Aufgabe 2: Spießgesellen

Teilnahme-Id: 57429

Bearbeiter dieser Aufgabe:
Lucas Schwebler

18. April 2021

Inhaltsverzeichnis

1 Lösungsidee

1.1 Definition

Gegeben sind je n Mengen F_i und S_i, wobei F_i die Früchte auf dem i-ten beobachteten Obstspieß darstellt und S_i die Menge der Schüsseln, bei denen der Besitzer des Obstspießes war. Das Ziel ist es, aus diesen Mengen die Menge der Schüsseln G zu bestimmen, die der gegebenen Menge von Donalds Wunschsorten D entspricht. Die Menge aller m Früchte heißt $\mathbb{F}$.
Falls G nicht eindeutig bestimmt werden kann, soll das Programm nach der Aufgabenstellung „eine möglichst informative Meldung ausgeben“. Dies wird so interpretiert, dass die Menge der Nummern, die eindeutig zu Früchten auf Donalds Wunschspieß gehören, ausgegeben werden, sowie die Menge der Nummern, die auf keinen Fall für Donald in Frage kommen. Außerdem werden angaben der Form „Unter den Schüsseln der Menge M befinden sich k Früchte der Wunschsorten“ ausgegeben. Donald kann diese Information z.B. nutzen, um das Risiko einer unsicheren Schüssel abzuschätzen, dass sich in dieser eine ungewollte Sorte befindet.
Falls sich Frucht i in Schüssel j befindet, *gehören* i und j *zusammen*, was im Folgenden auch $i \sim j$ geschrieben wird.

1.2 Grundlegende Idee

Betrachtet wird die Menge A_i der Personen, die Frucht i auf ihrem Obstspieß haben, und die Menge B_j der Personen, die bei Schüssel j waren. Falls i und j zusammen gehören, so müssen beide Mengen gleich sein. Formal $i \sim j \implies A_i = B_j$. Dies liegt daran, dass sich nach Aufgabenstellung alle Stücke einer Sorte in einer eigenen Schüssel befinden. Somit können die Personen aus A_i im Fall von $i \sim j$ nur die Frucht i erhalten haben, indem sie zu Schüssel j gegangen sind. Andersherum haben alle Gäste, die bei Schüssel j waren, Frucht i auf ihren Spieß aufgenommen.
Umgekehrt bedeutet das, dass $i \sim j$ nur dann gelten kann, wenn auch tatsächlich $A_i = B_j$. Zwischen gleichen Mengen $A_{i_1} = A_{i_2} = ... = B_{j_1} = B_{j_2} = ...$ könnten beliebige Paare A_i und B_j zusammengehören, da sie aus den Beobachtungen nicht unterscheidbar sind. Wenn also die Früchte $i_1, i_2, ..., i_k \in D$ und $A_{i_1} = A_{i_2} = ... = A_{i_k}$, so gehören genau k Schüsseln j mit $B_j = A_i$ zu der gewünschten Menge. Falls es genau k von diesen gibt, ist die Antwort für diese Früchte eindeutig. Gehören aber nicht alle Früchte mit gleicher Personenmenge zu Donalds Wunschsorten, so kann die Antwort nicht eindeutig bestimmt werden. Aber für die geforderte „möglichst informative Ausgabe“ kann zumindest ausgegeben werden, dass sich unter den Schüsseln $\{j | B_j = A_i\}$ genau k mit einer gewünschten Fruchtsorte befinden.

1.3 Beispiel aus Aufgabe a)

Mit den bisherigen Überlegungen soll das Beispiel aus Teilaufgabe a) gelöst werden. Hier zunächst die Daten aus diesem Beispiel:

$$\mathbb{F} = \{Apfel, Banane, Brombeere, Erdbeere, Pflaume, Weintraube\}$$
$$D = \{Apfel, Brombeere, Weintraube\}$$
$$F_1 = \{Apfel, Banane, Brombeere\} \qquad S_1 = \{1, 4, 5\}$$
$$F_2 = \{Banane, Pflaume, Weintraube\} \qquad S_2 = \{3, 5, 6\}$$
$$F_3 = \{Apfel, Brombeere, Erdbeere\} \qquad S_3 = \{1, 2, 4\}$$
$$F_4 = \{Erdbeere, Pflaume\} \qquad S_4 = \{2, 6\}$$

