# Bonusaufgabe: Zara Zackigs Zurückkehr

Teilnahme-ID: 62048

Bearbeiter dieser Aufgabe:
Leandro Conte

16. April 2022

## Inhaltsverzeichnis

## 1 Lösungsidee

### 1.1 Generelle Beschreibung von Problem a)

Gegeben sind N 128-Bit Zahlen $\{a_0, \ldots, a_{N-1}\} = W$, die gegebenen Karten. (Indizes sind 0-basiert)
Gesucht sind k Indizes I, sodass $a_{i_0} \oplus \cdots \oplus a_{i_{k-2}} = a_{i_{k-1}}$ für $i_j \in I$, die Indizes der Karten, die nicht von den Freunden Zaras hinzugefügt wurden. (k ist im Rest der Dokumentation um eins größer als in der Aufgabenstellung!)
Da $x \oplus x = 0$ und $\oplus$ assoziativ und kommutativ ist gilt:

$$a_{i_0} \oplus \cdots \oplus a_{i_{k-2}} = a_{i_{k-1}} \iff a_{i_0} \oplus \cdots \oplus a_{i_k-1} = 0 \tag{1}$$

Man kann also die Sicherungskarte nicht von den Karten unterscheiden, aus denen sie zusammengestellt wurde.

Es seien:

$$Z := \text{Indizes in } [0; N)$$

$$\text{Für eine Indexmenge z: } xor(z) := \bigoplus_{x \in z} a_x$$

Es ist also die Menge $I$ gesucht, für die gilt $|I| = k$ und $xor(I) = 0$ Das Verfahren zum Finden der Menge I wird abgesehen vom nächsten Block im Rest der Dokumentation erklärt.

### 1.2 Problem b)

Hat man einmal die Menge I und damit die Schlüssel sowie die Sicherungskarte gefunden, stellt sich die Frage, wie Zara ein Haus ohne mehr als zwei Fehlversuche zu benötigen aufsperren kann.
Dies ist möglich, da die Codewörter aufsteigend sortiert sind. Das einzige Problem ist, dass die Sicherungskarte aufgrund von (1) nicht von den Schlüsseln unterscheidbar ist. Um ein Haus i aufzusperren muss Zara zuerst die i-te der sortierten Karten verwenden. Falls diese nicht passt weiß sie, dass die Sicherungs-Karte im Bereich [0;i) liegt und deshalb alle Indizes ab der Sicherungskarte um eins größer sind als sie seien sollten. Die gesuchte Karte ist dann die (i+1)-te

## 1.3 Naive Ansätze

Das Problem besitzt eine gewisse Ähnlichkeit zum Subset-sum Problem für eine feste Anzahl an Summanden, einziger Unterschied ist, dass bei diesem Problem $\oplus$ statt + verwendet wird. Eine DP-Lösung, wie sie beim Subset sum Problem für kleine Summen möglich wäre kann hier nicht verwendet werden, da die Zahlen im Bereich $[0, 2^{128})$ liegen.
Ein anderer möglicher Ansatz wäre alle k-Kombinationen an Indizes in [0;N) auszuprobieren, doch für N=111 und k=11 gibt es $\binom{N}{k} \approx 4.7 \cdot 10^{14}$ viele Möglichkeiten. Diese alle einzeln auszuprobieren würde zu lange dauern. Der Ansatz alle Kombinationen zu durchsuchen kann jedoch durch einige Beobachtungen beschleunigt werden.

