Aufgabe 2: Nummernmerker

Richard Wohlbold
Team-ID: 00487

30. Oktober 2019

Inhaltsverzeichnis

1 Lösungsidee

Das Problem kann rekursiv gelöst werden. Von einer Nummer können, bei ausreichender Länge, stets 2, 3 oder 4 Ziffern vom Anfang entfernt werden. Diese Ziffern können den ersten Block bilden, die restlichen Ziffern sind neue Nummern, auf die das Verfahren erneut angewendet werden kann. Um zu ermitteln, ob nun ein Block mit 2, 3 oder 4 Ziffern verwendet werden soll, wird die Anzahl von Blöcken, die mit 0 beginnen, für jede der drei Optionen verglichen und diejenige mit der geringsten Anzahl ausgewählt. Der Basisfall ist erreicht, falls die Länge der Nummer ≤ 4 ist:

1. Ist die Länge der Nummer 4, so gibt es nur eine Möglichkeit: Die gesamte Nummer zu einem aus vier Ziffern bestehenden Block machen.
2. Ist die Länge der Nummer 3, so gibt es nur eine Möglichkeit: Die gesamte Nummer zu einem aus drei Ziffern bestehenden Block machen.
3. Ist die Länge der Nummer 2, so gibt es nur eine Möglichkeit: Die gesamte Nummer zu einem aus zwei Ziffern bestehenden Block machen.
4. Ist die Länge der Nummer 1, so ist es unmöglich, diese Nummer in Blöcke umzuwandeln. Dies muss der aufrufenden Prozedur signalisiert werden.

Dieses Verfahren klingt zwar sehr aufwendig, da für n Ziffern insgesamt n^3 Aufrufe entstehen würden (Komplexität von $\mathcal{O}(n^3)$), jedoch kann der Algorithmus, durch Implementierung eines Caches, deutlich effizienter gestaltet werden. Im Cache wird zu jeder Nummer die entsprechende optimale Blockaufteilung gespeichert. Somit wird für sie maximal für jede Ziffer einmal berechnet, sodass die Komplexität des Algorithmus zu $\mathcal{O}(n)$ wird.

2 Umsetzung

Den Algorithmus habe ich in Python 3 umgesetzt. Dazu definiere ich eine Funktion `loesen`, die als Parameter die in Blöcke einzuteilende Nummer als `str` annimmt. Zurück gibt sie die optimale Blockaufteilung sowie die Anzahl der Blöcke, die mit einer 0 beginnen. Zuerst schaut die Funktion in den Cache, einem `dict` ($\mathcal{O}(1)$ Zugriffszeit), und schaut, ob `solve` bereits für die Nummer aufgerufen wurde. Daraufhin überprüft die Funktion, ob sie überhaupt etwas tun kann, d.h. ob die Länge der Nummer nicht 0 oder 1 ist. Ist sie 0, so gibt die Funktion eine leere Liste an Blöcken, sowie die Zahl 0 für die Anzahl an mit 0 anfangenden Blöcken zurück. Ist sie 1, so gibt die Funktion eine leere Liste an Blöcken zurück, sowie die Zahl ∞ (`math.inf`), die zeigt, dass die Nummer nicht aufgeteilt werden konnte. Der Vorteil ist, dass beim Vergleich mehrerer Optionen die Nummer, die nicht aufgeteilt werden konnte, automatisch ignoriert wird, da alle anderen Aufteilungen weniger mit 0 beginnende Blöcke haben als sie.

Nun kann die Funktion sich selbst aufrufen. Als erstes ruft sie sich mit derselben Nummer ohne ihre ersten beiden Ziffern auf und speichert die Blöcke sowie die Anzahl der Blöcke, die mit 0 anfängt. Falls die Nummer mindestens drei Ziffern enthält, ruft sie sich danach mit derselben Nummer ohne ihre ersten drei Ziffern auf, vergleicht die Ergebnisse und speichert das bessere, d.h. das mit weniger Blöcken, die mit 0 beginnen. Falls die Nummer mindestens vier Ziffern enthält, ruft sie sich danach mit derselben Nummer ohne ihre ersten vier Ziffern auf, vergleicht die Ergebnisse und speichert das bessere. Zum Schluss wird die Zahl an Blöcken, die mit 0 beginnen, um 1 erhöht, falls die Nummer mit 0 beginnt, die Ergebnisse werden in den Cache geschrieben und zurückgegeben.

