Flaschenzug

Aufgabe 3, Runde 1, 34. Bundeswettbewerb Informatik

Marian Dietz, Johannes Heinrich, Erik Fritzsche

1 Lösungsidee

Bei einer Aufgabe steht n für die Anzahl der Flaschen und m für die Anzahl der Behälter. Jeder Behälter hat einen eindeutigen Index. Es werden die Indizes 0 bis $m-1$ verwendet. b_i gibt an, wie viele Flaschen in den Behälter mit dem Index i hineinpassen. $f(i, j)$ gibt an, auf wie viele Arten j Flaschen auf die Behälter mit den Indizes 0 bis i verteilt werden können. Somit ist $f(m-1, n)$ die Lösung für die Aufgabe.

Eine Aufgabe kann als Tabelle mit $n+1$ Zeilen und m Spalten dargestellt werden (wobei Kopfspalten und -zeilen natürlich nicht dazugezählt werden). Jede Zeile steht für eine Anzahl von Flaschen, jede Spalte steht für einen Behälter. Eine Zelle ist der Funktionswert $f(i, j)$, dabei ist i der Index der Spalte und j der Index der Zeile, wobei die Indizes bei 0 starten. Somit gibt eine Zelle an, auf wie viele Arten die j Flaschen (siehe Tabellenkopf links der Zelle) auf alle Behälter links dieser Zelle inklusive des Behälters der Spalte selber verteilt werden können.

Ein Beispiel ist in Tabelle 1 zu sehen, bei dem 5 Flaschen auf 3 Behälter verteilt werden, in die 2, 4 und 4 Flaschen passen.

Tabelle 1 Beispielaufgabe 1

	Behälter 0 (2)	**Behälter 1 (4)**	**Behälter 2 (4)**
0 Flaschen	1	1	1
1 Flasche	1	2	3
2 Flaschen	1	3	6
3 Flaschen	0	3	9
4 Flaschen	0	3	12
5 Flaschen	0	2	13

In diesem Beispiel zeigt die grau hinterlegte Zelle an, dass die 4 Flaschen auf die Behälter mit den Indizes 0 und 1, die 2 und 4 Flaschen beinhalten können, auf genau drei Arten verteilt werden können.

1.1 Berechnen der Möglichkeiten

Es kann jetzt berechnet werden, wie viele Möglichkeiten es gibt, j Flaschen auf die Behälter mit den Indizes 0 bis i zu verteilen. In den Behälter i können genau 0 bis b_i Flaschen hineingetan werden. Daher muss nur überprüft werden, wie viele Möglichkeiten es gibt, in die restlichen Behälter die anderen Flaschen hineinzutun:

- Werden 0 Flaschen in den Behälter i hineingetan, dann müssen in die anderen Behälter genau j Flaschen. Daher gibt es in diesem Fall $f(i-1,j)$ Möglichkeiten. Es kann natürlich auch sein, dass $f(i-1,j)=0$, d. h. es gibt gar keine Möglichkeiten, weil in die restlichen Behälter nicht genug Flaschen passen. Dann gibt es natürlich auch keine Möglichkeiten, wenn in den Behälter i keine Flaschen hineingetan werden.
- Wird 1 Flasche in den Behälter i hineingetan, dann müssen in die anderen Behälter genau $j-1$ Flaschen. Daher gibt es in diesem Fall $f(i-1,j-1)$ Möglichkeiten. Auch hier gilt, dass es passieren kann, dass es 0 Möglichkeiten gibt.
- ...
- Werden b_i Flaschen in den Behälter i hineingetan, dann müssen in die anderen Behälter $j-b_i$ Flaschen. Daher gibt es in diesem Fall $f(i-1,j-b_i)$ Möglichkeiten.

Die Anzahl *aller* Möglichkeiten ist dann einfach die Summe aus allen hier errechneten Möglichkeiten. Ist $b_i > j$, dann ist die maximale Anzahl der Flaschen, die in den Behälter i hineingetan werden, natürlich j statt b_i. In diesem Fall ist einfach b_i durch j zu ersetzen.

