss dieser, wenn der Vektor v mit den ersten $|P|$ Vektoren aus M darstellbar ist, auch mit den Vektoren aus P darstellbar sein. So ist zu jedem Zeitpunkt eindeutig, ob v durch $v + P_i$ ersetzt werden soll oder nicht. Zu jeden Zeitpunkt kann nur eine der beiden Optionen, v durch $v + P_i$ ersetzen oder nicht, zu einer Lösung führen.
Somit muss zwangsläufig $v_{res} = 0$ sein, wenn v mit den Vektoren aus M darstellbar ist.

■

1.3 Problem

Dieser Ansatz hat jedoch noch ein Problem. Wenn auf die beschriebene Art und Weise festgestellt wird, ob ein Vektor mit den bisher betrachteten Vektoren darstellbar ist, kann es sein, dass man die Lösung nicht findet. Z.b. könnte ein Vektor sowohl mit $k - 1$ als auch mit einer anderen Anzahl an Vektoren darstellbar sein und es wird nur die zweite Variante gefunden.

Dies wird insbesondere zum Problem wenn es insgesamt mehr als d Vektoren gibt, wobei d die Anzahl der Dimensionen pro Vektor sind, also die Anzahl der Bits der Karten. Da der Vektorraum d Dimensionen hat, reichen d linear unabhängige Vektoren aus, um jeden möglichen Vektor darzustellen.

Damit also mit dem beschriebenen Algorithmus die Lösung gefunden werden kann, müssen $k - 1$ der originalen Vektoren unter den ersten d Vektoren sein. Die Wahrscheinlichkeit für dieses Ereignis E lässt sich folgendermaßen berechnen. Es sei n die Anzahl an Vektoren, also $n = |M|$:

$$P(E) = \frac{d}{n} \cdot \frac{d-1}{n-1} \cdot \frac{d-2}{n-2} \cdot \ldots \cdot \frac{d-k-2}{n-k-2} = \prod_{i=0}^{k-2} \frac{d-i}{n-i}$$

Sprich: der erste Vektor hat d mögliche Plätze von insgesamt n, der zweite nur noch $d-1$ von $n-1$ usw.

Berechnet man dies z.B. für das dritte Beispiel auf der BWINF Webseite mit $n = 161$ Karten und $d = 128$ Bits kommt man auf eine Wahrscheinlichkeit von ca. 7.3%. Dies ist natürlich zu wenig um sich darauf verlassen zu können.

1.4 Lösung des Problems

Um das Problem zu lösen, wird der oben beschriebene Prozess mehrmals mit einer jeweils zufälligen Reihenfolge der Vektoren ausgeführt. Dies ist deutlich effektiver als man auf den ersten Blick denkt.

Auch hierfür lässt sich die Wahrscheinlichkeit ausrechnen, dass das Verfahren erfolgreich ist. Es sei p die Wahrscheinlichkeit für einen Versuch. Also für das dritte Beispiel die oben berechneten 7.3%. Da es nur zwei mögliche Ergebnisse gibt ($k-1$ der originalen Vektoren sind unter den ersten d Vektoren oder nicht) und sich die Wahrscheinlichkeit p nicht ändert, handelt es sich beim mehrmaligem Ausführen, um eine Bernoulli-Kette.

Um die Wahrscheinlichkeit hierfür zu berechnen, geht man folgendermaßen vor. Zunächst sei n die Anzahl an Wiederholungen, die durchgeführt werden. Im folgenden bezeichne ich die Wahrscheinlichkeit, dass das Ergeinis E während den n Durchläufen mindestens einmal eintrifft mit $P(E_n)$. Um die Berechnung einfacher zu machen kann man das Gegenergeinis verwenden. $\overline{E_n}$: Während den n Durchläufen trifft das Ereignis E nie ein. Jetzt lässt sich die Wahrscheinlichkeit für $P(E_n)$ als $1 - P(\overline{E_n})$ berechnen.

$P(\overline{E_n})$ lässt sich als Binomialverteilung berechnen.

$$P(\overline{E_n}) = \binom{n}{0} p^0 \cdot (1-p)^{n-0} = (1-p)^n$$

Das heißt $P(E_n) = 1 - (1-p)^n$.

Wenn man nun die Wahrscheinlichkeit $P(E_n)$ für das dritte Beispiel ausrechnet mit beispielsweise $n = 30$, kommt man auf $P(E_{30}) \approx 0.89$. Jedoch braucht man in der Regel nicht so viele Durchläufe.

In der nachfolgenden Tabelle sieht man, wie viele Durchläufe der Algorithmus im Durschnitt für die Beispiele 0 - 5 braucht. Der Durchschnitt ist jeweils über 500 Ausführungen gebildet.

Tabelle 3: Durschnittliche Durchläufe über 500 Ausführungen

Beispiel	**Beispiel 0**	**Beispiel 1**	**Beispiel 2**	**Beispiel 3**	**Beispiel 4**	**Beispiel 5**
∅ Durchläufe	1	1	1	3.392	9.306	26.502
$\frac{x}{d}$	0.625	0.625	0.867	1.258	1.414	3.125

Beim 5. Beispiel werden deutlich mehr Durchläufe benötigt, da dort jede Karte nur 64 statt 128 Bits hat, d also nur 64 ist. Das sieht man auch am Verhältnis von x zu d. x ist hier die Anzahl der Karten. Beim Beispiel 5 ist $\frac{x}{d}$ am größten. Man sieht auch, dass wenn, wie bei den Beispielen 0 - 2, $x < d$ ist, immer nur ein Durchlauf benötigt wird.

1.5 Umsetzung

Das Programm habe ich in C++ umgesetzt.

Um die Karten bzw. Vektoren zu speichern habe ich die Klasse *DynamicBitset* implementiert. Dies ist eine allgemeine Implementation eines Bitsets mit im Grunde den gleichen Funktionalitäten wie das *bitset* aus der C++ Standardbibliothek. Allerdings muss man bei dem *std::bitset* die Anzahl der Bits zur Kompilierzeit wissen. Dies habe ich bei meiner Implementation verändert. Das Bitset besteht aus einem *std::vector* an *long longs*.

