Aufgabe 4: Zara Zackigs Zurückkehr

Teilnahme-ID: 60809

Bearbeiter dieser Aufgabe:
Tobias Steinbrecher

24. April 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 Abstraktion der *Kontravalenz*

Zunächst soll der Zusammenhang zwischen den Öffnungskarten w_i (im Folgenden für Karten, mit welchen die Häuser entsperrt werden) und der Sicherungskarte s untersucht werden. Nach der Aufgabenstellung gilt dabei der Zusammenhang

$$s = w_1 \oplus w_2 \oplus ... \oplus w_k \tag{1}$$

wobei k der Anzahl an Häusern entspricht.
Um die Erstellung von s nachzuvollziehen, muss das *exklusive Oder* bzw. die *Kontravalenz* (im Folgenden *XOR*) betrachtet werden. Dafür eignet sich eine Wahrheitstafel des *XOR* (Verknüpfung der beiden Wahrheitswerte A und B über ein *XOR*):

A	B	$A \oplus B$
1	0	1
0	1	1
1	1	0
0	0	0

Tabelle 1: Wahrheitstafel *XOR*

Nach Definition gilt $A \oplus B = 1$, wenn genau eine der beiden Eingaben wahr ist. Für die *XOR*-Operation gilt zudem das Kommutativ- und für mehrere Wahrheitswerte ($A \oplus B \oplus C$) das Assoziativgesetz:

1. *Kommutativgesetz:* $A \oplus B = B \oplus A$
2. *Assoziativgesetz* $(A \oplus B) \oplus C = A \oplus (B \oplus C)$

Somit ist es irrelevant, in welcher Reihenfolge die *XOR*-Operation auf die Öffnungskarten w_i angewandt wird.
Für die Anwendung auf Bitfolgen ist das Prinzip eines *bitweisen Operators* einzuführen. Es folgt, dass das *XOR* unabhängig für jeden einzelnen Bit der beiden Eingabe-Bitfolgen angewendet wird, z.B.:

	01010
$\oplus$	10110
	11100

Tabelle 2: Beispiel für bitweises *XOR*

Bei Kombination der genannten Vorgehensweisen folgt für das Erstellen einer Sicherungkarte s mit $k = 4$ und einer Länge der Öffnungskarten von $m = 5$ Bits, beispielsweise:

	01010	w_1
$\oplus$	10000	w_2
$\oplus$	10010	w_3
$\oplus$	10001	w_4
	11001	s

Tabelle 3: Beispiel für die Erstellung einer Sicherungskarte

Über die Definition (Tabelle 1) und das *Assoziativgesetz* folgt, dass das Ergebnis der *XOR*-Operation verknüpft über eine *XOR*-Operation, mit einer vorherigen Eingabe, die andere vorherige Eingabe ausgibt:

$$A \oplus B = C \qquad \Rightarrow \qquad C \oplus A = B \qquad B \oplus C = A \tag{2}$$

In Bezug auf die Problemstellung, bedeutet das, wie bereits in der Aufgabenstellung angedeutet: „Mithilfe dieser Sicherungskarte kann sie jetzt im Notfall eine verlorene Karte rekonstruieren.“, dass

$$w_1 \oplus ... \oplus w_k = s \qquad s \oplus w_1 \oplus ... \oplus w_{i-1} \oplus w_{i+1} \oplus ... \oplus w_k = w_i \tag{3}$$

Es besteht somit kein Unterschied zwischen der Sicherungskarte s und einer beliebigen Öffnungskarte w_i bei der *XOR*-Operation. Vereinfacht sind somit nicht mehr k Öffnungskarten gesucht, welche verknüpft über die *XOR*-Operation das Bitmuster einer Sicherungskarte s ergeben, sondern $k+1$ Karten, wobei das Bitmuster jeder Karte c_i das *XOR* der andere k Karten enthält, wobei es für eine c_i unklar ist, ob es sich um die Sicherungskarte s oder eine Öffnungskarte w handelt.
Insgesamt lässt sich von der *XOR* -Operation eine gewisse *Paritätslogik* ableiten. Das *XOR* ist genau immer dann wahr, wenn eine ungerade Anzahl an wahren Werten eingegeben wird (siehe Definition Tabelle 1). Dies gilt aufgrund der Assoziativität, auch für mehrere Eingaben. Daraus lässt sich schließen, dass wenn das Ergebnis der *XOR*-Operation in Kombination mit den Eingabewerten betrachtet wird, immer eine gerade Anzahl an wahren Werten vorhanden ist. Dieser Zusammenhang lässt sich auch an Tabelle 3 erkennen. In jeder Spalte befindet sich eine gerade Anzahl von wahren Werten. Diese Mechanik des *XOR*-Operators wird beispielsweise auch bei Fehlerkorrektionsalgorithmen bei der Datenübertragung verwendet (z.B. Hamming-Code). Dadurch, dass sich somit in jeder Spalte eine gerade Anzahl an wahren Werten befindet, lässt sich erneut ableiten, dass das *XOR* aller $k+1$ Karten, 0 ergibt. Wie bereits beschrieben, ergibt eine Anwendung des *XOR*-Operators immer dann 1, wenn eine ungerade Anzahl an wahren Werten verknüpft wird $\Rightarrow$ Der *XOR*-Operator ergibt 0, wenn eine gerade Anzahl an wahren Werten verknüpft werden. Mit dieser Abstraktion, des *XOR*-Operators, lässt sich eine saubere Formulierung des Problems treffen (Unterabschnitt 1.2).

1.2 Formulierung des Problems

Gegeben:
Die Eingabe lässt sich als Menge A von $a \in \mathbb{N}$ beschreiben. Wobei $|A| = n$.
Gesucht:
Auszugeben ist nun $B \subseteq A$, wobei $|B| = k+1$ und

$$\bigoplus_{b \in B} b = 0 \tag{4}$$

Über den Sachzusammenhang ist dabei davon auszugehen, dass für alle Beispieleingaben eine solche Menge B existiert. In Abschnitt 4, werden jedoch auch jene Fälle untersucht, in welchen eine solche Menge B nicht existiert.
Es lässt sich eine Ähnlichkeit zum *k-XOR-Problem* bzw. *k-SUM-Problem* erkennen.

1.2.1 Uneindeutigkeit des Problems

Dadurch, dass die restlichen Karten $r \in A \backslash B$ vollständig randomisiert sind: „Sie erstellten weitere [...] Karten mit zufälligen Codewörtern und mischten sie mit Zaras [...] Karten in einem großen Stapel“, ist es möglich, dass ein weitere Menge $B' \neq B$ mit $|B'| = k$ und $B' \subseteq A$, wobei $\bigoplus_{b \in B'} b = 0$, existiert. Diese Annahme der möglichen Uneindeutigkeit der Lösung, lässt sich mit der Anzahl an möglichen Mengen, welche die Eigenschaften von B bzw. B' erfüllen, beweisen.
Für jeden Bit, gibt es $\lfloor \frac{k+1}{2} \rfloor + 1$ (Anzahl an geraden Zahlen bis $k+1$) Möglichkeiten für eine gerade Anzahl an wahren Werten. Für $k = 4$ (Anzahl der Öffnungskarten) gibt es beispielsweise die Möglichkeiten, dass es 0, 2 oder 4 wahre Werte in den 5 Bits ($k+1$) gibt, um insgesamt ein *XOR* von 0 zu erzeugen. Um diese wahren Werte in den $k+1$ Bits unterzubringen, gibt es $\binom{k+1}{\lfloor \frac{k+1}{2} \rfloor + 1}$ Möglichkeiten (Binomialkoeffizient). Da es m Bits gibt, folgt für die an möglichen Lösungen (Mengen mit den Eigenschaften von B):

$$P = m \cdot \binom{k+1}{\lfloor \frac{k+1}{2} \rfloor + 1} \tag{5}$$

P Möglichkeiten sind jedoch nur vorhanden, wenn auch alle möglichen Karten mit m Bits in A vorhanden sind. Dafür müsste die Anzahl an Karten n somit 2^m entsprechen, wobei alle Karten einzigartig sind.
Somit ist klar, dass es nicht immer nur eine Lösung B existiert, sondern es durch die Randomisierung der Freunde durchaus möglich ist, dass es auch mehrere Lösungen gibt.

1.3 Übertragung auf Lineare Algebra

Ein Ansatz um die beschriebene Problematik (Unterabschnitt 1.2) zu lösen, ist die Modellierung des Problems in der Linearen Algebra. Die Eingabe lässt sich nicht nur als eine Menge von Zahlen A fassen,

sondern auch als binäre Matrix C definieren.
Der *algebraische Körper*, für den C definiert entspricht dabei $(\mathbb{Z}_2, \oplus, \wedge)$. Für später wird dabei relevant, dass $\mathbb{Z}_2$ kein unendlicher Körper ist $(\mathbb{Z}_2 = \{0, 1\})$:

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{bmatrix} \quad \Rightarrow \quad C \in \mathbb{Z}_2^{n \times m} \tag{6}$$

Dabei entspricht n der Anzahl an Reihen bzw. Karten und m der Anzahl an Spalten bzw. Bits. Die *Verknüpfungen des Körpers*, welche hierbei *abelsche Gruppen* sind, entsprechen dabei $(\mathbb{Z}_2, +)$ und $(\mathbb{Z}_2 \backslash \{0\}, \cdot)$. Wobei die $\cdot$ Verknüpfung in $\mathbb{Z}_2$, äquivalent zu einer $\wedge$-Verknüpfung ist und die $+$ Verknüpfung mit dem *XOR* bzw. $\oplus$ einhergeht, da $a + b \equiv a \oplus b \mod 2$. Mit dieser Modellierung lassen sich verschiedene Gesetze und Vorgehensweisen aus der Linearen Algebra anwenden, welche somit ein Werkzeug zum Lösen dieses Problems darstellt.
Gesucht sind nun $k + 1$-Reihen c_{i*} für welche das *XOR* in jeder Spalte 0 ergibt.