Mathematisch ausgedrückt:

$$f(i,j) = \sum_{k=0}^{min(b_i,j)} f(i-1,j-k)$$

Sieht man sich das in der Tabelle 1 für $f(2,5)$ an, so sieht man, dass es 2, 3, 3, 3 und 2 Möglichkeiten gibt, wenn man 0 bis 4 Flaschen in den Behälter 2 tut. Daraus ergeben sich dann insgesamt 13 Möglichkeiten.

Allerdings gibt es eine noch nicht definierte Stelle für den Fall $i = 0$. Da in diesem Fall aber nur berechnet werden muss, wie viele Möglichkeiten es gibt, j Flaschen auf einen einzigen Behälter aufzuteilen, ist die Lösung einfach: wenn $j \leq b_0$, dann gibt es genau eine Möglichkeit, die Flaschen in den Behälter zu tun, wenn $j > b_0$, dann gibt es keine Möglichkeit, die Flaschen in den Behälter zu tun.

Dies ist in der ersten Spalte der Tabelle 1 zu sehen. Es gibt eine Möglichkeit für $j \leq 2$ und keine Möglichkeiten für $j > 2$.

Mathematisch sieht die Funktion nun folgendermaßen aus:

$$f(i,j) = \begin{cases} 1 & \text{für } i = 0 \text{ und } j \leq b_0 \\ 0 & \text{für } i = 0 \text{ und } j > b_0 \\ \sum_{k=0}^{min(b_i,j)} f(i-1,j-k) & \text{für } i > 0 \end{cases}$$

1.2 Optimierungen

1.2.1 Speichern

Berechnet man diese Funktion so wie sie oben steht, so werden bei den meisten Eingaben viele unnötige Berechnungen durchgeführt. Das liegt daran, dass wenn zwei Zellen, die sich in derselben Spalte der Tabelle befinden, beide auf die Zellen der vorherigen Spalte zugreifen können, und diese Zellen u. U. mehrmals berechnet werden. Stattdessen sollten die Ergebnisse von Zellen, die bereits berechnet wurden, nicht erneut berechnet werden, sondern einfach gespeichert werden, damit später darauf zugegriffen werden kann.

1.2.2 Möglichkeiten schneller berechnen

Eine weitere Optimierung besteht darin, dass der Wert einer Zelle nicht immer durch eine Summe von Möglichkeiten aus der Spalte links daneben berechnet werden muss. Aus wie vielen Summanden diese Summe besteht, hängt nämlich von der Größe des Behälters und der Anzahl der Flaschen ab und kann daher sehr groß werden. Im Folgenden wird eine Formel zur Berechnung der Möglichkeiten für eine Zelle hergeleitet, bei der die Anzahl der Summanden nie größer als drei ist.

Fall 1, $j < b_i$: Da

$$f(i,j) = \sum_{k=0}^{j} f(i-1, j-k) = f(i-1,j) + \ldots + f(i-1,0)$$

und

$$f(i,j+1) = \sum_{k=0}^{j+1} f(i-1, j+1-k) = f(i-1,j+1) + f(i-1,j) + \ldots + f(i-1,0)$$

ist

$$f(i,j+1) = f(i,j) + f(i-1,j+1)$$

Fall 2, $j \geq b_i$: Da

$$f(i,j) = \sum_{k=0}^{b_i} f(i-1, j-k) = f(i-1,j) + \ldots + f(i-1,j+1-b_i) + f(i-1,j-b_i)$$

und

$$f(i,j+1) = \sum_{k=0}^{b_i} f(i-1, j+1-k) = f(i-1,j+1) + f(i-1,j) + \ldots + f(i-1,j+1-b_i)$$

ist

$$f(i,j+1) = f(i,j) + f(i-1,j+1) - f(i-1,j-b_i)$$

D. h., wenn $j < b_i$, dann kann die Zelle darunter (Zeile $j+1$) darunter durch $f(i,j+1) = f(i,j) + f(i-1,j+1)$ berechnet werden. Ist $j \geq b_i$, dann kann die Zelle darunter durch $f(i,j+1) = f(i-1,j+1) - f(i-1,j-b_i)$ berechnet werden.