Damit es etwas übersichtlicher wird, werden die Früchte durch ihre Indices in $\mathbb{F}$ ersetzt:

$$D = \{1, 3, 6\}$$
$$F_1 = \{1, 2, 3\} \qquad S_1 = \{1, 4, 5\}$$
$$F_2 = \{2, 5, 6\} \qquad S_2 = \{3, 5, 6\}$$
$$F_3 = \{1, 3, 4\} \qquad S_3 = \{1, 2, 4\}$$
$$F_4 = \{4, 5\} \qquad S_4 = \{2, 6\}$$

Nun können die Mengen A_i und B_j bestimmt werden:

$$\begin{aligned} A_1 &= \{1,3\} & B_1 &= \{1,3\} \\ A_2 &= \{1,2\} & B_2 &= \{3,4\} \\ A_3 &= \{1,3\} & B_3 &= \{2\} \\ A_4 &= \{3,4\} & B_4 &= \{1,3\} \\ A_5 &= \{2,4\} & B_5 &= \{1,2\} \\ A_6 &= \{2\} & B_6 &= \{2,4\} \end{aligned}$$

Es ist z.B. $A_1 = \{1,3\}$, da Frucht 1, also $Apfel$, in den Beobachtungen 1 und 3 auf dem Spieß sind, in den anderen aber nicht.
Besonders interessant sind A_1, A_3, A_6, da diese den Wunschsorten aus D entsprechen. Es fällt auf, dass $A_1 = A_3$. Somit ist es nicht möglich zu bestimmen, welche Schüssel Frucht 1 entspricht und welche Frucht 3. Allerdings gibt es genau zwei Mengen B_1, B_4, die identisch zu A_1, A_3 sind. Somit ist entweder $1 \sim 1$ und $3 \sim 4$ oder $1 \sim 4$ und $3 \sim 1$. Auch wenn die genaue Verteilung nicht bekannt ist, befindet sich so oder so bei den Schüsseln 1 und 4 jeweils eine Wunschsorte. Für A_6 ist nur B_3 identisch, womit klar ist, dass Frucht 6 zu Schüssel 3 gehört ($3 \sim 6$).
Donald sollte also zu den Schüsseln 1, 3 und 4 gehen.

Die Informationen reichen nicht aus Im Beispiel aus Aufgabe a) heißt es, dass die Informationen der ersten 3 Personen nicht ausreichen, um die Menge der Wunschschüsseln eindeutig zu bestimmen. Um dies zu zeigen, wird erneut A_i und B_j ohne die letzte Beobachtung betrachtet:

$$\begin{aligned} A_1 &= \{1,3\} & B_1 &= \{1,3\} \\ A_2 &= \{1,2\} & B_2 &= \{3\} \\ A_3 &= \{1,3\} & B_3 &= \{2\} \\ A_4 &= \{3\} & B_4 &= \{1,3\} \\ A_5 &= \{2\} & B_5 &= \{1,2\} \\ A_6 &= \{2\} & B_6 &= \{2\} \end{aligned}$$

Das Problem ist hierbei, dass Frucht 5 und 6 ($Pflaume$ und $Weintraube$) in denselben Beobachtungen vorkommt, aber nur 6 eine von Donalds Wunschsorten ist. Somit kann nicht bestimmt werden, ob Schüssel 3 oder 6 die gewünschte Sorte enthält. In diesem Fall weiß Donald nur, dass sich in einer der beiden Schüsseln eine gewünschte Sorte befindet, in der anderen aber nicht.

1.4 Algorithmische Umsetzung

Nun wird kurz auf die algorithmische Umsetzung der Idee eingegangen:

1. Erstellen der A_i und B_j. Dazu werden die Beobachtungen nacheinander durchgegangen. Für jede Frucht i bzw. Schüssel j wird A_i bzw. B_j der Index der Beobachtung hinzugefügt.
   Bei Früchten muss der Name zunächst in eine Zahl umgewandelt werden. Im Material heißt es, dass jede Sorte mit einem anderen Buchstaben beginnt. Somit ließe sich z.B. der Index dieses Buchstaben im Alphabet verwenden. Allerdings soll das Programm als Erweiterung mit beliebig vielen Sorten zurechtkommen, die beliebige Namen haben. Die Umwandlung in eine Zahl geschieht also auf einem anderen Weg über eine `map`-Struktur:
   a) Setze einen Zähler auf 0
   b) Beim Umwandeln eines Fruchtnamen in eine Zahl: Überprüfe, ob der Name bereits in der `map` Vorhanden ist.
      - Falls ja: Gib den Wert zurück, der in der `map` für diesen Namen gespeichert ist
      - Falls nein: Erhöhe den Wert des Zählers um 1. Speichere in der `map` für den Namen den Wert des Zählers. Gib den Wert des Zählers zurück.
2. Es wird über alle A_i iteriert. Dabei werden die Mengen einer `map`-Datenstruktur als Schlüssel hinzugefügt. Im zugehörigen Wert wird gespeichert, wie viele der Früchte i mit dieser Menge A_i eine Wunschsorte sind und wie viele keine sind.
3. Danach wird über alle B_j iteriert und auf den entsprechenden Wert in der `map` zugegriffen.

- Falls die Menge zu keiner Wunschsorte gehört, sollte Donald auf keinen Fall zu Schüssel j gehen.
- Falls sie zu keiner Nichtwunschsorte gehört, sollte Donald auf jeden Fall zu Schüssel j gehen.
- Sonst kann keine genaue Angabe über die Schüssel gemacht. Später werden die Schüsseln mit derselben Menge B_j mit der Anzahl an gezählten Wunschsorten ausgegeben.

1.5 Laufzeit

Die Laufzeit hängt von der verwendeten Art ab, Mengen bzw. `map`s zu speichern. Basieren diese auf einem binären Suchbaum, benötigt Schritt 2 des obigen Algorithmus $\mathcal{O}((s+m)\log(m))$ Zeit. Dabei ist s die Anzahl der Früchte, die insgesamt auf einem Spieß sind. Um diese Laufzeit zu verstehen, müssen die Größen der A_i näher untersucht werden. Für jede Frucht i in der Eingabe wird A_i ein Element hinzugefügt. Somit ist $s = \sum_i |A_i|$ die Anzahl der Elemente, die insgesamt in den A_i vorhanden sind. Pro A_i, das der `map` hinzugefügt wird, müssen $\mathcal{O}(|A_i|\log(m))$ Operationen gemacht werden, da $\mathcal{O}(\log(m))$ Vergleiche benötigt werden, welche jeweils $\mathcal{O}(|A_i|)$ Zeit benötigen. Wird dies für alle A_i gemacht, erhält man die Laufzeit $\mathcal{O}((\sum_i |A_i|)\log(m)) = \mathcal{O}(s\log(m))$. Für den Fall einer leeren Menge A_i wird noch m zu s addiert. Denn leere Mengen benötigen benötigen immer noch $\mathcal{O}(\log(m))$ Schritte, um in der `map` gefunden zu werden, auch wenn sie nur $\mathcal{O}(1)$ Zeit für Vergleiche benötigen. Es gibt höchstens $\mathcal{O}(m)$ leere Mengen, für die zu s 1 addiert werden muss, womit die Addition von m zustande kommt.
Die identische Laufzeit gilt für Schritt 3. Die Mengen A_i, B_j können auch in $\mathcal{O}(s\log(m))$ Zeit erstellt werden[1]. Somit lassen sich alle Schritte der Lösung in $\mathcal{O}((s+m)\log(m))$ umsetzen.
Das ist bereits eine sehr gute Laufzeit, wenn man bedenkt, dass die Eingabe eine Größe von $\Omega(s)$ hat. Somit haben alle Lösungen des Problems eine Laufzeit von $\Omega(s)$. Die Lösung ist also nur um einen log Faktor langsamer als die theoretisch schnellstmögliche.
Es ist aber auch möglich eine Laufzeit von $\mathcal{O}(nm)$ zu erzielen. Für ein s, das die Größenordnung $\Theta(nm)$ besitzt, wäre dies eine Verbesserung ohne den log Faktor. Diese Optimierung macht die Lösung zwar komplexer und auch nur in Randfällen minimal schneller, verwendet aber dennoch ein paar interessante Ideen. Daher wird sie im Folgenden vorgestellt:

1.6 Lösung ohne log Faktor

1.6.1 Binärstrings

Die Anzahl möglicher Elemente in A_i und B_j ist mit n begrenzt ist. Es kann für jedes Element gespeichert werden, ob es vorkommt. Somit kann eine Menge auch als Binärstring betrachtet werden. Als Beispiel soll erneut Aufgabe a) betrachtet werden:

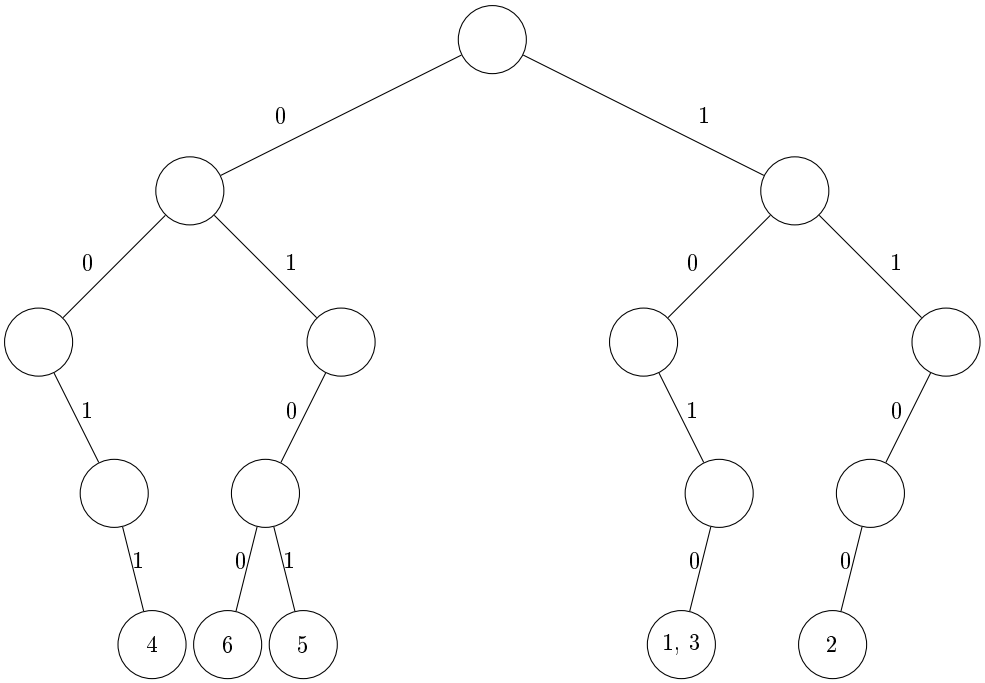
$$A_1 = \{1,3\} \mapsto 1010$$
$$A_2 = \{1,2\} \mapsto 1100$$
$$\ldots$$

Diese Repräsentation hat den Vorteil, dass in $\mathcal{O}(1)$ überprüft werden kann, ob sich ein Element in dieser Menge befindet. Allerdings besteht der Nachteil, dass immer $\mathcal{O}(n)$ Speicher benötigt wird.

1.6.2 Trie

Diese Lösung nutzt die **Trie**[2] Datenstruktur, die auch als Präfixbaum bekannt ist. Diese Datenstruktur ermöglicht das effiziente Hinzufügen von Strings zu einer Menge sowie das Überprüfen, ob sich eine String in der Menge befindet. Hierzu haben die Kanten des Baumes ein Label, das ein Zeichen aus dem Alphabet Σ der Strings ist. Ein String beschreibt, dann einen Pfad, der in $\mathcal{O}(n)$ abgelaufen werden kann, wenn n die Länge des Strings ist. In diesem Fall ist $\Sigma = \{0,1\}$, da die zu speichernden Mengen als Binärstring angesehen werden. Zusätzlich können in den einzelnen Knoten Werte gespeichert werden, die den Strings, die in diesen enden, zugeordnet werden. Dadurch kann ein Trie nicht nur ein `set` sondern auch eine `map` darstellen. Zur besseren Veranschaulichung soll ein Trie der Mengen A_i aus Teilaufgabe a) dienen. Die Blätter geben jeweils die Werte für i an, die zu dieser Menge führen:

[1] Für jede Frucht i in der Eingabe wird A_i der Index der entsprechenden Beobachtung hinzugefügt. Allerdings muss bei den Früchten noch ein Index i aus dem Fruchtnamen werden. Hierzu wird eine `map` verwendet und es sind $\mathcal{O}(\log(m))$ Vergleiche notwendig.
[2] https://en.wikipedia.org/wiki/Trie



