# Aufgabe 2: Spießgesellen

Teilnahme-ID: 56905
Chuyang Wang

7. April 2021

## Inhaltsverzeichnis

# 1 Lösungsidee

Das in der Aufgabe beschriebene Problem kann generell als ein (spezielle) lineares Gleichungssystem angesehen und mit dementsprechendem Verfahren gelöst werden. Im Folgenden wird mithilfe des Beispiels vom Aufgabe Teil a) erläutertet, wie man die gegebenen Daten in ein lineares Gleichungssystem umwandelt, wie man es löst und wie die Antwort gelesen werden kann.

In der Aufgabe 2.a) sind folgende Daten gegeben (hier werden sie vereinfacht dargestellt):

$$\begin{aligned} \{\text{Apfel, Banane, Brombeere}\} &\Rightarrow \{\text{Schüssel 1, Schüssel 4, Schüssel 5}\} \\ \{\text{Banane, Pflaume, Weintraube}\} &\Rightarrow \{\text{Schüssel 3, Schüssel 5, Schüssel 6}\} \\ \{\text{Apfel, Brombeere, Erdbeere}\} &\Rightarrow \{\text{Schüssel 1, Schüssel 2, Schüssel 4}\} \\ \{\text{Erdbeere, Pflaume}\} &\Rightarrow \{\text{Schüssel 2, Schüssel 6}\} \end{aligned} \tag{1}$$

Zur weiteren Vereinfachung kann man die Namen von Obst und Schüsseln mit Variablen ersetzen. Sei

$$\begin{aligned} x_1 &:= \text{Apfel}, x_2 := \text{Banane}, x_3 := \text{Brombeere} \\ x_4 &:= \text{Pflaume}, x_5 := \text{Weintraube}, x_6 := \text{Erdbeere} \end{aligned} \tag{2a}$$

und

$$s_i := \text{Schüssel i} \qquad \forall\, i \subseteq \mathbb{N}_{\leq 6} \tag{2b}$$

Somit erhalten wir aus (1) das folgende LGS:

$$\begin{aligned} x_1 + x_2 + x_3 &= s_1 + s_4 + s_5 \\ x_2 + x_4 + x_5 &= s_3 + s_5 + s_6 \\ x_1 + x_3 + x_6 &= s_1 + s_2 + s_4 \\ x_4 + x_6 &= s_2 + s_6 \end{aligned} \tag{3}$$

Man kann das LGS in erweiterte Matrix darstellen:

$$\begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \end{array} \begin{array}{c} \begin{array}{cccccc|cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{array} \\ \left[\begin{array}{cccccc|cccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right] \end{array} \tag{4}$$

$R_5$ ist durch die Aufgabe impliziert. Es ist ersichtlich, dass wenn man alle Obst bekommen möchte, dann muss man auch zu allen Schüsseln gehen.

Das LGS wird durch das Gauß-Jordan-Verfahren gelöst bzw. in die reduzierte Zeilenstufenform gebracht. Besonders ist aber, sofern durch eine Zeilenoperation negative Zahlen entstehen, wird die jetzige Zeile in zwei Zeilen geteilt, jeweils eine Zeile mit allen positiven Spalten und die andere mit allen negativen Spalten. Die Zeile mit negativen Spalten wird mit -1 multipliziert. Nachdem eine Spalte durch Zeilenoperationen in eine Pivotspalte umgewandelt wird, wird die Matrix sortiert, sodass es in der Zeilenstufenform bleibt. Am Anfang wie in (4) muss die Matrix natürlich auch sortiert werden, diese wird hier jedoch nicht nochmal gezeigt.

Es wird versucht, für jede Spalte ein Pivotelement zu finden. Ein Koeffizient der Zeile ist dann ein Pivotelement, wenn 1) er nicht null ist; und 2) alle andere Elemente dieser Zeile links vor dieser Spalte null sind.

