Bonus Aufgabe: LaTeX-Dokument

Team: ??? / Name
Team-ID: ???

12. April 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 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 hier ist um eins größer als in der Aufgabenstellung!)
Da $x \oplus x = 0$ und $\oplus$ assoziativ und kommutativ ist gilt:

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

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 $|I| = k \wedge xor(I) = 0$ 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 Durchquerung des Suchraums 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: (Zielwert?)

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

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 Hashtable 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. (Es sei an dieser Stelle angemerkt, dass das Durchsuchen von $\binom{n}{k}$ Kombinationen für $k \leq \frac{n}{2}$ besser durch $O(\sum_{j=1}^{k} \binom{n}{j})$ beschrieben wird. Da die Differenz jedoch nicht relevant zur Schätzung der Laufzeit ist wird sie ignoriert.)
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 l=0 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.

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 Schritee 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 berechen können Ergebnisse gespeichert, was in einer Laufzeit $O(N^2)$ (für es) resultiert.

Der gerade genannte Ansatz ist in der Praxis mit bestimmten Optimierungen schnell genug um das Beispiel mit n=111 und k=11 in ca. 13 Minuten zu lösen. Jedoch ist er zu langsam für die größeren Beispiele, schwer zu parallelisieren und unschön. Eine mögliche Optimierung ist die Zahlen zuerst zu sortieren.

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 $R_s := Z \setminus L_s$
Bild?
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. Die Laufzeit des zweiten Verfahrens ist ca. $O(N * \binom{N/2}{k/2})$

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: $z_s := |K \cap L_s|$
$x := z_0$
$y := z_{\lceil \frac{n}{2} \rceil}$
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.

Um den Satz zu beweisen 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.

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

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

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

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.
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 $a+1 \leq \frac{a+b}{2}+1 \leq b+1$ wird bereits im ersten Fall gezeigt.
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

Der zweite Ansatz kann gut parallelisiert werden und ist somit in der Lage alle Beispiel einigermaßen schnel 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 beispiel4 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” und ”ahash” Pakete. Mehr zum letzteren Packet unter 2.2

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 festgelegt: $N \leq 255$ 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 nativen *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 die 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();
    enum_combs(nums, pass.ca.1, &mut |x| {store.insert(x.0, x.1);}, pass.ca.0, shift,
↪  segment, Combination::default());
    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 => ()
        }
    };
    enum_combs(nums, pass.it.1, &mut it_func, pass.it.0, shift, segment,
↪  Combination::default());
    res
}
```

Die Funktionen *split...simple* und *assign...simple* teilen lediglich den vorgegeben Zahlenbereich in zwei Hälften auf. *enum_combs* geht alle Kombinationen der Länge k durch und ruft für jede Kombination eine Funktion auf. Diese ist für die eine Hälfte die Funktion, die die Kombination speichert und für die zweite Hälfte die Funktion *it_func*, die für die gegebene Kombination die entsprechende findet.

2.4 Verifikation

Zufällige Menge an Zahlen unwahrscheinlich

3 Erweiterungen

3.1 Verteilung des Rechenaufwands auf mehrere Computer

4 Beispiele

5 Quellcode