Bonusaufgabe: Zara Zackigs Zurückkehr

Teilnahme-ID: 61099

Bearbeiter/-in dieser Aufgabe:
Malte Voos

23. April 2022

Inhaltsverzeichnis

1 Lösungsidee

Gegeben ist eine Menge $C = \{c_1, \ldots, c_n\}$ von n Karten, die aus w von Zara stammenden echten Karten und $n - w$ von den „Freunden“ hinzugefügten Zufallskarten besteht. Jede Karte ist dabei als Bit-Vektor von b Bits zu verstehen. Gesucht ist die Untermenge $R \subseteq C$ der w echten Karten, sodass eine Karte der Untermenge (die Sicherungskarte) genau das exklusive Oder der restlichen Karten der Untermenge (der Öffnungskarten) ist. Es wird angenommen, dass diese Untermenge für jede gegebene Instanz des Problems eindeutig ist.

Bei einer solchen Untermenge kann aber jede Karte als Sicherungskarte betrachtet werden, da jede Karte das exklusive Oder der jeweils restlichen Karten ist. Es ist also geschickter, die Untermenge R der echten Karten dadurch zu charakterisieren, dass das exklusive Oder aller Karten in R 0 ist.

R kann auch durch einen Lösungsvektor $s = \begin{pmatrix} s_1 & s_2 & \cdots & s_n \end{pmatrix}^\top$ von n Bits und mit Hamming-Gewicht w ausgedrückt werden, wobei s_n genau dann 1 ist, wenn c_n eine echte Karte ist:

$$R = \{c_n \mid s_n = 1\} \tag{1}$$

Damit lässt sich folgendes Gleichungssystem aufstellen:

$$\begin{gathered}(c_{11} \wedge s_1) \oplus (c_{21} \wedge s_2) \oplus \cdots \oplus (c_{n1} \wedge s_n) = 0 \\ (c_{12} \wedge s_1) \oplus (c_{22} \wedge s_2) \oplus \cdots \oplus (c_{n2} \wedge s_n) = 0 \\ \vdots \\ (c_{1m} \wedge s_1) \oplus (c_{2m} \wedge s_2) \oplus \cdots \oplus (c_{nm} \wedge s_n) = 0\end{gathered}$$

c_{nm} ist dabei das m-te Bit der Karte c_n, $\oplus$ ist das exklusive Oder, und $\wedge$ ist die logische Konjunktion. Für jedes Bit der Karten wird eine Gleichung aufgestellt, wobei das Bit der Karte c_n immer nur dann eingerechnet wird, wenn $s_n = 1$ ist, also c_n eine echte Karte ist.

Bei diesem Gleichungssystem handelt es sich sogar um ein lineares Gleichungssystem, da die Bits $\{0, 1\}$ zusammen mit dem exklusiven Oder $\oplus$ als Addition und der logischen Konjunktion $\wedge$ als Multiplikation den endlichen Körper $\mathbb{F}_2$ bilden. In Matrixschreibweise lässt sich das obige Gleichungssystem also als

$$\mathbf{H}s = 0 \tag{2}$$

schreiben, wobei $\mathbf{H}$ die Matrix ist, die in der n-ten Spalte die Bits der Karte c_n enthält.

$\mathbf{H}$ lässt sich auch als Kontrollmatrix eines linearen Codes über $\mathbb{F}_2$ betrachten: Die Codewörter dieses Codes sind genau die s, für die Gleichung 2 gilt. Ein Codewort repräsentiert also eine Auswahl von Karten, die XOR einander 0 ergeben.

Gesucht ist aber nicht ein beliebiges Codewort, sondern ein Codewort mit dem Hamming-Gewicht w, da genau w Karten von Zara stammen. Damit lässt sich Zara Zackigs Problem auf folgendes reduzieren:

Problem 1 *Gegeben ist die Kontrollmatrix* $\mathbf{H}$ *eines linearen Binärcodes und ein positive ganze Zahl* w*. Der durch* $\mathbf{H}$ *definierte Code enthält ein Codewort des Hamming-Gewichts* w*. Wie lautet dieses Codewort?*

Ein verwandtes Problem lautet wie folgt:

Problem 2 *Gegeben ist die Kontrollmatrix* $\mathbf{H}$ *eines linearen Binärcodes und ein positive ganze Zahl* w*. Enthält der durch* $\mathbf{H}$ *definierte Code ein Codewort des Hamming-Gewichts* w*?*

Dieses Entscheidungsproblem ist in der Literatur als Subspace Weights bekannt und bewiesenermaßen NP-vollständig [1]. Das zugehörige Berechnungsproblem, ein Codewort mit Hamming-Gewicht w zu finden und eine Fehlermeldung auszugeben, falls keines existiert, gehört zur Komplexitätsklasse FNP und ist offensichtlich NP-schwer, da eine Lösung gleichzeitig auch eine Lösung für Subspace Weights ist. Zara Zackigs Problem (Problem 1) aber ist etwas einfacher: Es ist garantiert, dass ein Codewort des gegebenen Hamming-Gewichts existiert. Damit gehört es zur Komplexitätsklasse TFNP. Obwohl vermutet wird, dass TFNP keine NP-schweren Probleme enthält [2], wurde für viele Probleme in TFNP, wie zum Beispiel für das Faktorisierungsproblem für ganze Zahlen, bis jetzt kein in Polynomialzeit laufender Algorithmus entdeckt. Die Verwandtschaft zu Problem 2 legt auch für Problem 1 die Vermutung nahe, dass kein Algorithmus die Lösung in Polynomialzeit ermitteln kann.