In diesem Fall nehmen wir $a_{11}$ als das Pivotelement der Spalte $x_1$ und eliminieren alle anderen Nicht-Null-Elemente in dieser Spalte. Hier ist bei $a_{31}$ und $a_{51}$ der Fall, also muss man die Zeilenoperation $R_3 = R_1 - R_3$ und $R_5 = R_1 - R_5$ durchführen. Nun sieht die Matrix so aus:

$$\begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \end{array} \begin{array}{c} \begin{array}{cccccc|cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{array} \\ \left[\begin{array}{cccccc|cccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & -1 \end{array}\right] \end{array} \tag{5}$$

Wie oben beschrieben sollen die Zeilen aus (5), in denen negative Elementen auftauchen, also $R_3$ und $R_5$, in zwei Zeilen geteilt werden und die negativen Zeilen davon werden mit -1 multipliziert. Zeilen mit lediglich Null-Elemente und duplizierte Zeilen werden aus der Matrix entfernt. Die Zeilen werden nach Anzahl, wie viele Null-Koeffizienten es von links bis einen Nicht-Null-Koeffizient gibt, sortiert. Also:

$$\begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \\ R_6 \end{array} \begin{array}{c} \begin{array}{cccccc|cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{array} \\ \left[\begin{array}{cccccc|cccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \tag{6}$$

$R_3$ und $R_6$ stammen aus $R_3$ von (5). $R_5$ aus (5) wäre auch in zwei Zeilen geteilt worden, da es jedoch nur negative Koeffizienten gibt, wird die geteilte positive Zeile nur mit 0 gefüllt und somit von der Matrix entfernt. Die geteilte negative Zeile wird mit -1 multipliziert und wird die $R_5$ aus (6).

Danach wird dieselbe für alle anderen Spalten durchgeführt. Für jede Spalte sollte ein Pivotelement gefunden werden können, es sei denn, nicht jedes Obst kann einer eindeutigen Schüssel zugeordnet wird. Passiert das, heißt jedoch nicht, dass es keine eindeutige Antwort für die Frage gefunden werden kann. Es kann durchaus sein, dass zwei Obstsorten zu zwei Schüsseln zugeordnet werden, wobei die beiden Obstsorten nachgefragt werden.

Nach dem Lösen wird diese Matrix so aussehen:

$$\begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \end{array} \left[\begin{array}{cccccc|cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \tag{7}$$

Wenn wir die obige Matrix wieder in Obstnamen schreiben würden, dann heißt es:

$$\begin{aligned} \{\text{Apfel, Brombeere}\} &\Rightarrow \{\text{Schüssel 1, Schüssel 4}\} \\ \{\text{Banane}\} &\Rightarrow \{\text{Schüssel 5}\} \\ \{\text{Pflaume}\} &\Rightarrow \{\text{Schüssel 6}\} \\ \{\text{Weintraube}\} &\Rightarrow \{\text{Schüssel 3}\} \\ \{\text{Erdbeere}\} &\Rightarrow \{\text{Schüssel 2}\} \end{aligned} \tag{8}$$

Donald möchte gerne Weintraube, Brombeere und Apfel haben, also

$$\{\text{Apfel, Weintraube, Brombeere}\} \Rightarrow \{\text{Schüssel 1, Schüssel 3, Schüssel 4}\} \tag{9}$$

muss er zu Schüsseln 1, 3, und 4 gehen.

Somit ist die Aufgabe a) gelöst. Die Zeitkomplexität dieses Verfahrens wird am Ende der Umsetzung erläutert.

# 2 Umsetzung

Das Programm wird in C# .Net Core 3.1 umgesetzt. Der Name der Test-Datei muss beim Durchführen des Programms eingegeben werden.