Diese Formeln können nicht für Zellen in Zeile 1 verwendet werden, da sie keine weiteren Zellen über sich haben. Da bei der ersten Zeile $j = 0$, gibt es jedoch immer genau eine Möglichkeit (0 Flaschen verteilen geht immer, egal wie groß die Behälter sind). Hierfür muss also nichtmal eine Addition durchgeführt werden. Die Anzahl der Summanden ist daher jetzt immer höchstens drei.

1.2.3 Überspringen von Berechnungen

Eine weitere Optimierung ist das Überspringen von unnötigen Berechnungen, welche durch die vorherige Optimierung enstehen. Durch diese werden nämlich *alle* Zellen berechnet, da für jede Zelle mindestens die Zelle darüber und die Zelle links daneben berechnet werden müssen. Dies ist jedoch bspw. für die Zelle unten rechts in der Ecke nicht notwendig. Wenn für sie die letzte Formel aus Abschnitt „Berechnen der Möglichkeiten“ verwendet wird (im Folgenden auch „Summenformel“ genannt), um ihr Ergebnis zu berechnen, werden nur einige Zellen in der Spalte links daneben benötigt, statt alle sich darüber befindlichen Zellen sowie deren linke Nachbarn.

Immer, wenn die Summenformel verwendet wird, beginnt die Summe mit der obersten Zelle und geht weiter von oben nach unten. Für die Zelle in der Ecke unten rechts, welche die „Startzelle“ bildet, wird die Summenformel verwendet. Für die oberste berechnete Zelle wird ebenfalls die Summenformel verwendet. Für die anderen Zellen darunter wird die kürzere Formel mit der Berechnung durch den oberen Nachbarn verwendet usw.

Es kann also verallgemeinert werden, dass für eine Zelle, deren oberer Nachbar noch nicht berechnet wurde, immer die Summenformel verwendet wird. Damit dies funktioniert, muss bei der Verwendung dieser Formel von oben nach unten vorgegangen und berechnet werden, da ansonsten ebenfalls für die anderen Zellen die Summenformel verwendet wird, was jedoch nicht optimal wäre. Daher steigt die oberste zu berechnende Zelle, für die die Summenformel verwendet wird, immer weiter nach oben, je weiter man in der Tabelle nach links geht. In den meisten Fällen muss irgendwann die oberste Zeile berechnet werden, dann kann die Summenformel nicht mehr verwendet werden.

Man stelle sich eine Aufgabe vor, bei der 1000000 Flaschen auf 1000000 Behälter aufgeteilt werden müssen, wobei in jeden Behälter keine Flasche passt. Ohne diese Optimierung würde die gesamte Tabelle berechnet werden. Da jedoch in jeder Zeile die Summenformel mit nur einem einzigen Summanden aus der Spalte links daneben verwendet wird, benötigt der Algorithmus nicht viel Zeit. Diese Optimierung bringt am meisten, wenn die oberste zu berechnende Zelle möglichst weit unten ist. Daher funktioniert die Optimierung am besten, wenn die Behälter so sortiert sind, dass am Anfang möglichst kleine Behälter berechnet werden und die Behälter erst zum Schluss größer werden (die kleinsten Behälter in der Tabelle also ganz rechts sind). Dadurch wird die oberste zu berechnende Zelle möglichst weit unten gehalten, wodurch wiederum die Berechnung von einigen Zellen übersprungen wird. Da das Sortieren jedoch weitere Zeit benötigen würde und nicht immer etwas bringt, wird dieser Schritt bei diesem Algorithmus nicht durchgeführt.

