Vorwort

Alle Lösungen zur Aufgabe sind getestet und geschrieben auf einem 64-Bit GNU/Linux-Rechner mit Python 2.7.8 und 3.4.2. Die Dokumentation ist geschrieben in Markdown und später konvertiert zu einem PDF.

Alle Lösungen sind innerhalb von zwei Wochen während der Schulzeit entstanden.

Der Code und die Dokumentation werden mit Git verwaltet und werden nach Abgabetermin auf GitHub verfügbar sein.

Alphametiken

Das Programm ist in zwei Teile gegliedert:

- `loese(zu_lösendes_Alphametikum)` gibt eine Liste mit allen Lösungen eines Alphametikums in dem Format `[['Rechnung', {'Buchstabe': zugehörige_Zahl, ...}], ...]` zurück (zum Beispiel `[['8928-3164=5764', {'Ü': 9, 'F': 8, 'W': 7, 'N': 2, 'Z': 5, 'I': 4, 'E': 6, 'D': 3, 'R': 1}]]` für `FÜNF-DREI=ZWEI`).
- `generiere(Länge)` gibt ein Alphametikum als String zurück (zum Beispiel `FÜNF-DREI=ZWEI`), für welches mindestens eine Lösung vorhanden ist.

Alphametikum lösen

Die Funktion `ersetze`, aufgerufen durch `loese`, ersetzt jeweils den ersten Buchstaben im Alphametikum durch eine Zahl, nacheinander 0 bis 9. Für den Rest des Alphametikum-Strings wird die Funktion rekursiv aufgerufen. Ist der letzte Buchstabe ersetzt, wird mithilfe von Pythons `eval`-Funktion überprüft, ob die ersetzten Zahlen zu einer wahren Gleichung führen. Ist dies der Fall, wird die gefundene Lösung in einer Liste festgehalten und nach Überprüfen aller Kombinationen zurückgegeben. Der Ausnahmefall, dass die Lösung eine Null als erste Ziffer einer Zahl hat, wird durch einen regulären Ausdruck `re.search` überprüft. Um gleiche Zahlen nicht an mehrere Buchstaben zu verteilen wird in der rekursiven Funktion die Liste an bereits verwendeten Zahlen mitgegeben.

Beispiel für eine Lösung:

```
>>> import alphametiken
>>> alphametiken.loese("SEND + MORE = MONEY")
[['9567 + 1085 = 10652', {'O': 0, 'N': 6, 'M': 1, 'E': 5, 'Y': 2, 'S': 9, 'D': 7, 'R': 8}]]
```

Alphametikum finden

Zum Finden von Alphametiken in der Form `FÜNF-DREI=ZWEI` ist es zuerst nötig, eine valide Gleichung - hier `5-3=2` zu finden. Um dies für eine beliebige Länge zu tun, wird in der Funktion `generiere` ein Template erstellt, in dem `_` dann durch ein zufälliges Rechenzeichen, `x` und `y` durch eine zufällige Zahl von 0-9 ersetzte wird, sodass bei einer Länge von 1 dieses Schema entsteht:

```
x_x=y
```

das nun auf Richtigkeit, wieder durch `eval`, geprüft wird. Eine Gleichung wie `5-3=2` wird schließlich mithilfe von Zahlwörtern abgebildet: `FÜNF-DREI=ZWEI`. Jetzt wird mit `loese` geprüft, ob eine Lösung vorhanden ist. Wenn ja, endet die Funktion, sonst wird nach einer neuen Gleichung gesucht. `random.choice` und `random.randrange` dienen hier zum Finden zufälliger Rechenzeichen und Zahlen.