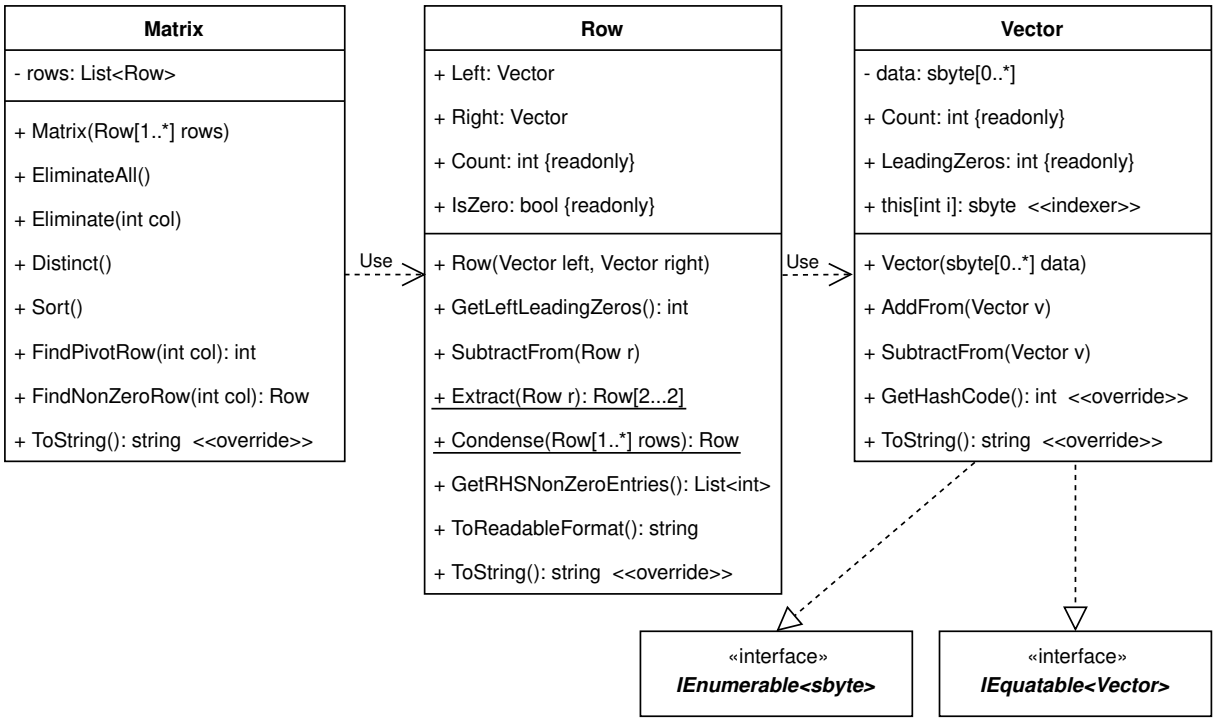
In der `Main()` Methode wird zunächst die Test-Datei eingelesen. Danach wird die neu entstehende Matrix gelöst und die Information analysiert und ausgegeben.

```
// public static void Main(string[] args)
Tuple<bool, int, string[], Dictionary<string, int>,
    string[], Matrix> data = ReadInput();
    if (data.Item1)
    {
      Matrix matrix = data.Item6;
      matrix.Sort();
      matrix.EliminateAll();
      ......
```

Durch die Methode, `ReadInput()`, wird die Test-Datei eingelesen und in das Matrixformat, das in der Lösungsidee beschrieben wurde, transformiert.

```
// Fordert den Nutzer auf, einen Dateinamen einzugeben, liest diese,
// und parst in eine Matrix
// RETURN:
// bool Item1 - ob die Datei erfolgreich eingelesen ist
// int Item2 - Anzahl der verfuegbaren Obstsorten
// string[] Item3 - Wunschsorten
// Dictionary<string, int> Item4
// string[] Item5
// Matrix Item6 - Matrix wie in Loesungsidee beschrieben
private static Tuple<bool, int, string[],
  Dictionary<string, int>, string[], Matrix> ReadInput()
```

Die Klasse `Matrix` wird wie Folgende implementiert:




Um die Matrix zu lösen, muss alle Nicht-Pivotelement der Pivotspalten eliminiert werden. Dies erfolgt durch die Methode `ElinimateAll()`.

```
// Loest die Matrix
public void EliminateAll()
{
  for (int i = 0; i < rows[0].Count; i++)
    Eliminate(i);
}
```

Die Methode, `Eliminate(int col)`, versucht, ein Pivotelement für die vorgegebene Spalte zu finden. Falls es möglich ist, wird alle Elemente dieser Spalte, außer dem Pivot, eliminiert. Das Pivotelement bzw. die Pivotzeile wird mithilfe der Methode `FindPivotRow(int col)` gefunden.

```
// Versucht, das Pivotelement dieser Spalte zu finden
// Falls es nicht moeglich ist, dann gib -1 zurueck
private int FindPivotRow(int col)
{
  for (int i = 0; i < rows.Count; i++)
  {
    if (rows[i].Left[col] != 0 && rows[i].Left.LeadingZeros == col)
      return i;
  }
  return -1;
}
```

Innerhalb der Methode `Eliminate(int col)` wird die obige Methode aufgerufen. Ist das Ergebnis -1, wird die Methode terminiert.

```
// ObstspiessSolver.Matrix.Eliminate
int pivotRow = FindPivotRow(col);
if (pivotRow == -1)
{
  return;
}
```