Das bedeutet aber nicht, dass das Problem nicht effizienter als mit der Brute-Force-Methode gelöst werden kann. Im Folgenden wird zunächst ein einfacher Brute-Force-Algorithmus und dann ein effizienterer Las-Vegas-Algorithmus vorgestellt, und im Anschluss werden die beiden Algorithmen hinsichtlich ihrer Rechenkomplexität verglichen.

1.1 Brute-Force-Algorithmus

Es wird über alle $\binom{n}{w}$ Möglichkeiten, aus den n Karten w auszuwählen, iteriert, und jeweils überprüft, ob es sich um die echten Karten handelt (ob das exklusive Oder der ausgewählten Karten 0 ist). In diesem Fall ist die Lösung gefunden.

1.1.1 Rechenkomplexität

Die echten Karten werden im Durchschnitt nach $\frac{1}{2}\binom{n}{w}$ Versuchen gefunden. Bei jedem Versuch wird $(w-1)b$ das exklusive Oder zweier Bits gebildet, und anschließend werden b Bits mit 0 verglichen. Ein Versuch benötigt demnach ungefähr $(w-1)b + b = wb$ Bit-Operationen, und die durchschnittliche Bit-Komplexität des Brute-Force-Algorithmus liegt näherungsweise bei

$$C_{BF} = \binom{n}{w}\frac{wb}{2} \tag{3}$$

Bit-Operationen.

1.2 Lee-Brickell-Algorithmus

Der Lee-Brickell-Algorithmus [4] ist einer von vielen Algorithmen für Problem 1, die nach dem Prinzip des *information set decoding* (ISD) arbeiten. Die Grundidee hinter ISD ist, dass es einfacher ist, eine

bestimmte Verteilung der Einsen im gesuchten Codewort richtig zu erraten, als alle Bits des Codewortes richtig zu erraten (wie es der Brute-Force-Algorithmus tut). Die genaue Verteilung ist dabei von Algorithmus zu Algorithmus unterschiedlich. Gemeinsam ist den ISD-Algorithmen aber, dass sie alle Las-Vegas-Algorithmen sind, d.h., dass sie ausgehend von einer Zufallsvariable die Lösung mit einer bestimmten Wahrscheinlichkeit finden oder anderenfalls eine Fehlermeldung zurückgeben. Dieses Vorgehen wird so oft in unabhängigen Iterationen wiederholt, bis die Lösung gefunden ist.

Eine Iteration des Lee-Brickell-Algorithmus läuft wie folgt ab:

1. Eine zufällige Permutation $\mathbf{P}$ von n Elementen wird generiert und ihre umgekehrte Permutation $\mathbf{P}^{-1}$ wird berechnet.

2. Die zufällige Permutation $\mathbf{P}$ wird auf die Spalten der Kontrollmatrix $\mathbf{H}$ angewandt.

3. Die permutierte Kontrollmatrix $\mathbf{PH}$ wird mithilfe des Gauß-Jordan-Algorithmus in die reduzierte Stufenform gebracht. Dabei werden die Spaltenindizes der Basisvariablen bzw. freien Variablen in zwei Listen gespeichert. Hier ist anzumerken, dass die Zuteilung der Spalten zu Basisvariablen und freien Variablen stark von der gewählten Permutation $\mathbf{P}$ abhängt: Je weiter links eine Spalte in der permutierten Kontrollmatrix liegt, desto eher gehört die Spalte zu einer Basisvariable in der reduzierten Stufenform.

4. Nun wird angenommen, dass bei dem ebenfalls permutierten gesuchten Codewort $\mathbf{P}s$ mit Hamming-Gewicht w genau p Einsen zu den k freien Variablen, und genau $w-p$ Einsen zu den $n-k$ Basisvariablen gehören. Ob diese Annahme stimmt, hängt davon ab, ob in Schritt 1 eine glückliche Permutation gewählt wurde. $p \in \mathbb{Z}^+$ ist ein Parameter des Lee-Brickell-Algorithmus, für den offensichtlich ein paar Einschränkungen gelten:

$$\begin{gathered} p \leq w \\ p \leq k \\ w - p \leq n - k \iff p \geq -n + k + w \end{gathered}$$

5. Um festzustellen, welche p freien Variablen in $\mathbf{P}s$ den Wert 1 annehmen, wird über alle $\binom{k}{p}$ Möglichkeiten, aus den k freien Variablen p auszuwählen, iteriert.
   a) Zunächst wird das exklusive Oder x der zu den p ausgewählten freien Variablen gehörenden Spalten errechnet. Aus der Struktur der reduzierten Stufenform folgt, dass genau in den Zeilen, in denen x den Wert 1 hat, auch die Basisvariable der Zeile den Wert 1 haben muss, damit jede Zeile aufsummiert 0 ergibt und Gleichung 2 erfüllt ist.
   b) Es wird überprüft, ob x ein Hamming-Gewicht von $w-p$ hat. In diesem Fall haben nämlich genau $w-p$ Basisvariablen in $\mathbf{P}s$ den Wert 1, und die Annahme aus Schritt 4 hat sich bewahrheitet. Im permutierten Lösungsvektor $\mathbf{P}s$ haben dann von den freien Variablen die ausgewählten p den Wert 1, und von den Basisvariablen die der $w-p$ Zeilen, in denen x eine 1 hat. Damit ergibt sich die Menge R der echten Karten wie folgt:
      i. Der Lösungsvektor s kann durch Anwendung der umgekehrten Permutation $\mathbf{P}^{-1}$ auf den permutierten Lösungsvektor $\mathbf{P}s$ rekonstruiert werden: $s = \mathbf{P}^{-1}\mathbf{P}s$
      ii. Die Menge der echten Karten R ist gegeben durch $R = \{c_n \mid s_n = 1\}$. Damit ist die Lösung gefunden.