## 1.4 Effizientere Suche durch Hashtable Strukur

Die gegebenen Zahlen werden in zwei Hälften L und R geteilt. Angenommen folgende Information ist bekannt: l Indizes liegen in L und r Indizes liegen in R Eine l-Kombination an Indizes $K_1$ aus der ersten Hälfte ist dann teil eines möglichen I, wenn eine korrespondierende r-Kombination an Indizes $K_2$ aus der zweiten Hälfte existiert, so dass:

$$\begin{aligned} & xor(K_1) \oplus xor(K_2) = 0 \\ \implies & xor(K_1) = xor(K_2) \end{aligned}$$

Anstatt für alle $\binom{N/2}{l}$ möglichen $K_1$ jeweils alle $\binom{N/2}{r}$ möglichen $K_2$ auszuprobieren
(Laufzeit: $O(\binom{N/2}{l} \cdot \binom{N/2}{r})$, können alle $\binom{N/2}{r}$ möglichen $K_2$ in einer Hashtabelle gespeichert werden, wo sie mit $xor(K_2)$ als Schlüssel in konstanter Zeit gefunden werden können (Laufzeit: $O(\binom{N/2}{l} + \binom{N/2}{r})$.
Für r=5 müssen nur $\binom{55}{5} \approx 3.5 \cdot 10^6$ viele Werte gespeichert werden.
Das Problem ist nun aber, dass l und r nicht bekannt sind. Zur Lösung gibt es zwei Ansätze

## 1.5 Ansatz 1: Alle möglichen Paare l, r ausprobieren

Da es für k=11 nur 11 mögliche Paare (l, r) gibt wäre es denkbar all diese Paare auszuprobieren. Eine Hürde stellen die Extremfälle dar, in denen zum Beispiel r=11 ist. Dann gibt es nämlich $\binom{55}{11} \approx 1.2 \cdot 10^{11}$ Kombinationen rechts. Zwei Optimierungen werden benötigt:

### 1.5.1 Kostenschätzfunktion und Rekursion

Falls die Anzahl an Kombinationen für ein (l, r) zu groß ist um komplett durchgegangen zu werden, soll der Suchalgorithmus für jede Kombination der geringeren Hälfte rekursiv auf die größere Hälfte angewandt werden. Das heißt, dass im Falle l=0 der Vorgang also einmal rekursiv auf die rechte Hälfte angewandt wird. Dabei muss beachtet werden, dass für eine Teilsuche $xor(K_1) \oplus xor(K_2) = s$ und s nicht immer 0 ist.

Um zu wissen wann es für ein für eine Bereichsgröße n und ein Paar l, r besser ist die Suche sofort mittels Hashtable auszuführen oder stattdessen mit erneuter Rekursion die Suche zu beschleunigen, werden die Funktionen $es(n, k)$, die Anzahl der benötigten Schritte für eine Problemstellung mit n Zahlen und k gesuchten Indizes und $si(n, l, r)$, die Anzahl der benötigten Schritte für ein bestimmtes $(l, r)$, benötigt.

$$es(n,k) = \sum_{(l,r)} si(n,l,r)$$

$$si(n,l,r) = \min\left(\binom{n/2}{l}^{\Sigma} + \binom{n/2}{r}^{\Sigma}, \quad \binom{n/2}{l}^{\Sigma} \cdot es(\frac{n}{2}, r)\right) \quad \text{, wobei } \binom{n}{k}^{\Sigma} = \sum_{j=1}^{k} \binom{n}{j} \text{ und } k \leq \frac{n}{2}$$

$si$ soll minimiert werden. In der oben gezeigten Formel ist das 1. Argument von min die Schrittzahl für eine "lokale" Suche und das 2. Argument die Schrittzahl der Rekursion. Um $es$ während der Verarbeitung effizient zu berechnen werden die Ergebnisse gespeichert, was in einer Laufzeit $O(N^2)$ (für $es$) resultiert.
$\binom{n}{k}^{\Sigma}$ ist eine Approximation für die zur Iteration aller Kombinationen benötigte Schrittzahl für "kleine"k's. In 2.3.1 wird eine effizientere Methode gezeigt.

### 1.5.2 Halbierung der Menge an auszuprobierenden (l, r) Paaren

Wenn die Zahlen vor der Verarbeitung sortiert werden, kann man sie statt exakt in der Hälfte, beim Übergang des bedeutendsten Bits teilen. So kann man bestimmte (l,r) Paare ausschließen, da für keine ungerade Kombination von Zahlen z, die alle an Stelle x ein gesetztes Bit haben, $xor(z) = 0$ gilt.

