Lösung „Spießgesellen“

Lars Müller, Teilnahme-ID 58886

Problem

Gegeben:

- Apfel, Banane und Brombeere: Schüsseln 1, 4 und 5
- Banane, Pflaume und Weintraube: Schüsseln 3, 5 und 6
- Apfel, Brombeere und Erdbeere: Schüsseln 1, 2 und 4
- Erdbeere und Pflaume: Schüsseln 2 und 6

Gesucht:

- Weintraube, Brombeere und Apfel: Welche Schüsseln?

Formalisierung

Mengen-Gleichungen: Jede Frucht ist eine Schüssel zuzuordnen; eine Menge von Früchten gleicht einer Menge von Schüsseln:

Seien die Variablen für die Schüsseln nach den Anfangsbuchstaben der Früchte [A]pfel, [B]anane, [P]flaume, [W]eintraube, [E]rdbeere benannt, mit Ausnahme der Variablen R für die Nummer der die Brombeeren enthaltenden Schüssel.

Gegeben:

- $\{A,B,R\} = \{1,4,5\}$ (1)
- $\{B,P,W\} = \{3,5,6\}$ (2)
- $\{A,R,E\} = \{1,2,4\}$ (3)
- $\{E,P\} = \{2,6\}$ (4)

Gesucht: Schlüsseln x,y,z, für die gilt $\{W,R,A\} = \{x,y,z\}$.

Lösung

Die Mengen-Gleichungen können über Vereinigung, Schnitt und Differenz so kombiniert werden, dass wir die gewünschte Gleichung erhalten:

1. (3) - (4): $\{A,R,E\} \setminus \{E,P\} = \{1,2,4\} \setminus \{2,6\} \Leftrightarrow \{A,R\} = \{1,4\}$ (5)
2. (2) - (1): $\{B,P,W\} \setminus \{A,B,R\} = \{3,5,6\} \setminus \{1,4,5\} \Leftrightarrow \{P,W\} = \{3,6\}$ (6)
3. (6) - (4): $\{P,W\} \setminus \{E,P\} = \{3,6\} \setminus \{2,6\} \Leftrightarrow \{W\} = \{3\}$ (7)
4. (5) + (7): $\{A,R\} \cup \{W\} = \{1,4\} \cup \{3\} \Leftrightarrow \{A,R,W\} = \{1,3,4\} \Leftrightarrow \{W,R,A\} = \{1,3,4\}$

Donald muss sich also aus den Schalen 1, 3 und 4 bedienen, um Weintraube W, Brombeere R und Apfel A zu erhalten.

Generalisierung

Jedes solche Problem lässt sich analog als Mengen-Gleichungssystem schreiben. Gesucht ist dann eine ableitbare Mengen-Gleichung, wo auf der linken Frucht-Seite die von Donald gewünschten Früchte und auf der rechten Schalen-Seite die diese enthaltenden Schalen stehen.

Alternativen

Manchmal lässt sich keine Menge von Schalen finden, die exakt die gewünschten Früchte enthält. Es lassen sich allerdings Mengen-Gleichungen finden, bei denen auf der Frucht-Seite manche Früchte fehlen oder andere unerwünscht sind. Drei mögliche Kriterien für die Auswahl einer alternativen Frucht-Menge sind:

1. Insgesamt-Abweichung: Das Fehlen einer Frucht ist genau gleich problematisch wie das Vorhandensein unerwünschter Früchte
2. Minimiere die Anzahl der fehlenden Früchte bei den Alternativen. Minimiere dann die Anzahl der unerwünschten Früchte („Lieblingsfrüchte“).
3. Minimiere die Anzahl der unerwünschten Früchte, dann die der fehlenden („wählerisch“).

Verfahren

Eine Art „Mengen-Gleichungs-Löser“ kombiniert über die genannten Mengenoperationen Gleichungen, und stellt weiter sicher, dass das Gleichungssystem währenddessen duplikatfrei bleibt. Dies macht man so lange, bis keine neuen Gleichungen mehr erzeugt werden können.

Nicht nur gemachte Beobachtungen stellen Gleichungen dar. Im gegebenen einfachen Beispiel reichen die gemachten Beobachtungen. In `spiesse1.txt` ist dies allerdings beispielsweise nicht der Fall: Die Grapefruit kommt in Donalds Wünschen vor, wurde von ihm aber noch nicht beobachtet. Dennoch kann er über ein Ausschlussverfahren bestimmen, in welcher Schale die Grapefruit sich befindet. Wir stellen fest, dass wir für solche Fälle die Gleichung „Menge aller Früchte = Menge aller Schalen“ unserem Mengen-Gleichungssystem hinzufügen müssen.