6. Falls in Schritt 5 die Lösung nicht gefunden wurde, ist die Annahme aus Schritt 4 falsch. Damit ist die Iteration fehlgeschlagen.

1.2.1 Rechenkomplexität

Die durchschnittliche Rechenkomplexität C_{LB} des Lee-Brickell-Algorithmus setzt sich aus der Komplexität einer einzelnen Iteration C_{LB1} und der zu erwartenden Anzahl an Iterationen N_{LB} zusammen. Letztere kann auch durch den Kehrwert der Erfolgswahrscheinlichkeit einer einzelnen Iteration P_{LB1} ausgedrückt werden:

$$C_{LB} = C_{LB1} \cdot N_{LB} = \frac{C_{LB1}}{P_{LB1}} \tag{4}$$

Die beiden ressourcenintensiven Schritte einer einzelnen Iteration sind erstens die Berechnung der reduzierten Stufenform mittels des Gauß-Jordan-Algorithmus und zweitens das Durchgehen der $\binom{k}{p}$ Möglichkeiten für die Position der p Einsen unter den k freien Variablen.

Der Gauß-Jordan-Algorithmus transformiert die Matrix zunächst in die „gewöhnliche“ (nicht-reduzierte) Stufenform. Dafür wird gleichzeitig „diagonal“ über die Zeilen und Spalten der Matrix iteriert, bis die letzte Zeile bzw. Spalte bearbeitet wurde. In den allermeisten nicht-trivialen Fällen bleiben auf der rechten Seite noch mehrere zu freien Variablen gehörende Spalten übrig, sodass diese äußere Schleife dann b-mal (b ist die Anzahl der Zeilen) durchlaufen wird. In jedem Durchlauf sucht der Algorithmus in der derzeitigen Spalte ein Pivotelement (eine 1), welches b Bit-Vergleiche im schlimmsten Fall benötigt. Falls ein Pivotelement gefunden wurde, wird die entsprechende Zeile nach oben gebracht, und die weiter unten liegenden Einsen der Spalte (nie mehr als b) werden eliminiert, indem die entsprechenden Zeilen mit der Zeile des Pivotelements XOR gerechnet werden. Für eine Zeile benötigt dies n Bit-Operationen. Die Berechnung der „gewöhnlichen“ Stufenform benötigt also grob $b(b + bn) = b^2(n + 1)$ Bit-Operationen.

Bei dieser Annäherung wurden aber tatsächlich zu viele Bit-Operationen eingerechnet, da nicht berücksichtigt wurde, dass für weiter rechts liegende Spalten weniger weiter unten liegende Einsen eliminiert werden müssen. Dies wird aber dadurch ausgeglichen, dass für die Berechnung der reduzierten Stufenform auch die Einsen über den Pivotelementen eliminiert werden müssen. Für die Berechnung der reduzierten Stufenform ergibt sich also eine Bit-Komplexität von ungefähr $b^2(n + 1)$ Bit-Operationen.

Um die Position der p Einsen unter den k freien Variablen zu ermitteln, wird über alle $\binom{k}{p}$ Möglichkeiten iteriert. Es wird jeweils das exklusive Oder x der ausgewählten Spalten berechnet und überprüft, ob dies ein Hamming-Gewicht von $w - p$ hat. Die Berechnung von x benötigt $(p - 1)b$ Bit-Operationen, und für die Überprüfung des Hamming-Gewichts müssen nochmals b Bits betrachtet werden. Der Schritt erfordert also ingesamt näherungsweise $\binom{k}{p}((p - 1)b) + b) = \binom{k}{p}pb$ Bit-Operationen.

Damit ergibt sich die Bit-Komplexität einer einzelnen Iteration zu

$$C_{LB1} = b^2(n + 1) + \binom{k}{p}pb = b[b(n + 1) + \binom{k}{p}p] \tag{5}$$

Eine Iteration ist genau dann erfolgreich, wenn durch die zufällig gewählte Permutation p von den k freien Variablen und $w - p$ von den $n - k$ Basisvariablen im Lösungsvektor den Wert 1 annehmen. Die Wahrscheinlichkeit, dass die w Einsen im Lösungsvektor von n Elementen auf diese Weise verteilt sind, wenn die Zuteilung zu freien Variablen und Basisvariablen komplett zufällig ist, liegt bei

$$P_{LB1} = \frac{\binom{k}{p}\binom{n-k}{w-p}}{\binom{n}{w}} \tag{6}$$

Zugegebenermaßen lässt sich diese nicht direkt mit der Wahrscheinlichkeit gleichsetzen, eine glückliche Permutation zu wählen, da bei der Berechnung der Stufenform vereinzelt Spalten ohne Pivotelement übersprungen und somit zu freien Variablen gezählt werden, obwohl noch weitere Basisvariablen folgen. Die Zuteilung zu freien Variablen und Basisvariablen ist demnach nicht komplett zufällig. Hier wird jedoch angenommen, dass dies keinen wesentlichen Einfluss auf die Erfolgswahrscheinlichkeit hat und somit wird (6) als Erfolgswahrscheinlichkeit verwendet.

Mit (4), (5) und (6) ergibt sich die durchschnittliche Rechenkomplexität des Lee-Brickell-Algorithmus zu näherungsweise

$$C_{LB} = \frac{C_{LB1}}{P_{LB1}} = \frac{\binom{n}{w}b[b(n + 1) + \binom{k}{p}p]}{\binom{k}{p}\binom{n-k}{w-p}} \tag{7}$$

Bit-Operationen.

