Aufgabe 3: Telepaartie

Team-ID: 00062

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker

4. November 2019

Inhalt

Lösungsidee

Um für eine Biberverteilung mit der Biberanzahl n geeignete Telepaartien zu finden, um einen Behälter leer zu bekommen, werden alle Verteilungen mit dieser Biberanzahl n berechnet. Jede Verteilung gehört dabei zu einer Generation, wobei sich in der x. Generation (eine Generation vom Grad x) alle Biberverteilungen mit der Leerlauflänge x befinden. Das größte x für eine Biberanzahl ist L(n). Die LLL einer Verteilung wird ermittelt, indem für jede Generation geprüft wird, ob sie diese Verteilung enthält. Der Grad der Generation, die diese Verteilung enthält, ist die LLL.

Finden aller Verteilungen für eine Biberanzahl

Die 0. Generation für eine Biberanzahl wird gebildet, indem alle möglichen Verteilungen auf zwei Behälter gesucht werden (der dritte Behälter ist leer). Für eine Biberanzahl n gibt es n//2 + 1 [1] Möglichkeiten, denn für die Anzahl des einen Behälters werden alle Zahlen von 0 bis zur Hälfte von n durchgegangen und die Anzahl der zweiten Verteilung entspricht dem Rest. Für n=4 sieht die 0. Generation also folgendermaßen aus: (0,0,4), (0,1,3), (0,2,2).
Eine x. Generation wird gebildet, indem alle Verteilungen der (x-1). Generation durchgegangen werden und alle Biberverteilungen gesucht werden, die durch eine Telepaartie zu einer Verteilung der x. Generation werden können. Das heißt, dass die Anzahl für einen Behälter gleich bleibt, während sich die beiden anderen verändern. Es handelt sich sozusagen um eine „Rückwärts“-Telepaartie. Für jede Verteilung in der (x-1). Generation werden alle Permutationen für die Anzahlen durchgegangen und auf diese Verteilung wird eine Rückwärts-Telepaartie angewendet. Aus einer dieser Permutationen (x, y, z) wird dann die neue Biberverteilung (x//2, y+x//2, z). Um aus der neuen Biberverteilung die alte zu erhalten, müsste also eine Telepaartie mit den beiden ersten Behältern durchgeführt werden. Der erste Behälter ist der mit der kleineren, der zweite der mit der größeren Anzahl Bibern.
Damit eine neue Biberverteilung berechnet werden kann, müssen allerdings zwei Bedingungen erfüllt sein: x muss eine gerade Zahl sein und darf nicht 0 sein. Die erste Bedingung muss gelten, weil nach einer Telepaartie der Behälter mit weniger Bibern immer 2b Biber enthält, also eine gerade Zahl, und die zweite Bedingung ist nötig, da, wenn einer der ausgewählten Behälter bei einer Telepaartie keine Biber enthält, sowieso nichts passieren würde (2b=2*0=0).[2] Eine so neu entstandene Biberverteilung wird nur zur neuen Generation hinzugefügt, wenn sie noch in keiner anderen Generation vorhanden ist,[3] denn in diesem Fall gäbe es eine kleinere LLL für diese Verteilung. Die Suche nach allen möglichen Biberverteilungen ist beendet, wenn eine neu erstellte Generation leer ist. Dann wurden keine neuen Biberverteilungen gefunden.

Damit nicht eigentlich gleiche Biberverteilungen für unterschiedlich gehalten werden, weil sich die Reihenfolge der Anzahlen unterscheidet, werden wie in der Aufgabenstellung die Anzahlen in aufsteigend sortierter Reihenfolge dargestellt und gespeichert.

[1] Der //-Operator stellt eine ganzzahlige Division dar.
[2] Die zweite Bedingung könnte auch weggelassen werden, da aber sowieso keine neue Verteilung entsteht, würde unnötig viel Zeit aufgewandt werden.
[3] Zu den anderen Generation zählt auch die, die gerade erstellt wird.

Umsetzung

Das Programm wurde mit Python 3 geschrieben. Getestet wurde es mit Python 3.7. Aufgrund der Verwendung von f-Strings ist das Programm nicht mit Python älter als Version 3.6 kompatibel.

Es gibt insgesamt drei Funktionen:

- `create_beaver_distributions`, um alle Biberverteilung für eine Gesamtzahl Biber zu finden,
- `get_LLL`, um die Leerlauflänge einer Biberverteilung zu bestimmen, und
- `get_telepaartien`, um alle nötigen Telepaartien für eine Biberverteilung zu erhalten.