Stufen

Den Kombinationsprozess teilen wir in zwei Stufen auf. Beide Stufen sind im Grunde eine Breadth-First Search über die jeweiligen Mengenoperationen erzeugbaren Mengen - wobei allerdings ein großer, für die Lösung irrelevanter Teil weggelassen wird. Für die Bestimmung einer absolute oberen Grenze für die Anzahl der möglichen (Früchte-)Mengen - und somit auch Gleichungen - lässt sich die Potenzmenge der alle Früchte enthaltenden Mengen heranziehen: Alle Mengengleichungen können nur Teilmengen dieser Menge verwenden. Die Kardinalität der Potenzmenge ist 2^n - exponentiell. Tatsächlich liegen die bestimmten Mengengleichungen allerdings weit unter diesem Wert - eine Vielzahl der Elemente der Potenzmenge ist praktisch aus den gegebenen Beobachtungen einfach nicht ableitbar; andere ableitbare Gleichungen werden wie erwähnt weggelassen, oder wie die Teilmengen der Zielmenge in einer Gleichung

komprimiert. In der ersten Stufe werden nur die reduzierenden Mengenoperationen Differenz und Schnitt verwendet, in der zweiten nur die additive Vereinigung.

Es gilt: Jeder Term bestehend aus Mengen und den drei Mengenoperationen lässt sich als Vereinigung von Termen aus Schnitten und Differenzen darstellen. Dies folgt aus den Mengen-Gesetzen. Exemplarisch:

$(A \cup B) \cap C = A \cup (B \cap C)$

$A \cap (B \cup C) = A \cup (B \cap C)$

$(A \cup B) \setminus C = (A \setminus B) \cup (A \setminus C)$

$A \setminus (B \cup C) = (A \setminus B) \setminus C$

In der ersten „destruktiven“ Stufe behalten wir stets nur die neu entstandenen Gleichungen. Werden für alle Mengen A und B mit $A \neq B$ die Mengen $C = A\ B$, $D = B\ A$ und $E = A \cap B$ gebildet, geht keine Information verloren: $C \cup E = A$ und $D \cup E = B$.

Weiter ziehen wir jeweils Untermengen der Zielmenge „raus“: Entsteht eine Gleichung, deren Früchtemenge eine Untermenge der Zielmenge ist, entfernen wir die Gleichung vom Gleichungssystem. Wir verwalten dafür eine Insgesamt-Gleichung, die eine Vereinigung aller solcher Untermengen darstellt und sich immer weiter der Zielmenge annähert. Diese wird in jedem Schritt der ersten Stufe von allen anderen Gleichungen abgezogen.

Die übrig bleibenden Gleichungen werden in der zweiten Stufe miteinander vereinigt, insofern die Früchtemenge der entstehenden Gleichung „näher“ an der Zielmenge ist, also neue gewünschte Früchte hinzugekommen sind. Dies geschieht so lange, bis keine neuen Vereinigungen mehr gebildet werden können.

Nach den gegebenen Präferenzen werden dann von allen so entstandenen Gleichungen nach einem Vergleich der Früchtemenge mit der Zielmenge nur die „Besten“ ausgewählt.

Umsetzung

Das Programm akzeptiert bis zu 64 Früchte / Schalen und benötigt entsprechend keine Verschiedenen Anfangsbuchstaben.

Kompilierung

Go 1.13 oder neuer benötigt. `go build` (erzeugt eine ausführbare Datei namens `Spießgesellen`) oder `go build main.go` (nennt die Datei `main`). Die vorkompilierte ausführbare Datei ist für Linux.

Verwendung

`go run main.go <pfad> [praeferenzen]` oder `./main <pfad> [praeferenzen]`: Erstes Argument ist der Pfad zur Datei. Das zweite Argument ist optional und gibt Donalds „Präferenzen“ an; Standardwert ist `*`. Möglich sind:

- `*`: Bestimme für jedes der folgenden drei Kriterien alle Alternativen
- `+-`: Bestimme Alternativen nach Kriterium 1
- `+`: Bestimme Alternativen nach Kriterium 2
- `-`: Bestimme Alternativen nach Kriterium 3

Beispiel: `./main beispieldaten/spiesse0.txt`

Ausgabeformat

```
Die Früchte befinden sich in den Schalen: [...]
```

oder

```
Keine exakte Bestimmung der Schalen möglich. Alternativen:

Minimale insgesamte Abweichung ... vom Gewünschten (+-):
=A,B,C,... (erwünschte Früchte); +D,E,F,... (unerwünschte Früchte);
 ↪  -G,H,I,... (fehlende Früchte)  :  [...]

Nur ... fehlende und ... unerwünschte Früchte (+):
...

Nur ... unerwünschte und ... fehlende Früchte (-):
...
```

Je nach Präferenz wird nur ein Teil der Alternativen ausgegeben. Die nach dem Gleichheitszeichen aufgeführten Früchte entsprechen Donalds Wünschen; die hinter dem Pluszeichen sind unerwünscht, und die hinter dem Minuszeichen fehlen Donald.

Bibliotheken

- `bufio`: `Scanner` zum zeilenweisen Einlesen
- `fmt`: Ausgabe & Formattierung
- `os`: Programmargumente
- `strconv`: Zahl-Text-Konvertierung
- `strings`: Auftrennen von Text

`BitSet`

> Es gibt höchstens 26 verschiedene Obstsorten.