Beispiel für ein Alphametikum mit mehr als 20 Zeichen:

```
>>> import alphametiken
>>> alphametiken.generiere(3)
'FÜNF-ZWEI+VIER-ZWEI=FÜNF'
>>> alphametiken.loese('FÜNF-ZWEI+VIER-ZWEI=FÜNF')
...
['9589-3102+6204-3102=9589', {'Z': 3, 'V': 6, 'I': 2, 'Ü': 5, 'R': 4, 'E': 0, 'N': 8, 'F': 9,
...
>>> alphametiken.generiere()
'SECHS-VIER=ZWEI'
>>> alphametiken.loese('SECHS-VIER=ZWEI')
[['12051-3427=8624', {'H': 5, 'W': 6, 'E': 2, 'Z': 8, 'I': 4, 'S': 1, 'C': 0, 'R': 7, 'V': 3}]
>>> alphametiken.generiere(2)
'ACHT-ZWEI-DREI=DREI'
>>> alphametiken.loese('ACHT-ZWEI-DREI=DREI')
[['4567-2389-1089=1089', {'T': 7, 'A': 4, 'W': 3, 'E': 8, 'Z': 2, 'I': 9, 'H': 6, 'C': 5, 'D':
...
```

Buffet-Lotterie

Das Programm besteht aus der Funktion `denke(Anzahl_Teilnehmer, Silben_als_Liste)`, die die Lösung auf der Konsole ausgibt.

Gedanken dazu

- Es ist egal, wer welche Silbe sagt. Wichtig ist nur, wer die letzte spricht und wie oft das Geburtstagskind die Möglichkeit hat, zwei zu sprechen.
- Es ist - für das Geburtstagskind - egal, in welcher Reihenfolge es Eine und Zwei Silben spricht, solange die Anzahl an Zweisilbern insgesamt gleich bleibt, denn
- Es ist von Anfang an klar, nach wie vielen Runden das Geburtstagskind drankommt und wie viele Silben es spricht. Bei der gleichen Anzahl an Silben und Start-Teilnehmern ist die Anzahl an Runden, die das Geburtstagskind braucht, um zum Essen zu kommen konstant.

Lösung

Nach langem Nachdenken ergaben sich die Formeln

```
e = floor((len(silben) + offset) / teilnehmer)
offset = ((len(silben) + offset - 1) % teilnehmer) % (teilnehmer - 1
```

wobei `e` die Anzahl Runden darstellt, die das Geburtstagskind eine Silbe sagen durfte, bis jemand zum Essen ging und `offset` - mit dem Startwert 0 - die Nummer des Teilnehmers, der als nächstes anfangt hat. Um dann noch auszurechnen, wie weit diese Teilnehmernummer `offset` vom Geburtstagskind entfernt ist, kann man

```
fehlt_noch = (offset - 1) % (teilnehmer - 1) + 1
```

benutzen. Diese Berechnungen werden in einer Schleife durchgeführt. Um zu berechnen, wie oft das Geburtstagskind optimalerweise zwei Silben sagen muss, ist es nötig, die Gesamtzahl an `e`s zu behalten. Das geschieht in `einfluss`. Jedes Mal, nachdem ein Teilnehmer in der Rechnung die Runde verlässt, wird geprüft, ob der `einfluss` groß genug ist, um die Lücke `fehlt_noch` auszugleichen. Wenn ja, ist die Aufgabe gelöst und das Geburtstagskind kann `fehlt_noch`-mal zwei Silben nennen und früher Essen gehen. Die Lösung wird durch Zustands-Nachrichten verständlicher gemacht.

Beispiele

6 Teilnehmer mit "Informatik kann uns..."

```
>>> import buffet_lotterie
>>> buffet_lotterie.denke(6, ['In', 'for', 'ma', 'tik', 'kann', 'uns
Nach 2 Runden fängt der 4. an. Es sind dann noch 5 Teilnehmer im Spi
Es fehlten 3 bis zum Geburtstagskind
Nach 3 Runden fängt der 4. an. Es sind dann noch 4 Teilnehmer im Spi
Es fehlten 3 bis zum Geburtstagskind
Das Geburtstagskind muss erst 3-mal 'zwei Silben' und dann nur noch
```

