Aufgabe 3: Voll daneben

Team-ID: 00133

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

29. Oktober 2018

Inhalt

Lösungsidee

Ziel ist, möglichst gute Zahlen für Al zu suchen. Alle Kombinationen durchzugehen würde viel zu lange dauern, deshalb wird nach Konzentrationen von Glückszahlen – also Bereichen, wo besonders viele Glückszahlen sind - gesucht, und zwar mit einem speziellen System. Die Liste der Glückszahlen dient dabei als Basis. Die Liste der Glückszahlen wird so vereinfacht, dass am Ende 10 Zahlen übrigbleiben, die möglichst an den Stellen liegen, wo besonders viele Glückszahlen liegen, um durch den geringen Abstand ein bestmögliches Ergebnis zu erzielen. Die Liste der Glückszahlen wird schrittweise vereinfacht: Immer die Zahlen mit dem geringsten Abstand werden zu einer neuen (dem arithmetischen Mittel der beiden Zahlen) zusammengeführt. Gibt es mehrere Zahlenpaare mit demselben Abstand, geschieht dies mit allen. Alle Zahlen haben zu Anfang die Gewichtung 1 erhalten. Nach der Zusammenführung eines solchen Zahlenpaares erhält die neue Zahl die Summe der Gewichtungen. Bei der Berechnung einer neuen Zahl wird eine der beiden Zahlen immer zuerst mit ihrer Gewichtung multipliziert, um das Ergebnis auch gemäß der Gewichtung zu beeinflussen. Eine neue Zahl errechnet sich also aus $\frac{n_a \cdot w_a + n_b \cdot w_b}{w_a + w_b}$, wobei n_a und n_b das Zahlenpaar und w die Gewichtung der Zahlen darstellt. Natürlich funktioniert dies auch für mehr als 2 Zahlen, wenn sie den gleichen Abstand haben, also zum Beispiel $[2, 4, 6, 8]$. Diese Liste von aufeinanderfolgenden Zahlen mit dem gleichen Abstand wird im Folgenden als Gruppe bezeichnet. Nachdem nur noch 10 Zahlen übrig sind, werden alle Zahlen noch gerundet (dies steht zwar nicht explizit in der Aufgabe, aber es macht auch keinen großen Unterschied, da eine von Als zahlen, wenn sie zwischen zwei Glückszahlen steht, den Abstand zur einen nur verkleinert und den zur anderen vergrößert).

Zusätzlich wird darauf geachtet, dass nicht mehr Zahlen zusammengeführt werden als nötig, um am Ende 10 Zahlen zu erreichen. Sobald in einem Vereinfachungsschritt die übrigbleibenden Zahlen kleiner als 10 wären, wird noch eine andere Technik angewandt. Statt dass jede Gruppe zu einer einzelnen Zahl vereinfacht wird, wird die Anzahl der übrigen Zahlen auf jede der Gruppe verteilt, je nachdem, welche Differenz die höchste und die niedrigste Zahl haben. Die Anteile werden natürlich gerundet, und deshalb kann es auftreten, dass doch zu viel oder zu wenig Zahlen verteilt würden (wenn z.B. besonders häufig aufgerundet wird). Gibt es zu viele, wird die Differenz von der größten Zahl abgezogen, gibt es zu wenig wird die Differenz zur kleinsten Zahl addiert. Das ist zwar nicht ganz genau, aber da die Differenz oft nur 1 beträgt, ist es nicht so wichtig.

Umsetzung

Das Programm ist in Python 3 geschrieben. Zusätzlich kommt zum Verteilen der Zahlen auf die Gruppen das Drittanbietermodul `numpy` zum Einsatz. Aufgrund der Größe ist es hier nicht enthalten, aber ich gehe davon aus, dass es bei den meisten Python-Installationen installiert sein sollte. Das Modul `math` aus der Standardbibliothek wird zum Runden verwendet.