1.3.1 Aufstellen eines Gleichungssystems

Wie bereits geschildert, ist folgende Matrix gegeben:

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{bmatrix} \tag{7}$$

Eine bestimmte Auswahl $\vec{x}$ der Karten bzw. Reihen c_{*i}, soll verknüpft mit dem *XOR*-Operator, 0 ergeben. Es folgt:

$$C^T = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix} \quad \Rightarrow \quad C^T \cdot \vec{x} = \vec{0} \tag{8}$$

wobei $k + 1$ zunächst ignoriert wird. Die transponierte Matrix C^T wird dabei für eine sinngemäße Multiplikation mit der Auswahl an Karten $\vec{x}$ nötig. Das Ergebnis soll dabei dem Nullvektor $\vec{0}$ entsprechen. Dabei gilt $\vec{x} \in \mathbb{Z}_2^m$ und $\vec{0} \in \mathbb{Z}_2^n$.
Der Zusammenhang lässt sich auch auf die Geometrie übertragen. Es sind $k+1$-Vektoren $\vec{v}_i \in \mathbb{Z}_2^m$ gesucht, für welche gilt, dass eine Addition (bzw. *XOR* wegen $\mathbb{Z}_2$), zum Ursprung führt und somit den Nullvektor ergibt.
An Gleichung 8 lässt sich die Form eines linearen Gleichungssystems (LGS) erkennen: $A \cdot \vec{x} = \vec{b}$. Der Lösungsraum $L = \{\vec{x} \mid C^T \cdot \vec{x} = \vec{0}\}$, entspricht dabei wie angedeutet, der Auswahl an Karten, wobei für das Element $x_i \in \vec{x}$, gilt: wenn $x_i = 1$, dann ist die Karte $c_{i*} \in C$ für eine Lösung zu wählen.
Die einzelnen Gleichungen des LGS lassen sich formulieren:

$$\begin{aligned} c_{11}x_1 + c_{12}x_2 + \cdots + c_{1m}x_n &= 0 \\ c_{21}x_1 + c_{22}x_2 + \cdots + c_{2m}x_n &= 0 \\ &\vdots \\ c_{m1}x_1 + c_{m2}x_2 + \cdots + c_{mn}x_n &= 0 \end{aligned} \tag{9}$$

Die Annahme, dass die Lösung dieses LGS eine Lösung für das *Zara Zackings Zurückkehr (ZZZ)*-Problem liefert, lässt sich damit begründen, dass im Sachverhalt durch jedes Bit, eine weitere Bedingung für die $k + 1$ Karten entsteht (Spalte muss 0 ergeben). Diese Bedingungen (Spalten), lassen sich wie bereits im LGS sichtbar, als einzelne Gleichungen darstellen. In jeder Gleichung wird jede Karte $c_{*i} \in C^T$, mit der zugehörigen Variablen x_i multipliziert. Durch das Lösen des LGS, werden Werte für x_i herausgefunden, sodass alle Gleichungen aufgehen, insofern dies möglich ist. Dabei gilt ebenfalls $x_i \in \mathbb{Z}_2$. Wenn somit Werte für x_i herausgefunden wurden, können diese, wie bereits erläutert, zurück auf den Sachverhalt bezogen werden ($x_i = 1$ entspricht der Auswahl der Karte i).

Da $k+1$ bislang unberücksichtigt blieb, beinhaltet die Lösungsmenge L möglicherweise auch Lösungen mit $\sum_{i=0}^{k} x_i \neq k$. Es werden alle Lösungen des LGS gefunden. Auch im Rahmen der Uneindeutigkeit, werden beinhaltet die Lösungsmenge möglicherweise auch mehrere Lösungen, welche aus $k+1$ Karten bestehen (Unterunterabschnitt 1.2.1).

1.3.2 Lösung mithilfe des Gaussschen-Eliminationsverfahrens

Zur besseren Visualisierung eignet sich die Schreibweise einer erweiterten Matrix $(A\vec{x} \mid \vec{b})$:

$$\left(\begin{array}{cccc|c} x_1 & x_2 & \cdots & x_n & b \\ c_{11} & c_{12} & \cdots & c_{1n} & 0 \\ c_{21} & c_{22} & \cdots & c_{2n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} & 0 \end{array}\right) \tag{10}$$

Zum Lösen eines solchen LGS eignet sich das *gaußsche Eliminationsverfahren (GE)*. Das *GE* lässt sich in verschiedene Schritte gliedern:

1. Finden der ersten Reihe $c_{i*} \in C^T$ mit $c_{i1} = 1$ (Wert in der ersten Spalte gleich 1). Sollte es keine Reihe mit dieser Eigenschaft geben, folgt das gleiche mit $c_{i2} = 1$.
2. Addition (bzw. *XOR*) von c_{i*} auf alle anderen Reihen $c_{j*} \in C^T$ mit $c_{j*} \neq c_{i*}$ und $c_{j1} = 1$. Aufgrund der $\oplus$ Verknüpfung gilt nun, dass $\forall c_{j*} \in C^T : c_{j1} = 0$.
3. Wiederholung der ersten beiden Schritte, für alle weiteren Spalten.

Somit kann mit jeder Reihe, in welcher für mindestens einen Koeffizienten $c_{ij} = 1$ gilt, eine Spalte *eliminiert* (bei allen anderen Reihen auf 0 gesetzt) werden. Dabei ist jedoch zu beachten, dass Fälle eintreten, in welchen aufgrund einer ungünstigen Belegung, Reihen vollständig eliminiert werden. Dies kann beispielsweise eintreten, wenn zwei Reihen vollständig identisch sind. Ein weiteres Beispiel ist der Fall:

$$\left(\begin{array}{cccc|c} x_1 & x_2 & x_3 & x_4 & b \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{array}\right) \tag{11}$$

Hierbei würde durch Addition der ersten Reihe, in der dritten Reihe 0011 entstehen, was dafür sorgt, dass Reihen 2 und 3 gleich sind. Folgend würde Reihe 3 von Reihe 2 vollständig eliminiert werden (0-Reihe). Anschließend müssen die Fallunterscheidungen, nach dem Lösen des LGS mit dem *GE*, auf den Sachverhalt (*ZZZ*) bezogen werden.

1.3.3 Überdeterminierte Systeme & Unterdeterminierte Systeme

Ein *überdeterminiertes System* lässt sich daran erkennen, dass es mehr Gleichungen als Variablen gibt. Ein *unterdeterminiertes System* ist dabei durch weniger Gleichungen als Variablen gegeben. Für den Sachzusammenhang handelt es sich um eine *überdeterminiertes System*, wenn $m > n$. Für $m < n$ ist das *System unterdeterminiert*. Auffällig dabei ist, dass es sich bei den *einfachen* Beispielen (`stapel0.txt-stapel2.txt`) von der BWINF-Webseite um *überdeterminierte Systeme*, und bei den *schwierigen* Beispielen, um *unterdeterminierte Systeme* handelt.
Das liegt daran, dass insofern nicht sehr viele Reihen, wie in Gleichung 11 demonstriert eliminiert werden, ein *überdeterminiertes System*, dadurch, dass mehr Gleichungen erfüllt werden müssen, als es Variablen gibt, verglichen mit einem *unterdeterminierten System*, wenige Lösungen hat, wodurch es nachdem das *GE* durchgeführt wurde, unkompliziert ist alle Lösungen herauszufinden.
Dadurch, dass es sich hierbei um immer eine homogene Matrix handelt (das Ergebnis ist der Nullvektor $\vec{0}$), gibt es immer mindestens eine Lösung ($\forall x_i : x_i = 0$). Da diese Lösung jedoch nicht genau k-Karten beinhaltet (sondern 0), muss diese Lösung ignoriert werden.
Ein LGS, wird womöglich, da es im Sachverhalt mindestens eine Lösung mit $k+1$ Karten gibt, noch eine zweite Lösung, neben der 0-Lösung haben, diese werden in jedem Fall über eine unabhängige Variable

nach dem Anwenden des *GE* ausgedrückt. Beispielsweise:

$$\left(\begin{array}{cccc|c} x_1 & x_2 & x_3 & x_4 & b \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{12}$$

Hierbei wäre x_3 eine unabhängige Variable und könnte frei gewählt werden. Da $x_3 \in \mathbb{Z}_2$ und $\mathbb{Z}_2$ ein endlicher Körper ist, gibt es dennoch endlich viele Lösungen. Zum Einen kann $x_3 = 1$ gewählt werden. Um nun die anderen Variablen abhängig von x_3 zu bestimmten, lassen sich die Gleichungen formulieren:

$$\begin{aligned} x_1 + x_3 &= 0 \\ x_2 + x_3 &= 0 \\ x_3 &= x_3 \\ x_4 &= 0 \end{aligned} \tag{13}$$

Dadurch, dass dass inverse Element der Addition von x_3 aufgrund des Körpers ebenfalls x_3 ist, folgt:

$$\begin{aligned} x_1 &= x_3 \\ x_2 &= x_3 \\ x_3 &= x_3 \\ x_4 &= 0 \end{aligned} \tag{14}$$

Verallgemeinert lassen sich alle abhängigen Variablen über eine Addition bzw. *XOR* aller unabhängigen Variablen, welche in ihrer Reihe verblieben sind, bestimmen.
Zum Anderen ließe sich $x_3 = 0$ wählen, was wiederum der 0-Lösung entsprechen würde. Alle Lösungen ließen sich als

$$L = \{x_3 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \mid x_3 \in \mathbb{Z}_2\} \tag{15}$$

formulieren. Jede unbestimmte Variable, welche nach dem Anwenden des *GE* verblieben ist, sorgt für zusätzliche Lösungen. Da es sich um $\mathbb{Z}_2$ handelt, lässt sich ableiten:

$$|L| = 2^{\text{Anzahl an unbestimmten Variablen}} \tag{16}$$

Bei Vernachlässigung der Eliminierung von Reihen durch Duplikate und Fälle wie Gleichung 11, lässt die Anzahl an Lösungen für *unterdeterminierte Systeme* als

$$|L| = 2^{n-m} \tag{17}$$

beschreiben, da sich für jede Gleichung (jeden Bit), eine Variable (Karte) eliminieren lässt.