28 Teilnehmer, wie in der Aufgabe

```
>>> buffet_lotterie.denke(28, ['In', 'for', 'ma', 'tik', 'kann', 'un
Nach 0 Runden fängt der 16. an. Es sind dann noch 27 Teilnehmer im S
Nach 1 Runden fängt der 4. an. Es sind dann noch 26 Teilnehmer im Sp
Es fehlten 3 bis zum Geburtstagskind
Nach 0 Runden fängt der 19. an. Es sind dann noch 25 Teilnehmer im S
Nach 1 Runden fängt der 9. an. Es sind dann noch 24 Teilnehmer im Sp
Es fehlten 8 bis zum Geburtstagskind
Nach 1 Runden fängt der 1. an. Es sind dann noch 23 Teilnehmer im Sp
Es fehlten 23 bis zum Geburtstagskind
Nach 0 Runden fängt der 16. an. Es sind dann noch 22 Teilnehmer im S
Nach 1 Runden fängt der 9. an. Es sind dann noch 21 Teilnehmer im Sp
Es fehlten 8 bis zum Geburtstagskind
Nach 1 Runden fängt der 3. an. Es sind dann noch 20 Teilnehmer im Sp
Es fehlten 2 bis zum Geburtstagskind
Das Geburtstagskind muss erst 2-mal 'zwei Silben' und dann nur noch
```

Fähre füllen

Die Lösung zu Fähre füllen beinhaltet wie gefordert zwei Strategien `strategie_a`, `strategie_b`. Beide Funktionen nehmen eine Liste mit Autolängen als Eingabe und geben ein `dict` mit der Länge und Anzahl aller Autos zurück. Mit `print` wird angegeben, wo die Autos einsortiert werden.

Strategie A

Die Fahrzeuge werden möglichst weit links angeordnet. Am effizienten ist das bei einer Folge wie beispielsweise

```
6, 6, 6, 15, 15, 15, ...
```

da hierbei die meiste Anzahl an Fahrzeugen hineinpasst, verglichen mit B.

Strategie B

Die Fahrzeuge werden möglichst gleichmäßig auf die drei Spalten verteilt. Die Folge vom Beispiel in A) würde dazu führen, dass nur 3 Fahrzeuge (`6, 6, 6`) Platz hätten. In dem Beispiel

```
8, 8, 8, 5, 5, 5, 5, 5, 5, ...
```

werden für A) 8 Fahrzeuge eingeordnet und für B) 9.

Beispiele

```
>>> # Optimales Beispiel für A
>>> import faehre_fuellen
>>> faehre_fuellen.strategie_a([15, 15, 15, 15, 15, 6, 6, 6])
6 wurde eingeordnet in die Reihe 0
6 wurde eingeordnet in die Reihe 0
6 wurde eingeordnet in die Reihe 0
15 wurde eingeordnet in die Reihe 1
15 wurde eingeordnet in die Reihe 2
Das Fahrzeug 15 hat nicht mehr gepasst
{'anzahl': [3, 1, 1], 'länge': [18, 15, 15]}
>>> faehre_fuellen.strategie_b([15, 15, 15, 15, 15, 6, 6, 6])
6 wurde eingeordnet in Reihe 0
6 wurde eingeordnet in Reihe 1
6 wurde eingeordnet in Reihe 2
Das Fahrzeug 15 hat nicht mehr gepasst
{'anzahl': [1, 1, 1], 'länge': [6, 6, 6]}
>>>
>>> # Optimales Beispiel für B
>>> faehre_fuellen.strategie_a([5, 5, 5, 5, 5, 5, 5, 8, 8, 8])
...
Das Fahrzeug 5 hat nicht mehr gepasst
{'anzahl': [2, 3, 3], 'länge': [16, 18, 15]}
>>> faehre_fuellen.strategie_b([5, 5, 5, 5, 5, 5, 5, 8, 8, 8])
...
Das Fahrzeug 5 hat nicht mehr gepasst
{'anzahl': [3, 3, 3], 'länge': [18, 18, 18]}
>>>
>>>
>>> # Gegebene Beispiele befinden sich in einer Liste `Beispiel`
>>> # (Die Ausgabe ist hier auf das Ergebnis beschränkt)
>>> faehre_fuellen.strategie_a(faehre_fuellen.Beispiel[0])
{'anzahl': [3, 3, 1], 'länge': [18.47, 16.09, 9.12]}
>>> faehre_fuellen.strategie_b(faehre_fuellen.Beispiel[0])
{'anzahl': [2, 3, 2], 'länge': [15.989999999999998, 19.2, 8.49]}
>>> # Beispiel 1: weder A) noch B) ist effizienter (jeweils 7 Fahrze
>>>
>>> faehre_fuellen.strategie_a(faehre_fuellen.Beispiel[1])
{'anzahl': [2, 3, 3], 'länge': [19.92, 16.48, 11.69]}
>>> faehre_fuellen.strategie_b(faehre_fuellen.Beispiel[1])
{'anzahl': [2, 3, 3], 'länge': [18.69, 18.04, 11.36]}
>>> # Beispiel 2: weder A) noch B) ist effizienter (jeweils 8 Fahrze
>>>
>>> faehre_fuellen.strategie_a(faehre_fuellen.Beispiel[2])
{'anzahl': [5, 4, 3], 'länge': [18.91, 17.630000000000003, 12.780000
>>> faehre_fuellen.strategie_b(faehre_fuellen.Beispiel[2])
{'anzahl': [4, 4, 4], 'länge': [15.030000000000001, 16.86, 17.43]}
>>> # Beispiel 3: weder A) noch B) ist effizienter (jeweils 12 Fahrz
```