Das Hinzufügen sowie das Finden einer Menge A_i, B_j ist in $\mathcal{O}(n)$ möglich, in dem ein Pfad der Länge n auf dem Trie abgelaufen wird. Insgesamt wird also nur $\mathcal{O}(nm)$ Zeit benötigt, um für jede der m Schüsseln zu entscheiden, ob sich dort auf jeden Fall eine Wunschsorte befindet, auf keinen Fall oder möglicherweise.
Es ist auch zu berücksichtigen, dass die Fruchtnamen irgendwie in Zahlen umgewandelt werden müssen. Dies lässt sich nicht einfach mit einer `map` machen, da so der log Faktor zurück wäre. Daher wird hierfür ein weiterer Trie verwendet. Dieses mal werden als Alphabet aber sämtliche ASCII-Zeichen anstelle von $\{0, 1\}$ verwendet. Die Knoten, bei denen ein Wort endet, speichern dann die entsprechende Id. So können in $\mathcal{O}(s) \subseteq \mathcal{O}(nm)$ alle Strings in die entsprechenden Ids umgewandelt werden.
Somit bleibt eine Gesamtlaufzeit von $\mathcal{O}(nm)$ erhalten.

1.7 Vergleich der Lösungen

Um die Laufzeit besser vergleichen zu können, wird das Ergebnis der Laufzeitanalyse der ersten Lösung anders dargestellt: $\mathcal{O}((s+m)\log(m)) = \mathcal{O}((\phi \cdot n + m)\log(m))$, wobei $\phi = \frac{s}{n}$ die durchschnittliche Anzahl an Früchten pro beobachtetem Spieß darstellt. Die Laufzeit der Trie-Lösung ist $\mathcal{O}(nm)$. Diese Laufzeit ist besser als die andere, falls $m < \phi \log(m)$, also falls die Anzahl an Früchten pro Spieß nur um einen kleinen konstanten Faktor von der gesamten Auswahl an Früchten abweicht[3].
Um einen besseren Vergleich der Laufzeit zu haben, werden beide Programme auf verschiedene Arten von Eingaben getestet. Das sind die Ergebnisse meiner Tests[4]:

n	m	ϕ	Zeit Lösung 1 (in s)	Zeit Lösung 2 (in s)
100	10^6	10	0.0265	0.6321
10	10^5	$5 \cdot 10^4$	0.4543	0.2450
100	10^4	5000	0.1995	0.1255
10^4	100	50	0.1161	0.1341

Die Daten zeigen, dass bei einem großen m und einem ϕ, welches nahe an diesem liegt, Lösung 2 mit der Laufzeit $\mathcal{O}(nm)$ schneller ist. Dieser Unterschied ist aber recht klein, da es sich nur um einen log Faktor handelt. Dieser Effekt ist bei kleinem m (letzte Messung) aber nicht mehr vorhanden. Hier ist sogar Lösung 1 minimal schneller.[5]
Allerdings ist Lösung 1 deutlich schneller, wenn ϕ klein ist und m groß (erste Messung). Lösung 2 ersetzt den Faktor $\phi \log(m)$ ja durch m. Wenn ϕ konstant ist, führt dies zur Ersetzung eines log Faktors durch m, was sich deutlich eher bemerkbar macht, als das Wegoptimieren eines log-Faktors.

[3] Funfact: Die Lösung ist besser, wenn $\frac{m}{\log(m)} \approx \pi(m) < \phi$, also im Schnitt mehr Früchte genommen werden als Primzahlen unter den ersten m natürlichen Zahlen sind.

[4] Dazu wurde das Skript `tester.py` verwendet. Es generiert zufällige Beispiele mit bestimmten Parametern. Die Fruchtnamen und Zahlen sind dabei identisch und es sind immer gleich viele Früchte in den Beispielen. Somit sind es nicht gerade real sinnvolle Beispiele, aber sie eignen sich zum Testen der Laufzeit.

[5] Entweder liegt das an den zeitlichen Ungenauigkeiten, die beim Messen / Ausführen entstehen oder es liegt am konstanten Faktor.

In der Realität ist aber wahrscheinlich ϕ konstant ist. Selbst, wenn es tausende von Früchten zur Auswahl gäbe[6], würde wahrscheinlich niemand auf die Idee kommen, einen Spieß mit 1000 Früchten essen...
Somit ist es in der Regel sinnvoller, die erste Lösung zu verwenden.