1.4 Brute-Force der unabhängigen Variablen

1.4.1 Verallgemeinerung des Lösungsraumproblems

Eine Problematik, welche sich ergibt, nachdem der *GE*-Algorithmus durchgeführt wurde, und somit ein Lösungsmenge L bekannt ist, besteht darin, jene Lösungen herauszufinden, welche genau $k+1$ Karten einbinden. Eine verallgemeinerte Lösungsmenge L sieht wie folgt aus:

$$L = \{x_i \cdot \begin{pmatrix} a_{i1} \\ a_{i2} \\ \vdots \\ a_{in} \end{pmatrix} + \cdots + x_j \cdot \begin{pmatrix} a_{j1} \\ a_{j2} \\ \vdots \\ a_{jn} \end{pmatrix} \mid x_i, \cdots, x_j \in \mathbb{Z}_2\} \tag{18}$$

bzw. mit Logikverknüpfungen ausgedrückt:

$$L = \{x_i \wedge \begin{pmatrix} a_{i1} \\ a_{i2} \\ \vdots \\ a_{in} \end{pmatrix} \oplus \cdots \oplus x_j \wedge \begin{pmatrix} a_{j1} \\ a_{j2} \\ \vdots \\ a_{jn} \end{pmatrix} \mid x_i, \cdots, x_j \in \mathbb{Z}_2\} \tag{19}$$

Dabei geht $\wedge$ vor $\oplus$. Es gilt $L \in \mathbb{Z}_2^n$, wobei jeder Wert $l_i \in L$, für Karte i steht. Das Ziel ist es somit eine Auswahl für $x_i, \cdots, x_j$ zu finden, sodass genau $k+1$ Zeilen, 1 ergeben.
Dabei ist sicher, dass $a_{ii}, \cdots, a_{jj} = 1$, da jede unabhängige Variable auch zu einer Karte gehört. Dadurch lässt sich einschränken, dass für Lösungen, welche genau $k+1$-Karten einbinden, nicht mehr als $k+1$ unabhängige Variablen auf 1 gesetzt werden dürfen.
Die Problematik lässt sich auch als Aussagenlogik- bzw. Erfüllbarkeitsproblem ausdrücken:
Gegeben sind n *XOR*-Klauseln mit variierender Anzahl an Literalen, verknüpft über ein $\wedge$. Welche Literale müssen auf „wahr“ gesetzt werden, um genau $k+1$ Klauseln zu erfüllen?

1.4.2 Problematik von $k+1$

Es stellt sich als schwierig heraus, die Anzahl an zu nutzenden Karten $k+1$ direkt in das LGS zu integrieren, da sich das gesamte LGS in $\mathbb{Z}_2$ befindet, während $k \in \mathbb{N}$. Eine Überlegung wäre eine weitere Reihe bzw. Gleichung c_{m+1*} in das LGS einzufügen, mit welcher $k+1$ integriert wird:

$$\left(\begin{array}{cccc|c} x_1 & x_2 & \cdots & x_n & b \\ c_{11} & c_{12} & \cdots & c_{1n} & 0 \\ c_{21} & c_{22} & \cdots & c_{2n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} & 0 \\ 1 & 1 & \cdots & 1 & k+1 \end{array}\right) \tag{20}$$

Die Summe aller Variablen muss k ergeben. Da gilt $C^T \in \mathbb{Z}_2^{m+1 \times n}$, wird lediglich die *Parität* von $k+1$ als Bedingung für die Summe der Variablen übernommen. Es geht somit Information von $k+1$ verloren. Schlussendlich würde die Lösungsmenge alle Lösungen enthalten, welche aus p Karten bestehen, wobei $p \equiv k+1 \mod 2$. Insgesamt wäre die Anzahl der Lösungen durch das Hinzufügen dieser Zeile somit halb so groß, was sich auch an Gleichung 16 ableiten lässt. Somit wären auch die Möglichkeiten, welche durchprobiert werden müssten, halbiert. Andererseits würde die Eigenschaft der *homogenen Matrix* (Ergebnis ist der Nullvektor) verloren gehen, was das *GE* etwas komplizierter machen würde (Rückwertseinsetzen benötigt).
In die andere Richtung könnte man versuchen, das LGS in $\mathbb{N}$ zu übersetzen. Dies ist jedoch nicht möglich, da nicht jedes Element $a \in \mathbb{N}$ ein multiplikativ Inverses besitzt und die natürlichen Zahlen somit kein valider *algebraischer Körper* sind. Wenn man das LGS über die reellen Zahlen $\mathbb{R}$ lösen würde, können man es später aufgrund von rationalen Zahlen nicht mehr zurück auf $\mathbb{Z}_2$ und somit auf eine Auswahl von Karten umformen. Somit bleibt das Einbringen von $k+1$ nur bedingt möglich.

1.4.3 Naiver Ansatz

Letztendlich wurde ein naiver Ansatz gewählt. Alle Kombinationen von maximal $k+1$ unabhängigen Variablen können ausprobiert werden. Sollte sich bei einer beliebigen Kombination ergeben, dass genau $k+1$ Reihen „wahr“ sind, wurde eine Lösung gefunden. In Bezug auf den Sachzusammenhang sind genau diese $k+1$ Karten, Zaras gesuchten Karten und somit die Lösung des *ZZZ*. Insgesamt müssen somit

$$N = \sum_{i}^{k+1} \binom{\text{Anzahl der unabhängigen Variablen}}{i} \tag{21}$$

Kombinationen ausprobiert werden, da auch Kombinationen mit weniger als $k+1$ gewählten unabhängigen Variablen eine Lösung darstellen können, dadurch, dass eine unabhängige Variable beispielsweise mehrere Reihen auf „wahr“ setzt.

Eine Annahme könnte lauten, dass die Anzahl der unabhängigen Variablen, $n-m$ entspricht, da für jede Bedingung eine Variable eliminiert werden kann (Bei Vernachlässigung von Duplikaten):

$$N = \sum_{i}^{k+1} \binom{n-m}{i} \tag{22}$$

Der Brute-Force-Algorithmus besteht in einer Art *binärem Baum*, wobei die Tiefe der Anzahl an unabhängigen Variablen entspricht. An jedem Knoten wird eine unabhängige Variable entweder auf 1 oder auf 0 gesetzt. Sollten an einer Stelle mehr als $k+1$ unabhängige Variablen auf 1 gesetzt sein, wird abgebrochen und eine neuer Zweig, weiter oben, angefangen. Sollte die Anzahl der Einsen im Lösungsraumvektor L an einer Stelle $k+1$ entsprechen, ist die gesuchte Lösung gefunden (Näheres zur genauen Implementierung ist in Unterabschnitt 2.3 zu finden).
Da bei *überdeterminierten Systemen*, die Anzahl der unabhängigen Variablen sehr klein ist, können die *einfache* Beispiele der BWINF-Webseite mit diesem Verfahren ohne Probleme effizient gelöst werden.

1.5 Brute-Force des Gesamtproblems

Das ganze Problem soll noch einmal aus einer anderen Perspektive betrachtet werden und möglicherweise direkt (ohne *GE*) mit Brute-Force gelöst werden. Wie bereits in Unterabschnitt 1.2 formuliert, muss eine Menge $B \subseteq A$ gefunden werden, wobei $|B| = k+1$. Die *XOR*-Summe dieser Menge muss 0 sein. Somit ist die Teilsumme der Größe $k+1$ der Menge A gesucht.
Ein Brute-Force Ansatz könnte darin bestehen, alle Teilsummen der Größe $k+1$ zu berechnen und auszuprobieren. Die Anzahl dieser Teilsummen lässt sich mit einem Binomialkoeffizienten bestimmen:

$$N = \binom{n}{k+1} \tag{23}$$

Für Beispiel `stapel3.txt` wären dies bereits $\approx 3.3 \cdot 10^{16}$, was nicht umsetzbar wäre.
Eine wichtige Eigenschaft des *XORs*, um die Anzahl der zu bildenden Teilsummen zu reduzieren, ist, dass $A \oplus A = 0$. Da alle Teilsummen gesucht sind, welche 0 ergeben, lässt sich durch diese Eigenschaft, das Problem etwas abwandeln: Es sind zwei Mengen $B, B' \subseteq A$ mit $B \cap B' = \{\}$, $|B| + |B'| = k+1$ und

$$\bigoplus_{b \in B} b = \bigoplus_{b' \in B'} b' \tag{24}$$

gesucht. Eine Kombination von solchen zwei Mengen, würde nach $A \oplus A = 0$ zu der gesuchten Gesamtteilsumme von 0 führen und die Lösung des Problems darstellen.
Die optimale Größe für die Mengen B und B' wäre $\lceil \frac{k+1}{2} \rceil$ und $\lfloor \frac{k+1}{2} \rfloor$, da in allen anderen Fällen größere Mengen gefunden werden müssten. Für die Anzahl dieser Teilsummen halber Größe folgt:

$$N = \binom{n}{\lceil \frac{k+1}{2} \rceil} \tag{25}$$

Die Kombination von B und B' macht das Ganze deutlich effizienter, da durch eine Sortierung der gefundenen Teilsummen halber Größe, ohne Probleme gleiche Teilsummnen gefunden werden können. Dafür müsste lediglich einmal die sortierte Liste an Teilsummen angeschaut werden.
Für das Beispiel `stapel3.txt` gäbe es nun $2.2 \cdot 10^{10}$ Teilsummen halber Größe.