Der gerade genannte Ansatz ist in der Praxis mit bestimmten Optimierungen schnell genug um das Beispiel 2 mit N=111 und k=11 in ca. 13 Minuten zu lösen. Jedoch ist er zu langsam für die größeren Beispiele und schwer zu parallelisieren.

## 1.6 Ansatz 2: Verschieben der Hälften

Es sind die Paare (l, r) am aufwendigsten, bei denen l und r weit auseinander liegen:

$$\frac{\binom{56}{0} + \binom{55}{11}}{\binom{56}{5} + \binom{55}{5}} \approx 3.3 \cdot 10^3$$

Der folgend beschriebene Algorithmus iteriert über verschiedene Zusammensetzung der Hälften und belässt dabei die l r Verteilung gleich, so dass l und r so nah aneinander wie möglich sind. Es werden maximal $\lfloor \frac{N}{2} \rfloor + 1$ Iterationen benötigt und diese werden von 0 an in einer Schritten gezählt. Wie werden die Hälften zusammengesetzt und ist garantiert, dass alle Kombinationen betrachtet werden?
Die Hälften kann man sich als zusammenhängende Bereiche vorstellen, die bei jeder Iteration verschoben werden. Beispiel für N=8:

- s=0: ****....
- s=1 .****...
- s=2 ..****..
- s=3 ...****.
- s=4 ....****

Allgemein entsprechen bei der Iteration s:
Die erste Hälfte $L_s := [s; s + \lceil N/2 \rceil)$
Die Zweite Hälfte $R_s := Z \setminus L_s$
**Hilfssatz: Für jede k-Kombination aus Z existiert mindestens ein** $0 <= s <= \lfloor \frac{N}{2} \rfloor$ **, so dass** $\lceil \frac{k}{2} \rceil$ **der Elemente der k-Kombination in** $L_s$ **sind** und daraus folgend $\lfloor \frac{k}{2} \rfloor$ in $R_s$.
Aus dem Hilfssatz folgt die Korrektheit des Verfahrens, da dann jede k-Kombination bei mindestens einer Iteration in Betracht gezogen wird.

### 1.6.1 Beweis des Hilfssatzes

Um denn Satz zu beweisen, reicht es zu zeigen, dass für eine beliebige k-Kombination C ein s gefunden werden kann, dass die erwartete Bedingung erfüllt.
$z_s$ sei die Anzahl der Kombinationselemente in der ersten Hälfte: $z_s := |C \cap L_s|$
$x := z_0$
$y := z_{\lfloor \frac{N}{2} \rfloor}$
Zentral für den Beweis ist folgende Beobachtung über die Veränderung von $z_s$ bei Änderung von s:

$$|z_{s+1} - z_s| = \begin{cases} 1 \\ 0 \end{cases} \tag{2}$$

Dies ergibt sich daraus, dass bei einer "Verschiebung" von $L_s$ maximal ein Element entfernt werden muss und maximal eines hinzugefügt werden muss.
Da jede ganze Zahl zwischen inklusiv x und inklusiv y deswegen sicher bei einer Iteration $z_s$ entspricht, muss nun gezeigt werden, dass:

$$\lceil \frac{k}{2} \rceil \text{ zwischen inklusiv } x \text{ und } y \text{ liegt.} \tag{3}$$

Der Beweis wird nur für $x \leq y$ geführt, kann jedoch symmetrisch (mann muss x mit y tauschen) auch für $y \leq x$ geführt werden.

Zuerst wird eine Variable u eingeführt, da sich für ein ungerades N, durch das Überlappen von $L_0$ und $L_{\lfloor \frac{N}{2} \rfloor}$ ein Spezialfall ergibt, falls der "überlappte" mittlere Index in C ist.

$$u := \begin{cases} 1, \text{falls n ungerade ist und für das mittlere immer in } L_s \text{ enthaltene Element gilt} \lfloor n/2 \rfloor \in C \\ 0, \text{in allen anderen Fällen} \end{cases}$$