1.2.2 Wahl des Parameters p

Bei der Wahl des Parameters p muss zwischen Ressourcenverbrauch und Erfolgswahrscheinlichkeit einer Iteration abgewogen werden: Bei kleinem p gibt es weniger Möglichkeiten für die Positionen der p Einsen unter den k freien Variablen, und somit benötigt eine einzelne Iteration weniger Bit-Operationen bzw. Zeit. Andererseits kann ein größeres p die Erfolgswahrscheinlichkeit einer einzelnen Iteration erhöhen.

$p = 2$ scheint in diesem Anwendungsfall ein gutes Gleichgewicht zu schaffen, da für alle gegebenen Beispieleingaben die theoretische Bit-Komplexität C_{LB} bei $p = 2$ minimal ist. Auch bei empirischen Messungen benötigte die Implementierung des Lee-Brickell-Algorithmus stets bei $p = 2$ durchschnittlich am wenigsten Zeit.

					$p = 1$	$p = 2$	$p = 3$	$p = 4$
Eingabedatei	n	w	b	k	C_{LB} ∅Zeit	C_{LB} ∅Zeit	C_{LB} ∅Zeit	C_{LB} ∅Zeit
`stapel3.txt`	161	11	128	33	$1{,}18 \cdot 10^7$ 32 ms	$9{,}22 \cdot 10^6$ 30 ms	$2{,}02 \cdot 10^7$ 56 ms	$2{,}03 \cdot 10^8$ 254 ms
`stapel4.txt`	181	11	128	53	$3{,}12 \cdot 10^7$ 125 ms	$1{,}59 \cdot 10^7$ 77 ms	$4{,}49 \cdot 10^7$ 88 ms	$6{,}93 \cdot 10^8$ 717 ms
`stapel5.txt`	200	5	64	136	$2{,}44 \cdot 10^7$ 53 ms	$1{,}32 \cdot 10^7$ 22 ms	$2{,}44 \cdot 10^8$ 263 ms	$1{,}01 \cdot 10^{10}$ 12 s

Tabelle 1: Theoretische Bit-Komplexität und durchschnittliche Laufzeit für verschiedene p

Die Beispieleingaben `stapel0.txt` bis `stapel2.txt` sind hier uninteressant, da sie mit nur einer freien Variable ($k = 1$) gewissermaßen trivial lösbar sind und p zwangsläufig gleich 1 ist.

1.3 Vergleich der beiden Algorithmen

Die obige Analyse der Rechenkomplexität hat ergeben, dass weder der Brute-Force-Algorithmus noch der Lee-Brickell-Algorithmus das Problem in Polynomialzeit lösen können. Dennoch lässt sich feststellen, dass der Lee-Brickell-Algorithmus wesentlich effizienter als der Brute-Force-Algorithmus arbeitet: Im Durchschnitt benötigt ersterer circa

$$\frac{C_{BF}}{C_{LB}} = \frac{w\binom{k}{p}\binom{n-k}{w-p}}{2[b(n+1) + \binom{k}{p}p]} \tag{8}$$

-mal weniger Bit-Operationen als letzterer. Dieser Quotient liegt für die Beispieleingaben `stapel0.txt` bis `stapel5.txt` mit $p = min(2, k)$ jeweils in der Größenordnung von 10^1, 10^2, 10^{10}, 10^{12}, 10^{12} und 10^4.

Den größten Vorteil gegenüber dem Brute-Force-Algorithmus hat der Lee-Brickell-Algorithmus bei Eingaben mit vielen Karten und vielen Bits pro Karte (wenigen freien Variablen), da bei solchen Eingaben durch die Anwendung des Gauß-Verfahrens am meisten Arbeit eingespart wird.

2 Umsetzung

Das Programm wurde in der Programmiersprache Rust geschrieben und enthält im Kern eine Implementierung des in Abschnitt 1.2 beschriebenen Lee-Brickell-Algorithmus. Zusätzlich wurden die Programmbibliotheken *bitvec* [5], welche eine platzsparende Datenstruktur für Bit-Vektoren bereitstellt, und *rand* [3], welche das Arbeiten mit Zufallszahlen erleichtert, verwendet.

Der Einstiegspunkt für den eigentlichen Algorithmus ist die Funktion `solve_task`, welche eine Aufgabe (Datentyp `Task`), bestehend aus dem Kartenstapel und der Anzahl von Zaras Öffnungskarten, erhält und die Lösung (Datentyp `Solution`), bestehend aus Zaras echten Karten, zurückgibt. Die Funktion `lee_brickell_iteration` implementiert eine Iteration des Lee-Brickell-Algorithmus und gibt nur mit einer bestimmten Wahrscheinlichkeit die Lösung zurück; `solve_task` ruft `lee_brickell_iteration` also wiederholt auf, bis die Lösung gefunden ist.

In `lee_brickell_iteration` wird die Matrix zusätzlich zweimal transponiert. Das hat den Vorteil, dass bei dem Gauß-Verfahren die Zeilen der Matrix zusammenhängend im Speicher liegen und so schneller miteinander XOR gerechnet (addiert) werden können. Später wird die Matrix nochmals transponiert, damit bei der Berechnung von x in Schritt 5a) wieder die Spalten zusammenhängend im Speicher liegen.

Ansonsten entspricht die Implementierung ziemlich direkt dem in Abschnitt 1.2 beschriebenen Lee-Brickell-Algorithmus. Weitere Details zur Implementierung sind im angehängten, ausgiebig kommentierten Quellcode zu finden.

3 Beispiele

Bei einigen Beispielen wurde die Programmausgabe am Seitenrand abgeschnitten, um pro Zeile die Bits einer Karte darstellen zu können. Die vollständigen Ausgaben sind im Order „Beispielausgaben“ zu finden.