Faires Füllen

Die Aufgabe ist unbearbeitet, nur Lösungsideen sind vorhanden. Ideen zur Aufgabe:

- Lösen durch (rekursives) Probieren aller Kombinationen
- oder: sortieren nach Größe; nacheinander immer in das rechte Gefäß kippen und (anschließend) nach der richtigen Summe suchen; zurückkippen
- nur lösbar, wenn
  - eine Differenz zweier Behälter Teiler der Hälfte der Flüssigkeit ist?

Mobile

Textuelle Darstellung

Die textuelle Darstellung setzt sich aus einer Liste von Position-Gewichts-Tuples zusammen. Ein Gewicht ist hierbei ebenfalls eine Liste. Positionen sind auf der einen Seite negativ, auf der anderen positiv. Ein Balken lässt sich daran erkennen, dass er mehr als ein Gewicht hat. Eine Figur ist an der Position 0 mit dem Gewicht der entsprechenden Figur. Beispiel für ein Mobile mit einer Figur des Gewichts 2: `[(0, 2)]` Beispiel für einen Balken mit zwei Figuren des Gewichts 3: `[(-3, 3), (3, 3)]`

Das abgebildete Mobile sieht in der textuellen Darstellung wie folgt aus:

```
[(-4, [(-2, [(0, 1)]), (1, [(0, 2)])]), (-1, [(0, 2)]), (2, [(0, 3)]
```

Umsetzung

Die Lösung besteht aus zwei Funktionen, `split(alle_Figuren_an_einem_Balken)`, welche eine textuelle Darstellung verteilt. Diese eingehende Darstellung ist ein einziger Balken mit allen Figuren, die mithilfe dieser Funktion ausgeglichen werden. Im Endergebnis hat dann jeder Balken maximal vier Gewichte. Die Liste, die zurückgegeben wird, ähnelt der textuellen Darstellung, jedoch ist statt der Position das Gewicht des Gewichts in dem Tuple enthalten. Diese Liste wird von der zweiten Funktion `weight(Liste_erstellt_durch_split)` angeglichen. Aus Benutzerfreundlichkeitsgründen fasst eine dritte Funktion `mobile(List)` diese zwei zusammen. Sie benötigt nur noch eine Liste aller Gewichte für das Mobile.

Die Funktion `split` sortiert alle Items der mitgelieferten Liste in vier Plätze ein (realisiert mit dem Modulo-Operator). Das Gesamtgewicht dieser vier Plätze wird berchnet, dann wird jedes Gewicht in diesen vier Plätzen durch einen rekursiven Funktionsaufruf erneut verteilt.