`create_beaver_distributions` erstellt eine Liste, die alle Generationen enthält (Index 0 enthält eine Liste mit allen Biberverteilung der 0. Generation), und ein Dictionary, welches einer Biberverteilung aus der x. Generation die Biberverteilung aus der (x-1). Generation zuordnet, aus der diese Biberverteilung per Rückwärts-Telepaartie entstanden ist.[4] Mit diesem Dictionary kann erstens einfach festgestellt werden, ob es eine neu gebildete Biberverteilung bereits gibt, indem alle Schüssel nach dieser Verteilung durchsucht werden, und zweitens wird dieses Dictionary von `get_telepaartien` benutzt, um alle Telepaartien für eine Verteilung zu ermitteln.

`create_beaver_distributions` erstellt also zunächst eine Liste mit einem Element, welches selbst eine Liste ist, die alle Biberverteilungen der 0. Generation enthält. Danach werden in einer While-Schleife die Verteilungen aller anderen Generationen berechnet. Dafür werden die Verteilungen der alten Generation durchgegangen und für jede Verteilung werden mit `itertools.permutations` alle Permutationen dieser Verteilung durchgegangen. Sind die zwei in der Lösungsidee beschriebenen Bedingungen für eine Permutation erfüllt, wird eine neue Verteilung nach beschriebenen Regeln berechnet und, falls sie noch nicht gefunden wurde, zur neuen Generation und zum Dictionary hinzugefügt. Sollte eine neue Generation leer sein, bricht die While-Schleife ab und die Funktion gibt die Liste mit allen Generationen und das Dictionary zurück.

`get_LLL` ruft zunächst `create_beaver_distributions` für die Gesamtbiberzahl auf und geht dann alle Generationen beginnend bei der 0. Generation durch. Wenn eine Generation die Verteilung enthält, wird der Grad dieser Generation zurückgegeben. Die Funktion kann beschleunigt werden, wenn `create_beaver_distributions` ein Generator wäre, die jede neu gefundene Verteilung mit `yield` zurückgibt, weil dann nur alle Verteilungen bis zur gesuchten Verteilung berechnet werden müssten. Diese Variante ist mit der Funktion `generate_beaver_distributions` implementiert, die von `get_LLL_generate` statt `create_beaver_distributions` benutzt wird.[5] Es muss allerdings berücksichtigt werden, dass dies nicht immer ein Vorteil ist: Wenn auch noch die Telepaartien berechnet werden sollen, ist es besser, `create_beaver_distributions` zu benutzen, weil diese alle berechneten Ergebnisse mit `functools.lru_cache` cacht und so die Verteilungen nicht mehrfach berechnet werden.

`get_telepaartien` ruft `create_beaver_distributions` auf, um das Dictionary zu erhalten. In diesem Dictionary wird dann für jede Biberverteilung (beginnend bei der übergebenen

[4] Hier kann es mehrere Möglichkeiten geben, da aber eine ausreicht, wird nur eine gespeichert. Den Verteilungen der 0. Generation wird `None` zugeordnet.

[5] Da die beiden Funktionen für die Lösung der Aufgabe nicht relevant sind, werden sie unter „Quellcode“ auch nicht besprochen.

Biberverteilung) die Verteilung gesucht, aus der die alte Biberverteilung entstanden ist, und zwar solange, bis die neue Biberverteilung `None` ist, also die alte Biberverteilung zur 0. Generation gehörte.

Beispiele

Die Beispiele werden berechnet, wenn das Skript aufgabe3.py ausgeführt wird.

Für die Leerlauflängen gibt es folgende Ausgabe:

```
LLL (2,4,7) = 2
Telepaartien für (2,4,7): [(2, 4, 7), (4, 4, 5), (0, 5, 8)]

LLL (3,5,7) = 3
Telepaartien für (3,5,7): [(3, 5, 7), (2, 6, 7), (4, 4, 7), (0, 7, 8)]

LLL (80,64,32) = 2
Telepaartien für (80,64,32): [(32, 64, 80), (48, 64, 64), (0, 48, 128)]
```

Es ist gut zu erkennen, dass die vorletzte Verteilung immer zwei gleiche Anzahlen enthält, da dies der einzige Weg ist, einen Behälter leer zu bekommen.