Ist es nicht der Fall, dann wird die Variable `bool isUnique` angelegt und der ein Anfangswert von `true` gegeben. Diese Variable gibt an, ob es überhaupt Zeilenoperationen für diese Spalte durchgeführt werden muss und wird zu `false` gesetzt, wenn mindestens eine Zeilenoperation stattfindet.

```
// ObstspiessSolver.Matrix.Eliminate
bool isUnique = true;
for (int r = 0; r < rows.Count; r++)
{
  if (r == pivotRow) continue;
  if (rows[r].Left[col] != 0)
  {
    rows[r].SubtractFrom(rows[pivotRow]);
    isUnique = false;
  }
}
```

Kurz zu der Methode `Row.SubtractFrom(Row r)`:

```
// ObstspiessSolver.Row
// Die jetzige Zeile (Minuend)
// wird von der gegebene Zeile (Subtrahenden) subtrahiert
// Die Differenz wird der neue Wert des jetzigen Objekts
public void SubtractFrom(Row r)
{
  left.SubtractFrom(r.Left);
  right.SubtractFrom(r.Right);
}
```

Falls Zeilenoperationen durchgeführt worden sind, dann wird jede Zeile in positiv und negativ geteilt (vgl. Lösungsidee). Die Negative wird mit -1 multipliziert und wird somit positiv. Danach werden Duplikate entfernt (durch `Distinct()`) und die Matrix wird wieder sortiert (durch `Sort()`).

```
// ObstspiessSolver.Matrix.Eliminate
if (!isUnique)
{
  List<Row> rs = new List<Row>();
  for (int i = 0; i < rows.Count; i++)
  {
    rs.AddRange(Row.Extract(rows[i]));
  }
  rows = rs;
  // Entfernt Duplikate und die Null-Zeile
```

```
    Distinct();
    // Sortiert nach Anzahl von Nulls jeder Zeile
    Sort();
}
```

Die statische Methode `Row.Extract(Row r)`:

```
// Teilt die gegebene Zeile in positive und negative Zeilen
// Die negative Zeile wird zu positiv umgewandelt
// indem *-1 durchgefuehrt wird
// BSP: Eingabe: {1,0,0,-1,1} => {0,1,1,0,-1}
//      Ausgabe: [{1,0,0,0,1} => {0,1,1,0,0},
//                {0,0,0,1,0} => {0,0,0,0,1}]
public static Row[] Extract(Row r)
```

Somit terminiert die Methode `Eliminate(int col)`.

Dann wird die `Main()` Methode weitergeführt. An dieser Stelle ist die Aufgabe (fast) gelöst, jedoch muss die Matrix noch analysiert werden und als lesbares Format ausgegeben. Die Zeilen aus `resultRowsWithConflict` werden zu einer einzigen Zeile zusammengefügt und somit werden die gesuchte Schüsseln auf der rechten Seite (vgl. erweiterte Matrix in Lösungsidee) gelesen. Dieser Bereich wird ausführlich kommentiert und ist dem Quellcode zu entnehmen.

```
// public static void Main(string[] args)
#region  Analysiere Data
......
#endregion
```

Die Ausgabe (vgl. `Quellcode -> Main() -> #region Output`) wird hier nicht im Einzelnen dargestellt.

## 2.1 Zeitkomplexität

Das Gauß-Jordan-Verfahren hat typischerweise eine Zeitkomplexität von $O(n^3)$. Bei der Umsetzung ist `Matrix.EliminateAll()` die Methode, die die Matrix löst und wird hier detailliert betrachtet.

```
// O(n^3)
public void EliminateAll()
{
  for (int i = 0; i < rows[0].Count; i++) // O(n)
    Eliminate(i); // O(n^2*n) = O(n^3)
}
```

Die Zeitkomplexität von der Methode `Eliminate(int col)` ist $O(n^2)$. Hier werden insbesondere die Schleife-Strukturen gezeigt.

```
// O(n^2)
public void Eliminate(int col)
{
  int pivotRow = FindPivotRow(col); // O(n)
  ...
  for (int r = 0; r < rows.Count; r++) // O(n)
  {
    ...
    rows[r].SubtractFrom(rows[pivotRow]); // O(n)
  } // O(n^2)

  for (int i = 0; i < rows.Count; i++) // O(n)
  {
    rs.AddRange(Row.Extract(rows[i])); // O(n)
  } // O(n^2)
  Distinct(); // O(nlogn)
  Sort(); // O(n^2)
}
```

Die anderen Teile des Programms weisen keine höhere Zeitkomplexität auf und somit ist das gesamte Programm $O(n^3)$ für $n \to \text{inf}$. Die Variable $n$ ist hierbei die Anzahl von Obstsorten.