`weight` berechnet die Position mit der Gegenzahl des Gesamtewichts des Gewichts an der gegenüberliegenden Position. Ist die Anzahl an Gewichten auf dem Balken ungerade, wird ein Gewichte in die Mitte (Position 0) platziert. Durch diese Verteilung gleichen sich die jeweils äußersten und innersten Gewichtpaare aus. Auch diese Funktion ruft sich selbst auf, um die Gewichte von der Spitze des Mobiles hinab bis zu den Figuren alle zu verteilen.

Beispiele

```
>>> import mobile
>>> mobile.mobile([1, 4])
[(-4, 1), (1, 4)]
>>> mobile.mobile([4, 6, 2, 3])
[(-3, 4), (-2, 6), (6, 2), (4, 3)]
>>> mobile.mobile([1, 3, 5, 2, 3, 6])
[(-2, [(-3, 1), (1, 3)]), (-5, [(-6, 3), (3, 6)]), (9, [(0, 5)]), (4
>>> mobile.mobile([4, 6, 2, 3, 45, 6, 3, 6, 1, 8, 5, 3, 67, 24])
[(-12, [(-67, 4), (-1, 45), (45, 1), (4, 67)]), (-10, [(-24, 6), (-8
```

Pong

Der Pong-Bot ist als 'TPong' des Benutzers 'tmo' angemeldet. Der Bot berechnet die Flugbahn des Balls und versucht, diese mit einem Delta-X und Delta-Y verschiedener Messwerte hervorzusagen. Ist der Ball für eine Berechnung zu nahe, begibt sich der Schläger auf Höhe des Balls.

Aufgrund Zeitmangels ist der Code nicht fertig geworden und der Bot erfüllt seine Funktion nicht perfekt.

Zahlenspiel

Die Aufgabe ließe sich mit `random`s `randrange()` lösen. Zu Beginn der Bruchsuche müssen `p` und `q` gefunden werden, die der Anforderung des Schwierigkeitsgrades entsprechen und sich, geschrieben als Bruch, nicht kürzen lassen. Das Nicht-Kürzen-Lassen lässt sich mit dem Greatest Common Divisor (größter gemeinsamer Teiler) berechnen - ist er eins, lässt sich `p/q` nicht kürzen. Um die Zahlen `p` `q` innerhalb der Schwierigkeitsgrenzen zu halten ist es sinnvoll, nach dem Generieren einer zufälligen Zahl `Maximum > p > 1` die Zahl `q` abhängig davon zu finden: `Maximum - p > q > Maximum - 10 - p`

Als Nächstes gilt es, einen Bruch zu finden, der der Schwierigkeitsstufe entsprechend lang genug ist. Dazu sucht die Funktion, angefangen bei einem Wert von 2, nach einem `x` welches für eine zufriedenstellende Länge des Bruches sorgt. Schießt die Länge über das Ziel hinaus, wird ein neuer Bruch gesucht.

`bruch(Stufe)` gibt die Variablen `a, b, p, q` zurück, mit denen ein Bruch `a/b = p/q` der Schwierigkeitsstufe gebildet werden kann. Durch `aufgaben(Stufe, Anzahl)` können mehrere Aufgaben der gleichen Stufe mit `print` ausgegeben werden. Schwierigkeitsstufen gehen von 0 - leicht über 1 - mittel bis 2 - schwer.

Beispiele

```
>>> import zahlenspiel
>>> zahlenspiel.aufgaben(1, 5)
75 / 105 = 5 / 7
104 / 39 = 8 / 3
100 / 10 = 10 / 1
112 / 24 = 14 / 3
108 / 45 = 12 / 5
>>> zahlenspiel.aufgaben(2, 1)
49 / 105 = 7 / 15
>>> zahlenspiel.aufgaben(0, 10)
60 / 10 = 6 / 1
70 / 10 = 7 / 1
10 / 15 = 2 / 3
70 / 10 = 7 / 1
10 / 40 = 1 / 4
15 / 12 = 5 / 4
10 / 50 = 1 / 5
10 / 30 = 1 / 3
20 / 12 = 5 / 3
80 / 10 = 8 / 1
```