Die Implementation der Lösung befindet sich in der Klasse *Loeser* in der gleichnamigen Datei. Der Algorithmus ist in der Methode *findeKarten()* implementiert.

Die Liste P wird als eine *std::deque* implementiert. Wenn ein Vektor zu P hinzugefügt wird, wird dieser zunächst vorne eingefügt. Die *std::deque* ermöglicht dies in $\mathcal{O}(1)$. Danach wird einmal über die Liste iteriert. Beim Index i wird überprüft, ob die Vektoren an der Stelle i und $i+1$ relativ zueinander richtig sortiert sind. Ist das nicht der Fall, werden sie getauscht. Da der neu eingefügte Vektor der einzige an der falschen Position ist, ist P danach wieder sortiert.
Um zu speichern, aus welchen der originalen Vektoren aus M die Vektoren aus P aufgebaut sind, gibt es eine weitere *std::deque*, die für jeden Vektor in P ein *std::set* mit den originalen Vektoren speichert. Diese Liste wird immer gleich sortiert wie P, sodass die Indizes beider Listen jeweils übereinstimmen.

Die Funktion $msb(v)$ wurde in der Methode *berechneMSB* umgesetzt.

Addition Modulo 2 das gleiche ist wie die XOR Operation. Deshalb kann im Code, um zwei Vektoren zu addieren, das Exklusive Oder dieser gebildet werden.

2 Algorithmus 2: Meet in the middle

2.1 Lösungsidee

Wenn k, die Anzahl der originalen Karten (inklusive der Sicherungskarte), relativ klein ist, gibt es noch einen weiteren Weg die Lösung zu berechnen. Hierzu werden zunächst alle möglichen XOR Kombinationen mit $\lceil \frac{k}{2} \rceil$ Karten generiert und in einer Hashmap H, zusammen mit den verwendeten Karten, gespeichert.

Danach werden alle möglichen XOR Kombinationen mit $\lfloor \frac{k}{2} \rfloor$ Karten generiert. Bei jeder hier generiereten Kombination wird überprüft, ob diese in H ist. Wenn dies der Fall ist, hat man eine Lösung gefunden.

Da das exklusive Oder aller originalen Karten ohne die Sicherungskarte gleich dem Wert der Sicherungskarte ist, ist das exklusive Oder aller originalen Karten 0. Hat man also einmal $\lceil \frac{k}{2} \rceil$ Karten und einmal $\lfloor \frac{k}{2} \rfloor$ Karten, also insgesamt k Karten, deren Exklusives Oder gleich ist, ist das Exklusive Oder beider Kombinationen 0 und somit sind diese k Karten eine Lösung.

Hier muss nur noch aufgepasst werden, dass nicht eine Karte mehrmals verwendet wird. Wenn also im zweiten Schritt eine Menge G an Karten ausgewählt wird, deren XOR Kombination in H ist und H_i die Menge der in H zu dieser Kombination gespeicherten Karten ist, muss gelten $H_i \cap G = \emptyset$.

2.2 Umsetzung

Auch dieser Algorithmus ist in der Klasse *Loeser* implementiert.

Um die XOR Kombinationen zu generieren, wird die *std::prev_permutation()* Methode verwendet. Hierzu wird ein Binärstring mit Länge gleich der Anzahl an Karten erstellt. Die ersten $\lceil \frac{k}{2} \rceil$ beziehungsweise $\lfloor \frac{k}{2} \rfloor$ werden auf 1 gesetzt, der Rest auf 0.

Um die XOR Kombinationen zu speichern und zu repräsentieren, wird die bereits beschriebene *Dynamic-Bitset* Klasse verwendet. Um die Kombinationen zu speichern, wird eine *std::unordered_map* verwendet. Diese Map verwendet als Key die Bitsets. Der Wert, der gespeichert wird, sind alle bereits gefunden Möglichkeiten den Key als Exklusives Oder der Karten darzustellen.
Da die Bitsets als Keys für die Hashmap verwendet werden, wird eine Hash Funktion benötigt. Hierfür wird das Exklusive Oder aller *unsigned long longs* des Bitsets gebildet und als Hash verwendet.

Es gibt zwei separate Methoden *meetInTheMiddleGerade()* und *meetInTheMiddleUngerade()* für gerade k beziehungsweise ungerade k. Die Hilfsmethode *meetInTheMiddle()* ruft je nach k die entsprechende Methode auf.

Der Unterschied zwischen den beiden Methoden ist, dass in der Methode für ungerade k zwei while Schleifen benötigt werden. In der ersten werden alle Kombinationen aus $\lceil \frac{k}{2} \rceil$ Karten generiert und in der Hashmap gespeichert. In der zweiten Schleife werden alle Kombinationen aus $\lfloor \frac{k}{2} \rfloor$ Karten generiert und jeweils überprüft, ob diese in der Hashmap gespeichert sind.

Wenn k gerade ist, lässt sich auf eine der beiden Schleifen verzichten, da dann $\lceil \frac{k}{2} \rceil = \lfloor \frac{k}{2} \rfloor$ ist. In diesem Fall wird, wenn eine Kombination aus $\frac{k}{2}$ Karten generiert wurde, direkt überprüft, ob diese Kombination bereits in der Hashmap gespeichert ist und danach selber gespeichert. Zusätzlich wird natürlich noch überprüft, ob eine Karte mehrfach verwendet wurde. Auf diese Weise spart man sich die zweite Schleife.

3 Teilaufgabe b

Nachdem man den ersten oder zweiten Algorithmus ausgeführt hat, wurden die originalen Karten identifiziert. Nun ist die Frage, wie man mit diesen das $x-$te Haus aufschließen kann ohne mehr als zwei Fehlversuche zu benötigen.