1.3 Laufzeitanalyse

Das Berechnen des Wertes einer Zelle benötigt Zeit $\mathcal{O}(1)$, wenn nicht die Summenformel verwendet wird und bereits alle Werte der benötigten anderen Zellen gegeben sind. Es gibt jedoch auch Zellen, welche die Summenformel verwenden. Dies geschieht jedoch immer nur für die oberste zu berechnende Zelle einer Spalte. Alle Zellen darüber werden überhaupt nicht berechnet. Da die Summenformel auch nicht mehr Summanden haben kann, als es Zellen über dieser Zelle (insklusive der Zelle selber) gibt, kann die Laufzeit der obersten zu berechnenden Zelle „aufgeteilt“ werden auf alle Zellen darüber, sodass für jede dieser Zellen die benötigte Zeit ebenfalls $\mathcal{O}(1)$ ist. Da es $n \cdot m$ Zellen gibt und zur Lösung der Aufgabe auch nur eine Zelle berechnet werden muss, beträgt die gesamte Laufzeit im schlimmsten Fall $\mathcal{O}(n \cdot m)$.

2 Umsetzung

Das Programm wurde in der Programmiersprache Swift geschrieben und ist lauffähig auf OS X 10.9 Mavericks und neuer. Da für diese Aufgabe Lösungen bzw. Anzahlen entstehen, für deren Darstellung 64 Bit nicht genügen, und Swift keine größeren Zahlen unterstützt, wird OSXGMP (`https://github.com/githotto/osxgmp`) verwendet, eine Library, die verschiedene Klassen enthält, mit denen man größere Zahlen verwenden kann. Daher wird im Quelltext auch `BigInt` anstelle des normalen `Int` von Swift verwendet. Der Quelltext der verwendeten Klasse dieser Library sowie weitere Dateien befinden sich im Ordner dieser Einsendung.

Zunächst muss ein Befehl wie „chmod 111 PFAD_ZUM_PROGRAMM“ im Terminal ausgeführt werden. Wegen der verwendeten Library müssen die drei Dateien „libgmp.10.dylib“, „libgmp.a“ und „libgmp.dylib“, welche sich im Ordner „Aufgabe3-Implementierung/Aufgabe3-OSXGMP/GMP/“ dieser Einsendung befinden, in den Ordner „/usr/local/lib/“ des Computers, auf dem das Programm ausgeführt wird, kopiert werden. Der letzte Ordner, „/lib/“, muss möglicherweise noch erstellt werden. Da der Ordner „/usr/“ standardmäßig versteckt ist, muss dieser durch den Befehl Shift-Command-G geöffnet werden. Wahrscheinlich wird zur Bearbeitung dieses Ordners das Administratorkennwort benötigt. Danach kann das Programm gestartet werden, indem der Pfad im Terminal eingegeben und durch die Eingabetaste bestätigt wird. Alternativ kann das Icon des Programmes in den Terminal gezogen werden, wodurch der Pfad automatisch eingegeben wird.

2.1 Programmstruktur

- Datei *Task.swift* - Implementation des Algorithmus.
- Datei *TaskReader.swift* - Liest die Aufgabe ein.
- Datei *main.swift* - Startet den `TaskReader` und übergibt diese Eingaben `Task`. Gibt schließlich die Lösung aus.

Das Wichtige geschieht also in *Task.swift*. Dort ist die Klasse `Task` enthalten, welche den Algorithmus umsetzt.

2.2 `Task`

`Task` wird mit der Anzahl der Flaschen und mit einem Array mit `Int`s für die Größen der Behälter erstellt. Es wird dann ein zweidimensionales Array optionaler `BigInt`s erstellt, welche anfangs `nil` sind. Dieses Array beinhaltet die bereits berechneten Werte. Der erste Index dafür ist immer der Index des Behälters und der zweite Index ist die Anzahl der Flaschen. Dieses Array ist stellt daher die Tabelle aus Abschnitt „Lösungsidee“ dar.

Um die Berechnung zu starten, wird `run()` ausgeführt. Dort wird die Funktion `search()` aufgerufen. `search()` benötigt als Parameter den Index des Behälters und die Anzahl der Flaschen, für die die Anzahl der Möglichkeiten berechnet werden soll. `run()` verwendet als Startwerte für diese Methode den Index des letzten Behälters sowie die Anzahl der Flaschen insgesamt, denn die Anzahl der Möglichkeiten für diese Parameter ist die Lösung für die Aufgabe, welche dann zum Schluss zurückgegeben wird.