Entsprechend enthält eine Menge von Früchten, und genauso eine Menge von Schalen, maximal 26 verschiedene Elemente. Dann kann die Menge aber als „Bit-Set“, als „Menge von Bits“, dargestellt werden: Jedes Bit steht für eine Frucht bzw. Schale; ist es gesetzt, ist die Frucht oder Schale Teil der Menge, ansonsten nicht. 32 Bits, 4 Bytes, reichen für 26 mögliche Elemente; auf einem 64-Bit System kostet es aber fast keine Laufzeit, auf 64 Bits zu erhöhen.

BitSets sind nicht nur besonders speichereffizient, sondern auch effizient zu verarbeiten: Eine Vereinigung ist über bitweises Oder möglich, ein Schnitt über bitweises And; bitweises Xor

stellt die symmetrische Differenz dar, bitweises „A und nicht B“ kann für die Differenz $A \; B$ verwendet werden.

Kardinalitätsbestimmung

Die Zählung eines Bit-Sets ist die einzige Operation, die nicht mit ein oder zwei simplen bitwise-operators erledigt werden kann. Ein naiver Ansatz wäre, alle Bits bis zur Anzahl an Früchten durchzugehen und zu zählen. Effizienter ist es allerdings, das Bit-Set in Bytes, 8-Bit-Sets, zu unterteilen, und ein „lookup“-Array zu verwenden: Für alle 256 möglichen 8-Bit-Sets bestimmt man die gesetzten Bits und speichert dies in einem Array. Index ist dann immer ein Bit-Set als 8-Bit-Unsigned-Integer. Für jede der 8-Bit Teilmengen bestimmt man nun über das Array die Anzahl der Elemente und summiert schließlich.

Gleichungssystem

Das Gleichungssystem wird als „Map“ vom BitSet der Früchte zum BitSet der Schalen dargestellt. So können effizient ($O(logn)$) Duplikate verhindert werden.

Parallelisierbarkeit

Das Verfahren ist schlecht parallelisierbar: Es gibt ein Mengen-Gleichungssystem, auf dem ständig operiert wird; dieses zwischen Threads aufzuteilen wäre mit einem erheblichen Aufwand verbunden. Eine Parallelisierung ist aufgrund der festgestellten Laufzeiten für die Beispiele allerdings nicht unbedingt notwendig.

Ablauf

Zuerst werden die Programmargumente eingelesen und validiert. Dann wird die Datei eingelesen: In der Reihenfolge des Auftauchens wird jeder Frucht inkrementell eine Nummer zugeordnet, sodass sich die Früchtemenge als Menge von Zahlen von 0 bis Anzahl Früchte (letzteres exklusive) darstellen lässt. Bei allen Operationen müssen entsprechend nur die untersten Anzahl Früchte Bits berücksichtigt werden. Dies wird genutzt, um die Zählung kleiner Mengen weiter zu vereinfachen.

Da die nachgestellten Leerzeichen bei den gegebenen Beispielen habe ich zwar entfernt, dennoch enthält das Programm ein „workaround“ (annotiert als „HACK“) um diese zu ignorieren: Leere Strings werden nach dem Auftrennen nach Leerzeichen einfach ignoriert.

Es ist möglich, dass nicht alle Früchte benannt werden. In diesem Fall muss aufgefüllt werden, denn es ist möglich, dass am Ende eine Menge gebildet werden muss, in der die unbenannten Früchte vorkommen, ausgehend von der „Insgesamt-Gleichung“, in der alle Früchte vorkommen. Für eine nützliche Ausgabe auch in diesem Fall werden die unbekannten Früchte einfach durchnummeriert. Siehe `spiesse8.txt`.

Sobald die Zielmenge als `BitSet` und die Gleichungen als `map[BitSet]BitSet` vorliegen, kann das Verfahren beginnen.

Beide Stufen werden jeweils durch eine `while`-Schleife (in Go: `for` ohne Bedingung) realisiert, die Verlassen wird, sobald keine neuen Einträge der Gleichungs-Map hinzugekommen sind. Nach der ersten Stufe werden alle Gleichungen ohne Schnitt mit der Zielmenge von der Gleichungs-Map entfernt.

Gibt es mehrere Gleichungen mit gleichem Schnitt mit der Zielmenge, können und sollten wir jene mit minimal vielen unerwünschten Früchten bestehen lassen und alle anderen entfernen.