Lemma 3.1. Aus den gefundenen Karten alleine lässt sich die Sicherungskarte nicht identifizieren

Beweis. Das einzige, was die Sicherungskarte ausmacht, ist, dass das Exklusive Oder aller originalen Karten die Sicherungskarte ergibt. Das gilt jedoch nicht nur für die Sicherungskarte, sondern für alle der gefundenen Karten. Gehen wir davon aus, dass Zara am Anfang n Karten hatte. Mit der Sicherungskarte s hat sie also $n+1$. Per Definition gilt für die Sicherungskarte: $s = k_1 \bigoplus k_2 \bigoplus k_3 \bigoplus \cdots \bigoplus k_n$. Das heißt,

bildet man das Exklusive Oder aller Karten außer der Sicherungskarte, erhält man die Sicherungskarte. Das gleiche passiert jedoch auch für jede andere Karte. Lässt man z.B. die 2-te Karte weg und bildet das Exklusive Oder aller anderen inklusive der Sicherungskarte erhält man:

$$k_1 \bigoplus k_3 \bigoplus \cdots \bigoplus k_n \bigoplus s$$

Jetzt kann man für s die Definition von oben einsetzen und erhält:

$$k_1 \bigoplus k_3 \bigoplus \cdots \bigoplus k_n \bigoplus k_1 \bigoplus k_2 \bigoplus \cdots \bigoplus k_n$$

Da XOR kommutativ ist, kann man die Karten, die in der Rechnung mehrmals vorkommen, nebeneinander schreiben. Da eine Zahl xor sich selber immer 0 ergibt, kürzen sich alle Karten, die zweimal vorkommen aus dem Term heraus. Im vorderen Teil kommt k_1 bis k_n ohne k_2 jeweils einmal vor. In s kommt k_1 bis k_n jeweils einmal vor. Das heißt k_2 kommt als einzige Karte nur einmal vor und der Term vereinfacht sich zu k_2. Das gilt nicht nur für die zweite Karte, statt der zweiten Karte lässt sich eine beliebige der n Karten wählen. Damit hat die Sicherungskarte nichts, was sie von den anderen unterscheidet und ist damit nicht identifizierbar.

■

Denoch lässt sich das x-te Haus mit weniger als zwei Fehlversuchen aufschließen. Da Zara ihre Schlüssel aufsteigend sortiert hatte, werden die gefundenen originalen Karten auch wieder aufsteigend sortiert. Jedoch stimmt die Reihenfolge danach noch nicht ganz, da die Sicherungskarte in den gefundenen Karten auch enthalten ist. Man hat jetzt also eine aufsteigend sortierte Liste an Karten, von denen eine die Sicherungskarte ist.

Wenn die Karte für das $x-$te Haus gesucht ist, müssen zwei Fälle unterschieden werden. Die Sicherungskarte habe Position i. Dann gilt entweder $i \leq x$ oder $i > x$. Wenn $i \leq x$, muss die Karte für das x-te Haus in der Liste an der Positon $x + 1$ sein. Wenn $i > x$, muss die Karte für das $x-$te Haus in der Liste an der Position x sein. Die Karte für das $x-$te Haus ist in der Liste also immer an Position x oder $x + 1$. Somit können die Karten an beiden Positionen ausprobiert werden und man benötigt maximal einen Fehlversuch, um die richtige Karte zu finden.

4 Komplexitätsanalyse

4.1 Algorithmus 1

4.1.1 Laufzeitanalyse

Zunächst werde ich die Zeitkomplexitäts des Algorithmus für einen Durchlauf beschreiben. Zunächst iteriert man über die Liste aller Vektoren M. Für jeden dieser Vektoren wird überprüft, ob dieser mit den bisher betrachteten darstellbar ist. Man geht also die Liste P durch. Wenn man den i-ten Vektor in M betrachtet ist $|P|$ maximal $i-1$. Das heißt für den i-ten Vektor muss man maximal $i-1$ Vektoren aus P durchgehen. Wenn man den Vektor mit einem Vektoren $p_j \in P$ verkleinern kann, muss noch gespeichert werden, aus welchen Vektoren das Ergebnis aufgebaut ist. Der Vektor p_j ist maximal aus j Vektoren aufgebaut. Im schlimmsten Fall geht man also für alle Vektoren $m_i \in M$ $i-1$ Vektoren in P durch und für jeden dieser $i-1$ Vektoren nochmal $i-1$. Das heißt $\mathcal{O}(n^3)$ lässt sich als Obergrenze für die Laufzeit feststellen.

Diese Obergrenze lässt sich jedoch noch etwas verfeinern. Es beschreibe die Funktion $f(n)$ die Laufzeit eines Durchlaufs mit $|M| = n$. Da für den i-ten Vektor in M nur jeweils maximal $(i-1) \cdot (i-1)$ andere Vektoren durchgegangen werden müssen gilt:

$$f(n) = \sum_{i=1}^{n} (i-1)^2 = \sum_{i=1}^{n} (i^2 - 2i + 1) = \sum_{i=1}^{n} (i^2) - \sum_{i=1}^{n} (2 \cdot i) + \sum_{i=1}^{n} 1$$

Die einzelnen Terme kann man nun noch jeweils in explizite Formeln umwandeln.

Lemma 4.1. Es gilt

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

für alle $n \geq 1$.

Beweis. Zunächst der Fall $i = 1$:
$\sum_1^1 (i^2) = 1.$
$\frac{1*(1+1)(2*1+1)}{6} = 1.$
Die Bedingung ist für $i = 1$ also erfüllt.

Da es ein n gibt, für das die Bedingung erfüllt ist, ist jetzt per Induktion zu zeigen, dass es dann auch für $n + 1$ gilt.

Es muss also gelten:

$$\frac{n(n+1)(2n+1)}{6} + (n+1)^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$$

Zunächst können beide Seiten mit 6 multipliziert werden, um die Brüche aufzulösen. Man erhält:

$$n(n+1)(2n+1)6(n+1)^2 = (n+1)(n+2)(2(n+1)+1)$$

Um die Gleichheit zu zeigen löse ich jetzt auf beiden Seiten die Klammern auf.

$$2n^3 + n^2 + 2n^2 + n + 6n^2 + 12n + 6 = 2n^3 + 3n^2 + 4n^2 + 6n + 2n^2 + 3n + 4n + 6$$

Vereinfacht ergibt sich:

$$2n^3 + 9n^2 + 13n + 6 = 2n^3 + 9n^2 + 13n + 6$$

■

Lemma 4.2. Es gilt

$$\sum_{i=1}^{n} (2 \cdot i) = n(n+1)$$

für alle $n \geq 1$.