Zunächst die Beispiele von der BwInf-Webseite:

```
$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel0.txt
Randinformationen (siehe Dokumentation):
n = 20; w = 5; b = 32; k = 1; p = 1

Echte Karten:
00111101010111000110100110011001
10101100111111011010100011100000
10111000011001110000101010111110
11010111111010111101101111110000
11111110001011010001000000110111

Laufzeit ohne I/O: 279.946µs

$ AusführbaresProgramm bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel1.txt
Randinformationen (siehe Dokumentation):
n = 20; w = 9; b = 32; k = 1; p = 1

Echte Karten:
00010001110100110001111101100100
00100000111100111110111101111100
00100011100111011010111011100011
00110100001010100100001111010010
00110110000110101101011111111010
11000111111010110100000101110100
11010011010110110101001101010111
11110011101011001001000010111110
11110111100100010100100001001110

Laufzeit ohne I/O: 287.396µs

$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel2.txt
Randinformationen (siehe Dokumentation):
n = 111; w = 11; b = 128; k = 1; p = 1

Echte Karten:
00101000011000010010111011101011011010111000100100110101111011011110111100101100001001110010101
00101011111000101011010110111110010011000000000001101001100111101100101100100001000110101011011
01101001001011000101001111111110101100000100010110011101010010101101100010000000110000110001101
01101011101000110111010001100001110000011000110101100010111011100110011011110111011110011010110
01111011001111000111001111000110110111010010100000010000010110000101000111010100000001101001100
10000000000100100110011001000110000000000010101011010010010000100011101011011010101001010100010
10101011000001101100000101111111111001100011001110010101101111101100011111111011111010001111110
10101111110010010010100111101100010011111000010101001100100001111000100010010010011010010101010
11000011000100110111000101100100101110101011001101101010110100100001111010001000100101000011001
11011110000101001101111100110000111010011011101111010111110110111011010001001101101100111010
11101110101011100111101111000111001101111011101010101111110001101000110100011000001111010000100

Laufzeit ohne I/O: 10.394463ms

$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel3.txt
Randinformationen (siehe Dokumentation):
n = 161; w = 11; b = 128; k = 33; p = 2
```

```
Echte Karten:
00100000111001110001101010001111110011110001011110100110010101100010011000100111101011001111000
01010000101101110011110001110011010011001111111000001000000100000010111100001110100001001001110
01110001111110101000010011100011111111100111101101110001010100010110001100010101000010100010100
01110110100111001000011011100100101010111110010111100000010110011011001011001110010111000000100
01111101100010101100110011101011010101001101000110011111101010110000000111000110101001111111110
10110000001001100110110101000100110011100101100110101111011111101000001110100011000001100000010
10110111010010111110110011000101011101000011111100001000001100111111111001001100011100011110010
10111000101111100010111110101010101100110001000011011001100010110110000001100001101111010100000
10111111010101001100000101101100111101000010100010100100001001111010110100100101100011101100010
11000001011001000110100111011111111011111011011010111110100010110000101111000111001010100101100
11001011010111111110111010001000100100000101100111010100111110100000100111110001110001011000000
```

```
Laufzeit ohne I/O: 45.019015ms
```

```
$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel4.txt
Randinformationen (siehe Dokumentation):
n = 181; w = 11; b = 128; k = 53; p = 2
```

```
Echte Karten:
00000111011010010101101110001111101001110010100011000001000001101110101111101001000100000001111
00101100000111101111000010000001011011111110000110011111111110001111000000101111101100010010000
00101100001110001110011110000100110000000000011101101100111010001010000101000011011001000011111
00110110010111001001001111001111101010011100000100000001100010101000111001000100111000100110111
01000011001001101011001111011101111010010110111010111110111011111100100010001011010110010111111
10000001000101110011010100011000110100110100111100100010001000011011000001010111100110111011100
10001100001011001101001011000110111010100110100001000010110101010110011011011111101011001100100
10101010000011111111001111011110001000100111010000010010111101000001010001100010110011111011111
11000101110001001000001011101000100110011001111011101001011010110100111000100001000100000011000
11100010000000111101001111111001001100111011101001000111001111001111000010000101100001000011000
11110010110001100010100110001001110001111010011100100011010100101001000100111100101000001000010
```

```
Laufzeit ohne I/O: 63.322147ms
```

```
$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/stapel5.txt
Randinformationen (siehe Dokumentation):
n = 200; w = 5; b = 64; k = 136; p = 2
```

```
Echte Karten:
01011111110001110000001011111000101110101101010001000000011001000
10000100111010100011111001001101100110111001010101000010000001001
10100001101011001011101110011000110111101111110101110001011111110
10101110110011001001100011001100010111010010000000110111111100100
11010100010011010001111111100001101000101001110001000010010110110
```

```
Laufzeit ohne I/O: 21.107732ms
```

Bei dem Beispiel `wahre_freunde.txt` haben Zaras Freunde keine Zufallskarten hinzugefügt:

```
$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/wahre_freunde.txt
Randinformationen (siehe Dokumentation):
n = 9; w = 9; b = 32; k = 1; p = 1
```

```
Echte Karten:
000100011101001100011111101100100
001000001111001111110111101111100
001000111001110110101110111100011
001101000010101001000011110100 10
```

```
00110110000110101101011111111010
11000111111010110100000101110100
11010011010110110101001101010111
11110011101011001001000010111110
11110111100100010100100001001110

Laufzeit ohne I/O: 97.26µs
```

Das Beispiel `nur_ein_haus.txt` deckt den Fall ab, dass Zara nur ein Ferienhaus besitzt:

```
$ AusführbaresProgramm/bonusaufgabe-x86_64-linux-static Beispieleingaben/nur_ein_haus.txt
Randinformationen (siehe Dokumentation):
n = 20; w = 2; b = 32; k = 1; p = 1

Echte Karten:
00010001110100110001111101100100
00010001110100110001111101100100

Laufzeit ohne I/O: 259.803µs
```