# 3 Beispiele

Beim Ausführen des Programms wird der Nutzer aufgefordert, den Namen der Datei einzugeben. Die Dateien sollen in demselben Orden wie das Programm stehen. Die erwartete Eingabe wird unten bei dem ersten Beispiel blau markiert.

## 3.1 spiesse0.txt

Gegeben, entspricht dem Beispiel in der Aufgabe. Gemessene Laufzeit: 12ms.

Kommandozeile-Ausgabe:

```
Bitte Name der Test-Datei eingeben...
spiesse0.txt

Gesucht: Weintraube,Brombeere,Apfel

Eine eindeutige Antwort wurde gefunden: 1,3,4

----------------
Weitere Einzelheiten:
Weintraube => 3
Apfel, Brombeere => 1, 4

Drueck eine beliebige Taste zu schliessen...
```

## 3.2 spiesse1.txt

Gegeben. Größeres Beispiel. Gemessene Laufzeit: 12ms.

Kommandozeile-Ausgabe:

```
Gesucht: Clementine,Erdbeere,Grapefruit,Himbeere,Johannisbeere

Eine eindeutige Antwort wurde gefunden: 1,2,4,5,7

----------------
Weitere Einzelheiten:
Clementine => 1
Erdbeere, Himbeere => 2, 4
Grapefruit => 7
Johannisbeere => 5
```

## 3.3 spiesse2.txt

Gegeben. Größeres Beispiel. Gemessene Laufzeit: 13ms.

Kommandozeile-Ausgabe:

```
Gesucht: Apfel,Banane,Clementine,Himbeere,Kiwi,Litschi

Eine eindeutige Antwort wurde gefunden: 1,5,6,7,10,11

----------------
Weitere Einzelheiten:
Apfel => 1
Banane, Clementine, Himbeere => 5, 10, 11
Kiwi => 6
Litschi => 7
```

## 3.4 spiesse3.txt

Gegeben. Beispiel für nicht eindeutige Eingabe. Da Grapefruit und Litschi in der Datei immer gleichzeitig auftauchen, ist es unmöglich, genau zu wissen, wo Litschi allein hingehört. Gemessene Laufzeit: 14ms.

Kommandozeile-Ausgabe:

```
Gesucht: Clementine,Erdbeere,Feige,Himbeere,Ingwer,Kiwi,Litschi

Keine eindeutige Antwort konnte gefunden werden:
Schuesseln, die besucht werden muessen, um ALLE gewuenschten
Obst zu bekommen (aber moeglicherweise kann man auch weitere
unerwuenschte Obstsorte bekommen):
1,2,5,7,8,10,11,12

Schuesseln, die besucht werden muessen, um NUR gewuenschte
Obst zu bekommen (aber moeglicherweise kann man nicht alle
gewuenschten Obstsorten bekommen):
1,5,7,8,10,12

----------------
Weitere Einzelheiten:
(Zeilen, in denen keine eindeutige Antwort fuer die gesuchten
Obstsorten gefunden werden kann, werden mit "**" am Anfang
der Zeile markiert)

Clementine => 5
Erdbeere => 8
Feige, Ingwer => 7, 10
Himbeere => 1
Kiwi => 12
**Grapefruit, Litschi => 2, 11
```

## 3.5 spiesse4.txt

Gegeben. Größeres Beispiel. Gemessene Laufzeit: 14ms.

Kommandozeile-Ausgabe:

```
Gesucht: Apfel,Feige,Grapefruit,Ingwer,Kiwi,Nektarine,Orange,Pflaume

Eine eindeutige Antwort wurde gefunden: 2,6,7,8,9,12,13,14

----------------
Weitere Einzelheiten:
Apfel => 9
Feige => 13
Grapefruit => 8
Ingwer => 6
Kiwi => 2
Nektarine => 7
Orange => 14
Pflaume => 12
```

## 3.6 spiesse5.txt

Gegeben. Größeres Beispiel. Gemessene Laufzeit: 15ms.

Kommandozeile-Ausgabe:

```
Gesucht: Apfel,Banane,Clementine,Dattel,Grapefruit,Himbeere,
Mango,Nektarine,Orange,Pflaume,Quitte,Sauerkirsche,Tamarinde

Eine eindeutige Antwort wurde gefunden: 1,2,3,4,5,6,9,10,12,
14,16,19,20

----------------
Weitere Einzelheiten:
Apfel, Grapefruit, Mango => 1, 4, 19
Banane, Quitte => 3, 9
Clementine => 20
Dattel => 6
Himbeere => 5
Nektarine => 14
Orange, Sauerkirsche => 2, 16
Pflaume => 10
Tamarinde => 12
```

## 3.7 spiesse6.txt

Gegeben. Größeres Beispiel. Gemessene Laufzeit: 14ms.