2 Umsetzung

2.1 Speichern der Mengen

In ersten Lösung werden die Mengen A_i und B_j als `vector<int>` gespeichert. Dies funktioniert, da die Beobachtungen in aufsteigender Reihenfolge bearbeitet werden und somit die A_i und B_j automatisch aufsteigend sortiert sind. Da mehrere A_i und B_j gespeichert werden müssen, werden insgesamt also `vector<vector<int>> A, B` gespeichert. Über `A[i]` bzw. `B[j]` wird dann auf A_i bzw. B_j zugegriffen. Daher wird mit 0-indizierten Werten gearbeitet. Somit muss von den Schüsselnummern jeweils 1 abgezogen werden, da diese in den BWINF-Beispielen 1-indiziert sind. Die Menge D der Wunschsorten wird als `set<int>` gespeichert, um in $\mathcal{O}(\log(m))$ überprüfen zu können, ob eine Frucht zu den Wunschsorten gehört.[7]
In der zweiten Lösung werden die Mengen A_i bzw. B_j, wie in der Lösungsidee beschrieben als Binärstring gespeichert. Präziser als Boolean-Vektor `vector<bool>` der intern als Bitset gespeichert wird. Zur Umwandlung eines `vector<int>` in diesen `vector<bool>` dient die Funktion `toBools`. Die Fruchtnamen werden hier über einen Trie in Zahlen umgewandelt.

2.2 Daten

Für jede Menge der A_i muss gespeichert werden, wie viele Früchte mit dieser Menge zu den Wunschsorten gehören und wie viele nicht. Diese Daten werden in der Klasse `Data` gespeichert:

```
struct Data
{
  int good = 0;
  int bad = 0;
  vector<int> group;
};
```

Dabei speichert `good` die Anzahl an Wunschsorten, `bad` die Anzahl der Nichtwunschsorten. `group` wird verwendet, um zu rekonstruieren, welche Schüsseln möglicherweise eine Wunschsorte enthalten. Falls sich beim Iterieren über die B_j herausstellt, dass sich sowohl gute als auch schlechte Sorten in diesen befinden könnten, werden die Schüsselnummern `group` hinzugefügt. Am Ende wird dann erneut über alle Mengen B_j iteriert. Fall sich Elemente in `group` befinden, werden diese zusammen mit der Anzahl an Wunschsorten unter ihnen ausgegeben. Dies bildet die „informative Ausgabe“. Danach wird `group` entleert, damit die Gruppen nicht mehrfach ausgegeben werden.
In der zweiten Lösung werden diese Daten als Variablen in der Knotenklasse gespeichert. Die Namen sind hier gleich.

2.3 Tries

Die Tries haben jeweils eine Klasse für ihre Knoten, nämlich `Node` und `StrNode`. Anstelle einer Pointerstruktur werden die Knoten in einem Vektor gespeichert und ein Trie-Knoten verweist auf den Index in dem Vektor. Diese Vektoren heißen `nodes` und `strNodes`. Man kann über die Funktionen `getNode` bzw. `getFruitIndex` auf die Trie-Elemente zugreifen. Falls das Element noch nicht vorhanden ist, wird es dynamisch hinzugefügt. Dabei gibt `getNode` den Knoten aus dem Trie der Personenmengen A_i zurück und `getFruitIndex` die Id der Frucht aus dem String-Trie.

[6] Wo auch immer man so viele unterschiedliche Früchte herbekommen möchte. In der Realität könnte es jedenfalls schwierig werden, mehr als 2000 aufzutreiben (`https://www.fruitsinfo.com/fruit-list.php#:~:text=There%20are%20around%202000%20types,uses%20only%2010%25%20of%20those.`)

[7] Dies könnte man hier auch über einen sortierten Vektor machen, da die A_i in aufsteigender Reihenfolge (nach i) durchgegangen werden. Dann könnte man einen Pointer auf dem Vektor weiter bewegen und schauen, ob das Element zu den Wunschsorten gehört. Allerdings ist in der Laufzeit so oder so ein log Faktor vorhanden, weshalb auf diese Optimierung verzichtet wird.