4 Quellcode

Teile des Quellcodes, die nur der Ein- bzw. Ausgabe dienen, werden hier nicht abgedruckt.

```rust
type Card = BitVec;

// Typ, der eine zu lösende Aufgabe beschreibt.
struct Task {
    cards: Vec<Card>,
    num_pass_cards: usize,
}

// Die Lösung enthält die von Zara stammenden echten Karten.
struct Solution {
    real_cards: Vec<Card>,
    // Der Anschaulichkeit halber werden ein paar weitere Randinformationen
    // mit ausgegeben.
    num_cards: usize,
    num_real_cards: usize,
    bits_per_card: usize,
    num_free_vars: usize,
    p: usize,
}

// `solve_task` löst eine gegebene Aufgabe und beinhaltet den
// eigentlichen Algorithmus.
fn solve_task(task: &Task) -> Solution {
    // `num_real_cards` Karten im Stapel stammen von Zara Zackig, nämlich
    // die `num_pass_cards` Öffnungskarten plus eine Sicherungkarte.
    let num_real_cards = task.num_pass_cards + 1;

    // Der Lee-Brickell-Algorithmus erzeugt nur mit einer gewissen
    // Wahrscheinlichkeit eine Lösung. Daher führen wir ihn wiederholt aus,
    // bis eine Lösung gefunden wurde.
    loop {
        match lee_brickell_iteration(&task.cards, num_real_cards) {
            None => continue,
            Some(solution) => break solution,
        }
    }
}

// Parameter für den Lee-Brickell-Algorithmus.
const P: usize = 2;

// `lee_brickell_iteration` beinhaltet eine Iteration des
// Lee-Brickell-Algorithmus. Die Funktion liefert nur mit einer gewissen
// Wahrscheinlichkeit eine Lösung, und gibt anderenfalls `None` zurück.
```

```rust
fn lee_brickell_iteration(cards: &[Card], num_real_cards: usize) -> Option<Solution> {
    let num_cards = cards.len();
    let bits_per_card = cards[0].len();

    // `permutation` ist eine zufällige Permutation der Karten, repräsentiert
    // durch ein Array, in dem jeder Index in `cards` einmal vorkommt.
    let mut permutation = (0..num_cards).into_iter().collect::<Vec<usize>>();
    permutation.shuffle(&mut rand::thread_rng());

    let mut permutation_pairs = permutation
        .iter()
        .cloned()
        .enumerate()
        .map(|(from, to)| (to, from))
        .collect::<Vec<(usize, usize)>>();
    permutation_pairs.sort();

    // `inverse_permutation` ist die zu `permutation` umgekehrte Permutation.
    let inverse_permutation = permutation_pairs
        .into_iter()
        .map(|(_to, from)| from)
        .collect::<Vec<usize>>();

    // `ppcm` steht für "permuted parity-check matrix". Wie in der
    // Dokumentation beschrieben, ist die Matrix, welche die gegebenen Karten
    // als ihre Spalten enthält, Kontrollmatrix eines linearen Codes über den
    // endlichen Körper GF(2). Die Codewörter dieses Codes repräsentieren
    // jeweils eine Auswahl von Karten, die XOR einander null ergeben. Die
    // Lösung des Problems ist gegeben durch ein Codewort dieses Codes mit
    // Hamming-Gewicht `num_real_cards`.
    // `ppcm` ist eine solche Kontrollmatrix (repräsentiert als Iliffe-Vektor),
    // auf dessen Spalten zusätzlich noch die Permutation `permutation`
    // angewandt wurde.
    let mut ppcm = transpose_and_optionally_permute_columns(cards, Some(&permutation));

    // `basic_vars` und `free_vars` enthalten jeweils die Spaltenindizes der
    // Basisvariablen bzw. freien Variablen von `ppcm`.
    let mut basic_vars = Vec::new();
    let mut free_vars = Vec::new();

    // Zunächst wird `ppcm` mittels des Gaußschen Eliminierungsverfahrens
    // in die reduzierte Stufenform gebracht.
    let mut current_row = 0;
    let mut current_col = 0;
    while current_row < bits_per_card && current_col < num_cards {
        // Wir suchen in der derzeitigen Spalte ein Pivotelement (eine 1).
        let pivot_row = match (current_row..bits_per_card).find(|row| ppcm[*row][current_col]) {
            Some(row) => row,
            None => {
                // Wurde kein Pivotelement gefunden, gehört diese Spalte zu
                // einer freien Variable, und es kann zur nächsten Spalte
                // übergegangen werden.
                free_vars.push(current_col);
                current_col += 1;
                continue;
            }
        };
        // Wurde ein Pivotelement gefunden, gehört die Spalte zu einer
        // Basisvariable.
        basic_vars.push(current_col);
        // Die Zeile mit dem Pivotelement wird nach oben gebracht, indem sie
        // mit der derzeitigen Zeile getauscht wird.
        ppcm.swap(current_row, pivot_row);
        // Alle weiter unten liegenden Einsen dieser Spalte werden eliminiert,
        // indem die entsprechenden Zeilen mit der Zeile des Pivotelements XOR
        // gerechnet (addiert) werden.
        for lower_row in (current_row + 1)..bits_per_card {
            if ppcm[lower_row][current_col] {
                let current_row_cloned = ppcm[current_row].clone();
                ppcm[lower_row] ^= current_row_cloned;
            }
        }
        // Es kann zur nächsten Zeile und Spalte übergegangen werden.
```

```
        current_row += 1;
        current_col += 1;
    }
    // Übrig gebliebene Spalten, die nicht mehr betrachtet wurden, da die
    // letzte Zeile der Matrix erreicht wurde, gehören zu freien Variablen.
    free_vars.extend(current_col..num_cards);

    let num_free_vars = free_vars.len();
    let num_basic_vars = basic_vars.len();

    // Sanity Check: Nach dem Rangsatz immer erfüllt
    assert_eq!(num_basic_vars + num_free_vars, num_cards);

    // Hier ist `ppcm` in Stufenform.

    // Um die reduzierte Stufenform zu erreichen, wird noch einmal rückwärts
    // über die Spalten mit Pivotelement iteriert. Die weiter oben liegenden
    // Einsen werden eliminiert, indem die entsprechenden Zeilen mit der Zeile
    // des Pivotelements XOR gerechnet (addiert) werden. Diese Erweiterung
    // wird auch als Gauß-Jordan-Algorithmus bezeichnet.
    for (pivot_row, pivot_col) in basic_vars.iter().enumerate().rev() {
        for upper_row in 0..pivot_row {
            if ppcm[upper_row][*pivot_col] {
                let pivot_row_cloned = ppcm[pivot_row].clone();
                ppcm[upper_row] ^= pivot_row_cloned;
            }
        }
    }

    // Im letzten Schritt wird versucht, aus der Kontrollmatrix `ppcm` ein
    // Codewort mit Hamming-Gewicht `num_real_cards` zu extrahieren, indem
    // angenommen wird, dass in der Lösung genau `p` der freien Variablen
    // den Wert 1 haben. Ob diese Annahme stimmt, hängt davon ab, ob im ersten
    // Schritt eine glückliche Permutation gewählt wurde.

    // In einigen Fällen muss der Parameter `P` des Lee-Brickell-Algorithmus
    // an die Eingabe angepasst werden, z.B. bei sehr leicht lösbaren Aufgaben
    // mit nur einer freien Variable. Details sind in der Dokumentation,
    // Abschnitt 1.2, Schritt 4 zu finden.
    let p = ((P as isize).clamp(
        -(num_cards as isize) + num_free_vars as isize + num_real_cards as isize,
        num_real_cards.min(num_free_vars) as isize,
    )) as usize;

    // Zunächst wird die Matrix transponiert, damit die Bits der Spalten
    // jeweils zusammenhängend im Speicher liegen und somit die Spalten
    // schneller miteinander XOR gerechnet werden können.