Kommandozeile-Ausgabe:

```
Gesucht: Clementine,Erdbeere,Himbeere,Orange,Quitte,Rosine,
Ugli,Vogelbeere

Eine eindeutige Antwort wurde gefunden: 4,6,7,10,11,15,18,20
----------------
Weitere Einzelheiten:
Clementine => 7
Erdbeere => 10
Himbeere => 18
Orange => 20
Quitte => 4
Rosine, Ugli => 11, 15
Vogelbeere => 6
```

## 3.8 spiesse7.txt

Gegeben. Größeres Beispiel mit nicht eindeutiger Eingabe. Da Banane und Ugli immer gleichzeitig auftauchen, ist es also unmöglich, genau zu wissen, in welcher Schüssel sich Ugli befindet. Gleiches Argument gilt für Apfel, Grapefruit und Xenia. Gemessene Laufzeit: 14ms.

Kommandozeile-Ausgabe:

```
Gesucht: Apfel,Clementine,Dattel,Grapefruit,Mango,Sauerkirsche,
Tamarinde,Ugli,Vogelbeere,Xenia,Yuzu,Zitrone

Keine eindeutige Antwort konnte gefunden werden:
Schuesseln, die besucht werden muessen, um ALLE gewuenschten
Obst zu bekommen (aber moeglicherweise kann man auch weitere
unerwuenschte Obstsorte bekommen):
3,5,6,8,10,14,16,17,18,20,23,24,25,26

Schuesseln, die besucht werden muessen, um NUR gewuenschte
Obst zu bekommen (aber moeglicherweise kann man nicht alle
gewuenschten Obstsorten bekommen):
5,6,8,14,16,17,23,24

----------------
Weitere Einzelheiten:
(Zeilen, in denen keine eindeutige Antwort fuer die gesuchten
Obstsorten gefunden werden kann, werden mit "**" am Anfang der
Zeile markiert)

**Apfel, Grapefruit, Litschi, Xenia => 3, 10, 20, 26
Clementine => 24
Dattel, Mango, Vogelbeere => 6, 16, 17
Sauerkirsche, Yuzu => 8, 14
Tamarinde, Zitrone => 5, 23
**Banane, Ugli => 18, 25
```

# 4 Quellcode

Unwichtige Quellcode-Elemente werden hier nicht gezeigt und werden durch ..... ersetzt. Der vollständige Quelltext ist in dem Ordner `Aufgabe2/Quelltext/Aufgabe2.cs` zu finden.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace Aufgabe2
{
  public static class ObstspiessSolver
  {
    public static void Main(string[] args)
    {
      ......
      Tuple<bool, int, string[], Dictionary<string, int>,
      string[], Matrix> data = ReadInput();
      // Falls das Einlesen von Daten erfolgreich ist
      if (data.Item1)
      {
        Matrix matrix = data.Item6;
        matrix.Sort();
        matrix.EliminateAll();

        #region  Analysiere Data
        // Spalte (Obstsorte), die von der Aufgabe gefragt sind
        int[] entriesToFind = data.Item3.
          Select(s => data.Item4[s]).ToArray();
        List<Row> resultRowsWithConflict = new List<Row>();
        // Bezieht sich auf resultRowsWithConflict
        // Speichert den Index, falls diese Zeile nicht nur Wunschsorten,
        // Sondern auch andere nicht gewuenschte Sorten enthalten
        List<int> conflictIndex = new List<int>();
        for (int i = 0; i < entriesToFind.Length; i++)
        {
          // Um Duplikate zu vermeiden
          if (resultRowsWithConflict.All(e => e.Left[entriesToFind[i]] == 0))
          {
            // Speichert die Zeile, in denen
            // das Element in dieser Spalte nicht null ist
            resultRowsWithConflict.Add(
              matrix.FindNonZeroRow(entriesToFind[i]));
            // Testet, ob diese Zeile nicht gewuenschte Obstsorte enthaelt
            for (int j = 0; j < resultRowsWithConflict[^1].Count; j++)
            {
              if (resultRowsWithConflict[^1].Left[j] != 0 &&
               !entriesToFind.Contains(j))
              {
                conflictIndex.Add(resultRowsWithConflict.Count - 1);
                // Eine Nicht-Uebereinstimmung reicht
                break;
              }
            }
          }
        }
```