Beweis. Für $i = 1$ ergibt sich $2 \cdot 1 = 2$ und $1 \cdot (1 + 1) = 2$. Die Formel stimmt also für $n = 1$.

Nun werden ich per Induktion zeigen, dass die Formel, wenn sie für n gilt auch für $n + 1$ und damit für alle Zahlen größer gleich 1 gilt. Es muss also die Formel für n plus $2(n + 1)$ gleich sein wie die Formel für $n + 1$. Sprich:

$$n(n+1) + 2(n+1) = (n+1)((n+1)+1)$$

Auflösen der Klammern ergibt:

$$n^2 + 3n + 2 = n^2 + 3n + 2$$

■

Das $\sum_{i=1}^{n} 1 = n$ gilt ist einfach zu sehen.

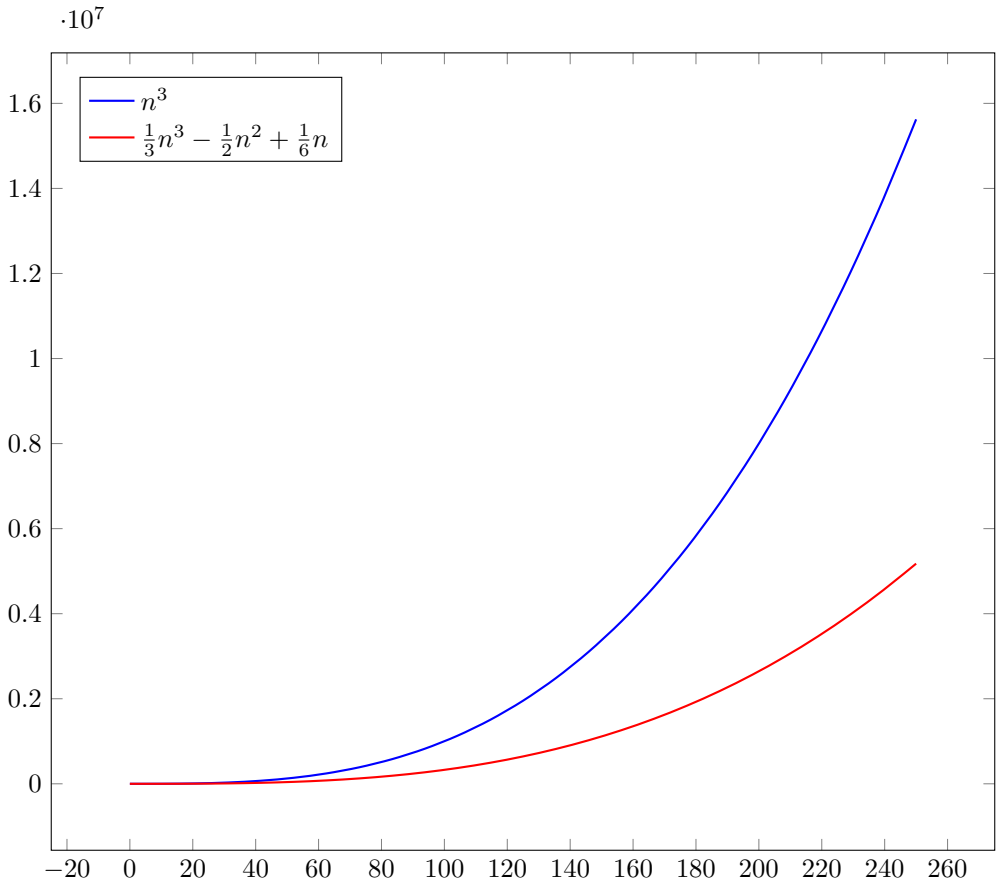
Nun kann $f(n)$ also folgendermaßen formuliert werden:

$$f(n) = \frac{n(n+1)(2n+1)}{6} - n(n+1) + n$$

Dies lässt sich noch weiter vereinfachen zu:

$$f(n) = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$$

Vergleicht man jetzt $f(n)$ mit der n^3 sieht man, dass die Obergrenze für die Laufzeit tatsächlich weniger als halb so groß wie n^3 ist. Wobei $f(n)$ immernoch der theoretisch schlechteste Fall ist.



Jedoch ist das nur die Laufzeit für einen Durchlauf. Normalerweise werden jedoch mehrere Durchläufe benötigt. Wie oben bereits gezeigt ist die Anzahl der Durchläufe in der Regel nicht sonderlich groß. Man könnte also sagen, dass die Laufzeit des kompletten Programms $\mathcal{O}(n^3)$ mit einem etwas größeren konstanten Faktor ist. Jedoch hängt dies auch stark von der Anzahl der Bits ab. Wenn die Anzahl der Bits im Verhältnis zu der Anzahl der Karten zu klein wird, steigt die Laufzeit stark an. Bei allen Beispielen auf der BWINF Webseite ist ist die durchschnittliche Laufzeit jedoch ungefähr $\mathcal{O}(n^3)$, da man hier selbst beim Beispiel 5 nur durchschnittlich 26,5 Durchläufe benötigt.

4.1.2 Speicherverbrauch

Um eine am Ende eine Lösung wieder konstruieren zu können, wird gespeichert aus welchen originalen Vektoren die Vektoren in P aufgebaut sind. Wie bereits beschrieben besteht der i-te Vektor in P aus maximal i Vektoren. Das heißt, wenn $|P| = u$ so werden maximal $1+2+3+\cdots+u$ Vektoren gespeichert. Man erhält die Summe der ersten u natürlichen Zahlen also $\frac{u^2+u}{2}$. Jeder dieser Vektoren ist b Bits groß. Somit ist der Speicherverbrauch $\mathcal{O}(u^2 \cdot b)$. Jedoch lässt sich dies noch ein wenig mehr eingrenzen.

