Aufgabe 2: Spießgesellen

Teilnahme-ID: 58138

Bearbeiter dieser Aufgabe:
Benedikt Kuder

18. April 2021

Inhaltsverzeichnis

Teilaufgabe a)....2
Lösungsidee....3
Diskussion....4
Umsetzung....4
Datentypen....4
Implementierung des Algorithmus....5
Diskussion....7
Beispiele....8
spiesse1.txt....8
spiesse2.txt....8
spiesse3.txt....9
spiesse4.txt....9
spiesse5.txt....9
spiesse6.txt....11
spiesse7.txt....11
Quellcode....12

Teilaufgabe a)

Zuerst wird eine Tabelle erstellt, mit einer Spalte pro Schüssel. In dieser Tabelle wird in der ersten Zeile die Schüsselnummer eingefügt. Die Änderung zwischen zwei Schritten wird im Folgenden mit roten Einträgen in den Tabellen gekennzeichnet. Das sieht dann folgendermaßen aus:

1	2	3	4	5	6

Darauf werden alle Früchte des ersten Spießes in alle Schüsseln des ersten Spießes geschrieben. Das sieht dann nach Mickeys Spieß folgendermaßen aus:

1 Apfel Banane Brombeere	2	3	4 Apfel Banane Brombeere	5 Apfel Banane Brombeere	6

Dies wird mit allen Spießen wiederholt, wenn in der jeweiligen Spalte noch keine Früchte sind. Sind dort bereits Früchte, wird nur die Schnittmenge übernommen.
Mit Minnies Spieß sieht die Tabelle Folgendermaßen aus:

1 Apfel Banane Brombeere	2	3 Banane Pflaume Weintraube	4 Apfel Banane Brombeere	5 ~~Apfel~~ Banane ~~Brombeere~~	6 Banane Pflaume Weintraube

Nun werden noch die Spieße von Gustav und Daisy nach demselben Verfahren in die Tabelle übernommen, diese sieht dann folgendermaßen aus:

1 Apfel ~~Banane~~ Brombeere	2 ~~Apfel~~ ~~Brombeere~~ Erdbeere	3 Banane Pflaume Weintraube	4 Apfel ~~Banane~~ Brombeere	5 ~~Apfel~~ Banane ~~Brombeere~~	6 ~~Banane~~ Pflaume ~~Weintraube~~

Nun wird für jede Schüssel mit eindeutiger Frucht, diese Frucht aus allen anderen Schüsseln gestrichen. Dies sieht nach Schüssel 2, 5 und 6 folgendermaßen aus:

1 Apfel ~~Banane~~ Brombeere	2 ~~Apfel~~ ~~Brombeere~~ **Erdbeere**	3 ~~Banane~~ ~~Pflaume~~ Weintraube	4 Apfel ~~Banane~~ Brombeere	5 ~~Apfel~~ **Banane** ~~Brombeere~~	6 ~~Banane~~ **Pflaume** ~~Weintraube~~

Dies wird solange wiederholt bis sich nichts mehr ändert. Danach sieht die Tabelle folgendermaßen aus:

1	2	3	4	5	6
Apfel	~~Apfel~~	~~Banane~~	Apfel	~~Apfel~~	~~Banane~~
~~Banane~~	~~Brombeere~~	~~Pflaume~~	~~Banane~~	**Banane**	**Pflaume**
Brombeere	**Erdbeere**	**Weintraube**	Brombeere	~~Brombeere~~	~~Weintraube~~

Nun werden alle Spalten entfernt welche keine von Donald gewünschte Frucht mehr enthalten. Darauf sieht die Tabelle folgendermaßen aus:

1	3	4
Apfel	~~Banane~~	Apfel
~~Banane~~	~~Pflaume~~	~~Banane~~
Brombeere	**Weintraube**	Brombeere

Nun sind exakt drei Spalten übrig. Da Donald sich exakt drei Früchte wünscht, sind nun die Schüsseln 1, 3 und 4 die, welche Donalds Früchte beinhalten.
Es ist nicht eindeutig, ob der Apfel oder die Brombeere in Schüssel 1 oder 3 sind, dies ist allerdings egal, da beide Schüsseln genau eine der beiden Früchte enthalten müssen, und so in Summe jede gewünschte Frucht einmal vorkommen muss.