Für L(1) bis L(10) sieht die Ausgabe folgendermaßen aus:

```
 n | L(n)
 1 |  0  (size=1)
[(0, 0, 1)]

 2 |  0  (size=2)
[(0, 0, 2), (0, 1, 1)]

 3 |  1  (size=3)
[(0, 0, 3), (0, 1, 2)]
[(1, 1, 1)]

 4 |  1  (size=4)
[(0, 0, 4), (0, 1, 3), (0, 2, 2)]
[(1, 1, 2)]

 5 |  1  (size=5)
[(0, 0, 5), (0, 1, 4), (0, 2, 3)]
[(1, 2, 2), (1, 1, 3)]

 6 |  2  (size=7)
[(0, 0, 6), (0, 1, 5), (0, 2, 4), (0, 3, 3)]
[(1, 1, 4), (2, 2, 2)]
[(1, 2, 3)]

 7 |  2  (size=8)
[(0, 0, 7), (0, 1, 6), (0, 2, 5), (0, 3, 4)]
[(1, 3, 3), (1, 1, 5), (2, 2, 3)]
[(1, 2, 4)]

 8 |  2  (size=10)
```

```
[(0, 0, 8), (0, 1, 7), (0, 2, 6), (0, 3, 5), (0, 4, 4)]
[(1, 1, 6), (2, 3, 3), (2, 2, 4)]
[(1, 3, 4), (1, 2, 5)]

 9 |  2  (size=12)
[(0, 0, 9), (0, 1, 8), (0, 2, 7), (0, 3, 6), (0, 4, 5)]
[(1, 4, 4), (1, 1, 7), (3, 3, 3), (2, 2, 5)]
[(2, 3, 4), (1, 2, 6), (1, 3, 5)]

10 |  2  (size=14)
[(0, 0, 10), (0, 1, 9), (0, 2, 8), (0, 3, 7), (0, 4, 6), (0, 5, 5)]
[(1, 1, 8), (2, 4, 4), (2, 2, 6), (3, 3, 4)]
[(1, 4, 5), (1, 3, 6), (1, 2, 7), (2, 3, 5)]
```

Es wird jeweils n, die Gesamtbiberzahl, ausgegeben, gefolgt von der Anzahl der Generationen minus eins, was L(n) entspricht, und der Anzahl von Verteilungen (`size`). Darauf folgen alle Generationen, beginnend bei der 0. Generation.

Für die folgenden L(11) bis L(100) wurde auf eine Ausgabe der Generationen verzichtet:

```
11 |  3  (size=16)
12 |  3  (size=19)
13 |  3  (size=21)
14 |  3  (size=24)
15 |  4  (size=27)
16 |  3  (size=30)
17 |  3  (size=33)
18 |  3  (size=37)
19 |  4  (size=40)
20 |  3  (size=44)
21 |  4  (size=48)
22 |  4  (size=52)
23 |  5  (size=56)
24 |  4  (size=61)
25 |  5  (size=65)
26 |  4  (size=70)
27 |  6  (size=75)
28 |  4  (size=80)
29 |  5  (size=85)
30 |  5  (size=91)
31 |  5  (size=96)
32 |  4  (size=102)
33 |  6  (size=108)
34 |  4  (size=114)
35 |  5  (size=120)
36 |  4  (size=127)
37 |  5  (size=133)
38 |  5  (size=140)
39 |  6  (size=147)
40 |  4  (size=154)
```

```
41 |  5  (size=161)
42 |  5  (size=169)
43 |  6  (size=176)
44 |  5  (size=184)
45 |  7  (size=192)
46 |  5  (size=200)
47 |  5  (size=208)
48 |  5  (size=217)
49 |  6  (size=225)
50 |  6  (size=234)
51 |  7  (size=243)
52 |  5  (size=252)
53 |  6  (size=261)
54 |  6  (size=271)
55 |  6  (size=280)
56 |  5  (size=290)
57 |  7  (size=300)
58 |  5  (size=310)
59 |  6  (size=320)
60 |  6  (size=331)
61 |  7  (size=341)
62 |  6  (size=352)
63 |  7  (size=363)
64 |  5  (size=374)
65 |  6  (size=385)
66 |  7  (size=397)
67 |  6  (size=408)
68 |  5  (size=420)
69 |  7  (size=432)
70 |  6  (size=444)
```

```
71 |  6  (size=456)
72 |  5  (size=469)
73 |  6  (size=481)
74 |  6  (size=494)
75 |  7  (size=507)
76 |  6  (size=520)
77 |  7  (size=533)
78 |  6  (size=547)
79 |  6  (size=560)
80 |  5  (size=574)
81 |  8  (size=588)
82 |  6  (size=602)
83 |  7  (size=616)
84 |  6  (size=631)
85 |  7  (size=645)
86 |  6  (size=660)
87 |  7  (size=675)
88 |  6  (size=690)
89 |  7  (size=705)
90 |  7  (size=721)
91 |  7  (size=736)
92 |  6  (size=752)
93 |  8  (size=768)
94 |  6  (size=784)
95 |  8  (size=800)
96 |  6  (size=817)
97 |  7  (size=833)
98 |  7  (size=850)
99 |  8  (size=867)
100|  7  (size=884)
```