Eine weitere Idee wäre auch, dieses Vorgehen des Halbierens und Kombinierens der Teilsummen auf das Brute-Force-Problem der unabhängigen Variablen (Unterunterabschnitt 1.4.3) zu übertragen und somit eine deutliche Effizienzsteigerung zu erzeugen. Das Problem der unabhängigen Variablen ist dabei jedoch nicht eine bestimmte *XOR*-Summe zu erzeugen, sondern durch Addition (bzw. *XOR*), eine besimmte Anzahl von Einsen zu erzeugen.

Beide beschriebenen Verfahren (Reines Brute-Force und *GE*-Brute-Force), lassen sich anwenden. Insofern jedoch die Eingabe in einer *überdeterminierten* Matrix ($n < m$) besteht, sollte immer das *GE* angewendet werden. Da damit eine Lösung in Polynomialzeit mögliche ist (siehe Abschnitt 3). Für den Vergleich dieser beiden Verfahren lassen sich womöglich die Anzahle an auszuprobierenden Kombinationen bzw. Teilsummen vergleichen:

$$N = \binom{n}{\lceil \frac{k+1}{2} \rceil} \quad \vee \quad N = \sum_{i}^{k+1} \binom{n-m}{i} \tag{26}$$

2 Umsetzung

2.1 main() und parseInput()

Die *main()*-Funktion initiiert den Lösungsprozes und übergibt die durch *parseInput()* gelsenene Eingabe. Zunächst soll der in Unterabschnitt 1.5 beschriebene Vergleich der auzuprobierenden Werte angestellt werden. Zuerst wird überprüft ob es sich um ein *überdeterminiertes System* ($m > n$) handelt $\Rightarrow$ (*GE*-Verfahren). Sollte dies nicht der Fall sein, werden mithilfe der *math*-Bibliothek, die beiden Binomialkoeffizienten berechnet, wobei die Summe vernachlässigt wird. Dementsprechend, welches Verfahren eine geringere Anzahl an durchzuprobierenden Elementen verspricht, wird die Funktion *bfAll()* oder *gaussElim()* ausgeführt.
Dabei wird direkt $k+1$ übergeben, da der Unterschied zwischen Öffnungskarten und Sicherungskarte, erst beim Öffnen der Häuser wieder relevant wird. Zudem wird in der *main()*-Funktion die Lösung ausgegeben und zeilenweise in eine `.txt`-Datei mit dem Präfix *ergebnis_* geschrieben. Dabei ist relevant, dass die, zu den durch herausgefundenen Indizes gehörigen, Karten in die Datei geschrieben werden.
In der *parseInput()*-Funktion wird die in einem Argument übergebene Datei eingelesen. Mithilfe der *readlines()*-Funktion, werden die entscheidenden Variablen n,k,m und eine Liste von Strings der *karten* gelesen. Anschließend werden diese zurückgegeben.

2.2 gaussElim()

Als Parameter für die *gaussElim()*-Funktion müssen n, k und m als *int* übergeben werden. Die Variable k entspricht hierbei jedoch bereits der Anzahl an gesuchte Karten (inkl. Sicherungskarte). Zudem wird die Liste an Karten übergeben (*Strings*).

2.2.1 Eliminationsverfahren

Zunächst muss, wie in der Lösungsidee beschrieben, eine transponierte Matrix erstellt werden. Diese wird zunächst über „einzeilige“ *for-Schleifen* initialisiert und daraufhin in einer Iteration über $n \times m$, mit den Werten der Karten gefüllt. Wobei jedoch das vertauschen bzw. transponieren der Indizes wichtig ist, damit letztendlich jede Spalte zu einer Karte gehört. Als Nächstes könnte wahlweise eine weitere Reihe mit ausschließlich Einsen für die *Parität* von k hinzugefügt werden (siehe Unterunterabschnitt 1.4.2).
Nachdem die transponierte *tMatrix* erfolgreich erstellt wurde, wird das *GE* durchgeführt. Dafür muss immer die erste Spalte, mit einer 1 in jeder Reihe gefunden werden.
Somit wird über alle Reihen und Spalten iteriert. Sobald ein Wert mit 1 gefunden wurde, muss die aktuelle Reihe auf alle anderen Reihen, welche über eine Eins in dieser Spalte verfügen, addiert (in $\mathbb{Z}_2$) werden. Dafür muss wiederum über alle Reihen iteriert werden und überprüft werden inwiefern eine Eins vorhanden ist und es sich nicht um die gleiche Reihe handelt. Daraufhin kann die gefundene Reihe mit dem Summe Modulo Zwei ($\mathbb{Z}_2$) aktualisiert werden. Nachdem die Addition für alle anderen Reihen abgeschlossen wurde, wird die Iteration über alle Spalten abgebrochen, da der geschliderte Prozess nur für die erste Spalte welche 1 entspricht vollzogen werden soll. Es wird mit der nächsten Reihe fortgefahren.

2.2.2 Bestimmung der unabhängigen Variablen

Folgend müssen die in Unterabschnitt 1.4 beschriebenen unabhängigen Variablen, welche nach dem *GE* übrig geblieben sind bestimmt werden. Dafür sind zwei Schritte nötig.

1. Zunächst müssen die Indizes der unabhängigen Variablen sowie die Indizes der Karten, welche von der jeweiligen Variable abhängen, bestimmt werden. Dafür eignet sich die Datenstruktur einer *Dictionary*. Als Schlüssel wird dabei der 0-basierte Index der Variablen verwendet (z.B. für $x_5 \Rightarrow 4$). Der zugehörige Wert besteht in einer Liste von ebenfalls 0-basierten Indizes, welche jene Karten bzw. Reihen verkörpern, welche von dem gegebenen Schlüssel abhängig sind. Die *Dictionary variablen* wird daraufhin in einer Iteration über die Reihen der transponierten und eliminierten Matrix gefüllt. Für jede Reihe muss dabei der Index der ersten Spalte ermittelt werden, da diese signifikant für die zugehörige Karte ist (Jede Spalte gehört zu einer Karte). Folglich wird über alle Spalten der Reihe iteriert und überprüft, inwiefern es sich um die erste Eins handelt. Sollte eine Eins an der iterierten Stelle bestehen, so sorgt diese, wenn es sich nicht um die erste Eins handelt, für eine Abhängigkeit zu einer unabhängigen Variable. Somit muss entweder eine neue unabhängige Variable zu den *variablen* hinzugefügt werden, oder ein weitere Index zu einer bestehenden Variable hinzugefügt werden.

Dieser hinzuzufügende Index, muss dabei der erste Index in der aktuellen Reihe. Da die Matrix nicht nach den Reihen sortiert wurde und somit keine Dreiecksmatrix erzeugt wurde, sondern die Reihen zwar eliminiert, jedoch durcheinander sind, ist die beschriebene Mechanik, mit der ersten Eins der Reihe nötig. Somit können alle unabhänigen Variablen und die von ihnen abhängigen Kartenindizes herausgestellt werden.

2. Der zweite Schritt dient zur Vereinfachung für das folgende Brute-Force-Vorgehen. Die beschriebenen herausgefundenen *variablen*, sollen vereinfacht über ihren Vektor (siehe Unterunterabschnitt 1.4.1), welcher die Abhängigkeiten der Kartenindizes beinhaltet dargestellt werden, ohne die beschrieben *Dictionary beizubehalten*. Jede unabhängige Variable soll durch ein *Bitstring* in der Liste *processedVariablen* vorhanden sein.
   Dafür wird über alle Variablen iteriert und für jede ein neuer leerer *String* eingefügt. Dieser wird folgend in einer Iteration über alle Kartenindizes mit einer 0, wenn der Kartenindex nicht in der entsprechenden Liste der (*Dictionary*-Wert) vorhanden ist, und mit einer 1, wenn der Kartenindex abhängig von der iterierten Variable ist oder die unabhängige Variable selbst ist, erweitert.

Nachdem somit für jede unabhängige Variable ein *Bitstring* mit den entsprechenden Abhängigkeiten der Karten erzeugt wurde, kann der Brute-Force-Prozess gestartet werden (Unterabschnitt 2.3). Dabei wird die Anzahl an gesuchten Karten k, die erstellte Liste *processedVariablen*, die Gesamtanzahl an Karten n und der Startindex 0 übergeben.
Nachdem eine Lösung durch die *bfVars()*-Funktion zurückgegeben wurde, wird diese in die entsprechenden Indizes der Karten umgewandelt, da es sich beim Rückgabewert ebenfalls um ein *Bitstring* handelt. Dies kann mithilfe einer Iteration über jedes Zeichen des *Bitstrings* vollführt werden. Schlussendlich werden die herausgefundenen Indizes der zu wählenden Karten zurückgegeben.