Lösungsidee

Der Ansatz geht von folgender Annahme aus:

- Jede verschiedene Frucht darf in maximal einer Schüssel vorkommen

Die Lösung findet in vier Schritten statt.

Im **ersten** Schritt wird angenommen, dass in jeder Schüssel jede Frucht sein kann.
Immer wenn jemand mit einem Spieß von einer Schüssel kommt, könnten nur noch Früchte in der Schüssel sein, welche auch auf dem Spieß sind. Alle anderen Früchte werden von der Liste der möglichen Früchte entfernt. Dies wird für alle Spieße wiederholt.

Im **zweiten** Schritt wird jede Frucht, welche eindeutig einer Schüssel zuzuordnen ist, aus der Liste der möglichen Früchte aller anderen Schüsseln entfernt. Dieser Schritt wird so lange wiederholt, bis sich nichts mehr ändert.

Nun gibt es Fälle, in welchen Früchte eindeutig zugeordnet werden können, aber dies bisher nicht wurden. Deshalb wird nun im **dritten** Schritt über alle Schüsseln, welche eine von Donald gewünschte Frucht enthalten, alle möglichen Kombinationen „durchprobiert“.

Im **vierten** und letzten Schritt wird überprüft, ob die Anzahl der Schüsseln mit Früchten gleich der Anzahl der gewünschten Früchte ist. In diesem Fall ist das Ergebnis für die relevanten Früchte eindeutig, andernfalls ist die Situation nicht eindeutig lösbar.

Diskussion

Theoretisch hätte man gleich den brute-force Algorithmus des dritten Schritts nutzten können. Ohne die vorherige Vorsortierung durch Schritt eins und zwei ist dies allerdings nicht hinreichend performant.

Umsetzung

Das Programm wurde in Rust geschrieben.

Datentypen

Fruit

Das *Fruit* Struct besteht aus einem einzigen *String* welcher den Name der Frucht darstellt. Laut Beispielwebseite könnte eine Frucht auch mit ihrem Anfangsbuchstaben repräsentiert werden. Dies würde aber Früchte mit dem selbem Anfangsbuchstaben („Apfel“, „Apfelsine“, „Ananas“, „Aprikose“, etc.) ausschließen.

Bowl

Das *Bowl* Struct besteht aus einem einzigen *u32*, welcher die Nummer der Schüssel darstellt.

FruitSkewer

Das *FruitSkewer* Struct stellt einen Früchtespieß dar und besteht aus zwei *HashSet*. Eines gibt die Früchte auf dem Spieß an und das andere die besuchten Schüsseln.

BowlInfo

Das *BowlInfo* Struct repräsentiert die Menge der möglichen Früchte einer Schüssel. Es besteht aus einem *HashSet* welches alle möglichen Früchte beinhaltet. Ist das *HashSet* leer, könnten alle Früchte in der Schüssel sein.

Solution

Das *Solution* Struct wird alleinig zur Ausgabe genutzt und besteht aus einem *HashSet* aus Früchten welche Donalds Wünsche darstellt und einem *Vec* mit *BowlInfo* wobei der jeweilige Info die Nummer der Schüssel darstellt.

Solver

Das *Solver* Struct ist für das Lösen der Situation zuständig. Es besteht aus einem *HashSet* mit *Fruit* welches Donalds Wünsche darstellt, und einem *Vec* mit *FruitSkewer* welcher alle Spieße beinhaltet.

Implementierung des Algorithmus

Der Hauptteil findet in der *Solver::solve()* Methode statt.

Es wird zuerst eine *HashMap* angelegt welche für jede Schüssel eine *BowlInfo* beinhält.

Schritt eins

Der erste Schritt wurde durch eine for Schleife umgesetzt, welche über alle Früchtespieße iteriert. In der Schleife wird darauf über alle Schüsseln des Spießes in einer weiteren for Schleife iteriert. Darauf wird für jede Schüssel auf der mit dem entsprechenden Key gespeicherten *BowlInfo* die Methode *BowlInfo::add_skewer(früchtespieß)* ausgeführt.