Die aufrufende Prozedur stellt sicher, dass nur numerische Nummern akzeptiert werden und behandelt den Fehler, dass ∞ Blöcke mit 0 beginnen.

3 Beispiele

3.1 1. Beispiel von der Website

Dies ist das erste Beispiel von der `bwinf`-Website. Selbst ohne Programm ist es ersichtlich, dass mindestens zwei Blöcke mit 0 anfangen müssen, nämlich der erste und einer, der in der Mitte der langen Kette an 0en anfängt:

```
$ python a2-nummernmerker/Implementierung/main.py 005480000005179734
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 0054, 8000, 0005, 1797, 34
2 von 5 Blöcken fangen mit 0 an.
$
```

3.2 2. Beispiel von der Website

Hier ist es ersichtlich, dass nur der erste Block mit einer 0 beginnen muss. Das Programm bevorzugt längere Blöcke, solange nicht mehr Blöcke mit 0 beginnen, um die Anzahl der Blöcke für Sarahs Mutter zu reduzieren.

```
$ python a2-nummernmerker/Implementierung/main.py 034959295337901544126 60
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 0349, 5929, 5337, 9015, 4412, 660
1 von 6 Blöcken fangen mit 0 an.
$
```

3.3 3. Beispiel von der Website

Die meisten Nummern brauchen keine Blöcke, die mit 0 anfangen. Auch bei langen Nummern braucht das Programm nur $0.037s$.

```
$ python a2-nummernmerker/Implementierung/main.py 531997487902272560762 0179
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 5319, 9748, 7902, 2725, 6076, 201,
    ↪ 79
0 von 7 Blöcken fangen mit 0 an.
$
```

3.4 4. Beispiel von der Website

```
$ python a2-nummernmerker/Implementierung/main.py 90887610516994827890383312671
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 9088, 7610, 5169, 9482, 7890, 3833,
    ↪ 1267
0 von 7 Blöcken fangen mit 0 an.
$
```

3.5 5. Beispiel von der Website

Bei binären Nummern treten naturgemäß viele Blöcke auf, die mit 0 beginnen:

```
$ python a2-nummernmerker/Implementierung/main.py 01100000001100010011111110101 1
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 0110, 0000, 001, 1000, 1001, 1111,
    ↪ 110, 1011
3 von 8 Blöcken fangen mit 0 an.
$
```

3.6 Eine sehr lange Nummer

Auch sehr lange Nummern werden vom Programm aufgrund des Caches schnell (0.048s) verarbeitet:

```
$ python a2-nummernmerker/Implementierung/main.py
    ↪ 12839582109385092138509218509823095809328509125092183127359872153912 35
Die Blöcke, die sich Sarahs Mutter merken muss, sind: 1283, 9582, 1093, 8509, 2138, 5092,
    ↪ 1850, 982, 3095, 8093, 2850, 912, 5092, 1831, 2735, 9872, 1539, 1235
0 von 18 Blöcken fangen mit 0 an.
$
```

4 Quellcode (ausschnittsweise)

Einige Module importieren und den Cache initialisieren:

```python
import math
import sys

cache = {}
```

imports.py

Die Funktion `solve`:

```python
def loesen(nummer):
    if nummer in cache:
        return cache[nummer]

    if len(nummer) == 0:
        return [], 0
    elif len(nummer) == 1:
        return [], math.inf

    bloecke, nullen = loesen(nummer[2:])
    laenge = 2

    if len(nummer) >= 3:
        bloecke3, nullen3 = loesen(nummer[3:])
        if nullen3 <= nullen:
            bloecke = bloecke3
            nullen = nullen3
            laenge = 3

    if len(nummer) >= 4:
        bloecke4, nullen4 = loesen(nummer[4:])
        if nullen4 <= nullen:
            bloecke = bloecke4
            nullen = nullen4
```

```
            laenge = 4

    if nummer[0] == '0':
        nullen += 1

    bloecke = [nummer[:laenge]] + bloecke
    cache[nummer] = (bloecke, nullen)
    return bloecke, nullen
```

solve.py

Die Prozedur, die `solve` aufruft, wurde wegen ihrer geringen Relevanz für das zu lösende Problem hier ausgelassen.