Lemma 4.3. In P sind maximal b Vektoren.

Beweis. Nach Lemma 1.5 hat jeder Vektor in P einen anderen Wert der *msb* Funktion. Die *msb* Funktion kann insgesamt b verschiedene Werte haben. Nach dem Taubenschlagprinzip muss $|P| \leq b$ gelten. ■

Der Speicherverbrauch ist durch $\mathcal{O}(u^2 \cdot b)$ bergrenzt. Da $u = |P|$ und $|P| \leq b$ gilt, ist der Speicherverbrauch $\mathcal{O}(b^3)$. Jedoch ist der i-te Vektor in P nicht unbedingt aus wirklich i Vektoren aufgebaut, sondern oft auch weniger. Was den tatsächlichen Speicherverbrauch noch ein Wenig reduziert.

Zudem müssen natürlich auch die Karten an sich gespeichert werden, was bei n Karten, die jeweils b Bits groß sind, $\mathcal{O}(n \cdot b)$ Speicherplatz verbraucht.

4.2 Algorithmus 2

4.2.1 Laufzeitanalyse

Die Anzahl der Kombinationen der Größe $\frac{k}{2}$, aus insgesamt n Karten ist $\binom{n}{\frac{k}{2}}$.

Lemma 4.4. $\binom{n}{x} = \mathcal{O}(n^x)$

Beweis.

$$\binom{n}{x} = \prod_{i=1}^{x} \frac{n+1-i}{i}$$

Da immer $i \geq 1$ gilt, gilt für den Bruch immer $\frac{n+1-i}{i} \leq n$. Somit ist das Ergebnis auf jeden Fall nicht größer als x mal n mit sich selbst multipliziert, also n^x.

■

Da jede Kombination als Bitstring repräsentiert ist, muss dieser Bitstring jedesmal zuerst in die XOR Kombination der entsprechenden Karten umgewandelt werden. Da der String die Länge n hat, für jede Karte eine Stelle, dauert dies $\mathcal{O}(n)$. Das Hinzufügen zur Hashmap und das Überprüfen, ob eine Kombination in der Hashmap enthalten ist, ist jeweils in konstanter Zeit möglich. Um zu überprüfen, ob eine Karte doppelt verwendet wurde, muss man einmal die verwendeten k Karten durchgehen, da $k < n$ ist dies vernachlässigbar.

Für jede Kombination hat man also die Laufzeit $\mathcal{O}(n)$ bei insgesamt $\binom{n}{\lceil \frac{k}{2} \rceil} = \mathcal{O}(n^{\lceil \frac{k}{2} \rceil})$ Kombinationen. Zusammen also $\mathcal{O}(n^{\lceil \frac{k}{2} \rceil + 1})$. Der Algorithmus ist also nur für kleinere k geeignet.

Man kann noch die Laufzeit bei geraden und ungerade k unterscheiden. Da man bei geraden k nur eine Schleife benötigt, ist die Laufzeit genau die oben beschriebene. Bei ungeraden k benötigt man zwei Schleifen, die eine über $\binom{n}{\lceil \frac{k}{2} \rceil}$ Kombinationen, die andere über $\binom{n}{\lfloor \frac{k}{2} \rfloor}$ Kombinationen. Insgesamt also $\mathcal{O}(n^{\lceil \frac{k}{2} \rceil + 1} + n^{\lfloor \frac{k}{2} \rfloor + 1})$

4.2.2 Speicherverbrauch

Es werden jeweils $\binom{n}{\frac{k}{2}}$ Kombinationen gespeichert. Für jede dieser Kombinationen werden $\frac{k}{2}$ Integers gespeichert - die Indizes der entsprechenden Karten. Das heißt der Speicherverbrauch ist

$$\mathcal{O}(\binom{n}{\frac{k}{2}} \cdot \frac{k}{2}) = \mathcal{O}(n^{\frac{k}{2}} \cdot \frac{k}{2}) = \mathcal{O}(\sqrt{n^k} \cdot \frac{k}{2})$$

5 Beispiele

Ich habe das Programm auf Manjaro Linux 5.10 getestet. Das Programm nimmt 2 oder 3 Parameter über die Konsole. Zunächst die Beispieldatei und der Algorithmus (1 oder 2). Optional kann als drittes Argument ein Haus angegeben werden, für das der Schlüssel gesucht wird.

5.1 Beispiel 0

Hier rufe ich das Programm mit Algorithmus 1 auf. Zudem werden die beiden Karten ausgegeben, die für das 1. Haus möglich sind. Das Programm terminiert in ca. 0.003s.

```
./aufgabe4 stapel0.txt 1 1

Originale Karten (inklusive Sicherungskarte):
00111101010111000110100110011001
10101100111111011010100011100000
10111000011001110000101010111110
11010111111010111101101111110000
11111110001011010001000000110111

Fuer das 1-te Haus kommen folgende Karten in Frage:
00111101010111000110100110011001
10101100111111011010100011100000
```

5.2 Beispiel 1

Hier verwende ich den zweiten Algorithmus. Das Programm terminiert nach ca 0.02s.

```
./aufgabe4 stapel1.txt 2

Originale Karten (inklusive Sicherungskarte):
00110110000110101101011111111010
11110111100100010100100001001110
00100011100111011010111011100011
11000111111010110100000101110100
00010001110100110001111101100100
00100000111100111110111101111100
11010011010111011010011010110111
00110100001010100100001111010010
11110011101011001001000010111110
```