Die Methode BowlInfo::add_skewer(früchtespieß) geht wie folgt vor:

- Wenn das *HashSet* von möglichen Früchten leer ist, werden alle Früchte des Spießes in dieses eingefügt.
- Wenn das *HashSet* bereits Einträge hat, wird über alle möglichen Früchte in *BowlInfo* iteriert und alle nicht auf dem Früchtespieß vertretenen Früchte werden entfernt.

Dieses Vorgehen erspart es zuerst alle Früchte in alle *BowlInfo* zu speichern, da eine Schüssel mindestens eine Frucht enthält, und so der erste Fall nur einmalig eintreten kann.

Der Schritt eins hat somit eine Laufzeit-Komplexität von O(n*k) wobei n die Anzahl der Früchtespießen ist und k die durchschnittliche Anzahl an Früchten pro Spieß. Die Worst-Case Speicher-Komplexität liegt bei O(n*k).

Schritt zwei

Nach Schritt eins wird in der *Solver::solve()* Methode zwei Variablen angelegt. *is_changeing* ist ein *bool* und wird auf *true* gesetzt und die Variable *taken_fruits* ist ein leeres *HashSet*.

Nun wird in einer *while* Schleife solange *is_changeing* wahr ist folgendes ausgeführt:

1. Es wird das *HashSet newly_taken_fruits* initialisiert/zurückgesetzt.
2. *is_changeing* wird auf *false* gesetzt.
3. Es wird über alle *BowlInfo* iteriert, welche exakt eine möglich Frucht beinhalten und diese eine Frucht nicht in *taken_fruits* ist.
    - Diese Frucht wird dann geklont und in *newly_taken_fruits* sowie *taken_fruits* eingefügt.
    - *is_changeing* wird wieder auf true gesetzt
4. Nun wird über alle *BowlInfo* iteriert
    - Für jede Frucht in *newly_taken_fruits* wird die Methode *BowlInfo::add_impossible_skewer()* ausgeführt.
      Diese Methode entfernt eine Frucht von dem *HashSet* der mögliche Früchte, wenn das *HashSet* mehr als einen Eintrag hat.

Dieser Schritt hat also eine Worst-Case Laufzeit-Komplexität von O(n^2) wobei n die Anzahl der Schüsseln ist. Häufig wird allerdings vorher abgebrochen, was die Average-Case Laufzeit-Komplexität weitaus geringer ausfallen lässt. Die Worst-Case Speicher-Komplexität liegt bei O(n).

Schritt drei

Nach Schritt zwei wird in der *Solver::solve()* Methode die *Solver::brute_force_remaining()* Methode ausgeführt.

Diese Methode wiederum filtert zu erst nach *BowlInfos* welche mindestens eine von Donalds Wunschfrüchten enthält.

Darauf werden folgende Variablen für den Brute-Force Algorithmus initialisiert:

- *bowl_infos* ist ein *Vec* der vorherig gefilterten *BowlInfos* und jeweiligem Key
- *possible_combinations* ist ein Vec aus *BowlInfo* zur Speicherung aller möglichen Kombinationen.
- *current_combination* ist ein *Vec* aus *Fruit* zur Speicherung der aktuellen Früchtekombination.

Nun wird die Methode *Solver::brute_force_helper(bowl_infos, current_combination, possible_combinations)* gestartet.

Diese rekursiv funktionierende Methode agiert folgendermaßen:

1. Wenn *current_combination* weniger Elemente hat als *bowl_infos*:
    1. Es wird über alle möglichen Früchte iteriert welche die *BowlInfo* mit dem Index *current_combination.len()* beinhaltet, und noch nicht in *current_combination* enthalten ist.
        1. Es wird *Solver::brute_force_helper(…)* mit den aktuellen Variablen aufgerufen
        2. *Es wird das letzte Element aus current_combination* entfernt, damit die Schleife wiederholt werden kann.
2. Andernfalls wurde eine korrekte Kombination gefunden, welche in *possible_combinations* wie folgt gespeichert werden soll:
    1. Es wird über alle Index von *current_combination* iteriert.
        1. Es wird in die *BowlInfo* in *possible_combinations* mit dem aktuellen index die *Fruit* aus *current_combination* als mögliche Frucht hinzugefügt.