```
54          // ResC enthaelt auch nicht-gewuenschte Obstsorten
            Row resC = Row.Condense(resultRowsWithConflict.ToArray());
56          // ResO enthaelt nur Wunschsorten
            // Jedoch koennte es sein,
58          // dass nicht alle Wunschsorten vorhanden sind
            Row resO = Row.Condense(Enumerable.
60            Range(0, resultRowsWithConflict.Count).
              // wo dieser Index nicht in conflictIndex vorhanden ist
62            Where(idx => !conflictIndex.Contains(idx)).
              Select(i => resultRowsWithConflict[i]).
64            ToArray());
            // Schuesseln von ResC
66          List<int> schuesselWithConflict = resC.GetRHSNonZeroEntries();
            // Schuesseln von ResO
68          List<int> schuesselWithoutConflict =
              resO.GetRHSNonZeroEntries();
70          #endregion

72          #region OutPut
            ......
74          #endregion
        }
76      else
        {
78        ......
        }
80      ......
    }
82
    // Fordert den Nutzer auf, einen Dateinamen einzugeben, liest diese,
84  // und parst in eine Matrix
    // RETURN:
86  // bool Item1 - ob die Datei erfolgreich eingelesen ist
    // int Item2 - Anzahl der verfuegbaren Obstsorten
88  // string[] Item3 - Wunschsorten
    // Dictionary<string, int> Item4
90  // string[] Item5
    // Matrix Item6 - Matrix wie in Loesungsidee beschrieben
92  private static Tuple<bool, int, string[],
      Dictionary<string, int>, string[], Matrix> ReadInput()
94    ......

96  private class Matrix
    {
98    private List<Row> rows;
      public Matrix(Row[] rows)
100     ...... // Nichts Besonders

102   // Loest die Matrix
      // O(n^3)
104   public void EliminateAll()
      {
106     for (int i = 0; i < rows[0].Count; i++) // O(n)
          Eliminate(i); // O(n^2*n) = O(n^3)
108   }

110   // Versucht, ein Pivotelement in dieser Spalte zu finden
```

```
    // Erfolgt es, wird alle andere Elemente eliminiert
112 // Sonst return
    // O(n^2)
114 public void Eliminate(int col)
    {
116   int pivotRow = FindPivotRow(col); // O(n)
      if (pivotRow == -1)
118     return;

120   bool isUnique = true;
      for (int r = 0; r < rows.Count; r++) // O(n)
122   {
        if (r == pivotRow) continue;
124     if (rows[r].Left[col] != 0)
        {
126       // Eliminiert andere Elemente dieser Spalte
          rows[r].SubtractFrom(rows[pivotRow]); // O(n)
128       isUnique = false;
        }
130   } // O(n^2)

132   // Falls eine Elimination stattfindet
      if (!isUnique)
134   {
        List<Row> rs = new List<Row>();
136     for (int i = 0; i < rows.Count; i++) // O(n)
        {
138       rs.AddRange(Row.Extract(rows[i])); // O(n)
        } // O(n^2)
140     rows = rs;
        Distinct(); // O(nlogn)
142     Sort(); // O(n^2)
      }
144 }

146 // Entfernt Duplikate und die Null-Zeile
    // O(nlogn)
148 public void Distinct()
    {
150   rows = rows.Where(e => !e.IsZero).
        GroupBy(r => r.Left.GetHashCode()).
152     Select(e => e.First()).
        ToList();
154 }

156 // Bubble sort
    // Sortiert nach Anzahl von Nulls jeder Zeile
158 // O(n^2)
    public void Sort()
160   ......

162 // Versucht, das Pivotelement dieser Spalte zu finden
    // Die Pivotzeile wird zurueckgegeben
164 // Falls es nicht moeglich ist, dann gib -1 zurueck
    // O(n)
166 private int FindPivotRow(int col)
    {
168   for (int i = 0; i < rows.Count; i++)
```

```
        {
          if (rows[i].Left[col] != 0 &&
            rows[i].Left.LeadingZeros == col)
            return i;
        }
        return -1;
      }

      // Nachdem die Matrix bereits in reduzierte ZSF ist
      // Findet das Element dieser Spalte, welches nicht null ist
      // RETURN:
      // Die ganze Zeile dieses Elements
      public Row FindNonZeroRow(int col)
        ..... // Einfach eine For-Schleife zum Suchen