$x = a + u$
$y = b + u$
$a + u + b = k$

$$x \leq \lceil \frac{k}{2} \rceil \leq y \iff$$
$$a + u \leq \lceil \frac{a + u + b}{2} \rceil \leq b + u$$

Es wird zwischen u=0 und u=1 unterschieden:
Im ersten Fall gilt: $a \leq \lceil \frac{a+b}{2} \rceil \leq b$, da der Durchschnitt immer zwischen seinen zwei Teilen liegt und a und b ganz sind.
Im zweiten Fall muss zwischen $(a + b) \mid 2$ und $(a + b) \nmid 2$ unterschieden werden.
Falls $(a + b) \mid 2$ gilt $\lceil \frac{a+b+1}{2} \rceil = \frac{a+b}{2} + 1$.
Das $a + 1 \leq \frac{a+b}{2} + 1 \leq b + 1$ gilt aus dem gleichen Grund, wie im ersten Fall.
Falls $(a + b) \nmid 2$:

$$\lceil \frac{a + b + 1}{2} \rceil = \frac{a + b}{2} + \frac{1}{2}$$
$$\frac{a + b}{2} + \frac{1}{2} \leq \text{b+1, da } a < b$$
$$a + 1 \leq \frac{a + b}{2} + \frac{1}{2},$$

da der einzige Fall in dem das nicht so wäre a=b fordert und dies $(a + b) \nmid 2$ wiederspricht.

### 1.7 Das komplette Verfahren

Die Laufzeit des zweiten Ansatzes ist $O(N \cdot \binom{N/2}{k/2})$, wobei der zweite Faktor eigentlich eabhängig von der Laufzeit ist, mit der man alle Kombinationen tatsächlich iterieren kann. Siehe 2.3.1 Der zweite Ansatz kann gut parallelisiert werden und ist somit in der Lage alle Beispiel einigermaßen schnell zu lösen. Je größer die Eingabe wird, desto größer wird auch die Größe der verwendeten Hashtabellen und benötigten Rams. 16 GB reichen aus um für Beispiel 4 8 Kerne parallel zu verwenden und das Beispiel in weniger als einer halben Stunde zu lösen. Es wäre denkbar für noch größerere Eingaben Ansatz 1 und 2 zu kombinieren, um den Speicherbedarf unter Kontrolle zu behalten.

## 2 Umsetzung

Die Umsetzung ist in Rust geschrieben und verwendet die "rand", "serde" und "ahash" Pakete. Mehr zum letzteren Packet unter 2.2. Für Windows und x86-Linux kompilierte Programme findet man in /bin

### 2.1 I/O

Die Eingabe und Ausgabe findet über die structs *TInput* und *TOutput* statt, die sich beide in "io.rs" befinden. In dieser Datei sind auch die Obergrenzen für die Berechnung des Binomialkoeffizienten festgelegt: $N \leq 256$ und $k \leq 20$.

## 2.2 Strukturen

Die Implementierung verwendet ausschließlich 128 Bit Zahlen als Darstellung der Schlüsselkarten, da keine der Beispielsangaben größere Zahlen verlangt und flexibel große Zahlen die Verarbeitung nicht entscheidend beschleunigen würden. Trotzdem enthält die Datei, "structs.rs", die verschiedene nützliche Strukturen enthält, auch eine sehr einfache Implementierung einer 256-Bit Zahl: *u256*. Benutzt wird diese Zahl von dem struct *Combination*. Dieses struct enthält für eine Kombination deren *xor* Wert, sowie die Elemente der Kombination als 256-Bit Bitmaske. *Combination* können mittels *add* weitere Elemente hinzugefügt werden und mit *combine* können zwei Objekte kombiniert werden.

Die Kombinationen werden in einer *Hashmap<u128, u256>* gespeichert, welche die *xor* Werte als Schlüssel hat und die Kombinationsmasken als Wert. Die Hashmap ist in hinter dem struct *HashMapStore* versteckt welches über die Implementierung des folgenden Interfaces benutzt wird

```
pub trait CombStore : Clone {
    fn new(size: usize) -> Self;
    fn insert(&mut self, k: u128, v: u256);
    fn get(&mut self, k: u128) -> Option<u256>;
    fn clear(&mut self);
}
```

Statt Hashmaps könnten auch B-Bäume oder sortierte Arrays benutzt werden, aber diese haben sich als langsamer erwiesen.