Dadurch werden alle möglichen Kombinationen durchprobiert. Die Worst-Case Laufzeit-Komplexität dieses Schrittes beträgt O(k!/(k-n)!) wobei n die Anzahl an Schüsseln ist, und k die durchschnittliche Anzahl an Früchten pro Spieß. Im Normalfall ist k und n durch die Vorsortierung in Schritt eins und zwei kleiner. Im Best-Case beträgt die Laufzeitkomplexität O(s) wobei s die Anzahl der von Donald gewünschten Früchte ist.

Schritt vier

Der vierte Schritt wird in *Solution::new(bowlinfo)* ausgeführt. Dafür werden zu erst alle Schüsseln entfernt welche keine von den gewünschten Früchten enthalten. Darauf werden die *BowlInfo* nach Nummer der Schüssel sortiert. Ob ein Ergebnis eindeutig ist wird erst bei der Formatierung der Ausgabe ermittelt. Dafür wird verglichen, ob die Anzahl an Schüsseln die selbe ist, wie die Anzahl an Wünschen von Donald.

Diskussion

Die Einschränkung, dass jede Frucht einen eindeutigen Anfangsbuchstaben hat, ist nur für die Beispiele des BWInf zutreffend. Da es in der Realität aber mehr als 27 verschiedene Früchte gibt, wird diese Einschränkung hier nicht berücksichtigt. Das Programm geht deshalb von unendlich vielen verschieden möglichen Früchten aus. Daher gibt es kein Ausschlussverfahren basierend auf einer angenommenen maximalen Anzahl von Früchten. Dieses Verhalten wird in dem Beispiel von spiesse1.txt (siehe unten) deutlich.

Gibt es für eine Schüssel keine Informationen, geht der Algorithmus davon aus, dass diese leer ist. Dies ist aber in der Realität sicher nicht immer der Fall. Eindeutige Situationen werden dennoch weiter erkannt. Lediglich die Empfehlung bei manchen uneindeutigen Situationen verliert an Aussagekraft. Dies ist aber auch sinnvoll, denn wenn man davon ausgeht, dass es unendlich viele verschiedene Früchte gibt, ist die Wahrscheinlichkeit, dass in einer unbekannten Schüssel eine Wunschfrucht ist, vernachlässigbar gering.

Die Komplexität des Algorithmus ist die Summe der Komplexität der einzelnen Schritte. Nach Big-O-Notation wird nur das stärkste Wachstum berücksichtigt, welches mit $O(k!/(k-n)!)$ (n = Anzahl an Schüsseln; k = durchschnittliche Anzahl an Früchten pro Spieß) bei Schritt drei am größten ist. Da aber Schritt zwei und drei abhängig von der Situation auch weitaus langsamer als im Worst-Case an Komplexität gewinnen, ist die Worst-Case Komplexität nicht unbedingt aussagekräftig. Die Average-Case Komplexität wurde aufgrund der Abhängigkeit von den Beispielen nicht ermittelt.

Beispiele

Das Programm wir mit dem Dateipfad zur Beispieldatei als Parameter ausgeführt.

z.B. „aufgabe2 ./spiesse1.txt“

spiesse1.txt

Ausgabe:

```
Das Ergebnis ist nicht eindeutig.
Hier sind Schüsseln welche Donalds Wunschfrüchte beinhalten könnten:
 Schüssel 1 enthält Clementine
 Schüssel 2 enthält Erdbeere oder Himbeere
 Schüssel 4 enthält Erdbeere oder Himbeere
 Schüssel 5 enthält Johannisbeere
Donald hatte sich folgende Früchte gewünscht:
  Grapefruit Himbeere Johannisbeere Clementine Erdbeere
```

Das Beispiel hatte eine durchschnittliche Laufzeit von 11 µs.