`run()` arbeitet folgendermaßen:

- Ist der Wert für die übergebenen Parameter bereits im Array gespeichert, so wird dieser einfach zurückgegeben.
- Ist der Index des Behälters 0, d. h., der Behälter ist der erste, dann wird die Anzahl der Flaschen überprüft.
  - Ist die Anzahl der Flaschen kleiner oder gleich dem Fassungsvermögen des Behälters, dann ist der Rückgabewert 1.
  - Ansonsten ist der Rückgabewert 0.
- Ist die Anzahl der Flaschen 0, dann ist der Rückgabewert 1.
- Ist der Wert der Zelle darüber bereits berechnet worden (also nicht `nil`), wird dieser aus dem Array genommen. Dazu wird `search()` mit dem Behälter des um eins kleineren Index und derselben Anzahl an Flaschen addiert. Dies ist, wie unter Lösungsidee beschrieben, die Formel $f(i, j+1) = f(i, j) + f(i-1, j+1)$.
  - Von diesem Wert muss, wenn für die Zelle darüber $j \geq b_i$, d. h. für diese Zelle $j - b_i - 1 \geq 0$ gilt, noch $f(i-1, j-b_i-1)$ subtrahiert werden. Daher wird `search()` mit dem Behälter des um eins kleineren Index und als Anzahl der Flaschen $j - b_i - 1$ aufgerufen und von der bisherigen Anzahl subtrahiert.
- Ansonsten wurde der Wert darüber noch nicht berechnet und die Summenformel wird angewandt. Dafür wird eine Schleife durchlaufen, welche alle Zahlen durchläuft, die angeben, wie viele Flaschen in die vorherigen Behälter getan werden müssen, damit die restliche Anzahl der Flaschen in den aktuellen Behälter hineinpasst. Für jede dieser Zahlen wird dann zu einer Summe das Ergebnis von `search()` mit dem Behälter des um eins kleineren Index und der durch die Schleife berechneten Zahl als Anzahl der Flaschen hinzuaddiert. Diese Summe bildet den Rückgabewert.

Wenn nicht der erste Fall eingetreten ist, dann wird der Rückgabewert zunächst im Array mit allen Werten gespeichert und danach erst zurückgegeben.

3 Beispiele

Die Beispieleingaben sind wie auf der Website angegeben formatiert. In der ersten Zeile befindet sich die Anzahl der Flaschen, in der zweiten Zeile die Anzahl der Behälter und in der dritten Zeile die Anzahl der Flaschen, die in die Behälter hineinpassen.

Alle Beispiele inklusive Lösungen befinden sich zusätzlich im Ordner dieser Einsendung.

3.1 Beispiel 0

```
7
2
3 5
```

Das ist das Beispiel aus der Aufgabenstellung. Es gibt 2 Möglichkeiten, die 7 Flaschen auf die Behälter zu verteilen. Ein gewöhnlicher Computer benötigte für diese Aufgabe deutlich weniger als eine Millisekunde.

3.2 Beispiel 1

```
5
3
2 4 4
```

Für dieses Beispiel gibt es genau 13 Möglichkeiten. Ein gewöhnlicher Computer benötigte für diese Aufgabe weniger als eine Millisekunde.

3.3 Beispiel 2

```
10
3
5 8 10
```

Bei diesem Beispiel gibt es 48 Möglichkeiten. Hier benötigte ein gewöhnlicher Computer weniger als eine Millisekunde.