Interessant ist, dass L(n) zwar insgesamt mit steigendem n zunimmt, aber auch L(n) < L(m) für n > m gelten kann.

Quellcode

Bedeutung der Skripte

```
python3 aufgabe3.py
```

Dieses Skript enthält die vier Funktionen und berechnet, wenn es direkt ausgeführt wird (und nicht importiert wird), die Beispiele.

```
aufgabe3_LLL.py
```

Wird dieses Skript ausgeführt, können Biberverteilungen eingegeben werden, woraufhin deren LLL berechnet werden und die nötigen Telepaartien ausgegeben werden.

```
aufgabe3_L(n).py
```

Dieses Skript berechnet L(n) für eingegebene n.

create_beaver_distributions

Der Funktion wird die Gesamtbiberanzahl `beaver_count` übergeben. Die Funktion sieht folgendermaßen aus:

```
@lru_cache()
def create_beaver_distributions(beaver_count):
    # 0. Generation berechnen
    generation = [(0, second_count, beaver_count-second_count)
                                                for second_count in range(beaver_count//2+1)]
    # Dictionary, das allen Verteilungen die Verteilung zuordnet, aus der sie entstanden sind.
    all_distributions = {dist: None for dist in generation}
    generations = [generation]  # alle Generationen, im Moment nur die 0. Generation
    while True:
        generation = []
        for parent in generations[-1]:  # alle Verteilungen der letzten Generation durchgehen
            for beaver_dist_permutation in itertools.permutations(parent):  # alle
                                                                    # Permutationen durchgehen
                # die zwei Bedingungen für eine Permutation überprüfen:
                if beaver_dist_permutation[0] % 2 == 0 and beaver_dist_permutation[0] != 0:
                    half = beaver_dist_permutation[0] // 2
                    # die Anzahlen werden aufsteigend sortiert, damit eigentlich gleiche
                    # Verteilungen erkannt werden
                    new_beaver_dist = tuple(sorted((half, beaver_dist_permutation[1] + half,
                                                            beaver_dist_permutation[2])))
                    if new_beaver_dist not in all_distributions:
                        # new_beaver_dist gibt es noch nicht, an neue Generation anfügen und im
                        # Dictionary zuordnen
                        generation.append(new_beaver_dist)
                        all_distributions[new_beaver_dist] = parent
        if not generation:
            # es gab keine neuen Verteilungen; alle Verteilungen wurden gefunden
            break
        generations.append(generation)
    return generations, all_distributions
```

get_LLL

```python
def get_LLL(beaver_dist):
    beaver_dist = tuple(sorted(beaver_dist))  # Anzahlen aufsteigend sortieren
    generations, _ = create_beaver_distributions(sum(beaver_dist))
    for i, generation in enumerate(generations):
        if beaver_dist in generation:
            # beaver_dist wurde in der i. Generation gefunden
            return i
    raise ValueError  # hierhin sollte das Programm nicht gelangen
```

get_telepaartien

```python
def get_telepaartien(beaver_dist):
    beaver_dist = tuple(sorted(beaver_dist))  # Anzahlen aufsteigend sortieren
    _, dists = create_beaver_distributions(sum(beaver_dist))
    way = [beaver_dist]  # erste Verteilung: die Verteilung selbst
    while True:
        beaver_dist = dists[way[-1]]  # neue Biberverteilung, die aus way[-1] per
                                      # Telepaartie entstehen kann
        if beaver_dist is None:
            return way  # beaver_dist gehört zur 0. Generation
        way.append(beaver_dist)
```