    let transposed_ppcm = transpose_and_optionally_permute_columns(&ppcm, None);

    // Um die `p` freien Variablen mit dem Wert 1 zu finden, wird über alle
    // möglichen `p`-Untermengen der freien Variablen iteriert und jeweils
    // angenommen, dass von den freien Variablen nur die in der Untermenge den
    // Wert 1 haben.
    let mut subset_iter = SubsetIterator::new(num_free_vars, p);
    while let Some(subset) = subset_iter.next() {
        // Zunächst wird das exklusive Oder der `p` Spalten in der Untermenge
        // errechnet. Aus der Struktur der reduzierten Stufenform folgt, dass
        // in jeder Zeile, in der `subset_xor` den Wert 1 hat, auch die
        // Basisvariable dieser Zeile den Wert 1 haben muss, damit die Zeile
        // insgesamt den Wert 0 hat (es handelt sich um eine Zeile der
        // Kontrollmatrix!)
        let mut subset_xor = BitVec::<usize, Lsb0>::repeat(false, bits_per_card);
        for free_var_idx in subset {
            subset_xor ^= &transposed_ppcm[free_vars[*free_var_idx]];
        }

        // Damit insgesamt `num_real_cards` Elemente des Lösungsvektors den
        // Wert 1 haben, müssen genau `num_real_cards - p` Basisvariablen den
        // Wert 1 haben, da oben angenommen wurde, dass von den freien
        // Variablen genau `p` den Wert 1 haben.
        if subset_xor.count_ones() == num_real_cards - p {
            // Ist dies der Fall, ist die Lösung gefunden. Die echten Karten
            // werden in `real_cards` gesammelt. Dabei wird die
```

```
            // Spaltenpermutation mittels `inverse_permutation` rückgängig
            // gemacht.
            let mut real_cards = Vec::new();
            for free_var_idx in subset {
                real_cards.push(cards[inverse_permutation[free_vars[*free_var_idx]]].clone());
            }
            for basic_var_row in subset_xor.iter_ones() {
                real_cards.push(cards[inverse_permutation[basic_vars[basic_var_row]]].clone());
            }
            // Der Benutzerfreundlichkeit gegenüber Zara halber werden die
            // echten Karten vor der Ausgabe aufsteigend sortiert.
            real_cards.sort();

            return Some(Solution {
                real_cards,
                num_cards,
                num_real_cards,
                bits_per_card,
                num_free_vars,
                p,
            });
        }
    }

    // Von den freien Variablen hatten nicht genau `p` den Wert 1. Diese
    // Iteration ist fehlgeschlagen, und in der nächten Iteration wird eine
    // andere Spaltenpermutation und somit auch eine andere Auswahl freier
    // Variablen probiert.
    None
}

// Diese Funktion transponiert eine Bit-Matrix und wendet anschließend, falls
// gefordert, auf die transponierte Matrix die gegebene Spaltenpermutation an.
// Durch die Vereinigung dieser beiden Operationen kann unnötiges Herumkopieren
// von Bits vermieden werden.
fn transpose_and_optionally_permute_columns(
    matrix: &[BitVec],
    permutation: Option<&[usize]>,
) -> Vec<BitVec> {
    let orig_num_rows = matrix.len();
    let orig_num_cols = matrix[0].len();

    let mut transposed = vec![BitVec::<usize, Lsb0>::repeat(false, orig_num_rows); orig_num_cols];
    for orig_row_idx in 0..orig_num_rows {
        let new_col_idx = match permutation {
            Some(permutation) => permutation[orig_row_idx],
            None => orig_row_idx,
        };
        for orig_col_idx in 0..orig_num_cols {
            let new_row_idx = orig_col_idx;
            transposed[new_row_idx].set(new_col_idx, matrix[orig_row_idx][orig_col_idx]);
        }
    }
    transposed
}

// Iterator, der über alle "n über k" k-Untermengen einer Menge von n Elementen
// iteriert. Die Elemente der Menge sind die natürlichen Zahlen von 0 bis n-1.
// Die Untermengen werden durch ein aufsteigend sortiertes Array repräsentiert.
struct SubsetIterator {
    n: usize,
    k: usize,
    fresh: bool,
    subset: Vec<usize>,
}

impl SubsetIterator {
    fn new(n: usize, k: usize) -> Self {
        Self {
            n,
            k,
            fresh: true,
            // Zu Anfang enthält die Untermenge die niedrigsten k Zahlen.
```

```
            subset: (0..k).into_iter().collect(),
        }
    }

    fn next(&mut self) -> Option<&[usize]> {
        // Falls dies die erste Iteration ist, geben wir einfach die in "new()"
        // (s.o.) definierte Anfangs-Untermenge zurück.
        if self.fresh {
            self.fresh = false;
            Some(&self.subset)
        } else {
            // `last_index` ist der Index des letzten Elements der Untermenge.
            let last_index = self.k - 1;
            // `index_to_increase` ist der Index des Elements der Untermenge,
            // das durch die nächstgrößere Zahl ersetzt wird.
            // Wir durchsuchen die Untermenge von links nach rechts nach einem
            // erhöhbaren Element, sodass immer das niedrigste Element zuerst
            // erhöht wird.
            let index_to_increase = self
                .subset
                .iter()
                .enumerate()
                .find(|(index, val)| {
                    if *index == last_index {
                        // Falls dies das letzte Element der Untermenge ist,
                        // kann es nurnoch erhöht werden, wenn die
                        // nächtsgrößere Zahl noch Teil der Gesamtmenge ist.
                        self.subset[*index] != self.n - 1
                    } else {
                        // Ansonsten kann das Element erhöht werden, wenn die
                        // nächstgrößere Zahl nicht schon durch ein anderes
                        // Element in der Untermenge repräsentiert ist.
                        self.subset[*index + 1] != **val + 1
                    }
                })? // das '?' gibt sofort `None` zurück, wenn kein erhöhbares
                // Element gefunden wurde und somit schon über alle Untermengen
                // iteriert wurde.
                .0;

            // Das im vorherigen Schritt gefundene Element wird um 1 erhöht,
            // und alle kleineren Elemente werden analog zum "normalen Zählen"
            // auf die kleinstmöglichen Werte gesetzt.
            self.subset[index_to_increase] += 1;
            for lower_idx in 0..index_to_increase {
                self.subset[lower_idx] = lower_idx;
            }

            // Die neu errechnete Untermenge wird zurückgegeben.
            Some(&self.subset)
        }
    }
}

// Einstiegspunkt für das Programm.
fn main() {
    let task_file_name = match env::args().nth(1) {
        Some(x) => x,
        None => {
            eprintln!("Nutzung: bonusaufgabe <dateiname>");
            process::exit(1);
        }
    };
    let task_str = fs::read_to_string(task_file_name).expect("Datei kann nicht gelesen werden");
    let task = Task::try_from(task_str.as_str()).expect("Datei enthält keine gültige Aufgabe");

    let start = time::Instant::now();
    let solution = solve_task(&task);
    let elapsed = start.elapsed();

    println!("{}", solution);
    println!("Laufzeit ohne I/O: {:?}", elapsed);
}
```