Quellcode

main.go

```go
package main

import (
	"bufio"
	"fmt"
	"os"
	"strconv"
	"strings"
)

// BitSet für bis zu 64 Einträge
type BitSet uint64
const SET_CAPACITY uint8 = 64

// Enthält Element
func (set BitSet) Has(index uint8) bool {
	return (set & (1 << index)) > 0
}

// Füge Element hinzu
func (set BitSet) Add(index uint8) BitSet {
	return set | (1 << index)
}

// Füge Schale hinzu: 1 abziehen
func (set BitSet) AddSchale(schale uint8) BitSet {
	return set.Add(schale - 1)
}

// Schalen-Liste für Ausgabe erstellen: Zu jedem Element wieder 1 hinzuaddieren
func (set BitSet) ToSchalen() []uint8 {
	schalen := make([]uint8, set.Count())
	cursor := 0
	for i := uint8(0); i < frucht_nummer; i++ {
```

```
35                  if set.Has(i) {
36                          schalen[cursor] = i + 1
37                          cursor++
38                  }
39          }
40          return schalen
41  }
42
43  // Symmetrische Differenz (xor)
44  func (set BitSet) Difference(other BitSet) BitSet {
45          return set ^ other
46  }
47
48  // Differenz / Abzug (and not)
49  func (set BitSet) Except(other BitSet) BitSet {
50          return set &^ other
51  }
52
53  // Schnitt (and)
54  func (set BitSet) Intersect(other BitSet) BitSet {
55          return set & other
56  }
57
58  // Vereinigung (or)
59  func (set BitSet) Union(other BitSet) BitSet {
60          return set | other
61  }
62
63  // Byte-Set counts lookup array
64  var counts [256]uint8
65
66  func (set BitSet) Count() uint8 {
67          count := uint8(0)
68          for shift := uint8(0); shift < frucht_nummer; shift += 8 {
69                  count += counts[(set >> shift) & 0xFF]
70          }
71          return count
72  }
73
74  // Ist set eine Obermenge von other?
75  func (set BitSet) Super(other BitSet) bool {
76          return set.Union(other) == set
77  }
78
79  // Ist set eine Untermenge von other?
```

```go
func (set BitSet) Sub(other BitSet) bool {
	return other.Super(set)
}

// Frucht-Operationen
// Frucht-Namen: [index] = name
var frucht_name []string
// Frucht-Nummer: Wird zwischenzeitlich (während dem Einlesen) als Cursor verwendet
// Ist danach = die Anzahl der Früchte
var frucht_nummer uint8 = 0
// Frucht-Nummern: [name] = index
var frucht_nummern map[string]uint8 = map[string]uint8{}

// Frucht hinzufügen
func (set BitSet) AddFrucht(frucht string) BitSet {
	index, found := frucht_nummern[frucht]
	// Hat die Frucht noch keine Nummer?
	if !found {
		// Nummer zuweisen
		frucht_name[frucht_nummer] = frucht
		frucht_nummern[frucht] = frucht_nummer
		index = frucht_nummer
		// Nächste Nummer für nächste Frucht
		frucht_nummer++
	}
	return set.Add(index)
}

// Frucht-Liste -> Frucht-Menge
func FromFruechte(fruechte []string) BitSet {
	var set BitSet
	for _, frucht := range fruechte {
		set = set.AddFrucht(frucht)
	}
	return set
}

// Frucht-Menge -> Frucht-Liste für Ausgabe
func (set BitSet) ToFruechte() []string {
	res := make([]string, set.Count())
	c := 0
	for i := uint8(0); i < frucht_nummer; i++ {
		if set.Has(i) {
			res[c] = frucht_name[i]
```

```
124                         c++
125                 }
126         }
127         return res
128 }
129
130 func main() {
131         // Argumente lesen
132         if len(os.Args) < 2 || len(os.Args) > 3 {
133                 println("Verwendung: <pfad> [praeferenzen]")
134                 return
135         }
136         praeferenzen := "*"
137         if len(os.Args) == 3 {
138                 praeferenzen = os.Args[2]
139                 if praeferenzen != "*" && praeferenzen != "+-" &&
                     ↪  praeferenzen != "+" && praeferenzen != "-" {
140                         println("Verwendung: <pfad> [praeferenzen]")
141                         return
142                 }
143         }
144         // Datei öffnen
145         file, err := os.Open(os.Args[1])
146         if err != nil {
147                 panic(err)
148         }
149         defer file.Close()
150         scanner := bufio.NewScanner(file)
151         scanner.Scan()
152         anzahl_fruechte, _ := strconv.Atoi(scanner.Text())
153         if anzahl_fruechte > int(SET_CAPACITY) {
154                 panic("Maximal " + strconv.Itoa(int(SET_CAPACITY)) + "
                     ↪  Früchte erlaubt")
155         }
156         frucht_name = make([]string, anzahl_fruechte)
157         scanner.Scan()
158         // Zielmenge der gewünschten Früchte erstellen
159         var ziel_fruechte BitSet
160         for _, frucht := range strings.Split(scanner.Text(), " ") {
161                 if frucht != "" {
162                         ziel_fruechte = ziel_fruechte.AddFrucht(frucht)
163                 }
164         }
165         scanner.Scan()
166         anzahl_beobachtungen, _ := strconv.Atoi(scanner.Text())
```

```
167         gleichungen := map[BitSet]BitSet{}
168         for i := uint8(0); i < uint8(anzahl_beobachtungen); i++ {
169                 // Schalen-Menge einlesen
170                 scanner.Scan()
171                 var schalen BitSet
172                 for _, schale := range strings.Split(scanner.Text(), " ") {
173                         if schale == "" {
174                                 // HACK wegen trailing spaces
175                                 continue
176                         }
177                         _schale, err := strconv.Atoi(schale)
178                         if err != nil {
179                                 panic(err)
180                         }
181                         schalen = schalen.AddSchale(uint8(_schale))
182                 }
183                 // Früchte-Menge einlesen
184                 scanner.Scan()
185                 var fruechte BitSet
186                 for _, frucht := range strings.Split(scanner.Text(), " ") {
187                         if frucht == "" {
188                                 // HACK wegen trailing spaces
189                                 continue
190                         }
191                         fruechte = fruechte.AddFrucht(frucht)
192                 }
193                 // Mengen-Gleichung in Map speichern
194                 gleichungen[fruechte] = schalen
195         }
196         if err := scanner.Err(); err != nil {
197                 panic(err)
198         }
199
200         // Initialisiere lookup-Array für effizientes Zählen der gesetzten Bits
201         // Erst danach darf BitSet.Count() verwendet werden!
202         for i := 0; i < 256; i++ {