Hauptsächlich gibt es das Modul `Aufgabe3/al_numbers.py`, welches die Funktion `get_al_numbers` enthält. Diese Funktion bekommt alle Glückszahlen der Teilnehmer. Es besteht natürlich theoretisch die Möglichkeit, dass es weniger oder genau 10 Glückszahlen gibt. In diesem Fall wird die Liste direkt zurückgegeben, und es werden (damit die Liste wirklich 10 zahlen enthält) so viele Nullen angefügt bis es 10 Zahlen gibt. Der Profit entspricht in so einem Fall der Anzahl der Teilnehmer (also der Anzahl der Glückszahlen) multipliziert mit 25.
Andernfalls werden die Glückszahlen aufsteigend sortiert und es wird eine Liste erstellt, die als Elemente Tupel enthält. Das erste Element eines Tupels enthält die Zahl, das zweite enthält die

Gewichtung, die am Anfang 1 beträgt. Zuerst wird nach allen Gruppen gesucht. Ein `dict` ordnet jedem Gruppenabstand – also dem Abstand zwischen den Zahlen in einer Gruppe – eine Liste zu, die für jede Gruppe mit diesem Abstand ein `slice`-Objekt enthält. Das `slice`-Objekt speichert als `start` den ersten Index und als `stop` den letzten Index + 1. Es werden nur Gruppen aufgenommen, die größer als 1 Element sind. Um das `dict` zu füllen, wird durch alle Indexe der Liste mit Nummern iteriert. Nachdem die zugehörige Nummer abgerufen wurde, wird die Differenz zur vorigen Nummer berechnet. Die vorige Nummer existiert bei dem ersten Element der Liste noch nicht, und deshalb wird mit der zweiten begonnen, während die erste Nummer am Anfang logischerweise die vorige Nummer darstellt. Es wird immer die Differenz der aktuellen Gruppe gespeichert. Weicht die neue Differenz von dieser ab, heißt das, dass hier eine neue Gruppe beginnt und die alte Gruppe – sollte sie mehr als 1 Element umfassen – gespeichert wird. Nach der Schleife zum Iterieren wird die letzte Gruppe, die ja noch nicht gespeichert wurde, auch noch hinzugefügt.

Mit den `slice`-Objekten mit dem kleinsten Gruppenabstand wird jetzt weitergearbeitet. Sollte die Länge der Liste nach Ersetzten der Gruppen durch eine einzelne Zahl nicht unter 10 liegen, wird jede Gruppe durch den Durchschnitt ersetzt (siehe „Lösungsidee“). Beträgt die Länge nun genau 10, werden die Zahlen zurückgegeben. Es wird mit der letzten Gruppe begonnen, sodass die Indexe der vorderen Gruppen nicht verändert werden. Beim Beginnen mit der ersten Gruppe verschieben sich alle folgenden Zahlen.
Ist die Länge der Zahlenliste kleiner als 10, wird zuerst die Anzahl an Zahlen berechnet, die nach Entfernen der Gruppen hinzukommen muss, um auf 10 Zahlen zu kommen. Das hier berechnete Ergebnis wird 10 Zahlen enthalten und wird deshalb danach auch zurückgegeben.
Für jede Gruppe wird die Anzahl an Zahlen berechnet, durch die sie ersetzt wird, und zwar gemäß ihrem Anteil an der gesamten Länge aller Gruppen. Die Länge einer Gruppe ist dabei durch die Differenz des Start- und End-Indexes definiert. Dies ist ausreichend, weil alle Gruppen sowieso denselben Abstand besitzen und deshalb der Anteil an der Länge automatisch dem Anteil an der Anzahl von Zahlen in der Gruppe entspricht. Die Anzahl an Zahlen für jede Gruppe wird gerundet, und deshalb kann es zu Differenzen zwischen der Summe aller Anzahlen für die Gruppen und der vorher ausgerechneten gesamten Anzahl kommen. Aus diesem Grund wird die Abweichung zu bestimmten Elementen addiert (siehe „Lösungsidee“). Jede Gruppe wird nun durch eine Liste statt einer Zahl ersetzt, wobei die Liste eine gleichmäßige Verteilung von Zahlen auf dem Bereich der Gruppe ist. Um die eben berechnete Anzahl der Zahlen zu verteilen, kommt die Funktion `numpy.arange` zum Einsatz. Diese verhält sich im Prinzip wie `range`, nur kann sie auch mit `float`s umgehen. Für `arange` wird ein `step` berechnet, und zwar der abgerundete Quotient aus der Differenz zwischen der kleinsten und der größten Zahl der Gruppe und der zustehenden Anzahl an neuen Zahlen. An jeder `step`-ten Stelle gibt es eine neue Zahl. Um die neuen Zahlen noch weiter zu verbessern, beginnen diese nicht mit der kleinsten Zahl der Gruppe bzw. hören mit der größten auf. Jeweils die Hälfte von `step` wird am Anfang addiert und am Ende subtrahiert, sodass sich die neuen Zahlen noch mehr auf dem Gebiet der Gruppe befinden. Am Rand sind neue Zahlen nämlich nicht so relevant, wie ich durch ausprobieren herausgefunden habe. Dass man die Range um eine `step`-Länge verkürzen kann, geht deshalb, weil es überhaupt keine Zahl auf der größten Zahl der Gruppe gegeben hätte. `arange` würde vorher abbrechen, weil der gegebene `stop`-Parameter nicht mehr vorkommen darf.