5.3 Beispiel 2

Das Programm terminiert nach ca. 0.04s.

```
./aufgabe4 stapel2.txt 1

Originale Karten (inklusive Sicherungskarte):
00101000011000010010111011101011011010111000100100110101111011011110
11110010110000100111001010000110100111000100010010011111100

00101011111000101011010110111110010011000000000001101001100111011001
01100100001000110101011011001010111010010000101110001101000 1

01101001001011000101001111111110101100000100010110011101010010101011
00010000000110000110001101011010101111011010000010010100 1011

01101011101000110111010001100001110000011000110101100010111011100110
01101111011101110011010110111100001111011101110101111110011 1

01110110011110001110011110001101101110100101000000100000101100001010
00111010100000000110100110000110100101101101001011111101101000

10000000000100100110011001000110000000000101010110100100100001000111
01011011010100101010001011101011001010001100101001001110 11

10101011000001101100000101111111110011000110011100101010111111011000
11111110111110100011111101000000101101101111111010011011110

10101111110010010010100111101100010011111000010101001100100001111000
10001001001001101001010101111110101100000111111000000000111011

11000011000100110111000101100100101101010110011011010101101001000011
11010001000100101000011001010101001000110001000110111010000 0

11011110000101001101111100110000111010011011101111010111110110110111
01101000100110110110011101000100001100000010101111001011111111
```

```
34 11101110101011100111101111000111001101111011010101011111000110100011
   01000110000011110100001000110010000001110110001000101110100 0
```

5.4 Beispiel 3

Bei den größeren Beispielen 3 - 5 variiert die Laufzeit des ersten Algorithmus ein wenig. Das liegt daran, dass mehrere Durchläufe benötigt werden und die Anzahl der benötigten Durchläufe aufgrund des involvierten Zufalls auch variiert. Die Zeit die ich angebe für die Dauer des Programms ist jeweils der Durchscnitt mehrerer Durchläufe.

Das Programm terminiert im Durchscnitt nach ca. 0.2s. mit Laufzeiten zwischen 0.04s und 0.6s.

```
1 ./aufgabe4 stapel3.txt 1 5

3 Originale Karten (inklusive Sicherungskarte):
  00100000111001110001101010001111110011110001011110100110010101100010
5 01100010011110101100111100111111101111011001111100101010000 1

7 01010000101101110011110001110011010011001111111000001000000100000010
  11110000111010000100100111110111001010011110110111110011011
9
  01110001111110101000010011100011111111100111101101110001010100010110
11 00110001010100001010001010000101000001010000100010111010 1100

13 01110110100111001000011011100100101010111111001011110000001011001101
   10010110011100101110000001101001001110011110010101101001 0111

15
   01111101100010101100110011101011010101001101000110011111101010110000
17 00011100011010100111111111101001000011000100111111000101101 00

19 10110000001001100110110101000100110011100101100110101111011111101000
   00111010001100000110000001010111111100000000001011110100100 01

21
   10110111010010111110110011000101011101000011111100001000001100111111
23 11100100110001110001111001111110100011101011100001111011010 1

25 10111000101111100010111110101010101100110001000011011001100010110110
   00000110000110111101010000110010001000110100011001001100 1100

27
   10111111010101001100000101101100111101000010100010100100001001111010
29 11010010010110001110110001101010000001101010100111010010 00111

31 11000001011001000110100111011111111011111011011010111110100010110000
   10111100011100101010010110000011101110101101010110011101 0100

33
   11001011010111111110111010001000100100000101100111010100111110100000
35 10011111000111000101100000000100111011001001110000011001 1110

37 Fuer das 5-te Haus kommen folgende Karten in Frage:
   01111101100010101100110011101011010101001101000110011111101010110000
39 00011100011010100111111111101001000011000100111111000101101 00

41 10110000001001100110110101000100110011100101100110101111011111101000
   00111010001100000110000001010111111100000000001011110100100 01
```

5.5 Beispiel 4

Das Programm terminiert durchschnittlich nach ca. 0.5s. mit Laufzeiten zwischen 0.2s und 1.5s.

```
   ./aufgabe4 stapel4.txt 1
2
   Originale Karten (inklusive Sicherungskarte):
4 00000111011010010101110111000111110100111001010001100000100000110111 01
   01111101001000100000001111011111100110101101000010001101111 0
6
```

```
00101100000111101111000010000001011011111110000110011111111110011110
00000101111101100010010000010100001010111110110000101001010

00101100001110001110011110000010011000000000011101101100111010001010000
01010000110110010000111110011000100111101001100100010010000

00110110010111001001001111001111101010011100000100000001100010101000 1
11001000100111000100110111111100100010010100100111011110100

01000011001001101011001111011101111010010110111010111110110111110010
00100010110101100101111110000011000100111011000001101000111

10000001000101110011010100011000110100110100111100100010000110110
00001010111100110111011101110001011110000100100001110000101

10001100001011001101001011000110110101001101000010000101101010101011001
10110111111010110011001001101100100001011110000111010000111

10101010000011111111001111011110001000100111010000010010111101000001 0
10001100010110011111011111101011100000001100011011110000110

11000101110001001000001011101000100110011001111011101001011010110100 1
11000100001000100000011000010101111001001101101010111011101

11100010000000111101001111111001001100111011101001000111001111001110 1
00010000101100001000011000011001011101101010000101111100001

11110010110001100010100110001001110001111010011100100011010100101001 0
00100111100101000001000011101010011101000111001000000001111
```

5.6 Beispiel 5

Mit dem ersten Algorithmus terminiert das Programm durchscnittlich nach ca. 0.6s mit Laufzeiten zwischen 0.04s und 1.3s. Mit dem zweiten Algorithmus terminiert das Programm nach ca. 1.4s.

```
./aufgabe4 stapel5.txt 2

Originale Karten (inklusive Sicherungskarte):
10101110110011001001100011001100010111010010000000110111111100100
01011111110001110000001011111000101110101101010000100000011001000
10100001101011001011101110011000110111101111110101110001011111110
10000100111010100011111001001101100110111001010101000010000001001
11010100010011010001111111100001101000101001110001000010011011011
```

