34. Bundeswettbewerb Informatik – Aufgabe 3

Team „ByteSector“

Lösungsidee

Für die Beispiele *flaschenzug0* bis *flaschenzug2* lässt sich die Anzahl an möglichen Verteilungen mithilfe einer simplen Implementation der Brute-Force-Methode berechnen. Für die weiteren Beispiele steigt die Berechnungszeit aufgrund der hohen Komplexität allerdings zu hoch, um vor weltuntergangsbedingtem Versagen der verwendeten Rechenmaschine das Ergebnis in Erfahrung zu bringen, weshalb die Eigenschaften des Problems genauer untersucht und Optimierungsmöglichkeiten gefunden werden müssen.

Der grundlegende Ablauf des Algorithmus besteht darin, die Behälter einzeln abzuarbeiten und jeweils festzuhalten, wie viele Möglichkeiten es gibt, die noch unverteilten Flaschen zwischen dem aktuellen Behälter und allen Folgebehältern aufzuteilen. Im Beispiel von *flaschenzug2* würde dies im ersten Schritt bedeuten, zu berechnen, wie viele Möglichkeiten es gibt, die zehn Flaschen zwischen dem ersten Zehner Behälter und den restlichen Behältern, sprich dem Achter- und Fünfer-Behälter aufzuteilen. In diesem Falle wäre es insgesamt elf Möglichkeiten, da alle zehn bis einschließlich null Flaschen in den Zehner-Behälter gestellt werden können. Jede der elf Möglichkeiten führt ggf. zu vielen weiteren Möglichkeiten, beim nächsten Behälter weiterzumachen. Verteilt man alle Flaschen auf den Zehner-Behälter, gibt es lediglich die Möglichkeit, alle anderen Behälter leer zu lassen, verteilt man allerdings alle Flaschen auf die Folgebehälter und lässt den Zehner-Behälter leer, gibt es vier Möglichkeiten, von dort aus weiterzumachen.

Die Anzahl an möglichen Verteilungen zwischen aktuellem Behälter und allen Folgebehältern ist durch vier Faktoren begrenzt:

- Es können nicht mehr Flaschen auf den aktuellen Behälter verteilt werden, wie...
  - Flaschen vorhanden sind.
  - Behälterkapazität vorhanden ist
- Es können nicht weniger Flaschen auf den aktuellen Behälter verteilt werden...
  - als 0.
  - als nötig, um die übrig bleibenden Flaschen noch auf alle Folgebehälter verteilen zu können. Die Summe der Kapazitäten aller Folgebehälter, von nun an „Folgekapazität“ genannt, muss im niedrigsten Fall gleich der Anzahl an übrig bleibenden Flaschen sein.

Statt nun alle Möglichkeiten zu überprüfen, lassen sich also folgende Formeln für die maximale und minimale Anzahl an auf den aktuellen Behälter verteilten Flaschen aufstellen:

Sei F_{max} das Maximum, F_{min} das Minimum, F_{all} die Anzahl an verfügbaren Flaschen, F_{sel} die Anzahl an auf den aktuellen Behälter verteilten Flaschen, $F_{all}-F_{sel}=F_{rem}$ die Anzahl an übrig bleibenden Flaschen, C_{cur} die Kapazität des aktuellen Behälters und C_{rem} die Folgekapazität,

gilt für das Maximum $F_{max}=F_{all} \vee F_{max}=C_{cur}$ **(je nachdem, welches niedriger ist).**

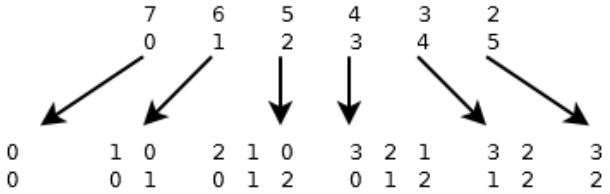
Für das Minimum gilt $F_{min}=0 \vee F_{min}=F_{all}-C_{rem}$ **(je nachdem, welches höher ist).**

Letztere Formel leitet sich aus $F_{rem}=C_{rem}$ her:

$$F_{rem} = C_{rem} \quad | \quad F_{all} - F_{sel} = F_{rem}$$
$$F_{all} - F_{sel} = C_{rem} \quad | \quad F_{sel} = F_{min}$$
$$F_{all} - F_{min} = C_{rem} \quad | \quad -C_{rem} \quad | \quad +F_{min}$$
$$F_{min} = F_{all} - C_{rem}$$

Mit dieser Optimierung wird nun die Bestimmung aller gültigen Verteilungen beim Bearbeiten eines einzelnen Behälters verkürzt. Es wird allerdings nicht verhindert, dass bei größeren Anzahlen an Behältern und Behälterkapazitäten extrem viele Verzweigungen während der Bearbeitung aller Behälter entstehen. Um die Berechnungszeit dieser Verzweigungen ausschlaggebend zu reduzieren, muss eine weitere Eigenschaft des Problems beachtet werden:

Erreicht der Algorithmus im Laufe der Bearbeitung von *flaschenzug3* die Verteilungen 10; 5 und 13; 2 für den 20er- und den 19er-Behälter, versucht er nun in beiden Fällen zu berechnen, wie viele Möglichkeiten es gibt, die restlichen 15 Flaschen auf die selben restlichen 18 Behälter zu verteilen. Das Ergebnis wird offensichtlich das gleiche sein, die Rechnung aber unnötigerweise zweimal stattfinden. Um diese verschwendete Rechenarbeit zu sparen, wird eine Tabelle geführt, in der zu jeder bereits vollständig durchgerechneten Kombination an F_{all} , C_{cur} und C_{rem} die dazugehörige Anzahl an gültigen Verteilungen notiert wird. In diesem Falle würde zur Kombination 15; 18; 306 die Anzahl 374.153.699 gespeichert werden. Das Größenverhältnis der Möglichkeiten, die nun nur noch ein einziges Mal berechnet werden müssen, zeigt auf, wie viel Arbeit dadurch gespart wird. Dass diese Optimierung sehr häufig Wirkung zeigt, kann man auch in folgendem Diagramm ablesen:



Es sollen sieben Flaschen auf vier Behälter mit den Kapazitäten sieben, drei, eins und eins verteilt werden. Die obere Zahl beschreibt jeweils F_{sel} , die untere F_{rem} . Es ist erkennbar, dass alle zwölf Pfade in der Berechnung der möglichen Verteilung von entweder zwei, einer oder null Flaschen auf die restlichen zwei Einser-Behälter enden. Mit dieser Optimierung werden also nur drei Pfade vollständig berechnet und die neun restlichen werden mithilfe der Tabelle in jeweils einem Schritt gelöst.

Umsetzung

Die Umsetzung erfolgt als ein in Java geschriebenes Programm. Exportiert wird diese Version als Jar-Datei gepaart mit einer Batch-Datei zur Ausführung. Getestet wurde das Programm unter Windows 7 Home Premium SP1 64-Bit mit JDK 1.8.0_65 64-Bit. Die Parameter für die Berechnung werden im Format der Beispielprobleme als Textdatei übergeben, indem ein Pfad zu dieser durchs Drag&Drop'en der Textdatei auf die Batch-Datei als Startparameter über die Batch-Datei an das Programm weitergeleitet wird.

Im Algorithmus wurde als Datentyp zur Speicherung der Anzahl an Kombinationen *BigInteger* verwendet, da mit Zahlen bis über 2^{131} gerechnet wird, was weit über die Grenzen eines *unsigned long*, sprich $[0;2^{64}-1]$ hinausgeht.

Als in der Lösungsidee genannte Tabelle wird eine HashMap verwendet, dessen Keys Strings sind, welche durch Konkatenation der einzelnen Eigenschaften der Kombination und jeweils einem Unterstrich zwischen den Werten zwecks Trennung erzeugt werden:

```
// HashMap zum Speichern der Kombinationen (fAll, cCur, cRem) und ihrer Möglichkeiten
poss = new HashMap<String, BigInteger>();
[...]
private String intComb2String(int fAll, int cCur, int cRem){
	return fAll + "_" + cCur + "_" + cRem;
```

```
}
```

Der Algorithmus selbst: (Der erste Aufruf geschieht im Konstruktor der Main-Klasse mit den Argumenten *Anzahl an Flaschen insgesamt*, *Index des letzten Behälters in der Eingabe*, *Folgekapazität aus Sicht dieses Behälters)*

```
private BigInteger calcPoss(int fAll, int iBeh, int cRem){
	// Wenn dies der letzte Behälter ist, gibt es nur die Möglichkeit, alle verfügbaren
	// Flaschen in den aktuellen Behälter zu stellen.
	if (iBeh == 0)
		return new BigInteger("1");

	// Wenn Anzahl an Möglichkeiten für gegebene Kombination an verfügbaren Flaschen,
	// aktueller Kapazität und Folgekapazität bereits bekannt, Berechnung überspringen
	// und stattdessen direkt den gespeicherten Wert ausgeben.
	BigInteger found = poss.get(intComb2String(fAll, beh[iBeh], cRem));
	if (found != null){
		return found;
	}

	// Alle gültigen Verteilungen durchgehen
	// Rückgabewert erstellen
	BigInteger ret = new BigInteger("0");

	// Minimum und Maximum an fSel berechnen
	int fMax = Math.min(fAll, beh[iBeh]),
		fMin = Math.max(fAll - cRem, 0);

	// Alle gültigen Verteilungen an Flaschen für den aktuellen Behälter durchgehen
	for (int fSel = fMin; fSel <= fMax; fSel++){
		/* Nächst-unterer Schritt wird mit den übrig bleibenden Flaschen,
		 * der Kapazität des nächsten Behälters und der Folgekapazität
		 * ausgehend vom nächsten Behälter durchgeführt.
		 * Dadurch, dass immer direkt der erste Schritt auf die Berechnung des nächsten
		 * Behälters weiterleitet, findet eine Tiefensuche statt, was dafür sorgt,
		 * das die Optimierungsverfahren oft Anwendung finden. */
		ret = ret.add(calcPoss(fAll - fSel, iBeh - 1, cRem - beh[iBeh - 1]));
	}

	// Nach der Berechnung die aktuelle Kombination und Anzahl an Möglichkeiten abspeichern,
	// damit die Berechnung in Zukunft übersprungen werden kann.
	poss.put(intComb2String(fAll, beh[iBeh], cRem), ret);
	return ret;
}
```

BEISPIELE

Beispielproblem	Anzahl an möglichen Verteilungen	… in gerundeter wissenschaftlicher Notation
flaschenzug0	2	2.00 * 10 ^0
flaschenzug1	13	1.30 * 10^1
flaschenzug2	48	4.80 * 10^1
flaschenzug3	6.209.623.185.136	6.21 * 10^12
flaschenzug4	743.587.168.174.197.919.278.525	7.44 * 10^23
flaschenzug5	4.237.618.332.168.130.643.734.395.335.220.863.408.628	4.24 * 10^39

Alle Probleme wurde mithilfe dieses Algorithmus in jeweils unter zwei Sekunden gelöst.