2.3 bfVars()

Bei der *bfVars()*-Funktion handelt es sich um eine rekursive Funktion. Es sollen alle Kombinationen der unabhängigen Variablen (*processedVariablen*) gemäß Unterunterabschnitt 1.4.3 ausprobiert werden, wobei eine Kombination gesucht ist, bei der der Lösungsraumvektor genau k Einsen beinhaltet.
Zunächst werden die Anzahl an gesuchten Karten k, Anzahl an Gesamtkarten n und die ermittelten Variablen übergeben. Für die Rekursion werden dabei der *index* und der *zweig* übergeben. Der *index* entspricht dabei der als Nächstes auszuprobierenden unabhängigen Variable. Der *zweig* ist eine Liste aus zwei Elementen. Zum Einen beinhaltet diese, die Anzahl an gewählten unabhängigen Variablen. Zum Anderen einen *Bitstring*, welcher dem aktuellen Lösungsvektor entspricht. Sollte die Anzahl an Einsen in diesem *Bitstring* bereits k entsprechen, so ist die Lösung gefunden und kann zurückgegeben werden.
Sollte die Anzahl der gewählten unabhängigen Variablen, k überschreiten, ist dieser Zweig nicht mehr weiter zu erweitern und es wird eine Instanz der *NoneType*-Klasse zurückgegeben.
Zudem muss überprüft werden, ob der *index* noch valide ist (Nicht größergleich der Anzahl an Variablen ist).
Sollte keine Abbruchbedingung zutreffen, kann der *zweig* durch das Hinzufügen der *variable[index]*, erweitert werden. Dafür muss die Anzahl an genutzten Variablen im *zweig* um Eins erhöht werden. Für den *Bitstring*, welcher dem Lösungsvektor entspricht, kann die *XOR* Operation angewendet werden. Der neue Lösungsvektor entspricht dem *XOR* des alten Lösungsvektors und des *Bitstrings* bzw. Lösungsvektors der *variable[index]*. Das Ergebnis dieser Verknüpfung wird sofort in einen *String* konvertiert.
Folgend wird die *bfVars()*-Funktion rekursiv aufgerufen, dabei wird der erweiterte Zweig übergeben und der Index erhöht. Sollte diese Erweiterung des Zweiges eine Lösung liefern, wird diese ebenfalls zurückgegeben. Sollte dieser Zweig eine *NoneType*-Instanz zurückgeben und somit keine Lösung für das Problem liefern, muss der Zweig in in die andere Richtung, ohne die *variable[index]* auf 1 zu setzen, erweitert werden. Die rekursive Funktion wird dementsprechend nur mit erhöhtem Index aufgerufen und das Ergebnis dieses Aufrufes direkt zurückgegeben.

2.4 bfAll()

Mit dieser Funktion soll die Kombination aus k Karten, welche verknüpft mit dem *XOR* 0 ergibt, durch Brute-Force bestimmt werden.
Zunächst werden dafür die Parameter n, k (inkl. Sicherungskarte), m und die Liste an *karten* (*Bitstrings*) übergeben. Im Folgenden sollen alle Kombinationen aufgenommen werden, welche aus $\frac{k+1}{2}$ Karten bestehen. Für $k \equiv 1 \mod 2$, ist dabei wichtig, dass es zwei verschiedenen Längen gibt, welche für Kombinationen als passend gelten (aufgerundet und abgerundet). Dafür können die *ceil()* und *floor()* Funktionen

der *math*-Bibliothek genutzt werden. Bei $k \equiv 0 \mod 2$ folgt, dass diese beiden Längen gleich groß sind. Folgend wird die Liste aller Kombinationen angelegt, diese beinhaltet jene Kombinationen, welche zwischengespeichert werden müssen, um die Kombinationen mit den Größen *längeA* und *längeB* zu generieren. Die Liste *finalKombinationen* beinhaltet alle Kombinationen der besagten Größen.
Daraufhin beginnt das Brute-Force-Vorgehen. Es wird über alle Karten iteriert. Für jede Karte wird über alle bisherigen *kombinationen* iteriert. Dabei ist zu beachten, dass aufgrund des Hinzufügens während der Iteration, vom letzten Index, bis zum ersten Index iteriert wird. Zudem wird eine zusätzliche Iteration getätigt. In dieser Iteration wird eine vollständig neue, allein aus der aktuelle Karte bestehende, Kombination hinzugefügt. Eine Kombination besteht dabei aus einer Liste mit zwei Elementen. Zunächst beinhaltet eine Kombination eine Liste mit den teilhabenden Indizes der Karten. Das zweite Element ist die aktuelle Teilsumme (*Bitstring*). Jedes Mal, wenn eine Karte zu einer Kombination hinzugefügt wird, wird der neue Index hinzugefügt und ein *XOR* der Karte mit der aktuellen Teilsumme durchgeführt. Folgend wird überprüft, ob die Anzahl an beteiligten Karten, einer der gesuchten Längen entspricht. Falls dem so ist, wird die Kombination zu *finalKombinationen* hinzugefügt.
Sollten mehr Karten beteiligt sein, als *längeA*, so ist die Kombination für folgende Iterationen unbrauchbar, und darf nicht zu *kombinationen* hinzugefügt werden.

Um die verschiedenen *finalKombinationen* nun gemäß der Lösungsidee zu einer Lösungskombination zu vereinen, wird die Liste der *finalKombinationen* nach dem *Bitstring* sortiert, wobei die *sort(key)*-Argumentübergabe genutzt wird.
Abschließend wird über die *finalKombinationen* iteriert und permanent die zuletzt gesehene Kombination gespeichert, sodass überprüft werden kann, ob der *Bitstring* übereinstimmt und somit ein *XOR* von 0 erzeugt werden kann. Dabei ist auch zu überprüfen, dass die Längen der beiden Kombinationen mit *längeA*+*längeB* übereinstimmen (bei ungeradem k). Weiterführend muss überprüft werden, inwiefern eine Karten in beiden Kombinationen auftaucht, und die Zusammenführung somit ungültig macht. Dies kann verinfacht mit einer *Set-Intersection* vollzogen werden. Wenn keine Überschneidungen zwischen den eingebundenen Indizes vorhanden sind, wurde eine Lösung gefunden, welche zurückgegeben wird.

3 Zeitkomplexität

3.1 Komplexitätsklasse

Dadurch, dass es sich bei *ZZZ* um eine Art Suchproblem handelt, und somit kein Optiumum gefunden oder eine Entscheidung getroffen werden muss, ist es schwierig eine Formulierung als Entscheidbarkeitsproblem zu treffen. Dies verhindert zudem eine gewöhnliche Polynomialzeitreduktion, wodurch womöglich bewiesen werden kann, dass $ZZZ \in NPC$. Vor Allem durch die Formulierung als partielles Erfüllbarkeitsproblem (Unterunterabschnitt 1.4.1) lässt sich stark vermuten, dass $ZZZ \in NPC$. Womöglich ist der Algorithmus jedoch mit einigen Optimierungen so zu gestalten, dass selbst die *unterdeterminierten Systeme* ohne Probleme löst.
Im Folgenden sollen somit lediglich die asymptotischen Laufzeiten der einzelnen Programmabschnitte herausgestellt werden.

3.2 Gaussches Eliminationsverfahren

Die Bestimmung der Zeitkomplexität des *GE* stellt sich als unkompliziert heraus. Für jede der m Reihen der transponierten Matrix muss jede der n Spalten angeschaut werden. Daraufhin muss die Reihe auf jede andere Reihe addiert werden. Da hierbei jeder Koeffizient einzeln addiert wird, muss nochmals für die Addition über jede der n Spalten iteriert werden.
Daraus ergibt sich insgesamt eine asymptotische Laufzeit von $O(n^2m^2)$.