203                 for j := 0; j < 8; j++ {
204                         counts[i] += uint8(1 & (i >> j))
205                 }
206         }
207
208         // Auffüllen: Alle unbenannten, unbekannten Früchte erhalten einen
          ↪  Namen zugewiesen
209         // Diese kommen ausschließlich in der "Insgesamt"-Gleichung vor
210         for i := 0; frucht_nummer < uint8(anzahl_fruechte); frucht_nummer++ {
```

```
211                 i++
212                 name := "Unbekannt_" + strconv.Itoa(i)
213                 frucht_name[frucht_nummer] = name
214         }
215
216         // "Insgesamt"-Gleichung: Menge aller Früchte = Menge aller Schalen
217         // Setze unterste anzahl_fruechte Bits auf 1
218         insgesamt := ^(^BitSet(0) << BitSet(anzahl_fruechte))
219         gleichungen[insgesamt] = insgesamt
220
221         // Teilmengen der Zielmenge
222         var teil_fruechte, teil_schalen BitSet
223         for {
224                 neue_gleichungen_erzeugt := false
225                 neue_gleichungen := map[BitSet]BitSet{}
226                 neueGleichung := func(fruechte, schalen BitSet) {
227                         // Neue Gleichung: Schon bestimmte Teilmengen abziehen
228                         neue_fruechte := fruechte.Except(teil_fruechte)
229                         // Ermitteln, ob eine neue Gleichung erzeugt wurde
230                         _, alt := gleichungen[neue_fruechte]
231                         neue_gleichungen_erzeugt = neue_gleichungen_erzeugt
    ↪ || !alt
232                         // Gleichung setzen
233                         neue_gleichungen[neue_fruechte] =
    ↪ schalen.Except(teil_schalen)
234                 }
235                 for fruechte, schalen := range gleichungen {
236                         if (fruechte.Sub(ziel_fruechte)) {
237                                 // Untermenge der Zielmenge: Zu
                                    ↪ insgesamt-Teilmengen hinzufügen
238                                 teil_fruechte =
    ↪ teil_fruechte.Union(fruechte)
239                                 teil_schalen = teil_schalen.Union(schalen)
240                                 continue
241                         }
242                 }
243                 for fruechte, schalen := range gleichungen {
244                         for fruechte_2, schalen_2 := range gleichungen {
245                                 // Differenz zweier Mengen
246                                 neueGleichung(fruechte.Except(fruechte_2),
    ↪ schalen.Except(schalen_2))
247                                 if fruechte_2 > fruechte {
248                                         // Schnitt zweier Mengen; Kommutativ,
                                            ↪ daher die if-Bedingung
```

```
249                                         neueGleichung(fruechte.Intersect(fruechte_2),
    ↪ schalen.Intersect(schalen_2))
250                                 }
251                         }
252                 }
253
254                 gleichungen = neue_gleichungen
255                 if !neue_gleichungen_erzeugt {
256                         // Aufhören, wenn keine neue Gleichungen mehr erzeugt
                             ↪ werden können
257                         break
258                 }
259         }
260
261         if teil_fruechte == ziel_fruechte {
262                 fmt.Println("Die Früchte befinden sich in den Schalen:",
    ↪ teil_schalen.ToSchalen())
263                 return
264         }
265
266         neue_gleichungen := map[BitSet]BitSet{}
267         nuetzlicheGleichungen: for fruechte, schalen := range gleichungen {
268                 intersection :=
    ↪ fruechte.Intersect(ziel_fruechte).Except(teil_fruechte)
269                 if intersection == 0 {
270                         // Kein Schnitt mit Zielmenge oder alle Schalen schon
                             ↪ bekannt
271                         // Gleichung nutzlos
272                         continue
273                 }
274                 for fruechte_2 := range gleichungen {
275                         intersection_2 :=
    ↪ fruechte_2.Intersect(ziel_fruechte).Except(teil_fruechte)
276                         if intersection == intersection_2 &&
                             ↪ fruechte_2.Count() < fruechte.Count() {
277                                 // Gleicher Schnitt, aber Früchte 2 besitzt
                                     ↪ weniger unerwünschte Früchte
278                                 continue nuetzlicheGleichungen
279                         }
280                 }
281                 neue_gleichungen[fruechte] = schalen
282         }
283
284         neue_gleichungen[teil_fruechte] = teil_schalen
285
```

```
286         // 2. Stufe
287         gleichungen = neue_gleichungen
288         for {
289                 neue_gleichungen_erzeugt := false
290                 neue_gleichungen := map[BitSet]BitSet{}
291                 for fruechte, schalen := range gleichungen {
292                         for fruechte_2, schalen_2 := range gleichungen {
293                                 fruechte_vereinigung :=
    ↪ fruechte.Union(fruechte_2)
294                                 abweichung :=
    ↪ ziel_fruechte.Except(fruechte_vereinigung).Count()
295                                 // Schnitt ist kommutativ.
296                                 if fruechte_2 < fruechte {
297                                         continue
298                                 }
299                                 // Außerdem soll sich die vereinigte Menge
                                     ↪  nicht von der Zielmenge "entfernen".
300                                 if abweichung >
                                     ↪  ziel_fruechte.Except(fruechte).Count()
                                     ↪  || abweichung >
                                     ↪  ziel_fruechte.Except(fruechte_2).Count()
                                     ↪  {
301                                         continue
302                                 }
303                                 schalen_vereinigung :=
    ↪ schalen.Union(schalen_2)
304                                 neue_gleichungen[fruechte_vereinigung] =
    ↪ schalen_vereinigung
305                         }
306                 }
307                 for fruechte, schalen := range neue_gleichungen {
308                         _, alt := gleichungen[fruechte]
309                         if !alt {