Wenn die Zahlen nur 64-Bit hätten müsste man sie überhaupt nicht hashen und auch für 128-Bit Zahlen könnte man eine Hashfunktion benutzen, die sie nur Modulo $2^{64}$ nimmt. Die von dem "ahash" Paket bereitgestellte Hashfunktion ist leicht schneller als die gerade eben genante Methode, stellt aber ansonsten keine bedeutende Beschleunigung dar.

Um die effiziente Berechnung von dp Werte zu erlauben, enthält "structs" noch *DPArray*, ein bis zu 3-dimensionales Array. Dieses wird zum Beispiel von *BinomC*, einem struct aus "math.rs", welches den Binomialkoeffizienten berechnet, verwendet.

## 2.3 Verarbeitung

Die Verarbeitung, welche in der "processing.rs" Datei implentiert ist, beginnt mit der *process* Funktion. Diese erhält die Eingabe, sowie ein *Constraints* Objekt, welches die Größe der gleichzeitig gespeicherten *Combination* Werte begrenzt, sowie die Anzahl der gleichzeitig verwendeten Rechen-Threads. Es machen maximal so viele Threads Sinn, wie der CPU Cores hat. Die weitere Verarbeitung findet mit einem *Solver* Objekt statt, welches die Methode *shift_search* besitzt. Diese Methode berechnet wie viele Threads tatsächlich parallel benutzt werden können, ohne zu viel Speicher zu verbrauchen, und spawnt dann diese Threads, wobei die HashMaps statt jedesmal neu erstellt zu werden zwischen den Threads ausgetauscht werden. Innerhalb eines dieser Rechen-Threads wird folgende Funktion verwendet:

```
pub fn search_single_shift<T: CombStore>(nums: &[u128], segment: Segment, k: usize,
↪   shift: usize, target: u128, store: &mut T) -> SearchRes {
    let mut res: SearchRes = None;
    let l = (k as f64/2.0).ceil() as usize;
    let r = k-l;
    let blocks = split_segment_simple(segment);
    let pass = assign_k_simple(blocks, l, r);
    store.clear();
    map_combs_adv(nums, pass.ca.1, &mut |x| {store.insert(x.0, x.1);}, pass.ca.0,
↪   shift);
    let mut it_func = |x: &Combination| {
        let compl = x.0 ^ target;
        match store.get(compl) {
            Some(c) => {res = Some(x.combine(&Combination(compl, c)));},
            None => ()
        }
    };
    map_combs_adv(nums, pass.it.1, &mut it_func, pass.it.0, shift);
    res
}
```

Die Funktionen *split...simple* und *assign...simple* teilen lediglich den vorgegeben Zahlenbereich in zwei Hälften auf. *map_combs_adv* ruft eine zweite Funktion für alle Kombinationen der Länge k aufruft. Diese zweite Funktion ist für die eine Hälfte die Funktion, die die Kombination speichert

```
store.insert(x.0, x.1);
```

und für die zweite Hälfte die Funktion *it_func*, die für die gegebene Kombination die entsprechende findet.

```
store.get(compl)
```

Nachdem eine gültige k-Kombination gefunden wurde werde die enthaltenen Zahlen mit *combination_nums* aus der Bitmaske gelesen.