3.4 Beispiel 3

```
30
20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Hier gibt es 6209623185136 Möglichkeiten, die 30 Flaschen auf die Behälter zu verteilen. Ein gewöhnlicher Computer benötigte ungefähr 2 Millisekunden.

3.5 Beispiel 4

```
3000
11
5 100 100 200 500 500 600 800 800 1000 1000
```

Bei diesem Beispiel existieren genau 743587168174197919278525 Möglichkeiten. Hier benötigte ein gewöhnlicher Computer bereits ca. 100 Millisekunden.

3.6 Beispiel 5

```
2000
18
100 110 130 170 180 200 220 220 230 250 280 300 340 350 350 370 380
    390
```

Die Anzahl der Möglichkeiten lautet: vier Sextilliarden zweihundertsiebenunddreißig Sextillionen sechshundertachtzehn Quintilliarden dreihundertzweiunddreißig Quintillionen einhundertachtundsechzig Quadrilliarden einhundertdreißig Quadrillionen sechshundertdreiundvierzig Trilliarden siebenhundertvierunddreißig Trillionen dreihundertfünfundneunzig Billiarden dreihundertfünfunddreißig Billionen zweihundertzwanzig Milliarden achthundertdreiundsechzig Millionen vierhundertachttausendsechshundertachtundzwanzig. Ein gewöhnlicher Computer benötigte dafür ungefähr 110 Millisekunden.

Die Zahl im Dezimalsystem lautet 42376183321681306437343953352208634086 28. Übrigens: die zugehörige Binärzahl hat 132 Stellen.

4 Quelltext

Listing 1: main.swift

```swift
import Foundation

let solutions = TaskReader().read().run().toString()

print("\(solutions) Lösungen")
```

Listing 2: Task.swift

```swift
import Foundation

// Berechnet die Anzahl der Möglichkeiten
class Task {

  let bottles: Int // Anzahl der Flaschne
  let containers: [Int] // Die Größen der Behälter
```

```
// Bereits berechnete Anzahlen als zweidimensionales Array. Noch nicht
    berechnete Anzahlen sind nil.
// results[x][y] ist die Anzahl der Möglichkeiten für alle Behälter von 0
    bis x und y Flaschen.
var results: [[BigInt?]]

init(bottles: Int, containers: [Int]) {
  self.bottles = bottles
  self.containers = containers
  self.results = [[BigInt?]](count: containers.count, repeatedValue:
      [BigInt?](count: bottles + 1, repeatedValue: nil))
}

// Startet die Berechnung
func run() -> BigInt {
  let start = NSDate()
  let r = search(containers.count-1, bottles: bottles)
  let t = NSDate().timeIntervalSinceDate(start)
  print(t)
  return r
}

// Berechnet die Anzahl der Möglichkeiten, wenn die Behälter 0 bis container
// und bottles Flaschen verwendet werden
func search(container: Int, bottles: Int) -> BigInt {

  if let result = results[container][bottles] {
    // Für diese Zahlen wurde bereits die Anzahl der Möglichkeiten
    // ausgerechnet, daher kann dieses Ergebnis zurückgegeben werden.
    return result
  }

  var result: BigInt

  if container == 0 {
    // Der erste Behälter
    if bottles <= containers[container] {
      result = BigInt(intNr: 1)
    } else {
      result = BigInt(intNr: 0)
    }

  } else if bottles == 0 {
    // Keine Flaschen, daher gibt es eine Möglichkeit.
    result = BigInt(intNr: 1)

  } else if let above = results[container][bottles - 1] {
    // Zelle darüber wurde bereits berechnet
```

```
      // f(i, j+1) = f(i, j) + f(i-1, j+1) berechnen:
      result = above + search(container - 1, bottles: bottles)

      if bottles - containers[container] - 1 >= 0 {
        // - f(i-1, j-b_i), wenn notwendig
        result = result - search(container - 1, bottles: bottles -
            containers[container] - 1)
      }
    } else {
      // Zelle darüber nicht berechnet, also Summe aus den benötigten anderen
          Zellen bilden:
      result = BigInt(intNr: 0)
      for i in max(0, bottles - containers[container])...bottles {
        result += search(container - 1, bottles: i)
      }
    }

    // Ergebnis abspeichern und zurückgeben
    results[container][bottles] = result
    return result
  }

}
```