Beispiele

Beispiel zur Vereinfachung der Nummern

Das folgende Beispiel zeigt, wie eine Liste von Nummern vereinfacht wird. Dabei kommt es allerdings nicht vor, dass bereits vereinfachte Zahlen nochmals vereinfacht werden, weshalb Gewichtungen vernachlässigt werden.

Die Zahlen sollen hier auf insgesamt 6 Zahlen vereinfacht werden. Die kleinsten Abstände sind schwarz, die anderen rot eingefärbt.

Glückszahlen	0	1	1	1	2	3	5	3	8	1	9	3	12	2	14	2	16	3	19
1. Vereinfachung	1					4	5	3.5	8.5			3.5	12	2	14	2	16	3	19
2. Vereinfachung	1					4	5	3.5	8.5			6.5	13			15		4	19
Rundung	1						5		9				13			15			19

Zu erkennen ist besonders die Vereinfachung von 3 Zahlen auf 2 im letzten Schritt. Würden die Zahlen 12, 14 und 16 zu 14 zusammengefasst, so gäbe es nur 5 Zahlen. Stattdessen muss diese Gruppe aber zu 2 Zahlen vereinfacht werden. Der `step` beträgt hier $(16 - 12) \div 2 = 2$, da hier der Bereich von 14 bis 16 durch zwei Zahlen ersetzt wird. Die Hälfte ist 1, deshalb ist die erste Zahl $12 + 1 = 13$ und die zweite $13 + 2 = 15$.

Beispieldaten der BwInf-Webseite

Um die Beispieldaten der BwInf-Seite zu berechnen, wird das Skript `Aufgabe3/aufgabe3.py` ausgeführt:

```
Beispiel1
Als Zahlen: [54, 153, 252, 351, 450, 549, 648, 747, 846, 945]
Als Profit: 25$

Beispiel2
Als Zahlen: [55, 127, 194, 312, 397, 544, 666, 773, 869, 942]
Als Profit: 362$

Beispiel3
Als Zahlen: [66, 159, 253, 388, 485, 581, 678, 775, 871, 980]
Als Profit: 31$
```

Hier zeigt sich auch, dass dieses Verfahren zwar sehr gute Ergebnisse liefert, aber nicht die besten. Schon durch Veränderung der 253 im dritten Beispiel auf eine 260 wird ein Profit von 38$ statt 31$ erzielt. Diese Veränderungen vorzunehmen ist aber nicht allzu kompliziert, da es hier z.B. einen auffallend großen Bereich nach oben (zur 388) gibt und viele auf die 260 gesetzt haben, die 253 aber bei jedem einen Verlust von 7$ generiert. Und die Steigung auf 38$ macht auch etwa „nur“ 20% aus. Beim 2. Beispiel habe ich keine Verbesserungsmöglichkeit gefunden (was nicht heißt, dass es nicht doch eine gibt), aber hier geht es auch um einen größeren Betrag, bei dem 20% schon mehr ausmachen würden.

