Flaschenzug

Kevin Schier

30. November 2015

1 Lösungsidee

Für diese Aufgabe habe ich die Lösung in mehreren Schritten verfasst, von daher werde ich hier auch diese Schritte schildern. Anstatt von Flaschen und Körben wird in dieser Lösung generell von Gegenständen und Containern geredet.

Die erste Lösung, die einem für diese Aufgabe in den Sinn kommt ist eine rekursive Methode, die als Parameter die Anzahl an übrigen Gegenständen und die Nummer des Containers hat. Die Methode soll die Anzahl an möglich Permutationen zurückgeben. In dieser Methode wird versucht jede mögliche Anzahl an Gegenständen, die noch übrig sind in den Container zu legen. Dann wird die Methode rekursiv mit der reduzierten Anzahl an Gegenständen und der Nummer des nächsten Containers aufgerufen und die Ergebnisse der Methodenaufrufe zusammenaddiert. Der Basisfall ist der Aufruf mit der Nummer des letzten Containers. Dieser wird immer eins zurückgeben, da es nur eine Möglichkeit gibt, eine beliebige Anzahl an gleichen Gegenständen in einem Container zu verteilen.

Diese Lösung ist allerdings extrem langsam, da die Menge an Funktionsaufrufen lächerlich groß wird. Im Beispiel flaschenzug3.txt sind 20 Container mit aufsteigender Größe. Das sorgt für etwa $20! = 2432902008176640000$ Funktionsaufrufe. Das ist absolut nicht in einem akzeptablem Rahmen. Insgesamt ist diese Version des Algorithmus von der Komplexität her mindestens $\mathcal{O}(N^k)$, wenn nicht noch schlimmer.

Um diese Methode zu verbessern, kam mir die Idee, die berechneten Lösungen abzuspeichern. Da es im Beispiel flaschenzug3.txt nur 30 Gegenstände und 20 Container gibt, sollte es höchstens $30 \cdot 20 = 600$ oder kN tatsächliche Berechnungen geben, alle anderen werden aus dem Speicher geladen. Die vielen Methodenaufrufe und die teuren Lookups im Speicher sorgen aber dafür, dass die eigentlich geringe Anzahl an Berechnungen nochmal stark überschattet wird. Diese Optimierung sorgte allerdings dafür, dass ich das erste mal überhaupt die Lösungen der komplexeren Beispiele sehen konnte. Die Komplexität hat sich hier auf $\mathcal{O}(kN^2)$ verbessert.

Beim Begutachten des internen Speichers der Methode ist mir aufgefallen, dass ein ziemlich großer Prozentsatz der Ergebnisse immer bei allen Beispielen berechnet wurde. Um die Anzahl Methodenaufrufen und die Komplexität eines Lookups im Speicher zu reduzieren, habe ich den Algorithmus nochmal abgeändert. Er berechnet nun alle Ergebnisse iterativ im Vorraus und speichert sie in einem simplen linearen Speicher mit der Größe kN. Damit wurde die Methode noch mal sehr viel simpler und vom System viel einfacher zu berechnen.

Die letzten Optimierungen sind mir aufgefallen, als ich mir überlegt habe wie die berechneten Werte von einander abhängen. Nennen wir alle Werte mit dem gleichen Containerindex als Parameter eine Wertereihe, die Spalten sind alle Werte mit dem gleichen übrige-Gegenstände-Parameter. Die Werte in der ersten Reihe sind alle eins, wie bereits vorhin erklärt, da dort nur

ein Container übrig ist. Alle Werte in den folgenden Reihen sind im Grunde nur die Summe von einem kontinuierlichen Bereich an Werten aus der Reihe darüber. Um die Summen aller Bereiche schnell auf einmal zu berechnen, kann man die partiellen Summen dieser Reihe betrachten, d.h. die Summe aller Werte bis zu einem bestimmten Punkt. Um die Summe über einem Bereich der Reihe zu bekommen, muss man nur die partielle Summe bis zum letzten Element des Bereichs minus der partiellen Summe bis direkt vor dem ersten Element des Bereichs rechnen. Damit lässt sich die Komplexität jetzt auf $\mathcal{O}(kN)$ verbessern.

Da jeder Wert einer Reihe immer nur abhängig von Werten in der Reihe direkt darüber abhängig ist, reicht es immer nur zwei Reihen im Speicher zu haben. Der Speicherverbrauch ist von daher $\mathcal{O}(k + N)$.

2 Umsetzung

Die Umsetzung findet in Visual-C++ 2015 statt.

Um die gigantischen Zahlen zu speichern, wird die BigInteger Bibliothek[1] verwendet.