5 Teilaufgabe b)

> *Wie kann nun Zara mithilfe der 11 gefundenen Karten am nächsten Wochenende das nächste Haus aufsperren, ohne dafür mehr als zwei Fehlversuche zu benötigen?*

Zara sortiert die gefundenen echten Karten ähnlich wie die Öffnungskarten der Häuser aufsteigend als $r_1, r_2, \ldots, r_w$. Die Öffnungskarte für Haus k ist jetzt entweder r_k oder r_{k+1}, je nachdem, ob sie im sortierten Stapel vor oder hinter der Sicherungskarte liegt.

Wenn Zara also am nächsten Wochenende Haus k besucht, muss sie nur die Karten r_k und r_{k+1} probieren. Mit dieser Strategie benötigt Zara höchstens einen Fehlversuch, um das Haus aufzusperren.

Literatur

[1] E. Berlekamp, R. McEliece und H. van Tilborg. „On the inherent intractability of certain coding problems (Corresp.)“ In: *IEEE Transactions on Information Theory* 24.3 (1978), S. 384–386. DOI: 10.1109/TIT.1978.1055873.

[2] Paul W. Goldberg und Christos H. Papadimitriou. „Towards a Unified Complexity Theory of Total Functions“. In: *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*. Hrsg. von Anna R. Karlin. Bd. 94. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 37:1–37:20. ISBN: 978-3-95977-060-6. DOI: 10.4230/LIPIcs.ITCS.2018.37. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8340.

[3] Diggory Hardy u. a. *rand*. Version 0.8.5. URL: https://github.com/rust-random/rand.

[4] P. J. Lee und E. F. Brickell. „An Observation on the Security of McEliece's Public-Key Cryptosystem“. In: *Advances in Cryptology — EUROCRYPT '88*. Hrsg. von D. Barstow u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, S. 275–280. ISBN: 978-3-540-45961-3.

[5] Alexander Payne u. a. *bitvec*. Version 1.0.0. URL: https://github.com/bitvecto-rs/bitvec.