3.3 Brute-Force der unabhängigen Variablen

Beim Brute-Force-Ansatz, für *unterdeterminierte Systeme*, nachdem das *GE* durchgeführt wurde, müssen wie erörtert

$$N = \sum_{i}^{k+1} \binom{n-m}{i} \tag{27}$$

Kombinationen ausprobiert werden. Dadurch, dass jede dieser Kombinationen durch ein Knoten in dem beschriebenen binären Baum besichtigt wird, entspricht die Laufzeit dieses Abschnittes, der Anzahl an Kombinationen. Dadurch, dass es sich um eine asymptotische Laufzeit handelt, ist lediglich der größte Summand relevant:

$$O\left(\binom{n-m}{k+1}\right) = O\left(\frac{(n-m)!}{(k+1)!\cdot(n-m-(k+1))!}\right) \tag{28}$$

Um die asymptotische Laufzeit in einem exponentiellen Zusammenhang oder einer potentiellen Zusammenhang auszudrücken, eignet sich die *Stirlingformel*, um die Fakultät eines Wertes anzunähern:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{29}$$

Nach Einsetzen folgt:

$$\begin{aligned}
&O\left(\frac{\sqrt{2\pi(n-m)}\left(\frac{n-m}{e}\right)^{n-m}}{\sqrt{2\pi(k+1)}\left(\frac{k+1}{e}\right)^{k+1}\cdot\sqrt{2\pi(n-m-k-1)}\left(\frac{n-m-k-1)}{e}\right)^{n-m-k-1}}\right)\\
&\approx O\left(\frac{\sqrt{n-m}\cdot(n-m)^{n-m}\cdot e^{m-n}}{\sqrt{k+1}\cdot(k+1)^{k+1}\cdot e^{1-k}\cdot\sqrt{n-m-k-1}\cdot(n-m-k-1)^{n-m-k-1}\cdot e^{m+k+1-n}}\right)\\
&\approx O\left(\frac{\sqrt{n-m}\cdot(n-m)^{n-m}}{\sqrt{k}\cdot k^k\cdot\sqrt{n-m-k}\cdot(n-m-k)^{n-m-k}}\right)
\end{aligned} \tag{30}$$

3.4 Brute-Force des Gesamtproblems

Ähnlich wie in Unterabschnitt 3.3, lässt sich die Zeitkomplexität anhand des Binomialkoeffizienten:

$$O\left(\binom{n}{\frac{k+1}{2}}\right) \tag{31}$$

abschätzen. Nach der *Stirlingformel* folgt:

$$\begin{aligned}
&O\left(\frac{\sqrt{n}\left(\frac{n}{e}\right)^n}{\sqrt{\frac{k+1}{2}}\left(\frac{\frac{k+1}{2}}{e}\right)^{\frac{k+1}{2}}\cdot\sqrt{n-\frac{k+1}{2}}\left(\frac{n-\frac{k+1}{2}}{e}\right)^{n-\frac{k+1}{2}}}\right)\\
&\approx O\left(\frac{\sqrt{n}n^n e^{2(k-n)}}{\sqrt{k}k^k\cdot\sqrt{n-k}(n-k)^{n-k}}\right)
\end{aligned} \tag{32}$$

4 Beispiele

4.1 Randinformationen

Das Programm kann durch `python A4.py stapel[X].txt` ausgeführt werden.
Alle Lösungen werden zusätzlich in einer Datei mit dem Präfix *ergebnis_* abgegeben.
Bei der Beurteilung der tatsächlichen Laufzeit, muss die Ineffizienz der Programmiersprache *Python*, sowie die Nutzung der CPU: *Intel Core i7-4770 3.40 GHz* beachtet werden.
Es wird jeweils eine der $k+1$ Karten pro Zeile ausgegeben.

4.2 BWINF-Beispiele

stapel0.txt

Kommentierungen: Hierbei wird eine 32×20 Matrix gelöst $\Rightarrow$ *überdeterminiert*. Nach dem *GE* ist eine unabhängige Variable vorhanden. Somit werden zwei bzw. sogar nur eine Möglichkeit ausprobiert.

```
python A4.py stapel0.txt
10111000001100111000010101011111 0
00111101010111000110100110011001
11111110001011010001000000110111
```

```
11010111111010111101101111110000
10101100111111011010100011100000
```
— 0.0019 Sekunden —

stapel1.txt

Kommentierungen: Hierbei wird eine 32×20 Matrix gelöst $\Rightarrow$ *überdeterminiert*. Nach dem *GE* ist eine unabhängige Variable vorhanden. Somit werden zwei bzw. sogar nur eine Möglichkeit ausprobiert.

```
python A4.py stapel1.txt
00100000111100111110111101111100
11010011010110110101001101010111
00110100001010100100001111010010
11110011101011001001000010111110
00110110000110101101011111111010
11110111100100010100100001001110
00100011100111011010111011100011
11000111111010110100000101110100
00010001110100110001111101100100
```
— 0.0021 Sekunden —

stapel2.txt

Kommentierungen: Hierbei wird eine 128×111 Matrix gelöst $\Rightarrow$ *überdeterminiert*. Nach dem *GE* ist eine unabhängige Variable vorhanden. Somit werden zwei bzw. sogar nur eine Möglichkeit ausprobiert.

```
python A4.py stapel2.txt
01101011101000110111010001100001110000011000110101100010111011100110011011110111011100110101101111000011110111011101011111100111
00101011111000101011010110111100100110000000000011010011001111011001011001000010001101010110110010101110100100001011100011010001
10101011000001101100000101111111110011000110011100101011011111011000111111101111101000111111010000000101101101111111101001101111 0
10000000000100100110011001000110000000000010101011010010010000100011101011011010101001010100010111010110010100011001010010011101 1
00101000011000010010111011101011011010111000100100110101111011011110111100101100001001110010100001101001110001000100010011111100
11000011000100110111000101100100101101010110011011010101101001000011110100010001001010000110010101010010001100010001101110100000
10101111110010010010100111101100010011111000010101001100100001111000100010010010011010010101011111101011000001111110000000111011
11011110000101001101111100110000111010011011101111010111110110110111011010001001101101100111010001000011000001010111100101111111
01101001001011000101001111111101011000001000101100111010100101011011000100000000110000110001101011010101011110110100000100101001011
01110110011110001110011110001101101110100101000000010000010110000101000111010100000000110100110000110100101101101001011111101101000
11101110101011100111101111000111001101111011010101011111000110100011010001100000111101000010001100100000001110110001000101110100 0
```
— 0.0729 Sekunden —

stapel3.txt

Kommentierungen: Hierbei wird eine 128×161 Matrix gelöst $\Rightarrow$ *unterdeterminiert*. Nach dem *GE* sind 33 unabhängige Variable vorhanden. Beide Vefahren gestalten sich aufgrund der großen Werte von k und n als Ineffizient. Die Reallaufzeit ist dementsprechend groß.

```
python A4.py stapel3.txt
01101011101000110111010001100001110000011000110101100010111011100110011011110111011100110101101111000011110111011101011111100111
00101011111000101011010110111100100110000000000011010011001111011001011001000010001101010110110010101110100100001011100011010001
10101011000001101100000101111111110011000110011100101011011111011000111111101111101000111111010000000101101101111111101001101111 0
10000000000100100110011001000110000000000010101011010010010000100011101011011010101001010100010111010110010100011001010010011101 1
00101000011000010010111011101011011010111000100100110101111011011110111100101100001001110010100001101001110001000100010011111100
11000011000100110111000101100100101101010110011011010101101001000011110100010001001010000110010101010010001100010001101110100000
10101111110010010010100111101100010011111000010101001100100001111000100010010010011010010101011111101011000001111110000000111011
11011110000101001101111100110000111010011011101111010111110110110111011010001001101101100111010001000011000001010111100101111111
01101001001011000101001111111101011000001000101100111010100101011011000100000000110000110001101011010101011110110100000100101001011
```

```
01110110011110001110011110001101101110100101000000100000101100001010001110101000000011010011000011010010110110100101111101101000
11101110101011100111101111000111001101111011010101011111000110100011010001100000111101000010001100100000001110110001000101110100
```
— 1293.5714 Sekunden —

stapel4.txt

Kommentierungen: Dieses Beispiel ist das schwerste, dadurch, dass das k und n entsprechend groß sind. Es würde eine 128×181 Matrix bestehen $\Rightarrow$ *unterdeterminiert*. Nach dem *GE* sind 53 unabhängige Variable vorhanden. Das Beispiel konnte während des Ausprobierens der Idee, das Halbieren von Teilsummen auf den Brute-Force-Algorithmus der unabhängigen Variablen zu übertragen, frühzeitig gelöst werden, dadurch, dass lediglich $\lfloor \frac{k}{2} \rfloor$ unabhängig Variablen kombiniert werden mussten. Die erläuterte Lösungsidee ist jedoch zu ineffizient, weswegen das Lösen dieses Beispiels **nicht mit in die Bewertung einfließen darf**. Es wird der Vollständigkeit halber trotzdem aufgelistet.

```
python A4.py stapel4.txt
11100010000000111101001111111001001100111011101001000111001111001111000010000101100001000011000011001011101101010000101111100001
00101100000111101111000010000000101101111111000011001111111111000111100000010111110110001001000001010000101011111011000010100101
00000111011010010101101110001111101001110010100011000001000001101110101111101001000100000000111101111100110101101000010001101111
00110110010111001001001111001111101010011100000100000000110001010100011100100010011100010011011111111001000100101001001110111101
00101100001110001110011110000100110000000000111011011001110100010100001010000110110010000111110011000100111101001100100010010000
01000011001001101011001111011101111010010110111010111110110111111001000100010110101100101111110000011000100111011000001101000111
10000000100010111001101010001100011010011010011110010001000100001101100000101011110011011101110111000101111000010010000111000010
11000101110001001000001011101000100110011001111011101001011010110100111000100001000100000001100001010111100100110110101011101110
10101010000011111111001111011110001000100111010000010010111101000001010001100010110011110111111010111000000001100011011111100001
10001100000101100110100101100011011010100110100001000010110101010110011011011111101011001100100110110010000101111000011101000011
11111001011000110001010011000100111000111101001110010001101010010100100010011110010100000100001110101001110100011100100000000111
```
— 12.1962 Sekunden —

stapel5.txt

Kommentierungen: Dadurch, dass das k vergleichsweise klein ist, gestaltet sich durch die weitere Halbierung, das *bfAll()* Verfahren effizient.

```
python A4.py stapel5.txt
1010111011001100100110001100110001011101001000000011011111100100
1010000110101100101110111001100011011110111111010111000101111110
1000010011101010001111100100110110011011100101010100010000001001
1101010001001101000111111110000110100010100111000100001001011011
0101111111000111000000101111100010111010110101000100000011001000
```
— 2.7450 Sekunden —