310                                 neue_gleichungen_erzeugt = true
311                                 gleichungen[fruechte] = schalen
312                         }
313                 }
314                 // Aufhören, wenn keine neuen Gleichungen mehr erzeugt wurden
315                 if !neue_gleichungen_erzeugt {
316                         break
317                 }
318         }
319
320         minAbweichungInsgesamt := func() (uint8, map[BitSet]BitSet) {
321                 min_abweichung := uint8(255)
```

```
322                 beste_gleichungen := map[BitSet]BitSet{}
323                 // Iteriert alle Gleichungen und bestimmt die mit der
                     ↪  geringsten Abweichung
324                 for fruechte, schalen := range gleichungen {
325                         // Abweichung: Anzahl Elemente der symmetrischen
                             ↪  Differenz
326                         abweichung :=
    ↪  fruechte.Difference(ziel_fruechte).Count()
327                         if abweichung < min_abweichung {
328                                 min_abweichung = abweichung
329                                 beste_gleichungen = map[BitSet]BitSet{}
330                         }
331                         if abweichung <= min_abweichung {
332                                 beste_gleichungen[fruechte] = schalen
333                         }
334                 }
335                 return min_abweichung, beste_gleichungen
336         }
337         // Minimiert Anzahl fehlender oder unerwünschter Früchte je nach
             ↪  Argument
338         minAbweichung := func(minimiere_fehlend bool) (uint8, uint8,
    ↪  map[BitSet]BitSet) {
339                 min_1 := uint8(255)
340                 min_2 := uint8(255)
341                 beste_gleichungen := map[BitSet]BitSet{}
342                 // Gleichungen durchgehen
343                 for fruechte, schalen := range gleichungen {
344                         abweichung_1 :=
    ↪  fruechte.Except(ziel_fruechte).Count()
345                         abweichung_2 :=
    ↪  ziel_fruechte.Except(fruechte).Count()
346                         if minimiere_fehlend {
347                                 // Fehlende minimieren: Prioritäten tauschen!
348                                 abweichung_2, abweichung_1 = abweichung_1,
    ↪  abweichung_2
349                         }
350                         if abweichung_1 > min_2 {
351                                 continue
352                         }
353                         if abweichung_1 < min_2 {
354                                 min_2 = abweichung_1
355                                 min_1 = abweichung_2
356                                 beste_gleichungen = map[BitSet]BitSet{}
357                         } else if abweichung_2 > min_1 {
358                                 continue
```

```
359                         } else if abweichung_2 < min_1 {
360                                 min_1 = abweichung_2
361                                 beste_gleichungen = map[BitSet]BitSet{}
362                         }
363                         beste_gleichungen[fruechte] = schalen
364                 }
365                 return min_1, min_2, beste_gleichungen
366         }
367         // Gibt eine alternative Gleichung aus
368         ausgabeAlternative := func(gleichung map[BitSet]BitSet) {
369                 for fruechte, schalen := range gleichung {
370                         erwuenscht := fruechte.Intersect(ziel_fruechte)
371                         unerwuenscht := fruechte.Except(erwuenscht)
372                         fehlend := ziel_fruechte.Except(erwuenscht)
373                         teile := []string{}
374                         // Formattiert einen Teil der Frucht-Menge
375                         teil := func(prefix string, set BitSet) {
376                                 fruechte := set.ToFruechte()
377                                 if len(fruechte) > 0 {
378                                         teile = append(teile, prefix +
    ↪ strings.Join(fruechte, ","))
379                                 }
380                         }
381                         // Erwünschter Teil
382                         teil("=", erwuenscht)
383                         // Unerwünschter Teil
384                         teil("+", unerwuenscht)
385                         // Fehlender Teil
386                         teil("-", fehlend)
387                         fmt.Println(strings.Join(teile, "; "), " : ",
    ↪ schalen.ToSchalen())
388                 }
389         }
390         ausgabeAlternativen := func(gesucht string) {
391                 var min_insgesamt, min_fehlend, min_unerwuenscht uint8
392                 var gleichungen map[BitSet]BitSet
393                 // Ausgabe je nach gesuchten Alternativen
394                 if gesucht == "+-" {
395                         // Minimale insgesamte Abweichung
396                         min_insgesamt, gleichungen = minAbweichungInsgesamt()
397                         fmt.Println("Minimale insgesamte Abweichung",
    ↪ min_insgesamt, "vom Gewünschten (+-):")
398                 } else if gesucht == "+" {
399                         // Minimal viele unerwünschte Früchte
```

```
400                         min_fehlend, min_unerwuenscht, gleichungen =
    ↪ minAbweichung(false)
401                         fmt.Println("Nur", min_fehlend, "fehlende und",
    ↪ min_unerwuenscht, "unerwünschte Früchte (+):")
402                 } else if gesucht == "-" {
403                         // Minimal viele fehlende Früchte
404                         min_unerwuenscht, min_fehlend, gleichungen =
    ↪ minAbweichung(true)
405                         fmt.Println("Nur", min_unerwuenscht, "unerwünschte
    ↪ und", min_fehlend, "fehlende Früchte (-):")
406                 }
407                 ausgabeAlternative(gleichungen)
408         }
409         if praeferenzen == "*" {
410                 // Alle nach einem der drei Kriterien optimalen Alternativen
411                 fmt.Println("Keine exakte Bestimmung der Schalen möglich.
    ↪ Alternativen:")
412                 fmt.Println()
413                 ausgabeAlternativen("+-")
414                 fmt.Println()
415                 ausgabeAlternativen("+")
416                 fmt.Println()
417                 ausgabeAlternativen("-")
418         } else {
419                 ausgabeAlternativen(praeferenzen)
420         }
421 }
```