6 Quellcode

6.1 Algorithmus 1 - findeKarten Methode

```
/**
 * Repraesentiert die Karten als Vektoren im Restring Z/2Z um eine Loesung zu finden
 * @param karten Vector aller Karten
 * @param anzahlOriginalKarten Wie viele Karten Zara urspruenglich hatte
 * @return Gibt einen Vector mit den originalen Karten zurueck
 */
std::vector<DynamicBitset> Loeser::findeKarten(std::vector<DynamicBitset> &karten,
                                               int anzahlOriginalKarten) {

    // Loesungs-Array
    std::vector<DynamicBitset> originaleKarten;
    originaleKarten.reserve(anzahlOriginalKarten + 1);
    const int maximaleIterationen = 100000;

    for(int i = 0; i < maximaleIterationen; ++i) {
        // Karten zufaellig sortieren
        std::shuffle(karten.begin(), karten.end(), std::mt19937{std::random_device{}()});
```

```
18
        // Liste der Basisvektoren (P aus der Dokumentation)
20      std::deque<DynamicBitset> basisVektoren;

22      // Speichert fuer jeden Basisvektor, welche urspruenglichen
        // Karten gebraucht werden, um diesen zu bilden
24      std::deque<std::set<DynamicBitset>> benoetigteVektoren;

26      for(auto & momentaneKarte : karten) {

28          // Speichert fuer die momentane Karte, welche Vektoren zur Darstellung
            // verwendet werden
30          std::set<DynamicBitset> verwendeteKarten;
            verwendeteKarten.emplace(momentaneKarte);
32
            auto karte = momentaneKarte;
34          for(int k = 0; k < basisVektoren.size(); ++k) {

36              // Ueberpruefe, ob msb(karte) = msb(basisVektoren[k])
                if(berechneMSB(karte) == berechneMSB(basisVektoren[k])) {
38
                    // Speichere aus welchen der originalen Vektoren der neue
40                  // Vektor aufgebaut ist
                    for(const auto& bSet : benoetigteVektoren[k]) {
42                      // Wenn der Vektor bereits in verwendeteKarten enthalten ist,
                        // wird dieser entfernt, ansonsten hinzugefuegt.
44                      unsigned long entfernt = verwendeteKarten.erase(bSet);
                        if(entfernt == 0) {
46                          // Ist noch nicht enthalten
                            verwendeteKarten.emplace(bSet);
48                      }
                    }
50                  DynamicBitset xorErgebnis = basisVektoren[k] ^ karte;
                    karte = xorErgebnis; // Aktualisiere Karte
52              }
            }
54
            if(!karte.istNull()) {
56              // Karte kann nicht mit den bisher bekannten Vektoren dargestellt werden
                basisVektoren.emplace_front(karte);
58              // Speichere Vektoren, die verwendet wurde, um die Karte zu verkleinern
                benoetigteVektoren.emplace_front(verwendeteKarten);
60
                // Halte sortiert
62              for(int j = 0; j < basisVektoren.size() - 1; ++j) {
                    if(berechneMSB(basisVektoren[j]) >
64                          berechneMSB(basisVektoren[j + 1])) {

66                      std::swap(basisVektoren[j], basisVektoren[j + 1]);
                        std::swap(benoetigteVektoren[j], benoetigteVektoren[j + 1]);
68                  }
                }
70
            } else {
72              // Karte kann mit den bisher bekannten Vektoren dargestellt werden
                if(verwendeteKarten.size() == anzahlOriginalKarten) {
74                  // Loesung gefunden

76                  for(const auto& verwendet : verwendeteKarten) {
                        originaleKarten.emplace_back(verwendet);
78                  }

80                  return originaleKarten;
                }
82          }
        }
84  }

86  return originaleKarten;
}
88
/**
90 * Berechnet die erste Dimension, in der der gegebene Vektor eine 1 hat
```

```
 * @param bitset Ein Vektor
 */
int berechneMSB(const DynamicBitset &bitset) {
    for(int i = 0; i < bitset.bits; ++i) {
        if(bitset.get(i)) return i;
    }
    return bitset.bits + 1;
}
```