3 Beispiele

spiesse1.txt

```
Antwort gefunden:
1 2 4 5 7
```

spiesse2.txt

```
Antwort gefunden:
1 5 6 7 10 11
```

spiesse3.txt

```
Die Angaben reichen leider nicht aus!

Gehe zu diesen Schüsseln:
1 5 7 8 10 12
Gehe aber nicht zu diesen Schüsseln:
3 4 6 9 13 14 15
Die folgenden Schüsseln enthalten ein paar der Wunschsorten:
Anzahl Wunschsorten  | unter diesen Schüsseln
    1                | 2 11
```

spiesse4.txt

```
Antwort gefunden:
2 6 7 8 9 12 13 14
```

spiesse5.txt

```
Antwort gefunden:
1 2 3 4 5 6 9 10 12 14 16 19 20
```

spiesse6.txt

```
Antwort gefunden:
4 6 7 10 11 15 18 20
```

spiesse7.txt

```
Die Angaben reichen leider nicht aus!

Gehe zu diesen Schüsseln:
5 6 8 14 16 17 23 24
Gehe aber nicht zu diesen Schüsseln:
1 2 4 7 9 11 12 13 15 19 21 22
Die folgenden Schüsseln enthalten ein paar der Wunschsorten:
Anzahl Wunschsorten  | unter diesen Schüsseln
    3                | 3 10 20 26
    1                | 18 25
```

Da alle BWINF-Beispiele sehr klein sind, wurde auch noch ein eigenes größeres Beispiel mit $n = 10000$, $m = 1000$ generiert und gelöst. Es wäre aber zu groß, um es hier abzudrucken. Daher finden Sie es in den Beispielen in der Datei `big.txt`

4 Quellcode

4.1 Erstellen der Mengen A_i, B_j

```
// Beobachtungen, in denen die Frucht bzw. Schüssel vorkam
vector<vector<bool>> A, B;

...

A.resize(m); B.resize(m);
for(int i = 0; i < n; ++i)
{
  vector<int> nums, fruits;
  nums = readNumbers();
  fruits = readFruits();
  for(int j : fruits) A[j].push_back(i);
  for(int j : nums)   B[j].push_back(i);
}
```

4.2 Lösen der Aufgabe in `solve`

```
struct Data
{
  int good = 0;
  int bad = 0;
  vector<int> group;
};

void solve()
{
  // Iteriere über alle Früchte
  map<vector<int>, Data> f;
  for(int i = 0; i < m; ++i)
  {
    // Will Donald diese Frucht?
    bool ans = donald.count(i);
    Data& d = f[A[i]];
    if(ans) d.good++;
    else d.bad++;
  }

  // Schüsseln, die garantiert zu Donalds Wunschsorten gehören oder auf keinen Fall
  vector<int> take, dont;
  // Iteriere über alle Schüsseln
  for(int i = 0; i < m; ++i)
  {
    Data& d = f[B[i]];
    if(d.good == 0) dont.push_back(i);
    // Falls nur Wunschsorten unter diesen waren, sollte Donald sie auf jeden Fall nehmen
    else if(d.bad == 0) take.push_back(i);
    // Der Fall, in dem nur manche Früchte gewünscht wurden, wird später behandelt
    // Dazu wird die Information benötigt, welche Schüsseln in diese Kategorie fallen
    else d.group.push_back(i);
  }

  // ============= Output ================

  // Falls alle Wunschsorten eindeutig gefunden werden konnten, gib diese aus
  if(sz(take) == cnt_donald)
  {
    cout << "Antwort gefunden: \n";
    for(int i : take)
    {
      cout << i+1 << " ";
    }
    cout << "\n";
  }
  // Mache sonst die "Informative Ausgabe"
  else
```

```
  {
    cout << "Die Angaben reichen leider nicht aus!\n\n";
    cout << "Gehe zu diesen Schüsseln:\n";
    for(int i : take)
    {
      cout << i+1 << " ";
    }
    cout << "\nGehe aber nicht zu diesen Schüsseln:\n";
    for(int i : dont)
    {
      cout << i+1 << " ";
    }
    cout << "\nDie folgenden Schüsseln enthalten ein paar der Wunschsorten:\n";
    cout << "Anzahl Wunschsorten\t | unter diesen Schüsseln\n";
    // Gehe erneut alle Schüsseln durch
    for(int i = 0; i < m; ++i)
    {
      Data& d = f[B[i]];
      if(sz(d.group))
      {
        cout << "\t" << d.good << "\t\t\t\t | ";
        for(int j : d.group)
        {
          cout << j + 1 << " ";
        }
        cout << "\n";
        // Lösche die Daten, damit die Mengen nicht mehrfach ausgegeben werden
        d.group.clear();
      }
    }
  }
}
```