4.3 Eigene Beispiele

stapel_custom0.txt

Kommentierungen: In diesem Beispiel soll ausprobiert werden, wie sich doppelte Karten auf den Lösungsprozess auswirken:

```
5 2 3
001
110
111
111
111
```

Durch das *GE* werden gleichen Zeilen, was nach dem Transponieren in diesem Beispiel Zeilen 2 und 3 sind, eliminiert. Es ist nur noch eine Zeile übrig, wodurch 3 unabhängige Variablen vorhanden sind, mit welchen danach durch Brute-Force eine Lösung mit 3 Karten gefunden wird. Interessant ist hierbei, dass diese transponierte Matrix, *unterdeterminiert*, jedoch die Anzahl der Variablen nicht $n - m$ entspricht, dadurch dass Reihen eliminiert werden.

```
python A4.py stapel_custom0.txt
001
110
111
```

— 0.0010 Sekunden —

stapel_custom1.txt

Kommentierungen: In diesem Beispiel soll ausprobiert werden, wie sich keine Lösung auf den Lösungsprozess auswirkt:

```
16 5 16
1000000000000000
0100000000000000
0010000000000000
0001000000000000
0000100000000000
0000010000000000
0000001000000000
0000000100000000
0000000010000000
0000000001000000
0000000000100000
0000000000010000
0000000000001000
0000000000000100
0000000000000010
0000000000000001
```

Es wird lediglich die 0-Lösung gefunden, welche hierbei in *bfVars()* aussortiert wird.

```
python A4.py stapel_custom1.txt
```

— 0.0010 Sekunden —

stapel_custom2.txt

Kommentierungen: Hierbei ist interessant was passiert, wenn es mehrere Lösungen gibt. Zudem entspricht $k = 1$, weswegen lediglich 2 Karten gefunden werden müssen. Zudem könnte es Probleme geben, da sofort alle Reihen in der transponierten Matrix eliminiert werden. Denkbar wäre eine Art *Preprocessing*, wobei alle doppelten Karten entfernt werden:

```
10 1 16
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
```

Es verbleiben 9 unabhängige Variablen. Wie erwartet ist eine Kombination aus allen Karten möglich.

Dadurch, dass sobald eine Lösung gefunden wird abgebrochen wird, wird auch nur eine Lösung gefunden.

```
python A4.py stapel_custom2.txt
1111111111111111
1111111111111111
```

— 0.0020 Sekunden —

stapel_custom3.txt

Kommentierungen: Es soll überprüft werden was passiert, wenn $k = 0$. Dabei ist auch die Reaktion auf eine 0-Matrix interessant:

```
5 0 8
00000000
00000000
00000000
00000000
00000000
```

Dadurch, dass es keine 1 gibt, kann ist die einzige Lösung für das LGS, die 0-Lösung, welche hierbei nicht der Anzahl an gesuchten Karten (1) entspricht.

```
python A4.py stapel_custom3.txt
```

— 0.0010 Sekunden —

5 Quellcode

```python
# Für Messung der Laufzeit
from time import time
# Zum Überprüfen ob Dateien exisitieren
from os.path import exists
# Zur Übergabe von Argumenten im Terminal
from sys import argv
# Zum Berechnen von Fakultäten und Auf-/Abrundungen
from math import comb, ceil, floor
from typing import final

# Funktion zum Anwenden von Brute Force auf das Gesamtproblem (Kombination aller n Karten)
# n: Gesamtzahl an Karten
# k: Anzahl an gesuchten Karten (inkl. Sicherungskarte)
# m: Anzahl Bits pro Karte
# karten: Liste aller Karten (Strings)
def bfAll(n, k, m, karten):
    # Berechnen der beiden Längen
    # => welche die Listen haben müssen, welche kombiniert werden
    längeA = ceil(k/2)
    längeB = floor(k/2)
    # Liste aller Kombinationen der Karten (Zwischenspeicherung)
    kombinationen = []
    # Liste der Kombination mit Länge A oder B
    finalKombinationen = []
    # Iteration über alle Karten
    for i in range(n):
        # Ausgabe des Fortschritts
        print(i,n,len(kombinationen))
        # Iteration über alle bisherigen Kombinationen (+1
        # extra Iteration für eine vollständig neue Kombination)
        for j in range(len(kombinationen), -1, -1):
            # Überprüfung ob es sich um die erste Iteration handelt
            if j == len(kombinationen):
                # Hinzufügen einer neuen Kombination für die Karte i
                neueKombination = [[i], karten[i]]
```

```
            else:
                # Erweitern von Kombination j um die Karte i
                neueKombination = [kombinationen[j][0] + [i],
                ("{0:b}".format(int(kombinationen[j][1], 2) ^ int(karten[i], 2))).zfill(m)]
            # Überprüfung ob die Kombination komplett ist
            if len(neueKombination[0]) == längeA or len(neueKombination[0]) == längeB:
                finalKombinationen.append(neueKombination)
            # Überprüfung ob die Kombination noch erweitert werden kann
            if len(neueKombination[0]) < längeA:
                kombinationen.append(neueKombination)

    # Sortieren der Listen nach dem Wert des XOR-Bitstrings
    finalKombinationen.sort(key=lambda x: int(x[1], 2))

    # Suchen nach Duplikaten in der Liste
    # Aktuell letzte Kombination
    letzteKombination = finalKombinationen[0]
    # Iteration über alle restlichen Kombinationen
    for kombination in finalKombinationen[1:]:
        # Überprüfung ob der Bitstring identisch ist und somit das XOR = 0 ist
        # Überprüfung ob Längen den festgelegten Längen
        # entsprechen (und nicht z.B. zweimal die kürzere benutzt wird)
        if kombination[1] == letzteKombination[1] and
        len(kombination[0])+len(letzteKombination[0]) == längeA+längeB:
            # Überprüfung ob eine Karte in beiden Kombinationen
            # vorkommt und somit die Vereinung die valide ist
            if len(list(set(kombination[0]) & set(letzteKombination[0]))) == 0:
                # Zurückgeben der Vereinten Kombinationen
                return [kombination[0]+letzteKombination[0]]
        # Setzen der neuen letzten Kombination
        letzteKombination = kombination
    # Zurückgeben einer leeren Liste (keine Lösung gefunden)
    return []

# rekursive Funktion zum Anwenden von Brute Force auf das
# Restproblem (Kombination der Variablen nach dem Gauss-Algorithmus)
# => nur bei unterdeterminierten Eingaben/Matrizen (wenn n>m)
# k: Anzahl an gesuchten Karten (inkl Öffnungskarte)
# variablen: Liste der Bitstrings der Variablen (Auswirkung auf das Ergebnis des Gauss-Algorithmus)
# index: Index der aktuellen Variable
# zweig: [Anzahl an benutzten Variablen, Bitstring (XOR der in diesem Zweig benutzten Variablen)]
# n: Gesamtanzahl an Karten
def bfVars(k, variablen, index, zweig, n):
    # Überprüfung ob der aktuelle Zweig einer Lösung entspricht
    # (Anzahl an 1 = Anzahl an zu benutzenden Karten)
    if zweig[1].count("1") == k:
        return zweig[1]
    # Abbrechen, wenn bereits mehr Variablen genutzt wurden, als Karten gesucht sind
    elif zweig[0] > k:
        return None
    else:
        # Überprüfung, ob bereits keine Variablen mehr übrig sind
        if index > len(variablen)-1:
            return None
        # Festlegen des Wertes (Bitstings) der aktuellen Variable
        wert = variablen[index]
        # Berechnen (XOR) für den neuen Zweig
        neuerZweig = [zweig[0]+1,("{0:b}".format(int(zweig[1], 2) ^ int(wert, 2))).zfill(n)]
        # Erweitern des neuen Zweiges
        result = bfVars(k, variablen, index+1, neuerZweig, n)
        # Bei Fehlschlagen mit Wahl der Variable, wird ohne die Variable fortgefahren
        if result is None:
            return bfVars(k, variablen, index+1, zweig, n)
        else:
            # Andernfalls wird das Ergebnis zurückgegeben
            return result

# Funktion für das Gausssche Eliminationsverfahren und Brute-Force der Variabalen-Kombinationen
# n: Anzahl an Gesamtkarten (int)
# m: Anzahl an Bits pro Karte (int)
# k: Anzahl an gesuchten Karten (inkl. Sicherungskarte)
# karten: Liste an Karten (Strings)
def gaussElim(n, k, m, karten):
```

```
    # Anlegen einer Matrix (transponierte Matrix => m und n sind vertauscht)
    tMatrix = [[0 for _ in range(n+1)] for _ in range(m)]
    # Füllen der transponierten Matrix (Iteration über alle Reihen der ursprünglichen Matrix)
    for i in range(n):
        # Iteration über alle Spalten der ursprünglichen Matrix
        for j in range(m):
            # Einsetzen des Wertes in die transponierte Matrix
            tMatrix[j][i] = int(karten[i][j])
    # Hinzufügen einer Zeile für die Parität von k
    # tMatrix.append([1 for _ in range(n)]+[k % 2])
    # Eliminationsverfahren
    # Iteration über alle Reihen der transponierten Matrix (O(m))
    for r1 in range(len(tMatrix)):
        # Iteration über alle Spalten der transponierten Matrix (O(n))
        for c1 in range(len(tMatrix[r1])):
            # Überprüfung, ob es sich um die erste Eins in dieser Reihe handelt
            if tMatrix[r1][c1] == 1:
                # r1 muss auf alle anderen Reihen xored werden
                # => Iteration über alle anderen Reihen (O(m))
                for r2 in range(len(tMatrix)):
                    # Nur wenn eine 1 vorhanden ist und es sich nicht um die selbe Reihe handelt
                    if r1 != r2 and tMatrix[r2][c1] == 1:
                        # Die XOR-Operation muss auf jede Spalte angewendet werden O(m)
                        for c2 in range((len(tMatrix[r2]))):
                            # Anwenden der XOR Operation bzw. Addition in Z2
                            tMatrix[r2][c2] = (tMatrix[r2][c2] +
                                               tMatrix[r1][c2]) % 2
                # Dieser Prozess soll nur für die erste Eins in der Reihe durchgeführt werden
                break

    # Dictionary für unabhängige Variablen
    #   key: Index der Spalte bzw. Karte der Variable
    #   value: Liste an Indizes der Karten, welche von der Variable beeinflusst werden
    variablen = {}
    # Iteration über alle Reihen der transponierten Matirx
    for i in range(len(tMatrix)):
        # Setzen des Indexes auf die Spalte, in welcher die erste 1 der Reihe vorkommt
        index = None
        # Iteration über alle (bis auf die letzte => ist überall 0) Spalte
        for j in range(len(tMatrix[i])-1):
            if tMatrix[i][j] == 1:
                # Überprüfung, ob es sich um die erste Eins handelt
                if index is None:
                    # Setzen des Indexes für die erste Eins
                    index = j
                else:
                    # Es handelt sich nicht um die erste Eins
                    # => Es ist eine unabhängige Variable
                    # Überprüfung ob diese Variable bereits aufgenommen wurde
                    if j not in variablen.keys():
                        # Hinzufügen einer neuen Variable
                        variablen[j] = [index]
                    else:
                        # Hinzufügen des Indexes in die beeinflussten Variablen
                        variablen[j].append(index)

    # Liste von Bitstrings der Länge n für jede Variable
    # => 1, wenn die Karte i von der Variable beeinflusst wird
    processedVariablen = []
    # Iteration über alle Variablen
    for variable in variablen.keys():
        processedVariablen.append("")
        # Iteration über alle Karten
        for i in range(n):
            # Hinzufügen einer "1", wenn die Karte von der Variable beeinflusst wird
            if i in variablen[variable] or i == variable:
                processedVariablen[-1] += "1"
            else:
                # Andernfalls muss ein "0" eingefügt werden
                processedVariablen[-1] += "0"
    # Brute forcen der Variablen Kombination
    lösung = bfVars(k, processedVariablen, 0, [0,"0"], n)
    if lösung is None:
```

```
        return [[]]
    # Umwandeln des Bitstrings in die Indizes der Karten
    processedLösung = []
    # Iteration über die Zeichen des Bitstrings
    for i in range(len(lösung)):
        # Wenn es sich um eine 1 handelt, muss der Index gewählt werden
        if lösung[i] == "1":
            processedLösung.append(i)
    # Zurückgeben der Indizes
    return [processedLösung]

# Funktion zum Lesen des Inputs
def parseInput():
    # Überprüfung ob kein Argument angegeben wurde
    if(len(argv) == 1):
        # Fragen nach Eingabedatei
        file = input("Eingabedatei␣eingeben:")
    else:
        # Lesen der Eingabedatei aus den Argumenten
        file = argv[1]
    # Überprüfung ob die Eingabedatei existiert
    if(not exists(file)):
        # Ausgabe eines Fehlers
        print("\033[1;31mDatei␣nicht␣gefunden\033[0m")
        return None
    # Öffnen der Eingabedatei (im Lesemodus)
    with open(file=file, mode="r") as data:
        # Lesen aller Zeilen der Eingabedatei
        inhalt = data.readlines()
        # Bereinigen der Zeilen (Zeilenumbrüche entfernen)
        inhalt = [i.replace("\n", "") for i in inhalt]
        # Gesamtzahl an Karten auslesen
        n = int(inhalt[0].split("␣")[0])
        # Anzahl Öffnungskarten auslesen
        k = int(inhalt[0].split("␣")[1])
        # Anzahl der Bits auslesen
        m = int(inhalt[0].split("␣")[2])
        # Karten auslesen
        karten = inhalt[1:]
        # Zurückgeben der gelesenen Daten (...und File, für Benennung der Ergebnisdatei)
        return n, k, m, karten, file

# Hauptfunktion (Ausführung des Programms)
def main():
    # Sicheres Lesen des Inputs
    try:
        n, k, m, karten, file = parseInput()
    except Exception as e:
        # Ausgabe des Fehlers
        print("Input␣konnte␣nicht␣gelesen␣werden:␣{}".format(e))
        return
    # Anwenden des Gauss-Algorithmus auf den gelsenen Input
    # => Übergeben von k+1, da die Gesamtzahl an XOR-Karten = Öffnungskarten + Sicherungskarten
    lösungen = (bfAll(n, k+1, m, karten) if
    comb(n,ceil((k+1)/2)) < (0 if m > n else comb(n-m,k+1)) else gaussElim(n, k+1, m, karten))
    # Schreiben der Lösungsdatei
    with open("ergebnis_" + file, "w") as f:
        # Iteration über alle Lösungen
        for lösung in lösungen:
            # Itearation über alle Kartenindizes der Lösung
            for index in lösung:
                # Ausgabe der Lösung
                print(karten[index])
                # Schreiben der Kartenwerte in die Datei
                f.write(karten[index] + "\n")
            # Trennung der Lösungen über ein \n
            f.write("\n")
            print()
```