#### 2.3.1 Zu *map_combs_adv*

Wie kann man möglichst effizient über alle Kombinationen iterieren? Die folgende Funktion bietet eine einfache Lösung:

```
pub fn map_combs_simple(nums: &[u128], k: usize, func: &mut dyn FnMut(&Combination),
↪  block: Segment, shift: usize, cur: Combination) {
    assert!(block.1 <= nums.len());
    if k == 0 {
        func(&cur);
        return;
    }
    if block.0==block.1 {return;}
    for i in block.0..block.1 {
        let num_idx = (i+shift) % nums.len();
        map_combs_simple(nums, k-1, func, Segment(i+1, block.1), shift,
↪  cur.add(nums[num_idx], num_idx));
    }
}
```

Die Funktion wird jedoch sehr ineffizient, wenn $k > \frac{n}{2}$. *map_combs_adv* unterscheidet sich von der vorherigen Funktion darin, dass sie vor der Rekursion erst "Präfixxors" der Karten berechnet und dann, wenn $k > \frac{n}{2}$ denn übrigen Zahlenbereich in konstanter Zeit "invertiert". Statt Zahlen auszusuchen, die in der Kombination sind, werden dann Zahlen ausgesucht, die nicht in der Kombination sind. Bild der Aufrufe

### 2.4 Verifikation

Um sicherzustellen, dass die Implementierung nicht fehlerhaft ist, wird das Ergebnis mit *TOutput.verify()* verifiziert. Zusätzlich sind in "testing.rs" Funktionen enthalten, die zufällige Eingaben generieren. Die Wahrscheinlichkeit, dass eine zufällige Eingabe eine gültige Kombination enthält kann geschätzt werden

$$P = \binom{n}{k} \cdot (1 - (\frac{2^m - 1}{2^m})^{(n-k)}) \tag{4}$$

Da P in den meisten Fällen nahezu 0 ist, gibt es die Funktion *generate_solvable* die zufällige lösbare Eingaben generiert.

## 3 Erweiterungen

### 3.1 Verteilung des Rechenaufwands auf mehrere Computer

Obwohl das Programm dank Paralelisierung auch das schwerste Beispiel in weniger als einer halben Stunde lösen kann, gibt es immer noch Potential, den Lösungsvorgang zu beschleunigen. Die hier betrachtete Erweiterung ist ein einfaches Server-Client System, mit dem die Berechnung der verschiedenen Iteration auf mehrere Computer verteilt werden kann.

### 3.2 Zusätzliche Implementierung des 1. Ansatzes

Ich habe den 1. Ansatz, sowie die in der Lösungsidee vorgestellten Optimierungen zuerst implementiert und habe erst später den 2. Ansatz gefunden. Um die aktuelle Implementierung lesbar zu halten, ist sie nicht mehr mit dem 1. Ansatz kompatibel, doch die Entwicklung der Implementierung und damit auch eine funktionierende Umsetzung des 1. Ansatzes findet man in "BonusAufgabe_old". Die enthaltenen Dateien enthalten möglicherweise Fehler und sollten nicht kompiliert werden.

## 4 Beispiele

**Beispiel 0**
Eingabe:*hexmax0.txt*
Ausgabe:

```
1
00111101010111000110100110011001
11111110001011010001000000110111
11010111111010111101101111110000
10101100111111011010100011100000
10111000011001110000101010111110
```

**Beispiel 1**
Eingabe:*hexmax1.txt*
Ausgabe:

```
0
00100000111100111111011110111100
11010011010110110101001101010111
00110100001010100100001111010010
11110011101011001001000010111110
00110110000110101101011111111010
11110111100100010100100001001110
00100011100111011010111011100011
11000111111010110100000101110100
00010001110100110001111101100100
```

**Beispiel 2**
Eingabe:*hexmax2.txt*
Ausgabe:

```
13947
0110101110100011011101000110000111000001100011010110001011101110011001101111011101110
↪  0110101101111000011110111011101011111100111
0010101111100010101101011011110010011000000000000110100110011111011001011001000010001110
↪  1010110110010101110100100001011100011010001
1010101100000110110000010111111111100110001100111001010110111111011000111111101111101000
↪  0111111010000001011011011111111010011011110
1000000000010010011001100100011000000000001010101101001001000010001110101101101010100 1
↪  0101000101110101100101000110010100100111011
0010100001100001001011101110101101101011100010010011010111101101111011110010110000100
↪  1110010100001101001110001000100010011111100
1100001100010011011100010110010010110101011001101101010110100100001111010001000100101
↪  0000110010101010010001100010001101110100000
1010111111001001001010011110110001001111100001010100110010000111100010001001001001101
↪  0010101011111101011000001111110000000111011
1101111000010100110111110011000011101001101110111101011111011011011101101000100110110
↪  1100111010001000011000001010111100101111111
0110100100101100010100111111111010110000010001011001110101001010110110001000000000110000
↪  1100011010110101011110110100000100101001011
```

```
0111011001111000111001111000110110111010010100000010000010110000101000111010100000001
↪ 1010011000011010010110110100101111101101000
1110111010101110011110111100011100110111101101010101111100011010001101000110000011110
↪ 1000010001100100000011101100010001011101000
```