Kommentar:

Wenn man nach dem Ausschlussverfahren vorgehen würde, könnte man noch behaupten, dass in Schüssel 7 eine Grapefruit ist. Da aber auf keinem Spieß eine Grapefruit ist, kann nicht davon ausgegangen werden, dass diese in diesem Szenario überhaupt vorkommt.

spiesse2.txt

Ausgabe:

```
Eindeutiges Ergebnis:
Donald muss an die Schüsseln 1 5 6 7 10 11
Inhalt der jeweiligen Schüsseln:
 Schüssel 1 enthält Apfel
 Schüssel 5 enthält Himbeere oder Banane oder Clementine
 Schüssel 6 enthält Feige oder Dattel oder Kiwi
 Schüssel 7 enthält Litschi
 Schüssel 10 enthält Himbeere oder Banane oder Clementine
 Schüssel 11 enthält Himbeere oder Banane oder Clementine
Donald hatte sich folgende Früchte gewünscht:
  Litschi Banane Apfel Kiwi Clementine Himbeere
```

Das Beispiel hatte durchschnittliche Laufzeit von 25 µs.

spiesse3.txt

Ausgabe

```
Das Ergebnis ist nicht eindeutig.
Hier sind Schüsseln welche Donalds Wunschfrüchte beinhalten könnten:
 Schüssel 1 enthält Himbeere
 Schüssel 2 enthält Litschi oder Grapefruit
 Schüssel 5 enthält Clementine
 Schüssel 7 enthält Feige oder Ingwer
 Schüssel 8 enthält Erdbeere
 Schüssel 10 enthält Feige oder Ingwer
 Schüssel 11 enthält Litschi oder Grapefruit
 Schüssel 12 enthält Kiwi
Donald hatte sich folgende Früchte gewünscht:
  Himbeere Clementine Feige Kiwi Erdbeere Ingwer Litschi
```

Das Beispiel hatte durchschnittliche Laufzeit von 30 µs.

spiesse4.txt

Ausgabe

```
Eindeutiges Ergebnis:
Donald muss an die Schüsseln 2 6 7 8 9 12 13 14
Inhalt der jeweiligen Schüsseln:
 Schüssel 2 enthält Kiwi
 Schüssel 6 enthält Ingwer
 Schüssel 7 enthält Nektarine
 Schüssel 8 enthält Grapefruit
 Schüssel 9 enthält Apfel
 Schüssel 12 enthält Pflaume
 Schüssel 13 enthält Feige
 Schüssel 14 enthält Orange
Donald hatte sich folgende Früchte gewünscht:
  Pflaume Feige Ingwer Apfel Orange Grapefruit Kiwi Nektarine
```

Das Beispiel hatte durchschnittliche Laufzeit von 40 µs.

spiesse5.txt

Ausgabe

```
Das Ergebnis ist nicht eindeutig.
Hier sind Schüsseln welche Donalds Wunschfrüchte beinhalten könnten:
 Schüssel 1 enthält Tamarinde oder Banane oder Mango oder Sauerkirsche
oder Dattel oder Apfel oder Pflaume oder Orange oder Litschi oder
Himbeere oder Grapefruit oder Kiwi oder Quitte
```

```
 Schüssel 2 enthält Orange oder Sauerkirsche oder Litschi oder Kiwi
 Schüssel 3 enthält Banane oder Litschi oder Kiwi oder Quitte
 Schüssel 4 enthält Mango oder Banane oder Sauerkirsche oder Apfel
oder Dattel oder Tamarinde oder Pflaume oder Orange oder Litschi oder
Himbeere oder Grapefruit oder Kiwi oder Quitte
 Schüssel 5 enthält Litschi oder Kiwi oder Himbeere
 Schüssel 6 enthält Dattel oder Banane oder Sauerkirsche oder Orange
oder Litschi oder Kiwi oder Quitte
 Schüssel 8 enthält Quitte oder Banane oder Ingwer oder Tamarinde oder
Litschi oder Johannisbeere oder Erdbeere oder Himbeere oder Kiwi oder
Rosine
 Schüssel 9 enthält Banane oder Litschi oder Kiwi oder Quitte
 Schüssel 10 enthält Litschi oder Himbeere oder Sauerkirsche oder
Pflaume oder Kiwi oder Orange
 Schüssel 11 enthält Ingwer oder Himbeere oder Litschi oder
Johannisbeere oder Kiwi
 Schüssel 12 enthält Banane oder Tamarinde oder Himbeere oder Litschi
oder Kiwi oder Quitte
 Schüssel 14 enthält Rosine oder Dattel oder Banane oder Sauerkirsche
oder Orange oder Litschi oder Johannisbeere oder Nektarine oder Kiwi
oder Quitte
 Schüssel 16 enthält Sauerkirsche oder Orange oder Litschi oder Kiwi
 Schüssel 17 enthält Rosine oder Banane oder Kiwi oder Litschi oder
Johannisbeere oder Quitte
 Schüssel 19 enthält Mango oder Dattel oder Banane oder Apfel oder
Sauerkirsche oder Tamarinde oder Pflaume oder Orange oder Litschi oder
Himbeere oder Grapefruit oder Kiwi oder Quitte
 Schüssel 20 enthält Ingwer oder Sauerkirsche oder Orange oder Pflaume
oder Johannisbeere oder Litschi oder Clementine oder Himbeere oder
Kiwi
Donald hatte sich folgende Früchte gewünscht:
  Sauerkirsche Clementine Banane Himbeere Tamarinde Pflaume Apfel
Grapefruit Nektarine Quitte Dattel Mango Orange
```