6 Entsperren der Häuser (Aufgabenteil b)

6.1 Strategie für das Entsperren eines Hauses

Sobald alle $k+1$ Karten gefunden wurden, besteht die Problematik, inwiefern die k Öffnungskarten herausgefiltert und den richtigen Häusern zugeordnet werden können. Wie bereits in Unterabschnitt 1.1 erläutert, lässt sich über die *XOR*-Operation kein weiterer Unterschied zwischen der Sicherungskarte s und den Öffnungskarten w_i ausmachen. Aus diesem Grund muss direkt zum Ausprobieren übergegangen werden.
Eine Eigenschaft, welche dabei relevant wird, ist die Sortierung der Öffnungskarten w_i: „[...] sortierte sie anschließend aufsteigend als $w_1, ..., w_{[n]}$ und codierte die Schlösser der [...] Häuser, wobei Haus [i] das Codewort [w_i] erhielt.".
Nach der Annahme, dass die Reihenfolge der Häuser weiterhin bekannt ist, kann nun ein beliebiges Haus i in jedem Fall mit maximal zwei Fehlversuchen entsperrt werden.
Zunächst müssen dafür die ermittelten $k+1$ Karten nach ihrem Wert im binären Zahlensystem sortiert werden $B' = \langle B \rangle$. Weiterhin ist die Sicherungskarte s mitenthalten. Für das Haus i wird nun die i-te Karte $b_i \in B'$ aus den sortierten Karten ausprobiert. Sollte der Versuch fehlschlagen, ist klar, dass sich die Sicherungskarte unter den Karten $b_1 \to b_i$ befindet. Die Öffnungskarte für Haus i ist nun aufgrund der Verschiebung garantiert b_{i+1}. Somit kann ein beliebiges Haus, mit bekanntem Index i, sogar mit maximal einem Fehlversuch entsperrt werden.
Beispiel:

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}
w_1	w_2	w_3	s	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}

Tabelle 4: Wenn bei dieser Kartenbelegung bei Haus 5 die Karte b_5 ausprobiert wird, ist der erste Versuch fehlgeschlagen, da $b_5 \equiv w_4$. Der Informationsgewinn bei diesem Fehlversuch besteht darin, dass klar ist, dass sich die Sicherungskarte s unter den Karten b_1, b_2, b_3, b_4, b_5 befindet. Daraus lässt sich ableiten, dass mindestens die Karten $w_5, w_6, w_7, w_8, w_9, w_{10}$ um eins verschoben sind. Der nächste Versuch für Haus 5 mit b_6 wird somit erfolgreich sein.

Dieses Verfahren kann auf eine beliebige Anzahl an Karten ausgeweitet werden, da in jedem Fall genau eine Sicherungskarte s vorhanden ist.

6.2 Strategie für das Finden der Sicherungskarte

Da nach dem Entsperren eines Hauses womöglich immer noch nicht bekannt ist, wo sich die Sicherungskarte befindet, ist es womöglich nützlich diese so schnell wie möglich zu finden, um sich wieder an einem sicheren Ort zu lagern. Um optimal vorzugehen ist das Prinzip einer *binären Suche* anzuwenden. Mit jedem Haus i, welche erfolgreich mit Karte b_i entsperrt wird, geht ein bestimmter Informationsgewinn einher. Man weiß, dass der Index der Sicherungskarte größer ist als i, da andernfalls b_i nicht erfolgreich Haus i entsperrt hätte.
Man fange bei Haus bzw. Karte $\lfloor \frac{k+1}{2} \rfloor$ an, um den Informationgewinn beim Entsperren zu maximieren. Der Informationgewinn beträgt $\lceil \frac{k+1}{2} \rceil$, da nach dem Ausprobieren bekannt ist, in welcher Hälfte der Karten sich die Sicherungskarte s befindet (Bei Fehlschlagen: links, andernfalls rechts). Folglich halbiert man die Hälfte, in welcher sich s befindet usw.. Es braucht $\log(k+1)$ Versuche um die Sicherungskarte s zu finden.