Beispiele

Die verstrichene Zeit wurde nicht vom Programm selber, sondern von einem Shellscript gemessen.

spiesse0.txt

Zusätzliches Beispiel (der Aufgabenstellung entnommen):

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

```
2 6
Erdbeere Pflaume

Die Früchte befinden sich in den Schalen: [1 3 4]

Zeit verstrichen: 0m0,002s
```

Dieses Beispiel dient als „sanity-check“: Es kann leicht geprüft werden, ob das Ergebnis der per Hand ermittelten Menge der Schalen $1,3,4$ gleicht.

spiesse1.txt

```
Die Früchte befinden sich in den Schalen: [1 2 4 5 7]

Zeit verstrichen: 0m0,003s
```

spiesse2.txt

```
Die Früchte befinden sich in den Schalen: [1 5 6 7 10 11]

Zeit verstrichen: 0m0,003s
```

spiesse3.txt

```
Keine exakte Bestimmung der Schalen möglich. Alternativen:

Minimale insgesamte Abweichung 1 vom Gewünschten (+-):
=Clementine,Erdbeere,Feige,Himbeere,Ingwer,Kiwi; -Litschi  :  [1 5 7 8 10 12]
=Clementine,Erdbeere,Feige,Himbeere,Ingwer,Kiwi,Litschi; +Grapefruit  :  [1 2
 ↪  5 7 8 10 11 12]

Nur 1 fehlende und 0 unerwünschte Früchte (+):
=Clementine,Erdbeere,Feige,Himbeere,Ingwer,Kiwi; -Litschi  :  [1 5 7 8 10 12]

Nur 1 unerwünschte und 0 fehlende Früchte (-):
=Clementine,Erdbeere,Feige,Himbeere,Ingwer,Kiwi,Litschi; +Grapefruit  :  [1 2
 ↪  5 7 8 10 11 12]

Zeit verstrichen: 0m0,007s
```

spiesse4.txt

```
Die Früchte befinden sich in den Schalen: [2 6 7 8 9 12 13 14]

Zeit verstrichen: 0m0,039s
```

spiesse5.txt

```
Die Früchte befinden sich in den Schalen: [1 2 3 4 5 6 9 10 12 14 16 19 20]

Zeit verstrichen: 0m0,004s
```