Das Beispiel hatte durchschnittliche Laufzeit von 3,7 sec.

Kommentar

Die Laufzeit ist deshalb wesentlich höher, da im Beispiel sehr viele Schüsseln, mehrere Früchte enthalten können. So lässt die Vorsortierung in Schritt eins und zwei mehr Möglichkeiten übrig, welche dann in Schritt drei durchprobiert werden. Schritt drei ist dabei sehr ineffizient aber benötigt, um sicherzustellen, dass in der Vorsortierung keine Früchte zu viel einer Schüssel zugeordnet wurden.

spiesse6.txt

Ausgabe

```
Eindeutiges Ergebnis:
Donald muss an die Schüsseln 4 6 7 10 11 15 18 20
Inhalt der jeweiligen Schüsseln:
 Schüssel 4 enthält Quitte
 Schüssel 6 enthält Vogelbeere
 Schüssel 7 enthält Clementine
 Schüssel 10 enthält Erdbeere
 Schüssel 11 enthält Rosine oder Ugli
 Schüssel 15 enthält Rosine oder Ugli
 Schüssel 18 enthält Himbeere
 Schüssel 20 enthält Orange
Donald hatte sich folgende Früchte gewünscht:
  Rosine Clementine Quitte Ugli Vogelbeere Erdbeere Orange Himbeere
```

Das Beispiel hatte durchschnittliche Laufzeit von 66 µs.

spiesse7.txt

Ausgabe

```
Das Ergebnis ist nicht eindeutig.
Hier sind Schüsseln welche Donalds Wunschfrüchte beinhalten könnten:
 Schüssel 3 enthält Grapefruit oder Litschi oder Apfel oder Xenia
 Schüssel 5 enthält Zitrone oder Tamarinde
 Schüssel 8 enthält Sauerkirsche oder Yuzu
 Schüssel 10 enthält Grapefruit oder Litschi oder Apfel oder Xenia
 Schüssel 14 enthält Sauerkirsche oder Yuzu
 Schüssel 18 enthält Banane oder Ugli
 Schüssel 20 enthält Grapefruit oder Litschi oder Apfel oder Xenia
 Schüssel 23 enthält Zitrone oder Tamarinde
 Schüssel 24 enthält Clementine
 Schüssel 25 enthält Banane oder Ugli
 Schüssel 26 enthält Grapefruit oder Litschi oder Apfel oder Xenia
Donald hatte sich folgende Früchte gewünscht:
  Dattel Mango Xenia Zitrone Apfel Clementine Tamarinde Ugli Yuzu
Vogelbeere Grapefruit Sauerkirsche
```

Das Beispiel hatte durchschnittliche Laufzeit von 356 ms.

Kommentar

In diesem Beispiel herrscht ein ähnlicher Fall wie in Beispiel spiesse5.txt. Da aber in diesem Beispiel weniger Schüsseln nach der Vorsortierung übrig bleiben ist die Laufzeit dennoch wesentlich geringer als in spiesse5.txt.

spiesse99.txt

Diese Beispieldatei und ihre Ausgabe wurden der Dokumentation beigelegt.