6.2 Algorithmus 2

```

/**
 * Nutzt Meet in the Middle um eine Loesung zu brutforcen.
 * Dies ist eine Hilfsmethode, um die richtige Meet in the Middle Methode aufzurufen.
 * @param karten Vector aller Karten
 * @param anzahlOriginal Wie viele Karten Zara urspruenglich hatte
 * @param bits Anzahl der Bits pro Karte
 * @return Gibt einen Vector mit den originalen Karten zurueck
 */
std::vector<DynamicBitset> Loeser::meetInTheMiddle(
            const std::vector<DynamicBitset> &karten,
            int anzahlOriginal, int bits) {

    // Rufe entsprechende Methode auf, je nachdem, ob die Anzahl der
    // originalen Karten gerade oder ungerade ist.
    if(anzahlOriginal % 2 == 0) {
        return meetInTheMiddleGerade(karten, anzahlOriginal, bits);
    } else {
        return meetInTheMiddleUngerade(karten, anzahlOriginal, bits);
    }
}

/**
 * Nutzt Meet in the Middle um eine Loesung zu brutforcen. Diese Methode
 * sollte nur verwendet werden, wenn die Anzahl der Karten ungerade ist.
 * @param karten Vector aller Karten
 * @param anzahlOriginal Wie viele Karten Zara urspruenglich hatte
 * @param bits Anzahl der Bits pro Karte
 * @return Gibt einen Vector mit den originalen Karten zurueck
 */
std::vector<DynamicBitset> Loeser::meetInTheMiddleUngerade(
            const std::vector<DynamicBitset> &karten,
            int anzahlOriginal, int bits){

    // Speichert die finale Loesung
    std::vector<DynamicBitset> loesung;
    loesung.reserve(anzahlOriginal);

    // Speichert die gefundenen Kombinationen zusammen mit den Karten,
    // die fuer diese verwendet wurden
    std::unordered_map<DynamicBitset, std::vector<std::vector<int>>> m;

    // Baue Bitstring mit einer Ziffer fuer jede Karte.
    // Halb so viele Bits, wie es originale Karten gibt, sind auf 1 gesetzt.
    std::string s;
    int ceiledHalf = std::ceil((double) anzahlOriginal / (double) 2);
    for(int i = 0; i < ceiledHalf; ++i) {
        s += "1";
    }
    for(int i = ceiledHalf; i < karten.size(); ++i) {
        s += "0";
    }

    // Generiere alle Kombinationen mit anzahlOriginal/2 Karten
    do {
        DynamicBitset kombination(bits);
        std::vector<int> indizes;
        indizes.reserve(ceiledHalf);
        for(int i = 0; i < karten.size(); ++i) {
```

```
            if(s[i] == '1') {
                kombination ^= karten[i];
                // Speichere Indizes der verwendeten Karten
                indizes.emplace_back(i);
            }
        }

        // Speichere gefundene Kombination
        m[kombination].emplace_back(indizes);
    } while(std::prev_permutation(s.begin(), s.end()));

    int flooredHalf = std::floor((double) anzahlOriginal / (double) 2);

    s[ceiledHalf - 1] = '0';

    do {
        DynamicBitset kombination(bits);
        std::vector<int> indizes;
        indizes.reserve(flooredHalf);
        for(int i = 0; i < s.size(); ++i) {
            if(s[i] == '1') {
                kombination ^= karten[i];
                indizes.emplace_back(i);
            }
        }

        // Ueberpruefe, ob Kombination im ersten Durchgang auch schon gefunden wurde
        auto ind = m[kombination];
        if(!ind.empty()) {
            // Loesung gefunden
            int loesungsIndex = 0;
            bool loesungGefunden = false;
            for(const auto& vi : ind) {
                loesungGefunden = true;
                std::unordered_map<int, int> anzahl;
                anzahl.reserve(indizes.size() * 2);

                // Ueberpruefe, ob eine Karte mehrmals verwendet wurde
                for(int index : vi) {
                    anzahl[index]++;
                    if(anzahl[index] > 1) {
                        loesungGefunden = false;
                        break;
                    }
                }
                for(int index : indizes) {
                    anzahl[index]++;
                    if(anzahl[index] > 1) {
                        loesungGefunden = false;
                        break;
                    }
                }

                if(loesungGefunden) {
                    break;
                }
                loesungsIndex++;
            }

            // Es wurde eine Karte mehrmals verwendet
            if(!loesungGefunden) continue;

            // Eine Loesung wurde gefunden
            for(int index : ind[loesungsIndex]) {
                loesung.emplace_back(karten[index]);
            }
            for(int index : indizes) {
                loesung.emplace_back(karten[index]);
            }
            break;
        }

    } while(std::prev_permutation(s.begin(), s.end()));
```

```
133
135     return loesung;
}
137
/**
139  * Nutzt Meet in the Middle um eine Loesung zu brutforcen. Diese Methode
     * funktioniert nur, wenn die Anzahl der Karten gerade ist.
141  * @param karten Vector aller Karten
     * @param anzahlOriginal Wie viele Karten Zara urspruenglich hatte
143  * @param bits Anzahl der Bits pro Karte
     * @return Gibt einen Vector mit den originalen Karten zurueck
145  */
std::vector<DynamicBitset> Loeser::meetInTheMiddleGerade(
147             const std::vector<DynamicBitset> &karten,
             int anzahlOriginal, int bits) {
149
    // Speichert die finale Loesung
151     std::vector<DynamicBitset> loesung;
    loesung.reserve(anzahlOriginal);
153
    // Speichert die gefundenen Kombinationen zusammen mit den Karten,
155     // die fuer diese verwendet wurden
    std::unordered_map<DynamicBitset, std::vector<std::vector<int>>> map;
157
    // Baue Bitstring mit einer Ziffer fuer jede Karte.
159     // Halb so viele Bits, wie es originale Karten gibt, sind auf 1 gesetzt.
    std::string s;
161     for(int i = 0; i < anzahlOriginal / 2; ++i) {
        s += '1';
163     }
    for(int i = anzahlOriginal / 2; i < karten.size(); ++i) {
165         s += '0';
    }
167
    // Gehe alle Kombinationen durch
169     do {
        DynamicBitset kombination(bits);
171         std::vector<int> indices;
        indices.reserve(anzahlOriginal / 2);
173         for(int i = 0; i < karten.size(); ++i) {
            if(s[i] == '1') {
175                 kombination ^= karten[i];
                // Speichere, welche Karten verwendet wurden
177                 indices.emplace_back(i);
            }
179         }

181         // Suche in map, ob dieselbe Kombination bereits gefunden wurde
        auto it = map.find(kombination);
183         if(it != map.end()) {
            bool loesungGefunden = false;
185
            // Speichert den Index der passenden Kombination, wenn eine gefunden wird
187             int loesungsIndex = 0;

189             // Iteriere ueber alle bisher gefunden Wege die momentane Kombination
            // darzustellen
191             for(const auto& vi : it->second) {
                bool moeglich = true;
193
                // Ueberpruefe, ob eine Karte mehrmals verwendet wurde
195                 std::unordered_map<int, int> anzahl;
                anzahl.reserve(indices.size() * 2);
197                 for(int index : vi) {
                    anzahl[index]++;
199                 }
                for(int index : indices) {
201                     anzahl[index]++;
                    if(anzahl[index] > 1) {
203                         // Eine Karte wurde mehrmals verwendet
                        moeglich = false;
205                         break;
```

```
                    }
207                 }

209                 if(moeglich) {
                        loesungGefunden = true;
211                     break;
                    }
213                 loesungsIndex++;
                }
215
                if(!loesungGefunden) {
217                 // Keine Loesung
                    continue;
219             }

221             // Eine Loesung wurde gefunden
                for(int index : indices) {
223                 loesung.emplace_back(karten[index]);
                }
225             for(int index : it->second[loesungsIndex]) {
                    loesung.emplace_back(karten[index]);
227             }
                return loesung;
229         }
            // Speichere neu gefundene Kombination
231         map[kombination].emplace_back(indices);

233     } while(std::prev_permutation(s.begin(), s.end()));

235
    return loesung;
237 }
```