4.3 Eingabe

```
// Util Funktion für den Input
// Liest eine ganze Zeile und trennt den Input nach Leerzeichen
// T muss dazu kompatibel mit dem std::stringstream sein!
template<typename T>
vector<T> readLine()
{
  string line;
  getline(cin, line);
  stringstream ss(line);
  vector<T> vec;
  T val;
  while(ss >> val) vec.push_back(val);
  return vec;
}

// Komprimiert einen Vector mit Fruchtnamen in einen, der ihre Indizes enthält
vector<int> compress(const vector<string>& fruitstr)
{
  vector<int> vec;
  for(int i = 0; i < sz(fruitstr); ++i)
  {
    if(!f2i.count(fruitstr[i])) f2i.emplace(fruitstr[i], sz(f2i));
    vec.push_back(f2i[fruitstr[i]]);
  }
  return vec;
}

// Liest eine Zeile mit Fruchtnamen ein und bringt die Menge in das Bitstring Format
vector<int> readFruits()
{
  return compress(readLine<string>());
}
// Liest eine Zeile mit Schüsseln ein und bringt die Menge in das Bitstring Format
vector<int> readNumbers()
{
```

```
  vector<int> ints = readLine<int>();
  for(int i = 0; i < sz(ints); ++i) ints[i]--; // Input ist 1-indiziert
  return ints;
}


// Gibt einen BitVekor zurück, der die Bits an den Indices gesetzt hat, die den int Werten entsprechen.
vector<bool> toBools(const vector<int>& ints)
{
  vector<bool> res(m, false);
  for(int i : ints)
  {
    res[i] = true;
  }
  return res;
}
```

In der zweiten Lösung wird die Funktion `toBools` auf das Ergebnis von `readFruits` bzw. `readNumbers` angewendet.

4.4 Tries

```
// Trie Knoten
struct Node
{
  int next[2] = {-1, -1}; // Die Indices der Kindknoten
  // Falls der Knoten ein Blatt ist:
  int good = 0;
  int bad = 0;
  // Schüsseln, die diese Personenmenge haben
  vector<int> group;
};
// Enthält die Knoten des Tries
vector<Node> nodes;

// Falls die args im Trie ist:
//       Gibt den Wert des zugehörigen Blattes zurück
// sonst:
//       Fügt args zum Trie hinzu und gibt den Wert des neu erstellten Blattes zurück
Node& getNode(const vector<bool>& args)
{
  int cur = 0;
  for(int i = 0; i < sz(args); ++i)
  {
    if(nodes[cur].next[args[i]] != -1)
    {
      cur = nodes[cur].next[args[i]];
    }
    else
    {
      nodes[cur].next[args[i]] = sz(nodes);
      cur = sz(nodes);
      nodes.push_back(Node());
    }
  }
  return nodes[cur];
}

// Knoten für den String Trie. Wird zum Komprimieren der Strings verwendet.
struct StrNode
{
  int next[128] = {};
  int id = -1;
};
// Enthält die Knoten des String Tries
vector<StrNode> strNodes;
// Counter für die Id der komprimierten Fruchtnamen
int strId = 0;
// vergleiche getNode. Gibt die Id zurück
int getFruitIndex(const string& fruit)
{
```

```
50    int cur = 0;
51    for(char c : fruit)
52    {
53      int nxt = (int)c;
54      if(strNodes[cur].next[nxt] == 0)
55      {
56        strNodes[cur].next[nxt] = sz(strNodes);
57        strNodes.push_back(StrNode());
58      }
59      cur = strNodes[cur].next[nxt];
60    }
61    if(strNodes[cur].id == -1)
62    {
63      strNodes[cur].id = strId++;
64    }
65    return strNodes[cur].id;
66  }
```

Die Implementation der Lösung und von `compress` sind fast identisch. Der einzige unterschied ist die Verwendung von `getNode()` statt `f[]`. Daher wird diese Version nicht erneut abgedruckt.