Das Interessante an diesen Beispiel ist, dass obwohl es weit über 26 verschiedene Früchte (278) und 112 Spieße gibt, das Programm nur ca. 256 µs benötigt um zu einer Lösung zu kommen.

Dies Liegt daran, dass die Meisten Früchte bereits in Schritt eins und zwei zugeteilt wurden, womit Schritt 3 welcher am rechen-aufwendigsten ist fast nichts mehr ‚durchprobieren' muss.

Es Lässt sich also daraus schließe, dass je mehr Informationen über die Schüsseln bekannt sind, desto schneller Läuft das Programm. Dies liegt daran, dass Schritt eins und zwei im Worts-Case eine Laufzeitkomplexität von nur O(n^2) + O(n * k) haben. Im Best-Case, welchem sich mit mehr Informationen immer weiter angenähert wird, hat Schritt drei auch nur eine lineare Komplexität hat.

Quellcode

```
pub fn solve(&self) -> Solution {
    let mut bowl_infos = HashMap::new();

    //Step one
    for skewer in &self.fruit_skewers {
        for possible_bowl in skewer.bowls() {
            let bowl_info = bowl_infos.entry(possible_bowl.clone()).or_insert(BowlInfo::new());
            bowl_info.add_skewer(skewer);
        }
    }

    //Step two
    let mut is_changeing = true;
    let mut taken_fruits = HashSet::new();
    while is_changeing {
        let mut newly_taken_fruits = HashSet::new();
        is_changeing = false;
        for bowl_info in bowl_infos.values() {
            if bowl_info.possible_fruits().len() == 1 {
                let fruit = bowl_info.possible_fruits().iter().next().unwrap();
                if !taken_fruits.contains(fruit) {
                    newly_taken_fruits.insert(fruit.clone());
                    taken_fruits.insert(fruit.clone());
                    is_changeing = true;
                }
            }
        }
        for bowl_info in bowl_infos.values_mut() {
            for fruit in &newly_taken_fruits {
                bowl_info.add_impossible_fruit(fruit);
            }
        }
    }
    //Step three
    return self.brute_force_remaining(bowl_infos);
}
```

```
fn brute_force_remaining(&self, mut bowl_infos: HashMap<Bowl, BowlInfo>) -> Solution {
    //Filter for bowls, which could contain one of Donalds wishes
    bowl_infos.retain(|_, value| !value.possible_fruits().is_disjoint(&self.donalds_wishes));

    //Prepare variables for brute force
    let bowl_infos: Vec<(Bowl, BowlInfo)> = bowl_infos.into_iter().collect();
    let mut possible_combinations: Vec<BowlInfo> = Vec::with_capacity(bowl_infos.len());
    possible_combinations.resize(bowl_infos.len(), BowlInfo::new());
    let mut current_combination: Vec<Fruit> = Vec::with_capacity(bowl_infos.len());

    //Brute Force
    self.brute_force_helper(bowl_infos.as_slice(), &mut current_combination, &mut
possible_combinations);

    let result: Vec<(Bowl, BowlInfo)> = possible_combinations.into_iter().enumerate().map(|e|
(bowl_infos[e.0].0.clone(), e.1)).collect();

    return Solution::new(result, self.donalds_wishes.clone());
}

fn brute_force_helper(&self, bowl_infos: &[(Bowl, BowlInfo)], current_combination: &mut
Vec<Fruit>, possible_combinations: &mut [BowlInfo]) {
    debug_assert_eq!(bowl_infos.len(), possible_combinations.len());

    if current_combination.len() < bowl_infos.len() {
        for fruit in bowl_infos[current_combination.len()].1.possible_fruits() {
            if current_combination.contains(fruit) {
                //Don't use the same fruit twice
                continue;
            }
            current_combination.push(fruit.clone());
            self.brute_force_helper(bowl_infos, current_combination, possible_combinations);
            current_combination.remove(current_combination.len()-1);
        }
    } else {
        debug_assert_eq!(possible_combinations.len(), current_combination.len());
        for i in 0..possible_combinations.len() {
            possible_combinations[i].add_possible_fruit(current_combination[i].clone());
        }
    }

}
```