      ..... // ToString()
    }

    private class Row
    {
      private readonly Vector left;
      private readonly Vector right;
      public Vector Left { get => left; }
      public Vector Right { get => right; }
      public int Count { get => left.Count; }
      // True falls alle Elemente dieser Zeile sind null
      public bool IsZero
        .....

      public Row(Vector left, Vector right)
        ..... // Nichts besonders

      public int GetLeftLeadingZeros()
      {
        return left.LeadingZeros;
      }

      // Die jetzige Zeile (Minuend) wird von der gegebene
      // Zeile (Subtrahenden) subtrahiert
      // Die Differenz wird der neue Wert des jetzigen Objekts
      public void SubtractFrom(Row r)
      {
        left.SubtractFrom(r.Left);
        right.SubtractFrom(r.Right);
      }

      // Teilt die gegebene Zeile in positive und negative Zeilen
      // Die negative Zeile wird zu positiv umgewandelt
      // indem *-1 durchgefuehrt wird
      // BSP: Eingabe: {1,0,0,-1,1} => {0,1,1,0,-1}
      //      Ausgabe: [{1,0,0,0,1} => {0,1,1,0,0},
      //                {0,0,0,1,0} => {0,0,0,0,1}]
      // O(n)
      public static Row[] Extract(Row r)
      {
        sbyte[] leftPos, leftNeg, rightPos, rightNeg;
        leftPos = new sbyte[r.Left.Count];
        leftNeg = new sbyte[r.Left.Count];
```

```
      rightPos = new sbyte[r.Left.Count];
      rightNeg = new sbyte[r.Left.Count];
      for (int i = 0; i < r.Left.Count; i++) // O(n)
      {
        if (r.Left[i] > 0)
        {
          leftPos[i] = 1;
          leftNeg[i] = 0;
        }
        else if (r.Left[i] < 0)
        {
          leftPos[i] = 0;
          leftNeg[i] = 1;
        }
        else
        {
          leftPos[i] = 0;
          leftNeg[i] = 0;
        }
      }

      for (int i = 0; i < r.Right.Count; i++)
      {
        if (r.Right[i] > 0)
        {
          rightPos[i] = 1;
          rightNeg[i] = 0;
        }
        else if (r.Right[i] < 0)
        {
          rightPos[i] = 0;
          rightNeg[i] = 1;
        }
        else
        {
          rightPos[i] = 0;
          rightNeg[i] = 0;
        }
      }
      return new Row[] { new Row(new Vector(leftPos),
      new Vector(rightPos)), new Row(new Vector(leftNeg),
      new Vector(rightNeg)) };
    }

    // Fuegt alle Zeilen zu einer Einzelnen
    public static Row Condense(Row[] rows)
    {
      if (rows.Length == 0) throw new ArgumentException(nameof(rows));
      Row rtr = new Row(
        new Vector(Enumerable.Range(0, rows[0].Left.Count).
          Select(_ => (sbyte)0)),
        new Vector(Enumerable.Range(0, rows[0].Left.Count).
          Select(_ => (sbyte)0)));
      for (int i = 0; i < rows.Length; i++)
      {
        rtr.Left.AddFrom(rows[i].Left);
        rtr.Right.AddFrom(rows[i].Right);
      }
```

```
        return rtr;
      }

      public List<int> GetRHSNonZeroEntries()
      {
        List<int> rtr = new List<int>();
        for (int colEntry = 0; colEntry < Count; colEntry++)
        {
          if (Right[colEntry] != 0) rtr.Add(colEntry);
        }
        return rtr;
      }
      ..... // ToString()
    }

    private class Vector : IEnumerable<sbyte>, IEquatable<Vector>
    {
      private readonly sbyte[] data;
      ..... // Die readonly properties und indexer werden hier nicht
            // gezeigt, die sind sehr einfach zu implementieren

      public Vector(IEnumerable<sbyte> data)
        ..... // Nichts besonders
      ..... // Overload von constructor

      public void AddFrom(Vector v)
        ..... // fast gleich wie SubtractFrom daunten

      // O(n)
      public void SubtractFrom(Vector v)
      {
        for (int i = 0; i < data.Length; i++)
          data[i] -= v[i];
      }

      public override int GetHashCode()
      {
        unchecked
        {
          int hash = 21;
          for (int i = 0; i < data.Length; i++)
            hash = hash * 23 + data[i].GetHashCode();
          return hash;
        }
      }
      ..... // Override fuer die geerbten Klassen
    }
  }
}
```