spiesse6.txt

```
Die Früchte befinden sich in den Schalen: [4 6 7 10 11 15 18 20]

Zeit verstrichen: 0m3,191s
```

spiesse7.txt

Zusätzliches Beispiel (`spiesse6.txt` ohne die letzte Beobachtung):

```
23
Clementine Erdbeere Himbeere Orange Quitte Rosine Ugli Vogelbeere
7
13 18 3 22 14 2 12 15 23 21 20 11 8
Apfel Banane Feige Himbeere Ingwer Johannisbeere Kiwi Litschi Nektarine
 ↪  Orange Rosine Ugli Weintraube
11 20 18 2 9 15 4 19 12 5
Dattel Himbeere Johannisbeere Nektarine Orange Pflaume Quitte Rosine
 ↪  Tamarinde Ugli
4 8 1 11 23 17 21 9 13 7 15 2 3 16
Apfel Banane Clementine Grapefruit Kiwi Litschi Mango Nektarine Pflaume
 ↪  Quitte Rosine Sauerkirsche Ugli Weintraube
1 5 9 17 19 8 3 7 22
Apfel Banane Clementine Dattel Feige Grapefruit Mango Pflaume Tamarinde
17 14 4 13 12 1 18 5 21 10 3
Apfel Dattel Erdbeere Grapefruit Himbeere Ingwer Johannisbeere Litschi Mango
 ↪  Quitte Weintraube
2 5 20 4 18 17 22 12 7 9 10 14 23 1
Clementine Dattel Erdbeere Feige Grapefruit Himbeere Ingwer Johannisbeere
 ↪  Kiwi Mango Nektarine Orange Pflaume Quitte
23 21 10 3 17 5 9 2 8 19 13 20 22
Apfel Banane Dattel Erdbeere Feige Grapefruit Kiwi Litschi Nektarine Orange
 ↪  Pflaume Tamarinde Weintraube

Keine exakte Bestimmung der Schalen möglich. Alternativen:

Minimale insgesamte Abweichung 1 vom Gewünschten (+-):
=Clementine,Erdbeere,Himbeere,Orange,Quitte,Rosine,Ugli,Vogelbeere;
 ↪  +Johannisbeere  :  [4 6 7 10 11 12 15 18 20]
=Clementine,Erdbeere,Orange,Quitte,Rosine,Ugli,Vogelbeere; -Himbeere  :  [4 6
 ↪  7 10 11 15 20]
```

```
Nur 1 fehlende und 0 unerwünschte Früchte (+):
=Clementine,Erdbeere,Orange,Quitte,Rosine,Ugli,Vogelbeere; -Himbeere  :  [4 6
 ↪  7 10 11 15 20]

Nur 1 unerwünschte und 0 fehlende Früchte (-):
=Clementine,Erdbeere,Himbeere,Orange,Quitte,Rosine,Ugli,Vogelbeere;
 ↪  +Johannisbeere  :  [4 6 7 10 11 12 15 18 20]

Zeit verstrichen: 0m1,170s
```

Das so entstandene Beispiel ist eine komplexere Variante von `spiesse3.txt`, um sicherzustellen, dass auch im Fall einer Bestimmung von Alternativen mit mehr Früchten als in `spiesse3.txt` die Laufzeit kurz bleibt.

spiesse8.txt

Zusätzliches Beispiel (`spiesse2.txt` mit Anzahl Früchte 11) - demonstriert die Notwendigkeit des Auffüllens. Als kleiner „stress test“ wird die Fruchtzahl außerdem auf 42 gesetzt:

```
42
Clementine Erdbeere Grapefruit Himbeere Johannisbeere
4
6 10 3
Banane Feige Ingwer
2 1 9 8 4
Apfel Clementine Dattel Erdbeere Himbeere
5 8 10 4 2
Apfel Erdbeere Feige Himbeere Johannisbeere
6 4 2 5 9
Dattel Erdbeere Himbeere Ingwer Johannisbeere

Keine exakte Bestimmung der Schalen möglich. Alternativen:

Minimale insgesamte Abweichung 1 vom Gewünschten (+-):
=Clementine,Erdbeere,Himbeere,Johannisbeere; -Grapefruit  :  [1 2 4 5]

Nur 1 fehlende und 0 unerwünschte Früchte (+):
=Clementine,Erdbeere,Himbeere,Johannisbeere; -Grapefruit  :  [1 2 4 5]

Nur 32 unerwünschte und 0 fehlende Früchte (-):
=Clementine,Erdbeere,Grapefruit,Himbeere,Johannisbeere;
 ↪  +Unbekannt_1,Unbekannt_2,Unbekannt_3,Unbekannt_4,Unbekannt_5,Unbekannt_6,Unbekannt_7,
 ↪  :  [1 2 4 5 7 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 ↪  31 32 33 34 35 36 37 38 39 40 41 42]
```

```
Zeit verstrichen: 0m0,003s
```

Für alle gegebenen Beispiele ist das Programm ausreichend schnell: In meiner Umgebung sind alle Laufzeiten kürzer als 10 Sekunden.