Quellcode

Bedeutung der Skripte

```
python3 Aufgabe3/aufgabe3.py
```

Berechnet Als Nummern für die Beispieldaten nach dem beschriebenen Verfahren und gibt diese mit dem Profit aus.

```
python3 Aufgabe3/aufgabe3-cmd.py <path>
```

Bekommt einen Pfad `path` zu einer Datei mit Glückszahlen. Gibt Als Zahlen und den Profit aus.

```
Aufgabe3/al_numbers.py
```

Stellt Funktionen bereit, um Als Zahlen zu berechnen (siehe folgenden Abschnitt).

Aufgabe3/al_numbers.py

`get_al_numbers(lucky_numbers)`

Bekommt eine Liste von Glückszahlen `lucky_numbers`. Berechnet daraus gute Zahlen für Al nach dem oben beschriebenen Verfahren. Hier ist die Funktion, zusammen mit erläuternden Kommentaren:

```python
def get_al_numbers(lucky_numbers):
  if len(lucky_numbers) < 11:
      # es gibt schon weniger als 11 Glückszahlen, Al erhält also einen Gewinn von
      # 25*(Anzahl der Teilnehmer).Es werden so viele Nullen angefügt, bis es
      # trotzdem 10 Zahlen gibt (was eigentlich nicht nötig wäre).
      return list(lucky_numbers) + [0]*(10-len(lucky_numbers))
  al_numbers = [(i, 1) for i in sorted(lucky_numbers)]  # wird im Folgenden
                                                        vereinfacht
  while True:
      # 1. Schritt: NACH GRUPPEN SUCHEN
      groups = {}  # ordnet jedem Gruppenabstand eine Liste mit slice-Objekten (für
                     eine Gruppe) zu
      group_diff = None  # es gibt zunächst noch keinen Gruppenabstand
      group_start = 0
      group_end = 0
      # es wird mit dem zweiten Element begonnen, das erste Element zählt also als
      # das vorige
      last_num = al_numbers[0][0]
      for i in range(1, len(al_numbers)):
          num = al_numbers[i][0]  # die Zahl für diesen Index (und nicht die
                                    Gewichtung)
          diff = num-last_num  # Differenz zwischen der vorigen und dieser Zahl
          if diff != group_diff:
              # die alte Gruppe ist zu Ende
              if group_end-group_start > 1:  # nur Gruppen länger als 1
                  # alte Gruppe hinzufügen
                  if group_diff not in groups:
                      # Gruppenabstand existiert noch nicht, eine neue Liste wird
                      # angefangen
                      groups[group_diff] = [slice(group_start, group_end)]
                  else:
                      # Gruppenabstand existiert bereits, also wird das slice-Objekt
                      # angefügt
                      groups[group_diff].append(slice(group_start, group_end))
              group_diff = diff  # der neue Gruppenabstand
              # die Gruppe beginnt mit der vorigen Nummer (also i-1)
              group_start = i-1
          # das Gruppenende ist nun die Zahl, die gerade dran ist
          group_end = i+1  # der Index + 1 wird für das (derzeitige) Gruppenende
                             gesetzt
          last_num = num  # die vorige Zahl für den nächsten Durchlauf
```

```
    # die letzte Gruppe ist noch nicht angefügt worden (es gab ja keinen
    # abweichenden Gruppenabstand)
    # diese wird jetzt angefügt (sofern sie mehr als 1 Element umfasst)
    if group_end - group_start > 1:
        if group_diff not in groups:
            groups[group_diff] = [slice(group_start, group_end)]
        else:
            groups[group_diff].append(slice(group_start, group_end))

    min_groups = groups[min(groups)]  # die Liste von Gruppen mit dem kleinsten
                                      Abstand
    group_len = sum(g.stop-g.start for g in min_groups)  # Größe der Gruppen
    # 2. Schritt: PRÜFEN, OB ES NACH DEM ZUSAMMENFÜHREN IMMER NOCH MEHR ALS 10
    # ZAHLEN WÄREN
    if len(al_numbers) - group_len + len(min_groups) < 10:
        # ES WÜRDE ZU WENIG ZAHLEN GEBEN, GRUPPEN WERDEN DESHALB DURCH MEHRERE
        # ZAHLEN ERSETZT
        max_len = 10 - (len(al_numbers) - group_len)  # Anzahl an Zahlen, die
                                                      insgesamt nach dem Entfernen der
                                                      Gruppen hinzukommen müssen
        # Anzahl an neuen Zahlen für jede Gruppe berechnen (gemäß dem Anteil an der
        # Gesamtgröße)