**Beispiel 3**

Eingabe:*hexmax3.txt*

Ausgabe:

```
279339
1011011101001011111011001100010101110100001111110000100000110011111111100100110001110
↪ 0011110011111101000111010111000011110110101
1011100010111110001011111010101010110011000100001101100110001011011000000110000110111
↪ 1010100001100100010001101000110010011001100
1100101101011111111011101000100010010000010110011101010011111010000010011111000111000
↪ 1011000000001001110110010011100000110011110
0101000010110111001111000111001101001100111111100000100000010000001011110000111010000
↪ 1001001111101111001010011110110111110011011
1011000000100110011011010100010011001110010110011010111011111010000011101000110000
↪ 1100000010101111111000000000101111010010001
0111110110001010110011001110101101010100110100011001111110101011000000011100011010100
↪ 1111111110100100001100010011111100010110100
1011111101010100110000010110110011111010000101000101001000001001111010110100100101100001
↪ 1101100011010100000011010101001110100010111
0111000111111010100001001110001111111111001111011011100010101000101100011000101010000 1
↪ 0100010100001010000010100001000101110101100
0111011010011100100001101110010010101011111100101111000000101100110110010110011100101
↪ 1100000011010010011100111100101011010010111
0010000011100111000110101000111111001111000101111010011001010110001001100010011110101
↪ 1001111001111111011110110011111001010100001
1100000101100100011010011101111111101111011011010111110100010110000101111000111001011
↪ 0100101100000111011101011010101100111010100
```

**Beispiel 4**

Eingabe:*hexmax4.txt*

Ausgabe:

```
1387652
1110001000000011110100111111100100110011101110100100011100111100111100001000010110000
↪ 1000011000011001011101101010000101111100001
0010110000011110111100001000000101101111111000011001111111111000111100000010111110110
↪ 0010010000010100001010111110110000101001010
0000011101101001010110111000111110100111001010001100000100000110111010111110100100010
↪ 0000001111011111001101011010000100011011110
0011011001011100100100111100111111010100111000001000000011000101010001110010001001110 0
↪ 0100110111111100100010010100100111011110100
0010110000111000111001111000010011000000000011101101100111010001010000101000011011001
↪ 0000111110011000100111101001100100010010000
0100001100100110101100111101110111101001011011101011111011011111100100010001011010110
↪ 0101111110000011000100111011000001101000111
1000000100010111001101010001100011010011010011110010001000100001101100000101011110011
↪ 0111011101110001011110000100100001110000101
1100010111000100100000101110100010011001100111101110100101101011010011100010000100010
↪ 0000011000010101111001001101101010111011101
1010101000001111111100111101111000100010011101000001001011110100000101000110001011001
↪ 1110111111010111000000011000110111110000110
1000110000101100110100101100011011010100110100001000010110101010110011011011111101011
↪ 0011001001101100100001011110000111010000111
1111001011000110001010011000100111000111101001110010001101010010100100010011110010100
↪ 0001000011101010011101000111001000000001111
```

**Beispiel 5**
Eingabe:*hexmax5.txt*
Ausgabe:

```
26
10000100111010100011111001001101100110111001010101000100000001001
11010100010011010001111111100001101000101001110001000010010110 11
10101110110011001001100011001100010111010010000000110111111001 00
01011111110001110000001011111000101110101101010001000000110010 00
10100001101011001011101110011000110111101111110101110000101111110
```

# 5 Quellcode