Die Umsetzung ist eigentlich eins-zu-eins wie in der Lösungsidee beschrieben. Sie ist in der Klasse Flaschenzug zu finden. Die Größen der Container werden in der Klasse gespeichert. Die Methode GetNumberOfPermutationsInternalIterative enthält den eigentlichen Algorithmus. Ich habe darauf verzichtet, die tatsächlichen Verteilungen auszugeben, da diese trivial und viel zu viele sind.

3 Beispiele

Listing 1: Das sind alle gegebenen Beispiele. Die Berechnungszeit ist zu kurz um ernsthaft messbar zu sein.

```
1: Load file
2: All given demos
0: Exit
> 2
flaschenzug0.txt: 2 permutations
flaschenzug1.txt: 13 permutations
flaschenzug2.txt: 48 permutations
flaschenzug3.txt: 6209623185136 permutations
flaschenzug4.txt: 7435871681741979192785 25 permutations
flaschenzug5.txt: 42376183321681306437343953352208634086 28 permutations
```

Listing 2: Eine genauere Betrachtung von flaschenzug2.txt (mit einer modifizierten Version des Programms) um deutlich zu machen wie genau die Ergebnisse nacheinander generiert werden.

```
1: Load file
2: All given demos
0: Exit
> 1
Enter filename:
> flaschenzug2.txt
1  1  1  1  1  1  1  1  1  1
1  2  3  4  5  6  7  8  9  9
1  3  6 10 15 21 27 33 39 44 48
There are 48 ways to arrange the bottles.
```

Listing 3: Ein größeres Beispiel mit 8000 Gegenständen und 49 Containern. Rechenzeit ca. 200ms.

```
1: Load file
2: All given demos
0: Exit
> 1
Enter filename:
> flaschenzugY.txt
There are 158210865344007153958045253261051080988844226777306549563452416511
  146821164963638752037287985440395610705203051754 1 ways to arrange the bottles.
```

[1] https://mattmccutchen.net/bigint/

4 Quelltext

Der Algorithmus.

```
BigUnsigned Flaschenzug::Flaschenzug::GetNumberOfPermutationsInternalIterative(uint_fast32_t items)
{
  uint_fast32_t numContainers = containers.size();
  //these are the combined capacities of the containers beginning at every index
  std::vector<uint_fast32_t> combinedRestContainerCapacities(numContainers);
  std::partial_sum(containers.rbegin(), containers.rend(), combinedRestContainerCapacities.begin());
  //only two rows of item parameters are needed because the dependencies are always exactly one container down
  std::array<std::vector<BigUnsigned>, 2> resultsVectors{};
  resultsVectors.fill(std::vector<BigUnsigned>(items + 1, 1U));
  std::vector<BigUnsigned> *lastColumn = &resultsVectors[0], *currentColumn = &resultsVectors[1];

  //Last container gets implicitly calculated in the construction of the vectors
  //there is always only one way to arrange any amount of equal items in a single container
  if (numContainers <= 1) return 1U;

  for (int numContainersLeft = 2; numContainersLeft <= numContainers; ++numContainersLeft)
  {
    //this is the index of the container currently examined
    uint_fast32_t containerIndex = numContainers - numContainersLeft;
    //this stores the capacity of all containers after the current one
    uint_fast32_t combinedRestContainerCapacity = combinedRestContainerCapacities[numContainersLeft - 2];
    //this calculates the highest items parameter needed from the last row (lower bound is always 0)
    uint_fast32_t highestNeededSum = std::min(items, std::max(combinedRestContainerCapacity, items - containers[
        containerIndex]));
    //this preemptively sums up all the entries from the last row
    std::partial_sum(lastColumn->begin(), lastColumn->begin() + highestNeededSum + 1, lastColumn->begin());
    for (int currentItems = 0; currentItems <= (int) items; ++currentItems)
    {
      //every entry in a row is the sum over a range of entries in the row before
      //the partial sums up to the upper bound - the sums to the lower bound give the sum over the desired range
      //it's like using the antiderivative to get the integral on a range
      uint_fast32_t lowerBound = currentItems - std::min((int) containers[containerIndex], currentItems);
      uint_fast32_t upperBound = clamp(combinedRestContainerCapacity, lowerBound, (uint_fast32_t) currentItems);
      (*currentColumn)[currentItems] = (*lastColumn)[upperBound] -
        (lowerBound == 0 ? 0 : (*lastColumn)[lowerBound - 1]);
    }
    //the newly calculated row is then used as the source for the next one
    std::swap(currentColumn, lastColumn);
  }
  //the last value in the row that just got calculated is the desired result
  return (*lastColumn)[items];
}
```