        lengths = [round(((g.stop-g.start)/group_len)*max_len)
                                                            for g in min_groups[::-1]]
        # da die Anzahlen gerundet sind, könnte es eine Abweichung geben
        diff = sum(lengths) - max_len
        if diff > 0:
            # die Anzahlen sind insgesamt zu klein,
            # deshalb erhält die kleinste Länge die Differenz dazu
            lengths[lengths.index(min(lengths))] += diff
        elif diff < 0:
            # die Anzahlen sind zu groß,
            # die Differenz wird von der größten Länge abgezogen
            # (immer noch '+', weil diff negativ ist)
            lengths[lengths.index(max(lengths))] += diff

        # Gruppen von hinten durchgehen und durch ihre Länge ersetztn
        for group_slice, length in zip(min_groups[::-1], lengths):
            if length == 0:
                # die Anzahl an neuen Zahlen ist 0,
                # deshalb wird die Gruppe durch eine leere Liste ersetzt
                replace = []  # die neuen Zahlen
            else:
                start_num = al_numbers[group_slice.start][0]  # die erste Zahl der
                                                                Gruppe
                end_num = al_numbers[group_slice.stop-1][0]  # die letzte Zahl
                num_diff = end_num-start_num  # Differenz der größten/kleinsten
                                              Zahl
                # die neuen Zahlen berechnen
                step = num_diff / length
                replace_nums = list(
                    np.arange(math.floor(start_num + step / 2),
                              math.floor(end_num - step / 2 + 1), step))
                # Gewichtungen anfügen
                replace = [(i, 1) for i in replace_nums]
            # Gruppe ersetzten
            al_numbers[group_slice.start:group_slice.stop] = replace
        # al_numbers enthält nun 10 Zahlen, also werden sie (gerundet und in int
        # umgewandelt, da die Rückgabewerte von 'round' trotzdem immer noch floats
```

```
            # sind) zurückgegeben
            return [int(round(i[0])) for i in al_numbers]
        # die Gruppen können zusammengefasst werden, ohne dass al_numbers zu klein wird
        for group_slice in min_groups[::-1]:
            group = al_numbers[group_slice.start:group_slice.stop]
            group_weight_sum = sum(i[1] for i in group)  # Summe der Gewichtungen
            group_num_sum = sum(i[0]*i[1] for i in group)  # Wert aller Zahlen
                                                             multipliziert mit der
                                                             Gewichtung
            new_num = group_num_sum / group_weight_sum  # Mittelwert
            al_numbers[group_slice.start:group_slice.stop] = [(new_num,
group_weight_sum)]
        if len(al_numbers) == 10:
            # al_numbers enthält 10 Zahlen, die zurückgegeben werden
            return [int(round(i[0])) for i in al_numbers]
        elif len(al_numbers) < 10:
            # dieser Teil sollte theoretisch nie erreicht werden. Zur Sicherheit wird
            # es trotzdem abgefangen, wenn al_numbers plötzlich weniger als 10 Zahlen
            # enthält
            raise ValueError("An unexpected error: Too many numbers have been"
                             "removed")
```

`get_profit(lucky_numbers, al_numbers)`

Berechnet den Profit mit den Glückszahlen `lucky_numbers` und Als Zahlen `al_numbers`. Das Ergebnis der Funktion `get_winnings` wird von der Länge von `lucky_numbers` multipliziert mit 25 (dem Einsatz) subtrahiert.

`get_winnings(al_numbers, lucky_numbers)`

Berechnet die Summe der Gewinne der Teilnehmer (den Einsatz nicht mitberücksichtigt). Zu jeder Glückszahl wird mit `get_nearest_number` die nächste aus `al_numbers` ausgewählt und die Differenz ist der Gewinn für die jeweilige Glückszahl. Die Summe wird zurückgegeben.

`get_nearest_number(iterable, value)`

Gibt die Zahl aus `iterable` mit dem kleinsten Abstand zu `value` zurück. Mit der `min`-Funktion werden alle Elemente in `iterable` verglichen, und der Schlüssel für ein Element ist die Differenz zu `value`.